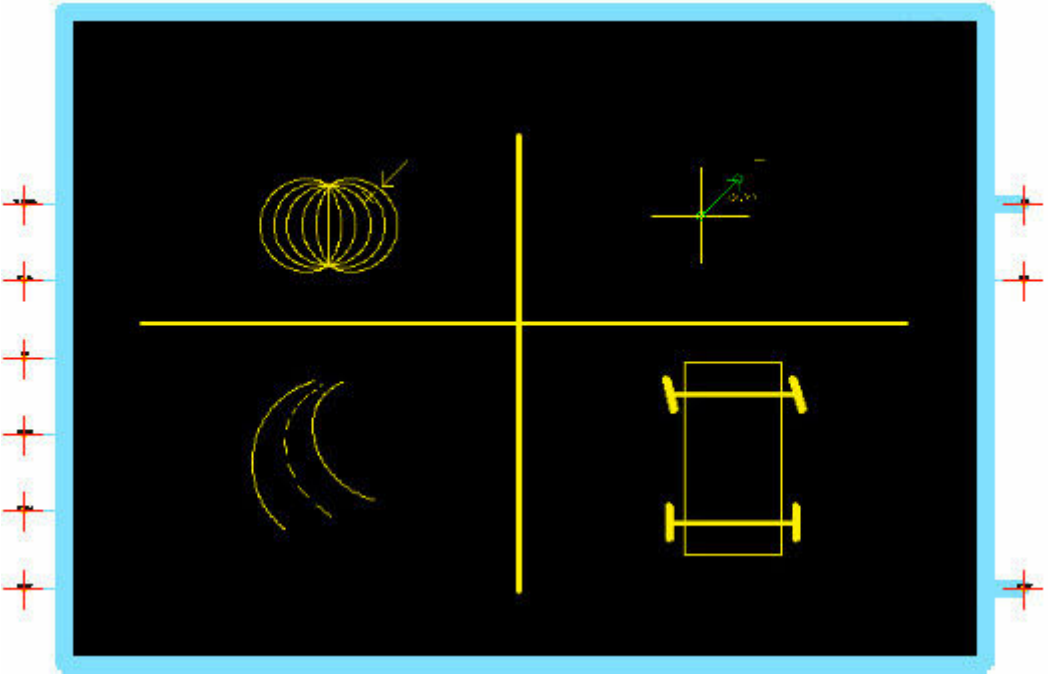
ENGINEERING
TOMORROW

User Manual

PLUS+1® GUIDE Software
Autonomous Control Library Function Blocks



Revision history

Table of revisions

Date	Changed	Rev
June 2025	Updated document for ACL 4.0. Added boundary function blocks and more background information.	0109
August 2024	Updated document for ACL 3.2.	0108
February 2024	Updated document for ACL 3.1.	0107
October 2023	Added path blocks and licensing information.	0106
December 2022	Added Line Follower and Line Yaw Estimate function blocks, and updated function blocks.	0105
May 2022	Added and updated function blocks.	0104
December 2021	Two new function blocks, Post Detection and Projected path.	0103
October 2021	Added new function blocks to the library.	0102
January 2020	First edition	0101

Contents

Contents

Contents

Contents

Contents

Library Introduction

The autonomous control function block library offers a quick and easy way to develop PLUS+1® GUIDE applications that provide the foundation for autonomous machine control systems.

Using the function blocks in this library, developers can create applications that allow machines to navigate environments without an operator.

Autonomous machine applications are comprised of several core sub-systems built with the autonomous control function blocks:

- Perception: These blocks use radar systems, laser-based (LiDAR) systems, and ultrasonic sensors to help the machine see its surrounding environment. The LiDAR scanner detects reflectors, posts, walls, and other obstacles.
- Positioning: These function blocks provide information about a machine's location. The machine either needs GPS to know where it is, or a specific point must be entered for the machine to know where it starts. Machines use 2-D orientation using a X and Y plane, rather than a full 3-D with Z (height) orientation. The machines must be on one level.
- Navigation: These function blocks help autonomous machines reach their destination. Navigation blocks can only be utilized after the machine is aware of its position.
- Utility: These generic blocks are not specific to the perception, positioning, or navigation categories, but they are useful for implementing autonomous functions.

Acronyms

Acronyms used in the library user manual are described.

Acronym	Meaning
ACL	Autonomous Control Library
CAN	Controller Area Network
ECU	Electronic Control Unit
ENU	East/North/Up
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
NED	North/East/Down
UTM	Universal Transverse Mercator

Autonomy Glossary

The following list describes glossary terms used in the Autonomous Control Library (ACL), *Plus+1 Compliant Ouster Block User Manual*, and *Plus+1 Compliant LeiShen Block User Manual*. Most terms are explained in *Background on Autonomous Machines* on page 13.

Azimuth - Horizontal angle or an angle in the XY plane from a reference direction to a point of interest. For example, a LiDAR sensor's field-of-view is limited to make a specific azimuth.

Azimuth Window - The contiguous horizontal space within a 360 degree scanning area where LiDAR sensors collect data about their environment.

Bearing - Absolute or relative angle from the machine to the target. This includes the polar coordinates along with a distance.

Band - In the Universal Transverse Mercator (UTM) system of making a grid of the Earth to get a location, a band is every 8 degrees in the latitude direction, using the labels C to X.

Channel - In ACL, a channel refers to one point cloud ring row around a LiDAR sensor, so a 64 channel LiDAR means that there are 64 rows of stacked rings.

Course Over Ground (COG) - The machine's actual direction of travel. This is equivalent to yaw for most land-based machines without slipping or sliding.

Library Introduction

Curvature - In ACL, this refers to how sharp the machine should turn. Curvature is defined as the inverse of the turning radius. Due to East-North-Up (ENU), positive values are left curves and negative values are right curves when driving forward.

Data locker - In ACL, this is a shared memory infrastructure that allows other function blocks to store and access data simultaneously without having large data structures in PLUS+1® GUIDE. See *Data_Lockers Helper Block* on page 77.

East-North-Up (ENU) - A reference frame where coordinates start at 0° facing East, move in a positive direction going North, and move in a negative direction going South. East (X), North (Y), and Up (Z) are each relative to a local origin. ACL uses ENU and not the other common reference frame, North-East-Down (NED).

Easting - Generally used in UTM to refer to the X coordinate being East of the UTM zone's origin. In ACL, Easting aligns with latitude and is written as X in many function blocks.

Elevation - In ACL, this is usually the vertical angle from a LiDAR sensor to a point of interest.

Field-of-View (FOV) - This refers to the horizontal and/or vertical areas where a LiDAR sensor collects data or visualizes its environment. For example, a LiDAR could have a 90 degree vertical field-of-view.

Global or **World coordinate** - A unique position on Earth. For example, a machine's position could be at 45° N latitude and 135° W longitude.

Heading - Direction the front of the machine is facing. Heading is 0° when pointing East and 90° when pointing North.

LiDAR (Light Detection and Ranging) - A sensor that provides data about the machine's surrounding environment. It measures the distance from itself to objects with lasers.

Local coordinate - A position relative to a local landmark. For example, a machine's position relative to a building or the machine's starting position.

Machine's origin - The coordinate origin on the machine, which moves with the machine. It is recommended to have this origin at the machine's steering point.

Machine - In ACL, a machine refers to any type of vehicle or mobile machinery.

Northing - Generally used in UTM to refer to the Y coordinate being North of the UTM band's origin. In ACL, Northing aligns with longitude and is written as Y in many function blocks.

Orientation - Angles of features relative to the sensor, or sensor relative to machines. This references the angles and not the position.

Origin - The base point of a rigid reference frame with the coordinates (0,0,0). There are many reference frames which all have origins and axes. In ACL, there could be a global origin, local origin, machine origin, and sensor origins for each hardware device.

Pose - A term referring to both position and orientation. This is commonly used in the Robot Operating System (ROS) simulation program.

Pitch - The angle of rotation around the y-axis. In ACL, a machine pitching down and forward has a positive angle, and up and backward has a negative angle.

Point - In ACL, LiDAR sensor points refer to a coordinate point with X, Y, and Z data. These include data about reflectivity, ambiance, and spherical coordinates.

Point cloud - All the LiDAR sensor points from the LiDAR scan are known collectively as a point cloud and allow a machine to perceive its surrounding environment. Each point's position has its own set of Cartesian coordinates, which can be 2D or 3D (X, Y, Z) depending on the LiDAR.

Range - In ACL, this refers to the maximum distance a LiDAR sensor sees in its environment, such as 9000 mm. It also indicates a span between numbers, such as 50-100.

Relative Position - The position of something relative to something else. In ACL, this usually refers to a machine relative to something in the environment, such as the local origin.

Resolution - A vertical resolution refers to the number of laser beams or channels the LiDAR sensor uses to scan the environment, impacting the density and detail of the point cloud. A horizontal resolution

Library Introduction

refers to the number of times a laser pulses in one full rotation. Higher resolution means more points per unit area, allowing for finer distinctions between objects and surfaces.

Right-hand rule - A convention which defines the orientation of axes in three-dimensional space. For example, the right hand extends in front of a person with their thumb pointing to the sky, indicating Z. The fingers pointing straight ahead indicate the direction something travels, and curling the fingers into a fist indicates a positive rotation direction.

Ring - In ACL, rings refer to a circular point cloud row around a LiDAR sensor. One ring row is known as a channel, so a 64 channel LiDAR means that there are 64 rows of stacked rings.

Roll - The angle of rotation around the x-axis. In ACL, a machine rolling down to the right has a positive angle, and down to the left has a negative angle.

Steering Point - The location or point in the machine with no lateral motion. It is recommended to have this as the machine's origin. In ACL, the machine's origin is sometimes simultaneously referred to as the steering point.

Universal Transverse Mercator (UTM) - A standard way of turning the round Earth into a rectangular grid which consists of zones and bands.

Waypoint - An intermediate point on a path or route. In ACL, a waypoint usually contains information about a machine's relative XY position, yaw, velocity and ancillary data.

Yaw - The angle of rotation around the z-axis. In ACL, a machine rotating counterclockwise is positive and clockwise is negative. Yaw can also refer to the direction the machine faces (heading).

Yaw rate - The rate in degrees per second of the machine turning around the z-axis. For example, a machine turning from 45 to 50 degrees in one second has a yaw rate of 5 degrees per second.

Zone - ACL includes two different and unrelated types of zones. Zone boxes in function blocks like **Obstacle_Detect** show a small area, or zone, around the machine. In the Universal Transverse Mercator (UTM) system of making a grid of the Earth to get a location, a UTM zone is every 6 degrees in the longitude direction, using the labels 1 to 60.

Licenses Required

The Autonomous Control Library (ACL) function blocks require a license after the 3.0 version.

Licenses are divided into two categories, Essentials and Advanced. Advanced includes more complex function blocks that have the ability to record paths and have the machine follow them. The levels of autonomy, which refer to how much humans are involved in driving the machine, do not affect the categories.

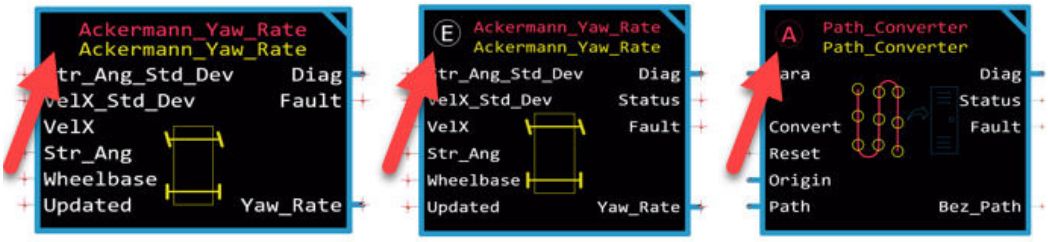
Software License Information

Question	Essentials	Advanced
How much does this license cost?	Free with Plus+1 GUIDE Professional subscription.	Reach out to the account manager.
How does payment work?	These blocks will always be free with the purchase of a PLUS+1 GUIDE Pro subscription.	This is a one-time payment which grants access to all Advanced blocks keyed to a specific piece of hardware, even new blocks in future releases. Buy the specific hardware with this license on it, in addition to the PLUS +1 GUIDE Pro subscription. Each hardware requires its own license.

Library Introduction

Software License Information (continued)

Question	Essentials	Advanced
Which hardware does the license cover?	All past and future XM100 and DM1000 products.	New XM100 hardware. If using old hardware, contact the account manager to discuss options. Additional hardware products will be added in the future.
What are the benefits of each category?	• Free. • Includes blocks from ACL version 1.0 - 3.0.	• More complex blocks. • Easier to connect in applications. • Ability to record paths and follow them.



Blocks include a logo in the corner marking their category. The images show the Essential function blocks which either have no logo or an E, depending on their version. The Advanced function blocks have an A.

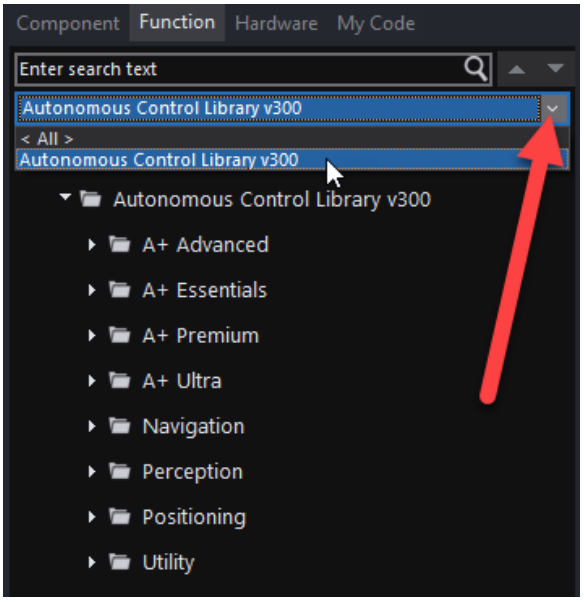
Additionally, certain function blocks fall into each category. Advanced includes all the blocks in the Essentials list.

ACL Blocks in Each Category

Essential Blocks	Advanced Blocks
• **Ackermann_Yaw_Rate** • **Angle_To_Curv** • **Curv_To_Angle** • **Data_Lockers** • **Edge_Detect** • **Extract_Ring** • **LiDAR_Mask** • **Line_Follower** • **Line_Yaw_Estimate** • **Obstacle_Avoidance** • **Obstacle_Detect** • **Origin** • **Path_Follower** • **Position_Filter** • **Post_Detect** • **Projected_Path** • **Reflector_Detect** • **Relative_Pos** • **Transform_3D** • **Transform_GNSS** • **Two_Point_Planner** • **UTM_Conv** • **UTM_Conv_Zone** • **Wall_Detect** • **Yaw_Estimate**	Includes all Essential blocks, as well as these: • **Boundary_Converter** • **Boundary_Extract** • **Boundary_Loader** • **Boundary_Recorder** • **Geofence_Check** • **LiDAR_Filter** • **Obstacle_Detect_Area** • **Path_Converter** • **Path_Extract** • **Path_Follower_Adv** • **Path_Loader** • **Path_Recorder** • **Planar_Surface_Segmentation** • **Projected_Path_Area**

Library Introduction

To see which function blocks belong in different categories in PLUS+1® GUIDE, click on the drop-down within the **Function** tab and select a version of the Autonomous Control Library.



The image above displays the categories and filtering options associated with version 3.0 or later. Future releases will include more categories, such as Premium and Ultra.

Versions Required

The Autonomous Control Library version 4.0 function blocks successfully compile in PLUS+1® GUIDE version 2024.2 and later, Service Tool version 12.2 or later, and XM100 hardware version 3.21 or later. It is recommended to use Service Tool version 2024.1 or later when using the media file manager.

Documents to use

Below are recommended documents to use in conjunction with the Autonomous Control Library (ACL) function blocks.

Find comprehensive technical literature online at *https://www.danfoss.com*.

Documents Needed for Autonomy

Title	Type	Document Number
LeiShen LiDAR Block	User Manual	AQ448276254591
Ouster LiDAR Function Block	User Manual	AQ404281942428
PLUS+1® Function Block Library - Autonomous Control Function Blocks	User Manual	AQ295075513101
PLUS+1® GUIDE Software	User Manual	AQ152886483724
PLUS+1® XM100 Autonomous Controller	Data Sheet	AI379058006235
PLUS+1® XM100 Autonomous Controller	Technical Information	BC394784770000
PLUS+1® XM100 Reliability Data MTTF	Safety Manual	BH409064980476
XM100 HW Description - Application Interface*	API Specification	70493872v322

*Note that the most accurate API Specification is found within PLUS+1® GUIDE under **Project Manager** > **HWD** > **XM100** rather than the **Hardware** tab. LiDAR documentation only applies when using that hardware.

Background on Autonomous Machines

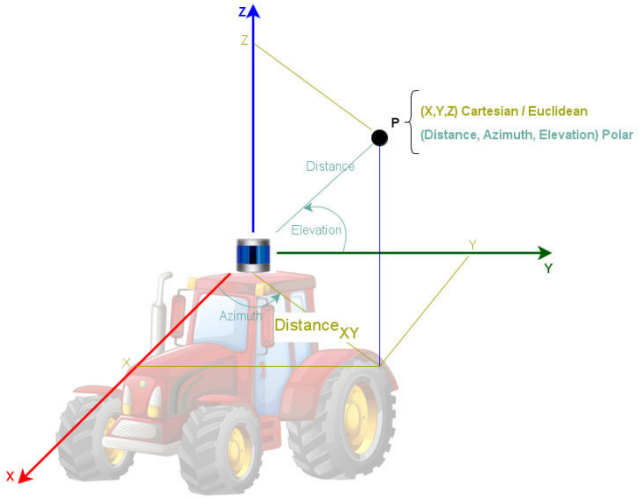
Perception

An autonomous machine requires sensors to detect and avoid obstacles during navigation.

Ultrasonic sensors can detect objects that are in close range to the autonomous machine. With their conical detection zone and an offering of a scalar distance value to the nearest object, these sensors are useful for emergency braking and in safety curtain scenarios.

Control Area Network, or CAN-based, sensors can easily integrate with the Autonomous Control Library. Radar systems are well suited to detect obstacles for autonomous machines due to their ability to operate in harsh conditions.

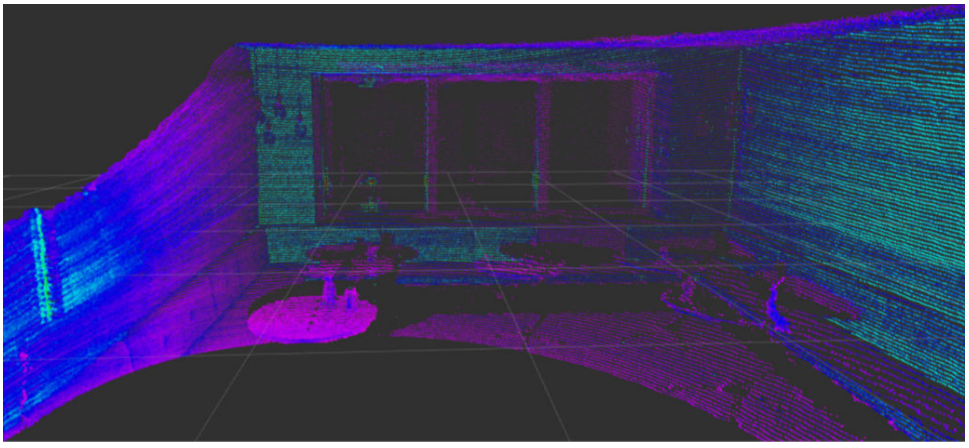
LiDAR (Light Detection and Ranging) systems also provide some obstacle data due to their more accurate distance readings. LiDAR sensor hardware uses lasers to scan the environment, picking up distance and reflective surfaces. Many LiDAR sensors capture information in 3D.



The image above shows a LiDAR sensor on top of a machine getting information about the surrounding environment in the X, Y, and Z plane. Azimuth refers to the horizontal angle, and elevation refers to the vertical angle.

Dots, referred to as points, appear in a circular ring around the LiDAR scanner that stack in rows. One ring row is known as a channel, so a 64 channel LiDAR means that there are 64 rows of stacked rings. However, the first row is written as row zero, so the 64-channel LiDAR has rows 0-63. Whether ring row 0 starts at the ground or the top depends on the LiDAR hardware.

LiDAR scanners come with different channel counts, and higher channel counts have more data points with a clearer picture of the environment. Most LiDAR's come in 16, 32, 64, or 128 channels.



Background on Autonomous Machines

The image above shows rows of horizontal rings with objects identified by the LiDAR scanner. The image below shows a photo of the room.

Ouster LiDAR scanners work well with ACL function blocks, and there is an **Ouster_LiDAR** compliance block to go with the scanner. See the *Plus+1 Compliant Ouster Block User Manual* for information.

Point Clouds

All the points from the LiDAR scan are known collectively as a point cloud.

A point cloud is a set of data points in space, and each point's position has its own set of Cartesian coordinates, which can be 2D or 3D (X, Y, Z) depending on the LiDAR. The Autonomous Control Library (ACL) blocks use data from the point clouds to perceive the environment so the machine can decide what actions to take.

ACL blocks utilize ordered point clouds, which means each point is accounted on the scan with its own coordinates. However, to speed up data processing time, ACL blocks can also use unordered point clouds. This is when some data points are skipped over so a smaller picture forms, such as removing a few rows of rings from what the LiDAR calculates. Ordered point clouds can convert to unordered point clouds, but unordered point clouds cannot convert to ordered point clouds.

The point cloud information from one function block automatically flows to a storage space known as a data locker, and then other function blocks can use that information. ACL blocks use the following signals to indicate types of point clouds and whether they are input or output signals:

Point Cloud Signal Names

Signal Name	Description
O_PtCld	The data locker ID of an ordered point cloud data.
O_PtCld_In	The data locker ID of input ordered point cloud data.
O_PtCld_Out	The data locker ID of output ordered point cloud data.
PtCld	The data locker ID of ordered or unordered point cloud data.
U_PtCld	The data locker ID of unordered point cloud data.

Positioning

The physical location of the machine must be determined so the machine can operate autonomously.

The machine determines its starting place from a Global Navigation Satellite System (GNSS) or from a point being programmed into the autonomy software. There are multiple origins and coordinate systems used, including global, local, machine, and sensor.

Data from GNSS, Inertial Measurement Units (IMU), steering angle sensors, and wheel speed sensors can produce a constant and reliable estimate of a machine's location when processed by a position filtering algorithm.

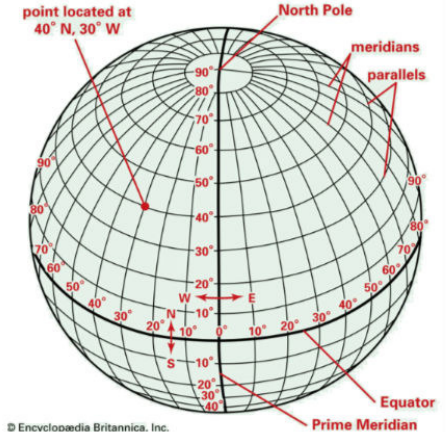
Background on Autonomous Machines

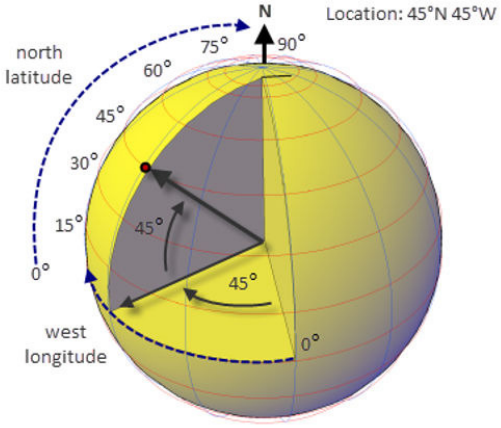
Ideally, align the various coordinate systems to the machine's coordinate system. It is recommended to calibrate the coordinate systems to align at zero degrees.

Global or World Coordinate System

The global coordinate system is a reference frame which tracks where the machine is located in the world. Machines need to know their location within the surrounding environment by using either global or local coordinate data.

A global coordinate system uses the Equator and Prime Meridian to specify the location of objects on Earth.





The left image from Encyclopedia Britannica, Inc.© shows the Earth with parallels of latitude and meridians of longitude, which are used in a global coordinate system. The right image from AutoCAD 2023 Help© displays angles with the center of the Earth as the origin. For example, an object at 45°N, 45°W means that it is at a 45° angle north of the Equator and a 45° angle west of the Prime Meridian.

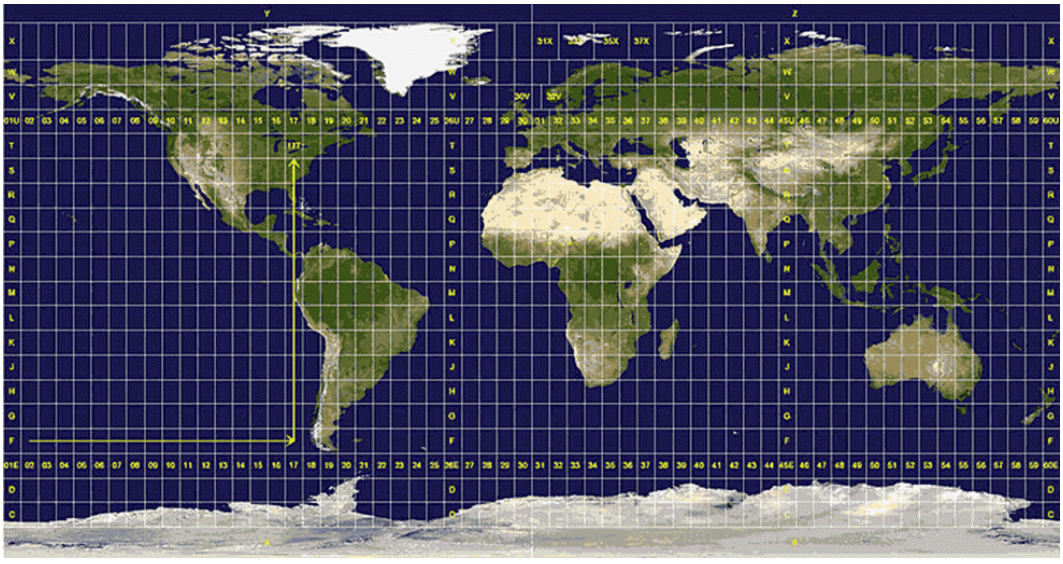
Latitude and longitude in ACL blocks consist of the following:

- Use as decimals rather than degrees, minutes, seconds. For example, 45° 30' 2'' would be written as 45.5005556
- East and North are positive. For example, 45°N, 25°E would be written as 45.0000000, 25.0000000.
- West and South are negative. For example, 45°S, 25°W would be written as -45.0000000, -25.0000000.

The global longitude and latitude coordinates must be converted into the Universal Transverse Mercator (UTM) coordinate system. The UTM takes the world, flattens it out, and imposes zones over the earth. This allows the machine to be tied to a distance measuring system.

The East and West UTM directions are divided into 60 zones, with each zone being 6 degrees. Further, each zone is divided into a band every 8 degrees in the latitude direction, using the labels C to X. The letters I and O are not used, to avoid their potential confusion with the numbers one (1) and zero (0). A, B, Y, and Z represent two polar regions.

Background on Autonomous Machines



The image above from Wikimedia Commons © shows UTM grid zones on a projected map of the world.

UTM Zone and Band ASCII Values				
67	C		78	N
68	D		80	P
69	E		81	Q
70	F		82	R
71	G		83	S
72	H		84	T
74	J		85	U
75	K		86	V
76	L		87	W
77	M		88	X

For example, if a machine is located at 45°N, 25°E, the UTM is Band T, Zone 35. The band and zone lines are now 0, so the specific number is positive and to the left or above the band or zone line. In the ACL blocks for this example, the UTM readings would output the band, zone, as well as 34269000 millimeters Easting and 4984896000 millimeters Northing. Easting refers to X, which is the distance from the zone origin to the machine. Northing refers to Y, which is the distance from the band origin to the machine.

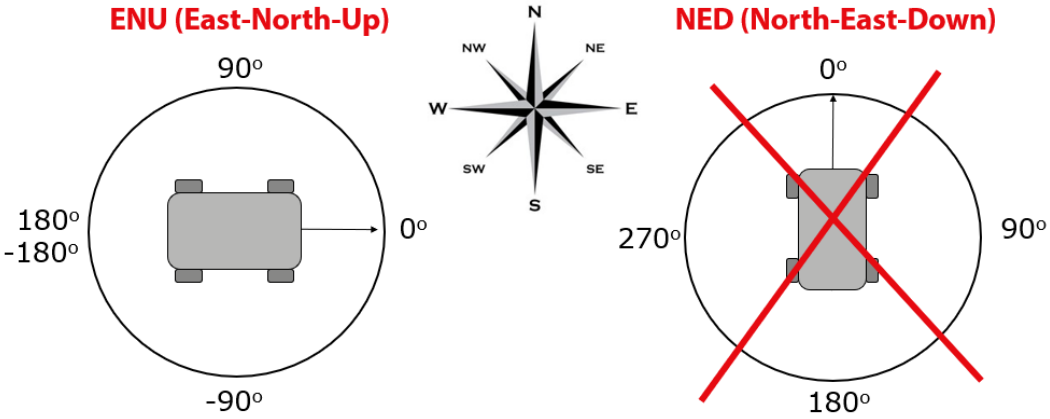
Global positioning requires Global Navigation Satellite System (GNSS). It is also recommended to use Inertial Measurement Unit (IMU) sensors and wheel odometry to get the best location results. IMU sensors consist of:

- Gyroscopes, which give angular velocity along the x, y, and z-axis.
- Accelerometers, which give linear acceleration data along the x, y, and z-axis.
- Magnetometers or compasses, which give the direction of magnetic north.

GNSS could give errors based on atmospheric delays, orbital position, and issues with the clocks. Services such as Real Time Kinematics (RTK) and Satellite Based Augmentation Systems (SBAS) can be used to correct the errors.

Danfoss programs their autonomy function blocks and hardware to work in an East-North-Up (ENU) coordinate system, which follows the right-hand rule. This means that coordinates start at 0° facing East, move in a positive direction going North, and move in a negative direction going South. Other coordinate systems such as North-East-Down (NED) start at 0° facing North and rotate the full 360° instead of the half circle. NED coordinates must be converted to ENU to work with Danfoss hardware and software.

Background on Autonomous Machines



The image above shows the East-North-Up and North-East-Down global coordinate systems. Danfoss uses ENU.

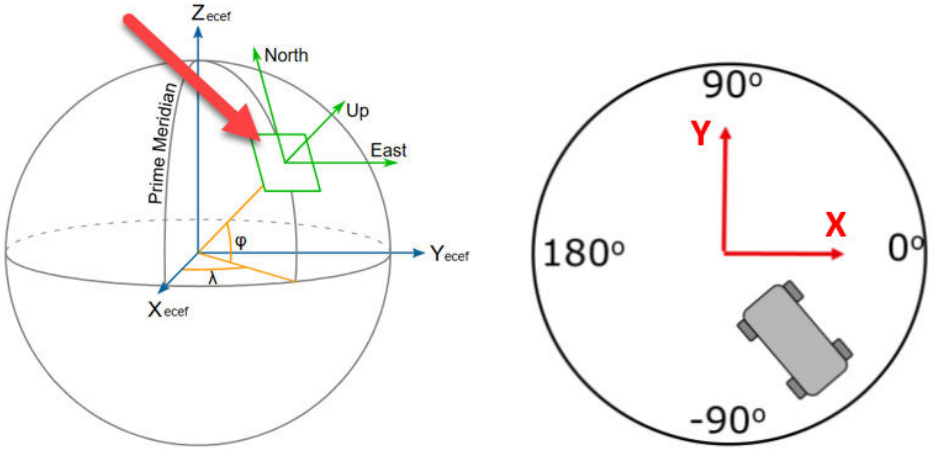
Turning South in the ENU system, the machine passes through -1°, -2°until reaching -180° when facing West. Turning North, the machine passes through positive degrees until facing West at 180°. If the machine kept turning, it would pass through -179° and keep going back in negative degrees until facing East again.

Currently, autonomy does not factor in altitude (Up) and refers to the global coordinate system as if it is a 2D world.

Local Coordinate System

A local reference frame refers to a coordinate system or frame of reference that is expected to function over a small region or restricted regional space within the global frame.

The local coordinate system is fixed on Earth and does not move with the machine, referred to as Earth Centered Earth Fixed (ECEF). It moves from large global coordinates to a smaller surface section of the Earth. Danfoss's Autonomous Control Library function blocks require local coordinates to match East and North like the global coordinates, except in special circumstances.



The left image from Wikipedia© shows the local coordinate system uses a section of the Earth's surface (green) in relation to the larger global coordinates. The right image displays the East-North-Up local reference frame where positive X is at 0°, positive Y is at 90°, and they are not in relation to the machine.

East (X), North (Y), and Up (Z) are each relative to a local origin. Z must always be up.

Background on Autonomous Machines

If working in outdoor environments, the local frame could be the UTM where the origin is chosen by the developer. For example, the origin could be a building corner, but it should be somewhere within the operating area of the machine to keep the numbers small.

In the Autonomous Control Library, the **Relative_Pos** function block is used for local coordinates based on GNSS.

Additionally, for indoor environments, the x-axis, y-axis, and origin can be chosen wherever is most convenient, as long as X is 0° and Y is 90°. GNSS is not needed indoors, so the local X and Y coordinates do not need to align with East at 0° and North at 90°. However, it is recommended to align them in case the machine goes outside.

Machine Coordinate System

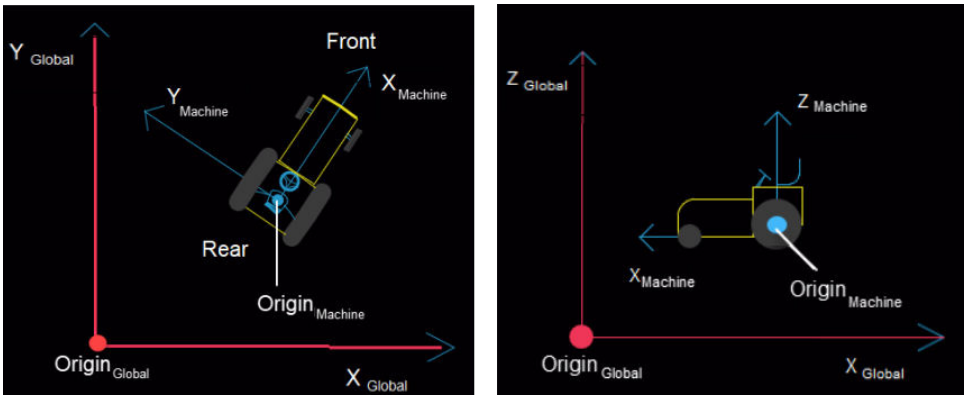
The developer must also define an origin on the machine, known as the machine coordinate system.

The Autonomous Control Library blocks can be programmed in infinite ways, but it is recommended to place the origin of the machine where there is very little lateral movement when steering. The Autonomous Control Library refers to the machine's origin as the steering point.

The image above shows recommended areas to place the origin (red dot) based on how the machine steers while driving. In differential machines, these steering points shift depending on the weight the machine carries and where that resides, such as a heavy bucket moving forward and back. In articulated machines, place the steering point in the part of the machine that steers in the direction of driving. For example, when moving forward, place the steering point on the axle between the front wheels.

Currently, crab steering vehicles require custom code.

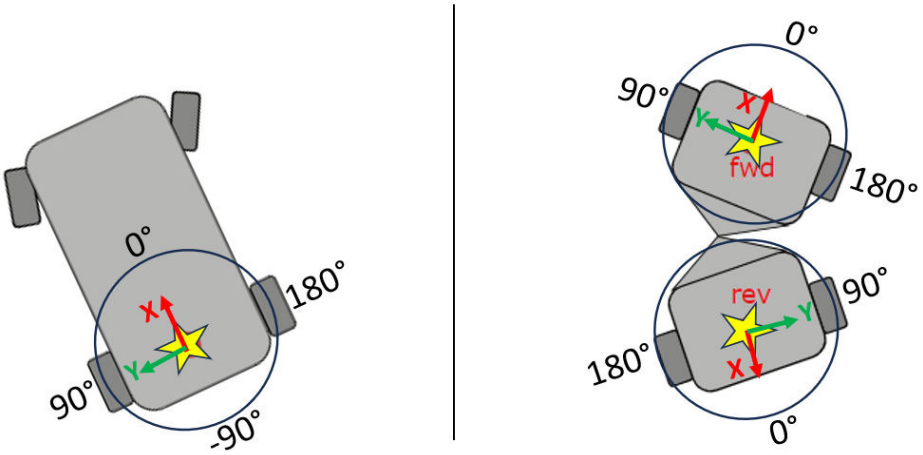
In this 3D machine coordinate system, the x-axis points from the rear towards the front of the machine. The y-axis points from the center towards the left of the machine. The z-axis points from the ground up.



In the images above, the global coordinate system appears as the red lines. The machine coordinate system is the blue lines. All sensors and offsets should be defined in the machine coordinate system. Each sensor has an origin and coordinate system, too.

Match all coordinate systems to the machine's coordinate system at the steering point. All types of machines except for articulated and crab assume positive X goes through the front of the machine at 0°. Positive Y is to the left of the machine at 90°. Up (height) is positive. Articulated machines include positive X going in the reverse direction, as well.

Background on Autonomous Machines

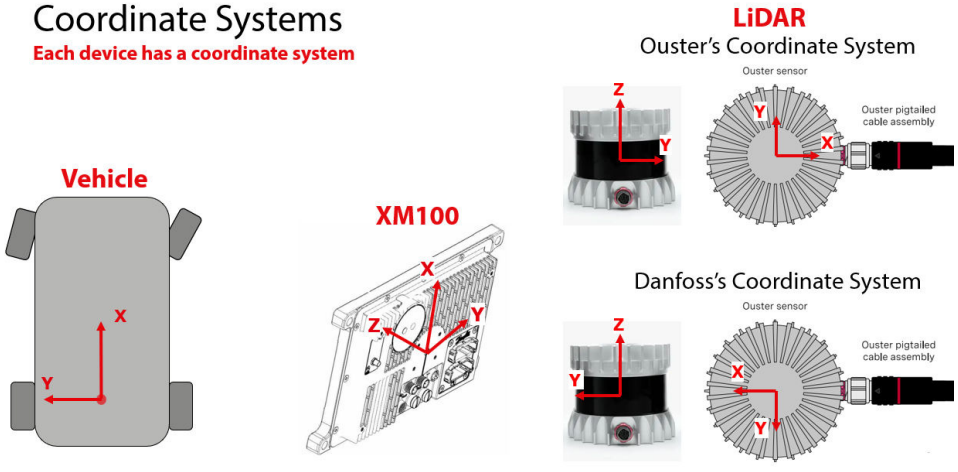


The left image above shows Danfoss machine coordinates move with the machine's origin. The right image shows articulated machines have two machine coordinates and machine origins. In ACL, the machine origins are referred to as steering points.

Sensor Coordinate System

Light Detection and Ranging (LiDAR), Inertial Measurement Unit (IMU), radar, ultrasonics, and cameras are types of sensors that have their own origin and coordinate system.

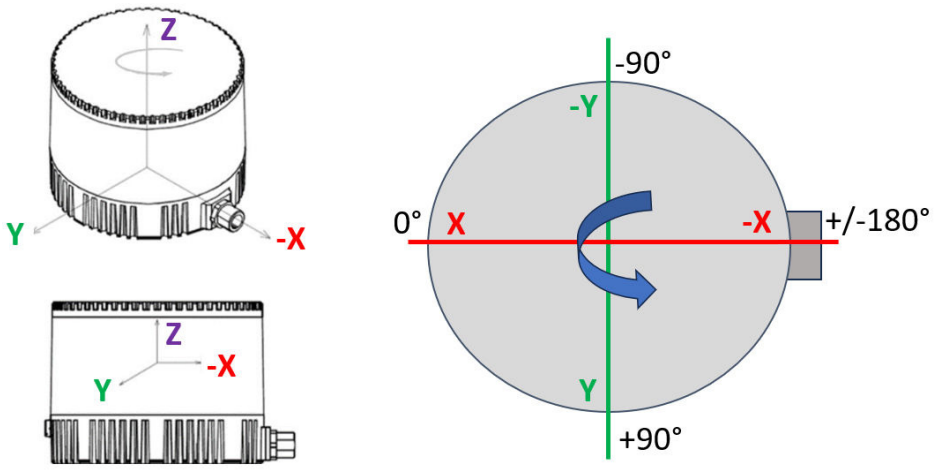
The XM100 contains an IMU and therefore requires an origin and coordinate system, as well.



The image shows Danfoss's coordinate frames for a machine, XM100 controller, and LiDAR. Danfoss recognizes the front of LiDARs as opposite the cable. The LiDAR manufacturing company, Ouster, uses a different coordinate frame where the cable is the front of the LiDAR.

Danfoss's LiDAR convention regards positive X as opposite the LiDAR connection at 0°, positive Y at 90°, and Z pointing up to the sky. The LiDAR rotates counter-clockwise and is divided into two 180 degree semicircles. Most LiDAR sensors do not match Danfoss's convention and need to be adjusted in the software code.

Background on Autonomous Machines



The image above shows Danfoss's LiDAR convention from a side view and top-down view.

Mount the XM100 parallel to the ground, and align the XM100's coordinate frame to the machine's coordinate frame. This can be done by mounting the XM100 over the machine's coordinate frame at the machine's origin, or by shifting the XM100's coordinate frame in the software to match the machine's coordinate frame.

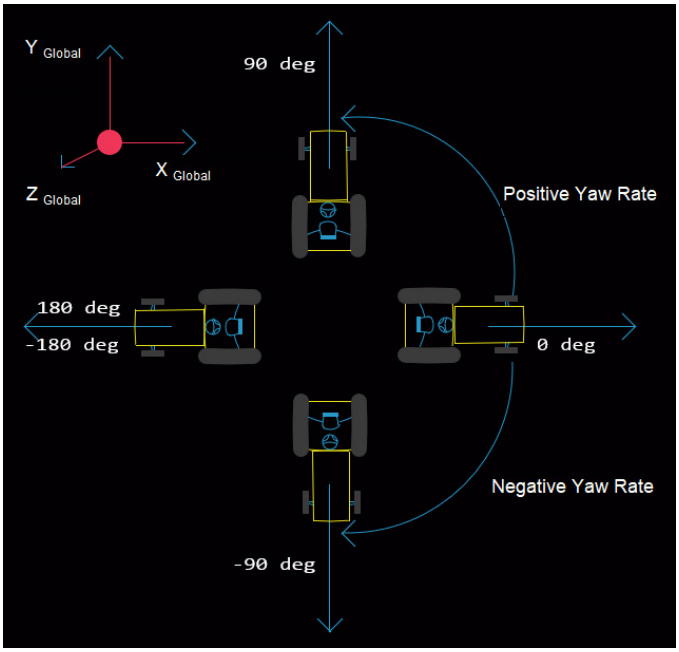
Process the raw sensor coordinate system data and then transform the output of the processed data. Or, transform the raw data into the correct coordinate system and then process the transformed data.

Yaw, Yaw Rate, and Velocity

Yaw refers to the direction the machine faces, and yaw rate refers to the machine turning.

In a global coordinate system:

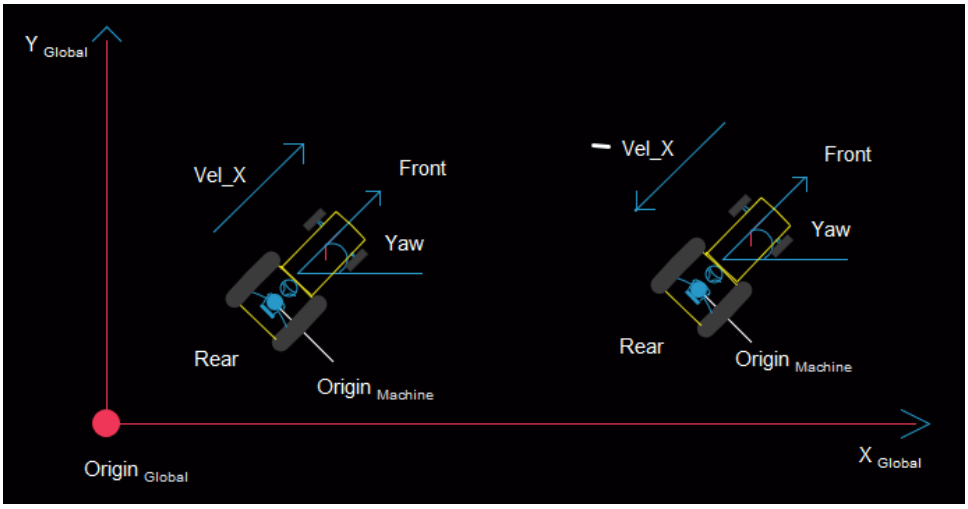
- Yaw is zero when the machine faces East.
- Yaw is 90 degrees when the machine faces North.
- Yaw is both 180 and -180 degrees when the machine faces West.
- Yaw is -90 degrees when the machine faces South.



Background on Autonomous Machines

Yaw rate refers to the machine turning with counter-clockwise yaw rate positive and clockwise yaw rate negative over a certain amount a time. For example, if a machine turns from 45 degrees to 90 degrees in 5 seconds, the yaw of the machine starts at 45 degrees, the yaw ends at 90 degrees, and the change between 45 and 90 is the yaw rate in 5 seconds. If a machine's yaw started at -100 degrees and stopped at -120 degrees in 5 seconds, the yaw rate is -20 deg/5 s = -4 deg/s.

A positive velocity corresponds to the machine moving forward. Therefore, the machine reversing should have a negative velocity. The yaw angle remains the same in both forward and backward motions.



The image above shows positive velocity when a machine moves forward and negative velocity when reversing.

Autonomous Library Blocks

Autonomy blocks use positioning in these ways.

The Autonomous Control function block library uses a Position_Filter to provide a method to combine GNSS data, IMU data, and odometry data. The **Position_Filter** function block processes the combined data to produce an improved estimate of the autonomous machine's position and orientation over time.

To produce location estimates, the Position Filter requires sensor data to be formatted to fit standard conventions. For example, raw GPS data is conventionally reported in latitude and longitude. The data must be transformed into the Universal Transverse Mercator (UTM) coordinate system.

The Autonomous Control Library supplies the **UTM_Conv**, **Origin** and the **Relative_Pos** function blocks to format the data properly for the Position Filter to use.

The UTM Conversion blocks (with and without zone selection) are used to convert from latitude and longitude to the UTM coordinate frame. The **Origin** function block captures the starting UTM position of the machine.

The **Relative_Pos** is stored as X (East being positive) and Y (North being positive) distance from the **Origin** in millimeters. This simplifies operations for downstream blocks, so calculations do not need to be done on raw latitude and longitude values.

The **Ackermann_Yaw_Rate** function block uses steering and speed sensor data to form a machine-centric odometry pair. Sensor standard deviation is used in the covariance matrix in the Position Filter.

Navigation

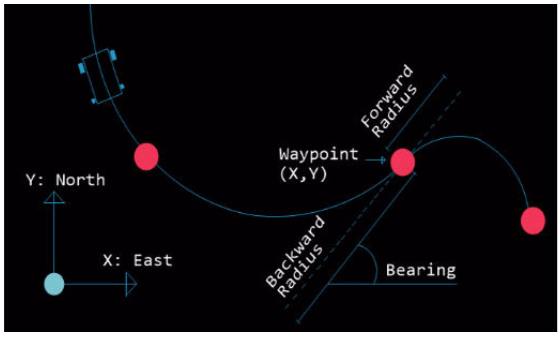
The machine navigates from one point to another after it establishes its position.

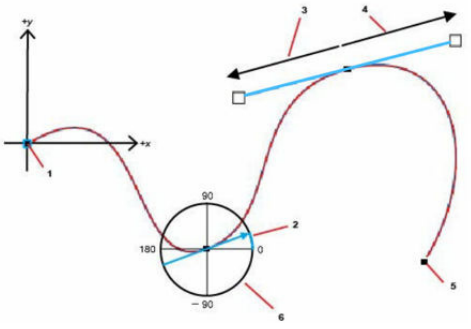
Blocks in the Autonomous Control Library use GPS to move, but some also work without GPS indoors. Machines navigate within their environmental frame of reference to their goal.

Background on Autonomous Machines

The figure that follows shows a basic path, which is a series of Bezier curves. A Bezier curve is a parametric curve used in graphics which uses points to create a smooth, continuous curve by means of a formula. Referred to as waypoints, these points include coordinate information and sometimes other data.

The Bearing is the angle at which the machine should pass through the waypoint. The curve between two waypoints is weighted by the length of the Forward Radius of the first point and the Backward Radius of the second point.





The left image shows a machine following a path by moving gently toward different waypoints. The right image is explained in the table below.

Item	Description
1	Origin
2	Bearing
3	Backward Radius
4	Forward Radius
5	Waypoint
6	East-North-Up (ENU) Notation. 0 (East), 90 (North), 180 (West), -90 (South)

Application Recommendations

Autonomous machines require certain features at a system, hardware, and software level in order to run. Then, Autonomous Control Library (ACL) function blocks can be pieced together in an application.

For an autonomous machine work properly:

- Have electronic steering and propel control in order to autonomously control the machine.
- Know whether the machine operates indoors or outdoors to be able to use GNSS.
- Use features in the environment to establish a location if the machine operates indoors, such as reflectors, posts, and walls.
- Understand what actions the machine should do autonomously and create machine states in the application.

Hardware and System Compatibility

Hardware must be compatible with PLUS+1® GUIDE.

Machines need to know their surrounding environment, where they are located, the direction they face, and whether they are tipped or turning. Use the XM100 controller to get this data, and mount it carefully or the machine will get inaccurate results about its orientation.

Examples of compatible hardware that connect to the XM100 include MC controllers, joysticks, and steering valves. Additionally, third party hardware such as Ouster LiDARs, LeiShen LiDARs, and Preco radars work in PLUS+1® GUIDE.

A Hemisphere GNSS antenna is supported with higher accuracy than the standard XM100 GNSS antenna, but requires a developer to write custom code to use it.

Software Libraries

Some libraries work well when creating an autonomy application.

The table below shows common libraries of pre-made code to download from the PLUS+1® Update Center and when to use them.

Go to **Select Files Manually** > **Edit** and look through the drop-down folders to select the correct libraries. Within PLUS+1® GUIDE, libraries appear under the **Function** or **Hardware** tabs.

Common Libraries

Category	Library Name	When to use it
Hardware	DM1000 HWD or DM1200 HWD	Used for visualization in conjunction with the XM100.
Hardware	XM100 HWD	Used for almost every autonomy project.
Hardware Partner Product	LeiShen	Used with LeiShen LiDAR hardware.
Hardware Partner Product	Ouster	Used with Ouster LiDAR hardware.
Hardware Partner Product	Preco Radars	Used with Preco radar hardware.
Software Application	Autonomous_Control_Library_Example_Application	Review this as an example application for an autonomy project.
Software Function	Autonomous Control Library SW00001310	Used for autonomy projects, which include path following, object detection, and much more.
Software Function	Filters Library	Used for filters and ramps.
Software Function	Middleware Library	Used to get all the fault manager information in one place.
Software Function	Remote Controls Library 70511659	Used to remote control into a machine if using a Danfoss RCT.
Software Function	Speed Sensor	Used to measure the machine's ground speed. There are many different speed sensor libraries.
Software Function	SW Template Library	Used when wanting the PLUS+1® template, which works well with function blocks. It is not required to be able to use function blocks.

Application Recommendations

Common Libraries (continued)

Category	Library Name	When to use it
Software Function	Utilities Library	Used for miscellaneous items and unit conversions, such as temperature, speed.
Software Tool	PLUS+1® GUIDE	Used with any autonomy project containing function blocks. Recommended to get the latest version.
Software Tool	PLUS+1® Service Tool	Used with any autonomy project containing function blocks. Recommended to get the latest version.

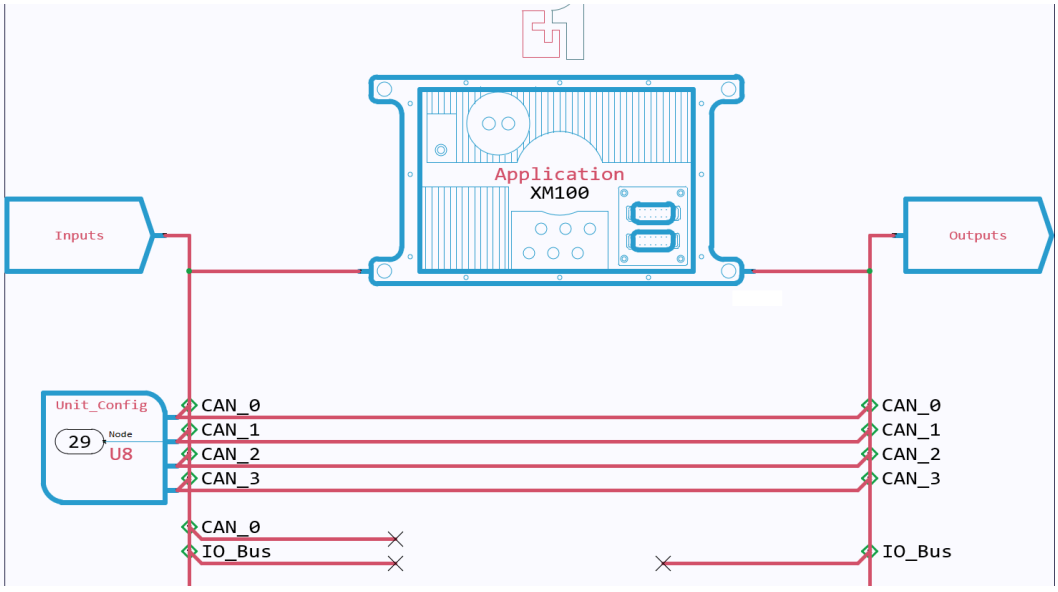
After downloading the libraries and connecting the hardware, set up the application in PLUS+1® GUIDE. Select a new project with XM100 as the hardware. Change the inputs and outputs depending on the sensors.

Autonomy Software System Template

It is recommended to use the XM100 template in conjunction with the Autonomous Control Library (ACL) function blocks.

Download the XM100 application template from the PLUS+1® Update Center. Review the API Specification sheet for more details, which is found within PLUS+1® GUIDE under **Project Manager** > **HWD** > **XM100** rather than the **Hardware** tab.

The ACL function blocks go in the middle **Application** section of the template. Go into **Unit_Config** and change the baud rate to match any sensors used in the system, such as Hemisphere. If no other sensors are used, then ignore **Unit_Config**.



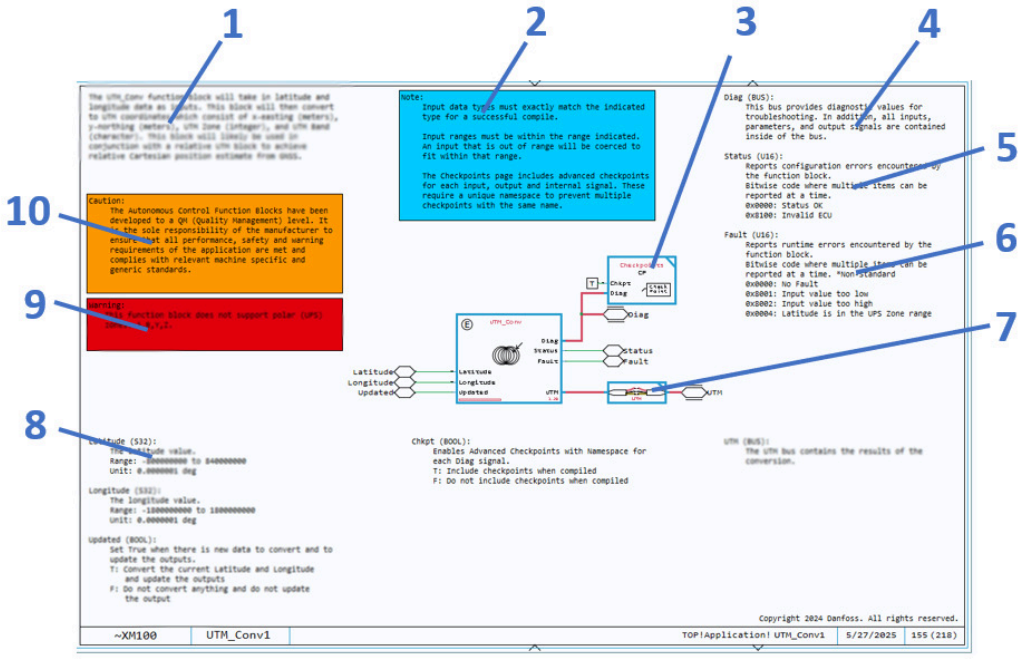
Note some autonomy specific items within **Inputs** and **Outputs**:

- Change **Ethernet**, but not **USBEthernet**, when connecting a LiDAR or another XM device.
- Go into **DirAPI** to set up interaction with a Linux file system.
- Go into **Interlink** to create a password to block access to a service tool.
- Optionally disable **Service Tool** so no one downloads the application while the machine moves.
- Go into **GNSS** to enable the GPS. Get time synchronized over satellites.
- Set up **Accelerometer** with the XM100's range and precision. The same range for XM100 occurs no matter which mode is used.
- Set up how fast the machine responds and the accuracy in **Gyroscope**. This usually uses mode 4.

Application Recommendations

Autonomy Function Block Template

Understand the autonomy function block template in order to use the blocks better.



Autonomy Function Block Template

Number	Item
1	The paragraph in the top left corner describes what the function block does and why to use it.
2	Any notes related to the function block appear in a blue box.
3	Click into the **Checkpoints** page to see what features in the function block are being monitored. This area includes **Internal Signals**, which show what happens internally inside the function block. Monitor these while testing or using the function block to see if any issues occur.
4	Diagnostic signals are used for debugging purposes and contain all signals related to the function block. This is always on the top right of the page. The **Diag** bus is the only place to view the internal signals.
5	**Status** shows any errors related to parameters that occur in the function block. This is listed on the top right of the page.
6	**Fault** shows errors related to inputs. This is listed on the top right of the page.
7	After the block processes the code, outputs appear on the right side. Outputs cannot be adjusted. If there are too many signals to list, a second page is created that includes the signal names and descriptions.
8	Code inputs and parameters appear on the left of the page. Parameters within the block can be adjusted, but inputs come from code entering the block and likely need to be adjusted upstream. Each signal includes the title, followed by the variable type in parenthesis, a description of the signal, the low to high span of the signal, and the units of measurement.
9	Warning messages appear in a red box and advise about major issues that could occur.
10	Caution messages appear in a yellow box and include information that should be followed.

Pre-Made Service Tool Screens

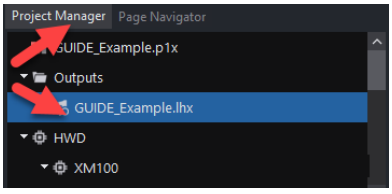
Each function block comes with an accompanying service tool screen with the input, parameter, output, and error signals already assembled for quick and easy viewing.

Pull many function block pre-made service tool screens together to view more of the application.

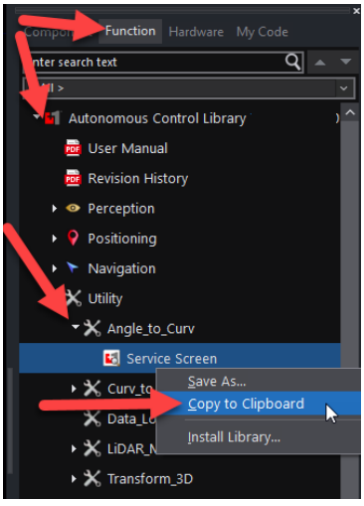
1. Connect the hardware, such as the XM100, to a computer and turn it on.

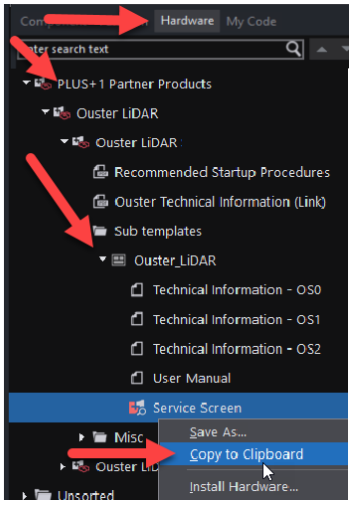
Application Recommendations

2. Open PLUS+1® GUIDE.
3. Download the application to the controller by double clicking .lhx.



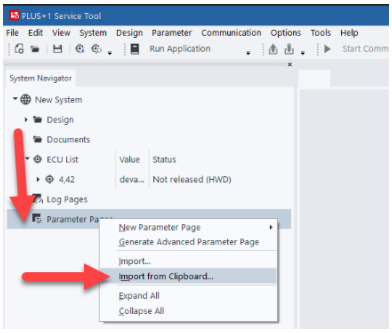
4. Find the function block whose Service Tool information will be viewed.
   - For ACL blocks, select **Function** > **Autonomous Control Library**.
   - For LiDAR related blocks, select **Hardware** > **Plus+1 Partner Products**.
5. Select the drop-down arrow next to any function block to view the **Service Screen** beneath it.
6. Right click on **Service Screen** and select **Copy to Clipboard**.





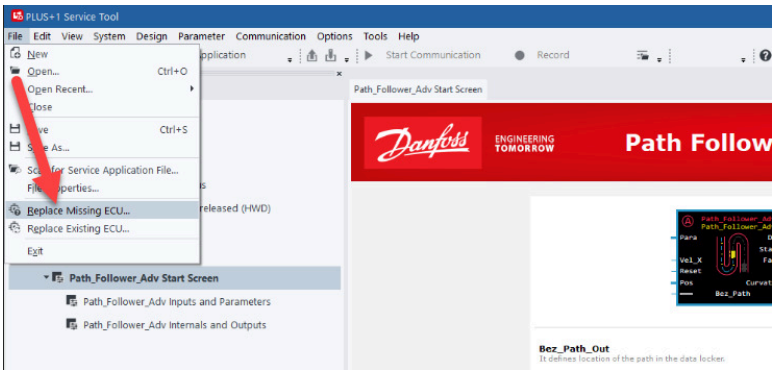
The image above shows how to copy to the clipboard.

7. Open PLUS+1® Service Tool.
8. Select the drop-down arrows for **New System** > **ECU List**.
9. Right click on **Parameter Pages** and **Import from Clipboard**.



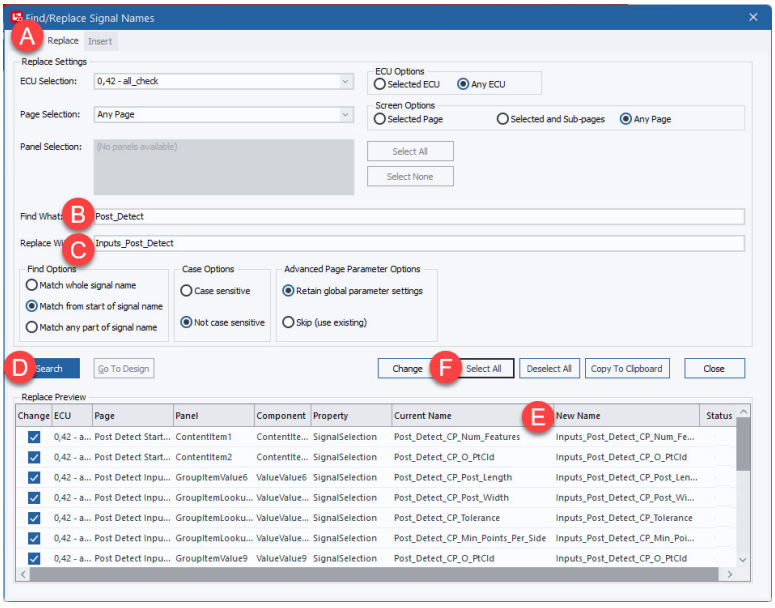
Application Recommendations

10. If this is the first time loading the page, select **File** > **Replace Missing ECU**.



The pre-made service tool screen appears. If no values appear in the fields, check that the hardware is connected and the PLUS+1® GUIDE code compiled.

11. Optionally, if changing the default namespace, modify the service tool pages to match the namespace. At the top of the Service Tool screen, select **Edit** > **Find/Replace Signal Names**.



The image above shows the steps to change namespaces with an example name filled in.

a) Go to the **Replace** tab.
b) Type which default namespace to replace in the **Find What** section.
c) Type a new name to replace the default with in the **Replace With** section.
d) Select **Search**.
e) Review the **New Name** list to see if the replacement names work well. If not, replace with a different name.
f) Select **Select All** > **Change**.
g) Select **Apply** when the pop-up appears to accept the changes. All the times the default name appears should be changed to the replacement name.

Modify JSON and Update MD5

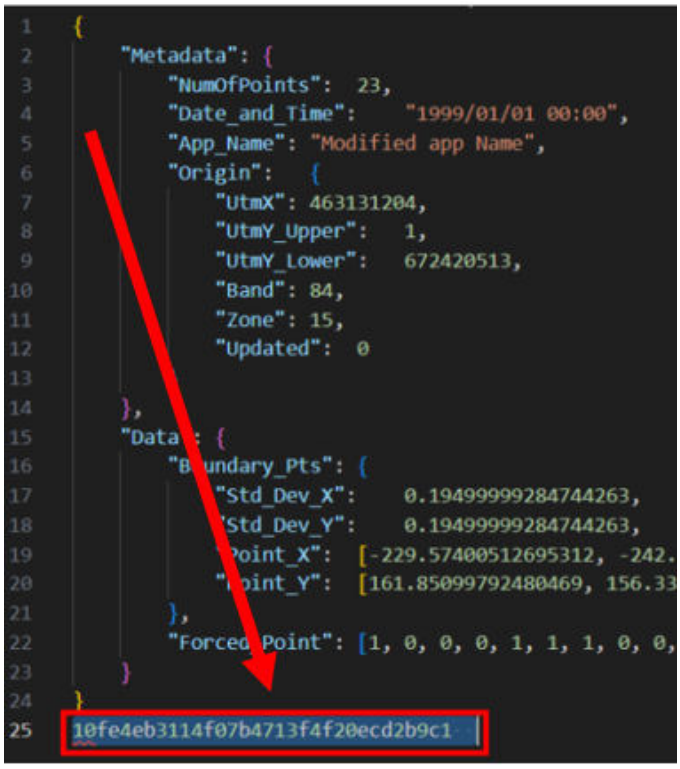
The following are steps to modify a JSON file and then update the MD5.

A JSON file is associated with **Boundary_Recorder**, **Boundary_Loader**, **Path_Recorder**, and **Path_Loader** function blocks.

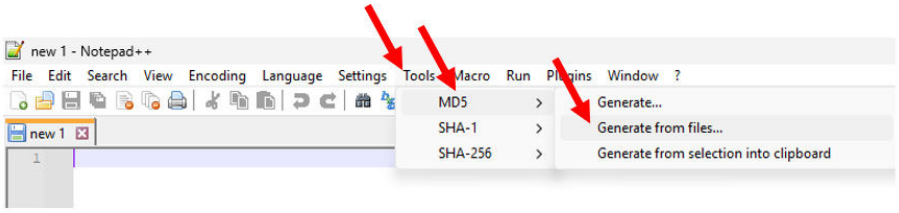
Application Recommendations

Before beginning, record a path or boundary to create a JSON file. Install **Notepad++** or a similar program to modify the JSON file.

1. Open the JSON file.
2. Optionally, modify any of the data. For example, modify the X and Y coordinate points.
   a) Delete the original MD5, as shown in the image below.



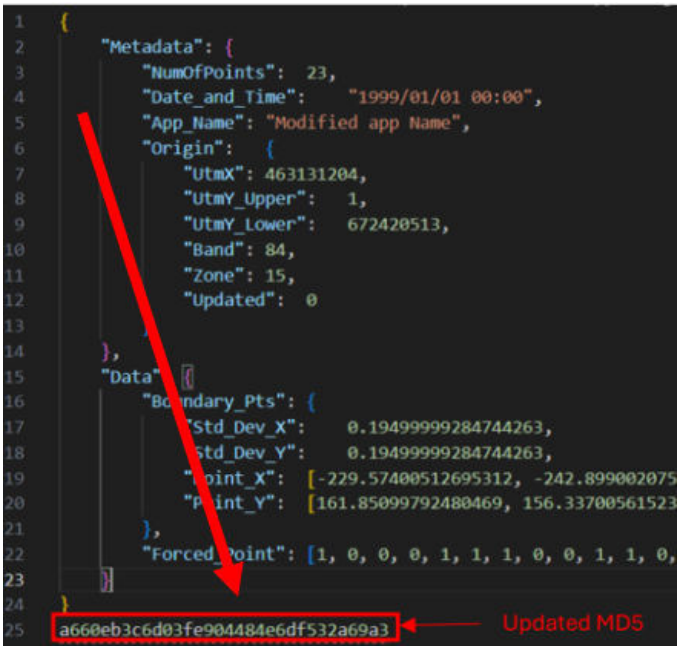
   b) Keep the cursor on the new line. In the image above, that is line 25.
   c) Save the file.
3. Open **Notepad++**. Install this program if not already installed. Similar programs can be used to modify the JSON file if **Notepad++** is not available.
   a) Go into **Tools** > **MD5** > **Generate from files**.



   b) Select **Choose file to generate MD5**.
   c) Browse for the modified JSON file.
   d) Copy the newly generated MD5 that appears.
   e) Save and close the file.

Application Recommendations

4. Go back to the JSON file.
   a) Paste in the new MD5. The image below shows this on line 25.



   b) Save the file.
5. Upload the JSON file to a controller, such as the XM100. See *Getting Files from XM100* on page 29.

Getting Files from XM100

Save and access files from the XM100.

1. Open the Service Tool program. The XM100 must be connected to the computer.
2. Select **New Service Application** or open an existing application.
3. Go to **New System** > **ECU List**. Find the XM100 hardware in the list.
4. Select **Manage ECU media files**, which should open a new window. The application will stop running in the background.
5. Select **Start**.
6. Go into **Optional Files** > **P1user**. A new folder could be created under there, too.
7. Select the file to save to the computer.
8. Select the **Save File** icon at the top of the screen to save the file.
9. Pick an area to save the file on the computer to access later.

   Other options include:
   - Import a file by selecting the **Add File** icon at the top of the screen. For example, import a JSON file from another machine recording for some of the Path function blocks.
   - Save a modified JSON file by updating the checksum numbers. Modifying a JSON file is not recommended.
   - Delete a file by selecting the trashcan icon.

Restart or Resume Recording After ECU Power Loss

If the ECU loses power, the current boundary or path recording is lost and must be recovered.

To recover, perform one of the following:

Application Recommendations

- Restart the boundary or path.
- Resume the path. This cannot be done with boundary recordings.

1. To restart the boundary or path:
   a) Move the machine to the beginning of the boundary or path.
   b) Reload the boundary or path the machine was trying to complete.
   c) Start following the boundary or path.
2. To resume the path:
   a) Reload the path while having **Search_Path** set to True. This ensures that the entire path is searched, to find the nearest point to the machine.
   b) Ensure the **Tracking_Error** is less than the **Lookahead_Dist** to ensure a smooth start.
   c) Start following the path.

How ACL Blocks Work Together

The Autonomous Control Library (ACL) includes blocks of pre-made code to use in applications for autonomous machines.

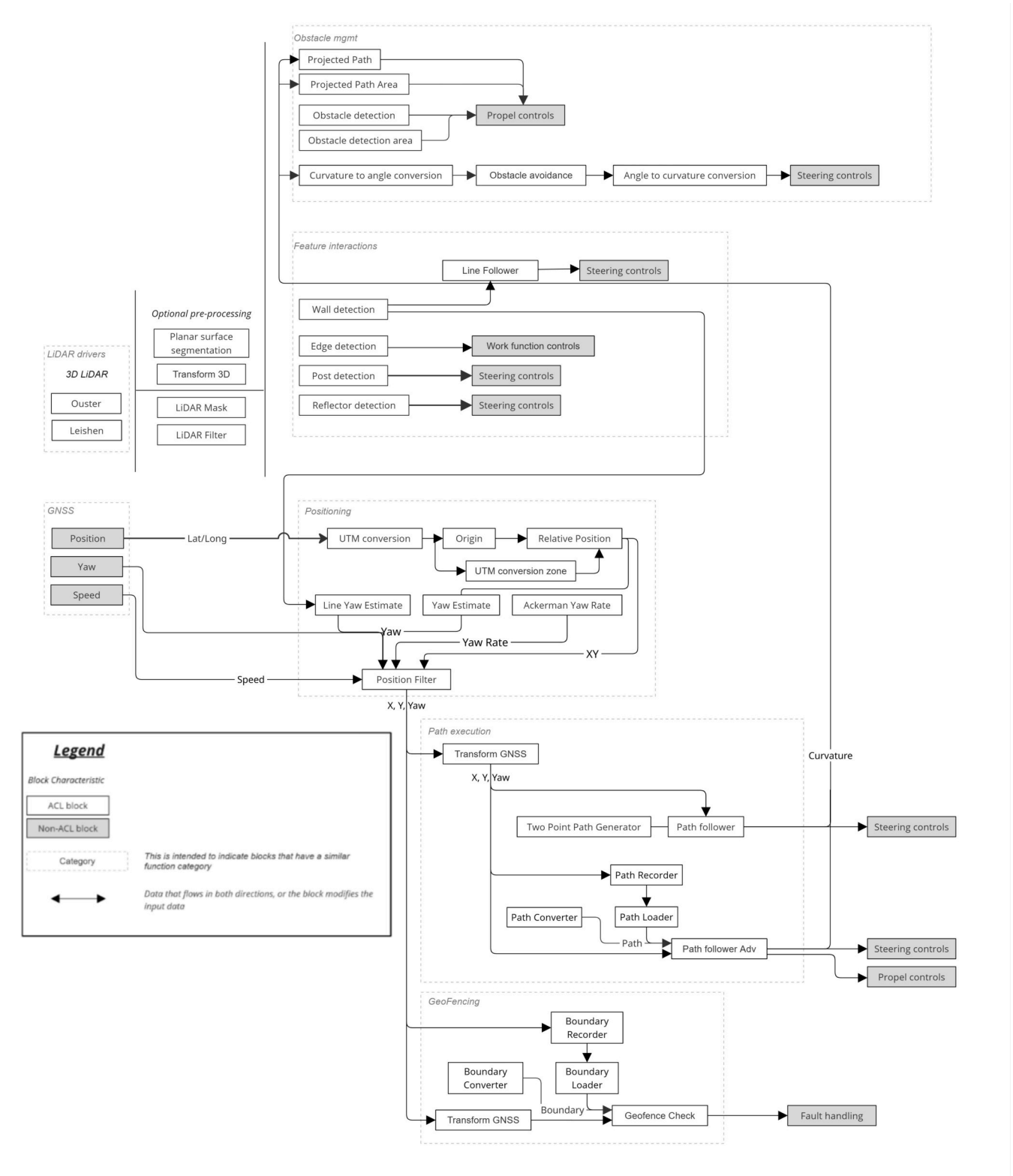
An autonomy application needs to satisfy perception, positioning, and navigation concepts for a machine to work autonomously. ACL includes function blocks that fall into each category.

Follow the diagram to see which ACL function blocks work in an application. The names of the function blocks appear in the ACL drop-down within PLUS+1® GUIDE with more descriptions. A LiDAR block or code for a LiDAR is required for the other ACL perception blocks to work.

Begin with the LiDAR block early in the application. Decide whether to use any of the optional pre-processing blocks, or skip to the function blocks. The flowchart shows which blocks or features come before others and how they relate to parts of the machine. There are many combinations of the blocks. Read a block's individual chapter to see more combinations.

The Autonomous Control Library function blocks have been developed to a QM (Quality Management) level. It is the sole responsibility of the manufacturer to ensure that all performance, safety and warning requirements of the application are met and complies with relevant machine specific and generic standards.

How ACL Blocks Work Together

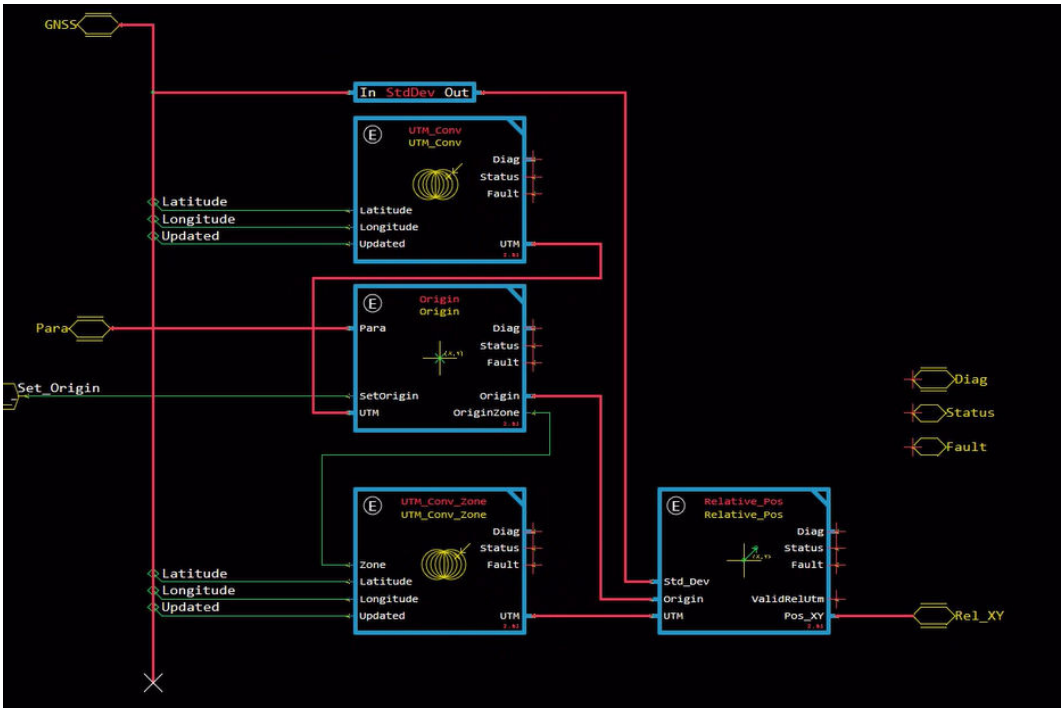


Common Software Set-Up

A common combination of four function blocks go into an autonomy application. They tell the machine its location.

How ACL Blocks Work Together

Additionally, view the autonomy application example from the PLUS+1® Update Center to review how the function blocks could be set up.

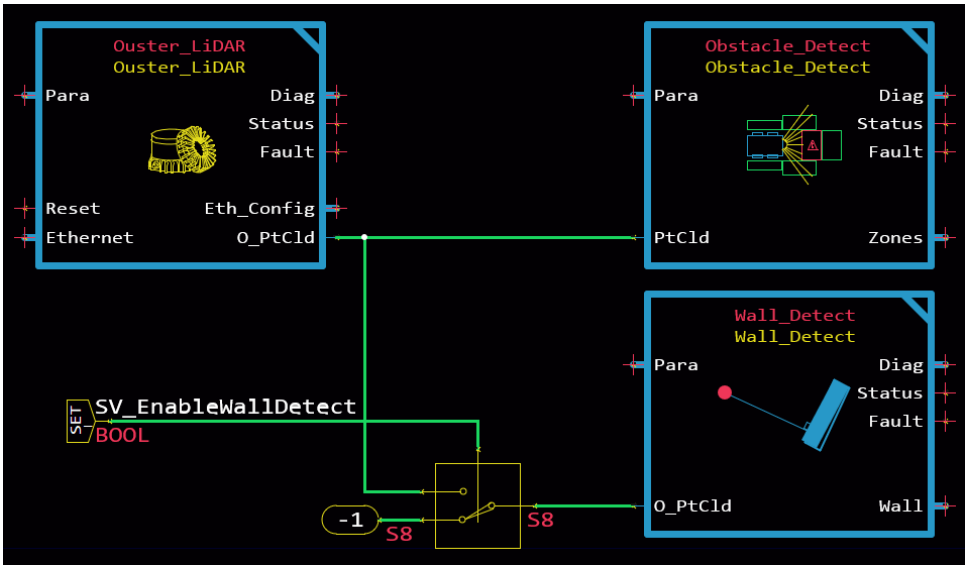


The image above shows a typical set-up with the **UTM_Conv**, **Origin**, **UTM_Conv_Zone**, and **Relative_Pos** function blocks, which establish the machine's location.

Save Processing Time

For a machine to function as efficiently as possible, configure the code to save processing time.

One way to save processing time is to turn off sections of the code when they are not in use.



In this example, when **SV_EnableWallDetect** is True, the **Wall_Detect** function block is enabled using the output from the LiDAR function block. When it is False, **Wall_Detect** is disabled, and the **O_PtCld** input value is set to -1.

How ACL Blocks Work Together

Using Namespaces

Namespaces help successfully compile an application that uses the same function block more than once.

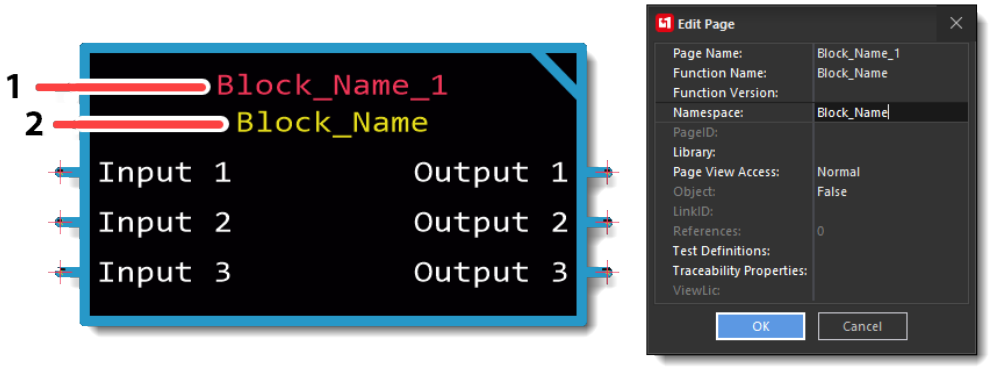
Change each function block's namespace by setting its **Namespace** value to something unique. The application cannot compile without changing the **Namespace** value.

The **Namespace** value adds a unique prefix to each component name.

Also, to use these function blocks' companion Service Tool screens, include the function block's advanced checkpoint with namespace in the application's compiled .lhx file. Use the function block's **Checkpoints** page to include the checkpoint.

Change Namespace Value

To successfully compile an application, change the namespace value for function blocks that are used more than once in an application.

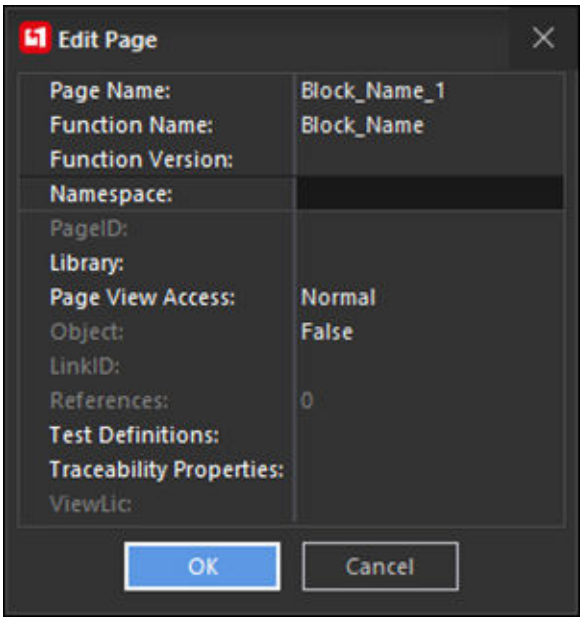


1	Page Name
2	Namespace

1. Enable Query/Change mode.
   - Select **Edit** > **Query/Change**.
   - Or, press **Q**.
2. Click on the function block to modify the namespace.
   The **Edit Page** dialog box opens.

How ACL Blocks Work Together

3. In the **Namespace** field, enter a unique Namespace value.



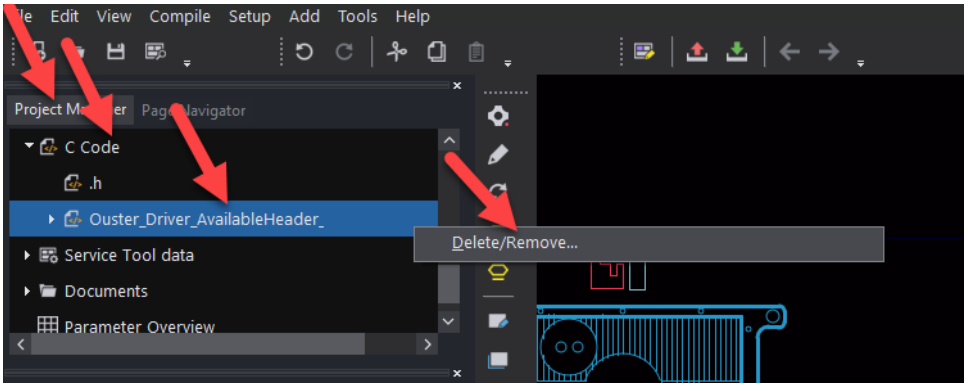
- **Namespace** values are case-sensitive.
- To save controller memory, use a short namespace value.

4. Click **OK**.
5. Repeat these steps to enter unique namespace values for other identical function blocks.

Delete the Old Function Block C Code

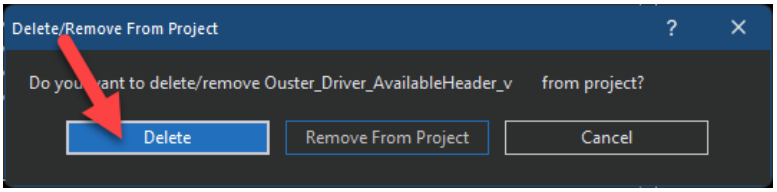
If updating to a new function block from an older version of itself, delete the old block from the C Code first. If this is the first time using a particular function block, skip this task.

1. Open the application that has the old function block in **PLUS+1 GUIDE**.
2. Within the application, go to **Project Manager** > **C Code** > **[LibraryName]_AvailableHeader.h**.
3. Delete all the old files by right clicking on the .h file, then selecting **Delete/Remove**.



How ACL Blocks Work Together

4. Select **Delete** from the message.



5. Delete the old block from the user interface.
6. Save and close the application. Now, the newest version of the function block can replace it.

Troubleshooting Common Errors

The following table describes common errors that could occur in many of the function blocks and ways to fix them. Multiple errors can be reported at a time.

Specific errors occur in individual blocks and are listed in each block's troubleshooting section in the user manual.

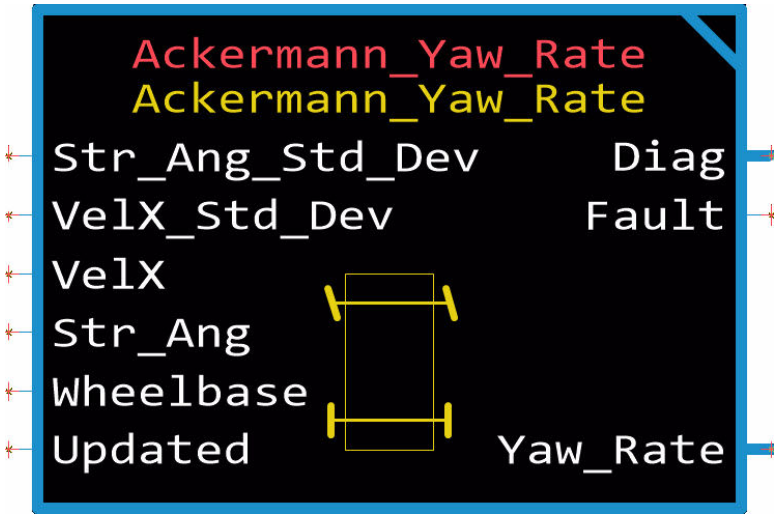
View these error numbers on the Service Tool screen for individual function blocks. In PLUS+1® GUIDE, these signals are on the **Checkpoints** page in the **Internal Signals** column.

Common Error Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware. Not enough memory to create internal structs.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
Bit 1	Execution time longer than expected or more than 500 ms.	If using a LiDAR, reduce the LiDAR resolution or delete other processing blocks. Turn the controller off and on, or use less code in the application.
Bit 2	Cannot create background thread.	Turn the controller off and on, or use less code in the application.

Ackermann_Yaw_Rate Function Block

The **Ackermann_Yaw_Rate** function block provides yaw rate information to the **Position_Filter**.



The function block converts machine speed and steering angle into a velocity and yaw rate machine-centric odometry pair. Use this block in place of a gyroscope sensor. The accuracy of this block depends on the accuracy of the machine sensors. A IMU gyroscope may provide superior accuracy depending on the application.

The pair consists of the linear velocity (meters/second) and the angular velocity (degrees/second). The values are defined relative to a coordinate frame where:

- The X-axis points forward along the machine.
- The Y-axis points left along the axle.
- The Z-axis points up.

Sensor variance—the noise observed in sensor data—can be manually set if it is not provided by the sensor.

The standard deviation of a sensor characterizes the amount of noise in the sensor. This is obtained from sensor documentation of manually calculating from a log of steady-state sensor data.

Inputs

Inputs to the **Ackermann_Yaw_Rate** function block are described.

Item	Type	Range	Description [Unit]
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.
Str_Ang_Std_Dev	U32	1-4294967295	The standard deviation of the steering angle. [0.01 degree]
VelX_Std_Dev	U32	1-4294967295	The standard deviation of VelX. [mm/s]
VelX	S32	-25000-25000	The linear velocity of the machine. [mm/s]
Str_Ang	S16	-7000-7000	The angle between the front of the machine and the steered wheel direction. Negative values are to the right. Positive values are to the left. [0.01 degree]

Ackermann_Yaw_Rate Function Block

Item	Type	Range	Description [Unit]
Wheelbase	U16	300-20000	The distance between the centers of the front and rear wheels. [mm]
Updated	BOOL	T/F	True when there is new data. T: New data is ready. F: New data is not ready.

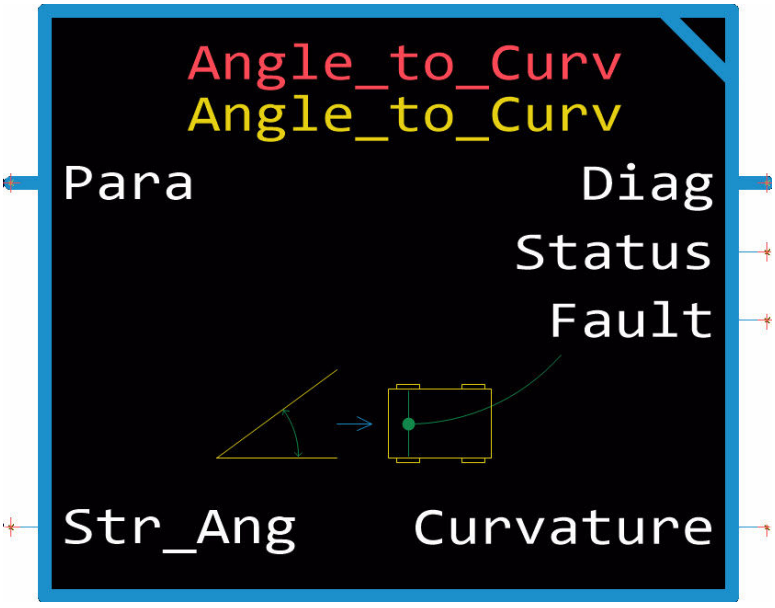
Outputs

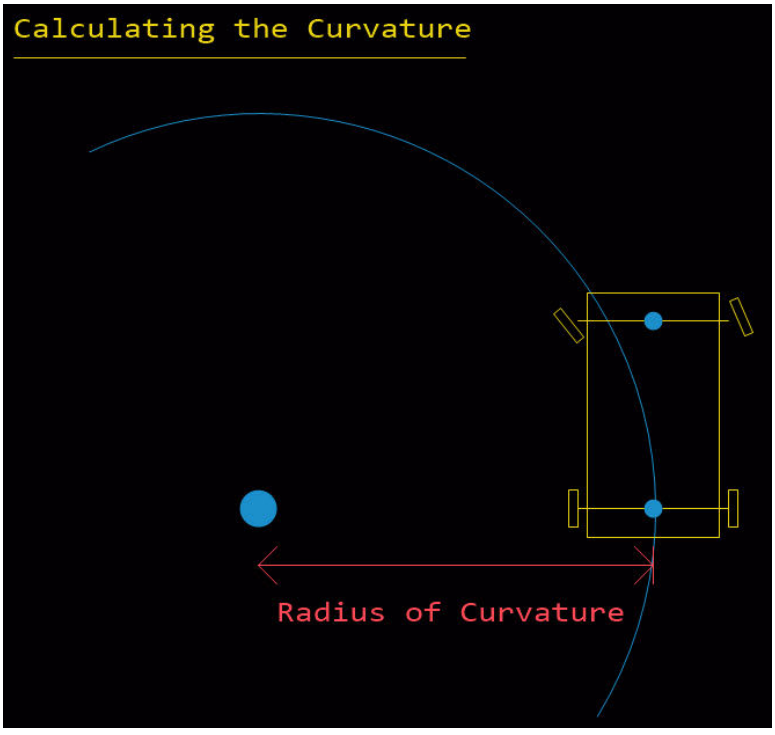
Outputs of the **Ackermann_Yaw_Rate** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Yaw_Rate	BUS	——	This bus contains Yaw Rate and its standard deviation data.
Yaw_Rate	S32	-1312080-1312080	The angular velocity of the machine relative to the machine's vertical axis. [0.01 deg/s]
Yaw_Rate_Std_Dev	U32	1-4294967295	The standard deviation of Yaw_Rate. [0.01 deg/s]
Updated	BOOL	T/F	True when new data is available from the conversion. T: New data is available. F: New data is not available.

Angle_To_Curv Function Block

The **Angle_To_Curv** function block converts a steering angle to curvature.





Angle_To_Curv Function Block

Inputs

The following table describes the inputs of the **Angle_To_Curv** function block.

Item	Type	Range	Description [Unit]
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.
Str_Ang	S16	-9000 to 9000	The angle between the front of the machine and the steered wheel direction. Negative values are to the right. Positive values are to the left. [0.01 deg]

Parameters

The following table describes the parameters of the **Angle_To_Curv** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routes from the application through the **Para** BUS.
Wheelbase	U16	300-20000	The distance between the centers of the front and rear wheels. [mm] Default: 5000

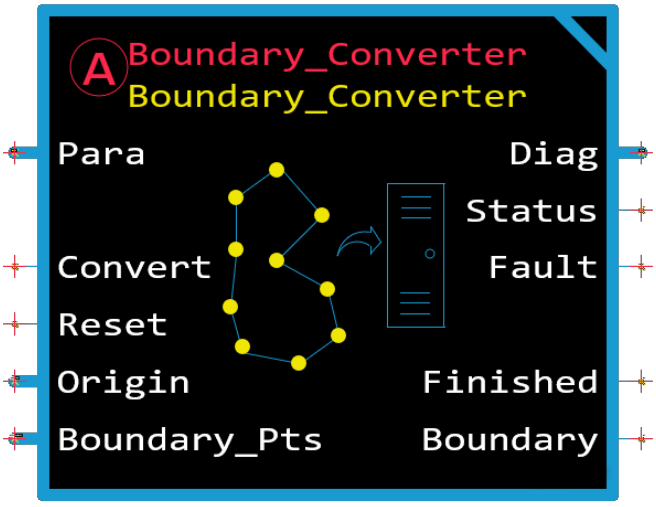
Outputs

The following table describes outputs for the **Angle_To_Curv** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Bus containing diagnostic values for troubleshooting. In addition, all inputs, parameters, and output signals are contained inside of the bus.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Curvature	S32	-2,147,483,648 to 2,147,483,647	Curvature calculated based on the steering angle and wheelbase of the machine. Negative values are right curves. Positive values are left curves [0.01/km]

Boundary_Converter Function Block

The **Boundary_Converter** function block allows manual boundary creation, and then passes that boundary information to a data locker for other code to access.



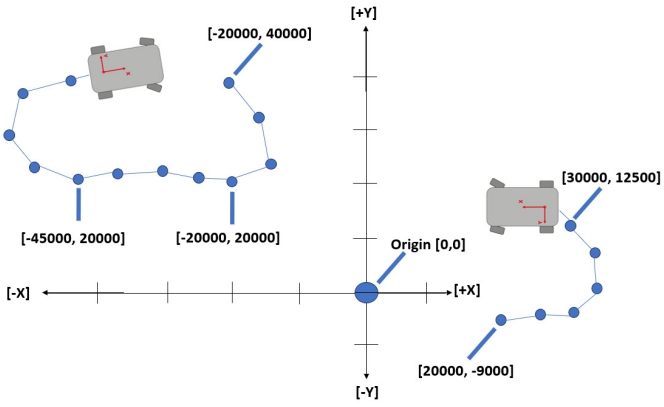
This block requires a license for A+ Advanced.

Boundary_Converter allows information about a geographic boundary to be entered directly into PLUS +1® GUIDE, rather than driving a machine around to record boundary information. The latter requires the **Boundary_Recorder** function block, and usually an application does not need both types of blocks. Use **Boundary_Converter** if very confident about the boundary point locations. It is best with easy shapes, such as squares.

Boundary_Converter reads the parameters, origin, and information about the boundary. It writes the boundary information into a data locker when the **Convert** pulse is given. **Geofence_Check** and **Boundary_Extract** function blocks use the converted information from the data locker.

Before using **Boundary_Converter**, establish the machine's position and the boundary origin earlier in the code. Use a new **Boundary_Converter** function block for each boundary, but use the same origin for all boundaries. Set the boundary origin in the same UTM zone as the machine's origin or the values will be very large.

The **Origin** bus brings origin data into **Boundary_Converter**. All the boundary points are with respect to their distance from the origin. **NumOfPoints** determines the length of the arrays, and then enter X and Y coordinates manually into the arrays. Ensure that the **NumOfPoints** corresponds to the number of X and Y points provided. If there are less points than the **NumOfPoints** number, an error occurs. If there are more points than **NumOfPoints**, then an unexpected boundary occurs using up to the **NumOfPoints** points.

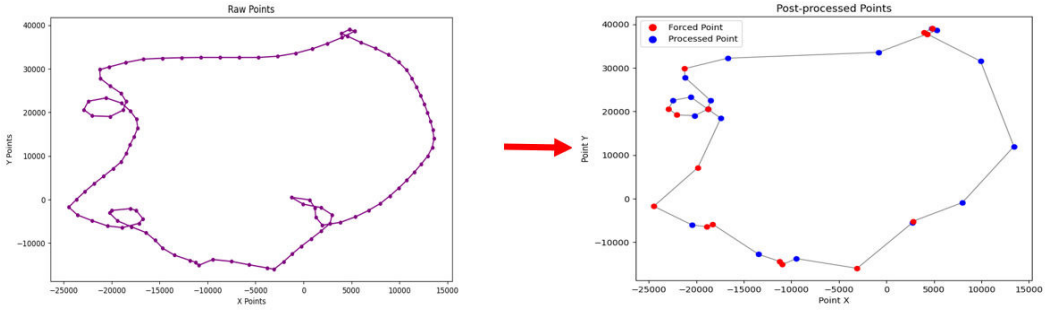


Boundary_Converter Function Block

The image shows two distinct boundaries. The XY boundary coordinates relate to an origin in the environment, not necessarily the machine's XY coordinates or origin. For example, one boundary point reads -45000 mm to the left of the origin along the x-axis, and 20000 mm above the origin along the y-axis. Straight lines automatically connect between points.

It is recommended to keep the first and last boundary points as the same value when inputting the array. For example: 100, 200, 250, 100. Use less than 25,000 points in a boundary.

All entered XY coordinates are raw data points **Boundary_Converter** uses, so they are not filtered by the position filter. All points entered in the X and Y arrays up to the **NumOfPoints** value are used in the boundary unless part of an inner loop. **Forced_Point** also makes a point part of the boundary. When a forced point is part of an inner loop, it is deleted and not part of the boundary. However, input and output arrays may differ if there are inner loops in the array. If an inner loop is needed, use two **Boundary_Converter** and **Geofence_Check** blocks. For example, a pond in a field would need two boundaries to keep a machine inside the field but outside the pond.



The image above shows inner loops inside the boundary eliminated and external loops kept. Only the outer boundary shell and forced points remain as the final boundary. Further processing occurred to eliminate some of the raw points in the right image.

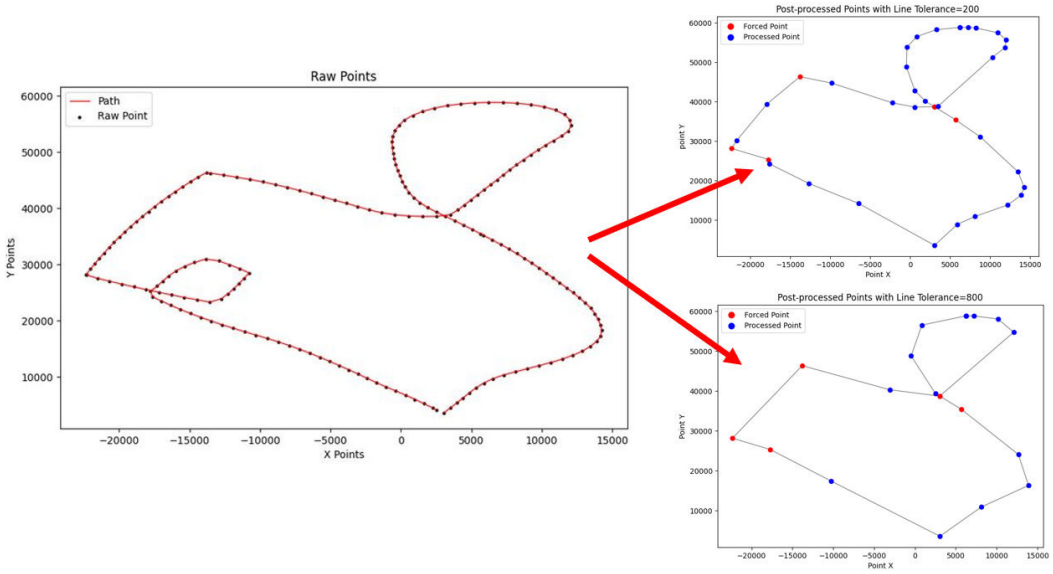
Manually write in an application name, date, and time, which can be anything the developer chooses. Keep the application character limit less than 255, and timestamp in format YYYY/DD/MM hh:mm.

The standard X deviation is the same for all X values, and the same concept applies to the Y standard deviation. Smaller numbers give more precise measurements. For example, entering a 5 for X means that the value could fall within the standard deviation range from 5 points behind and 5 points ahead of the value, for a total range of 10.

The **Closed_Polygon_Threshold** is the distance allowed between the first and last point in order for the boundary shape to be considered closed. For example, entering 500 mm means that if the actual distance between the first and last boundary points are 501mm, then an error occurs due to the gap between the first and last points being too large to complete the boundary.

Line_Fit_Tolerance adjusts the boundary. Entering 0 means the line points must match exactly. For example, if there are 10 coordinate points in a line with 0 **Line_Fit_Tolerance**, the line would be jagged to match the coordinates. Entering 1000 mm in **Line_Fit_Tolerance** gives more tolerance to create the boundary line and allows points to filter out, aside from forced points. That allows a smoother line, so higher **Line_Fit_Tolerance** values give more efficient boundaries.

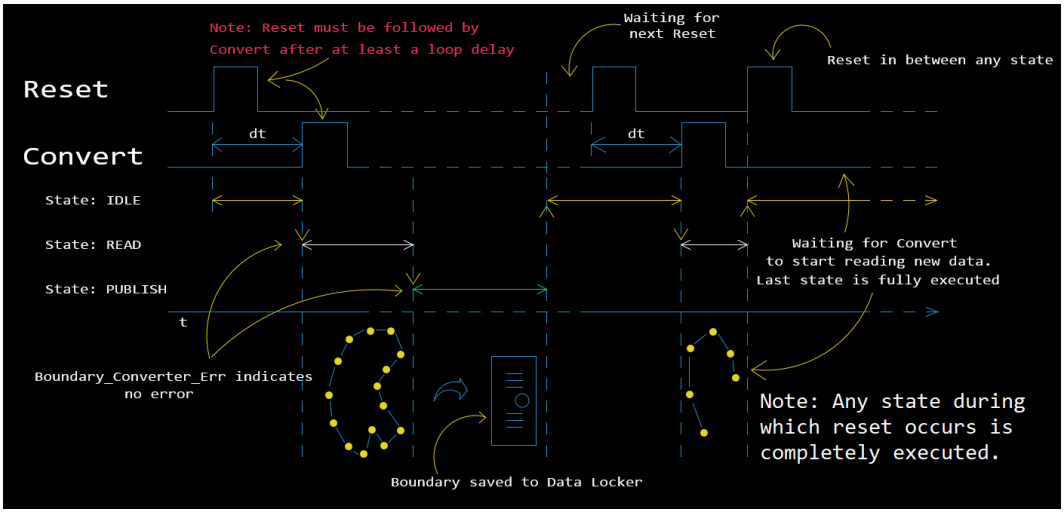
Boundary_Converter Function Block



The left image shows a boundary with raw data points. The right images show the same boundary with a small line fit tolerance (top) and larger line fit tolerance (bottom) after the filtration process. The areas where the loops intersect create an extra boundary point.

Set up the **Convert** and **Reset** input pulse signals last. **Convert** processes boundary data from the inputs and parameters. If there is a valid boundary, information flows into a boundary type data locker. Incorrect boundary data creates an error and changes the **Boundary_Converter** state to idle or paused. If **Reset** is True, then another boundary can be converted, but **Reset** cannot be used until the boundary is finished or the block would go to an idle state. During a test with the machine, check the **Boundary_Converter_State** internal signal to make sure the block is not in a read state, which is 1. If it is 1, then the conversion already completed.

If the boundary points are changed or modified after data went through, then the new information overwrites the previous boundary information. When the **Finished** flag turns to 1, the boundary data locker was written. On a service tool screen, monitor the standard deviation to check that it is within an allowable tolerance.



The image above shows what occurs in **Boundary_Converter** regarding reset, convert, and the state of the block.

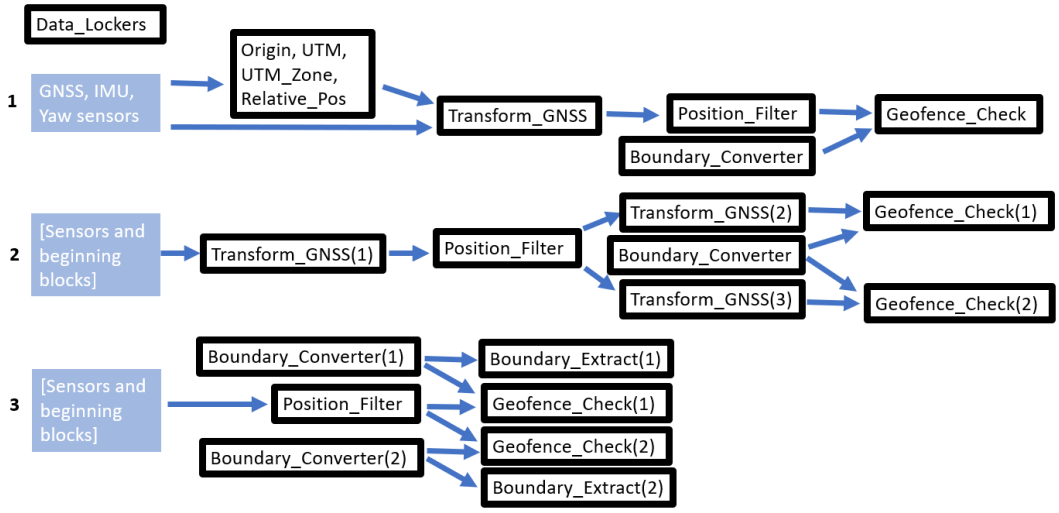
Boundary_Converter Function Block

After setting up **Boundary_Converter**, use the **Boundary_Extract** function block to validate the boundary. Plot the boundary in a separate program, such as Excel, to verify the boundary formed as expected.

Use less than 25000 points in the boundary. During the boundary conversion, the block may use a significant amount of controller memory. It is recommended to have the machine standing still during the boundary conversion.

Application Information

Common function blocks that work with **Boundary_Converter** are **Data_Lockers** and **Geofence_Check**.



1. Scenario one uses **Transform_GNSS** to determine when a section of the machine crosses a boundary in **Geofence_Check**. In this case, the coordinate information moves from the GNSS and yaw sensors to another part of the machine, such as the machine's origin. Boundary information flows from **Boundary_Converter** into **Geofence_Check**. Further code tells the machine how to react after the machine's origin crosses the boundary.
2. Scenario two adds more **Transform_GNSS** after **Position_Filter** determined the machine's origin. Two more **Transform_GNSS** locate two other coordinate points on or around the machine, such as the front and back of the machine. In parallel, **Boundary_Converter** gives boundary data to **Geofence_Check**, which reads if the two areas of the machine pass over the boundary.
3. Scenario three shows two boundaries, represented by two **Boundary_Converter** blocks. Each boundary requires a **Boundary_Extract** block to display the filtered boundary data on a Service Tool screen or hardware display, which could differ from the raw boundary data. **Position_Filter** determines the machine's position and passes that information into each **Geofence_Check**. **Geofence_Check(1)** determines whether the machine crosses the first boundary, and **Geofence_Check(2)** determines whether it crosses the second boundary.

Additionally, place the **Boundary_Converter** function block:

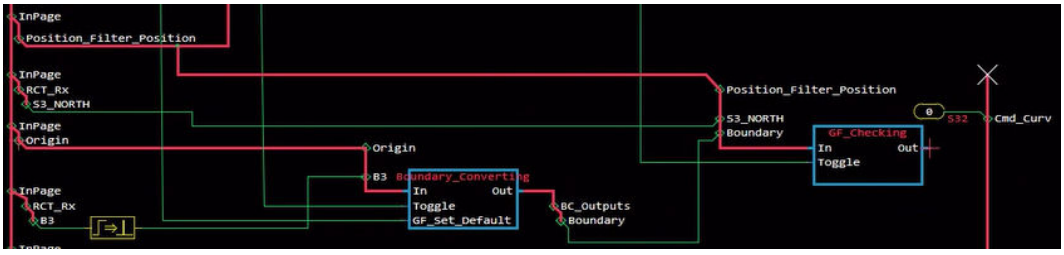
- With one **Data_Lockers** block, version 1.12 or later, which can be on any page in the application.
- One or more times in an application if there are multiple boundaries.

Example

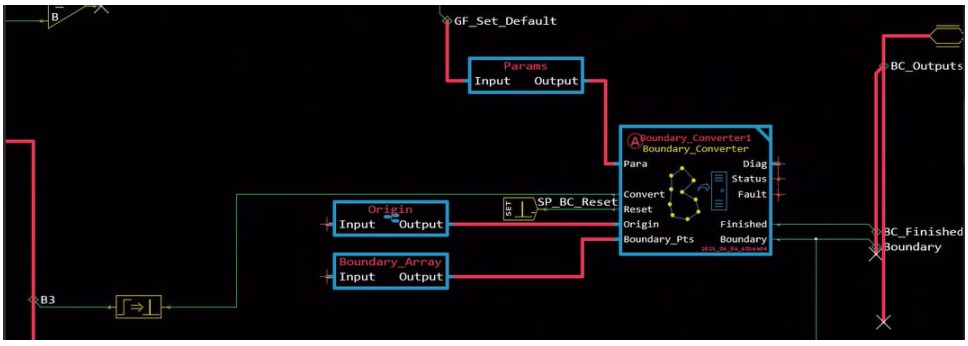
The example shows the **Boundary_Converter** function block generating a manually entered boundary that **Geofence_Check** uses. The boundary keeps a machine inside a parking lot.

Before beginning, establish the machine's origin and positioning system. See *Common Software Set-Up* on page 32. Determine where the boundary and boundary origin will be in the surrounding environment.

Boundary_Converter Function Block

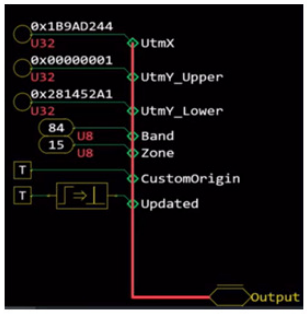


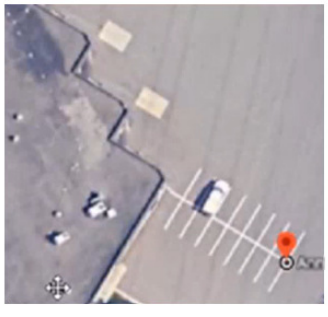
The image above shows the machine's origin and position information coming from **Position_Filter** into **Geofence_Check**, labeled here as **GF_Checking**. **RCT_Rx** refers to joystick buttons programmed to **Boundary_Converter**, labeled here as **Boundary_Converting**. Here, users press the joystick buttons for the boundary information to reset or flow into **Geofence_Check**. However, these could be set pulses or set values. Completed boundary information flows into **Geofence_Check**.



The image above shows inside the **Boundary_Converting** page. The **Origin** and **Boundary_Array** pages reflect what is inside **Origin** and **Boundary_Pts** buses.

1. Add the **Boundary_Converter** and **Geofence_Check** function blocks. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
2. Create a pulse to convert the boundary data to a boundary type data locker. This connects directly to **Convert** to convert the data. Here, a joystick labeled B3 sends a pulse.
3. Create a set pulse for **Reset** on **Boundary_Converter**. If **Reset** and **Convert** are both True in the same program loop, the program only performs the **Reset**. During the test, check the **Boundary_Converter_State** internal signal to make sure the block is not in a read state, which is 1. If it is 1, then the conversion already completed.
4. Connect the **Origin** bus to where the **Origin** function block data comes from. Here, the origin coordinates were already known, so the origin parameters are hard coded. If hard coding parameters, adjust data inside the **Origin** function block.



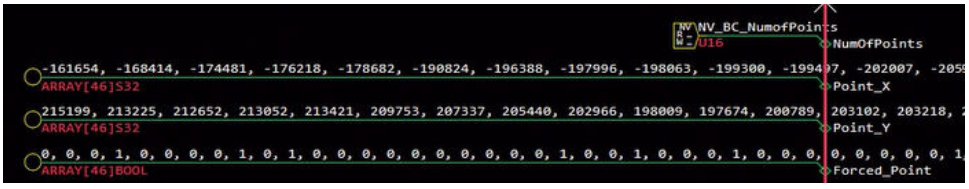


The image to the left shows the location origin data used for the boundary. The right image shows the parking lot location on a map with the origin (red).

5. Go into **Boundary_Pts** to set the boundary coordinates. Ideally, use easy or known boundary shapes, like a square. Negative coordinate points reside to the left and below the origin, and positive coordinate points reside to the right and above the origin. The x-axis runs left to right through the

Boundary_Converter Function Block

origin, and the y-axis runs forward and behind, using the East-North-Up (ENU) convention. Measure from the origin in the environment to each boundary point.



The image above shows an array of coordinates for the boundary. Here, this is inside the **Boundary_Array** page.

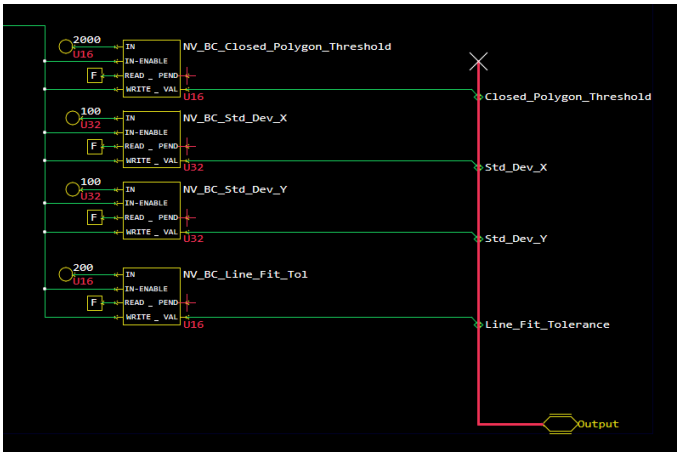
a) Enter the number of coordinate points that will be used to make the boundary in **NumOfPoints**. Here, there are 46 coordinate points, creating an array index of 0-45.

b) Enter the X coordinates in **Point_X**. Here, the first X coordinate is -161654, the second X coordinate -168414. These go into array index 0 and 1. This means the boundary begins 161654 mm to the left of the origin. Ideally create the same first and last point, so the 46th X coordinate would also be -161654.

c) Enter the Y coordinates in **Point_Y**. Here, the first Y coordinate is 215199, the second Y coordinate 213225. These go into array index 0 and 1. This means the boundary begins 215199 mm in front of the origin. Ideally create the same first and last point, so the 46th Y coordinate would also be 215199.

d) Enter any points required in the boundary into **Forced_Point**. These are not removed during the filtering process. By default, all points provided in the array are forced contingent on the **Line_Fit_Tolerance**. Points inside the boundary shape are deleted and not included in the final boundary. Enter 1 in the array index for any coordinate to keep. Here, the fourth boundary point is forced into the boundary.

6. Fill in the parameters page.



The image above shows parameter data, which is reflected in a new **Params** page with nonvolatile components. Nonvolatile components allow changing the values while the application runs. Use fixed values instead of the nonvolatile components if the application requires it.

Boundary_Converter Function Block

a) Optionally set the application name, date, and time. This manually entered data can represent anything. Here, these values were skipped.
b) Enter the distance required between the first and last coordinate to consider the boundary shape closed into **Closed_Polygon_Threshold**. Here, that is 2000 mm.
c) Enter the standard deviations for X and Y to account for a noisy GNSS. Putting in a 50 for X will mean that the value could be good within a 50 mm radius. Here, these are 100 mm.
d) Enter the **Line_Fit_Tolerance**, which is the maximum distance between points and the projected line during the filtering process to create the boundary. Here, it is 200 mm between points to include them in the boundary shape. Any points farther away are deleted.

7. Monitor the **Finished** flag to see that the array was converted into the boundary. Here, the finished signal is **BC_Finished**. Optionally, see each function blocks' *Pre-Made Service Tool Screens* on page 25. Information automatically flows into a boundary type data locker for **Geofence_Check** to use.
8. Plot the graph using any plotting tools, such as Excel, of the X and Y coordinates to verify that the boundary from the raw data points entered is what was expected compared to the filtered boundary after processing. Look at the **Boundary_Extract** function block connected to **Boundary_Converter** to see the filtered points.

Inputs

The following table describes inputs required for the **Boundary_Converter** function block.

Array range for X in the ARRAY[X] types should be between 3 to 32,767. X is dynamic.

Item	Type	Range	Description [Unit]
Convert	BOOL	T/F	False to True transition starts the conversion of boundary arrays to a boundary type data locker. T: The block state changes back to idle and checks for errors. It checks whether the boundary is closed, and if so, **Boundary_Converter** goes from an idle to a read state. F: If there is a boundary in the boundary type data locker, then the block keeps updating the data locker. If there is no boundary in the data locker, then the block stays in an idle state. If **Reset** and **Convert** are both True in the same program loop, the program only performs the **Reset**.
Reset	BOOL	T/F	False to True transition determines whether to clear the data locker that has been written. Reset will not take effect during the read state. T: The block state changes back to idle, and no more data goes into a boundary type data locker. F: If there is a boundary in the boundary type data locker, then the block keeps updating the data locker. If there is no boundary in the data locker, then the block stays in an idle state.
Origin	BUS	——	BUS containing UTM values of the boundary's origin. The data flows automatically to the boundary type data locker. The items in this bus are placeholders and do not do anything.
UtmX	U32	$0\text{-}10^9$	The UTM Easting (X) value of the origin. [mm]
UtmY	U32	$0\text{-}10^{10}$	The UTM Northing (Y) value of the origin. This uses two U32 types, equivalent to a U64. [mm]
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value. [mm]
UtmY_Lower	U32	0x00000000-0x540BE400	The 32 least significant bits of UtmY as stored in a U64 value. This is the range of the full U64 bit number. [mm]
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.

Boundary_Converter Function Block

Item	Type	Range	Description [Unit]
Boundary_Pts	BUS	——	A bus that contains ways to define the boundary.
NumOfPoints	U16	3-32767	The desired number of points to be written into the data locker. This takes affect when **Convert** transitions from False to True.
Point_X	ARRAY[X]S32	-2147483648-2147483647	Array of boundary coordinates along the x-axis with respect to the origin. Values to the left and below the origin are negative. Values to the right and above the origin are positive. Manually enter X point values. The boundary will include them, with the exception of inner loops. [mm]
Point_Y	ARRAY[X]S32	-2147483648-2147483647	Array of boundary coordinates along the y-axis with respect to the origin. Values to the left and below the origin are negative. Values to the right and above the origin are positive. Manually enter Y point values. The boundary will include them, with the exception of inner loops. [mm]
Forced_Point	ARRAY[X]BOOL	0-1	Forces a point to be included on the boundary. Forced boundary points are in the line fitting algorithm, which determines the final boundary shape. Points inside the polygon boundary shape are deleted and not included in the line fitting algorithm. 0: Does not force the boundary to include specific points. 1: Forces the boundary point in a specific location.

Parameters

The following table describes parameters for the **Boundary_Converter** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Metadata	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less. Default: 'Application Name'
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. This manually entered timestamp can represent anything necessary for the application. Leave blank if unneeded. Default: 1999/01/01 00:00
Std_Dev_X	U32	1-4294967295	The highest standard deviation of boundary points along the x-axis. Smaller numbers indicate a more precise boundary. Default: 1 [mm]
Std_Dev_Y	U32	1-4294967295	The highest standard deviation of boundary points along the y-axis. Smaller numbers indicate a more precise boundary. Default: 1 [mm]
Closed_Polygon_Threshold	U16	50-65535	The maximum distance allowed between the first and last points to consider the boundary shape closed. Boundaries with larger distance values are incomplete. This value uses raw data points rather than filtered points from post-processing. Default: 1000 [mm]
Line_Fit_Tolerance	U16	0-1000	Maximum distance between points to create a computationally more efficient boundary. If the distance between the point and the fitted line is smaller than the tolerance value, the point is discarded. Default: 0 [mm]

Boundary_Converter Function Block

Outputs

The following table describes outputs required for the **Boundary_Converter** function block. The data could go into the **Boundary_Extract** and **Geofence_Check** function blocks.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Boundary	S8	-1-99	The ID of the boundary type data locker.
Finished	BOOL	T/F	Indicates if the boundary has finished loading into the data locker. T: Boundary finished loading. F: Boundary has not finished loading.

Internal Signals

The following table describes what is happening internally in the **Boundary_Converter** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Boundary_Converter_Err_Specific	U16	0x0000 to 0x003F	Indicates when a specific error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Boundary_Converter Troubleshooting* on page 49.
Boundary_Converter_Err_Common	U16	0x0000 to 0x0007	Indicates when a generic error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Troubleshooting Common Errors* on page 36.
Boundary_Converter_State	U8	0-3	The state of the **Boundary_Converter** function block. 0: Idle, waiting for the convert signal. 1: Read the input and write it to the boundary type data locker. Filtering boundary data occurs during this post-processing state. 2: Increase the **Boundary** data locker sequence ID. 3: Error state, waiting for the reset command. The **Boundary** output data locker ID will be -1.
Num_Points	U16	0-65535	Number of points in the boundary after filtering data. Look at this value to see if too many points are excluded and no data is processing.
Progress	U16	0-10000	Indicates the progress converting the boundary in the data locker. Look here to see if **Boundary_Converter** stopped processing data. [0.01%]

Boundary_Converter Troubleshooting

The following table describes errors that could occur in the **Boundary_Converter** function block and ways to fix them.

View the **Boundary_Converter_Err_Specific** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

See *Troubleshooting Common Errors* on page 36 to fix errors that appear in many function blocks.

Boundary_Converter Function Block

If the PLUS+1® application does not compile, times out, or stalls, it may be due to using large constant arrays of points. Try disabling the **Checkpoints** page for **Boundary_Converter**.

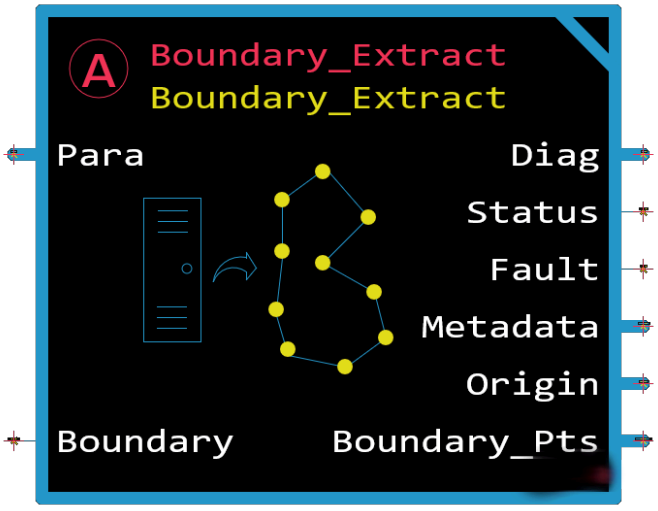
These errors can be visually represented in the bit, hexadecimal, and decimal versions in the service tool.

Boundary_Converter_Err_Specific Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	Input array sizes are not equal in length. Array size mismatch is checked before receiving the **Convert** signal.	Check that array inputs have the same length.
Bit 1	The signal **NumOfPoints** is not equal to the input arrays, creating the wrong number of points. This value size is checked before receiving the **Convert** signal.	Verify the size of input arrays are equal to the expected **NumOfPoints** value.
Bit 2	Less than three boundary points exist after post-processing. A boundary needs more than three points to be created.	Modify the **Line_Fitting** parameter and check the boundary points.
Bit 3	The distance between the first and last boundary points is larger than the **Closed_Polygon_Threshold** value.	Increase the **Closed_Polygon_Threshold** value or shorten the distance between the first and last points.
Bit 4	There is no outer boundary loop.	Check the boundary points.
Bit 5	The data locker cannot be updated.	Use the **Convert** signal again. Restart the controller. Check that there is enough memory.

Boundary_Extract Function Block

The **Boundary_Extract** function block reads boundary information from a data locker and displays up to 50 points of data about the boundary.



This block requires a license for A+ Advanced.

Boundary_Extract shows data from a data locker, which could be visually seen on a service tool screen. Use the pre-made service tool screen, or pull the signals into a service tool individually. The data extracted could be used in other blocks or parts of the application. Additionally, create a display of the information on a piece of hardware such as the DM1000.

After pulling data from the **Boundary** data locker ID, **Boundary_Extract** focuses on more specific point data. Choose a point in the array by writing the array number in **Point_Index**. For example, entering 0 shows information related to the first 50 points, with index values 0-49. To view more than 50 points, add extra **Boundary_Extract** function blocks to the application and increase **Point_Index** to 50 for the second instance to see values 50-99, 100 for the third instance to see values 100-149, and so on. Outputs include origin, position, and point information gathered from earlier blocks.

The **Updated** flag stays true as long as there is valid data in the data locker. It does not change if new data is available. It only goes false if invalid or no data comes, indicated by -1 in the boundary type data locker.

Place **Boundary_Extract** after gathering boundary information, which could be from **Boundary_Recorder**, **Boundary_Converter** or **Boundary_Loader** function blocks. Confirm a **Data_Lockers** block exists somewhere in the application.

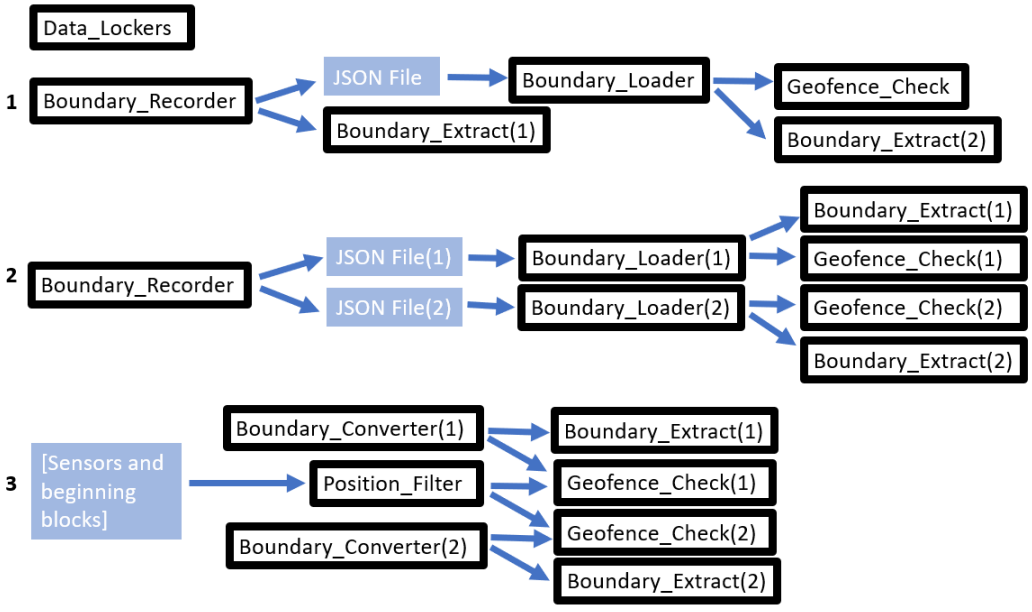
Out of the boundary blocks, **Boundary_Extract** helps but is not required to create a boundary.

Application Information

Common function blocks that work with **Boundary_Extract** are **Boundary_Converter**, **Boundary_Loader**, **Boundary_Recorder**, and **Data_Lockers**.

The **Boundary_Extract** function block visually displays data about the boundary on a service tool screen or a hardware display, such as the DM1000. It is helpful but not necessary in an application. Some basic boundary function block combinations include:

Boundary_Extract Function Block



These scenarios assume code earlier in the application establishes a machine's position. One **Data_Lockers** block is required in all applications and does not need to connect to anything.

1. Scenario one shows **Boundary_Recorder** recording a boundary, then writing it to a JSON file. Additionally, **Boundary_Extract(1)** displays information from **Boundary_Recorder** onto a Service Tool screen or other hardware display. **Boundary_Loader** reads the boundary from the JSON file and transmits it to a Service Tool screen or a hardware display using the **Boundary_Extract(2)** function block. Lastly, **Geofence_Check** determines whether a machine crosses the boundary.
2. Scenario two shows **Boundary_Recorder** recording two separate boundaries and writing each one to a JSON file. Then, **Boundary_Loader(1)** reads the first boundary from a JSON file and transmits it to a Service Tool screen or a hardware display using the **Boundary_Extract(1)** function block. **Boundary_Loader(2)** and **Boundary_Extract(2)** do the same with the second boundary. **Geofence_Check(1)** determines whether the machine crosses the first boundary, and **Geofence_Check(2)** determines whether it crosses the second boundary.
3. Scenario three shows two boundaries, represented by two **Boundary_Converter** blocks. Each boundary requires a **Boundary_Extract** block to display the filtered boundary data on a Service Tool screen or hardware display, which could differ from the raw boundary data. **Position_Filter** determines the machine's position and passes that information into each **Geofence_Check**. **Geofence_Check(1)** determines whether the machine crosses the first boundary, and **Geofence_Check(2)** determines whether it crosses the second boundary.

Additionally, place the **Boundary_Extract** function block:

- With one **Data_Lockers** block, version 1.12 or later, which can be on any page in the application.
- After any boundary function block to visually see what information those blocks are putting into a data locker. Many **Boundary_Extract** blocks could exist in an application. If multiple **Boundary_Loader** and **Boundary_Converter** function blocks exist, place **Boundary_Extract** after each one to see what is in each block.
- Do not place **Boundary_Extract** at the start of the boundary block flow because there will not be any data to see. It is not used after **Geofence_Check**.

Example

The **Boundary_Extract** function block is in examples with other boundary blocks.

See *Example* on page 66 with the **Boundary_Recorder**, **Boundary_Loader**, and **Geofence_Check** function blocks.

Boundary_Extract Function Block

Inputs

The following table describes inputs required for the **Boundary_Extract** function block. This data comes from a data locker with boundary information from the **Boundary_Loader**, **Boundary_Recorder**, or **Geofence_Check** function blocks.

Item	Type	Range	Description [Unit]
Boundary	S8	-1-99	The ID of the boundary type data locker.

Parameters

The following table describes parameters required for the **Boundary_Extract** function block.

Item	Type	Range	Description [Unit]
Point_Index	U16	0-65535	The array index of the first point read from the boundary. For example, to start the array index at point 15, enter 15 here. Add more **Boundary_Extract** function blocks into the application if there are more than 50 points to view, and set **Point_Index** to 50 or larger. The first boundary point will be in array value 0. Default: 0

Outputs

The following table describes outputs required for the **Boundary_Extract** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Origin	BUS	——	Stores the boundary's origin in the form of the UTM coordinate system.
UtmX	U32	$0\text{-}10^9$	The UTM Easting (X) value of the origin. [mm]
UtmY	U32	$0\text{-}10^{10}$	The UTM Northing (Y) value of the origin. This uses two U32 types, equivalent to a U64. [mm]
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value. [mm]
UtmY_Lower	U32	0x00000000-0x540BE400	The 32 least significant bits of UtmY as stored in a U64 value. The range represents the full U64 bit number. [mm]
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.
Metadata	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less.
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. This manually entered date and time information comes from **Boundary_Recorder** or **Boundary_Converter** function blocks.

Boundary_Extract Function Block

Item	Type	Range	Description [Unit]
Boundary_Pts	BUS	——	A bus that contains ways to define the boundary.
Updated	BOOL	T/F	Indicates if new boundary data is extracted from the **Boundary** input. T: New boundary data is extracted. This stays True even if the **Boundary** data changes, as long as it is still valid. F: No new boundary data available.
NumOfPoints	U16	0-50	The number of valid points in the output array. Add more **Boundary_Extract** function blocks into the application if there are more than 50 points to view, and set **Point_Index** to 50 or larger. The first boundary point will be in array value 0.
Point_X	ARRAY[50]S32	-2147483648-2147483647	Array of boundary coordinates along the x-axis with respect to the origin. Values to the left and below the origin are negative. Values to the right and above the origin are positive. [mm]
Point_Y	ARRAY[50]S32	-2147483648-2147483647	Array of boundary coordinates along the y-axis with respect to the origin. Values to the left and below the origin are negative. Values to the right and above the origin are positive. [mm]
Forced_Point	ARRAY[50]BOOL	0-1	Indicates if a point is forced in a boundary. 0: Point is not forced. 1: Point is forced.
Std_Dev_X	U32	1-4294967295	The highest standard deviation of boundary points along the x-axis. Smaller numbers indicate a more precise boundary. [mm]
Std_Dev_Y	U32	1-4294967295	The highest standard deviation of boundary points along the y-axis. Smaller numbers indicate a more precise boundary. [mm]
Closed_Polygon_Threshold	U16	0-65535	The maximum distance allowed between the first and last points to consider the boundary shape closed. Boundaries with larger distance values are incomplete. [mm]

Internal Signals

The following table describes what is happening internally in the **Boundary_Extract** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Boundary_Extract_Err_Specific	U16	0x0000 to 0x0003	Indicates when a specific error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Boundary_Extract Troubleshooting* on page 54.
Total_Num_Points	U16	0-65535	Total number of boundary points stored inside of the data locker after the post-filtering process. Use this value to determine where to start the **Point_Index** parameter.

Boundary_Extract Troubleshooting

The following table describes errors that could occur in the **Boundary_Extract** function block and ways to fix them. View the **Boundary_Extract_Err_Specific** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

These errors can be visually represented in the bit, hexadecimal, and decimal versions in the service tool.

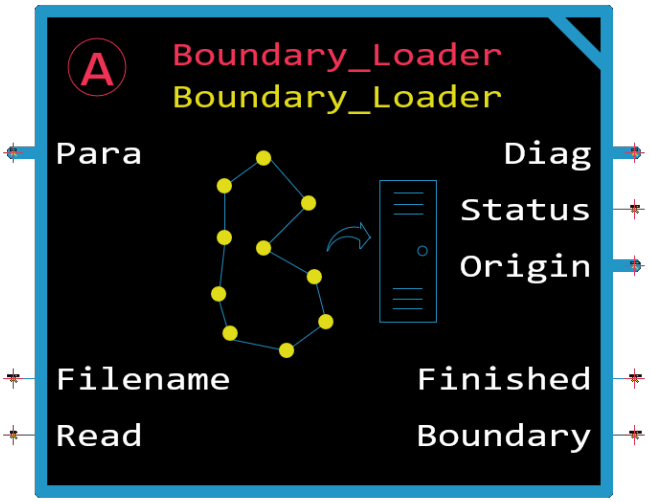
Boundary_Extract Function Block

Boundary_Extract_Err_Specific Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	The string in the metadata is longer than expected and invalid. The **Data_Lockers** block does not have enough room for the metadata.	Confirm each metadata string is less than 255 characters.
Bit 1	The **Point_Index** value is greater or equal in length to the boundary, making the index invalid.	Enter a valid **Point_Index** signal within the index range, which should be smaller than the desired boundary length.

Boundary_Loader Function Block

The **Boundary_Loader** function block reads a JSON file, which contains information about a boundary around a geographic area. This boundary is then uploaded into the **Data_Lockers** block for other blocks to access.



This block requires a license for A+ Advanced. It also requires hardware compatible with the media file system, such as the XM100. The minimum HWD version must be greater than 3.21.

Boundary_Loader requires a JSON file either produced by the **Boundary_Recorder** function block or user written. It reads the data from a JSON file, checks that the data is not corrupt, and then loads it to **Data_Lockers** for other blocks in the application to use.

The JSON file contains information about the filtered boundary recorded by **Boundary_Recorder**, rather than any raw data points gathered in its initial boundary creation. Or, the JSON file could contain raw data points written by a user if **Boundary_Recorder** was not used. When using **Boundary_Recorder**, the JSON's filtered boundary includes processing the raw data points, smoothing the raw boundary, and closing the boundary. If a JSON file needs modification, make any changes to it and save a new MD5. Then, use the new JSON file in **Boundary_Loader**. See *Modify JSON and Update MD5* on page 27.

If the JSON is modified, change the parameters in **Boundary_Loader** from the default. These parameters include **Closed_Polygon_Threshold**, which ensures the boundary is closed with the filtered boundary points from the modified JSON. Measure the distance between the first and last point in the boundary to ensure it is smaller than the **Closed_Polygon_Threshold**. Use **Boundary_Extract** to verify that the boundary closed.

Line_Fit_Tolerance further smooths the boundary by using the filtered points if the tolerance is a smaller value than in **Boundary_Recorder**. Smoothing the boundary here could be done without modifying the JSON file.

Boundary_Loader outputs the data contained in the JSON file into a boundary type data locker. Outputs include any errors that occurred, origin used during boundary creation, and if the boundary has loaded into **Data_Lockers** on the **Finished** output. Review errors on the **Checkpoints** page or service tool screen.

If a machine needs to move inside one boundary but outside of another boundary, create multiple boundaries. For example, this could be staying inside a large boundary like a field, but staying outside of a smaller boundary like a pond within the field. Each boundary needs a **Boundary_Loader** function block to upload the boundary data to its own data locker within the **Data_Lockers** block.

After uploading boundary data, the **Geofence_Check** function block uses the information to determine if a machine passes through the boundary. Each boundary needs a **Geofence_Check**.

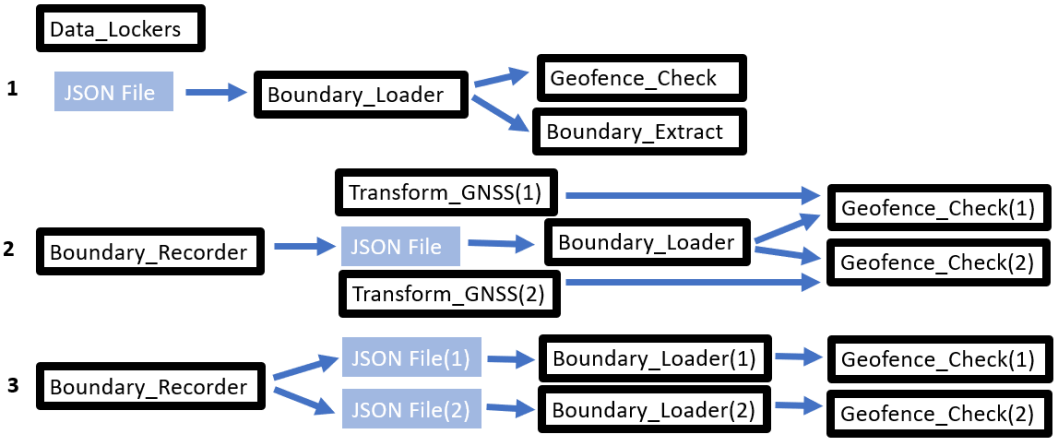
Use the **Boundary_Extract** function block to verify that something was loaded properly into the data locker. If there is an incorrect boundary, then the boundary will not load successfully. Use less than 25000 points in the boundary.

Boundary_Loader Function Block

Application Information

Common function blocks that work with **Boundary_Loader** are **Boundary_Extract**, **Boundary_Recorder**, **Data_Lockers**, and **Geofence_Check**.

The **Boundary_Loader** function block uploads the JSON file with boundary data gathered from the **Boundary_Recorder** function block into a **Data_Lockers** block, that could then be consumed by other blocks. One **Data_Lockers** block is required for all applications but does not connect to anything. Information about the position of the machine is required earlier in the application. Some basic boundary function block combinations are the following:



1. Scenario one shows **Boundary_Loader** reading the boundary from a JSON file that had been created a previous time. **Boundary_Loader** loads the JSON file information to a data locker, and then **Geofence_Check** uses the boundary data to detect whether the machine crosses the boundary. Additionally, the boundary data is transmitted to a Service Tool screen or a hardware display using **Boundary_Extract**.
2. Scenario two shows **Boundary_Recorder** recording the boundary to a JSON file, which is then stored on the controller. **Boundary_Loader** loads the JSON file information to a data locker for **Geofence_Check** to use. The **Transform_GNSS** function blocks move coordinate information to two different parts of the machine. Each **Geofence_Check** function block checks when that part of the machine crosses a boundary.
3. Scenario three shows **Boundary_Recorder** recording two separate boundaries and writing each one to a JSON file. Then, **Boundary_Loader(1)** reads the first boundary from a JSON file and transmits it to a Service Tool screen or a hardware display using the **Boundary_Extract(1)** function block. **Boundary_Loader(2)** and **Boundary_Extract(2)** do the same with the second boundary. **Geofence_Check(1)** determines whether the machine crosses the first boundary, and **Geofence_Check(2)** determines whether it crosses the second boundary.

Additionally, place the **Boundary_Loader** function block:

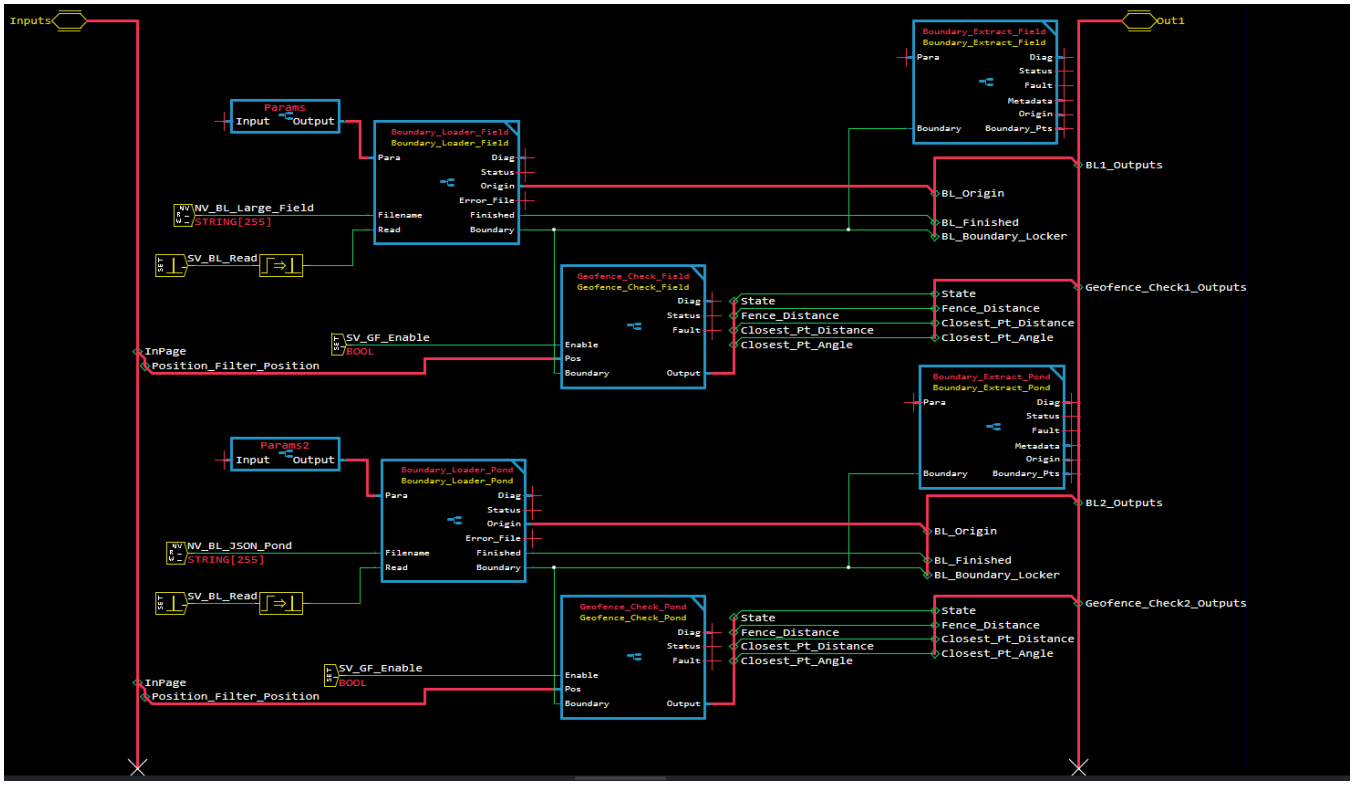
- With one **Data_Lockers** block, version 1.12 or later, which can be on any page in the application.
- After **Boundary_Recorder** if it is required to create the JSON file. If a JSON file exists from a different application, **Boundary_Recorder** is not required.
- After position information is obtained, such as after a **Position_Filter** function block.
- Multiple times in an application if there are many boundaries. There should be a **Boundary_Loader** block for each boundary, and a machine needs a new boundary for each region it stays inside or outside.

Example

The example shows loading two pre-recorded JSON files of two boundaries. A small machine needs to stay outside of one boundary around a pond, but stay inside a larger field boundary surrounding the pond.

Set up GNSS and positioning code earlier in the application, which could use the **Position Filter** function block.

Boundary_Loader Function Block



The example assumes code exists earlier in the application to establish a machine's position and origin.

1. Add two **Boundary_Loader** blocks. Each **Boundary_Loader** block corresponds to a boundary, which affects where a machine can go. Here, the machine stays inside of the field with **Boundary_Loader_Field** and outside of the pond with **Boundary_Loader_Pond**.
2. Add two **Geofence_Check** function blocks. **Geofence_Check_Field** determines when a machine crosses the field boundary, and **Geofence_Check_Pond** determines when a machine crosses the pond boundary.
3. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page, and there should only be one **Data_Lockers** in an application.
4. Look up the JSON file names in the XM100. See *Getting Files from XM100* on page 29. Here, the JSON file names are Field.json and Pond.json located in path '/media/p1user/' .
5. Connect one JSON's file path to the signal **Filename**. Here, the entire file name and path is '/media/p1user/Field.json' for the field boundary. **Boundary_Recorder** generates the JSON file, which is stored on a local file directory such as the XM100.
6. Connect the other JSON's file path to the signal **Filename** in the second **Boundary_Loader**. Here, the file name and path '/media/p1user/Pond.json' refers to the pond boundary.
7. Create pulse signals to **Read** the JSON files for both boundaries. This pulse triggers **Boundary_Loader** to read the file and write the information to a data locker. The blocks load the information they read into a data locker automatically if the **Data_Lockers** block exists somewhere in the application.

Boundary_Loader Function Block

8. Set up the parameters in each **Boundary_Loader**.
   a) Enter the distance required to close the field boundary in **Closed_Polygon_Threshold**. This applies to the filtered boundary points from the JSON file. Here, that is 3000 mm, which means if the first and last filtered boundary point are within 3000 mm, the boundary closes. If the distance between those points are larger, then an error occurs.
   b) Enter the value to further smooth the field boundary, if necessary, in **Line_Fit_Tolerance**. Here, this was left at 0.
   c) Enter the distance required to close the pond boundary in **Closed_Polygon_Threshold**. Here, that is 1000 mm because it is smaller than the large field.
   d) Enter the value to further smooth the pond boundary, if necessary, in **Line_Fit_Tolerance**. Here, this was left at 0.
9. Optionally, connect the **Origin** bus to code further downstream for both **Boundary_Loader** blocks. This origin is obtained from the JSON file.
10. Optionally, connect the **Finished** output from each **Boundary_Loader** to any downstream code. Monitor the service tool screen to see that the boundaries loaded, which should show 1 if successful.
11. Connect each **Boundary** output into a **Boundary_Extract** function block to display the filtered boundary data more clearly on a service tool screen. Here, the field boundary connects into **Boundary_Extract_Field** and the pond boundary connects to **Boundary_Extract_Pond**.
12. Connect each **Boundary** output into a **Geofence_Check**. Here, the field boundary connects to **Geofence_Check_Field**. The pond boundary connects to **Geofence_Check_Pond**.
13. Connect the machine position information into the **Pos** input on each **Geofence_Check**. This verifies where the machine is with respect to the boundary.
14. Create a set value for **Enable** on each **Geofence_Check**. Setting **Enable** to True activates the block.
15. Move the machine around the boundary to validate the **State** output coming from the **Geofence_Check** blocks. Monitor the service tool screens to see that the machine state is not 0 or 255, which means the block is deactivated or an error occurred.
16. Optionally, see the *Pre-Made Service Tool Screens* on page 25 for all function blocks for comprehensive results. Or, create a display of the information on a piece of hardware such as the DM1000.
17. Create code downstream to have the machine react when it crosses the boundaries. Here, the machine would need to stay inside the field but outside the pond boundary.
18. Log and plot the outputs of the **Geofence_Check** blocks, the boundaries of the field and pond, and the machine position onto a separate graph, such as in Excel. Verify the information is as expected.

Inputs

The following table describes inputs required for the **Boundary_Loader** function block. The JSON file data comes from the **Boundary_Recorder** block or could be written manually.

Item	Type	Range	Description [Unit]
Filename	STRING[255]	——	Name of the JSON file, which is made manually or from **Boundary_Recorder**. The default name is '/media/p1user/Recorded_Boundary.json'. The file must be within 'media/p1user'.
Read	BOOL	T/F	False to True transition starts reading the JSON file and writing the information into a boundary type data locker. T: Read the file. F: Do not read the file. For optimal performance, have **Read** pulse false after reading the JSON file.

Boundary_Loader Function Block

Parameters

The following table describes the parameters for the **Boundary_Loader** function block.

Item	Type	Range	Description [Unit]
Closed_Polygon_Threshold	U16	50-65535	The maximum distance allowed between the first and last points to consider the boundary shape closed. Boundaries with larger distance values are incomplete. For more accurate position information, this value uses filtered points from post-processing rather than raw data. Default: 2000 [mm]
Line_Fit_Tolerance	U16	0-1000	Maximum distance between points to create a computationally more efficient boundary. If the distance between the point and the fitted line is smaller than the tolerance value, the point is discarded. Default: 0 [mm]

Outputs

The following table describes outputs for the **Boundary_Loader** function block. The data could go into **Boundary_Extract** or **Geofence_Check** function blocks.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Origin	BUS	——	BUS containing UTM values of the boundary's origin, which are read from the JSON file.
UtmX	U32	$0\text{-}10^{9}$	The UTM Easting (X) value of the origin. [mm]
UtmY	U32	$0\text{-}10^{10}$	The UTM Northing (Y) value of the origin. This uses two U32 types, equivalent to a U64.
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	0x00000000-0x540BE400	The 32 least significant bits of UtmY as stored in a U64 value. This is the range of the full U64 bit number.
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.
Updated	BOOL	T/F	Indicates when new data is being stored for the origin. T: New data is available for the origin. F: No new data is available.
Finished	BOOL	T/F	Indicates if the boundary has finished loading into the data locker. Finished flag will always be 0 if **Read** transitions from False to True, there is an active Status, Boundary_Loader_Err_Specific or Boundary_Loader_Err_Common code. T: Boundary finished loading. F: Boundary has not finished loading.
Boundary	S8	-1-99	The ID of the boundary type data locker.

Internal Signals

The following table describes what is happening internally in the **Boundary_Loader** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Boundary_Loader Function Block

Item	Type	Range	Description [Unit]
Boundary_Loader_Err_Specific	U16	0x0000 to 0x0FFF	Indicates when a specific error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Boundary_Loader Troubleshooting* on page 61.
Boundary_Loader_Err_Common	U16	0x0000 to 0x0007	Indicates when a generic error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Troubleshooting Common Errors* on page 36.
Progress	U16	0-10000	Indicates the progress loading the boundary in the data locker. Look here to see if **Boundary_Loader** has stopped processing data. [0.01%]

Boundary_Loader Troubleshooting

The following table describes errors that could occur in the **Boundary_Loader** function block and ways to fix them.

View the **Boundary_Loader_Err_Specific** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

See *Troubleshooting Common Errors* on page 36 to fix errors that appear in many function blocks.

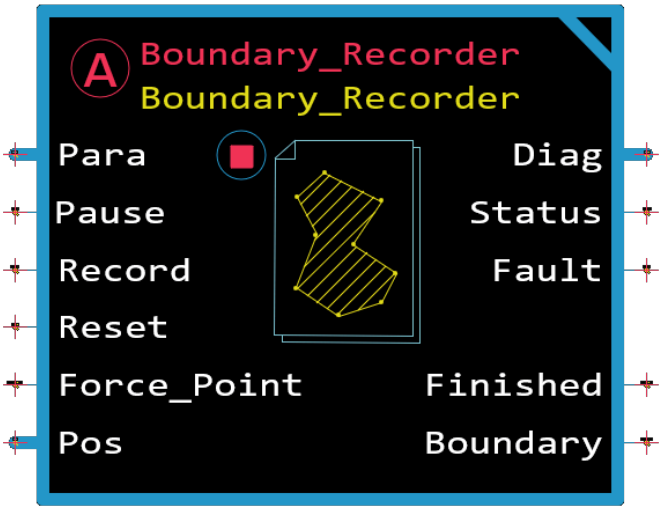
These errors can be visually represented in the bit, hexadecimal, and decimal versions in the service tool.

Boundary_Loader_Err_Specific Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	The JSON file is not available. The file could have the wrong name or not exist.	Verify the JSON file name is correct. Check that the block looks for it in the correct location, such as the XM100 or a USB connected to the XM100.
Bit 1	There may not be enough memory available to read the JSON file, or the file is too large. This could happen if the file was manually edited.	Take information out of the file.
Bit 2	The JSON file could not be read because it is empty or incomplete.	Verify the file has not been manually modified, and it contains the recorded boundary instead of wrong information.
Bit 3	The Cyclical Redundancy Check (CRC) failed. The JSON file is corrupted or manually modified.	Make sure the JSON file has not been modified. If creating a manual JSON file, create a valid 32-bit MD5 check for the file.
Bit 4	Wrong JSON data or incorrect JSON file.	Verify the JSON file is valid, not manually modified, and contains the recorded boundary.
Bit 5	There are less than three boundary points in the JSON file. There needs to be at least three points to create a boundary.	Check the information stored in the JSON file contains at least three points and re-record the boundary.
Bit 6	Wrong or missing metadata. This happens after manually creating or modifying a JSON file.	Check that all the metadata fields are present in the JSON file.
Bit 7	The number of points expected is different than the number recorded. This happens after manually modifying the file.	Re-record the boundary.
Bit 8	Standard deviations for X and Y are missing in the JSON file.	Check that all the standard deviation fields are present in the JSON file.
Bit 9	The distance between the starting and ending boundary point is bigger than **Closed_Polygon_Threshold**.	Increase the **Closed_Polygon_Threshold** value or make the boundary smaller than the **Closed_Polygon_Threshold**.
Bit 10	Errors occurred during the post-filtering process while determining the boundary points.	Make sure there are less than 65,535 points in the boundary. Review the boundary points if they are manually written.
Bit 11	Cannot update the output data locker.	Reset the hardware controller.

Boundary_Recorder Function Block

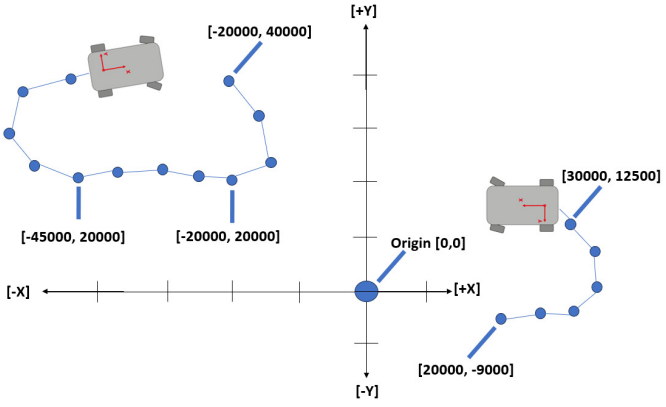
The **Boundary_Recorder** function block records a virtual boundary around a geographic region, and then stores the boundary data in a JSON file and **Data_Lockers**. Other code determines if a machine should stay inside or outside of the boundary.



This block requires a license for A+ Advanced. It also requires hardware compatible with the media file system, such as the XM100. The minimum HWD version must be greater than 3.21.

Set up origin location information earlier in the code before creating the boundary. Additionally, set up machine dimension information earlier in the code. This helps determine when parts of the machine cross the boundary later on. Only one **Boundary_Recorder** is allowed in an application for it to compile.

Boundary_Recorder records a virtual boundary, which can be any sort of closed shape. It plots a series of points which contain the boundary's global coordinates, XY position, and optionally other data. Then, this information goes to a JSON file to be used immediately or later.

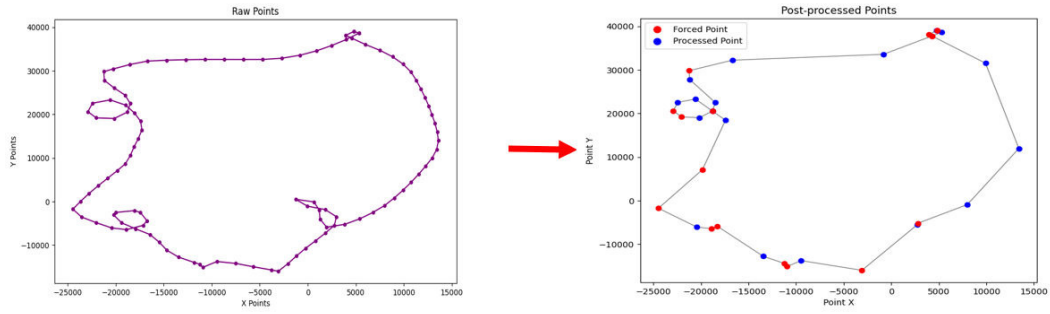


The image shows creating two distinct boundaries. The XY boundary coordinates relate to an origin in the environment, not necessarily the machine's XY coordinates or origin. For example, one boundary point reads -45000 mm to the left of the origin along the x-axis, and 20000 mm above the origin along the y-axis. Straight lines automatically connect between the points.

Create boundaries around areas a machine should stay inside or outside, for example staying inside of a field or outside of water. Boundaries can be any polygon shape, but create multiple boundaries if a machine needs to stay in one area but outside of another. Inner loops within the boundary are eliminated, so only the outer shell stays to form the boundary.

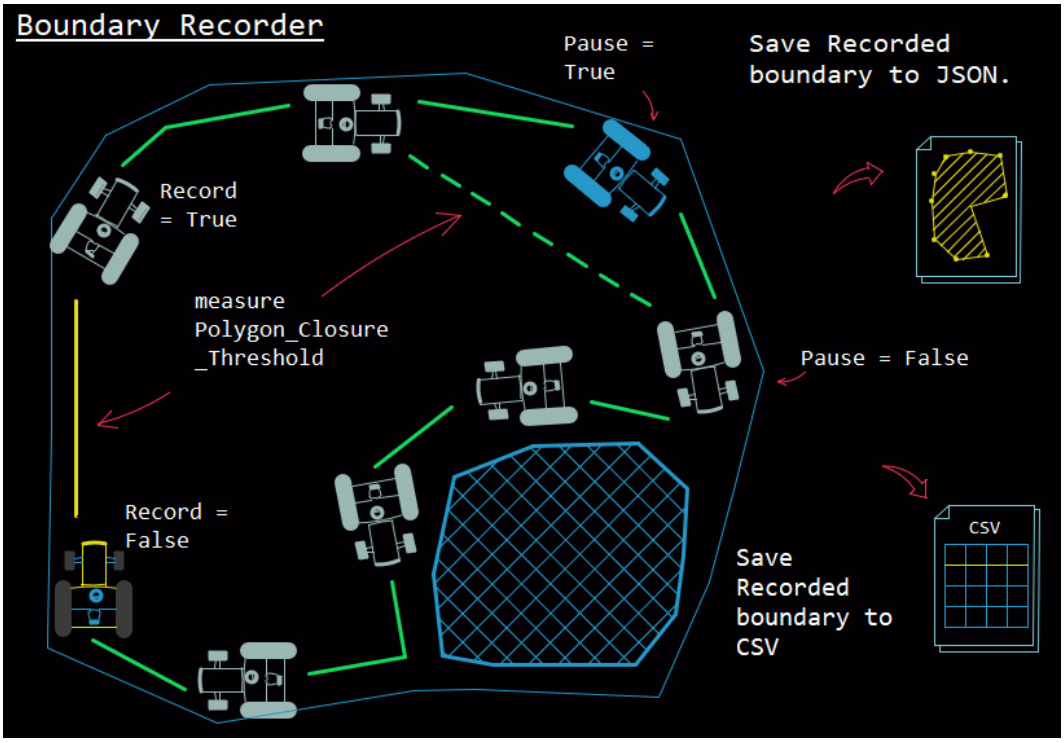
There is the ability to include certain areas into the boundary recording unless they are part of an inner loop, so they are not entirely filtered out, which is set in **Force_Point**. Optionally, more information can link to the points during the boundary recording. These include the application name, JSON file name, date, and time.

Boundary_Recorder Function Block



The image above shows inner loops inside the boundary eliminated and external loops kept. Only the forced points and outer boundary shell remain as the final boundary. Further processing occurred to eliminate some of the raw points in the right image.

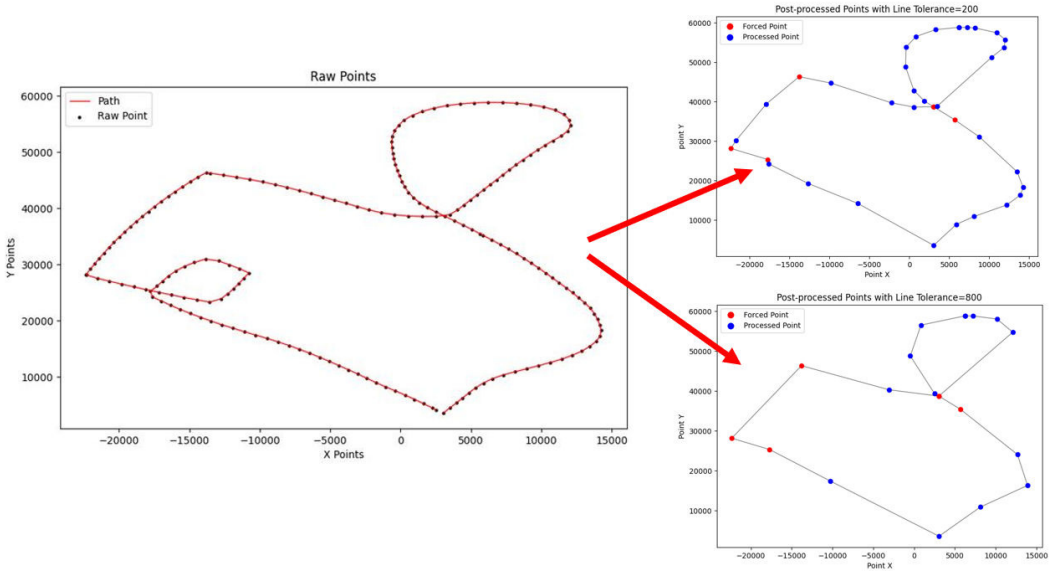
Boundary_Recorder includes options to pause, reset, and record the virtual boundary. Pausing stops gathering coordinate point data for the boundary. Transitioning **Pause** to True creates a new point, and transitioning to False creates another new point. **Reset** deletes the recorded boundary. **Record** must cycle from True to False to True afterward to start re-recording the boundary. It is not necessary to end the boundary exactly where it started. If the first and last points are close, or at least smaller than the **Polygon_Closure_Threshold**, then the boundary automatically closes.



The image above shows a machine recording a boundary when **Record** is True. The machine pauses, and the coordinate points do not factor into the boundary until it stops pausing. The boundary closes between the pauses and end of the recording if the coordinate points are withing the **Polygon_Closure_Threshold** value. The finished boundary saves to JSON and CSV files.

Additionally, the distance between each point, and whether points are too far away from the boundary line, factor into the filtration process. Distance thresholds for these are set in **Min_Movement** and **Line_Fit_Tolerance**. After recording, a filtering process creates a smooth, closed boundary.

Boundary_Recorder Function Block



The left image shows a boundary with raw data points. The right images show the same boundary with a small line fit tolerance (top) and larger line fit tolerance (bottom) after the filtration process. The areas where the loops intersect create an extra boundary point.

Raw data from the boundary recording saves in a CSV file but not the JSON. The processed boundary data goes into the data locker and JSON file. That information is accessed by **Boundary_Loader** to go into **Geofence_Check**.

The **Boundary_Recorder** function block only supports Linux-based controllers. The recording stops and deletes the raw data file if there are less than 100 megabytes of memory left on the controller. Use less than 25000 points in the boundary, and keep the machine still while the block processes the recording. If the XM100 loses power, the boundary is lost. See *Restart or Resume Recording After ECU Power Loss* on page 29.

If a JSON file needs modification, make any changes to it and save a new MD5. Then, use the new JSON file in **Boundary_Loader**. See *Modify JSON and Update MD5* on page 27.

Some items to note:

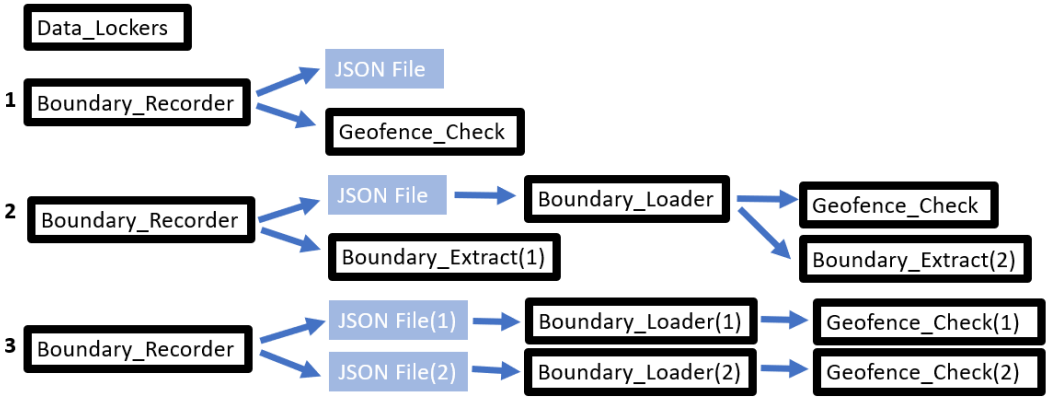
- When setting up an application, ensure the **Origin** function block is set up to the correct location of the boundary and check the GNSS. If an origin is reused from other applications, it may have unexpected coordinates.
- Use the **Boundary_Extract** function block to validate the boundary.
- It is recommended to use the **Position_Filter** function block to estimate the position of the machine.
- Check the values from **Position_Filter**, as well as the other sensors, during and after recording a boundary. Take into account wheel slippage and noise. If there is too much wheel slippage, do not use wheel odometry in the position filter.
- Monitor the standard deviation to check that it is within an allowable tolerance.
- After a recording completes, go into the JSON file and plot out the boundary in another program to check that the expected boundary appears.
- Ensure the boundary is correct in terms of size, forced points, and pauses.

Application Information

Common function blocks that work with **Boundary_Recorder** are **Boundary_Extract**, **Boundary_Loader**, **Data_Lockers**, and **Geofence_Check**.

Boundary_Recorder records boundary data by manually driving a machine along a boundary to record, rather than entering data into the function block. Information about the position of the machine is required earlier in the application. Some basic boundary function block combinations include:

Boundary_Recorder Function Block



These flows assume the machine position was established earlier in the code.

1. Scenario one shows **Boundary_Recorder** producing a JSON file without other boundary blocks to use it. Do this to save and copy a boundary to another machine to use later. Boundary information also saves in boundary type data locker. One **Data_Lockers** block is required in all applications and does not need to connect to anything. Information also flows directly into **Geofence_Check** to use immediately as the boundary records. Shutting the controller off loses the boundary data.
2. Scenario two shows **Boundary_Recorder** recording a boundary, then writing it to a JSON file. Additionally, **Boundary_Extract(1)** displays information from **Boundary_Recorder** onto a Service Tool screen or other hardware display. **Boundary_Loader** reads the boundary from the JSON file and transmits it to a Service Tool screen or a hardware display using the **Boundary_Extract(2)** function block. Lastly, **Geofence_Check** determines whether a machine crosses the boundary.
3. Scenario three shows **Boundary_Recorder** recording two separate boundaries and writing each one to a JSON file. Then, **Boundary_Loader(1)** reads the first boundary from a JSON file and transmits it to a Service Tool screen or a hardware display using the **Boundary_Extract(1)** function block. **Boundary_Loader(2)** and **Boundary_Extract(2)** do the same with the second boundary. **Geofence_Check(1)** determines whether the machine crosses the first boundary, and **Geofence_Check(2)** determines whether it crosses the second boundary.

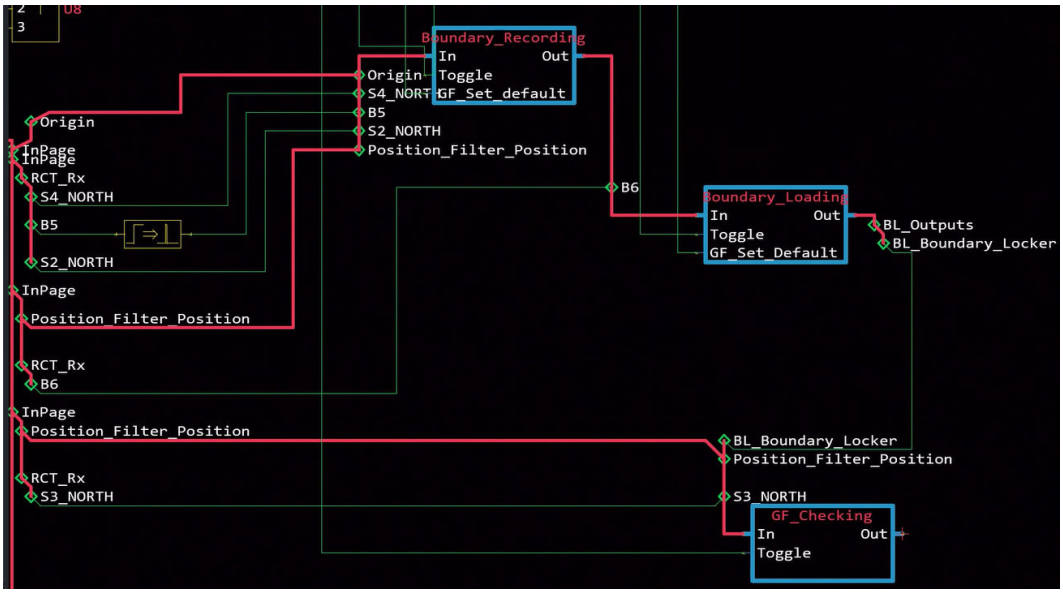
Additionally, place the **Boundary_Recorder** function block:

- Only once in each application, even though many boundaries can be recorded from the one block.
- With one **Data_Lockers** block, version 1.12 or later, which can be on any page in the application.
- After position information is obtained, such as after a **Position_Filter** function block.

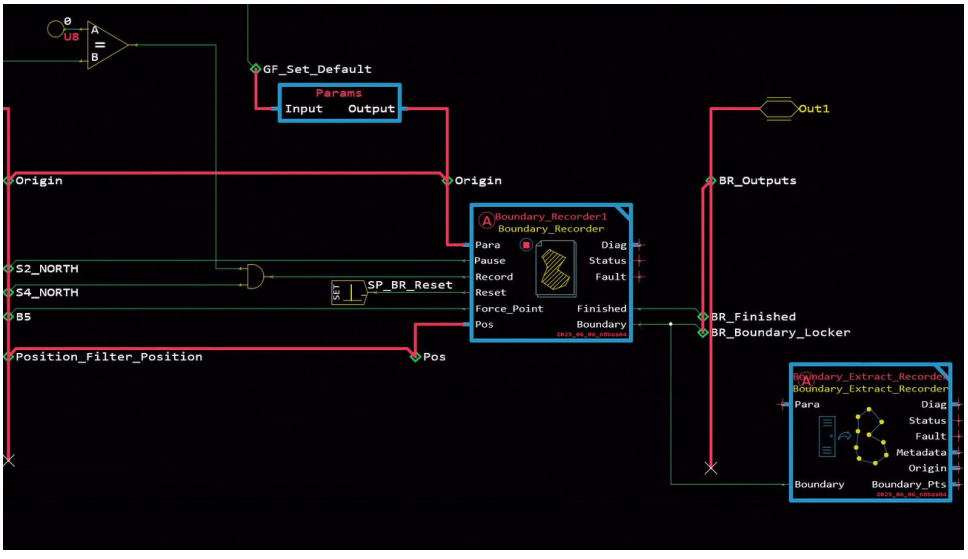
Boundary_Recorder Function Block

Example

The example shows the **Boundary_Recorder** function block recording a virtual boundary, known as a geofence, around a parking lot. **Boundary_Recorder** produces a JSON file of the boundary information which **Boundary_Loader** loads into a data locker. The boundary keeps a machine inside the parking lot.



The image above shows code related to recording and using a boundary. The origin information of the boundary goes into **Boundary_Recorder**, situated in this example inside the **Boundary_Recording** page. **RCT_Rx** refers to joystick buttons programmed to **Boundary_Recorder**. Here, users press the joystick buttons for the boundary information to reset or flow into **Geofence_Check**, situated inside the **GF_Checking** page. However, these could be set pulses or set values. Completed boundary information flows into **Boundary_Loader** and then **Geofence_Check**. The machine's origin and position information comes from **Position_Filter** into **Geofence_Check**.

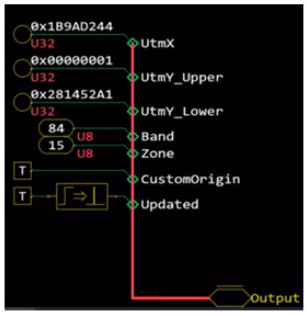


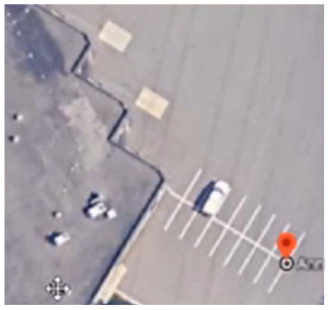
The image above shows inside the **Boundary_Recording** page.

1. Add the **Boundary_Recorder**, **Boundary_Loader**, and **Geofence_Check** function blocks. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.

Boundary_Recorder Function Block

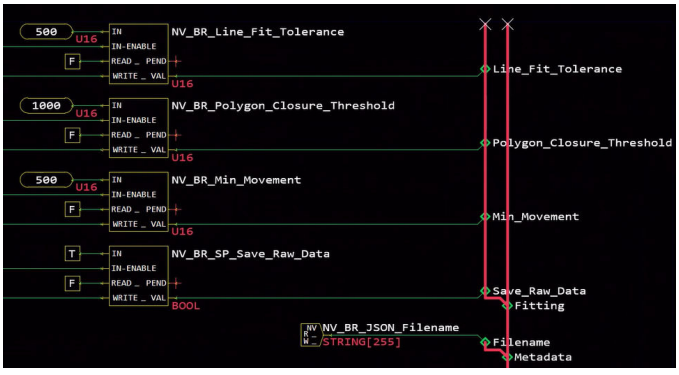
2. Connect the **Origin** bus to where the **Origin** function block data comes from.





The image to the left shows the location origin data used for the boundary, which is hard-coded in the **Origin** block in this example. The right image shows the parking lot location on a map with the origin (red).

3. Create code to pause the recording, such as connecting a set value. Here, a joystick button pauses the recording when pressed. The pause ensures no data records while **Pause** is True.
4. Set **Record** to True for the entire recording. The machine must be manually driven while the recording occurs. In this example, a joystick button is tied to the **Record** signal.
5. Create a set pulse for **Reset**. **Reset** deletes a recorded boundary. **Record** must cycle from True to False to True afterward to start re-recording the boundary.
6. Create a set pulse for **Force_Point**. A forced point means a particular boundary coordinate point will be included in the final processed boundary. If a point is not forced, that coordinate could be filtered out of the boundary. Here, **Force_Point** is linked to a joystick button which forced the point when pressed.
7. Connect **Pos** to the machine's origin and position information, recommended from the **Position_Filter** function block.
8. Fill in the parameters page.



The image above shows parameter data, which is reflected in a new **Params** page with nonvolatile components. Nonvolatile components allow changing the values while the application runs. Use fixed values instead of the nonvolatile components if the application requires it.

a) Optionally set the application name, date, and time. This manually entered data can represent anything. Here, these values were skipped.
b) Enter the **Line_Fit_Tolerance**, which is the maximum distance between raw points and the projected line during the filtering process to create the boundary. Here, it is 500 mm between points to include them in the boundary shape. Any points farther away are deleted.
c) Enter the distance required between the first and last raw point to consider the boundary shape closed into **Polygon_Closure_Threshold**. This includes the distance between pauses. Here, that is 1000 mm.

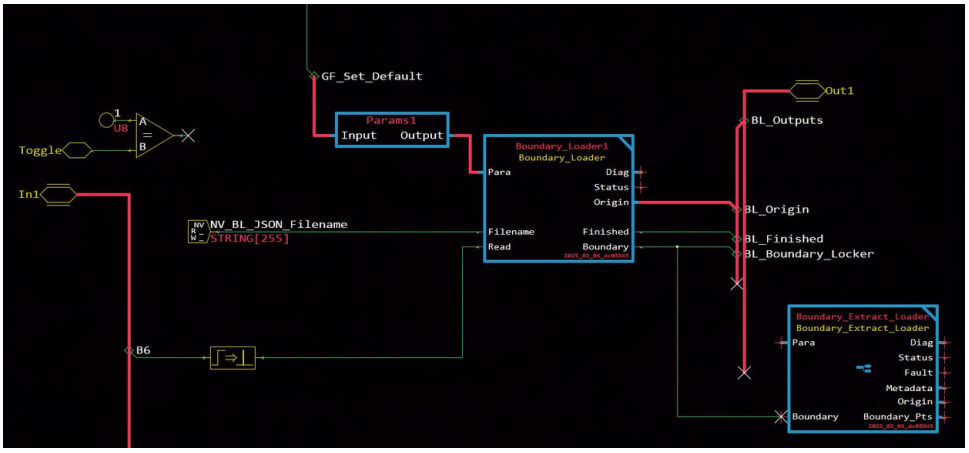
Boundary_Recorder Function Block

d) Enter the **Min_Movement**, which is the minimum distance a machine must move in order to record the next raw boundary point. Here, that is 500, so a point records after every 500 mm. Larger distances mean less points connect.

e) Determine whether to save the raw CSV file in **Save_Raw_Data**. If a reset occurs, the raw data is saved if this is set to True but deletes if it is False. The JSON is also not saved if a reset occurs.

f) Fill out the JSON file name. Here, that is **NV_BR_JSON_Filename**.

9. Connect the **Finished** and **Boundary** buses to **Boundary_Loader**. Boundary data automatically flows into a boundary type data locker via a JSON file, which **Boundary_Loader** uses. To change anything in the JSON file, see *Modify JSON and Update MD5* on page 27.
10. Optionally, connect the **Boundary_Extract** function block to the **Boundary** output to visually see the data inside **Boundary_Recorder**. The first 50 boundary points appear by default in point index 0-49.
11. Fill out **Boundary_Loader** information. This is situated inside the **Boundary_Loading** page.



The image above shows inside the **Boundary_Loading** page.

a) Create a pulse to read the filename that was set in **Boundary_Recorder**. If the filename is not present, an error occurs. Here, the file name is **NV_BL_JSON_Filename**.

b) Send a pulse to the **Read** input to read the particular JSON from the data locker. Here, the joystick button labeled **B6** sends a pulse when pressed.

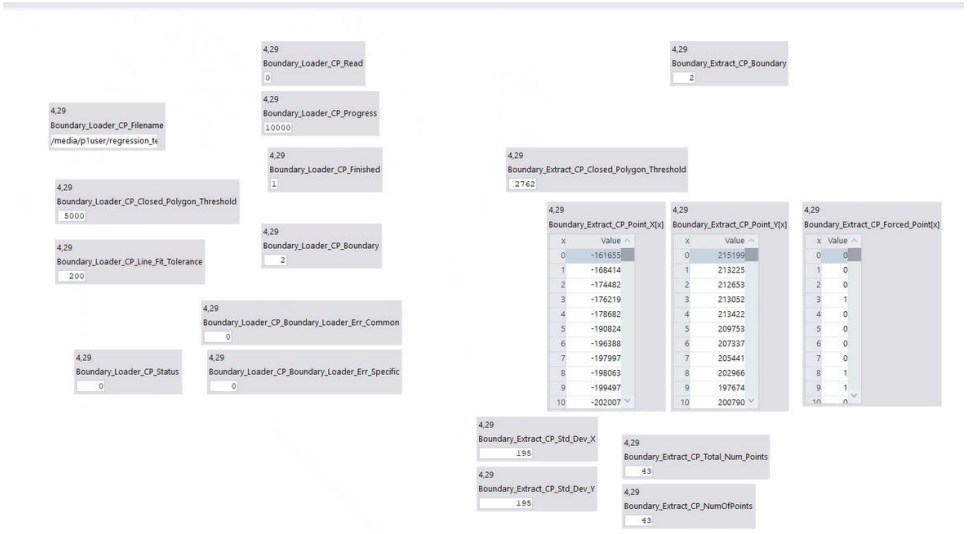
c) Enter the distance required between the first and last point to consider the boundary shape closed into **Closed_Polygon_Threshold**. These points refer to the filtered points after reading the JSON and not the raw data points in the original recording. Here, that is 2000 mm.

d) Enter the **Line_Fit_Tolerance**, which is the maximum distance between points and the projected line during the filtering process to create the boundary. Here, it is 500 mm between points to include them in the boundary shape. Any points farther away are deleted.

e) Connect the **Finished** and **Boundary** buses to **Geofence_Check**. Boundary data automatically flows into a boundary type data locker, which **Geofence_Check** uses.

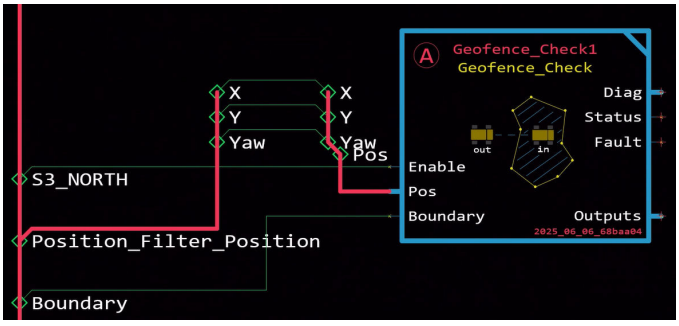
f) Optionally, connect the **Boundary_Extract** function block to the **Boundary** output to visually see the data inside **Boundary_Loader**. The first 50 boundary points appear by default in point index 0-49.

Boundary_Recorder Function Block



The image above displays a Service Tool screen which used two **Boundary_Extract** blocks, one after **Boundary_Recorder** and the other after **Boundary_Loader**. Loader information displays on the left, with recording information on the right. Standard deviation of 195 is a combination of the position filter and GNSS, and means X could be 195 mm to the left or right in a circle radius of the origin. Here, the space between the first and last point was too large to close the boundary, as represented in **Closed_Polygon_Threshold**, so the parameter was increased.

12. Fill out **Geofence_Check**.



The image above shows **Geofence_Check** inside of the **GF_Checking** page.

a) Set **Enable** to True. Here, pressing a joystick button labeled **S3_North** enables **Geofence_Check** to check if a machine is inside or outside of the boundary.

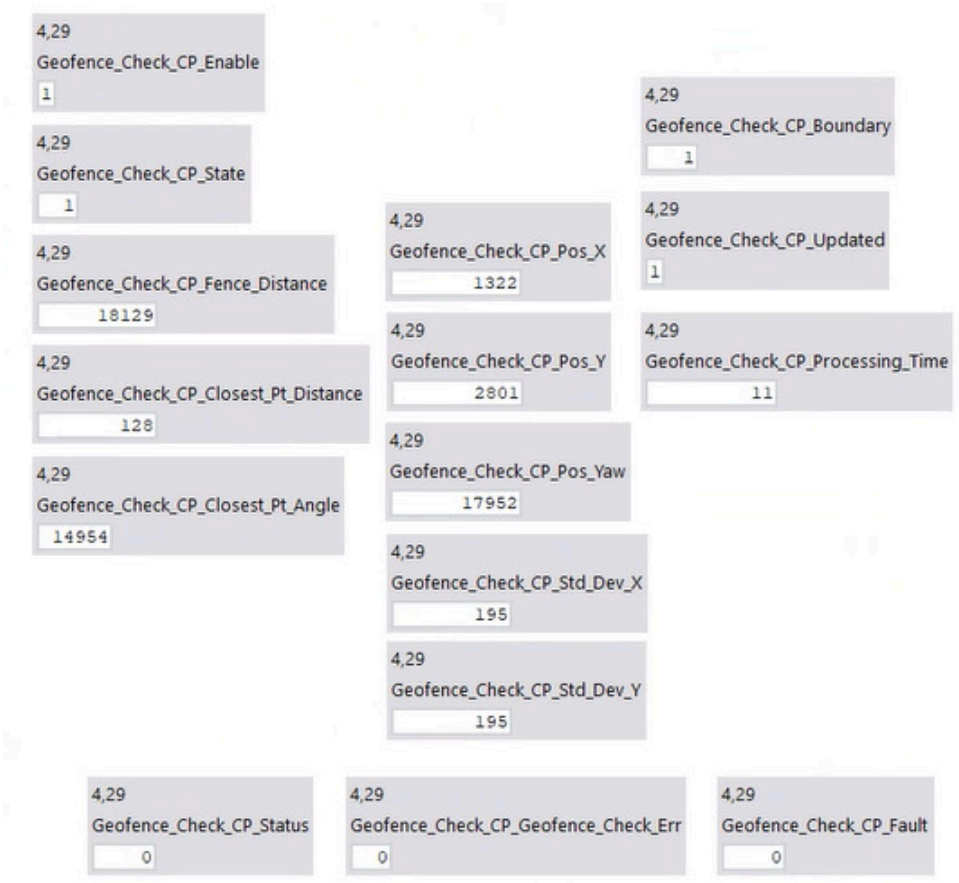
b) Connect machine position information into the **Pos** bus to give the machine's X, Y, and yaw information. This usually comes from **Position_Filter**.

c) Connect the boundary locker from **Boundary_Loader**. **Geofence_Check** automatically pulls information about the boundary from the data locker.

13. Drive the machine around the boundary area to create the boundary.

Boundary_Recorder Function Block

14. Check the service tool screen to see that the correct boundary loads. Do not move the machine while the boundary loads.



The image above shows a service tool screen with **Geofence_Check** data.

15. Plot the graph using any plotting tools, such as Excel, of the X and Y coordinates to verify that the expected boundary appears.

Inputs

The following table describes inputs required for the **Boundary_Recorder** function block. Most of this data comes from the **Position_Filter** block and optionally a wheel odometer.

Item	Type	Range	Description [Unit]
Pause	BOOL	T/F	Determines whether to pause the recording. This only applies while recording the raw data. Transition between True to False or False to True will record the point and forces it. T: Pauses the recording. It resumes when transitioning from True to False. F: Records.
Record	BOOL	T/F	Triggers the recording and saves the recorded data into a data locker. When the **Record** signal goes from True to False, it stops the recording and starts saving the data to a data locker. T: Records. F: Waits to record or saves the boundary data. Changing to false in the middle of the recording stops the recording but processes the data already gathered.
Reset	BOOL	T/F	False to True transition determines whether to stop the recording. When **Save_Raw_Data** is True, the recording stops after writing the recorded boundary data to a data locker and deletes the data. T: Stops the recording. F: Signal has no effect on the block functionality.

Boundary_Recorder Function Block

Item	Type	Range	Description [Unit]
Force_Point	BOOL	T/F	Forces a point to be included on the boundary. Forced boundary points are in the line fitting algorithm, which determines the final boundary shape. Points inside the polygon boundary shape are deleted and not included in the line fitting algorithm. T: Forces the boundary point in a specific location. The point information is stored in the final post-processed boundary. F: Does not force the boundary to include specific points.
Pos	BUS	——	Position and coordinate signals coming from the **Position_Filter** function block.
X	S32	-2147483648-2147 483647	Current X position of the machine location. [mm]
Y	S32	-2147483648-2147 483647	Current Y position of the machine location. [mm]
Std_Dev_X	U32	1-4294967295	The highest standard deviation of boundary points along the x-axis. Smaller numbers indicate a more precise boundary. [mm]
Std_Dev_Y	U32	1-4294967295	The highest standard deviation of boundary points along the y-axis. Smaller numbers indicate a more precise boundary. [mm]

Parameters

The following table describes parameters for the **Boundary_Recorder** function block. All these parameters can be hard-coded. However, the **Origin** bus could pull data from the **Origin** function block directly.

Item	Type	Range	Description [Unit]
Origin	BUS	——	Stores the machine's origin in the form of the UTM coordinate system.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. Default: 0x20EBC948 [mm]
UtmY	U32	0-10^{10}	The UTM Northing (Y) value of the origin. This uses two U32 types, equivalent to a U64. Default: 0x1540BE400 [mm]
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value. [mm]
UtmY_Lower	U32	0x00000000-0x54 0BE400	The 32 least significant bits of UtmY as stored in a U64 value. The range represents the full U64 bit number. [mm]
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters. Default: 85
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in. Default: 32
Updated	BOOL	T/F	Indicates when new data is being stored for the origin. T: New data is available for the origin. F: No new data is available.
Fitting	BUS	——	BUS containing additional information about creating the boundary.
Min_Movement	U16	50 - **Polygon_Closure _Threshold**	Minimum distance the machine must move in order to record the next boundary point. Default: 500 [mm]

Boundary_Recorder Function Block

Item	Type	Range	Description [Unit]
Polygon_Closure_Threshold	U16	50 - 65535	The maximum distance allowed between the first and last points to consider the boundary shape closed. This also includes the distance between pauses. Boundaries with larger distance values are incomplete. This value uses raw data points rather than filtered points from post-processing. Default: 500 [mm]
Line_Fit_Tolerance	U16	0-1000	Maximum distance between points to create a computationally more efficient boundary. If the distance between the point and the fitted line is smaller than the tolerance value, the point is discarded. Default: 300 [mm]
Save_Raw_Data	BOOL	T/F	Determines whether to save the boundary data. If a reset occurs, data is deleted. T: Save the data in raw_data.csv. F: Delete the raw data.
Metadata	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less. Default: 'Application Name'
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. This manually entered timestamp can represent anything necessary for the application. Leave blank if unneeded. Default: '1999/01/01 00:00'
Filename	STRING[255]	——	Name of the output JSON file. Rename the file, if desired. The default name is '/media/p1user/Recorded_Boundary.json'. The file must be within 'media/p1user'.

Outputs

The following table describes outputs for the **Boundary_Recorder** function block. The data could go into **Boundary_Extract** or **Geofence_Check** function blocks, or the data could sit in a JSON file for later. See *Getting Files from XM100* on page 29.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Boundary	S8	-1-99	The ID of the boundary type data locker.
Finished	BOOL	T/F	Indicates if the boundary has finished loading into the data locker. T: Boundary finished loading. F: Boundary has not finished loading.

Internal Signals

The following table describes what is happening internally in the **Boundary_Recorder** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Boundary_Recorder Function Block

Item	Type	Range	Description [Unit]
Boundary_Recorder_Err_Specific	U16	0x0000 to 0x07FF	Indicates when a specific error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Boundary_Recorder Troubleshooting* on page 73.
Boundary_Recorder_Err_Common	U16	0x0000 to 0x0007	Indicates when a generic error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Troubleshooting Common Errors* on page 36.
Progress	U16	0-10000	Indicates the progress of processing the raw recorded boundary data into a final boundary with filtered data. Look at this value to see if **Boundary_Recorder** has stopped processing data. [0.01%]
Loop_Closure_Dist	U32	0-4294967295	Distance from the current machine position to the initial point where the recording started. [mm]
Prev_Point_Dist	U32	0-4294967295	Distance from the current machine position to the last recorded point. [mm]
Closed_Polygon_Check	BOOL	T/F	Indicates whether the boundary shape can be closed. T: Boundary can be closed. F: Boundary cannot be closed.
Num_Raw_Points	U32	0-4294967295	Number of raw data points stored in the raw_data.csv file. See *Getting Files from XM100* on page 29 to access this data. If **Save_Raw_Data** is True, these data points are before the boundary filtering process occurs. Look at this value to see how many points were excluded from the filtered boundary or to see if there is an option of getting data from other points.
Num_Points	U16	3-65535	Number of points in the boundary after filtering data. Look at this value to see if too many points are excluded and no data is processing.

Boundary_Recorder Troubleshooting

The following table describes errors that could occur in the **Boundary_Recorder** function block and ways to fix them.

View the **Boundary_Recorder_Err_Specific** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

See *Troubleshooting Common Errors* on page 36 to fix errors that appear in many function blocks.

These errors can be visually represented in the bit, hexadecimal, and decimal versions in the service tool.

Boundary_Recorder_Err_Specific Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	Error writing the raw CSV file. The file cannot be opened or the new data cannot append to it. The file could be read-only or not available.	Verify that the CSV file '/media/p1user/raw_data_boundary.csv' is in the correct location and not read-only.
Bit 1	There is less than 100 megabytes of memory left on the controller.	Delete files on the controller's hard drive, '/media/p1user/raw_data_boundary.csv'.
Bit 2	The distance between the True to False transition for a pause is greater than the **Polygon_Closure_Threshold**. Or, the **Loop_Closure_Threshold** is greater than the **Polygon_Closure_Threshold**.	Restart the recording.
Bit 3	The **Num_Raw_Points** value is less than three. There must be at least three points to create a boundary.	Restart the recording.
Bit 4	The program cannot open the raw data file for post-processing or the file does not exist. When the CSV file is not generated after boundary recording, this error appears.	Re-record the boundary.

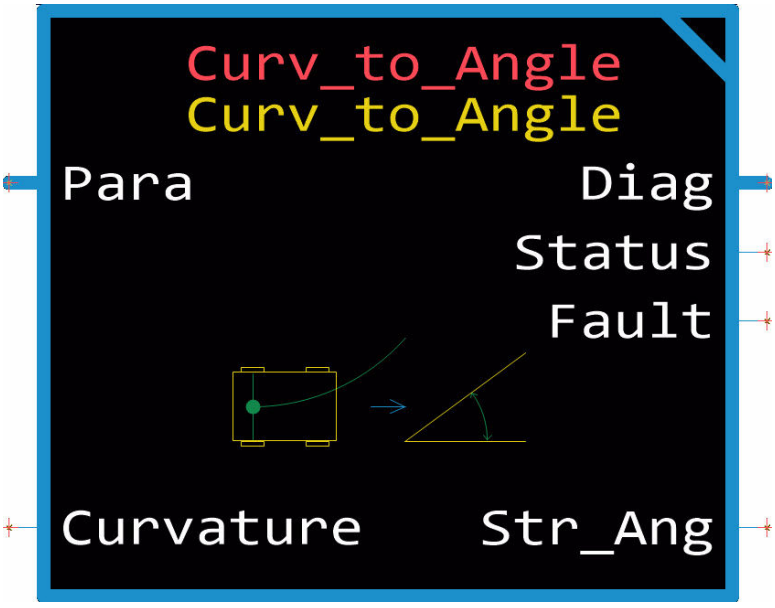
Boundary_Recorder Function Block

Boundary_Recorder_Err_Specific Descriptions and Fixes (continued)

Number	Description	How to Fix
Bit 5	The number of lines in the CSV file differs from the number of lines in the **Boundary_Recorder** function block. The data in the CSV does not match the JSON file.	Re-record the boundary.
Bit 6	Cannot open the JSON file '/media/p1user/Recorded_Boundary.json' for writing.	Check the JSON file location is correct or accessible, or re-record the boundary.
Bit 7	Cannot write some parts of the data to the JSON file. The JSON file might be read-only.	Check that the JSON file is not read-only, or re-record the boundary.
Bit 8	The JSON file size is larger than allowed and runs out of memory. There cannot be more than 1 million characters.	Re-record the boundary with less data points, and make sure the metadata has smaller strings.
Bit 9	The number of points exceeds the limit of 65,535 points.	Re-record the boundary with less points, or create multiple boundaries.
Bit 10	Cannot update the output data locker.	Reset the hardware controller.

Curv_To_Angle Function Block

The **Curv_To_Angle** function block converts curvature to steering angle.



To get the steering angle from the **Curv_To_Angle** function block you provide a curvature input in 0.01/km and a wheelbase parameter in mm.

Inputs

The following table describes inputs for the **Curv_To_Angle** function block.

Item	Type	Range	Description [Unit]
Curvature	S32	-2,147,483,648 to 2,147,483,647	Curvature calculated based on the steering angle and wheelbase of the machine. Negative values are right curves. Positive values are left curves [0.01/km]
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The following table describes parameters for the **Curv_To_Angle** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the Para BUS.
Wheelbase	U16	300 to 20000	The distance between the centers of the front and rear wheels. Default: 5000 [mm]

Curv_To_Angle Function Block

Outputs

The following table describes outputs for the **Curv_To_Angle** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Bus containing diagnostic data for the function block.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range. 0x8100: Invalid ECU.
Fault	U16	——	Reports the fault status of the function block. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Str_Angle	S16	-9000 to 9000	The angle between the front of the machine and the steered wheel direction. Negative values are right curves. Positive values are left curves [0.01 deg]

Data_Lockers Helper Block

The **Data_Lockers** block stores data for other function blocks to use. This enables large amounts of data to pass between function blocks in an efficient way.

The **Data_Lockers** block is used in applications with data such as point clouds or paths, so some Autonomous Control Library (ACL) function blocks may not require it. The data automatically travels from a function block producing data to **Data_Lockers** and then down to a second function block that consumes the data.

The **Data_Lockers** block:

- Does not connect to any buses or wires. Nothing should connect into the small x by the block.
- Does not have inputs, parameters, or outputs.
- Stands alone on a page.
- Can be placed on any page in an application.
- Does not need to be on the same page as other function blocks that use it.
- Should only be placed once in an application. There will never be two or more **Data_Lockers** blocks.
- Does not come with a preassembled service tool screen. It has one checkpoint to drag into a service tool, which is always true and cannot change.

Data_Lockers includes 100 locker IDs ranging from 0-99. The code for an invalid locker ID is -1, which could be used to turn off **Data_Lockers** to save processing power. Each ID is a unique number that references a data stream of a certain type. Different data streams cannot mix, such as a point cloud locker and path locker.

ACL function blocks use specific signal names for locker IDs to indicate their type and where they connect. For example, **U_PtCld** connects to other **U_PtCld**. Just connect one signal to the other, and data flows into the correct data locker automatically. There are no other actions, buses, or wires that need to connect into **Data_Lockers**. However, connecting incorrect signals together that use a data locker will compile in the code but not function. For example, connecting a function block with the signal **Bez_Path_In** to **U_PtCld**.

Execution order matters with function blocks that use or generate **Data_Lockers**. Any function blocks that require information from others to work need to be placed so information flows from a block earlier in the application to a data locker, and blocks later in the application can use that data. Place blocks that generate data in the parent page or top left corner of a page to give information first, and blocks which use data beneath it or to the right.

The most up-to-date **Data_Lockers** version works with all function blocks in an autonomous application. This block uses a different version than ACL and therefore does not match, but any updated ACL blocks likely require the latest **Data_Lockers** version. A compilation error appears for incompatible versions, and the error message says the required version. To replace an old **Data_Lockers** block with a new one, delete the older block's C code. See *Delete the Old Function Block C Code* on page 35.

Data_Lockers Helper Block

If issues occur, exit PLUS+1® GUIDE, reboot the computer, and compile again. Use XM or DM devices.

Example

The **Data_Lockers** function block is in several examples. It stands alone on the page without connecting to other blocks.

See *Example* on page 157 with the **Path_Recorder** function block.

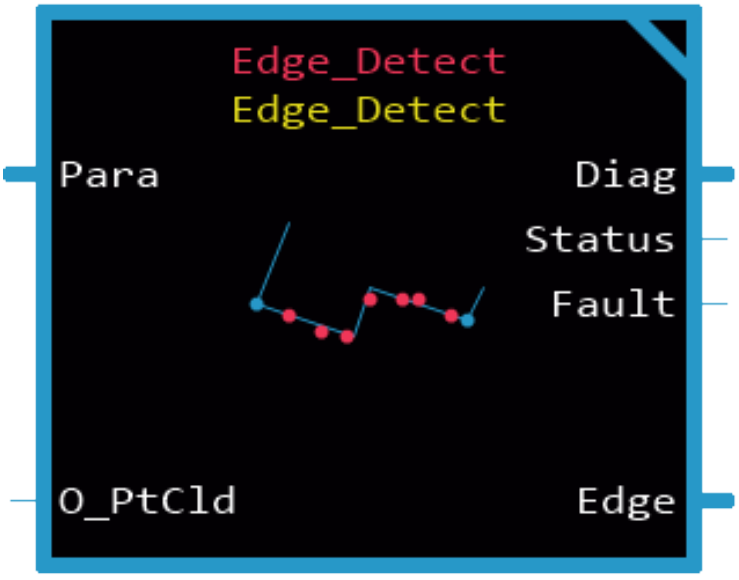
See *Example* on page 128 with the **Path_Converter** function block.

See *Example - One Path* on page 150 with one **Path_Loader** function block.

See *Example - Multiple Paths* on page 151 with multiple **Path_Loader** function blocks.

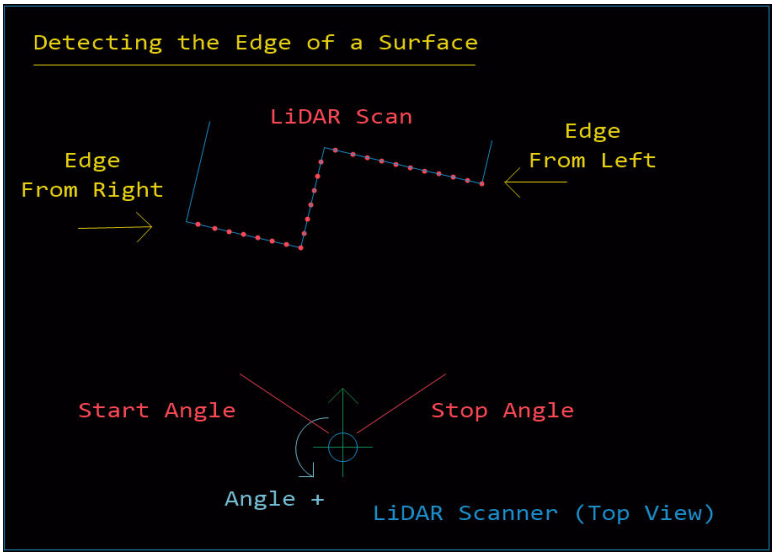
Edge_Detect Function Block

The **Edge_Detect** function block analyzes the ordered point cloud data. The block parses the data to look for a continuous surface followed by an edge, which appears as a discontinuity in the LiDAR scan.



This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about the Ouster LiDAR scanner and block.

The discontinuity in the LiDAR scan can be used to find any generic edge of a smooth feature, such as a wall or garage door. It can parse the scan from either direction, limit the range of the scan and the size of the feature to find more specific features.



Edge_Detect Function Block

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

The following table describes the limitations of the function block.

Item	Description
Parse Direction	The parsing direction is determined by the values chosen for the **Start_Angle** and **Stop_Angle**. If the **Start_Angle** is less than the **Stop_Angle**, the parsing occurs from right to left. If the **Start_Angle** is greater than the **Stop_Angle**, the parsing occurs from left to right.
Falling Edge	The algorithm finds a falling edge in the selected ring from the LiDAR scan, where the points are farther away or disappear. This block is not designed to find an interior corner where the points appear closer.

Inputs

Inputs to the **Edge_Detect** function block are described.

Item	Type	Range	Description
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The **Edge_Detect** function block's operating characteristics are set by **Para** bus input signals.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Start_Angle	S16	-18000-18000	Specifies which beam of the LiDAR scan to use to start parsing for an edge. Default: 0 [0.01 deg]
Stop_Angle	S16	-18000-18000	Specifies which beam of the LiDAR scan to use to stop parsing for an edge. Default: 18000 [0.01 deg]
Min_Feature_Size	U16	1-60000	The minimum required size of an object to be identified as a continuous surface before and after an edge. This filters out noise from the sensor or ignores very small objects. Default: 1000 [mm]
Ring	U16	0-65535	The ring parameter selects the horizontal row from the 3D LiDAR scan to detect the edge. For a 2D LiDAR, set the ring parameter to zero. Default: 0

Edge_Detect Function Block

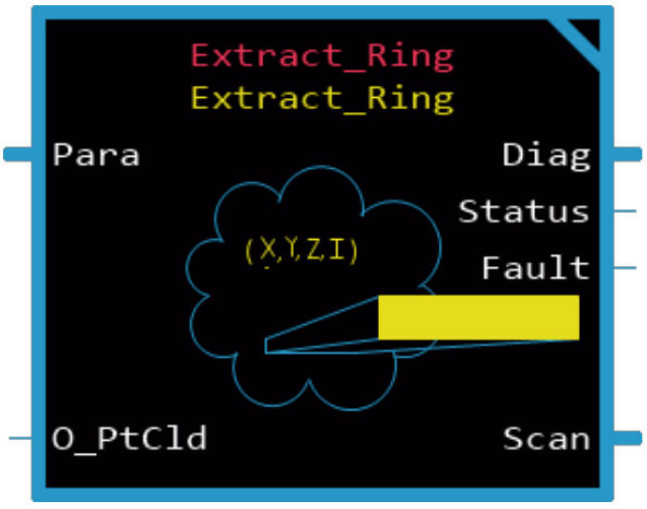
Outputs

Outputs of the **Edge_Detect** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Edge_Index	U32	0-4294967295	Specifies the index of the point in the row of point cloud that was marked as an edge.
Edge_Detect_Err	U8	0-4	Indicates errors occurred in the function block operation. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: Point cloud is unordered.
Processing_Time	U32	0-4294967295	The time taken to process the input point cloud data. [µs]
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Reports the fault status of the function block. Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Edge	BUS	——	The **Edge** bus contains the updated information about the location of the detected edge.
Updated	BOOL	T/F	Indicates new information is available from the block. T: Edge detected. F: Edge not detected.
Distance	U32	0-4294967295	Distance from the edge in radial coordinates. [mm]
Angle	S16	-18000-18000	Angle to the edge in radial coordinates. [0.01 deg]
Edge_X	S32	-100000-100000	X coordinate of the edge relative to the scanner. [mm]
Edge_Y	S32	-100000-100000	Y coordinate of the edge relative to the scanner. [mm]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Extract_Ring Function Block

The **Extract_Ring** function block is used to extract a ring of information from the latest LiDAR point cloud object found inside a data locker.



This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about the Ouster LiDAR scanner and block.

Read about LiDARs and point clouds in *Perception* on page 13.

The parameter **Ring** refers to the ring row from the ground up, starting at zero. For example, a ring input of zero means data is displayed for the first row from the ground. A ring input of 15 means data is displayed for the 16th row.

A value of zero displays on the service tool screen if the point is not valid. For example, this occurs when the LiDAR supplies the intensity signal and it is zero, and the X, Y, and Z values are zero because the point cloud was not transformed.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Inputs

Inputs to the **Extract_Ring** function block are described.

Item	Type	Range	Description [Unit]
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Extract_Ring Function Block

Parameters

Parameters to the **Extract_Ring** function block are described.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Ring	U16	0 - 127	Valid ring number to extract data from point cloud. Default: 0

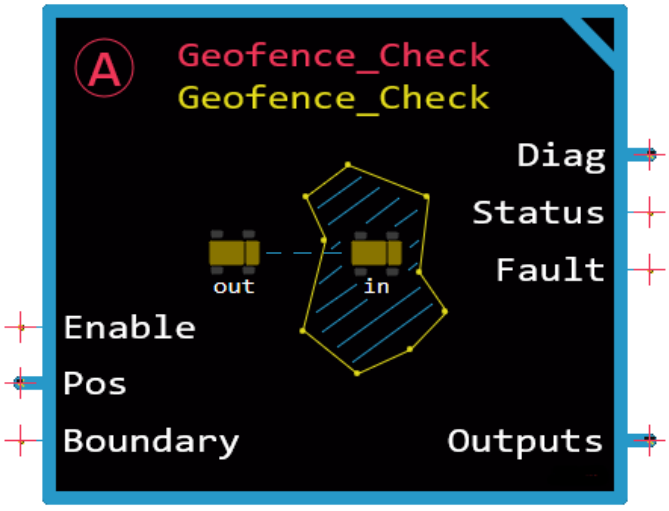
Outputs

Outputs of the **Extract_Ring** are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the sensor.
Extract_Ring_Err	U8	0-2	Indicates that an error occurred in the block functionality. 0: No error. 1: Input point cloud is unordered. 2. The ring parameter is invalid or not available in the point cloud data.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Scan	BUS	——	Outputs the points in the point cloud of a LiDAR with their positions and intensities.
Updated	BOOL	——	True if new data was processed.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.
Num_Points	U16	0-2048	Number of points in the row.
X	(Array[2048]S32)	-2147483648-2147483647	X coordinate of point in Cartesian coordinates. [mm]
Y	(Array[2048]S32)	-2147483648-2147483647	Y coordinate of point in Cartesian coordinates. [mm]
Z	(Array[2048]S32)	-2147483648-2147483647	Z coordinate of point in Cartesian coordinates. [mm]
I	(Array[2048]U16)	0-10000	Intensity values of points.

Geofence_Check Function Block

The **Geofence_Check** function block monitors a machine's position relative to a virtual boundary or geofence.



This block requires a license for A+ Advanced. It also requires hardware compatible with the media file system, such as the XM100. The minimum HWD version must be greater than 3.21.

Geofence_Check uses GNSS, IMU, and odometry data so the machine knows its location. This requires extra hardware, such as antennas, and their accompanying code. Align the machine's coordinate frame and the GNSS antenna's coordinate frame for the distance and angle information to function accurately.

A geofence is a virtual fence or perimeter corresponding to a physical location. When an object enters this area, something happens. The **Geofence_Check** function block:

- Determines whether the machine is inside or outside of the pre-defined geofence.
- Calculates the distance to the geofence, aiding in navigation planning.
- Calculates the angle to the closest point on the boundary, which can be used for steering adjustments.
- Allows flexible and complex boundary shapes, defined by a series of points on a map.
- Ensures responsive real-time operation by using threading without blocking the main program.

However, **Geofence_Check** only works in 2D and does not use height information. It does not provide path planning, integrate directly with GPS or other positioning systems, and does not predict future machine positions or provide warnings about approaching the geofence boundary. If needing any of these features, use the outputs of **Geofence_Check** to customize the application.

A geofence needs at least three coordinate points on a map to form a closed boundary. These create a polygon shape, and the points cannot be in a straight line. Geofences vary in size: small to as large as the whole earth. Because **Geofence_Check** uses a GNSS signal, use it with outdoor applications.

Geofence_Check comes after a series of blocks to establish the geofence boundary, and then **Geofence_Check** checks the machine position with respect to the boundary.

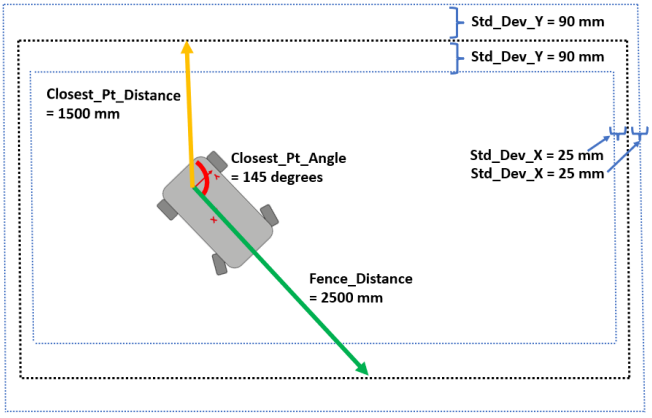
When **Geofence_Check** is enabled, the block pulls information from the data locker about the boundary using the **Boundary** input. The machine's X and Y coordinate information comes from the **Position_Filter** function block earlier in the code, which is normally measuring the machine's origin or steering point. This uses GNSS to locate the machine on a world map at all times, so the code detects when the machine crosses the boundary. Use **Transform_GNSS** to transfer the coordinates to corners of the machine for **Geofence_Check** to monitor when those areas cross the boundary. Information about which direction the machine faces is estimated by the **Yaw** input, which also comes through **Position_Filter**.

The **Updated** signal indicates if all the information comes into **Geofence_Check** without issues and processes successfully. **Geofence_Check** outputs include the machine state, which says what the

Geofence_Check Function Block

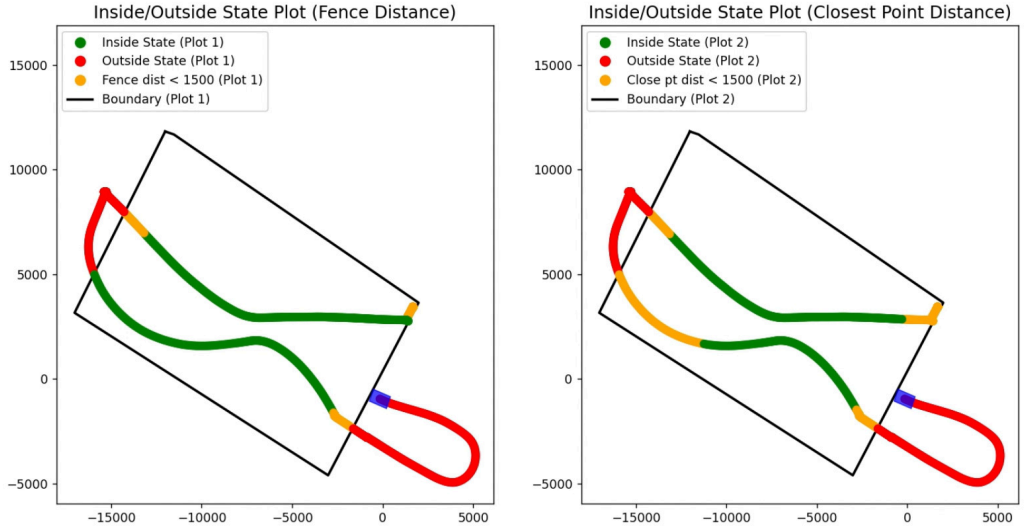
machine is doing. On the service tool screen, these states indicate whether the machine is inside or outside the boundary by displaying a 1 or 2. It also says if **Geofence_Check** is disabled so no boundary readings occur, displayed by 0. If the service tool screen displays 255, then the machine is in an invalid or safe state.

Other outputs include the machine's position in relation to the geofence, as well as the geofence boundary standard deviation. **Fence_Distance** reflects the distance from the machine's steering point along the x-axis through the front of the machine to the fence or boundary. The closest distance from the machine's steering point to the boundary is indicated by **Closest_Pt_Distance**. This shifts direction to wherever the closest part of the boundary is located in relation to the steering point. The closest point angle, **Closest_Pt_Angle**, is the angle between **Closest_Pt_Distance** and **Fence_Distance**. This angle also changes based on the machine's location. If checking multiple areas of the machine, these outputs would work from the locations where the coordinates were transferred with **Transform_GNSS**, rather than the steering point.



The image above shows the outputs of **Geofence_Check**.

Additionally, the boundary calculations take the largest X and Y standard deviations among all the points and uses those in the boundary calculations. For example, if the X standard deviation among three points is 10 mm, 20 mm, and 25 mm, then the final X standard deviation used to calculate all three outputs is 25 mm. This 25 mm appears in both an inner boundary and an outer boundary around the actual boundary. Large standard deviations could indicate a larger margin for error. Smaller standard deviations indicate a more precise boundary and therefore more precise outputs.



The images above show a machine driving over the virtual boundary. The left image displays data related to **Fence_Distance**. When the machine is less than 1500 mm from the boundary in the machine's x

Geofence_Check Function Block

direction, the path color turns yellow. The right image shows the same path but measuring **Closest_Pt_Distance**. When the machine's steering point is less than 1500 mm from the closest point on the boundary, the path turns yellow. The red path indicates the machine is outside of the boundary. These plots use the outputs of **Geofence_Check**, which could be used by downstream code to control the machine's reaction in relation to the boundary.

Some items to note:

- Monitor the **State** output for positioning while running the geofence checker. Check where the machine is with respect to the boundary and monitor for errors. The **State** turns to 255 on the service tool screen if errors occur. It could mean the data locker ID or boundary has changed.
- Monitor the machine's position with respect to the boundary in real time to ensure the outputs are as expected. Also, monitor the distance and angle outputs to ensure that the machine is physically inside or outside the boundary when it indicates it is on the service tool screen. Verify the expected **Closest_Pt_Distance** correlates between the service tool screen and the physical machine.
- It is recommended to use the **Position_Filter** function block to estimate the position of the machine.
- Use the **Boundary_Extract** function block to validate the boundary.
- Monitor the standard deviation to check that it is within an allowable tolerance.
- Plot the outputs of **Geofence_Check** and the machine's path in another tool, such as Excel.

Application Information

Common function blocks that work with **Geofence_Check** are **Boundary_Converter**, **Boundary_Extract**, **Boundary_Loader**, **Boundary_Recorder**, **Data_Lockers**, and **Transform_GNSS**.

Geofence_Check comes at the end of several function blocks which obtain a boundary. It detects if a machine crosses that boundary, and code downstream determines the machine reaction.

Review how **Geofence_Check** works in sequences with:

- **Transform_GNSS** in *Application Information* on page 216.
- **Boundary_Converter** in *Application Information* on page 44.
- **Boundary_Recorder** in *Application Information* on page 64.
- **Boundary_Loader** in *Application Information* on page 57.

Additionally, place the **Geofence_Check** function block:

- With one **Data_Lockers** block, version 1.12 or later, which can be on any page in the application.
- After position information is obtained, such as after a **Position_Filter** function block.
- After a boundary is obtained from either **Boundary_Converter**, **Boundary_Loader**, or **Boundary_Recorder**.
- Multiple times in an application if there are many boundaries. There should be a **Geofence_Check** block for each boundary, and a machine needs a new boundary for each region it stays inside or outside.
- Multiple times in an application if there are many areas on the machine to check. Use a **Geofence_Check** and **Transform_GNSS** block for monitored each area.

Example

The **Geofence_Check** function block is in examples with other boundary blocks.

- See the *Example* on page 66 with the **Boundary_Recorder** and **Boundary_Loader** function blocks.
- See the *Example* on page 57 with two **Boundary_Loader** function blocks.
- See the *Example* on page 44 with the **Boundary_Converter** function block.

Geofence_Check Function Block

Inputs

The following table describes inputs required for the **Geofence_Check** function block.

Item	Type	Range	Description [Unit]
Enable	BOOL	T/F	Determines whether to trigger **Geofence_Check**. T: **Geofence_Check** is used. F: **Geofence_Check** is ignored.
Boundary	S8	-1-99	The ID of the boundary type data locker.
Pos	BUS	——	Position and coordinate signals coming from the **Position_Filter** function block.
X	S32	-2147483648-2147483647	Current X position of the machine location. [mm]
Y	S32	-2147483648-2147483647	Current Y position of the machine location. [mm]
Yaw	S32	-72000-72000	The angle used to describe the machine's heading using the ENU (East-North-Up) reference frame. [0.01 degree]

Outputs

The following table describes outputs required for the **Geofence_Check** function block. This block comes last in the boundary block series.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Updated	BOOL	T/F	Indicates whether new data processed. T: Data processed, and **Enable** is True. F: No data processed, or **Enable** is False.
State	U8	0-255	The current machine state with respect to the boundary. 0: **Geofence_Check** is off, and **Enable** is False. 1: The machine is inside the boundary, and **Enable** is True. 2: The machine is outside the boundary, and **Enable** is True. 255: The machine is in an invalid or safe state, and **Enable** is True.
Closest_Pt_Distance	U32	0-4294967295	The shortest distance from the machine's steering point to the closest boundary point. This point could be located in any direction in relation to the machine. [mm]
Fence_Distance	U32	0-4294967295	The distance from the machine's steering point through the front of the machine to the boundary. If the machine is not facing the boundary, this value is 4294967295. [mm]
Closest_Pt_Angle	S32	-2147483648-2147483647	The angle of the machine's **Yaw** to the closest boundary point. This angle could be located in any direction in relation to the machine. [0.01 degree]
Std_Dev_X	U32	0-4294967295	The highest standard deviation of boundary points along the x-axis. Smaller numbers indicate a more precise boundary. [mm]
Std_Dev_Y	U32	0-4294967295	The highest standard deviation of boundary points along the y-axis. Smaller numbers indicate a more precise boundary. [mm]

Geofence_Check Function Block

Internal Signals

The following table describes what is happening internally in the **Geofence_Check** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Geofence_Check_Err_Specific	U16	0x0000 to 0x003F	Indicates when a specific error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Geofence_Check Troubleshooting* on page 88.
Geofence_Check_Err_Common	U16	0x0000 to 0x0007	Indicates when a generic error occurred in the block functionality. Bitwise code where multiple errors can be reported at the same time. See *Troubleshooting Common Errors* on page 36.
Processing_Time	U32	0-4294967295	The amount of time taken for **Geofence_Check** to receive data and process it. High processing time increases the latency for downstream function blocks, and machines react slower as the processing time increases. [µs]

Geofence_Check Troubleshooting

The following table describes errors that could occur in the **Geofence_Check** function block, as well as ways to fix them.

View the **Geofence_Check_Err_Specific** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

See *Troubleshooting Common Errors* on page 36 to fix errors that appear in many function blocks.

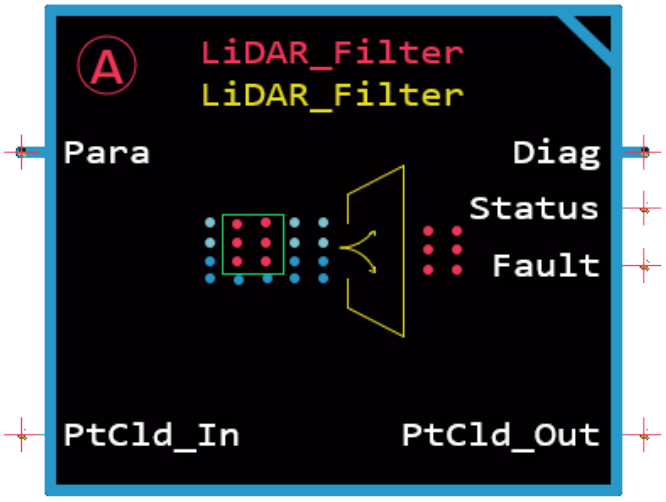
These errors can be visually represented in the bit, hexadecimal, and decimal versions in the service tool.

Geofence_Check_Err_Specific Descriptions and Fixes

Number	Description	How to Fix
0x0000	No errors.	Nothing needs to change.
Bit 0	The data locker is not available. The data locker may be an incorrect type or not have enough space.	Check the C code to see that the size of the data buffer is large enough and a boundary type data locker is used.
Bit 1	The Locker ID changed unexpectedly. The data in the data locker is invalid and cannot be read.	Verify the data locker ID is valid and not -1. Verify the input boundary has not become corrupt. Use the **Boundary_Extract** function block to check that the input boundary is correct.
Bit 2	Unique identifier validation failed, indicating potential data corruption or unauthorized modification.	Check that the sequence ID is increasing.
Bit 3	The checksum is invalid. There may be a change in the boundary data.	Use the **Boundary_Extract** function block to review the boundary data. To change the boundary, turn the **Enable** signal to False, load the boundary data, and turn **Enable** to True.
Bit 4	Data is not coming from a boundary function block earlier in the application code.	Check the data locker and code from previous boundary blocks.
Bit 5	Data stored in the data locker returns a null pointer.	Check the code to see if something is not being set to a null pointer.

LiDAR_Filter Function Block

The **LiDAR_Filter** function block eliminates unnecessary data from a LiDAR hardware scan, thereby saving processing power and gathering more focused data.

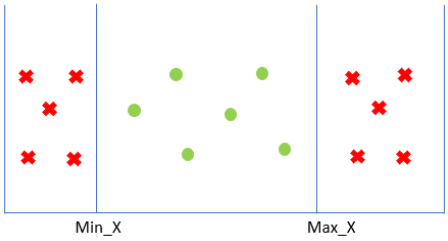


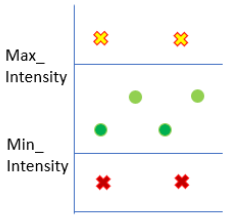
This block requires a license for A+ Advanced.

LiDAR_Filter requires LiDAR hardware and accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. See the *Plus+1 Compliant Ouster Block User Manual* for information, including background on LiDARs.

LiDAR_Filter includes many parameters to adjust a LiDAR scan and eliminate unwanted data. The LiDAR hardware scans the environment by sending out laser beams and recording the coordinate point in space when they land on something, thereby forming a point cloud. That point cloud data goes into the **LiDAR_Filter** function block. Then, parameters set boundaries between a minimum and maximum value to either keep or remove point cloud data. Boundaries could be physically in space or limits on light reflectivity. Removed data cannot be seen or saved.

LiDAR_Filter behaves similarly to **Planar_Surface_Segmentation**, but processes data faster and requires manually entering which LiDAR data to remove.

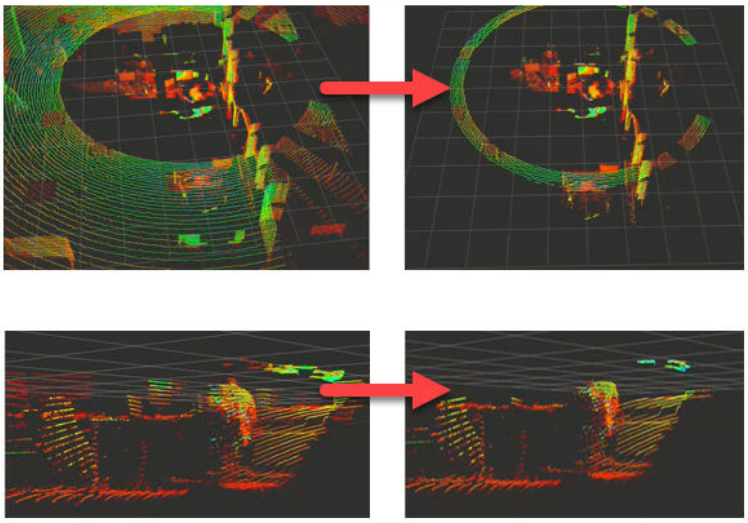




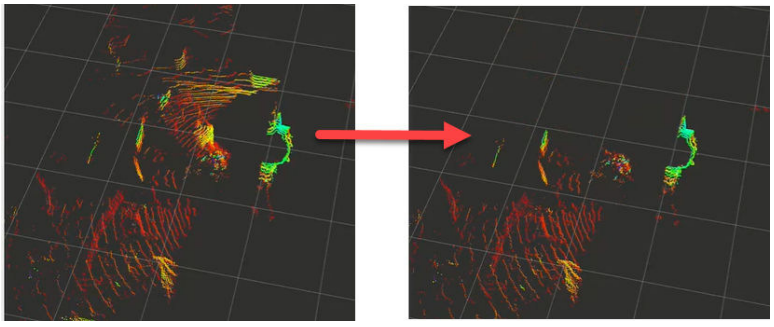
The images above show how point cloud data within minimum and maximum parameters are kept (circles), and point cloud data outside the parameter boundaries are removed (x's). Numbers within the minimum and maximum boundaries could indicate spatial dimensions (left image) or light reflectivity (right image).

LiDAR_Filter removes point cloud data surrounding the physical LiDAR hardware in various shapes. These shapes include a 2D or 3D rectangle with minimum and maximum values for X, Y, and Z. Other parameters remove sections of the 360 degree circle around the LiDAR hardware, such as range, elevation, and azimuth angles. Mix parameters to create unique shapes, if desired. By default, point cloud data is kept inside of a shape and removed outside of it. If parameters create overlapping shapes, the overlapped point cloud data is kept rather than data within the whole shape. Multiple shapes that do not overlap cannot remove any point cloud data. Instead, use multiple **LiDAR_Filter** function blocks in an application.

LiDAR_Filter Function Block

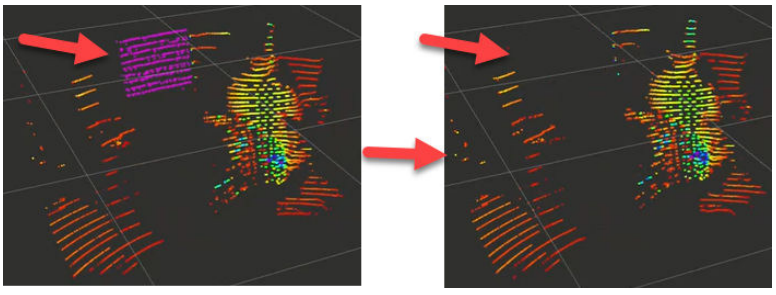


The images above show a person sitting at a desk with a LiDAR point cloud around them. The top images show the **Max_Range** parameter lowered so the point cloud does not go out as far from the LiDAR hardware, and the bottom images show the **Max_Elevation** parameter lowered so the person's head is out of view.



The image above shows the **Max_Azimuth** parameter lowered, which cuts out horizontal sections of the point cloud.

LiDAR_Filter parameters also adjust for reflectivity and light, allowing applications to ignore fog or construction vests. These ambient and intensity options could be used with or without the other shape cutting parameters.

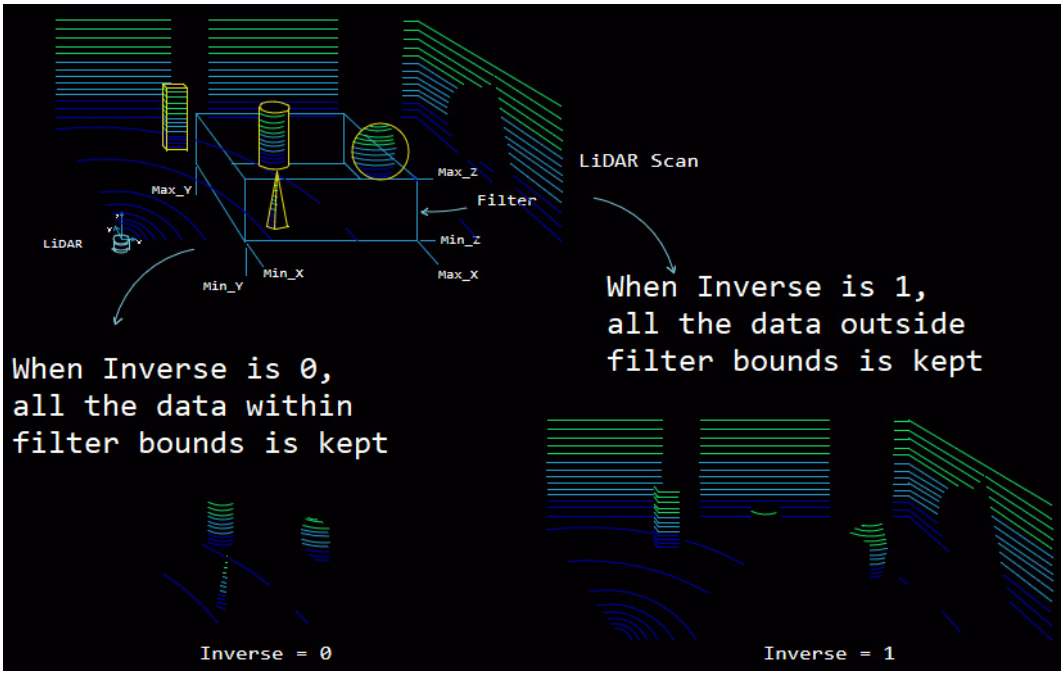


The image above shows a LiDAR scan point cloud with a rectangular reflective square next to a sitting person. Then, the **Max_Intensity** parameter is lowered to eliminate the reflective square from the point cloud.

If **Inverse** is selected or True, then point cloud data within a shape is removed, and data outside the shape is kept in the application. When **LiDAR_Filter** inverses reflectivity values, the LiDAR keeps point cloud data with very high and low reflectivity and removes point cloud data landing on reflective objects

LiDAR_Filter Function Block

within the middle of the spectrum. If some parameters require inverse point cloud data but not others, use multiple **LiDAR_Filter** function blocks in an application.



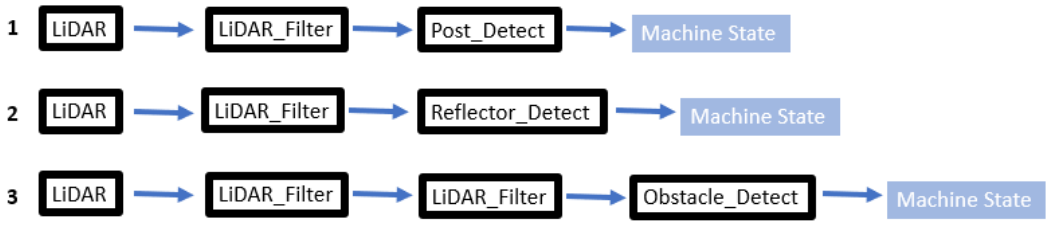
The image above shows how the **Inverse** parameter switches whether point cloud data is used or removed inside of the minimum and maximum parameter bounds.

Either ordered or unordered point cloud data could enter **LiDAR_Filter**. Ordered point cloud data includes X, Y, and Z values for each laser beam point that the LiDAR hardware sends out. These X, Y, and Z values are different than the **LiDAR_Filter** rectangular shape parameters. Unordered point cloud data eliminates the point position information but allows faster processing time. The **Ordered** parameter allows data to leave **LiDAR_Filter** in either an ordered or unordered state. However, after data becomes unordered, the X, Y, and Z point information can never be recovered to switch back to ordered.

After sectioning off the point cloud within **LiDAR_Filter**, data can pass to any detection blocks, such as **Obstacle_Detect**, **Post_Detect**, or **Reflector_Detect**.

Application Information

Common function blocks used with the **LiDAR_Filter** function block are **Data_Lockers**, **Obstacle_Detect**, **Post_Detect**, **Reflector_Detect**, and the **Ouster_LiDAR** function block if using an Ouster LiDAR hardware.



For an application to automatically divide a point cloud while the machine moves, use the **Planar_Surface_Segmentation** function block instead of creating fixed point cloud criteria in **LiDAR_Filter**.

LiDAR_Filter Function Block

1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. This gathers point cloud data which **LiDAR_Filter** uses. Sections of the point cloud could be filtered if no posts exist in that line of view. Then, **Post_Detect** could be programmed to react when posts are detected, such as a machine moving toward a post.
2. Scenario two shows **LiDAR_Filter** passing information into **Reflector_Detect**. The intensity and ambient light parameters could filter out high levels of brightness so construction vests are not detected by **Reflector_Detect**, but less bright reflectors are detected. Then, the machine can be programmed to react to the reflectors it sees.
3. Scenario three includes two **LiDAR_Filter** function blocks. Parameters within the first **LiDAR_Filter** eliminate sections of the point cloud data which the application does not need, such as removing points going into the sky. Then, the second **LiDAR_Filter** could remove a rectangle from the point cloud so points do not land on the machine itself. The smaller point cloud data left goes into **Obstacle_Detect** for a machine to react when an obstacle is detected, such as slowing down.

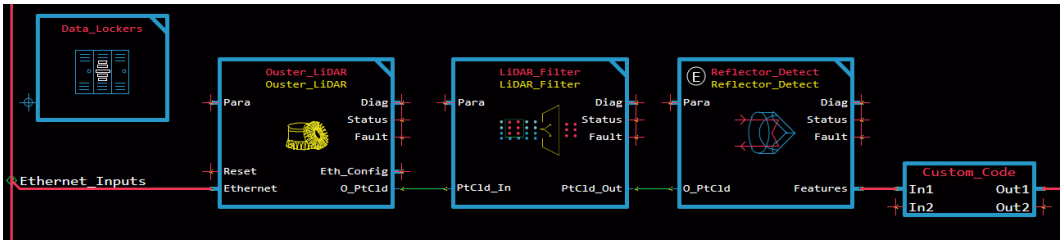
Additionally, place the **LiDAR_Filter** function block:

- After LiDAR data is collected, such as after the **Ouster_LiDAR** function block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- In a part of the application where removed point cloud data will not be needed by any function blocks downstream.

Example

The example shows the **LiDAR_Filter** function block used to filter out point cloud data from the ceiling, as well as fog.

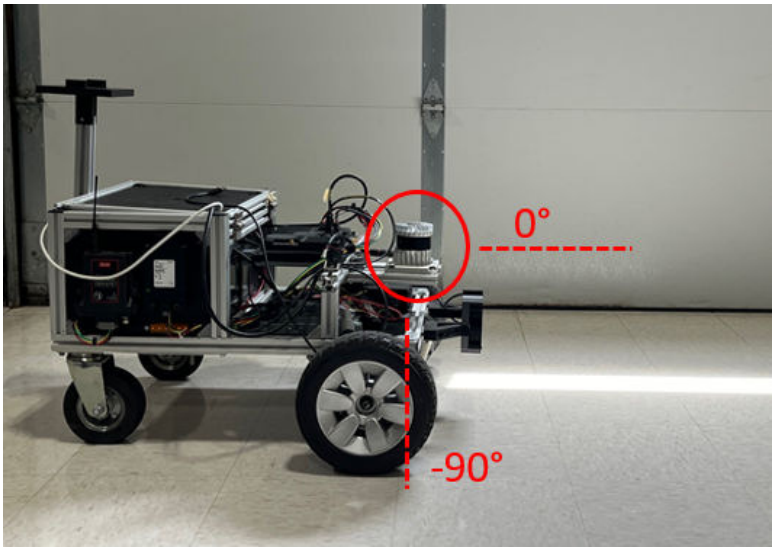
The environment where the machine runs matters a great deal for **LiDAR_Filter**. Test the machine in different environmental settings which relate to where the machine will be used in real life situations.



1. Set up a LiDAR scanner on the machine. This example uses Ouster LiDAR hardware. View the *Ouster LiDAR User Manual* to set up the hardware and **Ouster_LiDAR** function block.
2. Set up the LiDAR code. This example uses the **Ouster_LiDAR** function block. The LiDAR scan is determined by the physical hardware and accompanying LiDAR code. For example, if the hardware is set up so the ground or wall cannot be seen, or the **Ouster_LiDAR** function block eliminates part of the point cloud, these parameters cannot be adjusted in **LiDAR_Filter**.
3. Add in the **Data_Lockers** block if it was not already in the application. This block should not be connected to any buses or wires, only exist once in the application, and can go on any page within the application.
4. Connect **O_PtCld** from the **Ouster_LiDAR** block to **PtCld_In** in **LiDAR_Filter**. This allows point cloud information gathered from the LiDAR scan to be consumed by **LiDAR_Filter**. This information flows to a data locker and back automatically between these two blocks, so no numbers need to be adjusted.
5. Connect **LiDAR_Filter** to another Autonomous Control Library (ACL) perception block. Here, the filtered point cloud connects to the **Reflector_Detect** function block. Custom code after **Reflector_Detect** tells the machine how to react when it encounters reflectors.
6. Determine whether the LiDAR reads data within the minimum and maximum parameter boundaries that will be configured. Here, the **Inverse** is False so data is used inside the boundaries and eliminated outside of them.
7. In the environment surrounding the machine, measure the angle from the LiDAR hardware to the ceiling to determine which angle of the point cloud to cut off. Here, **Max_Elevation** is set to 0 so the

LiDAR_Filter Function Block

LiDAR hardware looks straight ahead and not above. **Min_Elevation** is set to -9000 for -90 degrees to look toward the ground.



The image shows a LiDAR (circled) and what it will view between 0 to -90 degrees (dotted lines).

8. Test if the point cloud eliminated points outside of the minimum and maximum elevation parameter values. Here, that is 0 to -90 degrees elevation, so the LiDAR scan should only include points straight from its eye and down to the ground.
   - Use Ouster Studio to visualize the point cloud if using an Ouster LiDAR.
   - Connect another function block, like **Obstacle_Detect**, and set a zone where the ceiling would be. See if the LiDAR detects something in that zone. If **LiDAR_Filter** works, then nothing should be picked up in that zone.
   - Check the internal signals to see that the **Vertical_Resolution** numbers change.
9. Put in values for **Min_Ambient** and **Min_Intensity** to filter out very low reflective light that occurs with fog. Reflective values occur at different rates based on each LiDAR hardware. By filtering out low reflectivity, the LiDAR will still pick up construction vests and higher reflective elements in the environment.
10. Test the reflective parameters by placing different reflective items in front of the LiDAR. Ideally, use fog. Leaving the **Max_Ambient** and **Max_Intensity** values as the default allow point cloud points with high reflectivity values to be used in the application.
    - Use Ouster Studio to visualize the point cloud reflectivity values.
    - Check the internal signals to see that the **Total_Valid_Points_Output** numbers change.
11. Leave any unwanted **LiDAR_Filter** parameters as the default. This ensures more sections of the point cloud will not be eliminated. Decide whether the point cloud should be ordered or unordered for the **Ordered** parameter.
12. The **Updated** flag on the service tool indicates that point cloud information passed out of **LiDAR_Filter** to **Reflector_Detect**. However, this cannot tell which points, if any, were eliminated from the LiDAR scan. Create code for the machine to react when the LiDAR scan detects reflective objects with higher ambient and intensity reflectivity than fog.

LiDAR_Filter Function Block

Inputs

The following table describes inputs required for the **LiDAR_Filter** function block. This block requires a LiDAR, LiDAR code such as the **Ouster_LiDAR** function block, and a **Data_Lockers** block. Any block that outputs point clouds can be used as an input to **LiDAR_Filter**.

Item	Type	Range	Description [Unit]
PtCld_In	S8	-1-99	The point cloud data locker ID where LiDAR scan data is stored. Point cloud data can be 2D, 3D, ordered, or unordered. [Locker ID]

Parameters

The following table describes parameters for the **LiDAR_Filter** function block. Most blocks take in ordered point cloud data, but only a few blocks can access unordered point cloud data.

Leave the default values unless removing point cloud data.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Inverse	BOOL	T/F	Determines whether to use point cloud data inside or outside the bounds of minimum and maximum parameters. T: Point cloud data outputs from outside minimum and maximum bounds. Point cloud data inside of these bounds is eliminated. F: Point cloud data outputs from inside minimum and maximum bounds. Point cloud data outside of these bounds is eliminated. Default: False
Ordered	BOOL	T/F	Determines whether the block outputs ordered or unordered point cloud data. This is only valid if **PtCld_In** is ordered. T: **PtCld_Out** is ordered. F: **PtCld_Out** is unordered. Default: True
Min_X	S32	-2147483648 to **Max_X** -1	The minimum X coordinate value determined by using the right hand rule. Point cloud data below this value is eliminated from the output point cloud unless **Inverse** is True. Default: -2147483648 [mm]
Max_X	S32	**Min_X** +1 to 2147483647	The maximum X coordinate value determined by using the right hand rule. Point cloud data above this value is eliminated from the output point cloud unless **Inverse** is True. Default: 2147483647 [mm]
Min_Y	S32	-2147483648 to **Max_Y** -1	The minimum Y coordinate value determined by using the right hand rule. Point cloud data below this value is eliminated from the output point cloud unless **Inverse** is True. Default: -2147483648 [mm]
Max_Y	S32	**Min_Y** +1 to 2147483647	The maximum Y coordinate value determined by using the right hand rule. Point cloud data above this value is eliminated from the output point cloud unless **Inverse** is True. Default: 2147483647 [mm]
Min_Z	S32	-2147483648 to **Max_Z** - 1	The minimum Z coordinate value determined by using the right hand rule. Point cloud data below this value is eliminated from the output point cloud unless **Inverse** is True. Default: -2147483648 [mm]
Max_Z	S32	**Min_Z** + 1 to 2147483647	The maximum Z coordinate value determined by using the right hand rule. Point cloud data above this value is eliminated from the output point cloud unless **Inverse** is True. Default: 2147483647 [mm]

LiDAR_Filter Function Block

Item	Type	Range	Description [Unit]
Min_Intensity	U16	0 to **Max_Intensity** - 1	Intensity refers to brightness illuminated by the laser in the LiDAR. Point cloud data below this minimum intensity value is eliminated from the output point cloud unless **Inverse** is True. Default: 0 [0.01%]
Max_Intensity	U16	**Min_Intensity** + 1 to 10000	Intensity refers to brightness illuminated by the laser in the LiDAR. Point cloud data above this maximum intensity value is eliminated from the output point cloud unless **Inverse** is True. Default: 10000 [0.01%]
Min_Ambient	U16	0 to **Max_Ambient** - 1	Ambient refers to light in the background. Point cloud data below this minimum intensity value is eliminated from the output point cloud unless **Inverse** is True. Default: 0 [0.01%]
Max_Ambient	U16	**Min_Ambient** + 1 to 10000	Ambient refers to light in the background. Point cloud data above this maximum intensity value is eliminated from the output point cloud unless **Inverse** is True. Default: 10000 [0.01%]
Min_Azimuth	S16	-18000 to **Max_Azimuth** -1	Azimuth refers to the side-to-side angle from the LiDAR. Point cloud data below this minimum azimuth value is eliminated from the output point cloud unless **Inverse** is True. Default: -18000 [0.01 degree]
Max_Azimuth	S16	**Min_Azimuth** +1 to 18000	Azimuth refers to the side-to-side angle from the LiDAR. Point cloud data above this maximum azimuth value is eliminated from the output point cloud unless **Inverse** is True. Default: 18000 [0.01 degree]
Min_Elevation	S16	-9000 to **Max_Elevation** -1	Elevation refers to height, or the up-and-down angle from the LiDAR. Point cloud data below this minimum elevation value is eliminated from the output point cloud unless **Inverse** is True. Default: -9000 [0.01 degree]
Max_Elevation	S16	**Min_Elevation** +1 to 9000	Elevation refers to height, or the up-and-down angle from the LiDAR. Point cloud data above this maximum elevation value is eliminated from the output point cloud unless **Inverse** is True. Default: 9000 [0.01 degree]
Min_Range	U32	0 to **Max_Range** -1	Range refers to close-to-far distance from the LiDAR. Point cloud data below this minimum elevation value is eliminated from the output point cloud unless **Inverse** is True. Default: 0 [0.01 degree]
Max_Range	U32	**Min_Range** +1 to 4294967295	Range refers to close-to-far distance from the LiDAR. Point cloud data above this maximum elevation value is eliminated from the output point cloud unless **Inverse** is True. Default: 4294967295 [0.01 degree]

LiDAR_Filter Function Block

Outputs

The following table describes outputs for the **LiDAR_Filter** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0X8008: At least one parameter is out of range. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
PtCld_Out	S8	-1-99	The point cloud data locker ID where filtered point cloud data is stored. [Locker ID]

Internal Signals

The following table describes what is happening internally in the **LiDAR_Filter** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
LiDAR_Filter_Err	U8	0-6	Indicates when an error occurred in the block functionality. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: Elevation or azimuth parameters are outside of the minimum and maximum ranges of the LiDAR. 5: Selected filtering attribute does not exist in the original point cloud. 6: Input point cloud is unordered and the **Ordered** flag is True.
Updated	BOOL	T/F	Indicates that a new point cloud is available in the output data locker. T: A new point cloud is available. F: No new data is available.
Vertical_Resolution	U32	0-4294967295	The number of LiDAR ring rows. These numbers decrease with elevation parameters applied. This value will be 1 if the output point cloud is unordered, indicating one compressed ring with Not A Number (NANs). If the numbers are 0 or are the same after elevation parameters are applied, then elevation is not working.
Horizontal_Resolution	U32	0-4294967295	The number of LiDAR points in a ring row. These numbers decrease with azimuth parameters applied. This value will be equal to **Total_Valid_Points_Output** when the output point cloud is unordered. If the numbers are 0 or are the same after azimuth parameters are applied, then azimuth is not working.
Total_Valid_Points_Input	U32	0-4294967295	All the point cloud points entering **LiDAR_Filter**. Valid LiDAR points require at least an X, Y, or Z coordinate, otherwise they appear as Not A Number (NANs). This should have higher numbers than **Total_Valid_Points_Output** to indicate the filtering parameters worked.
Total_Valid_Points_Output	U32	0-4294967295	All the point cloud points exiting **LiDAR_Filter**. Valid LiDAR points require an X, Y, or Z coordinate, otherwise they appear as Not A Number (NANs). This should have lower numbers than **Total_Valid_Points_Input** to indicate the filtering parameters worked. If this is 0, then all the LiDAR points were discarded and something likely went wrong.

LiDAR_Filter Function Block

Item	Type	Range	Description [Unit]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.
Processing_Time	U32	0-4294967295	The amount of time taken for **LiDAR_Filter** to receive data, process it, and produce a new point cloud. High processing time increases the latency for downstream function blocks, and machines react slower as the processing time increases. [µs]

LiDAR_Filter Troubleshooting

The following table describes errors that could occur in the **LiDAR_Filter** function block and ways to fix them.

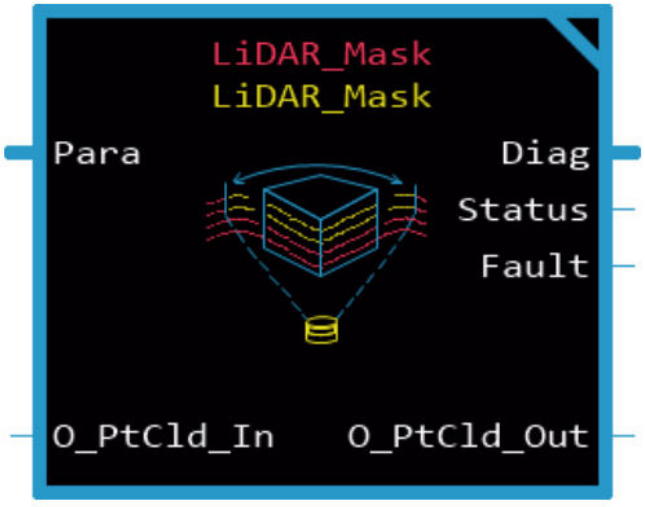
View the **LiDAR_Filter_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS +1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

LiDAR_Filter_Err Descriptions and Fixes

Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
2	Not enough memory available to create thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
3	Thread timeout.	There may be too much code creating a longer processing time. Reduce the LiDAR resolution or delete other processing blocks. See *Reduce Processing Time* on page 174. Turn the XM100 off, wait a bit, and restart it.
4	Elevation or azimuth parameters are outside of the capabilities of the LiDAR hardware.	Check the LiDAR hardware to determine the ranges allowed.
5	The reflective or spatial attribute filtered from the point cloud does not exist within the LiDAR hardware capabilities.	Check if the LiDAR hardware supports the desired attribute being filtered. Use the default parameters. Some LiDARs do not have reflective capabilities or have limited point cloud ranges.
6	Unordered point cloud data entered **LiDAR_Filter**, and the **Ordered** parameter is selected or True.	Deselect or change the **Ordered** parameter to False so unordered point cloud data outputs from **LiDAR_Filter**. Or, change the application code so ordered point cloud data enters **LiDAR_Filter**. In this case, ordered point cloud data outputs from **LiDAR_Filter** to the rest of the application, too.

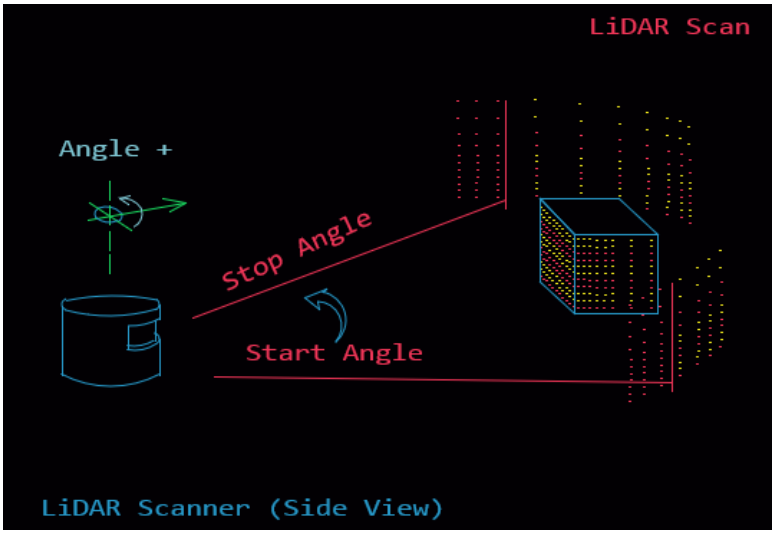
LiDAR_Mask Function Block

The **LiDAR_Mask** function block enables an application to use different sections from an input LiDAR scan.



LiDAR_Mask requires a piece of LiDAR hardware and accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. See the *Plus+1 Compliant Ouster Block User Manual* for information.

LiDAR_Mask takes in point cloud information from the LiDAR hardware and LiDAR code. Parameters within the block crop the point cloud field of view or hide some rows of the point cloud. This omits some data to save processing time.



The image shows a point cloud from the LiDAR scanner with some ring rows masked out.

Input data types must exactly match the indicated type to successfully compile.

Input ranges must be within the range indicated. An input that is out of range is coerced to fit within that range except arrays. Out of range array input values may result in unexpected behavior.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

LiDAR_Mask Function Block

Inputs

The following table describes inputs to the **LiDAR_Mask** function block.

Item	Type	Range	Description [Unit]
O_PtCld_In	S8	-1 - 99	The data locker ID of input ordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The following table describes parameters of the **LiDAR_Mask** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Start_Angle	S16	-18000 - [**Stop_Angle**-1]	Specifies which beam of the LiDAR scan is used to start masking the point cloud. Default: -18000 [0.01 degree]
Stop_Angle	S16	[**Start_Angle** +1] - 18000	Specifies which beam of the LiDAR scan is used to stop masking the point cloud. Default: 18000 [0.01 degree]
Bitmask	(ARRAY[256]U8)	0 - 1	Specifies which rings of the LiDAR scan are included in the masked point cloud. Each value in the array includes or excludes the corresponding ring based on the value. 0: Excludes the ring in the masked point cloud. 1: Includes the ring in the masked point cloud. The first value in the array controls the top-most ring in the LiDAR scan data. The final value controls the bottom-most ring in the LiDAR scan data. Default: ones(256)

Outputs

The following table describes outputs of the **LiDAR_Mask** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function. The bus contains all inputs, parameters, internal signals, and outputs. For more information on internal signals, see *Internal Signals* on page 100.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
O_PtCld_Out	S8	-1 - 99	The data locker ID of output ordered point cloud data.

LiDAR_Mask Function Block

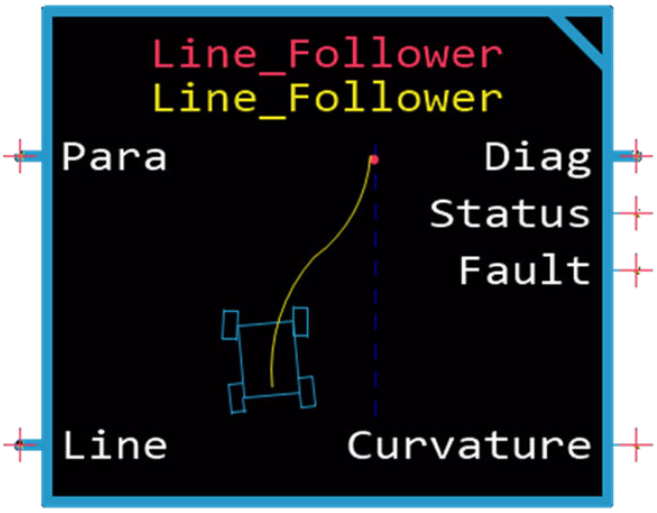
Internal Signals

The following table describes internal signals of the **LiDAR_Mask** function block.

Item	Type	Range	Description [Unit]
Lidar_Mask_Err	U8	0-8	Indicates errors occurred in the function block operation. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: **Start_Angle** is smaller than the sensor's minimum measuring angle. 5: **Stop_Angle** is bigger than the sensor's maximum measuring angle. 6: Invalid angular step. 7: No ring selected. 8: Point cloud is unordered.
Masked_Height	U32	0-4294967295	The total number of rings in the height of the masked point cloud. The height of the filtered point cloud depends on the **Bitmask** parameter. Height is equal to the total number of enabled rings.
Masked_Width	U32	0-4294967295	The total number of horizontal points in each ring constituting the width of the filtered point cloud. Width depends on the **Start_Angle**, **Stop_Angle** and the size of a single angular step.
Processing_Time	U32	0-4294967295	The amount of time taken to process the input point cloud. [µs]
Updated	BOOL	T/F	Indicates that a new point cloud is available in the output data locker. T: A new point cloud is available. F: No new data is available.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

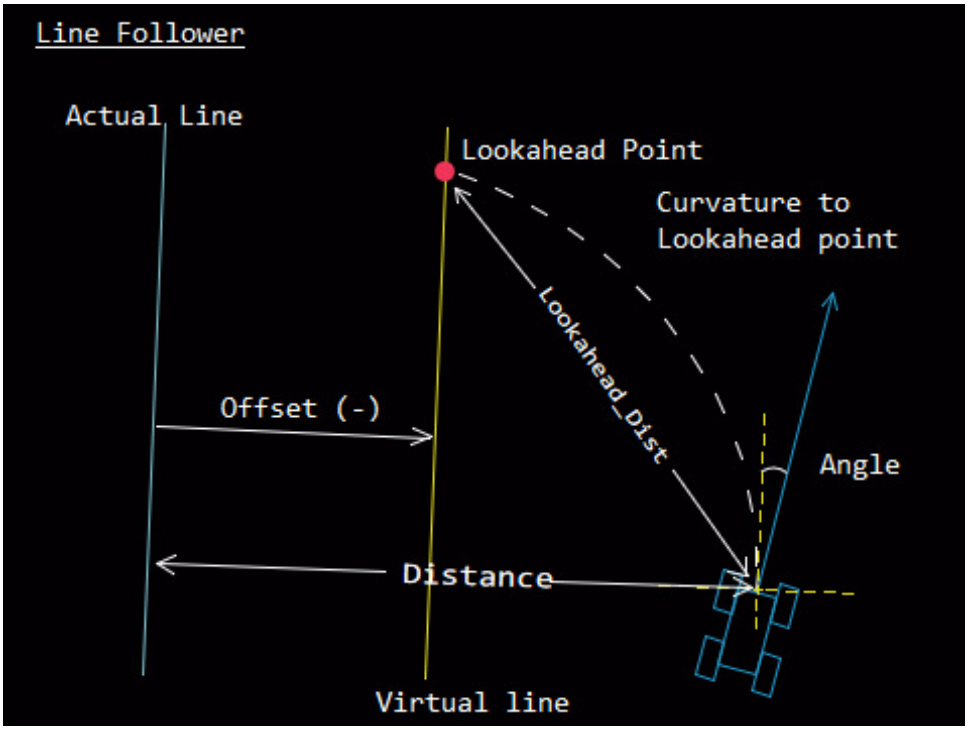
Line_Follower Function Block

The **Line_Follower** function block enables a machine to follow a line generated by a line detection block, such as **Wall_Detect**. The function block outputs a curvature value that brings the machine onto a path.



Use this block to have a machine travel in a line using a feature in the environment, such as a wall. The wall or other environmental feature registers as an actual line, and then the machine moves on an imaginary virtual line that is to the side of the actual line.

The **Line_Follower** diagram shows the machine aiming toward a point far ahead, called the Lookahead Point. The machine goes along a virtual line toward that point at a gentle curve if the Lookahead Point is far away, and the machine goes at a tighter curve if the point is very close. Because the **Line_Follower** function block uses angles to follow a line, the **Lookahead_Distance** must be greater than the distance to the virtual line. The output is a curvature, which can be used to control the machine's steering systems.

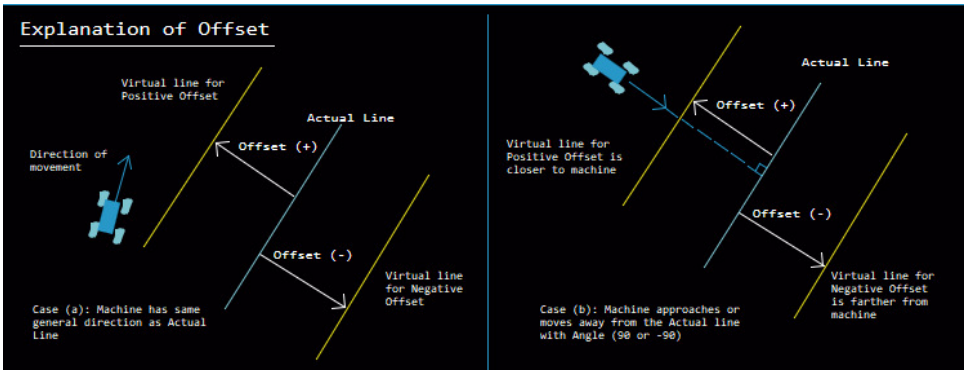


The offset parameter is the distance between the actual line and the virtual line. For example, paired with the **Wall_Detect** function block, the actual line is the wall. The machine runs on a virtual line alongside

Line_Follower Function Block

the actual line (wall), and the offset parameter could be how far away from the actual line (wall) the machine should be so it does not crash. The machine in this situation moves toward a point in the distance (Lookahead Point), and the machine goes over the virtual line in an "S" shape because it uses angles to detect the point ahead.

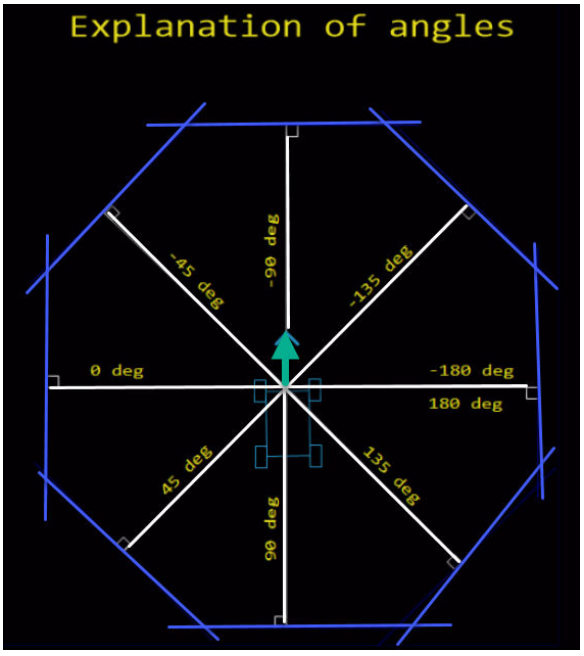
When the machine moves along a virtual line that is to the right of the actual line, the offset is a negative number. When the machine moves along a virtual line that is to the left of the actual line, the offset is a positive number.



When the machine approaches the actual line at 90 or -90 degrees, a positive offset means the virtual line is closer to the machine than the actual line. A negative offset means the virtual line is farther from the machine than the actual line. The machine faces directly away from the actual line at 90 degrees, and the machine faces the actual line at -90 degrees.

The **Explanation of angles** diagram shows the angles the machine makes relative to its position. A virtual line to the left of the machine shows 0 degrees. A virtual line ahead of the machine shows -90 degrees.

If the **Angle** input value is -90 or 90 degrees, the command might be unstable and result in incorrect **Curvature** output values to the left or right.



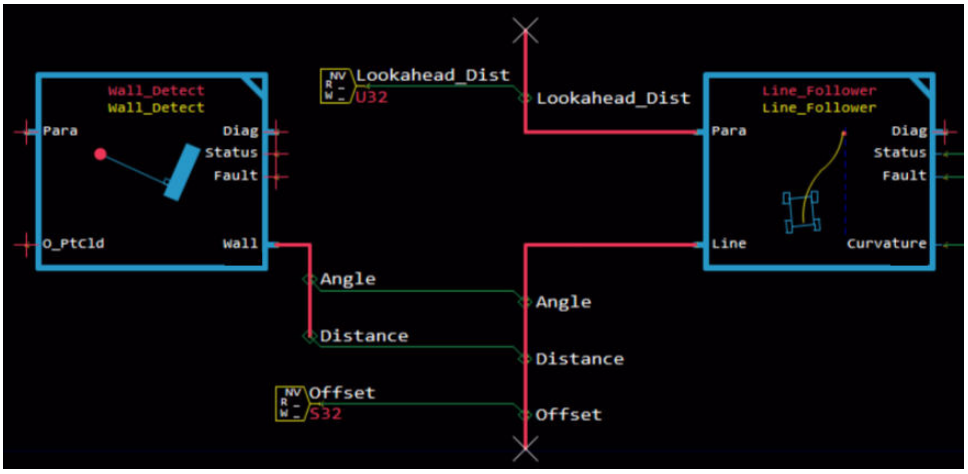
Line_Follower Function Block

Example

Review the example which uses the **Wall_Detect** function block to detect a wall and the **Line_Follower** function block to drive along the wall with an offset.

Follow these steps to add the function block to an application, and then route the inputs and outputs.

1. Configure **Wall_Detect**. For more information, see *Wall_Detect Function Block* on page 231.
2. Add **Line_Follower**.
3. Route the **Distance** and **Angle** into the **Line** input on **Line_Follower**.
4. Create an offset parameter to go into the **Line** input. For example, use the Non-volatile Memory Dynamic Input component in PLUS+1® GUIDE.



5. (Optional) Route the **Diag**, **Status**, and **Fault** outputs.
   a) Route **Diag** to report diagnostic data.
   b) Route **Status** to report data on the status of the function block.
   c) Route **Fault** to report data on issues relating to the function block.
6. Set the parameters for the **Line_Follower** function block. Use the Non-volatile Memory Dynamic Input component in PLUS+1® GUIDE to create the **Lookahead Dist**.
7. Save the application.

Inputs

The following table describes the inputs for the **Line_Follower** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Bus containing configuration signals for the function block.
Lookahead_Dist	U32	0 - 500000	Sets the distance ahead on the path that the machine targets while measuring the virtual line. Shorter distances are more accurate but the system is less stable. Set this to a value greater than the **Distance** input. Default: 3000 [mm]
Line	BUS	——	Bus containing polar coordinates of the virtual line for the machine to follow.
Distance	U32	0 - 500000	The normal (tangential) distance between the steering center of the machine and the actual line. [mm]
Angle	S16	-18000 - 18000	The angle between the actual line and where the machine is headed. If the **Angle** input value is -90 or 90 degrees, the command might be unstable and result in incorrect **Curvature** output values to the left or right. [0.01 deg]

Line_Follower Function Block

Item	Type	Range	Description [Unit]
Offset	S32	-500000 - 500000	Distance between the actual line and the virtual line. If the machine runs parallel to the actual line, positive offset means the virtual line is left of the actual line, and negative offset means the virtual line is right of the actual line. When the **Angle** is 90 or -90 degrees, positive offset creates a virtual line closer to the machine. [mm]
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

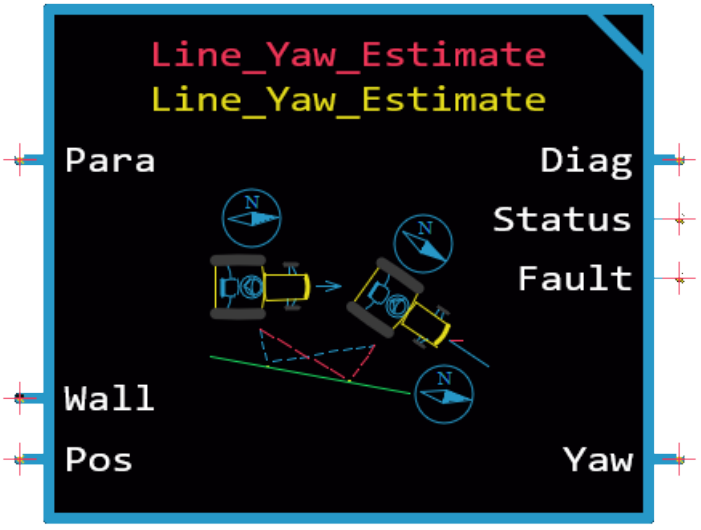
Outputs

The following table describes outputs from the **Line_Follower** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the condition of the function block. Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Reports issues experienced with the function. Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x0001: Input value too low. 0x0002: Input value too high. 0x0004: The total distance (**Distance** +/- **Offset**) is larger than or equal to the **Lookahead_Dist**.
Curvature	S32	-800000 - 800000	Curvature values needed to get to the virtual line. Positive values are left curves. Negative values are right curves. [0.01/km]

Line_Yaw_Estimate Function Block

The **Line_Yaw_Estimate** function block estimates the expected orientation of the machine from where it is pointed. It allows indoor navigation without GNSS data by using angles with reference to the walls to track the machine's orientation.

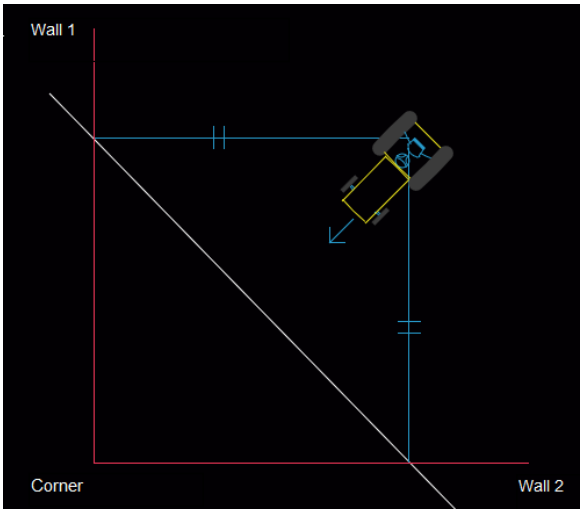


The **Line_Yaw_Estimate** function block improves the accuracy of the **Position Filter** yaw output in indoor environments where GNSS signals are weak. For example, the machine moves inside a room and uses a wall as a reference point to fix its bearing instead of unreliable GNSS data.

Use **Line_Yaw_Estimate** for machines working indoors and the **Yaw_Estimate** function block if the machine is outdoors with GNSS data.

Line_Yaw_Estimate estimates if the machine is angling away from the direction it was programmed to go. For example, if a machine is programmed to go alongside a wall, this block can detect if it is angled slightly toward or away from the wall after it starts moving.

However, if the machine is equidistant to two walls and pointed toward a corner, the output may be inconsistent. Move the machine slightly toward one of the walls to minimize the **Wall_Std_Dev** and correct the output.



Wall_Std_Dev should be utilized to prevent the scenario shown in the diagram.

Line_Yaw_Estimate Function Block

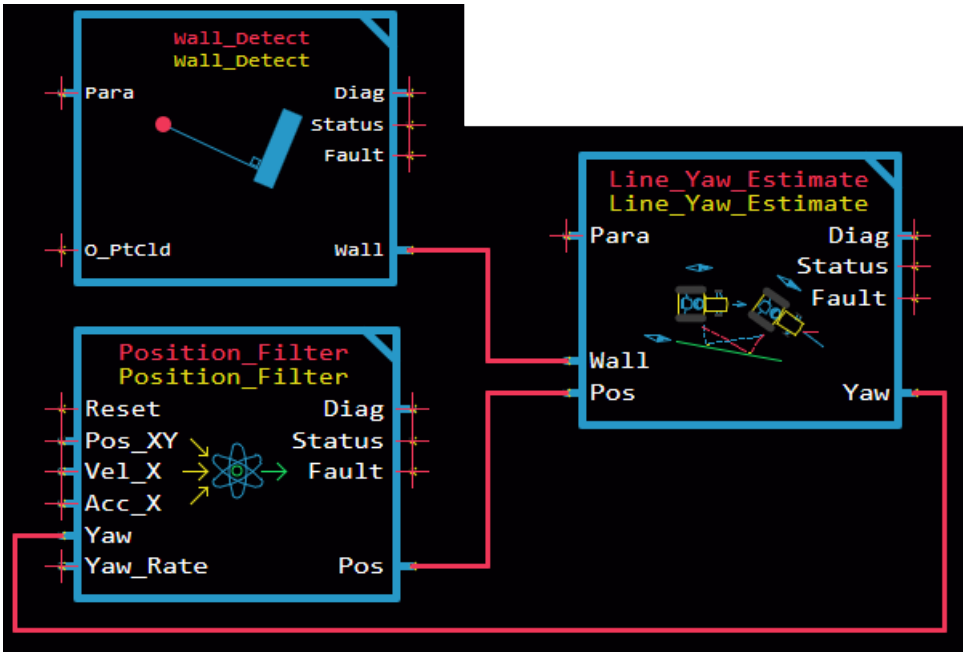
Example

Review the example using the **Wall_Detect**, **Position_Filter**, and **Line_Yaw_Estimate** function blocks together.

Before completing these steps:

- Set up the machine so it runs parallel alongside a wall and not at an angle. If the machine is turned off and back on again, it will need to be set up to run parallel to the wall.
- Verify that all walls are perpendicular to each other, such as a rectangular or square room.
- If used indoors, disable GNSS data coming from the **Position_Filter** block.
- If used outdoors with GNSS data, verify the wall used to estimate yaw is aligned to the geographical North with the machine facing east.

1. Configure **Wall_Detect**. For more information, see *Wall_Detect Function Block* on page 231.
2. Configure **Position_Filter**. Do not use GNSS data if using **Line_Yaw_Estimate** indoors. For more information, see *Position_Filter Function Block* on page 178.
3. Add **Line_Yaw_Estimate** to the application.
4. Route **Wall_Detect** and **Position_Filter** to **Line_Yaw_Estimate**.
5. Route the **Yaw** output from **Line_Yaw_Estimate** to the **Yaw** input on **Position_Filter** to create a feedback loop.



6. (Optional) Route the **Diag**, **Status**, and **Fault** outputs.
   a) Route **Diag** to report diagnostic data.
   b) Route **Status** to report data on the status of the function block.
   c) Route **Fault** to report data on issues relating to the function block.
7. Set the parameters for **Line_Yaw_Estimate**. Setting the threshold range very low calculates the value differences in the **Internal Signals** within the **Checkpoint** page.
8. Save the application.
9. Download the application with the machine in the correct position. The application needs to determine the wall data first in order for the whole program to work. If the machine is not oriented properly to the wall when it starts, the yaw will be incorrect.

Line_Yaw_Estimate Function Block

Inputs

The following table describes the inputs for the **Line_Yaw_Estimate** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Bus containing configuration signals for the function block.
Angle_Threshold	U16	0 - 4500	The maximum allowed yaw error that can be corrected by the **Line_Yaw_Estimate** function block. Default: 500 [0.01 deg]
Yaw_Std_Dev_Threshold	U32	1 - 4294967295	The maximum standard deviation allowed for deciding whether to update the output based on the **Position_Filter** function block's **Yaw_Std_Dev** signal. Default: 1000 [0.01 deg]
Wall_Std_Dev_Threshold	U32	1 - 4294967295	The maximum standard deviation allowed for deciding whether to update the output based on the **Wall_Detect** function block's **Angle_Std_Dev** signal. Default: 500 [0.01 deg]
Wall	BUS	——	Bus containing input values from the **Wall_Detect** function block.
Angle	S16	-18000 - 17999	The angle to the wall in radial coordinates provided by the **Wall** output on the **Wall_Detect** function block. [0.01 deg]
Angle_Std_Dev	U32	1 - 4294967295	The standard deviation of the **Angle** value from the **Wall_Detect** function block. It is an angular degree that the **Angle** input can deviate without modifying the value. [0.01 deg]
Updated	BOOL	T/F	The updated value from the Wall_Detect block.
Pos	BUS	——	Bus containing input values from the **Position_Filter** function block.
Yaw	S32	-18000 - 17999	The **Yaw** value from the **Position_Filter** function block. The **Yaw** value indicates the heading of the machine. [0.01 deg]
Yaw_Std_Dev	U32	1 - 4294967295	The standard deviation of the **Yaw** value. It is an angular degree that the **Yaw** input can deviate without modifying the value. [0.01 deg]
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Outputs

The following table describes outputs from the **Line_Yaw_Estimate** function block.

Function Block Outputs

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Difference	S16	-18000 - 18000	The relative difference between the **Angle** and **Yaw** input values. The function block uses this value to calculate the **Yaw** output. [0.01 deg]
Status	U16	——	Reports the current status of the function. It is bitwise code that reports multiple statuses. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.

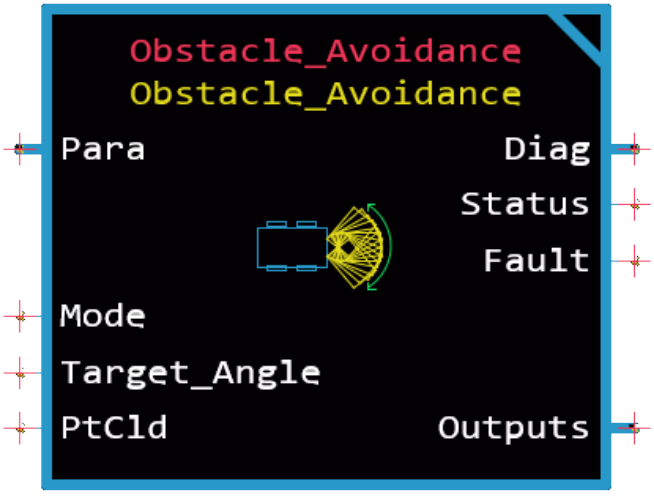
Line_Yaw_Estimate Function Block

Function Block Outputs (continued)

Item	Type	Range	Description [Unit]
Fault	U16	——	Reports issues related to the function. It is bitwise code that reports multiple faults. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Yaw	BUS	——	Bus containing **Yaw**, **Yaw_Std_Dev**, and **Updated** output signals. Connect this bus to the **Yaw** input on the **Position_Filter** function block.
Updated	BOOL	——	Indicates that new information is available from the function block. T: New data is available. F: No new data is available.
Yaw	S32	-18000 - 17999	The value of yaw for the machine. Use this value to correct the Yaw value in the **Position_Filter** function block. [0.01 deg]
Yaw_Std_Dev	U32	1 - 4294967295	The standard deviation of the **Yaw** output. [0.01 deg]

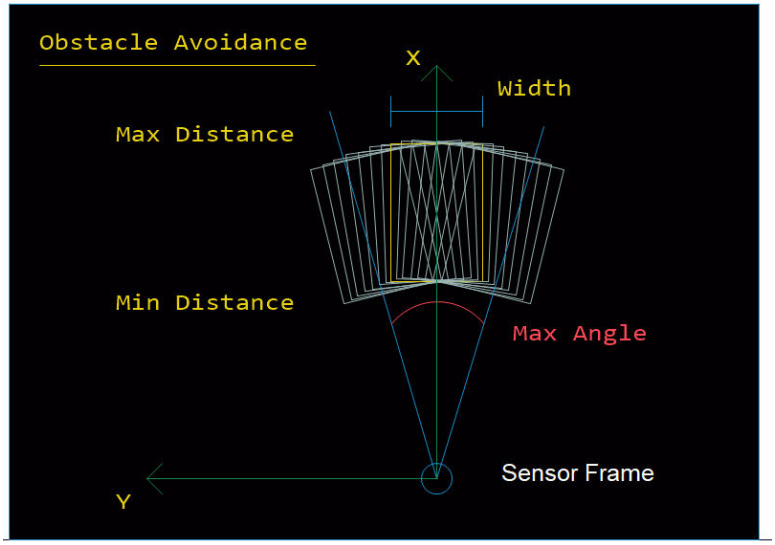
Obstacle_Avoidance Function Block

The **Obstacle_Avoidance** function block simplifies the evaluation of an incoming LiDAR scan and determines the best option.



The LiDAR scanner is typically placed at the front of a machine. Use the **Sensor_Offset** parameters to define where this LiDAR is with respect to the steering point of the machine. This block creates a fan of 15 zones centered around the steering point and spans the defined **Max_Angle**. This primary output is the **Best_Angle**, which is calculated based on the selected mode. Use the raw zone scores for a more customized interpretation.

Consider the width of the machine when calculating the best angle.



Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Obstacle_Avoidance Function Block

Inputs

Inputs to the **Obstacle_Avoidance** function block are described.

Item	Type	Range	Description [Unit]
Mode	U8	0-3	Selection of Best Angle calculation. 0: Raw, no additional processing of zone scores, **Best_Angle** = lowest score, a tie goes to the **Target_Angle**. 1: Weighted, scaling and addition factors are applied to each zone based on how far away they are from the **Target_Angle**. 2: Centered, designed for following a corridor and weighted to find the most open space in the middle. 3: Nearest Acceptable, next closest angle to **Target_Angle** that scores below the threshold unless **Target_Angle** is below the threshold.
Target_Angle	S16	-18000-18000	Desired angle for machine to drive. [0.01 deg]
PtCld	U8	-1-99	The data locker ID of ordered or unordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The following table describes parameters for the **Obstacle_Avoidance** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Max_Angle	U16	0-36000	Magnitude of max left to max right angle. Default: 3000 [0.01 deg]
Width	U16	0-65535	Width of the zones. All zones have the same width. Default: 1000 [mm]
Threshold	U16	1-10000	Higher limit of points for a zone to be invalid. Used only in mode 3. Default: 10 [number of points]
Weight_Scale	S16	-25000-25000	Scaling factor that is used to multiple the score of the zone. A negative value steers toward the highest scoring zone. Default: 1 [0.001]
Weight_Add	U16	0-1000	Factor (Integer) that is adding to each zone after scaling factor is applied. Default: 0
Min_Distance	U16	0 to **Max_Distance** - 1	Distance between the steering point of the machine and the start of the zone. Default: 0 [mm]
Max_Distance	U16	**Min_Distance** + 1 to 65535	Distance between the steering point of the machine and the end of the zone. Default: 1000 [mm]
Min_Height	S32	-50000 to **Max_Height** - 1	Minimum height of the zones with respect to the steering point. All zones have the same height. Default: 0 [mm]
Max_Height	S32	**Min_Height** + 1 to 50000	Maximum height of the zones with respect to the steering point. All zones have the same height. Default: 1000 [mm]

Obstacle_Avoidance Function Block

Item	Type	Range	Description [Unit]
Sensor_Offset_X	S32	-2147483648-2147483647	The distance from the steering point of the machine along the x-axis to the LiDAR scanner. LiDARs in front of the steering point have positive values and in back have negative values. Default: 0 [mm]
Sensor_Offset_Y	S32	-2147483648-2147483647	The distance from the steering point of the machine along the y-axis to the LiDAR scanner. LiDARs to the left of the steering point have positive values and to the right have negative values. Default: 0 [mm]
Sensor_Offset_Z	S32	-2147483648-2147483647	The distance from the steering point of the machine along the z-axis to the LiDAR scanner. LiDARs above the steering point have positive values and below have negative values. Default: 0 [mm]
Sensor_Orientation	S16	-18000-18000	Rotation of the LiDAR scanner around the z-axis in relation to the machine. Default: 0 [0.01 deg]

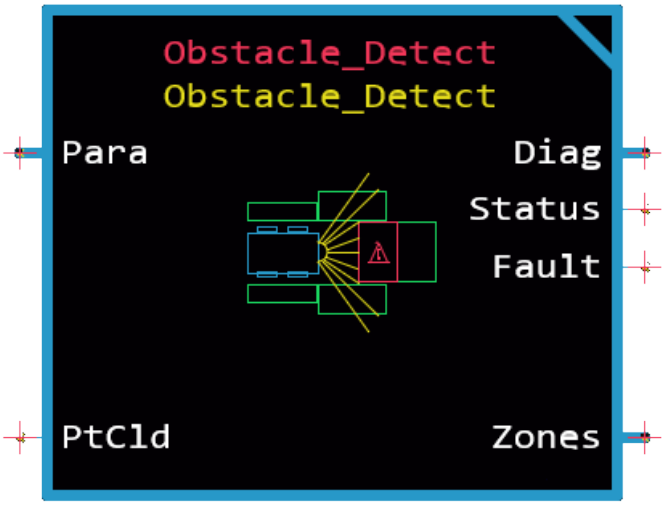
Outputs

Outputs of the **Obstacle_Avoidance** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Outputs	BUS	——	This bus provides zone scores and resulting angle information.
Updated	BOOL	T/F	New information is available from the block namespace for each Diag signal. T: New data is available. F: New data is not available.
Scores	(Array[15]U32)	0-4294967295	Reports the number of points in each zone from the LiDAR scan data.
Total_Valid_Points	U32	0-4294967295	The number of valid LiDAR points. These obtain data when landing on objects. An unusually low number may indicate issues. Invalid points include LiDAR points going into the sky or dark surfaces, which are not detected by the LiDAR. For ordered point clouds, this is the number of valid LiDAR points found within and close to the zones. LiDAR points are counted multiple times if zones overlap. For unordered point clouds, this is the number of valid LiDAR points the LiDAR sees in the whole point cloud, regardless of zones.
Zone_Angles	(Array[15]S16)	-32768-32767	Array of the calculated orientation of each zone. [0.01 deg]
Best_Score	U32	0-4294967295	Least points count in a single zone out of all the 15 zones.
Best_Angle	S16	-32768-32767	Angle of the zone with the best score. [0.01 deg]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Obstacle_Detect Function Block

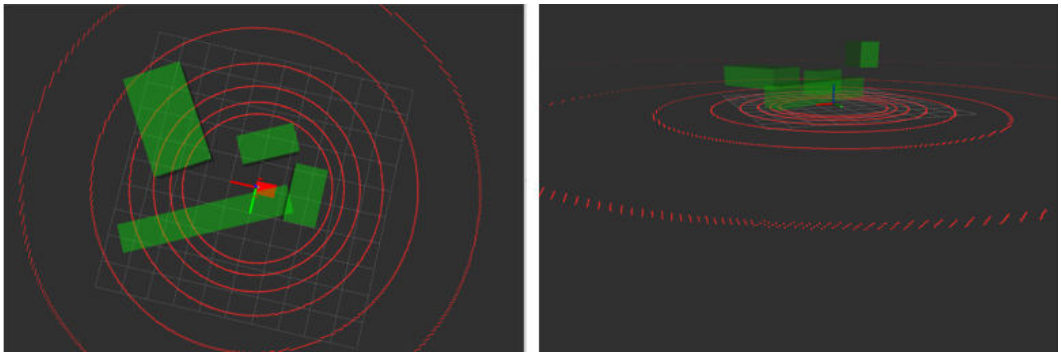
The **Obstacle_Detect** function block uses information from a piece of LiDAR hardware to see if there are objects around the LiDAR.



Obstacle_Detect requires a control device like the DM1000 or XM100, LiDAR scanner, and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about how LiDARs work, Ouster LiDAR hardware, and the **Ouster_LiDAR** function block.

If the LiDAR hardware sees solid objects within its field-of-view, **Obstacle_Detect** determines where the obstacle is located in relation to the LiDAR hardware, and then further code decides how the machine should react to those detected objects. In order to be seen, objects need some amount of reflectivity. Items far away, transparent, or very dark are harder to detect.

To find the location of objects, **Obstacle_Detect** divides physical space around the LiDAR into invisible zone boxes, which are not related to UTM zones. The LiDAR sends out invisible light points which send information back to the LiDAR when they land on something. The points include X, Y, and Z coordinates, which show where the object is physically located and which zone it belongs. Create these zone boxes manually within **Obstacle_Detect**, as well as the code telling the machine how to react.

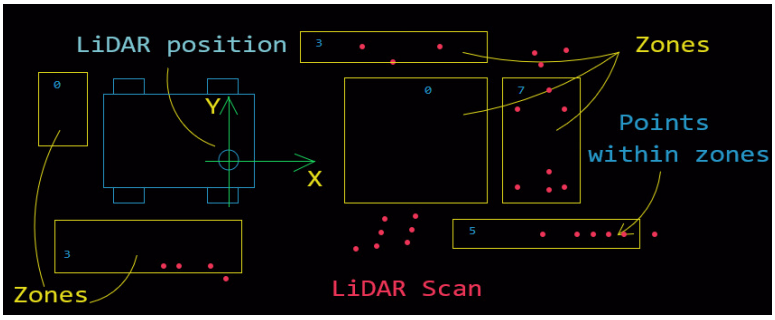


The images show a top and then side view of zones (depicted as green boxes) around a LiDAR (central red object) with spiraling rings to indicate the coordinate points.

If tuning **Obstacle_Detect**, determine the zone parameters correctly or the functionality will not work well. Additionally, make sure the zones and obstacles are within the field-of-view of the LiDAR. If they fall outside what the LiDAR detects, then adjust the hardware and code related to the LiDAR. Zones can overlap, with LiDAR points counting multiple times in the overlapping area.

The internal algorithm takes into consideration the parameters and creates a cuboid zone. Then, all the points in the LiDAR point cloud are checked if they are within the zone or not. Zones can also be at an angle to the LiDAR rather than parallel or perpendicular.

Obstacle_Detect Function Block



The image shows the LiDAR hardware on a machine and a top-down view of the LiDAR points within and outside of six zones. The number of points detected within the zone, known as **Scores** in **Obstacle_Detect**, are labeled in the corner of the zone boxes in this image. Points that are detected both inside and outside the zones are known as the **Total_Valid_Points** in **Obstacle_Detect** and factor into coding algorithms.

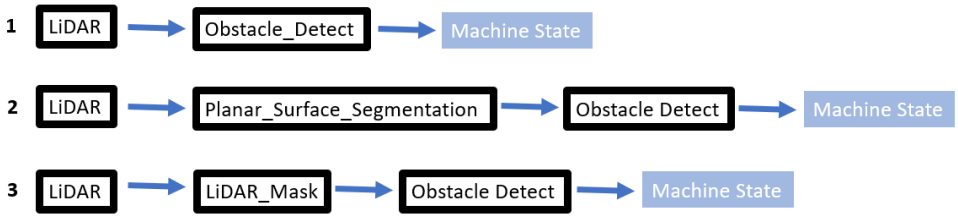
Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Application Information

Common function blocks used with the **Obstacle_Detect** function block are **Data_Lockers**, **Planar_Surface_Segmentation**, and the **Ouster_LiDAR** function block if using Ouster LiDAR hardware.



1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as Ouster LiDAR hardware and the **Ouster_LiDAR** function block. This gathers point cloud data which **Obstacle_Detect** uses. Program the machine to react depending on any objects detected, such as stopping or slowing down.
2. Scenario two includes **Planar_Surface_Segmentation**, which uses point cloud data from the LiDAR to detect a surface such as the ground or a wall. It outputs a divided point cloud which has either the ground or wall removed. This point cloud can then be passed into **Obstacle_Detect** to detect obstacles in a more accurate and computationally efficient way.
3. Scenario three includes the **LiDAR_Mask** function block after the LiDAR code, followed by **Obstacle_Detect**. **LiDAR_Mask** omits certain data from the LiDAR's point cloud, saving processing time. Then, the machine can be programmed to react a certain way based on whether any objects are detected.

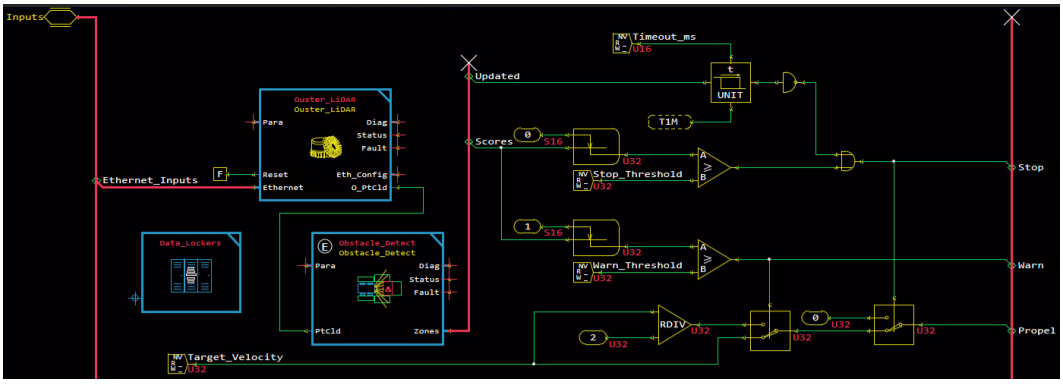
Additionally, place the **Obstacle_Detect** function block:

Obstacle_Detect Function Block

- After LiDAR data is collected, such as after the **Ouster_LiDAR** function block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.

Example

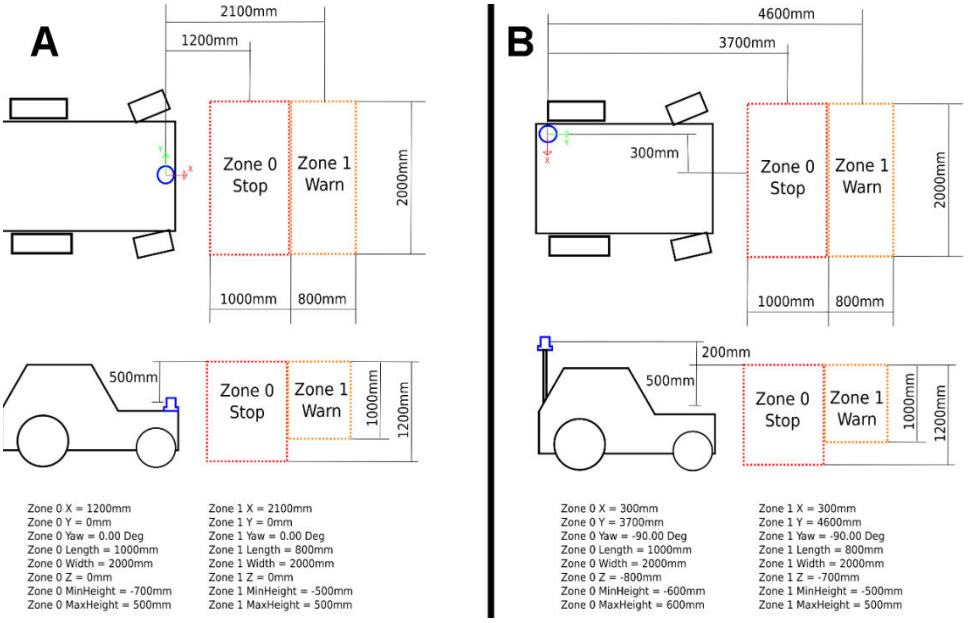
The example shows the **Obstacle_Detect** function block used as if a machine needs to detect something in front of it in order to slow down or stop.



1. Set up the LiDAR hardware and accompanying code. This example includes the **Ouster_LiDAR** function block and Ouster LiDAR hardware. See the *Plus+1 Compliant Ouster Block User Manual* for more information. The LiDAR hardware needs to see where the expected obstacles would be detected and the code set up to include point cloud information from that area. If the LiDAR code is programmed so the range the LiDAR sees is too small, then the code for the LiDAR must be changed in order for **Obstacle_Detect** to work.
2. Add the **Obstacle_Detect** function block. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
3. Outside of PLUS+1® GUIDE, determine physically in the environment where obstacles should be detected and therefore where zones should go. Create zones with respect to the LiDAR sensor acting as the origin, see *Sensor Coordinate System* on page 19. Zones could be a constant area around a moving machine, and in that case the LiDAR likely mounts on the machine. Or, create stagnant zones in the environment, and in that case the LiDAR likely sits in a non-moving place where it sees these zones at all times. Make sure the zone is within the LiDAR's field-of-view and valid range.

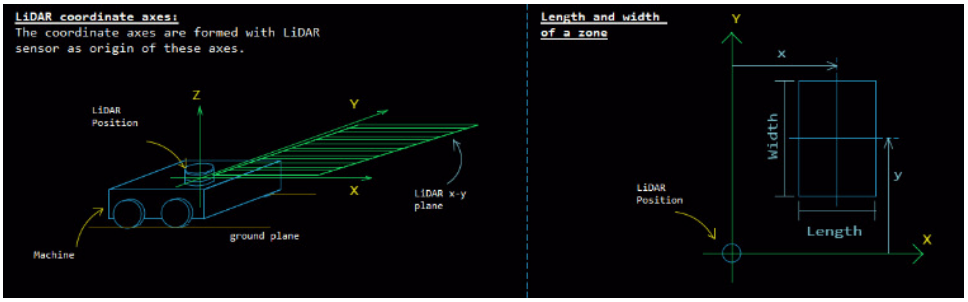
   The example code includes two zones in front of a machine. The first zone appears as 0 and the second zone as 1 in many places in the code because the range begins at 0. Objects detected in the close zone tell the machine to stop, and obstacles detected in a farther away zone warn the machine to slow down.

Obstacle_Detect Function Block



The images above show how the LiDAR on the machine determines the parameters of the zones. Image A includes a top and side view of the LiDAR mounted in the front of the machine. Image B shows the LiDAR moved to the back corner of the machine and mounted at an angle.

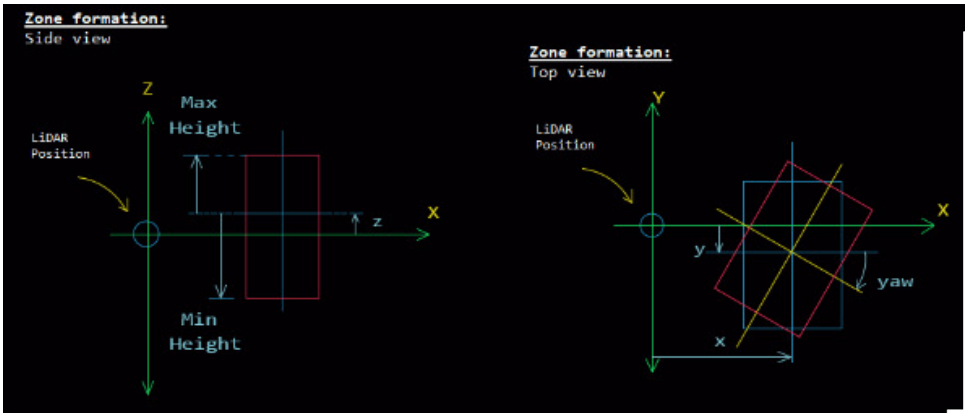
4. Decide the number of zone boxes to enter into **Num_Zones** on the parameters page within **Obstacle_Detect**. The examples above show two zones in both images A and B, which are each labeled Zone 0 and Zone 1. In this case, **Num_Zones** = 2 with the first zone as Zone 0.
5. Find the middle point within the first approximately defined zone box, and measure from there to the LiDAR hardware. This middle of the zone is the X, Y, and Z coordinate. Image A shows Zone 0 with X, Y, Z coordinates as 1200, 0, 0.
   a) Enter the **X** coordinates into the array in the **Obstacle_Detect** function block parameter. The zone number correlates with array values. Image A shows the **X** array as (1200, 2100, 0, ...) because Zone 0 is 1200, and Zone 1 is 2100. Leave all the other zone values in the **X** array as 0 because there are no more zones. Image B shows **X** as (300, 300, 0, …) because the x-axis shifted due to the LiDAR hardware being mounted in a different direction than image A. The center of the zones begin 300 mm in front of that LiDAR scanner.
   b) Enter the **Y**coordinates into the array. **Y** remains zero if the middle of the zone box is directly in front of the LiDAR scanner. Image A shows the **Y** array as (0, 0, 0, …). Image B shows the array as (3700, 4600, 0, …).
   c) Enter the **Z** coordinates into the array. **Z** matches to the same height as the LiDAR scanner if left at zero. Image A shows the **Z** array as (0, 0, 0, …). Image B shows **Z** as (-800, -700, 0, …) because the middle of the two zones are below the LiDAR scanner in different places.
6. From the X, Y, and Z center point, create the zone box by adding in the width and length. When yaw is zero, length is in the x-axis direction, and width is in the y-axis direction. Both images A and B show array **Length** as (1000, 800, 0, …) and **Width** as (2000, 2000, 0, …).



Obstacle_Detect Function Block

The image shows the zone created with respect to the LiDAR hardware as the origin. **Length** follows the x-axis, and **Width** follows the y-axis.

7. Create the zone height dimensions by entering the height above and below the **Z** coordinate. Image A shows **Z** as zero with **Max_Height** array as (500, 500, 0, …) and **Min_Height** array as (-700, -500, 0, …). Image B shows an alternate method of creating the height dimensions by adjusting the **Z** value to the center of the zones. In this case, the zones are below the LiDAR hardware and not zero (-800, -700, 0, …). **Max_Height** and **Min_Height** equal the total height divided by two, which are arrays (600, 500, 0, …) and (-600, -500, 0, …). Sometimes, the machine pitches forward on uneven surfaces, which make the zones detect the ground as an object. Set the minimum height higher off the ground in that case.



The left image shows how **Z** could move above or below the LiDAR origin, as well as how the zone height grows both above and below **Z** rather than the LiDAR. The right image displays a rotated zone using yaw.

8. Determine rotation of the zones or LiDAR hardware, which can be addressed with **Yaw**. Image A shows **Yaw** as zero because the LiDAR scanner is mounted facing forward on the machine, and the examples are taking into account the machine coordinates. Image B shows **Yaw** as -90 degrees to compensate for the LiDAR sensor facing sideways on the machine. In both cases, the resulting zones have the same orientation despite being mounted differently.
9. Create code to react to the obstacles detected within the zones.
   a) In this example, the **Scores** of the stop and warn zones adjust the propel speed of the machine. **Scores** say how many valid points the LiDAR found within each zone and does not include invalid LiDAR points. A larger score indicates a larger object within the zone or closer to the LiDAR hardware.
   b) In a different **Obstacle_Detect_Area** function block, **Areas** replaces **Scores**. **Areas** calculates the approximate cross-sectional area of objects within each zone and adjusts for their closeness to the LiDAR. The threshold numbers in this example would change to show area estimation.
10. Monitor the **Updated** flag to ensure new information comes through. If this stops updating, then the block is not processing data. Optionally, see the *Pre-Made Service Tool Screens* on page 25.

Inputs

The table describes the inputs to the **Obstacle_Detect** function block.

Item	Type	Range	Description [Unit]
PtCld	S8	-1-99	The data locker ID of ordered or unordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Obstacle_Detect Function Block

Parameters

The table describes parameters for the **Obstacle_Detect** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
X	(ARRAY[100]S32)	-2147483648-2147483647	Point on the x-axis which determines the center of the zone(s). X uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Default: zeros (100) [mm]
Y	(ARRAY[100]S32)	-2147483648-2147483647	Point on the y-axis which determines the center of the zone(s). Y uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Default: zeros (100) [mm]
Z	(ARRAY[100]S32)	-2147483648-2147483647	Point on the z-axis which relates to the height of the zone(s). Z uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Height values above this value are positive and height values below this value are negative. Default: zeros (100) [mm]
Yaw	(ARRAY[100]S16)	-18000-18000	Orientation of the zone(s) in the x-y plane. Default: zeros (100) [0.01 deg]
Width	(ARRAY[100]U16)	0-65535	Width of the zone(s). When the **Yaw** parameter is zero, **Width** is in the direction of the y-axis of the LiDAR. Default: 1000 * ones (100) [mm]
Length	(ARRAY[100]U16)	0-65535	Length of the zone(s). When the **Yaw** parameter is zero, **Length** is in the direction of the x-axis of the LiDAR. Default: 1000 * ones (100) [mm]
Num_Zones	U8	0-100	Number of zone boxes created to find obstacles. Leaving **Num_Zones** as 0 means there will be no zones. This saves processing power when **Obstacle_Detect** is not used. Default: 0
Min_Height	(ARRAY[100]S32)	-50000 to **Max_Height**-1	Minimum height of the zone(s) with respect to the **Z** parameter. Default: zeros (100) [mm]
Max_Height	(ARRAY[100]S32)	**Min_Height** +1 to 50000	Maximum height of the zone(s) with respect to the **Z** parameter. Default: 1000 * ones (100) [mm]

Outputs

The table describes outputs of the **Obstacle_Detect** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.

Obstacle_Detect Function Block

Item	Type	Range	Description [Unit]
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Zones	BUS	——	Contains the updated information for the zone boxes.
Updated	BOOL	T/F	New information is available from the block. T: New point cloud data has been processed. F: No new point cloud data processed.
Scores	(Array[100]S32)	0-4294967295	The number of valid LiDAR points within each zone box.
Total_Valid_Points	U32	0-4294967295	The number of valid LiDAR points. These obtain data when landing on objects. An unusually low number may indicate issues. Invalid points include LiDAR points going into the sky or dark surfaces, which are not detected by the LiDAR. For ordered point clouds, this is the number of valid LiDAR points found within and close to the zones. LiDAR points are counted multiple times if zones overlap. For unordered point clouds, this is the number of valid LiDAR points the LiDAR sees in the whole point cloud, regardless of zones.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Obstacle_Detect Troubleshooting

The following table describes errors that could occur in the **Obstacle_Detect** function block and ways to fix them.

View the **Obstacle_Detect_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Obstacle_Detect_Err Descriptions and Fixes

Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
2	Not enough memory available to create thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
3	Thread timeout.	There may be too much code creating a longer processing time. Reduce the LiDAR resolution or delete other processing blocks. See *Reduce Processing Time* on page 174. Turn the XM100 off, wait a bit, and restart it.

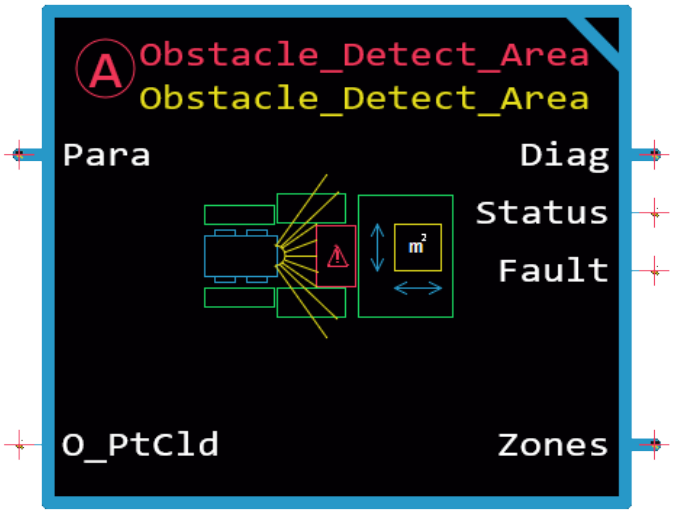
Obstacle_Detect Function Block

Other Errors and Fixes

Error Description	How to Fix
Total_Valid_Points numbers are very low. These point numbers vary, but they are based on the type of LiDAR. A high resolution LiDAR is expected to get thousands of points, so a low number of 10 could indicate issues.	Check the LiDAR hardware to make sure nothing is blocking the scanner, including dirt. See the LiDAR manufacturer's instructions for debugging hardware issues.
Zones do not pick up objects.	Check the parameters for the zones and adjust the numbers. Review the coordinates of the LiDAR. Test the edges of the zones by moving an object into and out of the zones.
The machine mistakenly detects an object in the zone or the ground as an object.	Adjust the zone height so **Min_Height** begins higher from the ground. The LiDAR could detect the ground as an object if it pitches forward while the machine travels over bumpy ground.

Obstacle_Detect_Area Function Block

The **Obstacle_Detect_Area** function block uses a LiDAR to detect and estimate the size of objects inside of a specified region.



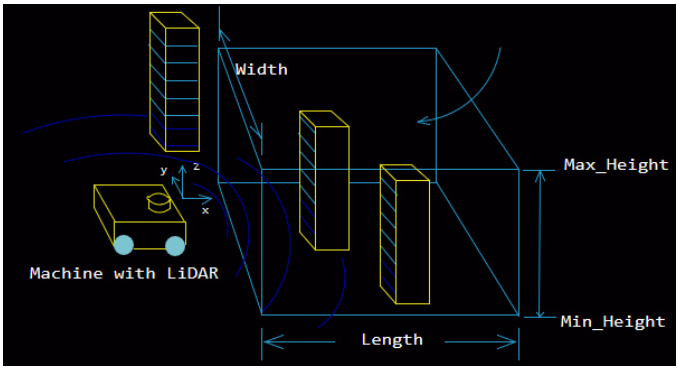
This block requires a license for A+ Advanced.

Obstacle_Detect_Area requires a control device like the DM1000 or XM100, LiDAR scanner, and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about how LiDARs work, Ouster LiDAR hardware, and the **Ouster_LiDAR** function block.

Obstacle_Detect_Area behaves very similarly to the **Obstacle_Detect** function block. See *Obstacle_Detect Function Block* on page 112 to set up zone boxes around the LiDAR.

A major difference is that **Obstacle_Detect_Area** determines the approximate cross-sectional area of items inside of the zones around the LiDAR, whereas the simpler **Obstacle_Detect** only detects when an object is in the zone, not the size.

Area estimation depends on the number of LiDAR points landing on objects, so neither the surface area nor the exact cross-sectional area can be determined. However, the block calculations adjust for objects far away or close to the LiDAR to give a rough size estimate. For example, if a bird flies close to the LiDAR, lots of LiDAR points land on it, but the block knows it is still small. A building in the distance with a few LiDAR points still reads as a large object. This allows fewer, larger zones to get detailed information instead of many smaller zones. If there are multiple objects within a zone, the **Areas** output combines them all.



The image above shows a LiDAR on a machine scanning objects inside of a zone box. The LiDAR detects the approximate size of objects inside of the zone based on the LiDAR points landing on them.

Reflective items around the LiDAR could reduce its ability to estimate the object's cross-sectional area.

Additionally, **Obstacle_Detect_Area** only takes in ordered point cloud data and not unordered. However, **Obstacle_Detect** supports both ordered and unordered point cloud data.

Obstacle_Detect_Area Function Block

Application Information

The **Obstacle_Detect_Area** function block behaves similarly to the **Obstacle_Detect** function block, except it gives a rough estimate of the cross-sectional area of items inside of the zone rather than just detecting that something is in a zone.

Review the *Application Information* on page 113 for **Obstacle_Detect**. The **Obstacle_Detect_Area** function block can be substituted in each scenario.

Example

The **Obstacle_Detect_Area** function block can be set up in a similar way as the **Obstacle_Detect** function block.

Review the *Example* on page 114 where **Obstacle_Detect** is used as if a machine needs to slow down or stop when anything is found in the zones around it. In addition, **Obstacle_Detect_Area** would give the approximate cross-sectional area of all items within each zone in the **Areas** output instead of **Scores**.

Inputs

The table describes the inputs to the **Obstacle_Detect_Area** function block.

Item	Type	Range	Description [Unit]
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The table describes parameters for the **Obstacle_Detect_Area** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
X	(ARRAY[100]S32)	-2147483648-2147483647	Point on the x-axis which determines the center of the zone(s). X uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Default: zeros (100) [mm]
Y	(ARRAY[100]S32)	-2147483648-2147483647	Point on the y-axis which determines the center of the zone(s). Y uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Default: zeros (100) [mm]
Z	(ARRAY[100]S32)	-2147483648-2147483647	Point on the z-axis which relates to the height of the zone(s). Z uses the Cartesian coordinate system with respect to the LiDAR hardware as the origin. Height values above this value are positive and height values below this value are negative. Default: zeros (100) [mm]
Yaw	(ARRAY[100]S16)	-18000-18000	Orientation of the zone(s) in the x-y plane. Default: zeros (100) [0.01 deg]
Width	(ARRAY[100]U16)	0-65535	Width of the zone(s). When the **Yaw** parameter is zero, **Width** is in the direction of the y-axis of the LiDAR. Default: 1000 * ones (100) [mm]

Obstacle_Detect_Area Function Block

Item	Type	Range	Description [Unit]
Length	(ARRAY[100]U16)	0-65535	Length of the zone(s). When the **Yaw** parameter is zero, **Length** is in the direction of the x-axis of the LiDAR. Default: 1000 * ones (100) [mm]
Num_Zones	U8	0-100	Number of zone boxes created to find obstacles. Leaving **Num_Zones** as 0 means there will be no zones. This saves processing power when **Obstacle_Detect_Area** is not used. Default: 0
Min_Height	(ARRAY[100]S32)	-50000 to **Max_Height**-1	Minimum height of the zone(s) with respect to the **Z** parameter. Default: zeros (100) [mm]
Max_Height	(ARRAY[100]S32)	**Min_Height** +1 to 50000	Maximum height of the zone(s) with respect to the **Z** parameter. Default: 1000 * ones (100) [mm]

Outputs

The table describes outputs of the **Obstacle_Detect_Area** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Zones	BUS	——	Contains the updated information for the zone boxes.
Updated	BOOL	T/F	New information is available from the block. T: New point cloud data has been processed. F: No new point cloud data processed.
Areas	(Array[100]U 32)	0-4294967295	The approximate cross-sectional area of objects within a zone. [mm^2]
Total_Valid_Points	U32	0-4294967295	The number of valid LiDAR points. These obtain data when landing on objects. An unusually low number may indicate issues. Invalid points include LiDAR points going into the sky or dark surfaces, which are not detected by the LiDAR. For ordered point clouds, this is the number of valid LiDAR points found within and close to the zones. LiDAR points are counted multiple times if zones overlap.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Obstacle_Detect_Area Troubleshooting

The following table describes errors that could occur in the **Obstacle_Detect_Area** function block and ways to fix them.

View the **Obstacle_Detect_Area_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Obstacle_Detect_Area Function Block

Obstacle_Detect_Area_Err Descriptions and Fixes

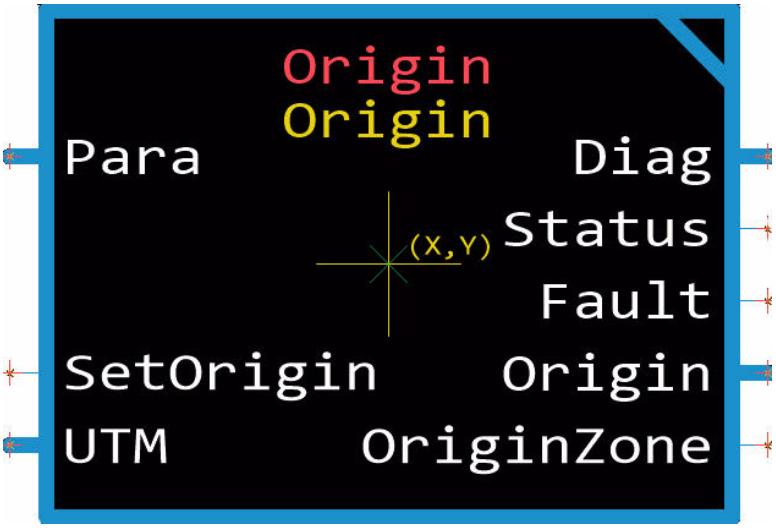
Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
2	Not enough memory available to create thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
3	Thread timeout.	There may be too much code creating a longer processing time. Reduce the LiDAR resolution or delete other processing blocks. See *Reduce Processing Time* on page 174. Turn the XM100 off, wait a bit, and restart it.
4	Point cloud is unordered. This means the point cloud data entering the block is unordered instead of ordered. Ordered LiDAR points include X, Y, and Z coordinates, whereas unordered LiDAR points have no coordinate data.	Change the point cloud data to ordered by reviewing previous code that caused it to become unordered. If unordered data is needed, use another block that does not need ordered point cloud data, such as **Obstacle_Detect**.
5	Point cloud has invalid size. This means that the point cloud dimension is so small it may only have one row or column.	Expand the parameters in previous blocks to have a larger point cloud. Look especially in filtering blocks or the LiDAR code.

Other Errors and Fixes

Error Description	How to Fix
Total_Valid_Points numbers are very low. These point numbers vary, but they are based on the type of LiDAR. A high resolution LiDAR is expected to get thousands of points, so a low number of 10 could indicate issues.	Check the LiDAR hardware to make sure nothing is blocking the scanner, including dirt. See the LiDAR manufacturer's instructions for debugging hardware issues.
Zones do not pick up objects.	Check the parameters for the zones and adjust the numbers. Review the coordinates of the LiDAR. Test the edges of the zones by moving an object into and out of the zones.
The **Areas** output seems too small for the situation.	If an object has a reflector close to or on it, then this block may report a smaller cross-sectional area than it should. Move or adjust the reflector. Try different LiDARs or adjust the LiDAR settings. Or, account for the reduced **Areas** values in other code.
The machine mistakenly detects an object in the zone or the ground as an object.	Adjust the zone height so **Min_Height** begins higher from the ground. The LiDAR could detect the ground as an object if it pitches forward while the machine travels over bumpy ground.

Origin Function Block

The **Origin** function block stores UTM coordinates of the machine's starting point, and uses this data to calculate the relative position of the autonomous machine as it operates.



The origin can be set on startup. It can also be updated during application runtime, which can be useful for repetitive algorithms, such as path coverage.

It is recommended to delay setting the origin location until valid position data (GNSS location) has resolved to an accurate position. Save the origin to non-volatile memory to keep the origin between power cycles.

Inputs

Inputs to the **Origin** function block are described.

Item	Type	Range	Description [Unit]
SetOrigin	BOOL	T/F	When true, the function block stores the current origin in the Para or UTM bus depending on the **CustomOrigin**. T: Update the origin being stored. F: Origin being stored stays the same.
UTM	BUS	——	GNSS data using the UTM coordinate system.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The band that the UtmX and UtmY values are in. ASCII values represent the letter of the band.
Zone	U8	1-60	The zone that the UtmX and UtmY values are in.

Origin Function Block

Item	Type	Range	Description [Unit]
Updated	BOOL	T/F	True when there is new data. The stored values are updated only if no data has been stored yet. **CustomOrigin** is False and **Updated** turns True. T: New data is ready. F: New data is not ready.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The **Origin** function block's operating characteristics are set by para bus input signals.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. Default: 0x20EBC948 [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value. Default: 0x00000001
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value. Default: 0x6B7EAF74
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters. Default: 85 Unit: NA
Zone	U8	1-60	The UTM zone where the UtmX and UtmY values are. Default: 32 Unit: NA
CustomOrigin	BOOL	T/F	Uses the custom origin values specified here in the Para bus when True. T: Uses values in Para BUS. F: Use values from the UTM BUS. Default: True
Updated	BOOL	T/F	True when there is new data. The stored values are updated only if no data has been stored yet. **CustomOrigin** is True and **Updated** turns True. T: New data is ready. F: New data is not ready. Default: True

Origin Function Block

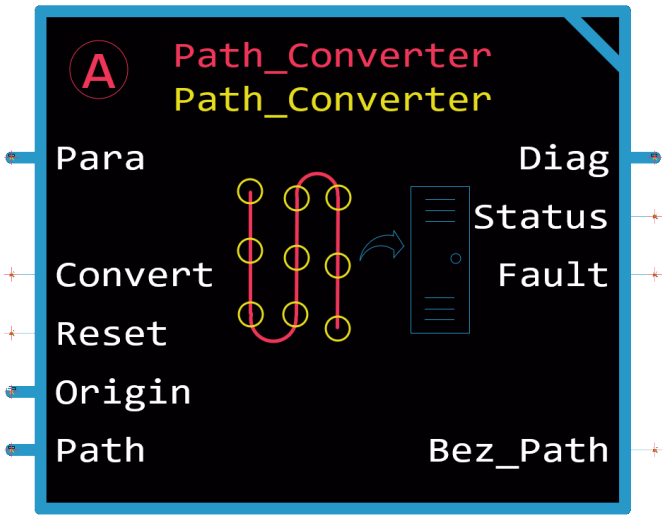
Outputs

Outputs of the **Origin** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high. 0x8010: Input value is out of range.
Origin	BUS	——	Stores the UTM values for the origin.
UtmX	U32	$0\text{-}10^{9}$	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	$0\text{-}10^{10}$	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The band that the UtmX and UtmY values are in. ASCII values represent the letter of the band.
Zone	U8	1-60	The UTM zone where the UtmX and UtmY values are.
Updated	BOOL	T/F	True when new data is being stored for the origin. T: New data is stored. F: The origin has not changed.
OriginZone	U8	1-60	The zone that the origin UTM is in. This is the same value as the zone in the Origin BUS.

Path_Converter Function Block

The **Path_Converter** function block takes in data about a path and passes it to a data locker, in order for other path blocks to access that data.



This block requires a license for A+ Advanced.

Path_Converter allows information about a machine's path to be entered directly into PLUS+1® GUIDE, rather than driving a machine manually over a path to record the path information. Manually driving a machine to record the path requires the **Path_Recorder** function block, and usually an application does not need both types of blocks.

Path_Converter reads the parameters, origin, and information about the path a machine needs to follow in the form of Bezier curves. It writes the path information into a data locker when the **Convert** pulse is given. **Path_Follower_Adv** and **Path_Extract** function blocks use the converted information from the data locker.

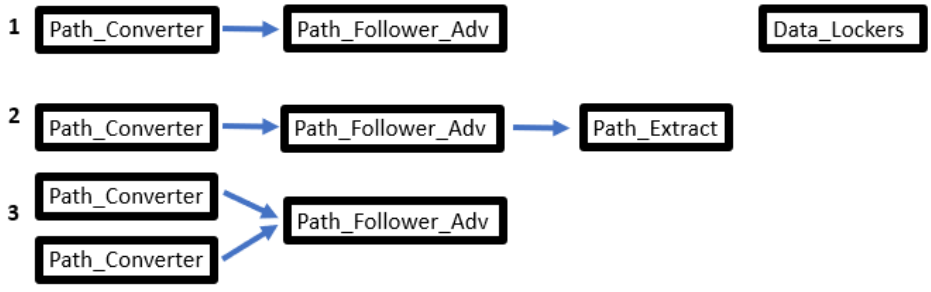
During path conversion and processing the converted path, the block might use a significant amount of controller memory. It is recommended to have the machine standing still during the path conversion.

When using **Path_Converter**, the velocity sign must always stay positive or negative, and it must match the driving direction where positive velocity means forward, and negative velocity means backwards. Velocity could also be zero. If the machine needs to move both backwards and forwards, create multiple paths. In that case, the path needs to reset before converting the path data again.

Application Information

Common function blocks that work with **Path_Converter** are **Data_Lockers**, **Path_Follower_Adv**, and **Path_Extract**.

Path_Converter allows known path data to be hard-coded into a PLUS+1® GUIDE application, rather than the machine physically driving a path and recording the data. Information about positions on the given origin is required earlier in the application, as well as a **Data_Lockers** block. Some basic path function block combinations include:



Path_Converter Function Block

1. Scenario one shows data entered into **Path_Converter** flowing into **Path_Follower_Adv** for the machine to follow the path. Data lockers do not hold data over power cycles, so if a machine is powered off, data must be converted again. One **Data_Lockers** block is required for all applications but does not connect to anything.
2. Scenario two shows data entered into **Path_Converter** flowing into **Path_Follower_Adv** for the machine to follow the path. Because the path is already known going into **Path_Converter**, it is not necessary to place **Path_Extract** directly after it. Instead, place it after **Path_Follower_Adv** to visually show trajectory data on a service tool screen or a display like a DM1000.
3. Scenario three shows two paths. Each path could have a **Path_Converter** block but need only one **Path_Follower_Adv** block. Alternately, hard-code multiple paths into one **Path_Converter** where a new path loop begins after **Path_Follower_Adv** processes the current path. **Path_Follower_Adv** reads only one **Path_Converter** at a time.

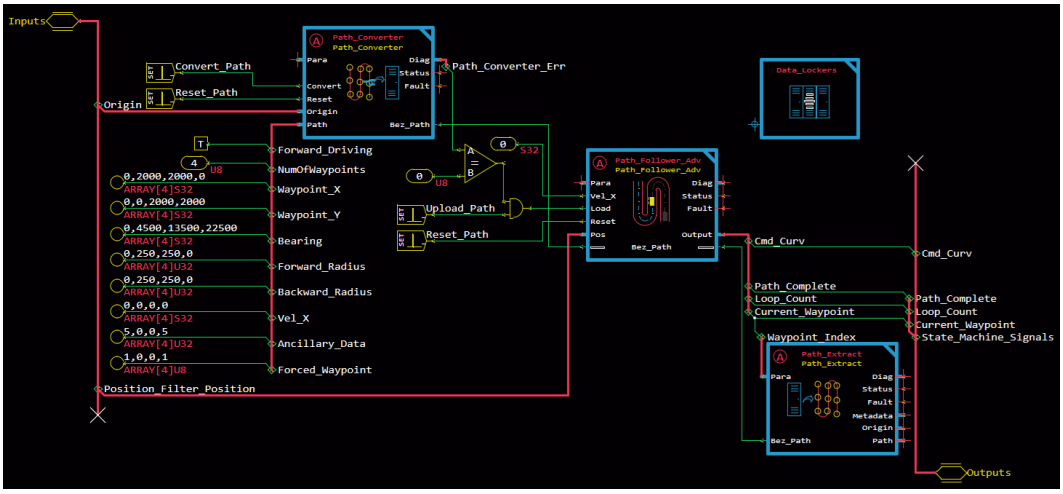
   If switching several paths into one **Path_Converter**, program **Path_Converter** to convert to load the new path. **Convert** in **Path_Converter** and **Load** in **Path_Follower_Adv** can occur in the same loop if **Path_Follower_Adv** executes after **Path_Converter**.

Additionally, place the **Path_Converter** function block:

- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- After position and yaw information is obtained, such as after a **Position_Filter** function block.
- One or more times in an application if there are multiple paths. Switching several paths into one **Path_Converter** saves processing power. A machine needs a new path every time its velocity changes direction.

Example

The example shows the **Path_Converter** function block generating a predetermined path that the **Path_Follower_Adv** then uses.



The example assumes code exists earlier in the application to establish a machine's position and origin. See an overview of steps and explanations about what they achieve:

1. Add the **Path_Converter** and **Path_Follower_Adv** function blocks. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
2. Create a pulse to convert the path data to a data locker. This connects directly to **Convert** to convert the data.
3. Create a set pulse for **Reset** on **Path_Converter**. If **Reset** and **Convert** are both True in the same program loop, the program only performs the **Reset**.
4. Connect the **Origin** to **Path_Converter**. The origin information comes from earlier in the application and ties the machine's position to a local coordinate frame. Optionally, hard-code origin information.

Path_Converter Function Block

5. Fill out the metadata about the path in the parameters bus, as well as the **Path** data. If there's a desire to record a machine physically driving the path instead of putting numbers into PLUS+1® GUIDE, then use the **Path_Recorder** function block instead of **Path_Converter**.
6. Check that **Path_Converter** had no errors when loading data to **Path_Follower_Adv**. If no errors occur, trigger the **Load** signal on **Path_Follower_Adv** to take in the path information from a data locker.

   If using multiple paths in **Path_Converter**:

   a. Trigger **Convert** on **Path_Converter** for the first path.
   b. Trigger **Load** on **Path_Follower_Adv** for the first path. This can happen in the same loop as triggering **Convert** as long as **Path_Follower_Adv** executes after **Path_Converter**.
   c. Ensure the first path completed without issues by looking at the **Path_Complete** and **State** signals in the **Path_Follower_Adv** function block. Fix any issues that might occur with **Path_Converter** before converting the next path.
   d. Repeat these steps for all other paths.
7. Connect a **Set Pulse** component to **Reset** on **Path_Follower_Adv**. This signal triggers the block to go into a safe state and overrides the **Load** signal.
8. Connect **Bez_Path** to **Path_Follower_Adv**. The data flows to the **Data_Lockers** block automatically while going between these two blocks.
9. Connect a **Constant** value to the **Vel_X** input to set the machine velocity manually, or use other data to input the velocity of the machine. In this example, the **Vel_X** input is set to 0.
10. Connect position to **Path_Follower_Adv**, which uses the **Position_Filter** block or other code that sends position information.
11. Create any desired **Output** data from **Path_Follower_Adv**. This example includes common data a state machine uses, such as:
    a) Notifying when the machine finishes the path in **Path_Complete**.
    b) Recording how many times the machine followed the path in **Loop_Count**.
    c) Displaying information about which waypoint the machine just drove through in **Current_Waypoint**.
12. Connect **Cmd_Curv** to the machine's steering control system and **Cmd_Vel** to the machine's propel control system. **Path_Follower_Adv** calculates curvature commands for the machine to use to stay on the path.
13. Connect **Path_Extract** to visually see data about the path. Here, **Path_Extract** is connected so it can look at the future path. See each function blocks' *Pre-Made Service Tool Screens* on page 25.

Inputs

The following table describes inputs required for the **Path_Converter** function block.

Array range for X in the ARRAY[X] types should be between 2 to 32,767. X is dynamic.

Item	Type	Range	Description [Unit]
Convert	BOOL	T/F	False to True transition starts the conversion of path arrays to a path type data locker. T: The block state changes back to idle and checks for errors. If driving direction and velocity sign match, the block goes into a read state from idle. F: If there is a path in the path type data locker, then the block keeps updating the data locker. If there is not a path in the data locker, then the block stays in an idle state. If **Reset** and **Convert** are both True in the same program loop, the program only performs the **Reset**.
Reset	BOOL	T/F	False to True transition determines whether to stop the conversion after the path is written to the data locker. T: The block state changes back to idle, and no more data goes into a path type data locker. F: If there is a path in the path type data locker, then the block keeps updating the data locker. If there is not a path in the data locker, then the block stays in an idle state.

Path_Converter Function Block

Item	Type	Range	Description [Unit]
Origin	BUS	——	BUS containing UTM values of the path's origin. The data flows automatically to the path type data locker. The items in this bus are placeholders and do not do anything.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin.
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin.
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	0x00000000-0x540BE400 This is the range of the full U64 bit number.	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in. Default: 32
Path	BUS	——	A bus that contains options for the path and a way to define the path for the machine to follow.
NumOfWaypoints	U16	2-32767	The desired number of waypoints to be written into the data locker. This takes affect when **Convert** transitions from False to True. For optimal performance, limit the path lengths to less than 25000 waypoints.
Waypoint_X	ARRAY[X]S32	-2147483648-2147483647	The X position of the waypoint. [mm]
Waypoint_Y	ARRAY[X]S32	-2147483648-2147483647	The Y position of the waypoint. [mm]
Backward_Radius	ARRAY[X]U32	0-4294967295	Distance from the waypoint to the backward control point. Smaller radii yield sharper turns. [mm]
Forward_Radius	ARRAY[X]U32	0-4294967295	Distance from the waypoint to the forward control point. Smaller radii yield sharper turns. [mm]
Bearing	ARRAY[X]S32	-72000-72000	Angle at which the machine goes through the waypoint. This uses the ENU convention and the right-hand rule. [0.01 degree]
Vel_X	ARRAY[X]S32	-2147483648-2147483647	Array of machine velocity for each waypoint. This is the desired velocity and may not indicate the actual velocity while the machine moves. [mm/s]
Ancillary_Data	ARRAY[X]U32	0-4294967295	Array of extra information attached to a specific waypoint. For example, this could be information about temperature, the state of the machine, or directions to stop.
Forced_Waypoint	ARRAY[X]U8	0-1	Indicates if a waypoint was forced in a path. 0: Waypoint is not forced. 1: Waypoint is forced.
Forward_Driving	BOOL	T/F	Indicates whether the machine drives forward or reverse. T: The machine drives in the direction of the machine face. F: The machine drives in the direction opposite to the machine face (negative velocity). For example, the machine starts at the beginning of the path and drives facing backwards to the end of the path.

Path_Converter Function Block

Parameters

The following table describes parameters for the **Path_Converter** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less.
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. This is data for the waypoints in the path recording.
Ancillary_Caption	STRING[255]	——	Describes the data in the **Ancillary_Data** signal.

Outputs

The following table describes outputs for the **Path_Converter** function block. The data could go into the **Path_Extract** and **Path_Follower_Adv** function blocks.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Bez_Path	S8	-1-99	Defines the ID of the path type data locker.

Internal Signals

The following table describes what is happening internally in the **Path_Converter** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Path_Converter_Err	U8	0-4	Indicates when an error occurred in the block functionality. See *Path_Converter Troubleshooting* on page 131.
Path_Converter_State	U8	0-2	The state of the **Path_Converter** function block. 0: Idle, waiting for the convert signal. 1: Read the input and write it to the path type data locker. 2: Conversion in progress.

Path_Converter Troubleshooting

The following table describes errors that could occur in the **Path_Converter** function block and ways to fix them.

View the **Path_Converter_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

If your PLUS+1® application does not compile, times out, or stalls, it may be due to using large constant arrays of waypoints. Try disabling the **Checkpoints** page for **Path_Converter**.

Path_Converter Function Block

Path_Converter_Err Descriptions and Fixes

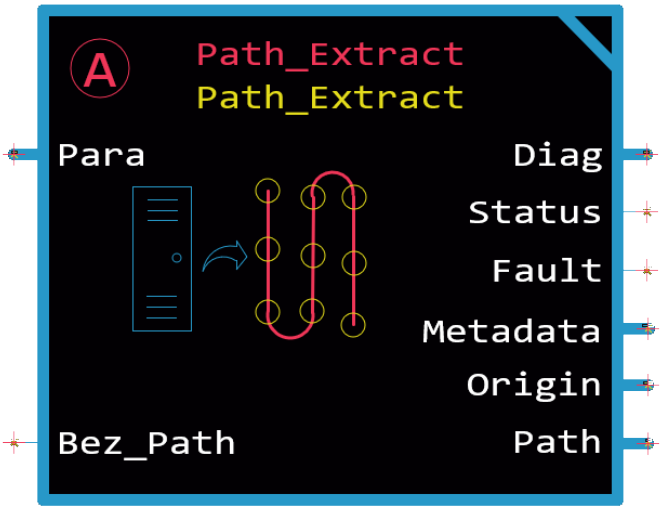
Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	The velocity sign changed direction from positive to negative or vice versa. Array numbers must be all positive or all negative, not mixed. Zero works for either positive or negative.	Verify the velocity sign is constant and does not switch between negative and positive.
2	Wrong driving direction. The velocity sign must be constant and match the **Forward_Driving** signal.	Verify that if **Forward_Driving** is true, velocity is positive or zero. If false, velocity is negative or zero.
3	Input array sizes are not equal in length.	Check that array inputs have the same length.
4	The signal **NumOfWaypoints** is larger than the input arrays, creating the wrong number of waypoints.	Verify the size of input arrays is less than the expected **NumOfWaypoints** value.

Turn off checkpoints if the application does not compile.

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss* on page 29.

Path_Extract Function Block

The **Path_Extract** function block reads Bezier path information from a data locker and displays up to 50 waypoints of data about a path.



This block requires a license for A+ Advanced.

This block shows data from a data locker, which could be visually seen on a service tool screen. Use the pre-made service tool screen, or pull the signals into a service tool individually. The data extracted could be used in other blocks or parts of the application. Additionally, create a display of the information on a piece of hardware such as the DM1000.

Out of the path blocks, **Path_Extract** helps but is not required to record a path or have a machine follow it.

Place this block after path information is gathered, which could be from **Path_Recorder**, **Path_Converter**, **Path_Loader**, or **Path_Follower_Adv** function blocks. Confirm a **Data_Lockers** block exists somewhere in the application.

The **Updated** flag stays true as long as there is valid data in the data locker. It does not change if new data is available. It only goes false if invalid or no data comes.

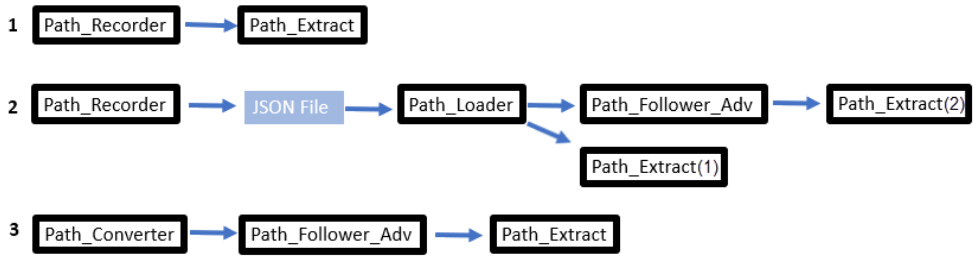
Positioning for this block is measured with a default assuming the GNSS antenna is mounted at the steering point of the machine. For example, this could be the rear axle of a front wheel steer Ackermann machine. If the antenna on the machine is located in a different area, code needs to be written to translate it to the steering point. The path waypoints correlate with the front wheels and may not match the GNSS during turns.

Application Information

Common function blocks used with **Path_Extract** are **Path_Converter**, **Path_Recorder**, **Path_Loader**, **Data_Lockers**, and **Path_Follower_Adv**.

The **Path_Extract** function block visually displays data about the path on a service tool screen or a hardware display, such as the DM1000. It is helpful but not necessary in an application. Some basic path function block combinations include:

Path_Extract Function Block



1. Scenario 1 shows **Path_Recorder** recording the path. **Path_Extract** displays information from **Path_Recorder** onto a Service Tool screen or other hardware display.
2. Scenario 2 shows a **Path_Recorder** function block recording the path then writing it to a JSON file. Then, **Path_Loader** reads the path from the JSON file and transmits it to a Service Tool screen or a hardware display using the **Path_Extract(1)** function block. **Path_Follower_Adv** calculates the remaining path then transmits it to a Service Tool screen or a hardware display using the **Path_Extract(2)** function block.
3. Scenario 3 shows data entered into **Path_Converter** flowing into **Path_Follower_Adv** for the machine to follow the path. Because the path is already known going into **Path_Converter**, it is not necessary to place **Path_Extract** directly after it. Instead, place it after **Path_Follower_Adv** to show what part of the path is left to drive. This appears on a service tool screen or display piece of hardware.

Additionally, place the **Path_Extract** function block:

- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- After any path function block to visually see what information those blocks are putting into a data locker. Many **Path_Extract** blocks could exist in an application. If multiple **Path_Loader** and **Path_Converter** function blocks exist, place **Path_Extract** after each one to see what is in each block.
- Do not place **Path_Extract** at the start of the path block flow because there will not be any data to see. It is usually unnecessary for **Path_Extract** to come directly after **Path_Converter**.

Example

The **Path_Extract** function block is in several examples with other path blocks.

See *Example* on page 157 with the **Path_Recorder** function block.

See *Example* on page 128 with the **Path_Converter** function block.

See *Example - One Path* on page 150 with one **Path_Loader** function block.

See *Example - Multiple Paths* on page 151 with multiple **Path_Loader** function blocks.

All the examples include the **Path_Follower_Adv** function block. Each function block includes *Pre-Made Service Tool Screens* on page 25.

Inputs

The following table describes inputs required for the **Path_Extract** function block. Most of this data comes from the **Path_Loader** and **Path_Recorder** function blocks.

Item	Type	Range	Description [Unit]
Bez_Path	S8	-1-99	Defines the ID of the path type data locker.

Path_Extract Function Block

Parameters

The following table describes parameters for the **Path_Extract** function block.

Item	Type	Range	Description [Unit]
Waypoint_Index	U16	0-65535	The array index of the first waypoint read from the path. For example, to start the array index at waypoint 15, enter 15 here.

Outputs

The following table describes outputs for the **Path_Extract** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Metadata	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less.
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. Date and time information comes from **Path Recorder** or **Path Converter** function blocks.
Ancillary_Caption	STRING[255]	——	Describes the data in the **Ancillary_Data** signal.
Path	BUS	——	A bus that contains options for the path and a way to define the path for the machine to follow.
Updated	BOOL	T/F	Indicates if new path data is extracted from the **Bez_Path** input. T: New path data is extracted. This stays True even if the **Bez_Path** data changes, as long as it is still valid. F: No new path data available.
Forward_Driving	BOOL	T/F	Indicates whether the machine drives forward or reverse. T: Machine drives in the direction of the machine face. F: The machine drives in the direction opposite to the machine face (negative velocity). For example, the machine starts at the beginning of the path and drives facing backwards to the end of the path.
NumOfWaypoints	U16	0-50	The number of waypoints in the output array.
Waypoint_X	(ARRAY[X]S32)	-2147483648-2147483647	The X position of the waypoint. [mm]
Waypoint_Y	(ARRAY[X]S32)	-2147483648-2147483647	The Y position of the waypoint. [mm]
Backward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the backward control point. Smaller radii yield sharper turns. [mm]
Forward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the forward control point. Smaller radii yield sharper turns. [mm]
Bearing	(ARRAY[50]S32)	-72000-72000	Angle at which the machine goes through the waypoint. This uses the ENU convention and the right-hand rule. [0.01 degree]
Vel_X	(ARRAY[50]S32)	-2147483648-2147483647	Array of machine velocity for each waypoint.

Path_Extract Function Block

Item	Type	Range	Description [Unit]
Ancillary_Data	(ARRAY[X]U32)	0-4294967295	Array of extra information attached to a specific waypoint. For example, this could be information about temperature, the state of the machine, or directions to stop.
Forced_Waypoint	(ARRAY[50]U8)	0-1	Indicates if a waypoint is forced in a path. 0: Waypoint is not forced. 1: Waypoint is forced.
Origin	BUS	——	BUS containing UTM values of the path's origin.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin.
UtmY	U32	0-10^{10}	The UTM Northing (Y) value of the origin. This uses two U32 types, equivalent to a U64.
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	0x00000000-0x540BE400 This is the range of the full U64 bit number.	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.

Internal Signals

The following table describes what is happening internally in the **Path_Extract** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Path_Extract_Err	U8	0-2	Indicates when an error occurred in the block functionality. See *Path_Extract Troubleshooting* on page 136.
Total_Num_Waypoints	U16	0-65535	Total number of waypoints stored inside of the data locker about the path. Only 50 waypoints are loaded at a time, so this allows visualization to how many waypoints exist and when the end of the path is reached.

Path_Extract Troubleshooting

The following table describes errors that could occur in the **Path_Extract** function block and ways to fix them. View the **Path_Extract_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

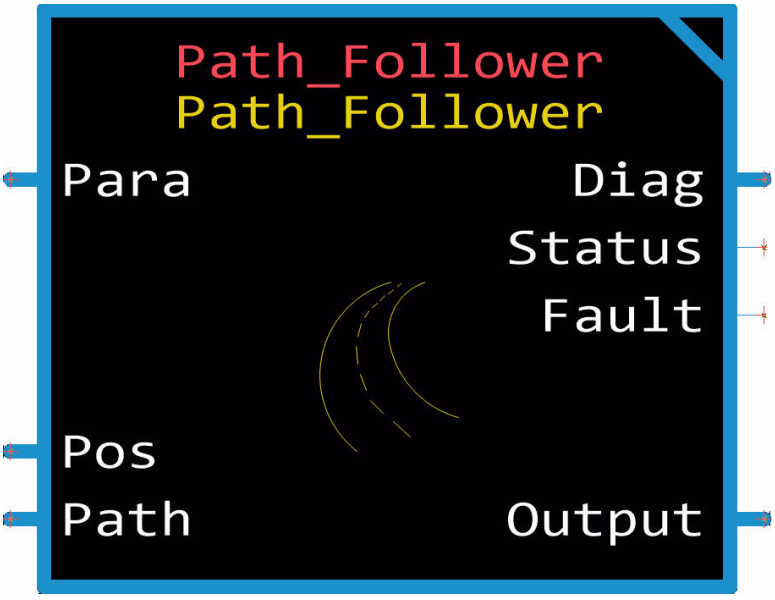
Path_Extract_Err Descriptions and Fixes

Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	The string in the metadata is longer than expected and invalid. The **Data_Lockers** block does not have enough room for the metadata.	Confirm each metadata string is less than 255 characters.
2	The **Waypoint_Index** value is greater or equal in length to the path, making the index invalid.	Enter a valid **Waypoint_Index** signal.

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss* on page 29.

Path_Follower Function Block

The **Path_Follower** function block compares the current machine position to the desired path and provides a steering curvature command to bring the machine to the path.



Path_Follower can be used on XM100 or MC controllers, add sections of paths onto a current path, and does not require a LiDAR.

Paths in **Path_Follower** are made up of one or more connected Bézier curves, which connect the waypoints to make the path. See *Navigation* on page 21 for information about Bézier curves.

By using Bézier curves, the number of points used to define the path is reduced. This lowers computation time and needed memory, while retaining the ability to navigate a complex path.

Positioning for this block is measured with a default assuming the GNSS antenna is mounted at the steering point of the machine. For example, this could be the rear axle of a front wheel steer Ackermann machine. If the antenna on the machine is located in a different area, code needs to be written to translate it to the steering point. The path waypoints correlate with the front wheels and may not match the GNSS during turns.

Path_Follower uses a configurable look-ahead distance that determines how the machine calculates its steering correction commands to reach the intended path. It also allows for looping and non-looping paths and contains a search feature to locate the nearest point on the path when the machine is not on the path.

Paths can be loaded as a fixed path when the machine starts or can be added dynamically.
Path_Follower outputs tracking errors relative to the path as well as a curvature, which steers the machine onto the path. The machine aims approximately for a point a certain distance ahead of itself on the path, known as **Lookahead_Dist** in the block. The block outputs a curvature command to steer the machine to the path, which can be limited by the **Max_Curvature** parameter to give smoother driving.

As a machine travels along the path, it passes through the **Current_Waypoint**, which is the most recently passed waypoint. Whenever the **NumOfWaypoints** is greater than zero, the waypoint input is added to the path. For example, if the machine drives in a circle and is told to add four waypoints, it will keep adding four waypoints and continue driving in the circle. Only append the waypoints one time to avoid going in a continuous loop. Additionally, **Target_Spacing**, which refers to segments between waypoints on the path, will always have at least four spaces between each waypoint. As the machine passes through a waypoint, if looping is False, the previous waypoint is discarded and one more waypoint slot becomes available.

Reverse means the machine drives the path starting from the end of the path and moving toward the start of the path. This **Reverse** behaves differently than the other Autonomous Control Library path blocks.

Path_Follower Function Block

If there's a straight line and a point with a tiny radius that is wrong, the bearing entered might be pointing the opposite way. Huge radii on a straight line could negatively affect target spacing and is not recommended. Radius should be small or zero for straight lines.

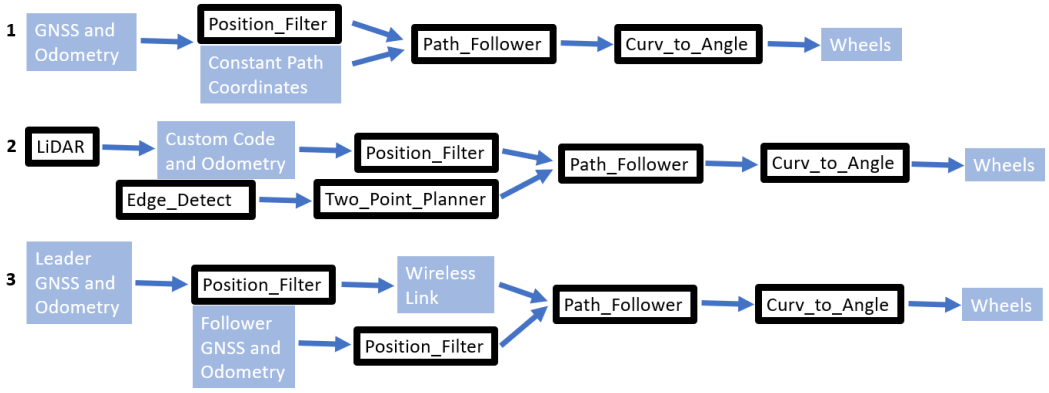
When the machine goes through the last waypoint, it completes the path if it is a non-looping path. The machine's curvature commands will keep targeting a straight line at the final waypoint's bearing. This block is not intended to be used after **Path_Complete** is True.

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss on page 29*.

Application Information

Common function blocks that work with **Path_Follower** are **Curv_To_Angle**, **Position_Filter**, and other GNSS function blocks. There will always be two parallel streams of path and machine position data going into **Path_Follower**, and information outputs to the machine wheels.

Path_Follower usually needs position information from odometry, as well as GNSS function blocks **Origin**, **Relative_Pos**, **UTM_Conv**, and **UTM_Conv_Zone**.



1. Scenario one shows machine position information obtained through odometry and GNSS function block combinations, which flow into **Position_Filter** to determine the machine's position. The machine's position constantly updates. A second stream of data for path information is loaded once into the application as constants. **Path_Follower** reads position information as it travels along the path, and path information does not append to the end of the path. Information flows into **Curv_To_Angle**, which tell the machine's wheels to turn when it reaches certain areas of the path.
2. Scenario two shows LiDAR hardware finding a target for a machine to drive toward. LiDAR code detects the environment around the machine, and custom code figures out where the machine is located before passing the information to **Position_Filter**. **Position_Filter** could use an object for positioning, like a post. Position information is constantly updating, but path information is procedurally generated. Some part of the path is known, and then how to navigate the path changes. The second stream of data for the path comes from **Edge_Detect** following an edge in the environment that the machine needs to drive toward. That information passes to **Two_Point_Planner**, which generates an optimal path to a destination. Both position and path information pass to **Path_Follower** for the machine to follow a path. Information flows into **Curv_To_Angle**, which tell the machine's wheels to turn when it reaches certain areas of the path.
3. Scenario three shows a machine following another machine. The lead machine broadcasts its position by a wireless link to the machine following it, and then the second machine appends those waypoints to its path to drive toward like a breadcrumb trail. In this case, GNSS is likely used for localization on both machines and need to use the same system. Both position and path information pass to **Path_Follower** for the machine to follow a path. Information flows into **Curv_To_Angle**, which tell the machine's wheels to turn when it reaches certain areas of the path.

Additionally, place the **Path_Follower** function block:

Path_Follower Function Block

- After position data is collected. This is either after GNSS and odometry function blocks, or after LiDAR code like the **Ouster_LiDAR** function block. If using a LiDAR, include one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- After the **Position_Filter** function block.
- After path data is collected or created.
- Before the **Curv_To_Angle** function block if using an Ackermann machine.

Inputs

Inputs to the **Path_Follower** function block are described.

Array range for X in the ARRAY[X] types should be between 1 and 50. X is dynamic.

Item	Type	Range	Description [Unit]
Pos	BUS	——	A bus that contains position and orientation data for the machine.
X	S32	-2147483648-2147483647	The X position of the machine. [mm]
Y	S32	-2147483648-2147483647	The Y position of the machine. [mm]
Yaw	S32	-72000-72000	The yaw of the machine. This uses the ENU convention and the right hand rule. [0.01 degree]
Path	BUS	——	A bus that contains options for the path and a way to define the path for the machine to follow.
Waypoint_X	(ARRAY[X]S32)	-2147483648-2147483647	The X position of the waypoint. [mm]
Waypoint_Y	(ARRAY[X]S32)	-2147483648-2147483647	The Y position of the waypoint. [mm]
Bearing	(ARRAY[X]S32)	-72000-72000	Angle at which the machine goes through the waypoint. This uses the ENU convention and the right-hand rule. [0.01 degree]
Forward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the forward control point. Smaller radii yield sharper turns. [mm]
Backward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the backward control point. Smaller radii yield sharper turns. [mm]
StartingWaypoint	U8	0-49	The index in the array that has the first waypoint to be added to this loop. If **Starting Waypoint** + **NumOfWaypoints** is greater than 50, then the index of waypoints wraps back to 0. Intended for use with ring buffers, normally set to 0.
NumOfWaypoints	U8	0-50	The number of waypoints to add to this loop. Whenever the number of waypoints is greater than zero, the waypoints are added to the path. Set this to zero to not add waypoints. Confirm with output signal **NumOfWaypointsAdded**.
Reverse	BOOL	T/F	Specifies the order that waypoints are added to the path. T: The waypoints are added to the path in reverse order. 180 degrees are added to the bearing. Backward and forward radii are swapped. The machine drives the path from the end of the path to the start. F: The waypoints are added based on their current order in the array. This **Reverse** behaves differently than **Forward_Driving** in the other path blocks.

Path_Follower Function Block

Item	Type	Range	Description [Unit]
Search_Path	BOOL	T/F	Specifies if the path is searched. T: Searches the entire path to find the nearest point to the machine to navigate toward. If **Tracking_Error** is less than two times the **Lookahead_Dist**, search is not performed. F: Does not search path. Follows the path from the beginning to the end.
Loop_Path	BOOL	T/F	Set True to loop the path. This is only updated during the first loop or when **Reset** is True. Cannot have more than 50 waypoints. T: The path keeps looping. F: The path is only done once.
Target_Spacing	U16	1-65535	Accuracy of the interpolation of the path segments. Smaller values increase the accuracy but also increase the processing time. This is only updated during the first loop or when **Reset** is True. Segments are split into at least 4 steps, and at most 1000 steps, even if the value of **Target_Spacing** specifies otherwise. [mm] **Target_Spacing** requires four spaces here but other path blocks do not require this.
Reset	BOOL	T/F	Clears the current path and stores the settings for the new path. T: Clears the path and stores the value for **Loop_Path** and **Target_Spacing** for the next path. F: Does not clear the path.
Chkpt	BOOL	——	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The following table describes parameters for the **Path_Follower** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Lookahead_Dist	U16	1-65535	The distance which the machine travels to get to the path. Shorter distances are more accurate but the system is less stable. The **Lookahead_Dist** should be larger than the wheelbase and **Target_Spacing** less than a third of **Lookahead_Dist**. [mm] Default: 3000
Max_Curv	S32	1-2147483647	The limit for the **Cmd_Curv** sent to the machine to drive through the lookahead point. This smooths the curve the machine takes. Default: 2147483647 [0.01/km]

Path_Follower Function Block

Outputs

The following table describes outputs of the **Path_Follower** function block.

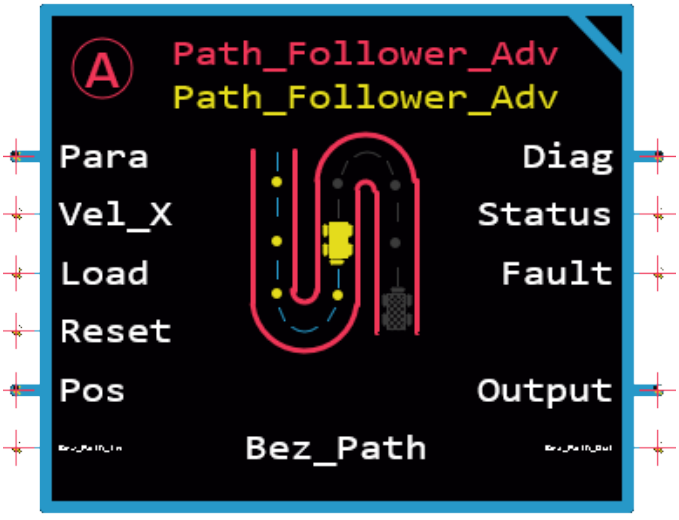
Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Bitwise code where multiple items can be reported at a time. *Non-standard 0x0000: Status OK. 0x0001: The path is empty. Less than two points are added to the path. 0x0002: It took too long to find the lookahead point. Target spacing is too small or the machine is going too fast. 0x0004: It took too long to find the nearest point to the machine. Target spacing is too small or the machine is moving too fast. 0x0008: Path is too small or lookahead distance is too big. 0x0010: At least one parameter is out of range. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high. 0x0004: Size of the input path arrays do not match. 0x0008: **NumOfWaypoints** is bigger than the size of the input path arrays.
Output	BUS	——	——
Cmd_Curv	S32	-2147483648-2147483647	The command of curvature needed to get to the **Lookahead_Dist**. Positive values are left curves. Negative values are right curves. [0.01/km] If the **Angle_Error** is greater than 90 degrees, the **Cmd_Curv** value equals **Max_Curv**. The range of this block's output is limited by the **Max_Curv** parameter.
Path_Yaw	S32	-18000-18000	The bearing of the path at its nearest interpolated point to the machine. [0.01 degree]
Tracking_Error	S32	-2147483648-2147483647	The orthogonal distance from the current position and the nearest interpolated point from the Bezier curve of the path. Negative values mean the machine is to the left of the path. Positive values mean the machine is to the right of the path. [mm]
Angle_Error	S32	-2147483648-2147483647	The difference between the machine's bearing and the bearing of the nearest interpolated point on the path. Negative values mean you are rotated to the left compared to the path. Positive values mean you are rotated to the right compared to the path. [0.01 degree] If the **Angle_Error** is greater than 90 degrees, **Cmd_Curv** goes to the **Max_Curv**.
NumOfWaypointsAdded	U8	0-50	The number of waypoints successfully added to the path during this loop.
Available_Waypoints	U8	0-50	The number of waypoints that can still be added to the path. For non-looping paths, this value increases as waypoints prior to the **Current_Waypoint** are consumed and are no longer used for path following or searching.

Path_Follower Function Block

Item	Type	Range	Description [Unit]
Current_Waypoint	U32	0-3276750	The waypoint that the machine most recently passed. For looping paths this value stays between 0 and (49 minus **Available_Waypoints**) that can still be added to the path. For non-looping paths this value keeps increasing as the machine passes waypoints. Before reaching the starting waypoint, which is Waypoint[0], the signal outputs Waypoint[0]. The second waypoint is Waypoint[1]. This **Current_Waypoint** behaves differently than the **Current_Waypoint** in other path blocks.
Loop_Count	U16	0-65535	The number of times the machine completed the full path. If **Loop_Path** is False, this output is always zero.
Path_Complete	BOOL	T/F	True when the current non-looping path is complete. This never goes True for looping paths. T: The current path is complete. The machine passed through the last **Current_Waypoint**. F: The current path is not complete.

Path_Follower_Adv Function Block

The **Path_Follower_Adv** function block uses data for a machine to follow a path. It usually comes at the end of a series of other path blocks.



This block requires a license for A+ Advanced.

Positioning for this block is measured with a default assuming the GNSS antenna is mounted at the steering point of the machine. For example, this could be the rear axle of a front wheel steer Ackermann machine. If the antenna on the machine is located in a different area, code needs to be written to translate it to the steering point. The path waypoints correlate with the front wheels and may not match the GNSS during turns.

Path_Follower_Adv is optimized for Ackermann steer machines. It works with other steering types but may not be as accurate and need further tuning.

Path_Follower_Adv comes after other path blocks obtained data about a path. If the machine is off the path, it calculates a path to return the machine to the original path. It actively consumes data from the **Bez_Path** outputs of other blocks which went to a data locker. If the data in this locker changes before **Path_Follower_Adv** finishes following the current path, it will stop following the path. For example, if the data locker ID changes from 1 to 2 when multiple paths are run together, the machine stops following the path and goes into a safe state. Use the **Load** signal to load the new path data and start moving on the path again. Or, use the **Reset** signal to override the path and put the machine in an idle state. Activating **Reset** and **Load** at the same time resets the function block and does not load new data.

Start_Distance refers to how close the machine needs to be to a path to start. The machine follows the path until the last waypoint, and if it is told to loop, then it goes on the path again. If **Search_Path** is True, then the machine navigates to the closest waypoint on the path and follows it from there if starting off the path. If False, the machine navigates back to the start of the path if it gets off the path. If the deviation from the original path is greater than **Lost_Distance**, then **Path_Follower_Adv** generates a custom path and temporarily drops the original path to follow the new path. This new path is visually available in **Bez_Path_Out** on a service tool screen. The new custom path still considers the machine kinematics, so if returning to the old path requires additional maneuvering, the machine does it.

Target_Spacing refers to the number of points between waypoints, which indicates how far apart waypoints should be from each other. Curves may need waypoints closer together. If the target spacing is too small, **Path_Follower_Adv** follows the path but automatically changes the parameter to buffer the turn. The input checkpoint will not change. **Max_Curvature** is machine dependent and defines how tight the curve could be. **Control_Gain** focuses on the machine orientation to the path and how close the machine is to the path. If control gain is low, it needs the same orientation. If control gain is too high, the machine needs to be as close to the path as possible and ends up oscillating around the path. This depends on the size of the machine.

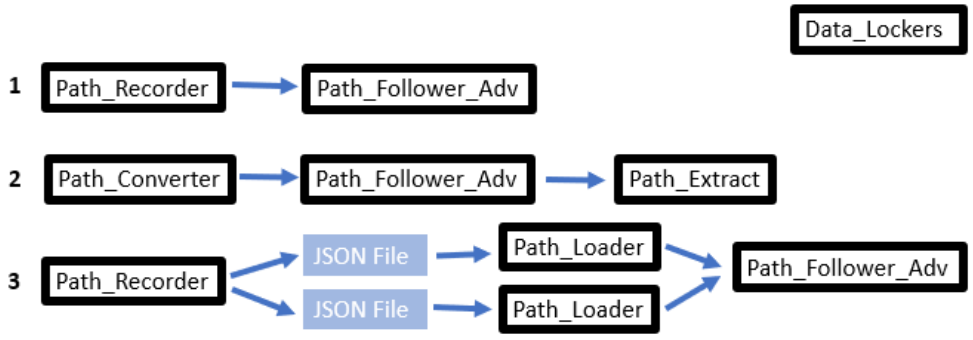
If decelerating too fast, the machine may not stop fully. If a machine cannot decelerate fast enough, it may overshoot the waypoint and start ramping to the desired velocity of the next waypoint. Monitor the state of the block in the **Internal Signals** on the service tool screen.

Path_Follower_Adv Function Block

Application Information

Common function blocks used with **Path_Follower_Adv** are **Path_Converter**, **Path_Recorder**, **Path_Loader**, **Data_Lockers**, and **Path_Extract**.

The **Path_Follower_Adv** function block consumes data about the path from a data locker so a machine can follow the path. **Path_Follower_Adv** is usually at or near the end of any path flow. Some basic path function block combinations include:



1. Scenario one shows information from the recorded path flowing into **Path_Follower_Adv** for a machine to follow the recorded route immediately. Do this for repeatable tasks that need a new path if the ECU loses power. Turning off the machine loses recorded information. One **Data_Lockers** block is required for all applications but does not connect to anything.
2. Scenario two shows data entered into **Path_Converter** flowing into **Path_Follower_Adv** for the machine to follow the path. Because the path is already known going into **Path_Converter**, it is not necessary to place **Path_Extract** directly after it. Instead, place it after **Path_Follower_Adv** to show what part of the path is left to drive. This appears on a service tool screen or display piece of hardware.
3. Scenario three shows one **Path_Recorder** block recording two paths with their own JSON files. **Path_Recorder** is used twice. The first **Path_Loader** loads one path's JSON file to a data locker, and the second **Path_Loader** loads the second path's JSON file to a different data locker. However, these data lockers are located within one **Data_Lockers** block. One **Path_Follower_Adv** block uses information from the data lockers for the machine to drive multiple paths. Add in logic for the paths to drive one after each other.

Additionally, place the **Path_Follower_Adv** function block:

- Only once in each application, even though multiple paths could exist.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.

Example

The **Path_Follower_Adv** function block is in several examples with other path blocks.

See *Example* on page 157 with the **Path_Recorder** function block.

See *Example* on page 128 with the **Path_Converter** function block.

See *Example - One Path* on page 150 with one **Path_Loader** function block.

See *Example - Multiple Paths* on page 151 with multiple **Path_Loader** function blocks.

All the examples include the **Path_Extract** function block. Each function block includes *Pre-Made Service Tool Screens* on page 25.

Path_Follower_Adv Function Block

Inputs

The following table describes inputs required for the **Path_Follower_Adv** function block.

Item	Type	Range	Description [Unit]
Bez_Path_In	S8	-1-99	Defines the ID of the path type data locker.
Load	BOOL	T/F	False to True transition loads the path to **Path_Follower_Adv**. T: Clears the current path, stores the new path, and sets the machine state to idle. Machine velocity and output curvature commands are both zero. F: If already following a path, the machine continues following it without loading new path data. If the machine is not following a path, it will not load data.
Reset	BOOL	T/F	False to True transition resets the path the machine follows. T: Clears the current path data and sets the machine into an idle state. The machine velocity and output curvature command are both zero. F: If already following a path, the machine continues following it without resetting. If the machine is not following a path, it cannot reset. This is like an emergency shutdown and not the same as **Reset** in **Path_Converter** or **Path_Recorder**.
Vel_X	S32	-2147483648-2147483647	Velocity of the machine. [mm/s]
Pos	BUS	——	Position and coordinate signals coming from the **Position_Filter** function block.
X	S32	-2147483648-2147483647	Current X position of the machine location. [mm]
Y	S32	-2147483648-2147483647	Current Y position of the machine location. [mm]
Yaw	S32	-72000-72000	The angle used to describe the machine's heading using the ENU (East-North-Up) reference frame. [0.01 degree]

Parameters

The following table describes parameters for the **Path_Follower_Adv** function block.

Item	Type	Range	Description [Unit]
Search_Path	BOOL	T/F	Specifies if the machine starts following the path from the closest waypoint. T: The machine finds the closest waypoint to follow the path. F: The machine follows the path from the starting waypoint.
Loop_Path	BOOL	T/F	After completing the path, determines if the machine follows the path again in a loop. T: The machine follows the path again from the starting waypoint and tracks new data. F: The machine does not follow the path in a loop.
Front_Wheel_Steer	BOOL	T/F	Determines whether the machine uses front wheel steering. T: The machine steers from the front wheels. F: The machine does not steer from the front wheels.
Target_Spacing	U16	1-65535	Distance between waypoints on the path. Too much distance could make the machine move away from the path, but too small distance could make the machine shake trying to stay on the path. If the spacing is too small, this value will be automatically overwritten to buffer turns. [mm] Default: 200 This **Target_Spacing** behaves differently than for **Path_Follower**.
Max_Curv	S32	6105-2147483647	The maximum curvature command the machine accepts. Greater values allow sharper turns, and smaller values allow wide turns. Default: 100000 0.01/km

Path_Follower_Adv Function Block

Item	Type	Range	Description [Unit]
Control_Gain	U16	1-65535	Tunes the accuracy of tracking. This depends on wheelbase, curve, loop time, position estimate accuracy, and steering unit response time. Greater values lead to smaller tracking errors but decrease the machine's stability. Smaller values lead to inaccurate tracking but better machine stability. Default: 1000
Wheelbase	U16	1-20000	The distance between the centers of the front and rear wheels. [mm] Default: 5000
Lost_Distance	U16	4*(10^8 / Max_Curv) to 65535	The maximum distance a machine is allowed from the original path before it needs to return to the path. For example, if it moves around an obstacle. [mm] Default: 5000
Start_Distance	U16	1-65535	The desired distance between the machine and the starting waypoint, which is Waypoint[0] in GUIDE. The second waypoint is Waypoint[1]. If the machine does not start at Waypoint[0], GUIDE reads Waypoint[-1] until the machine reaches Waypoint[0]. **Search_Path** does not affect this parameter. The distance is measured from both the rear and front axle of the machine, and the smaller distance is considered. Default: 500 [mm]

Outputs

The following table describes outputs for the **Path_Follower_Adv** function block. This block comes last in the path block series, but data could output to **Path_Extract**.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Output	BUS	——	BUS containing information about machine errors, curvature, and the path.
Cmd_Curv	S32	-2147483648-2147483647	Curvature command for the machine to follow the path. Negative values indicate the machine turns right, and positive values indicate the machine turns left. [0.01/km]
Cmd_Vel	S32	-2147483648-2147483647	Velocity command for the machine to follow the path. [mm/s]
Tracking_Err	S32	-2147483648-2147483647	This shows how far away the machine is from the path by measuring the distance between the machine's position and the nearest interpolated point on the path. The tracking error is perpendicular to the path. Negative values mean the path is on the machine's left side, and positive values mean right side. [mm]
Angle_Err	S32	-18000-18000	The difference between the machine's yaw and the yaw of the closest interpolated point on the path. For example, if the machine's yaw is 100 and the path's yaw is 90, the angle error is 10. Negative values indicate the machine is pointed to the left of the path, and positive values indicate the machine pointed to the right. [0.01 degree]

Path_Follower_Adv Function Block

Item	Type	Range	Description [Unit]
Current_Waypoint	S32	-1-65535	The waypoint that the machine most recently passed. Before reaching the starting waypoint, which is Waypoint[0], the signal outputs Waypoint[-1]. The second waypoint is Waypoint[1]. This **Current_Waypoint** behaves differently than the **Current_Waypoint** in other path blocks.
Loop_Count	U16	0-65535	The number of times the machine completed the full path. If **Loop_Path** is False, this output is always zero.
Path_Complete	BOOL	T/F	Indicates when the machine passes through the final waypoint, completing the path. T: The current path is complete. The machine passed through the last **Current_Waypoint**. F: The current path is not complete. This output is always False if **Loop_Path** is True.
Bez_Path_Out	S8	-1-99	This contains future path information. If the machine approaches the starting waypoint or gets lost, the ID points to a data locker that contains the desired path the machine will follow to get back to the original path and start the tracking. If -1 is passed into **Bez_Path_In**, this parameter holds the last available path.

Internal Signals

The following table describes what is happening internally in the **Path_Follower_Adv** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Path_Follower_Adv_Err	U8	0-7	Indicates when an error occurred in the block functionality. See *Path_Follwer_Adv Troubleshooting* on page 148.
Target_Index	U16	0-1000	The index of the interpolated point on the path segment the machine is currently following. Look here to see if an issue happened at a certain index number. This number should go up and not stall unless the machine stopped.
Search_Progress	U16	0-10000	Indicates the percentage of the path searched to find the closest interpolated point upon startup. It is non-zero only if **Search_Path** is True. If searching a long path, monitor this before the machine moves to confirm that **Search_Path** finished. [0.01%]
Distance_To_Start	U32	0-4294967295	The distance from the starting waypoint. Once the machine is within the **Start_Distance** range, its value will be zero. Monitor this value to see if the machine has a hard time finding the start of the path. [mm]
Ancillary_Prev	U32	0-4294967295	Extra information attached to the waypoint that has been passed last by the machine. Look here to see if data was triggered or came through. This data originally comes from **Path_Converter** or **Path_Recorder**.
Ancillary_Next	U32	0-4294967295	Extra information attached to the next waypoint the machine is driving towards. Look here to see if data was triggered or came through. This data originally comes from **Path_Converter** or **Path_Recorder**.
State	U8	0-6	The state of the **Path_Follower_Adv** function block. 0: Idle, waiting for reset. 1: Finding the starting waypoint. 2: Following the shortest path to get to the starting waypoint. 3: Following the path. 4: Lost and trying to get back to the path. 5: Path finished when **Loop_Path** is False. Outputs zero velocity and curvature command. 6: Safe, no velocity and curvature command at the output, waiting for a False to True transition on **Load** and **Reset** to be False.

Path_Follower_Adv Function Block

Path_Follwer_Adv Troubleshooting

The following table describes errors that could occur in the **Path_Follower_Adv** function block, as well as ways to fix them. View the **Path_Follower_Adv_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

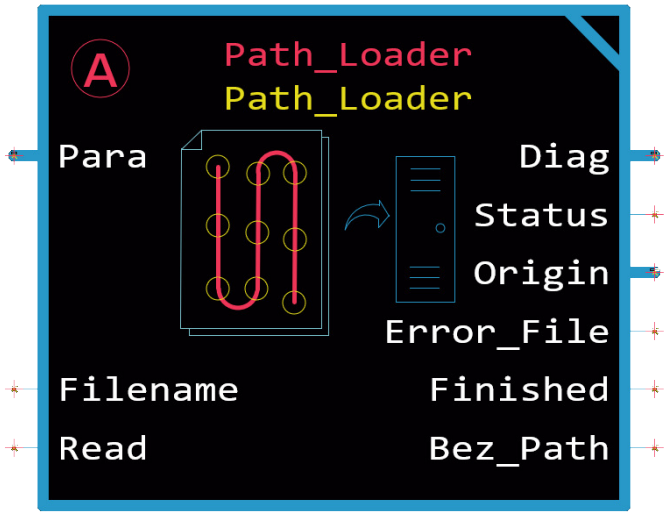
Path_Follower_Adv_Err Descriptions and Fixes

Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
2	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.
3	A background thread took more than 500 milliseconds to process the data. However, a valid output is still coming out.	Turn the controller off and on, reduce the amount of points processed, or record a shorter path.
4	Less than two waypoints exist on the path. There needs to be a starting waypoint plus two more.	Verify the correct path is used. Add more waypoints, if needed.
5	The **Target_Spacing** value is too small. The block still runs but tries to modify the parameter internally, creating a lower resolution.	Increase the **Target_Spacing** value, or break the path into multiple smaller paths. **Target_Spacing** needs at least two waypoints.
6	The machine cannot navigate to the starting waypoint from the JSON file.	Adjust the **Start_Distance** signal.
7	The path is recorded with **Forward_Driving** programmed in one direction, but the machine is manually driven in the opposite direction. The machine must follow the path in the recorded direction.	Record a new path in the desired direction, and manually drive the machine in the same direction of the recorded path.

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss* on page 29.

Path_Loader Function Block

The **Path_Loader** function block reads a JSON file, which contains information about a path a machine follows. This path is then uploaded into the **Data_Lockers** block for other blocks to access.



This block requires a license for A+ Advanced. It also requires hardware compatible with the media file system, such as the XM100. The minimum HWD version must be greater than 3.21.

Positioning for this block is measured with a default assuming the GNSS antenna is mounted at the steering point of the machine. For example, this could be the rear axle of a front wheel steer Ackermann machine. If the antenna on the machine is located in a different area, code needs to be written to translate it to the steering point. The path waypoints correlate with the front wheels and may not match the GNSS during turns.

Path_Loader requires a JSON file produced by the **Path_Recorder** function block. It reads the data from a JSON file, optionally checks that the data is not corrupt, and then loads it to **Data_Lockers** for other blocks in the application to use. It is highly recommended to enable parameter **Verify_CRC** to check for corrupted data, otherwise the block could load bad data. Data is checked by Cyclical Redundancy Check (CRC), using the MD5 protocol.

If a JSON file needs modification, make any changes to it and save a new MD5. Then, use the new JSON file in **Path_Loader**. See *Modify JSON and Update MD5* on page 27.

Path_Loader outputs the origin data contained in the JSON file. Outputs include the status of the hardware tier, faults in PLUS+1® GUIDE, and other errors that occurred. Then, **Path_Loader** indicates when the path has loaded into **Data_Lockers** on the **Finished** output. **Path_Loader** checks that the JSON file exists, the data is intact, and optionally checks that the data is not corrupted if the **Verify_CRC** parameter is true. If something does not check out, an error signal is indicated in the **Error_File** output. This error file can be checked on the **Checkpoints** page or service tool screen.

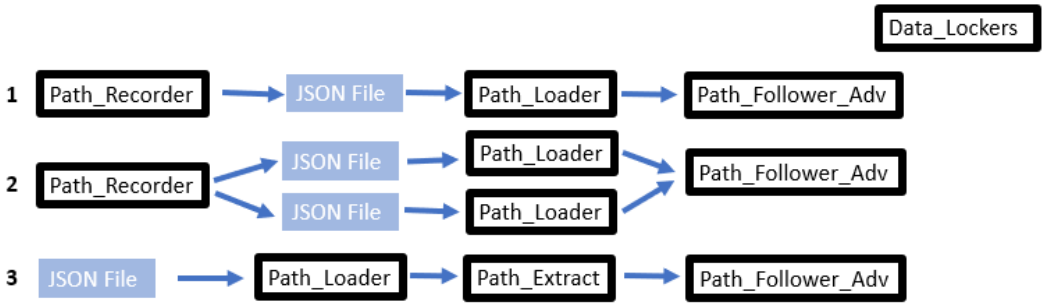
If the machine needs to move in both forward and backward directions, create multiple paths so the machine goes in one direction per path. Each path needs a **Path_Loader** function block to upload the path data to its own data locker within the **Data_Lockers** block.

Application Information

Common function blocks that work with **Path_Loader** are **Path_Extract**, **Path_Recorder**, **Data_Lockers**, and **Path_Follower_Adv**.

The **Path_Loader** function block uploads the JSON file with path data gathered from the **Path_Recorder** function block into a **Data_Lockers** block, that could then be consumed by other blocks. Information about the position of the machine is required earlier in the application. Some basic path function block combinations are the following:

Path_Loader Function Block



1. Scenario one shows **Path_Recorder** recording the path to a JSON file stored on the controller. **Path_Loader** loads the JSON file information to a data locker, and then **Path_Follower_Adv** uses the path data for the machine to follow the path. One **Data_Lockers** block is required for all applications but does not connect to anything.
2. Scenario two shows one **Path_Recorder** recording two paths with their own JSON files. The first **Path_Loader** loads one path's JSON file to a data locker, and the second **Path_Loader** loads the second path's JSON file to a different data locker. However, these data lockers are located within one **Data_Lockers** block. One **Path_Follower_Adv** block uses information from the data lockers for the machine to drive multiple paths. Add in logic for the paths to drive one after each other.
3. Scenario three shows **Path_Loader** reading the path from the JSON file and transmitting it to a Service Tool screen or a hardware display using **Path_Extract**. The JSON file existed with all the path information recorded from a previous time. Then, **Path_Follower_Adv** uses the path data for the machine to follow the path.

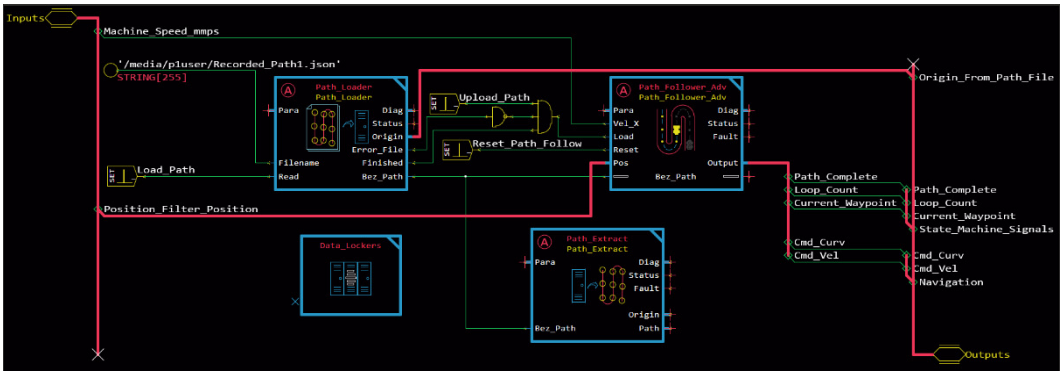
Additionally, place the **Path_Loader** function block:

- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- After **Path_Recorder** if it is required to create the JSON file. If a JSON file exists from a different application, that block is not required.
- After position and yaw information is obtained, such as after a **Position_Filter** function block.
- Multiple times in an application if there are many paths. There should be a **Path_Loader** block for each path, and a machine needs a new path if it changes between forward and reverse velocity.

Example - One Path

This example shows the **Path_Loader** function block uploading information to a data locker for one path, and the **Path_Follower_Adv** function block uses that data for a machine to follow the path. The **Path_Extract** function block visually displays the uploaded path information.

Set up GNSS and positioning code earlier in the application, which could use the **Position Filter** function block.



The example assumes code exists earlier in the application to establish a machine's position. See an overview of steps and explanations about what they achieve:

Path_Loader Function Block

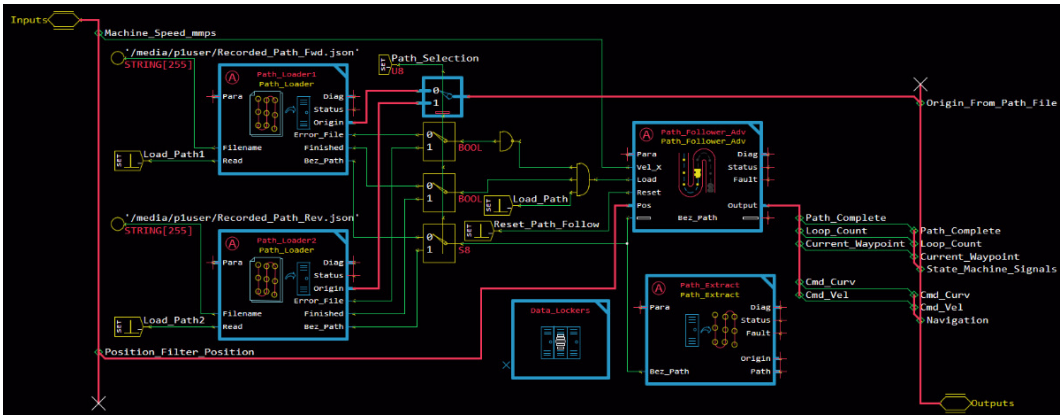
1. Add the **Path_Loader**, **Path_Extract**, and **Path_Follower_Adv** function blocks. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
2. Connect the JSON's file path to the signal **Filename**. Here, the file name is '/media/p1user/Recorded_Path1.json'. **Path_Recorder** generates the JSON file, which is stored on a local file directory such as the XM100.
3. Create a pulse signal to **Read** the JSON file. This pulse triggers the block to read the file and write the information to a data locker.
4. Create logic for **Path_Follower_Adv** to access the information loaded to a data locker. In this example, the logic checks that no errors occurred from **Path_Loader** loading the JSON file to the data locker, and it finished writing the information to a data locker.
5. Create logic to check the **Error_File** is false before uploading data.
6. Connect the origin. This example uses origin data gathered from the path recording JSON's file. Optionally, use the origin from the **Origin** function block.
7. Connect velocity, which is optional in many applications. If not using velocity, disregard the velocity command signal from **Path_Follower_Adv**.
8. Create a pulse for **Reset** on **Path_Follower_Adv**. This signal triggers the block to go into a safe state and overrides the **Load** signal.
9. Connect position, which uses the **Position_Filter** function block or other code that sends position information.
10. Connect **Bez_Path** from **Path_Loader** to **Path_Follower_Adv**. **Path_Loader** sends information about the recorded path to a data locker for **Path_Follower_Adv** to use. Nothing more needs to be done besides connecting the two function blocks because the data locker stores the data automatically.
11. Create any desired **Output** data from **Path_Follower_Adv**. This example includes common data a state machine uses, such as:
    a) Notifying when the machine finishes the path in **Path_Complete**.
    b) Recording how many times the machine followed the path in **Loop_Count**.
    c) Displaying information about which waypoint the machine just drove through in **Current_Waypoint**.
12. Connect **Cmd_Curv** to the machine's steering control system and **Cmd_Vel** to the machine's propel control system. The **Path_Follower_Adv** sends curvature commands to the machine to help it stay on the path.
13. Connect **Path_Extract** between the **Bez_Path** signals, which extracts information from a data locker. Here, **Path_Extract** could show the data about the path visually on a service tool screen, which helps with debugging issues.
14. Optionally, see the *Pre-Made Service Tool Screens* on page 25 for all three blocks for comprehensive results. Or, create a display of the information on a piece of hardware such as the DM1000.

Example - Multiple Paths

This example shows **Path_Loader** function blocks uploading information to a data locker for multiple paths. A machine can go in one direction on the path, so piecing together multiple paths allows the machine to go forward and backward. The **Path_Extract** function block visually displays the uploaded path data, and the **Path_Follower_Adv** function block uses that data for a machine to follow the path.

Set up GNSS and positioning code earlier in the application, which could use the **Position Filter** function block.

Path_Loader Function Block



The example assumes code exists earlier in the application to establish a machine's position and origin. See an overview of steps and explanations about what they achieve:

1. Add several **Path_Loader** blocks. Each **Path_Loader** block corresponds to a new path, which could be the machine going in a different direction. Here, the machine travels forward using the higher block and backwards using the lower block.
2. Add **Path_Extract** and **Path_Follower_Adv** function blocks. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page, and there should only be one **Data_Lockers** in an application.
3. Connect one JSON's file path to the signal **Filename**. Here, the file name is '/media/p1user/Recorded_Path_Fwd.json' for the path driving forward. **Path_Recorder** generates the JSON file, which is stored on a local file directory such as the XM100.
4. Connect the other JSON's file path to the signal **Filename**. Here, the file name '/media/p1user/Recorded_Path_Rev.json' refers to the path recorded while the machine was driving in a reverse direction.
5. Create pulse signals to **Read** the JSON files. This pulse triggers the block to read the file and write the information to a data locker. **Path_Loader** loads the information it reads into a data locker automatically if the **Data_Lockers** block exists somewhere in the application.
6. Create logic for **Path_Follower_Adv** to access the information loaded to a data locker. There only needs to be one **Path_Follower_Adv** block per application.
7. Create a switch so each **Path_Loader** goes to a different data locker within the **Data_Lockers** block.
8. Create logic to check the **Error_File** signals are false before uploading data.
9. Create a pulse for **Reset** on **Path_Follower_Adv**. This signal triggers the block to go into a safe state and overrides the **Load** signal.
10. Connect the origins. This example uses origin data gathered from the path recording JSON files. Optionally, use the origin from the **Origin** function block.
11. Connect velocity, which is optional in many applications. If not using velocity, disregard the velocity command signal from **Path_Follower_Adv**.
12. Connect position, which uses the **Position_Filter** function block or other code that sends position information.
13. Connect **Bez_Path** from **Path_Loader** to **Path_Follower_Adv**. **Path_Loader** sends information about the recorded path to a data locker for **Path_Follower_Adv** to use. Nothing more needs to be done besides connecting the two function blocks because the data locker stores the data automatically.
14. Create any desired **Output** data from **Path_Follower_Adv**. This example includes common data a state machine uses, such as:
    a) Notifying when the machine finishes the path in **Path_Complete**.
    b) Recording how many times the machine followed the path in **Loop_Count**.
    c) Displaying information about which waypoint the machine just drove through in **Current_Waypoint**.

Path_Loader Function Block

15. Connect **Cmd_Curv** to the machine's steering control system and **Cmd_Vel** to the machine's propel control system. The **Path_Follower_Adv** sends curvature commands to the machine to help it stay on the path.
16. Connect **Path_Extract** between the **Bez_Path** signals, which is the area that uploads information to a data locker. Here, **Path_Extract** could show the data about the path visually on a service tool screen, which helps with debugging issues.
17. Optionally, see the *Pre-Made Service Tool Screens* on page 25 for all four blocks for comprehensive results. Or, create a display of the information on a piece of hardware such as the DM1000.

Inputs

The following table describes inputs required for the **Path_Loader** function block. The JSON file data comes from the **Path_Recorder** block.

Item	Type	Range	Description [Unit]
Filename	STRING[255]	——	Name of the JSON file, which is made from **Path_Recorder**. The default name is '/media/p1user/Recorded_Path.json'. The file must be within 'media/p1user'.
Read	BOOL	T/F	False to True transition starts reading the JSON file and writing the information into a path type data locker. T: Read the file. F: Do not read the file. For optimal performance, have **Read** pulse false after reading the JSON file so it is not constantly reading the file.

Parameters

The following table describes the parameter for the **Path_Loader** function block.

Item	Type	Range	Description [Unit]
Verify_CRC	BOOL	T/F	Indicates whether to check for corrupt JSON file data using Cyclical Redundancy Check (CRC) MD5 protocol. It is recommended to check. T: Check for a corrupt JSON file. F: Do not check for a corrupt JSON file. Default: True

Outputs

The following table describes outputs for the **Path_Loader** function block. The data could go into **Path_Follower_Adv** or **Path_Extract** function blocks.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8100: Invalid ECU.
Origin	BUS	——	BUS containing UTM values of the path's origin, which are read from the JSON file.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin.

Path_Loader Function Block

Item	Type	Range	Description [Unit]
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	0x00000000-0x540BE400 This is the range of the full U64 bit number.	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.
Updated	BOOL	T/F	Indicates when new data is being stored for the origin. T: New data is available for the origin. F: No new data is available.
Error_File	U8	0-8	Indicates when an error happened in the JSON file. See *JSON File Path Errors* on page 155.
Finished	BOOL	T/F	Indicates whether the path loaded into the data locker. T: Path data loaded. F: Path data has not loaded.
Bez_Path	S8	-1-99	Defines the ID of the path type data locker.

Internal Signals

The following table describes what is happening internally in the **Path_Loader** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Path_Loader_Err	U8	0-2	Indicates when an error occurred in the block functionality. See *Path_Loader Troubleshooting* on page 154.
Progress	U16	0-10000	Indicates the progress loading the path in the data locker. Look here to see if **Path_Loader** has stopped processing data. [0.01%]

Path_Loader Troubleshooting

The following table describes errors that could occur in the **Path_Loader** function block and ways to fix them. View the **Path_Loader_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Path_Loader_Err Descriptions and Fixes

Error Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread. Too many threads are already running in the application.	Turn the controller off and on, or use less code in the application.
2	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss* on page 29.

Path_Loader Function Block

JSON File Path Errors

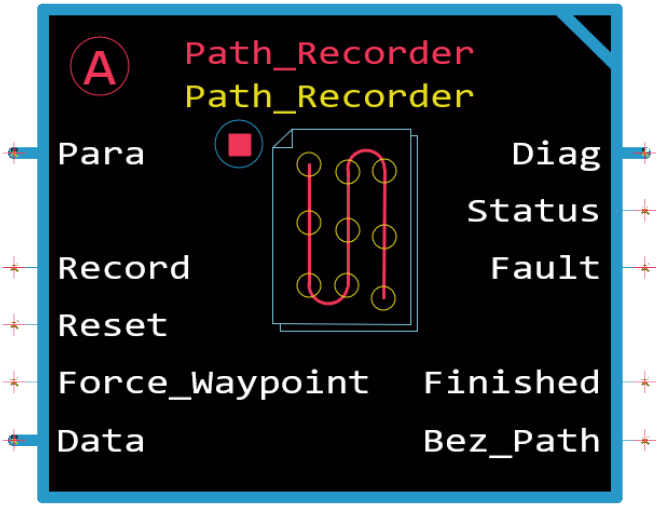
The table shows errors related to the JSON file path function blocks and ways to fix them. To change a JSON file, see *Modify JSON and Update MD5* on page 27.

JSON Error File

Error Number	Description	How to Fix
0 - LOADER_ERROR_OKAY	No errors.	Nothing needs to change.
1 - LOADER_ERROR_FILE_NOT_FOUND	The JSON file is not available. The file could have the wrong name or not exist.	Verify the JSON file name is correct. Check that the block looks for it in the correct location, such as the XM100 or USB connected to the XM100.
2 - LOADER_ERROR_FILE_SIZE	There may not be enough memory available to read the JSON file, or the file is too large. This could happen if the file was manually edited.	Take information out of the file, or use several path files and have them run one after the other.
3 - LOADER_ERROR_READ	The JSON file could not be read because it is empty or incomplete.	Verify the file has not been manually modified, and it contains the recorded path instead of wrong information.
4 - LOADER_ERROR_CRC	The Cyclical Redundancy Check (CRC) failed. The JSON file is corrupted or manually modified.	Disable the **Verify_CRC** parameter if the JSON file is modified on purpose. However, this is discouraged. Compile the block again after disabling to see if errors 5-8 appear, which indicate what specifically is going wrong.
5 - LOADER_ERROR_JSON_NULL	Wrong JSON data or incorrect JSON file.	Verify the JSON file is valid, not manually modified, and contains the recorded path.
6 - LOADER_ERROR_METADATA	Wrong or missing metadata. This happens after manually creating or modifying a JSON file if the **Verify_CRC** is disabled.	Check that all the metadata fields are present in the JSON file.
7 - LOADER_ERROR_ZERO_WAYPOINTS	The path is empty with zero waypoints.	Check the information stored in the JSON file and re-record the path.
8 - LOADER_ERROR_WRONG_WAYPOINT_LEN	The number of waypoints expected is different than the number recorded. This happens after manually modifying the file.	Re-record the path.

Path_Recorder Function Block

The **Path_Recorder** function block records the path a machine travels, and then stores the data in a JSON file and **Data_Lockers**, allowing the machine to follow that path again.



This block requires a license for A+ Advanced. It also requires hardware compatible with the media file system, such as the XM100. The minimum HWD version must be greater than 3.21.

The block records a series of waypoints, which are intermediate points on a route. These contain the machine's relative XY position, yaw, and optionally the velocity and ancillary data, over a user specified distance. Create paths based on either speed or position. The block does not record the time the machine stops. Create additional code or multiple paths to account for the machine pausing.

While recording, it is recommended to have less than 25000 waypoints.

After the recording stops, the block stores these raw data waypoints and creates a file called raw_data.csv, if the developer chooses. The .csv file takes up a lot of memory. That file includes values for x, y, yaw, velocity, forced waypoints, and ancillary data. The block filters the data so the machine drives smoothly when it retraces the recorded path later. The filtering process:

- Discards waypoints that are ±45 degrees different from the previous waypoint
- Verifies the velocity did not change between forward and backward
- Creates Bezier curves between waypoints
- Refines the Bezier curves based on the **Velocity_Tolerance** parameter

The filtered data appears in a **Bez_Path** signal and the JSON file. The **Bez_Path** signal allows the machine to follow the path immediately, but it does not save the data over power cycles. The JSON file is stored in the XM100 memory and includes the default name 'media/p1user/Recorded_Path.json'. Developers can optionally download JSON files from the XM100, see *Getting Files from XM100* on page 29.

If a JSON file needs modification, make any changes to it and save a new MD5. Then, use the new JSON file in **Path_Loader**. See *Modify JSON and Update MD5* on page 27.

The **Path_Recorder** function block records a machine going in one direction at a time, either forward or backward. If the machine moves in multiple directions, record multiple paths to piece together later. This block requires the machine to be manually driven along the path route to record the path. To write the path coordinates into the code without driving the machine first, use the **Path_Converter** function block. Usually, an application does not need both **Path_Recorder** and **Path_Converter** blocks together.

Positioning for this block is measured with a default assuming the GNSS antenna is mounted at the steering point of the machine. For example, this could be the rear axle of a front wheel steer Ackermann machine. If the antenna on the machine is located in a different area, code needs to be written to translate it to the steering point. The path waypoints correlate with the front wheels and may not match the GNSS during turns.

Path_Recorder Function Block

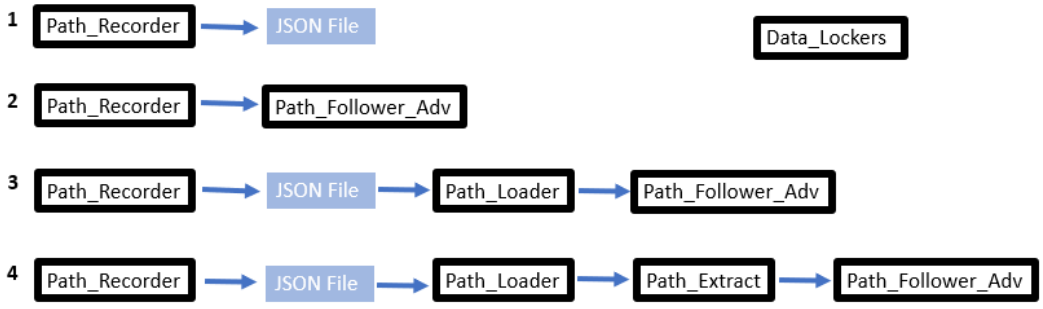
The **Path_Recorder** function block only supports Linux-based controllers. The recording stops and deletes the raw data file if there are less than 100 megabytes of memory left on the controller. If the machine moves forward and backward on the same recorded path, an error occurs.

When a machine moves in a back-and-forth direction, changes yaw suddenly, or experiences electrical or communication disturbances, the machine may do sharp turns. To avoid sharp turns, force enough waypoints, balance the tolerance values for curve fitting to skip disturbances, and check there are not too many waypoints slowing down the whole system.

Application Information

Common function blocks that work with **Path_Recorder** are **Data_Lockers**, **Path_Loader**, **Path_Follower_Adv**, and **Path_Extract**.

Path_Recorder records path data by manually driving a machine along a path to record, rather than entering data into the function block. Information about the position of the machine is required earlier in the application. Some basic path function block combinations include:



1. Scenario one shows **Path_Recorder** without other path blocks. Do this to save and copy a path to another machine to use later. Path information saves in a JSON file. One **Data_Lockers** block is required in all applications and does not need to connect to anything.
2. Scenario two shows information from the recorded path flowing into **Path_Follower_Adv** for a machine to follow the recorded route immediately. Do this for repeatable tasks that need a new path after the machine turns off. Turning off the machine loses recorded information.
3. Scenario three includes **Path_Loader** loading the recorded path information to a data locker from the JSON file, which allows the machine to follow the path immediately or later from the saved file.
4. Scenario four includes **Path_Extract**. This block visually displays data about the path on a service tool screen or a hardware display, such as the DM1000. Placing **Path_Extract** after **Path_Follower_Adv** could show information about what part of the path is left to drive.

Additionally, place the **Path_Recorder** function block:

- Only once in each application, even though many paths can be recorded from the one block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.
- After position and yaw information is obtained, such as after a **Position_Filter** function block.

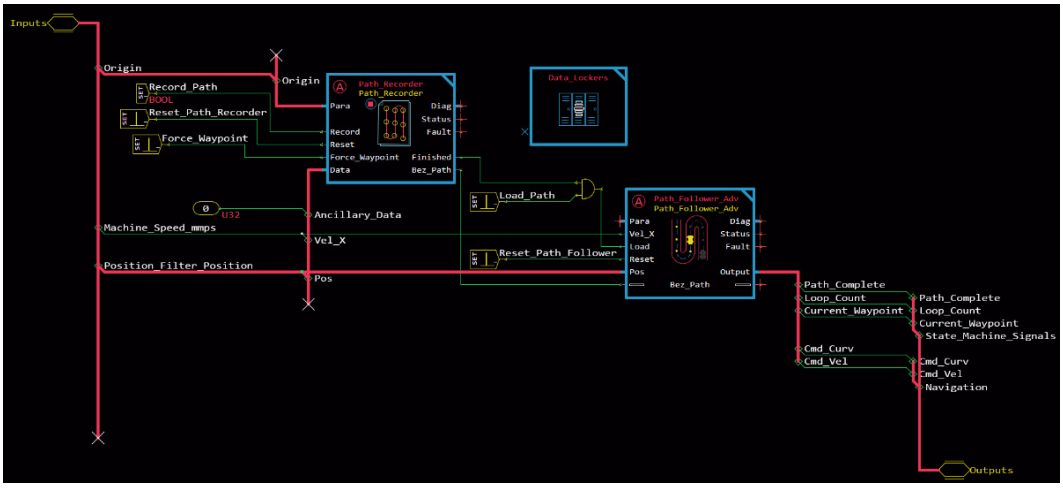
If the application needs velocity, get the signals from a wheel odometer. Ancillary data may also need to come from other hardware.

Example

The example shows the **Path_Recorder** function block recording a path that the **Path_Follower_Adv** function block then uses immediately. The program would be used in repeatable tasks that do not require the machine to remember paths over power cycles.

Even though **Path_Recorder** records the path and stores the recorded JSON file, **Path_Follower_Adv** consumes data from the **Data_Lockers** block for the machine to run. If the machine is turned off and on, it will not follow this path again.

Path_Recorder Function Block



The example assumes code exists earlier in the application to establish a machine's position and origin. See an overview of steps and explanations about what they achieve:

1. Add the **Path_Recorder** and **Path_Follower_Adv** function blocks. Additionally, add a **Data_Lockers** block if one does not already exist in the application. It can go on any page.
2. Connect the **Origin** to **Path_Recorder**. The origin information comes from earlier in the application and ties the machine's position to a local coordinate frame.
3. Set **Record** to True for the entire recording. The machine must be manually driven while the recording occurs. In this example, a set value component allows someone to set the value of the signal on a service tool screen.
4. Create a set pulse for **Reset** and **Force_Waypoint**. **Reset** deletes a recorded path. **Record** must cycle from True to False to True afterward to start re-recording the path. A forced waypoint means a particular waypoint will be included in the final processed path. If a waypoint is not forced, that coordinate could be filtered out of the path.
5. Leave **Ancillary_Data** as zero if there is nothing special to add. This input is a general area that can be used for any extra data, such as work function information.
6. Determine whether to use velocity for **Path_Recorder** to take in and **Path_Follower_Adv** to use. Velocity is optional for most applications. For example, the machine velocity could be manually controlled. If not using velocity, set it to zero.
7. Connect **Pos**, which is the position information established earlier in the application code. This lets the machine know where it is in relation to the path and could come from the **Position_Filter** function block.
8. Create code so **Path_Follower_Adv** checks that **Path_Recorder** finished processing the path before loading the data, and set a pulse to trigger the **Load** signal. This allows **Path_Follower_Adv** to use the recorded data so the machine can drive the path that was just recorded.
9. Create a pulse for **Reset** on **Path_Follower_Adv**. This signal triggers the block to go into a safe state and overrides the **Load** signal.
10. Connect **Bez_Path** from **Path_Recorder** to **Path_Follower_Adv**. Information about the recorded path goes directly to a data locker for **Path_Follower_Adv** to use. Nothing more needs to be done besides connecting the two function blocks because the data locker stores the data automatically.

Path_Recorder Function Block

11. Create any desired **Output** data from **Path_Follower_Adv**. This example includes common data a state machine uses, such as:
    a) Notifying when the machine finishes the path in **Path_Complete**.
    b) Recording how many times the machine followed the path in **Loop_Count**.
    c) Displaying information about which waypoint the machine just drove through in **Current_Waypoint**.
12. Connect **Cmd_Curv** to the machine's steering control system and **Cmd_Vel** to the machine's propel control system. The **Path_Follower_Adv** sends curvature commands to the machine to help it stay on the path.
13. Optionally, visually see the data from **Path_Recorder** and **Path_Follower_Adv** on their *Pre-Made Service Tool Screens* on page 25, or view each signal individually.

Inputs

The following table describes inputs required for the **Path_Recorder** function block. Most of this data comes from the **Position_Filter** block and optionally a wheel odometer.

Item	Type	Range	Description [Unit]
Record	BOOL	T/F	Triggers the recording and saves the recorded data into a data locker. When the **Record** signal goes from True to False, it stops the recording and starts saving the data to a data locker. T: Recording. F: Waiting to record or saving the path data. Changing to false in the middle of the recording stops the recording but processes the data already gathered.
Reset	BOOL	T/F	False to True transition determines whether to stop the recording. When **Save_Raw_Data** is True, the recording stops after writing the recorded path data to a data locker without discarding the data. T: Stops the recording. F: Signal has no effect on block functionality.
Force_Waypoint	BOOL	T/F	Triggers a waypoint to stay in the path rather than relying on the **Path_Recorder** algorithm to create all the waypoints. T: Forces the machine to go to a specific spot on the path. The waypoint information is stored in the final post processed path. F: Does not force the machine to travel to specific waypoints.
Data	BUS	——	BUS containing information about the path.
Ancillary_Data	U32	0-4294967295	Array of extra information attached to a specific waypoint. For example, this could be information about temperature, the state of the machine, or directions to stop.
Pos	BUS	——	Position and coordinate signals coming from the **Position_Filter** function block.
X	S32	-2147483648-2147483647	Current X position of the machine location. [mm]
Y	S32	-2147483648-2147483647	Current Y position of the machine location. [mm]
Yaw	S32	-72000-72000	The angle used to describe the machine's heading using the ENU (East-North-Up) reference frame. [0.01 degree]
Vel_X	S32	-2147483648-2147483647	Velocity of the machine. [mm/s]

Path_Recorder Function Block

Parameters

The following table describes parameters for the **Path_Recorder** function block. All these parameters can be hard-coded. However, the **Origin** bus could pull data from the **Origin** function block directly.

Item	Type	Range	Description [Unit]
Origin	BUS	——	BUS containing UTM values of the path's origin, which are either outputs from the **Origin** function block or hard-coded.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	0x00000000-0x00000002	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	0x00000000-0x540BE400 This is the range of the full U64 bit number.	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in.
Updated	BOOL	T/F	Indicates when new data is being stored for the origin. T: New data is available for the origin. F: No new data is available.
Fitting	BUS	——	BUS containing information about how smoothly the machine travels on the path.
Min_Movement	U16	100-10000	Minimum displacement to consider the position in the path. [mm] Default: 500
Max_Rollback_Distance	U16	1 to **Min_Movement**	Maximum distance machine is allowed to move opposite the driving direction without an error occurring. This feature disables if no velocity is input into the block. **Max_Rollback_Distance** must be greater than or equal to **Min_Movement**. [mm] Default: 1
Curve_Fit_Tolerance	U16	100-10000	Maximum distance between fitted and desired paths. [mm] Default: 500
Velocity_Tolerance	U16	100-10000	Maximum distance between fitted and desired velocity. [mm/s] Default: 500
Save_Raw_Data	BOOL	T/F	Determines whether to save the data gathered as the machine travels along a path. T: If the machine resets, save the data in raw_data.csv. F: If the machine resets, delete the raw data.
Zero_Vel_Threshold	U32	0-60000	The time limit the **Vel_X** input can be 0 after the path recording starts. This applies after the recording has started with the machine moving, and the program sees a non-zero velocity. Then, if the machine stop time is greater than this parameter, the recording automatically stops. Default: 1000 [ms]
Wheelbase	U16	1-20000	The distance between the centers of the front and rear wheels. [mm] Default: 5000

Path_Recorder Function Block

Item	Type	Range	Description [Unit]
Metadata	BUS	——	BUS containing extra data relevant to the block.
App_Name	STRING[255]	——	Name of the application. Use 255 characters or less.
Date_Time	STRING[255]	——	Timestamp in a format of YYYY/DD/MM hh:mm. This is data for the waypoints in the path recording.
Ancillary_Caption	STRING[255]	——	Describes the data in the **Ancillary_Data** signal.
Filename	STRING[255]	——	Name of the output JSON file. Rename the file, if desired. The default name is '/media/p1user/Recorded_Path.json'.
Forward_Driving	BOOL	T/F	Indicates whether the machine drives forward or reverse. T: The machine drives in the direction of the machine face. F: The machine drives in the direction opposite to the machine face (negative velocity). For example, the machine starts at the beginning of the path and drives facing backwards to the end of the path.

Outputs

The following table describes outputs for the **Path_Recorder** function block. The data could go into **Path_Follower_Adv** or **Path_Loader** function blocks, or the data could sit in a JSON file for later. See *Getting Files from XM100* on page 29.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Bez_Path	S8	-1-99	Defines the ID of the path type data locker.
Finished	BOOL	T/F	Finished signal is True when processing the recorded path data is done. T: Processing recorded data is done and the path is available in a data locker. F: Processing recorded data is not done.

Internal Signals

The following table describes what is happening internally in the **Path_Recorder** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Path_Recorder_Err	U8	0-12	Indicates when an error occurred in the block functionality. See *Path_Recorder Troubleshooting* on page 162.
Progress	U16	0-10000	Indicates the progress of processing the raw recorded path data into a final path with filtered data. Look here to see if **Path_Recorder** has stopped processing data. Short paths may not indicate incremental progress but only say when the processing is 100% completed. [0.01%]
Zero_Vel_Time	U32	0-4294967295	Indicates how long the **Vel_X** input of **Path_Recorder** was zero during recording. Monitor **Zero_Vel_Time** to see if the time surpasses the **Zero_Vel_Threshold** parameter and how long it lasted. The path recording automatically stops if the threshold is passed. [ms]

Path_Recorder Function Block

Item	Type	Range	Description [Unit]
Num_Raw_Points	U32	0-4294967295	Number of raw data points stored in the raw_data.csv file. See *Getting Files from XM100* on page 29 to access this data. If **Save_Raw_Data** is True, these data points are before the path filtering process occurs. Look here to see how many waypoints were excluded from the filtered path or to see if there is an option of getting data from other points.
Num_Waypoints	U16	0-65535	Number of waypoints in the path used by **Path_Recorder** after filtering data. Look at this value to see if too many waypoints are excluded and no data is processing.

Path_Recorder Troubleshooting

The following table describes errors that could occur in the **Path_Recorder** function block and ways to fix them. View the **Path_Recorder_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Path_Recorder_Err Descriptions and Fixes

Number	Description	How to Fix
0	There is no error.	Nothing needs to change.
1	Error writing the raw CSV file. The file cannot be opened or the new data cannot append to it. The file could be read only or not available.	Verify that the CSV file '/media/p1user/raw_data.csv' is in the correct location and not read only.
2	There is less than 100 megabytes of memory left on the controller.	Delete files on the controller's hard drive, '/media/p1user/raw_data.csv'.
3	The machine moved opposite the direction defined by the **Forward_Driving** signal, and this movement is greater than the **Max_Rollback_Distance** signal allows. A path cannot record both forward and reverse.	Increase the **Max_Rollback_Distance** signal, if possible. Or, record two paths with one in forward and the other in reverse.
4	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
5	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.
7	The program cannot open the raw data file for post-processing or the file does not exist. When the CSV file is not generated after path recording, this error appears.	Re-record the path.
8	The number of lines in the CSV file differs from the number of lines in the **Path_Recorder** function block. The data in CSV does not match the JSON file.	Re-record the path.
9	Cannot open the JSON file '/media/p1user/Recorded_Path.json' for writing.	Check the JSON file location or re-record the path.
10	Cannot write some parts of the data to the JSON file. The JSON file might be read only.	Check that the JSON file is not read only, or re-record the path.
11	The JSON file size is larger than allowed and runs out of memory. There cannot be more than 1 million characters.	Re-record the path with less data points, and make sure the metadata has smaller strings.
12	The number of waypoints exceeds the limit of 65,535 waypoints.	Re-record the path with less waypoints, or create multiple paths.

Other Errors and Fixes

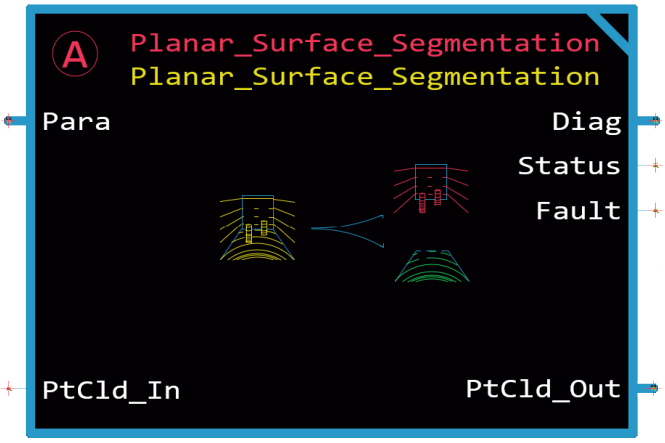
Error Description	How to Fix
The **Progress** bar stays at 0% and then jumps to 100%.	The **Progress** bar may not indicate incremental progress, even though the data is processing. Monitor the **Path_Recorder_Err** signal to see if errors occur. Otherwise, there should not be any issues related to the **Progress** bar jumping to 100%.
The **Progress** bar stays at 0%.	The **Progress** bar may not indicate incremental progress, but the data could be processing. Monitor the **Path_Recorder_Err** signal to see if errors occur. Monitor the **Finished** flag to confirm the path completed processing. Longer paths with more data take longer to process.

Path_Recorder Function Block

Additionally, the current path is lost if the ECU loses power unexpectedly or is power cycled while following a path. To recover from an ECU power loss, see *Restart or Resume Recording After ECU Power Loss* on page 29.

Planar_Surface_Segmentation Function Block

The **Planar_Surface_Segmentation** function block uses a LiDAR scanner to isolate a singular flat surface in the point cloud data. This includes any surface of interest like a ground or wall.



This block requires a license for A+ Advanced.

Planar_Surface_Segmentation requires LiDAR hardware and accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. See the *Plus+1 Compliant Ouster Block User Manual* for information.

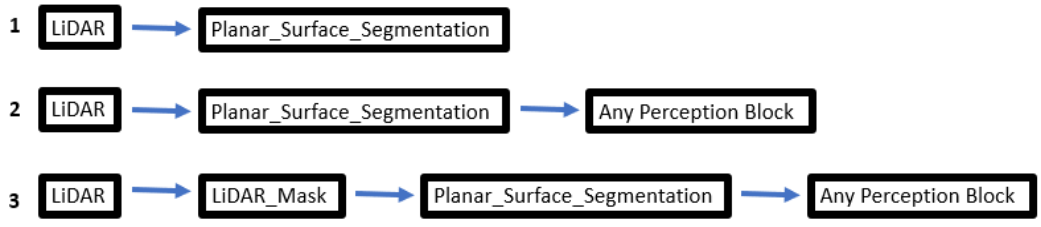
Machines driving over bumpy ground or taking weight on one side could tip the LiDAR scanner, giving false data about the environment around the machine. **Planar_Surface_Segmentation** helps with situations like these by consistently detecting the ground which could be used to offset data from the bumps or tipping. The block accurately finds walls, flat ground, and sloping ground.

The block algorithm finds a planar surface, which is a surface that is flat or level and extends infinitely in two dimensions. In doing so, the block splits the input point cloud into two different parts and outputs those.

Application Information

The **Planar_Surface_Segmentation** function block uses a LiDAR scanner to collect point cloud data and identify a planar surface.

Common function blocks that work with **Planar_Surface_Segmentation** are the **Ouster_LiDAR** function block if using Ouster LiDAR hardware, and perception blocks such as **Obstacle_Avoidance** and **Obstacle_Detect**.



Planar_Surface_Segmentation Function Block

1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as an Ouster LiDAR hardware and the **Ouster_LiDAR** function block. This gathers point cloud data which **Planar_Surface_Segmentation** uses.
2. Scenario two includes an Autonomous Control Library (ACL) perception function block after **Planar_Surface_Segmentation**. Any perception block that uses point cloud data works. For example, connect **Obstacle_Avoidance** to one of the **Planar_Surface_Segmentation** output point clouds to detect obstacles in a more accurate and efficient way.
3. Scenario three includes the **LiDAR_Mask** function block after the LiDAR code, followed by **Planar_Surface_Segmentation**. **LiDAR_Mask** omits certain data from the LiDAR's point cloud, saving processing time when detecting the planar surface with **Planar_Surface_Segmentation**. A perception block can receive input data for further analysis.

Additionally, place the **Planar_Surface_Segmentation** function block:

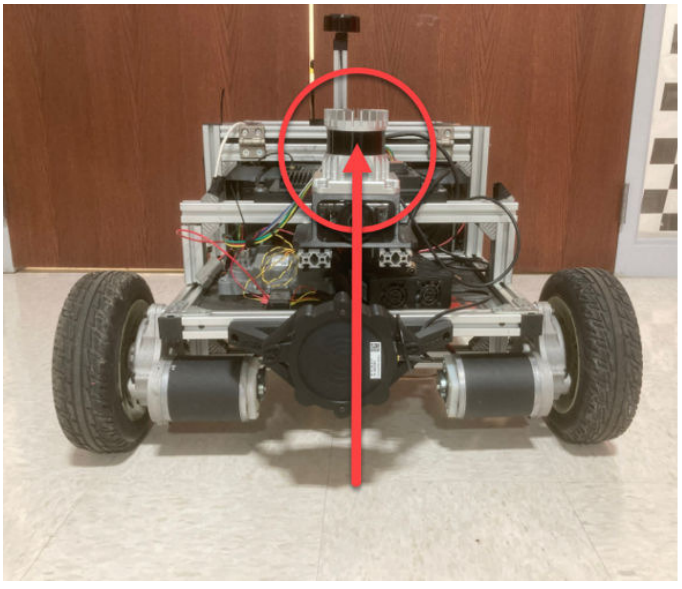
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application. If a **Data_Lockers** block already exists in the application, do not add more.
- After and not before the LiDAR code, as represented in the scenarios. LiDAR hardware is required.

The process begins with accessing the environment the machine is in and determining what the machine is expected to accomplish. The LiDAR code and hardware are set up with this in mind. Then, **Planar_Surface_Segmentation** gets point cloud data from the LiDAR, processes it, and splits the point cloud into two parts.

Configure the LiDAR

LiDAR hardware and software configuration are required to get the information the **Planar_Surface_Segmentation** function block needs to work.

Environment plays an important role in determining how to set up the LiDAR hardware because the machine needs to see certain features. Applications using **Planar_Surface_Segmentation** begin with LiDAR hardware and accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. See the *Plus+1 Compliant Ouster Block User Manual* for information about LiDARs, point clouds, and the **Ouster_LiDAR** function block. Parameters programmed into the LiDAR hardware and block earlier in the code determine the scope of the point cloud that **Planar_Surface_Segmentation** can use.

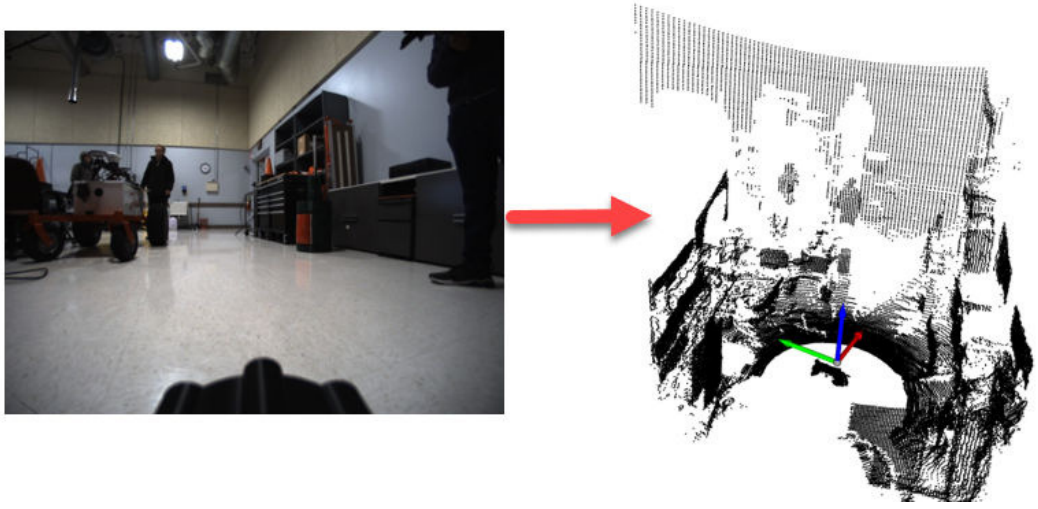


The images show a machine with a LiDAR on top (circled in red). Take measurements from the ground to the LiDAR origin, and take measurements of important items in the environment surrounding the machine.

Planar_Surface_Segmentation Function Block

The height and placement of the LiDAR hardware affect **Planar_Surface_Segmentation** parameters. The LiDAR should be positioned so it can find the plane, or surface, and measurements taken to see how high up the LiDAR sits and how far away it needs to see. For instance, if detecting the ground, the LiDAR hardware should have a view of the ground around the machine. If detecting a wall, it needs to view the wall.

The LiDAR's scope of view is usually programmed into the LiDAR's code, not **Planar_Surface_Segmentation**, although the scope could be narrowed within the block. Anything outside of the LiDAR's view will not be detected at all, and the LiDAR cannot see behind objects.



The image shows the room from the LiDAR scanner's perspective, and then it shows the point cloud generated from the LiDAR into the room.

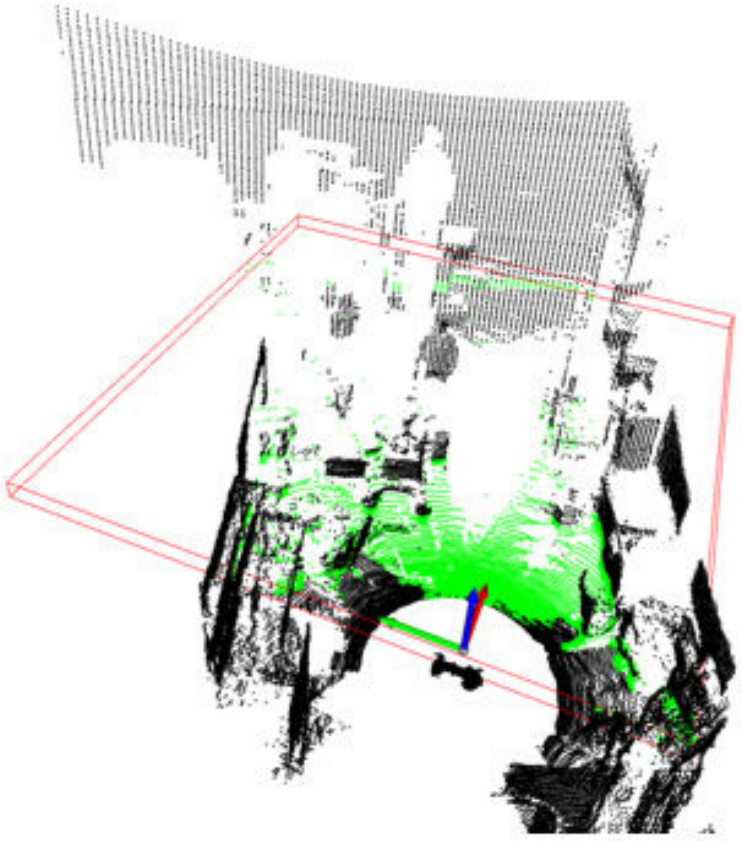
In PLUS+1® GUIDE, the point cloud data obtained from the LiDAR goes into a data locker and then inputs into **PtCld_In** on **Planar_Surface_Segmentation**. See *Data_Lockers Helper Block* on page 77 for more information on data lockers. On a service tool screen, the data locker ID number between 0-99 displays in **PtCld_In**.

Configure the Region of Interest

After the LiDAR is set up with incoming point cloud data, **Planar_Surface_Segmentation** parameters define a 3D region of interest zone to view this point cloud data and locate a surface.

This 3D region of interest zone acts as a filter determining which data to analyze rather than limiting the point cloud. If trying to locate the ground, the boundary dimensions should include the ground. If detecting a wall, the dimensions should include the wall. If detecting multiple surfaces, use one **Planar_Surface_Segmentation** function block per surface. However, multiple blocks take a lot of processing power.

Planar_Surface_Segmentation Function Block



The image shows the point cloud with the region of interest (red rectangle). Black dots represent the point cloud points the LiDAR sees, with white areas outside the LiDAR's view. The green dots represent the point cloud points **Planar_Surface_Segmentation** processes from the LiDAR after parameters are further tuned. The block looks for a surface within the region of interest boundary and point cloud it sees (represented with green points), and ignores the other data (black points, white spaces).

Start_Angle and **Stop_Angle** parameters define the LiDAR's scope of view, or Azimuth window, within **Planar_Surface_Segmentation**. Parameters for this window were also defined earlier in the LiDAR code, but they could be tuned here. For the LiDAR to see in front of itself, a **Start_Angle** of -60 degrees and **Stop_Angle** of 60 degrees gives it a 120 degree range of view ahead. The region of interest boundary should be within the LiDAR's window of view.

Max_Height and **Min_Height** parameters define the height range of the region of interest boundary where input point cloud data is analyzed. The LiDAR's origin point determines the height starting point, with negative numbers telling the LiDAR to look below itself and positive numbers telling it to look above. For instance, if the LiDAR is mounted 300 mm above the ground, to detect the ground, set **Min_Height** to -300 mm and **Max_Height** to -100 mm to see a 200 mm range just above the ground, or set it to 100 mm for a bigger 400 mm height range from the ground.

Max_X and **Min_X** define how far the region of interest boundary is in front of and behind the LiDAR, with the LiDAR as a zero starting point. For example, **Max_X** of 50000 mm and **Min_X** of -50000 mm indicates the region of interest extends a range of 100000 mm around the LiDAR. If the LiDAR needs to only process data in front of itself, **Min_X** of 20000 mm would start the region of interest boundary further ahead and save space from processing unnecessary data.

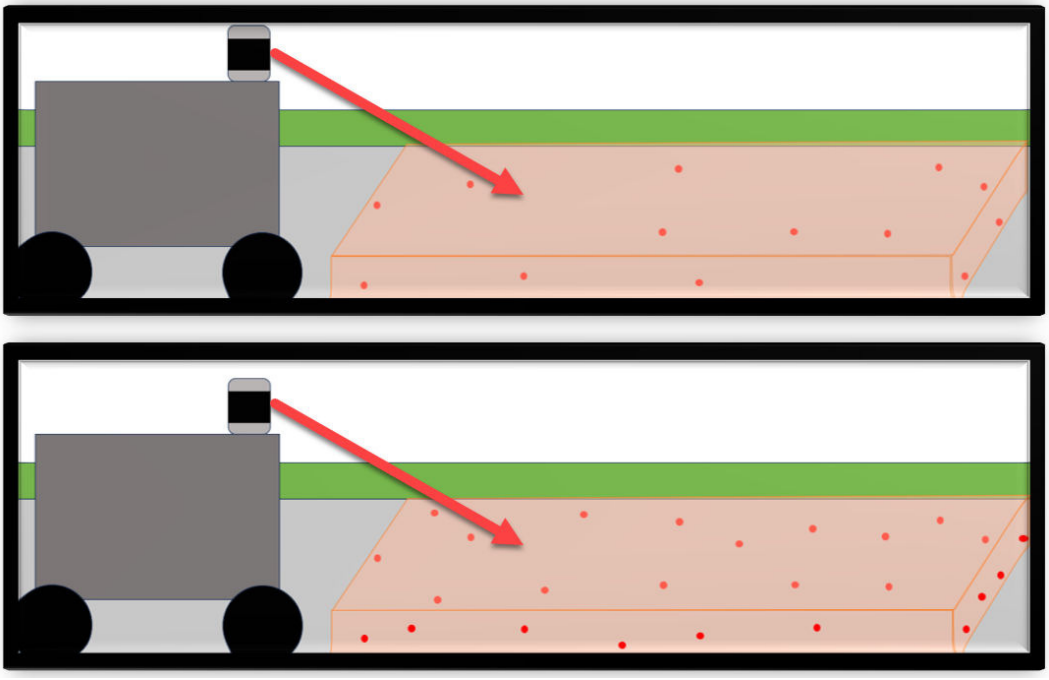
Max_Y and **Min_Y** define the width of the region of interest boundary. If the LiDAR needs to see a small hallway, these numbers could be small. For a wide field, they would be large. For example, with the LiDAR as the zero starting point, **Max_Y** of 50000 mm and **Min_Y** of -50000 mm indicate the boundary extends a range of 100000 mm around the LiDAR.

Planar_Surface_Segmentation Function Block

Find a Plane

After defining the 3D region of interest dimensions in the input point cloud, **Planar_Surface_Segmentation** finds the plane within the region.

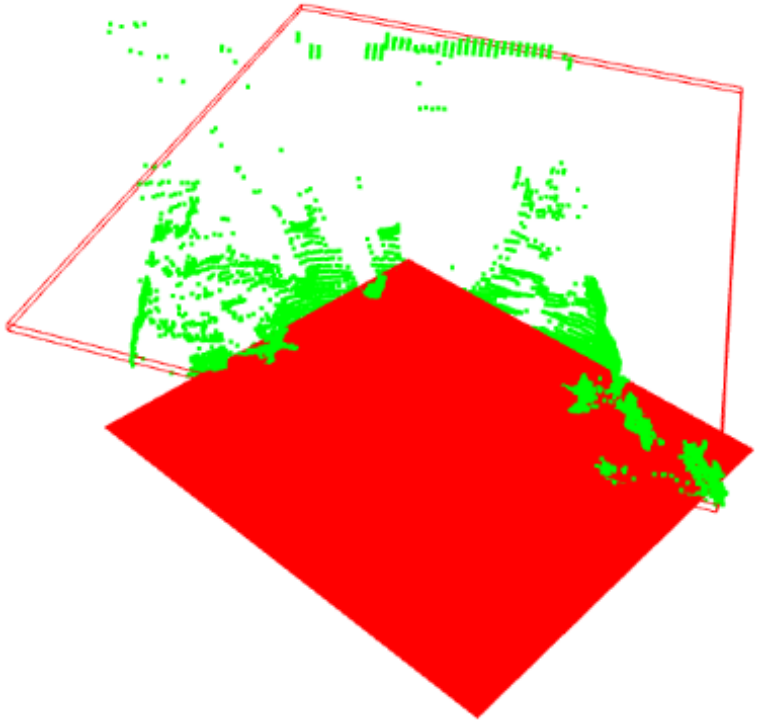
To begin finding the plane, **Planar_Surface_Segmentation** uses the **Fraction_Ransac_Points** parameter to randomly select a percentage of input point cloud points within the region of interest. **Num_Iterations** determines how many times to randomly select the points and average them out to create the plane. Lower **Fraction_Ransac_Points** and **Num_Iterations** lead to better processing times but less accurately portray the plane.



The images show the LiDAR viewing points within the region of interest (orange rectangle). The top image shows a lower number of **Fraction_Ransac_Points** being used than the bottom image.

Planar_Surface_Segmentation determines the most common plane and uses this dominant plane in the rest of the process.

Planar_Surface_Segmentation Function Block

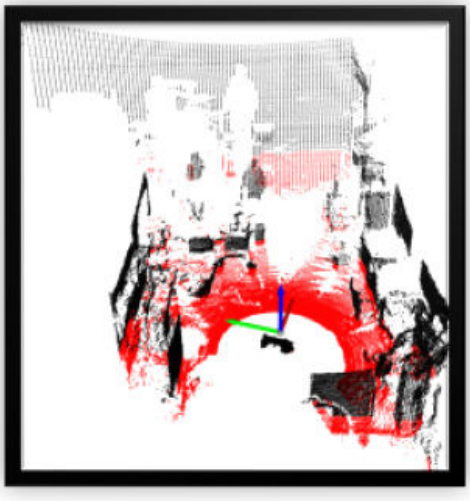


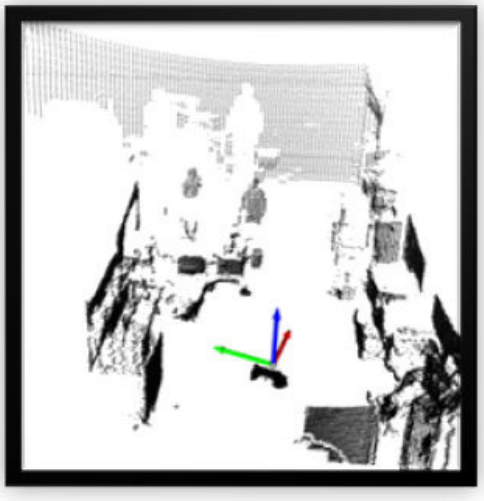
The image depicts the region of interest (rectangle) around the point cloud (green dots) with the solid red square depicting the identified most dominant plane. This plane extends infinitely, even outside of the region of interest bounds.

Configure the Inlier Point Cloud

After finding the most dominant plane, the **Planar_Surface_Segmentation** function block uses it to separate the input point cloud into two parts called inlier and outlier.

The region of interest boundary is no longer used after the plane is detected. Any points that fall within the plane become the inlier point cloud, and anything beyond that becomes the outlier point cloud. Both point clouds extend as far as the LiDAR hardware can see unless other code limited this scope.





The first image shows the inlier point cloud based on the planar surface (red points). The second image shows the outlier point cloud (black points).

Planar_Surface_Segmentation Function Block

The inlier point cloud begins with no depth, just the flat plane. However, the **Distance_Threshold** parameter is used to add volume to the plane and make it a surface. For instance, entering 10 mm into PLUS+1® GUIDE expands the inlier point cloud by 10 mm both above the plane and below the plane for a range of 20 mm.

The **Ordered** parameter indicates that **Planar_Surface_Segmentation** will output an ordered point cloud if True. Most perception blocks in the Autonomous Control Library require ordered instead of unordered point cloud data. If changed to False, the information associated with each point become Not a Number (NaN) and are lost. Unordered saves processing power, but the lost information cannot be retrieved later.

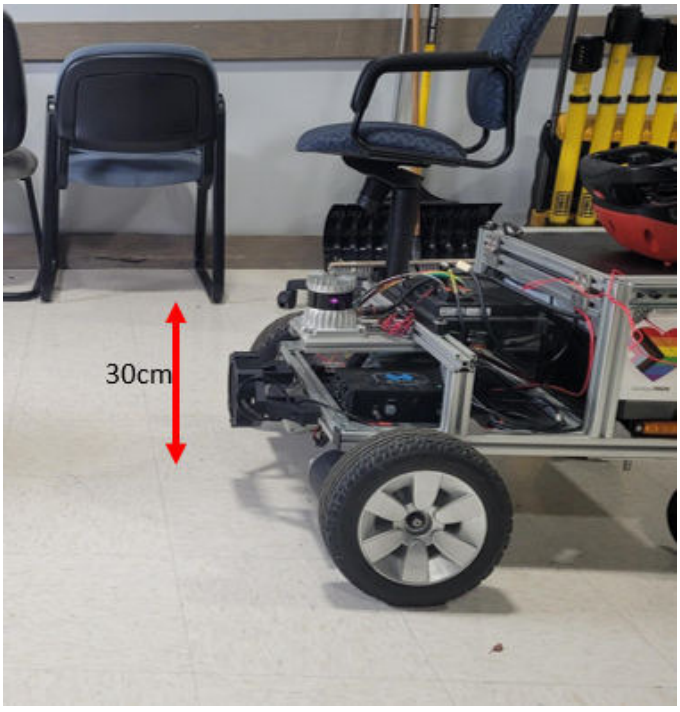
Tweak parameters to fine tune processing power by reviewing the *Pre-Made Service Tool Screens* on page 25. The inlier and outlier output point clouds could be programmed to do different things.

Example

This example explains how to connect the **Planar_Surface_Segmentation** function block from a high level, and then fill out the parameters to identify the ground plane.

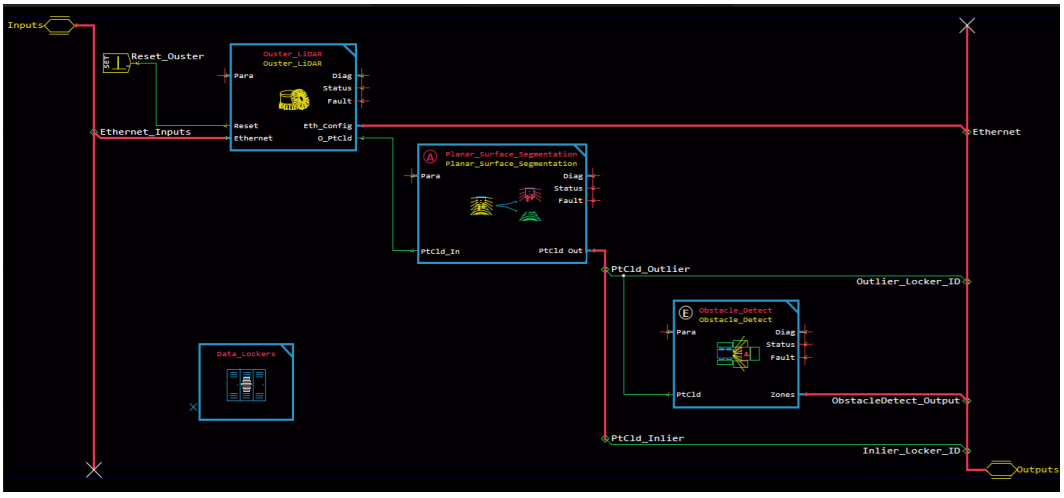
The environment where the machine runs matters a great deal for **Planar_Surface_Segmentation**. Test the machine in different environmental settings which relate to where the machine will be used in real life situations.

1. Set up a LiDAR scanner on the machine and measure how high it is from the ground to the LiDAR origin. This example uses Ouster LiDAR hardware and assumes the LiDAR is 300 mm above the ground on a small machine. View the *Ouster LiDAR User Manual* to set up the hardware and **Ouster_LiDAR** function block.

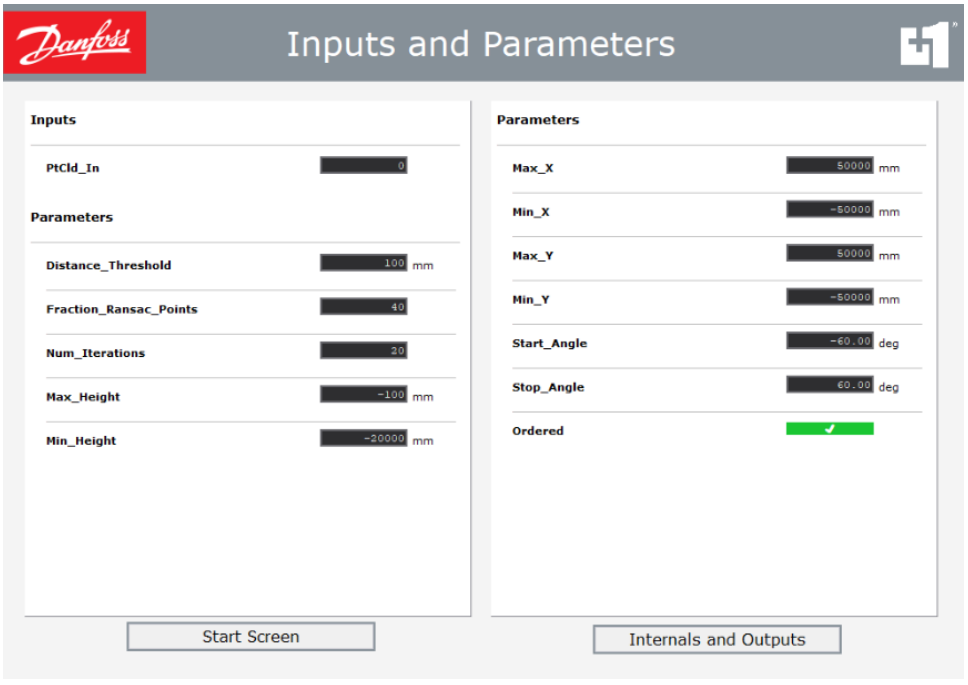


The image shows the machine with a LiDAR scanner on top measuring 30 cm from the ground to the LiDAR origin.

Planar_Surface_Segmentation Function Block



2. Set up the LiDAR code. This example uses the **Ouster_LiDAR** function block. The area the LiDAR sees is determined by the physical hardware and accompanying LiDAR code. For example, if a LiDAR scanner is set up so the ground or wall cannot be seen, this cannot be corrected in **Planar_Surface_Segmentation**.
3. Add in the **Data_Lockers** block if it was not already in the application. This block should not be connected to any buses or wires, only exist once in the application, and can go on any page within the application.
4. Connect **O_PtCld** from the **Ouster_LiDAR** block to **PtCld_In** in **Planar_Surface_Segmentation**. This allows point cloud information gathered from the LiDAR to be consumed by **Planar_Surface_Segmentation**. This information flows to a data locker and back automatically between these two blocks, so no numbers need to be adjusted.
5. Connect **Planar_Surface_Segmentation** to another Autonomous Control Library (ACL) perception block. Here, the outlier output point cloud connects to the **Obstacle_Detect** function block so the machine behaves a certain way when any objects in the outlier point cloud are detected.
6. Configure the parameters within **Planar_Surface_Segmentation**. Here, the code is set up to find the ground and assumes the LiDAR hardware is 300 mm from the ground to the LiDAR origin.



Planar_Surface_Segmentation Function Block

The image shows the parameters in the service tool screenshot. See *Pre-Made Service Tool Screens* on page 25 to find a pre-made service tool screen. **PtCld_In** shows the data locker ID of 0.

a) Set **Max_Height** and **Min_Height** to find a region of interest. In this example, the LiDAR is 300 mm from the ground to the LiDAR origin, so setting the parameters to -100 mm and -20000 mm means the LiDAR reads between 20000 mm and 100 mm below itself. This configuration filters out noise and unrelated points from above the ground.

b) Set **Max_X**, **Min_X**, **Max_Y**, and **Min_Y** to complete the region of interest boundary. Here, a large rectangle is created around the LiDAR to pick up point cloud points on a surface. X indicates length, and Y indicates width.

c) Set **Start_Angle** and **Stop_Angle** to limit the region of interest boundary. Here, to isolate the ground using points directly ahead of the machine, the parameters are assigned -60 degree and 60 degree values. This prioritizes the points ahead of the machine and removes irrelevant data to the sides of the machine.

d) Leave **Ordered** as True. This allows ordered point cloud data to process out of **Planar_Surface_Segmentation** and be used by other ACL perception blocks. This example eventually connects to **Obstacle_Detect**.

e) Set **Distance_Threshold**. Setting it to 100 mm means the plane found on the ground will have an input point cloud that is 100 mm above and 100 mm below the ground for a height range of 200 mm. However, the point cloud extends as wide and long as the LiDAR hardware can see and LiDAR code allows.

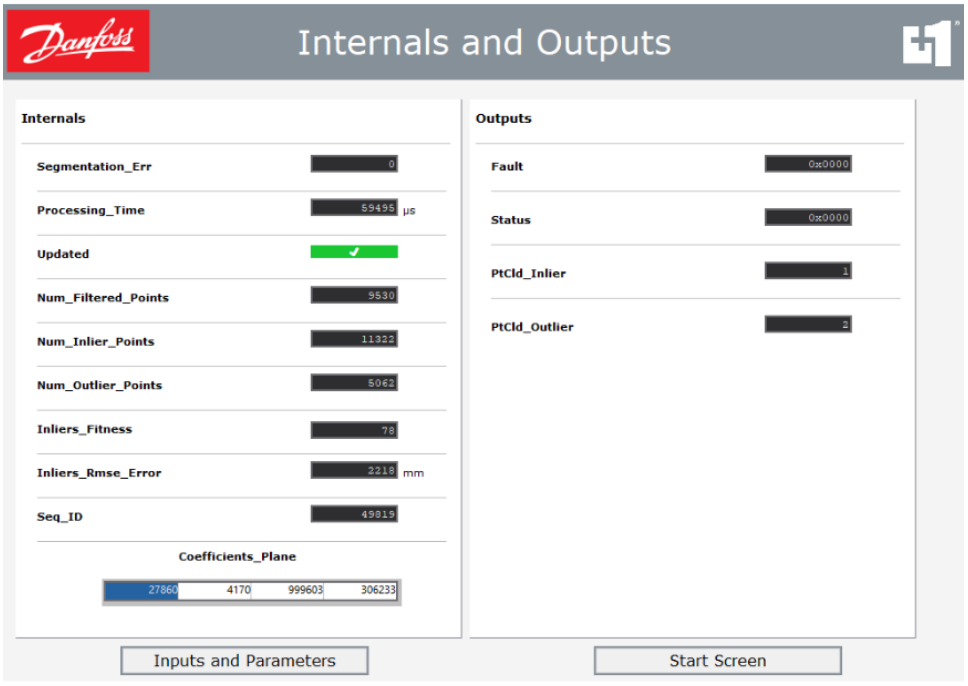
f) Set **Fraction_Ransac_Points** to randomly select a percentage of input point cloud points within the region of interest. Here, 40 percent of the points are selected.

g) Set **Num_Iterations** to determine how many times to randomly select the points and average them out to create the plane. Here, the points will be averaged out 20 times.

Check Internal Signals

After setting up the parameters, check the **Planar_Surface_Segmentation** function block internal signals to see if they are working well.

See *Pre-Made Service Tool Screens* on page 25 to find a pre-made **Internals and Outputs** service tool screen.



1. Check error numbers in **Segmentation_Err**. If a number besides zero appears, see how to fix it in *Planar_Surface_Segmentation Troubleshooting* on page 177. Zero indicates no error. Three indicates

Planar_Surface_Segmentation Function Block

slow processing times and requires further tuning in the block. See *Reduce Processing Time* on page 174.

2. Note the **Processing_Time**, which should be below 500000. To lower it, adjust parameters **Fraction_Ransac_Points** and **Num_Iterations** to smaller numbers.
3. See if the **Updated** signal appears green, which means data is processing. A red X means data stopped processing, which usually means the processing time was too long.
4. Check that the **Num_Filtered_Points** number is more than 100 so the block has enough points to find a plane. This number displays the number of input point cloud points that passes the constraints set in the region of interest boundary.
5. Add **Num_Inlier_Points** and **Num_Outlier_Points** to determine the total number of output point cloud points. The total number of input point cloud points can be calculated using the reference table. If the input point cloud point numbers match the output point cloud points, then the block is working correctly. If the numbers do not match, data may be lost.

Table showing the total number of input point cloud points for some common LiDAR configurations:

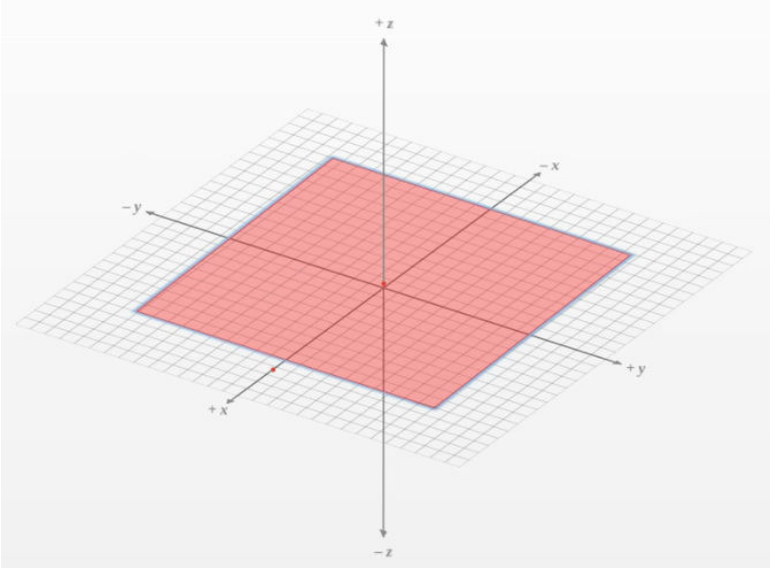
	Ouster 32 channel LiDAR (Any frequency)		Ouster 128 channel LiDAR (Any frequency)	
	Hor. Res: 512	Hor. Res: 1024	Hor. Res: 512	Hor. Res: 1024
Forward Only (-90° to +90°)	8192	16384	32768	65536
Full 360 (-180° to +180°)	16384	32768	65536	131072

In general, the formula to use for determining total number of input point cloud points is as follows:

$$\left(\frac{\text{Number of LiDAR Channels} \times \text{Horizontal Resolution}}{\frac{360}{\text{Total angle covered}}}\right)$$

The image shows the number of point cloud points common for the type of LiDAR channel.

6. Skip **Inliers_Fitness** and **Inliers_Rmse_Error**. These are for internal Danfoss use.
7. Check that the **Seq_ID** increases. This is a timestamp of when the LiDAR hardware captured the point cloud. With each LiDAR capture, the **Seq_ID** increases, indicating that new data processed.
8. Observe the **Coefficients_Plane** numbers to see the equation of the plane, which is ax + by + cz + d = 0. Here, it would be 27860x + 4170y + 999603z + 306233 = 0. This can also be visualized in other programs, such as *CPM 3D Plotter*.



Planar_Surface_Segmentation Function Block

The image displays a visual of the plane, which shows it is on the ground. The coefficients do not have units, but some external programs may require units. Here, the coefficient numbers were divided by 10^6 to get distance in meters.

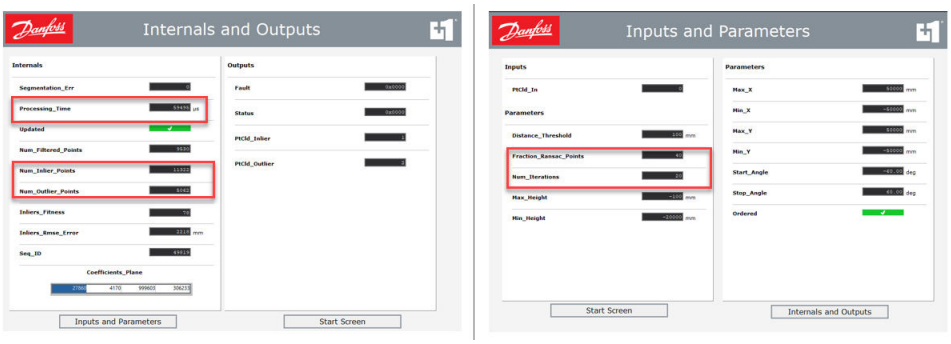
9. Monitor the **Fault** and **Status** codes to see if errors occur. No errors for each will read 0x0000.
10. Skip **PtCld_Inlier** and **PtCld_Outlier**. These are the data locker IDs and are assigned automatically.

Reduce Processing Time

Make adjustments to reduce the processing time for **Planar_Surface_Segmentation**. Use one **Planar_Surface_Segmentation** function block instead of multiple blocks for the best processing times.

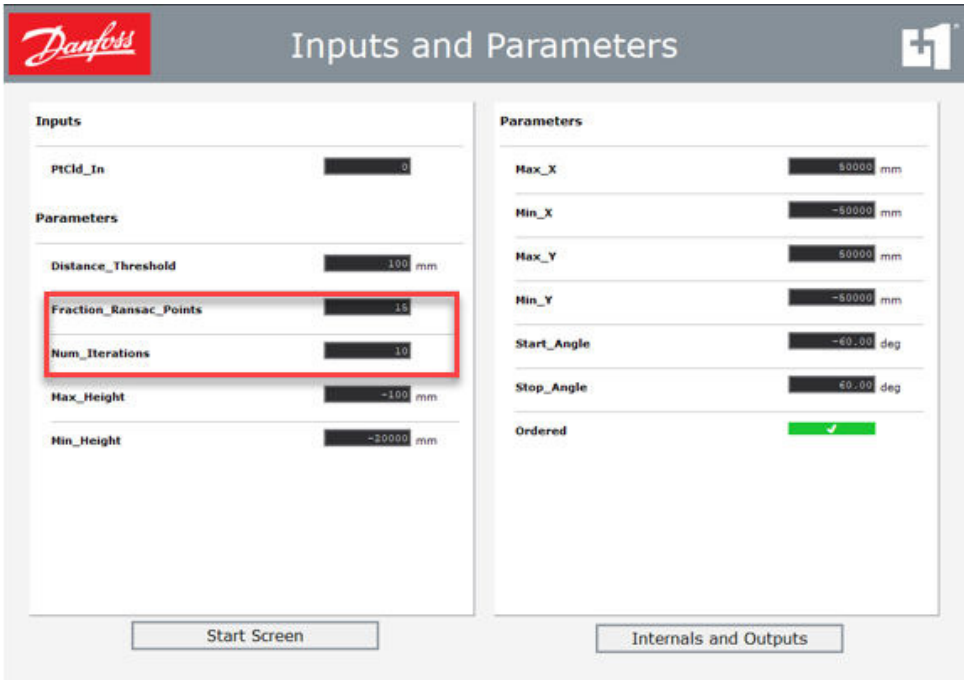
A high **Processing_Time** appears as **Segmentation_Err** number 3. Zero indicates no errors, but the processing time could still be reduced without compromising the accuracy.

1. Take a baseline measurement of the **Processing_Time** found in the service tool screen.



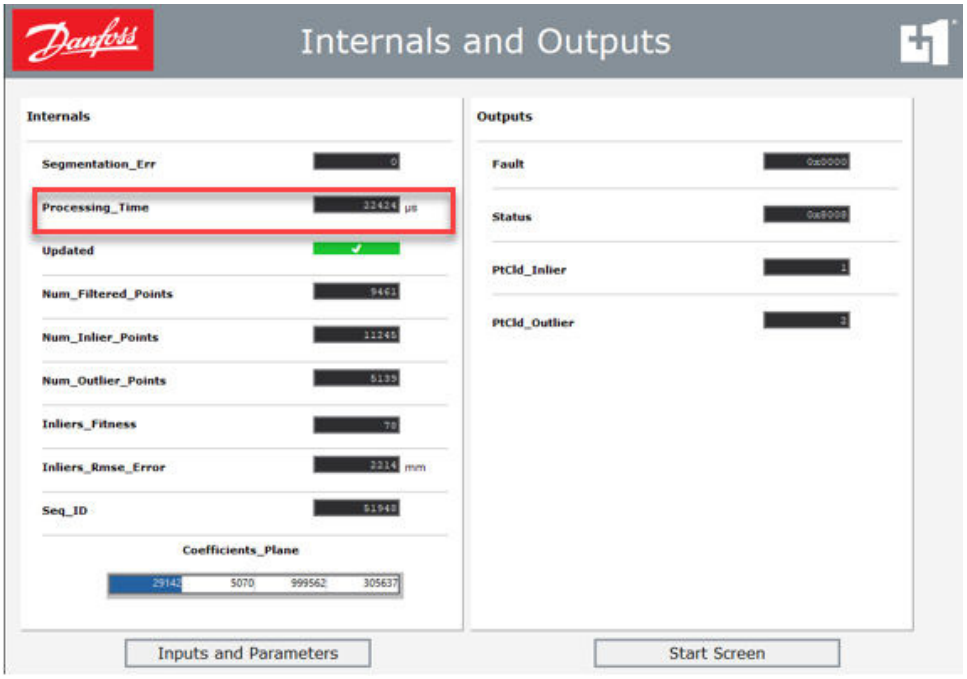
a) Record the **Processing_Time** with the initial **Num_Iterations** and **Fraction_Ransac_Points** parameters.
b) Record the initial **Num_Inlier_Points** and **Num_Outlier_Points** values.

2. Reduce the **Num_Iterations** and **Fraction_Ransac_Points** parameter values.



Planar_Surface_Segmentation Function Block

3. Record the processing time again after reducing the parameters.



4. Compare the new processing time with the old. A significant reduction in time improves efficiency but could limit accuracy. Here, the processing time decreased from about 60000 to 22500 microseconds.
5. Ensure that **Num_Inlier_Points** and **Num_Outlier_Points** remain close to the original baseline values. This indicates that the data has not been compromised.
6. Continue incrementally adjusting the **Num_Iterations** and **Fraction_Ransac_Points** parameters and recording the results. This iterative approach helps find a balance between processing speed and accuracy.
7. Maintain a record of the **Processing_Time**, **Num_Inlier_Points**, and **Num_Outlier_Points** after each parameter adjustment. This gives a clearer view of the tradeoffs in speed and accuracy.
8. After identifying the optimal settings that offer fastest processing times with accurate results, use these settings for **Planar_Surface_Segmentation**.

Inputs

The following table describes inputs required for the **Planar_Surface_Segmentation** function block. This block requires a LiDAR, LiDAR code such as the **Ouster_LiDAR** function block, and a **Data_Lockers** block. Any block that outputs point clouds can be used as an input to **Planar_Surface_Segmentation**.

Item	Type	Range	Description [Unit]
PtCld_In	S8	-1-99	The point cloud data locker ID where LiDAR scan data is stored. LiDAR data can be either an unordered or ordered point cloud.

Parameters

The following table describes parameters for the **Planar_Surface_Segmentation** function block. These parameters can be hard-coded or come from **Obstacle_Avoidance** or **Obstacle_Detect** function blocks.

Planar_Surface_Segmentation Function Block

Most blocks can take in ordered point cloud data, but only a few blocks can access unordered point cloud data.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Distance_Threshold	U32	0-30000	The threshold distance between the detected plane and the point cloud points. Points with distance value less than the threshold are inlier points. Default: 200 [mm]
Fraction_Ransac_Points	U16	20-65	The fraction of filtered points to be used for detecting the most dominant plane. Default: 40 [%]
Num_Iterations	U16	5-100	The number of iterations to be done to find the most dominant plane. Default: 20 [Iterations]
Max_Height	S32	**Min_Height** + 1 to 50000	The maximum height threshold value to limit the filtered points from the input point cloud. Default: -700 [mm]
Min_Height	S32	-50000 to **Max_Height** - 1	The minimum height threshold value to limit the filtered points from the input point cloud. Default: -20000 [mm]
Max_X	S32	**Min_X** +1 to 5000	The maximum X coordinate threshold value to limit the filtered points from the input point cloud. Default: 5000 [mm]
Min_X	S32	-50000 to **Max_X** -1	The minimum X coordinate threshold value to limit the filtered points from the input point cloud. Default: -50000 [mm]
Max_Y	S32	**Min_Y** +1 to 50000	The maximum Y coordinate threshold value to limit the filtered points from the input point cloud. Default: 50000 [mm]
Min_Y	S32	-50000 to **Max_Y** -1	The minimum Y coordinate threshold value to limit the filtered points from the input point cloud. Default: -50000 [mm]
Start_Angle	S16	-18000 to **Stop_Angle** -1	The minimum azimuth angle threshold value to limit the filtered points from the input point cloud. Default: -18000 [0.01 degree]
Stop_Angle	S16	**Start_Angle** +1 to 18000	The maximum azimuth angle threshold value to limit the filtered points from the input point cloud. Default: 18000 [0.01 degree]
Ordered	BOOL	T/F	Determines whether the block outputs ordered or unordered point cloud data. This is only valid if **PtCld_In** is ordered. T: **PtCld_Outlier** and **PtCld_Inlier** are both ordered. F: **PtCld_Outlier** and **PtCld_Inlier** are both unordered.

Planar_Surface_Segmentation Function Block

Outputs

The following table describes outputs for the **Planar_Surface_Segmentation** function block. The point cloud is divided into two sections: inlier and outlier data.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
PtCld_Out	BUS	——	The output bus containing point cloud data.
PtCld_Outlier	S8	-1-99	The point cloud data locker ID where outlier point cloud data is stored. [Locker ID]
PtCld_Inlier	S8	-1-99	The point cloud data locker ID where inlier point cloud data is stored. [Locker ID]

Planar_Surface_Segmentation Troubleshooting

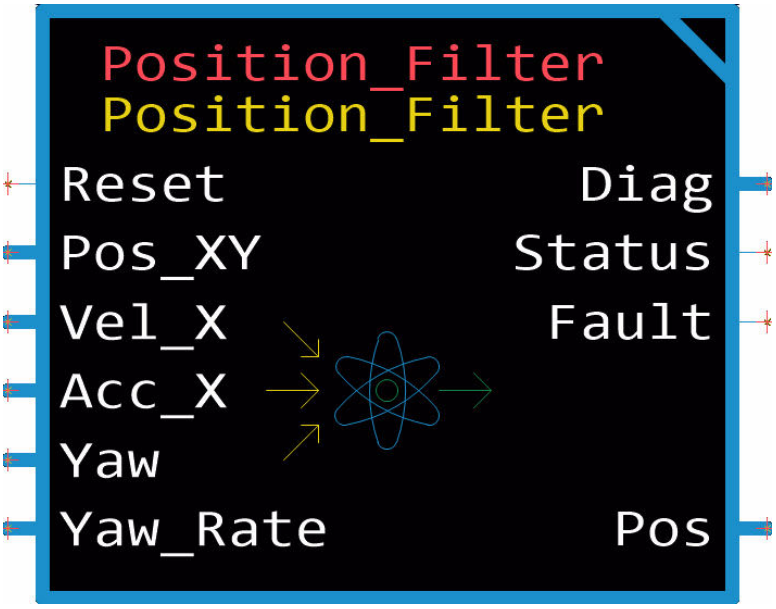
The following table describes errors that could occur in the **Planar_Surface_Segmentation** function block, as well as ways to fix them. View the **Segmentation_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Segmentation_Err Descriptions and Fixes

Error Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
2	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.
3	A background thread took more than 500 milliseconds to process the data. However, a valid output is still coming out.	Adjust the block parameters to speed up processing time. Reduce the size of the input point cloud. For example, use the **LiDAR_Mask** block or decrease the resolution of your LiDAR driver. If the processing time takes more than 10 seconds, turn the controller off and on.
4	The start angle is smaller than the LiDAR hardware minimum measuring angle. For example, the LiDAR hardware measures -60 to 60 degrees, but the block is told to measure -90 to 90 degrees, outside of the hardware's minimum angle (-60 degrees).	Adjust the **Start_Angle** to be within the LiDAR hardware's minimum measuring angles.
5	The stop angle is larger than the LiDAR hardware maximum measuring angle. For example, the LiDAR hardware measures -60 to 60 degrees, but the block is told to measure -90 to 90 degrees, outside of the hardware's maximum angle (60 degrees).	Adjust the **Stop_Angle** to be within the LiDAR hardware's maximum measuring angles.
6	The difference between the **Start_Angle** and **Stop_Angle** are smaller than the Azimuth resolution of the LiDAR hardware.	Adjust the **Start_Angle** and **Stop_Angle** to be more than the Azimuth resolution of the LiDAR hardware.
7	There are less than 100 points for the block to work, so the point cloud is lost.	Choose parameters such that the filtered data contains more than 100 points of point cloud data: Height_Max, Height_Min, X_Max, X_Min, Y_Max, Y_Min, Stop_Angle, and Start_Angle, the **Distance_Threshold** parameter.
8	The block took in unordered point cloud data and was asked to create ordered point cloud data. Blocks can only create unordered point cloud data if that is what they take in.	Check the **Ordered** parameter. If input point cloud data is unordered, this flag must be false.

Position_Filter Function Block

The **Position_Filter** function block produces a single position estimate from the position, velocity, and acceleration inputs. The filter then fuses these sensor measurements together to create an improved localization estimate.



The **Position_Filter** function block can handle input data coming in at different rates and varying levels of sensor accuracy. Each sensor input includes a **Std_Dev** (standard deviation) signal that describes the amount of potential error in a given reading. This helps the block appropriately filter the incoming signals based on this level of confidence.

The block receives absolute data from a Global Navigation Satellite System (GNSS) unit or LiDAR. It receives odometry data from relative sensors such as an Inertial Measurement Unit (IMU), or velocity sensor such as a Pulse Pick-up Unit (PPU). Then, the block uses position filtering to combine those inputs to determine the location of the machine.

Position_Filter uses an Extended Kalman Filter mathematical equation, which fuses all the data together to get a more accurate understanding of the position than one sensor could alone. It records GNSS (location error), wheel odometer (speed error + angle error), and yaw source (direction error). The output is a relative position.

The **Yaw** input could come from the **Yaw_Estimate** function block, **Yaw Rate** input could come from the **Ackermann_Yaw_Rate** function block, and **Pos_XY** could come from **Relative_Pos** function block.

If the machine drives in reverse and does not have a smart antenna, program logic to flip the yaw between the **Yaw_Estimate** function block output and **Position_Filter** input.

Inputs

Inputs to the **Position_Filter** function block are described.

Item	Type	Range	Description [Unit]
Reset	BOOL	T/F	Completely resets all values stored in the function block. T: Reset all values in the function block. F: Do not reset values.
Pos_XY	BUS	——	This bus contains X (Easting), Y (Northing) and its standard deviations.
X	S32	-2147483648-2147483647	Current X position of the machine location. Keep the X position within -16 to 16 kilometers. [mm]

Position_Filter Function Block

Item	Type	Range	Description [Unit]
Y	S32	-2147483648-2147483647	Current Y position of the machine location. Keep the Y position within -16 to 16 kilometers. [mm]
X_Std_Dev	U32	1-4294967295	Standard deviation of X. [mm]
Y_Std_Dev	U32	1-4294967295	Standard deviation of Y. [mm]
Updated	BOOL	T/F	True when there is new X and Y values. Use the new X and Y values. Do not use the new X and Y values.
Vel_X	BUS	——	This bus contains velocity data and the standard deviation for it.
VelX	S32	-100000-100000	The linear velocity of the machine. [mm/s]
VelX_Std_Dev	U32	1-4294967295	The standard deviation of VelX. [mm/s]
Updated	BOOL	——	True when there is new data. T: New data is ready. F: New data is not ready.
AccX	BUS	——	Contains acceleration data and the standard deviation for it.
AccX	S32	-100000-100000	Acceleration in the direction of the X-axis, forward direction for the machine. [mm/s^2]
AccX_Std_Dev	U32	1-4294967295	The standard deviation of AccX. [mm/s^2]
Updated	BOOL	——	True when there is new data. T: New data is ready. F: New data is not ready.
Yaw	BUS	——	Contains the Yaw data and the standard deviation for it.
Yaw	S32	-72000-72000	The angle used to describe the machine's heading using the ENU (East-North-Up) reference frame. [0.01 degree]
Yaw_Std_Dev	U32	1-4294967295	The standard deviation of the filtered Yaw value. [0.01 degree]
Updated	BOOL	——	True when there is new data. T: New data is ready. F: New data is not ready.
Yaw_Rate	BUS	——	The standard deviation of Yaw_Rate.
Yaw_Rate	S32	-1312080-1312080	The angular velocity of the machine relative to the machine's vertical axis. [0.01 deg/s]
Yaw_Rate_Std_Dev	U32	1-4294967295	The standard deviation of Yaw_Rate. [0.01 deg/s]
Updated	BOOL	T/F	True when new data is available from the conversion. T: New data is available. F: New data is not available.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Position_Filter Function Block

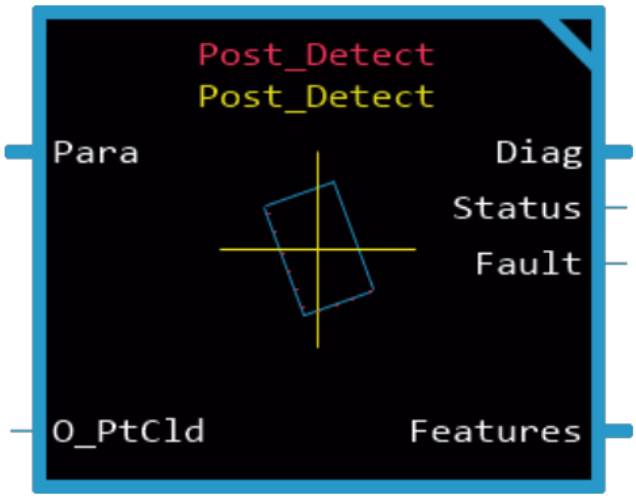
Outputs

Outputs of the **Position_Filter** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. *Non-standard 0x0000: Status OK. 0x0001: Computation error. If this error persists, reset the block. 0x0002: Bad microsecond timer value. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Pos	BUS	——	This bus contains all of the output data for the block.
X	S32	-2147483648-2147483647	The filtered X value out of the block. [mm]
Y	S32	-2147483648-2147483647	The filtered Y value out of the block. [mm]
Yaw	S32	-72000-72000	The filtered Yaw value out of the block. This uses the ENU convention. [0.01 deg]
X_Std_Dev	U32	1-4294967295	The standard deviation of the filtered X value. [mm]
Y_Std_Dev	U32	1-4294967295	The standard deviation of the filtered Y value. [mm]
Yaw_Std_Dev	U32	1-4294967295	The standard deviation of the filtered Yaw value. [0.01 deg]

Post_Detect Function Block

The **Post_Detect** function block searches an incoming LiDAR scan for features that match the profile of a rectangular post and outputs the locations and properties of up to 100 matching posts.



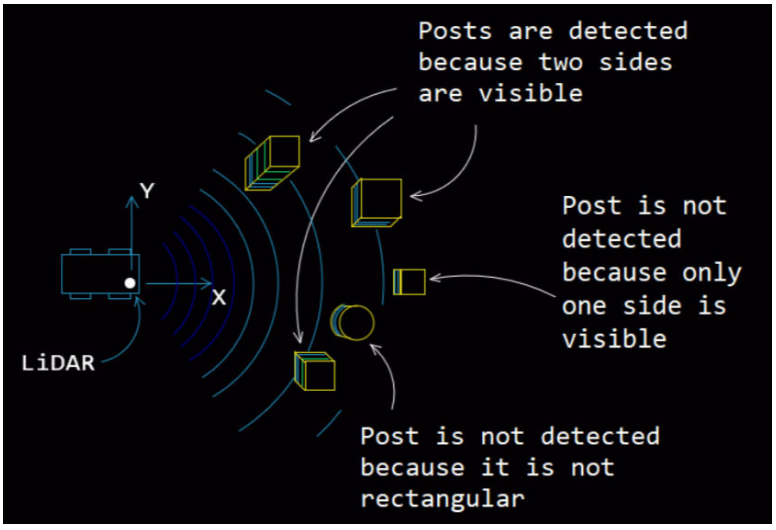
This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about the Ouster LiDAR scanner and block. Other types of sensors do not work as well as LiDARs.

Use **Post_Detect** if the environment surrounding the machine contains posts of similar dimension. Machines receive the position of the posts. After that, other code could utilize the post position so machines could move between them. See *Local Coordinate System* on page 17.

Post criteria:

- Posts must be three dimensional, not two dimensional. Two sides of the post must be visible to the LiDAR scanner.
- Posts must be rectangular, not cylindrical. Visible sides of the post should form a 90 degree angle, with a 5 degree angular tolerance. Anything larger than a 15 degree angular tolerance will never be detected.
- Multiple posts must have the same length and width as other posts.
- Posts must be at least 500 mm apart from an object, taking depth into account from the LiDAR's perspective. A post against a wall will only be detected if the distance between the two is greater than 500 mm.
- Height of the post is not a factor.
- Posts must be vertical with respect to the LiDAR's z-axis. Tilt the LiDAR to the appropriate angle to detect horizontal posts.
- Posts tilted within 5 degrees with respect to the z-axis are detected, and more than 15 degrees are never detected.
- At least two LiDAR rings must land on the post for it to be detected.

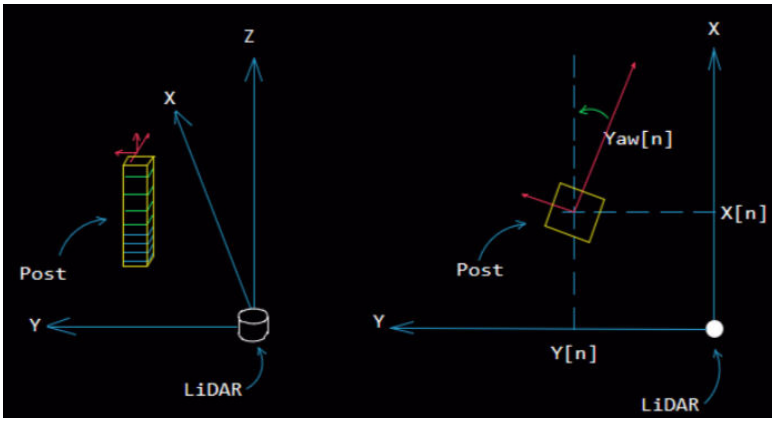
Post_Detect Function Block



The image above depicts some criteria required for post detection.

Posts can be spaced apart at different intervals within the range of the LiDAR, as long as the posts have the same length and width. Use another **Post_Detect** function block for posts with different length and width dimensions, or set the **Tolerance** parameter high to detect posts with different dimensions.

Ordered point cloud data from a LiDAR enters **Post_Detect** as an input. Then, **Post_Detect** gives the 2D (X, Y) position of the post center, along with the width and standard deviation of the post. Standard deviation in **Post_Detect** refers to the error in the (X, Y) position of the post center, along with other uncertainty. This block also gives the **Yaw** of the post, referring to the rotation of the post itself and not where the post is located around the LiDAR.

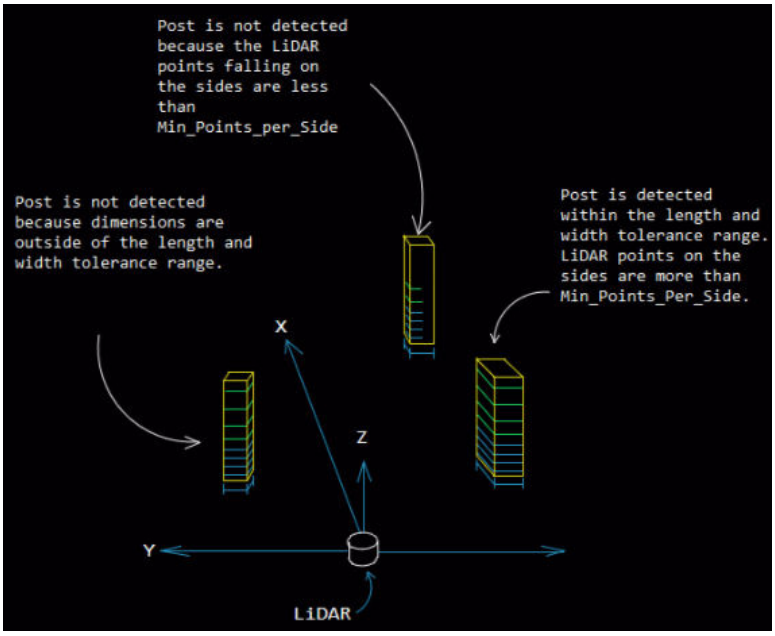


The image above shows the yaw rotation of the post.

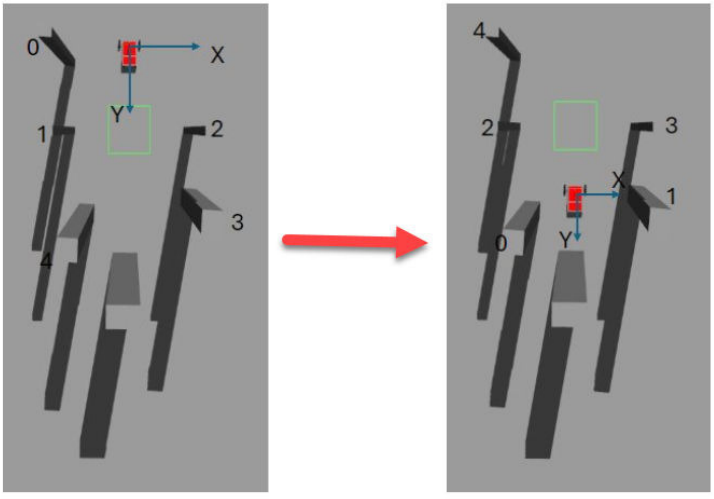
When programming, put in the expected width and length of the posts. Big or small numbers do not matter. If the posts vary in width or length, set a high **Tolerance** for the block to detect the posts. A **Tolerance** of zero means the **Post_Width** and **Post_Length** must exactly match the parameter number entered, but a tolerance of 10 mm means the block detects a post measuring 10 mm larger or smaller than both the length and width parameter number entered. **Post_Detect** ignores posts measuring outside of that range.

Additionally, enter the number of LiDAR points needed to detect a post into **Min_Points_Per_Side**. Fewer LiDAR points allow LiDARs to detect posts far away from it, but larger numbers improve accuracy of post detection. The number of LiDAR points applies to both the length and width surfaces of the rectangular post.

Post_Detect Function Block



As the LiDAR moves, it counts the number of posts and gives that number in **Num_Features**. Posts out of range of the LiDAR drop off the numbered list, so the **Num_Features** output may always change. The LiDAR reads the closest post to itself as the first post, and numbers other posts depending on how close they are to itself. When the LiDAR moves closer to a different post, it now reads that new post as the first post, so the number assigned to a post always changes as the LiDAR moves. Lower numbers indicate more confidence in the data because they are for the closest posts to the LiDAR.



The images depict a machine moving through many posts. In **Post_Detect**, the post numbers correspond with how close the post is to the LiDAR. Height is not a factor.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

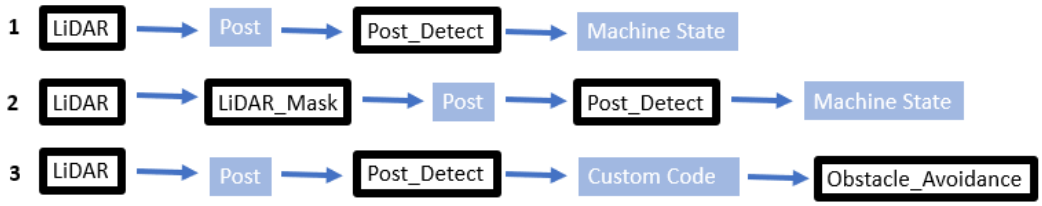
This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Post_Detect Function Block

Application Information

Common function blocks used with the **Post_Detect** function block are **Data_Lockers** and **LiDAR_Mask**. **Post_Detect** also requires LiDAR hardware with accompanying code.

Post_Detect works well in a localization application. See *Local Coordinate System* on page 17.



1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as the Ouster LiDAR hardware and the **Ouster_LiDAR** function block. This gathers point cloud data about the environment, including seeing posts. **Post_Detect** processes the data about the posts. Program the machine to react depending on any posts detected, such as moving toward or away from the posts.
2. Scenario two includes the **LiDAR_Mask** function block after the LiDAR code, followed by **Post_Detect**. **LiDAR_Mask** omits certain data from the LiDAR's point cloud, saving processing time. Then, the machine can be programmed to react a certain way based on whether any posts are detected.
3. Scenario three includes custom code after **Post_Detect**, which tell the block how to process the data for the machine to avoid objects in **Obstacle_Avoidance**.

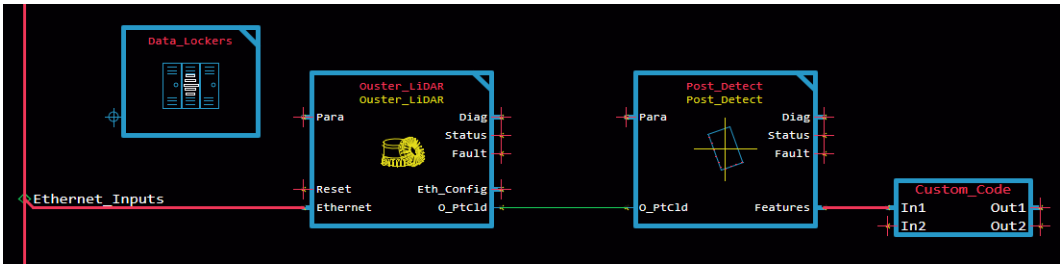
Additionally, place the **Post_Detect** function block:

- After LiDAR data is collected, such as after the **Ouster_LiDAR** function block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.

Post_Detect is not usually used with other detection function blocks, such as **Obstacle_Detect** or **Reflector_Detect**.

Example

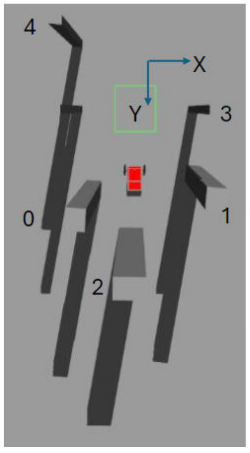
This example shows the **Post_Detect** function block used as if a machine needs to drive past a series of posts.

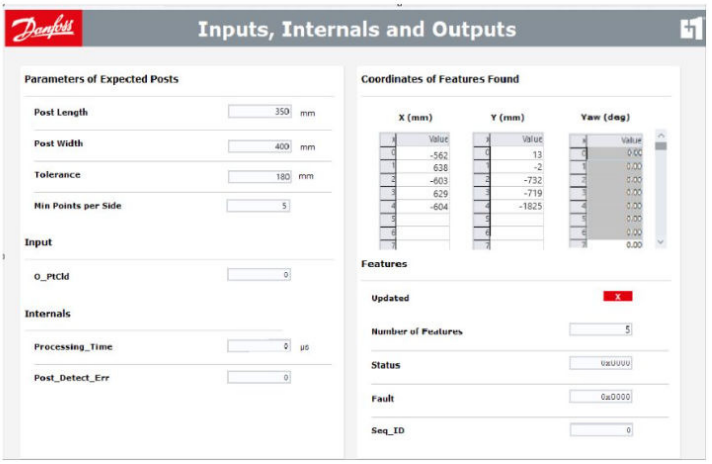


1. Set up the LiDAR hardware and accompanying code. This example includes the **Ouster_LiDAR** function block and Ouster LiDAR hardware. See the *Plus+1 Compliant Ouster Block User Manual* for more information. The LiDAR hardware needs to see where the expected posts would be detected and the code set up to include point cloud information from that area. If the LiDAR code is programmed so the range the LiDAR sees is too small, then the code for the LiDAR must be changed in order for **Post_Detect** to work.
2. Add the **Post_Detect** function block. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
3. Outside of PLUS+1® GUIDE, determine physically in the environment where posts should be detected. Posts are detected with respect to the machine as the origin. See *Local Coordinate System* on page 17. Point the LiDAR laser toward the posts to begin tuning.
4. Measure the length and width of posts in the environment. Length and width refer to the X and Y position in 2D. Height is not a factor.

Post_Detect Function Block

5. Enter these numbers into the **Post_Width** and **Post_Length** parameters. If many posts with different sizes need to be detected by the LiDAR, measure several posts to get an average, or enter the numbers for the largest post. Here, these values are 350 mm and 400 mm.





The image shows a machine driving by posts (left) and the pre-made service tool screen with values (right).

6. Enter the **Tolerance**, which allows **Post_Detect** to detect posts that are larger and smaller than the length and width by factoring errors in posts tilting, post size difference, and LiDAR errors. Here, **Tolerance** is 180 mm, so the LiDAR detects posts with lengths ranging from 170-530 mm and widths ranging from 220 - 580 mm.
7. Tune the **Min_Points_Per_Side** to indicate how many LiDAR points are required to land on the post to detect it. Here, 5 LiDAR points must land on the length side of a post and 5 points must land on the width side.
8. Optionally, visually see the data from **Post_Detect** on the *Pre-Made Service Tool Screens* on page 25, or view each signal individually.
   a) The **O_PtCld** indicates the number of LiDAR data points entering **Post_Detect**. These points could land on posts or anything else, as long as they are valid points. Invalid points go into the sky or the dark and do not collect data.
   b) The **Processing_Time** indicates how long LiDAR data takes to enter the **Post_Detect**, process inside of it, and output the detected post information.
   c) **Post_Detect_Err** indicates if an error occurs, and if so, what the error is by providing an error number. Here, 0 indicates there is no error and everything is behaving normally.
   d) **X** and **Y** indicate where the center of each post is located in relation to the LiDAR using the right-hand rule. Here, the post closest to the LiDAR, labeled 0 in the image, is 562 mm to the left and 13 mm ahead of the LiDAR. The second post, labeled as 1 in the image, is 638 mm to the right and 2 mm behind the LiDAR. How the LiDAR is mounted factors into the position.
   e) **Yaw** indicates how much the post itself is rotated. Here, that is not a factor.
   f) The **Updated** box appears green when information passes into **Post_Detect** and red when information does not come in.
   g) **Number of Features** shows the number of posts detected by the LiDAR because they match the post criteria. The first post data goes into the 0 spot of the array, the second post goes into the 1 spot, and so on. Here, 5 posts are detected.
   h) **Status** and **Fault** indicate if any issues occur. Here, there are no issues.
   i) **Seq_ID** shows which LiDAR frame the data came from. This could be used to correct for machine motion during processing.
9. After tuning **Post_Detect**, create custom code for a machine to drive toward a post.

Post_Detect Function Block

Inputs

Inputs to the **Post_Detect** function block are described.

Item	Type	Range	Description [Unit]
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Chkpt	BOOL	——	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

The following table describes parameters for the **Post_Detect** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Post_Width	U16	0-1000	Expected width of post to detect. Default: 250 [mm]
Post_Length	U16	0-1000	Expected length of post to detect. Default: 250 [mm]
Tolerance	U16	0-1000	Allows the block to detect posts with length and width greater than what was entered in **Post_Width** and **Post_Length**. Applies to both equally. Default: 0 [mm]
Min_Points_Per_Side	U16	2-1000	Minimum number of LiDAR points detected on the length and width sides of a post to consider it valid. Default: 5

Outputs

Outputs of the **Post_Detect** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Processing_Time	U32	0-4294967295	Time taken to process input point cloud data. [µs]
Post_Detect_Err	U8	0-4	Indicates errors occurred in the function block operation. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: Point cloud is unordered.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.

Post_Detect Function Block

Item	Type	Range	Description [Unit]
Features	BUS	——	This bus contains information about features identified in the LiDAR scan that match the criteria of a post. This includes the number of posts found, as well as the yaw, X coordinate, and Y coordinate of each post.
Num_Features	U16	0-100	Number features defines the number of valid posts found in the LiDAR scan. The valid elements of **X**, **Y**, and **Yaw** are from index 0 to Num_Features-1. Any elements after this are invalid and will be ignored.
Updated	BOOL	T/F	Indicates that new data is available. T: New data is available. F: New data is not available.
X	(Array[100]S32)	-2147483648-2147483647	X component of the center of the posts using the right-hand rule. [mm]
Y	(Array[100]S32)	-2147483648-2147483647	Y component of the center of the posts using the right-hand rule. [mm]
Yaw	(Array[100]S16)	-18000-18000	Rotation of the posts themselves. This is not the polar angle of the post with respect to the LiDAR. [0.01 deg]
Std_Dev	(Array[100]U16)	0-65535	Standard deviation refers to the errors in the post center position, along with other uncertainty. Higher values indicate a poor fit.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Post_Detect Troubleshooting

The following table describes errors that could occur in the **Post_Detect** function block and ways to fix them. View the **Post_Detect_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

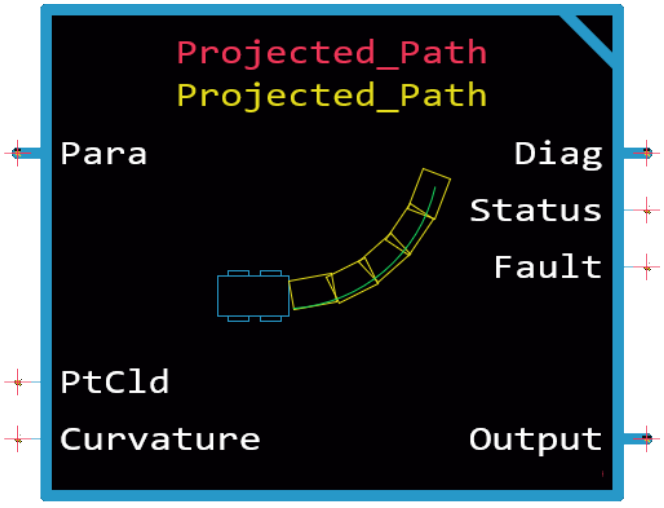
Processing_Time should be less than one microsecond.

Post_Detect_Err Descriptions and Fixes

Number	Description	How to Fix
0	There is no error.	Nothing needs to change.
1	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
2	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.
3	Thread timeout. The controller could be overloaded, which would affect many blocks.	Reduce the LiDAR resolution or delete other processing blocks.
4	Point cloud is unordered. This means the data changed from ordered to unordered so the structure of the point cloud data is lost.	Check the code to see if the data was filtered or modified to become unordered, and switch to the ordered output option.

Projected_Path Function Block

The **Projected_Path** function block detects objects ahead of a machine along a curved path.



Projected_Path requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about LiDARs, the Ouster LiDAR hardware, and **Ouster_LiDAR** function block.

Projected_Path behaves similarly to **Projected_Path_Area**, except it only detects objects within the zones rather than their cross-sectional area. See *Projected_Path_Area Function Block* on page 191 to read about the functionality, application recommendations, and an example. Some parameters and outputs change due to the area calculations.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Inputs

Inputs to the **Projected_Path** function block are described.

Item	Type	Range	Description [Unit]
PtCld	S8	-1-99	The data locker ID of ordered or unordered point cloud data.
Curvature	S32	-800000-800000	Curvature of the circle where the zones to be evaluated are positioned. [0.01/km]
Chkpt	BOOL	——	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Projected_Path Function Block

Parameters

The following table describes parameters for the **Projected_Path** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Width	U16	0-65535	Width of the zones. All zones have the same width. Default: 1000 [mm]
Min_Distance	U16	0 to **Max_Distance**-1	Distance between the steering point of the machine and the start of the first zone. Create enough distance to avoid the zone overlapping the machine. Default: 0 [mm]
Max_Distance	U16	**Min_Distance**+1 to 65535	Distance between the steering point of the machine and the end of the last zone. Default: 1000 [mm]
Min_Height	S32	-50000 to **Max_Height**-1	Minimum height of the zones with respect to the steering point. All zones have the same height. Default: 0 [mm]
Max_Height	S32	**Min_Height**+1 to 50000	Maximum height of the zones with respect to the steering point. All zones have the same height. Default: 1000 [mm]
Sensor_Orientation	S16	-18000-18000	Rotation of the LiDAR scanner around the z-axis in relation to the machine. Default: 0 [0.01 deg]
Sensor_Offset_X	S32	-2147483648-2147483647	The distance from the steering point of the machine along the x-axis to the LiDAR scanner. LiDARs in front of the steering point have positive values and in back have negative values. Default: 0 [mm]
Sensor_Offset_Y	S32	-2147483648-2147483647	The distance from the steering point of the machine along the y-axis to the LiDAR scanner. LiDARs to the left of the steering point have positive values and to the right have negative values. Default: 0 [mm]
Sensor_Offset_Z	S32	-2147483648-2147483647	The distance from the steering point of the machine along the z-axis to the LiDAR scanner. LiDARs above the steering point have positive values and below have negative values. Default: 0 [mm]

Outputs

Outputs of the **Projected_Path** function block are described.

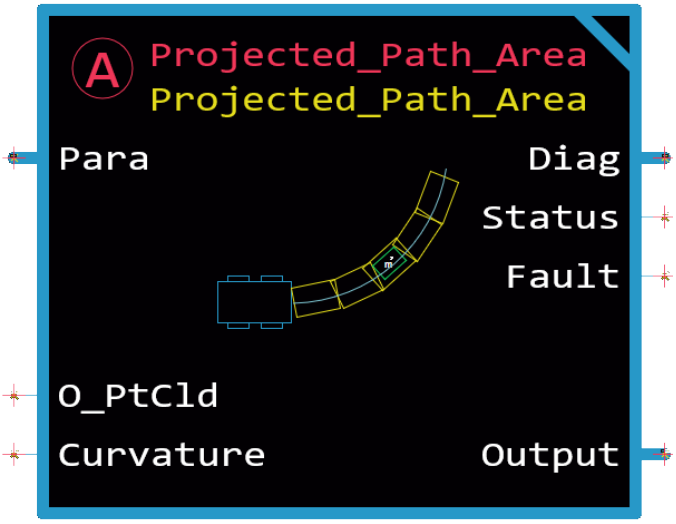
Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.

Projected_Path Function Block

Item	Type	Range	Description [Unit]
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x0001: Input value too low. 0x8002: Input value too high.
Output	BUS	——	The Output bus contains the score information for each zone being evaluated.
Updated	BOOL	T/F	New information is available from the block. T: New data is available. F: New data is not available.
Scores	(Array[5]U32)	0-4294967295	The number of valid LiDAR points within each zone box.
Total_Valid_Points	U32	0-4294967295	The number of valid LiDAR points. These obtain data when landing on objects. An unusually low number may indicate issues. Invalid points include LiDAR points going into the sky or dark surfaces, which are not detected by the LiDAR. For ordered point clouds, this is the number of valid LiDAR points found within and close to the zones. LiDAR points are counted multiple times if zones overlap. For unordered point clouds, this is the number of valid LiDAR points the LiDAR sees in the whole point cloud, regardless of zones.
Sum_Scores	U32	0-4294967295	Sum of all counts of points within all zones.
Highest_Score	U32	0-4294967295	Count of points within the zone that had the most points.
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Projected_Path_Area Function Block

The **Projected_Path_Area** function block shows a machine what is ahead of itself, allowing it to react based on how close an object is to the machine. The block detects the cross-sectional area of objects along a curved path.



This block requires a license for A+ Advanced.

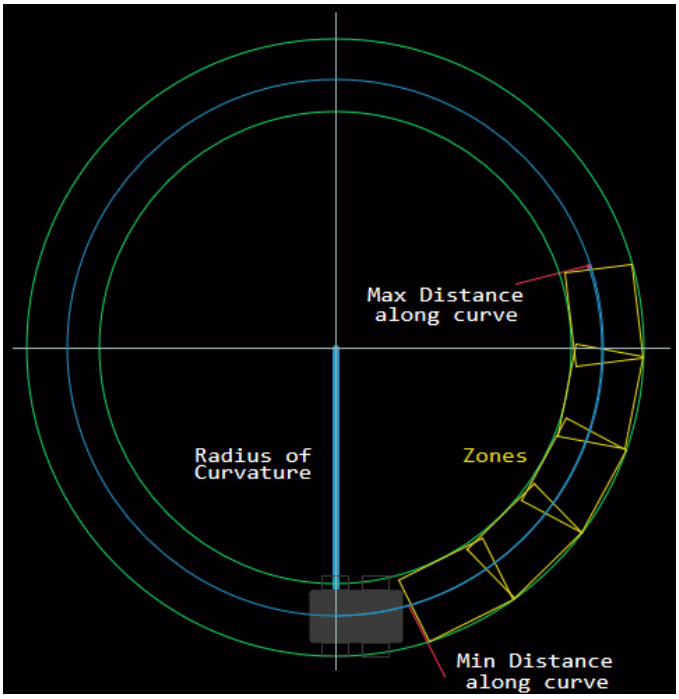
Projected_Path_Area requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about LiDARs, the Ouster LiDAR hardware, and **Ouster_LiDAR** function block.

Use this block to decelerate or stop a machine. The type of machine does not matter. Ideal environments include smooth rather than bumpy surfaces or the LiDAR could pitch too much and give bad data from the zones. The code works better with gradual turns rather than sharp turns, depending on machine size and attributes.

Information about the environment in front of the machine enters **Projected_Path_Area** as a point cloud. Additionally, information about the path the machine travels enters the block as a curvature. This allows the machine to see items on the path ahead of itself, and other code determines how the machine reacts.

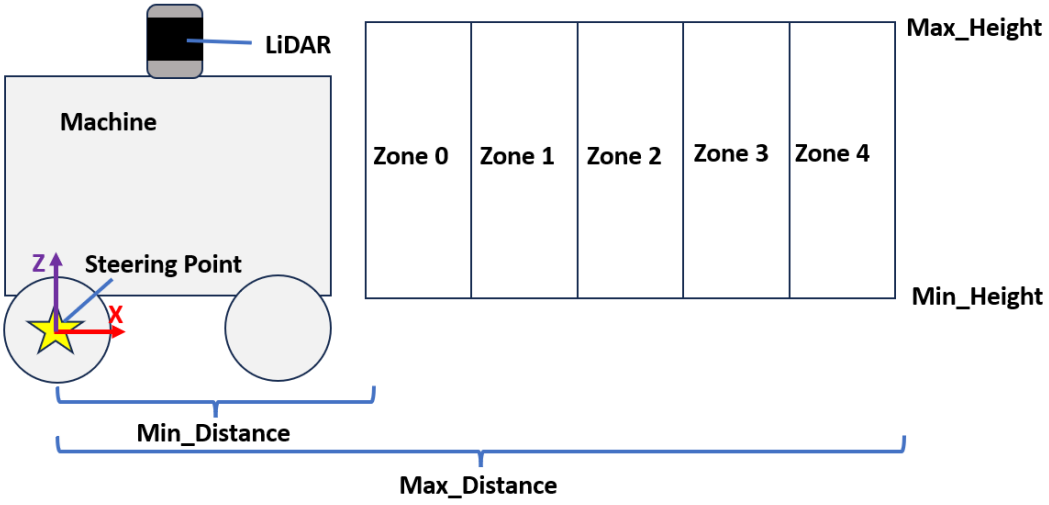
This block detects objects within five zone boxes in front of a machine: 0, 1, 2, 3, and 4. The machine can be programmed to react differently depending on the results of each zone. There will always be five zones, but if a machine does not need all of them, it should ignore the results in the unwanted zones. Then, the machine will not react to anything detected in them.

Projected_Path_Area Function Block



The image above shows a machine traveling along a curved path with five zones ahead of it. Zones overlap along the curve. Objects in multiple zones are counted multiple times.

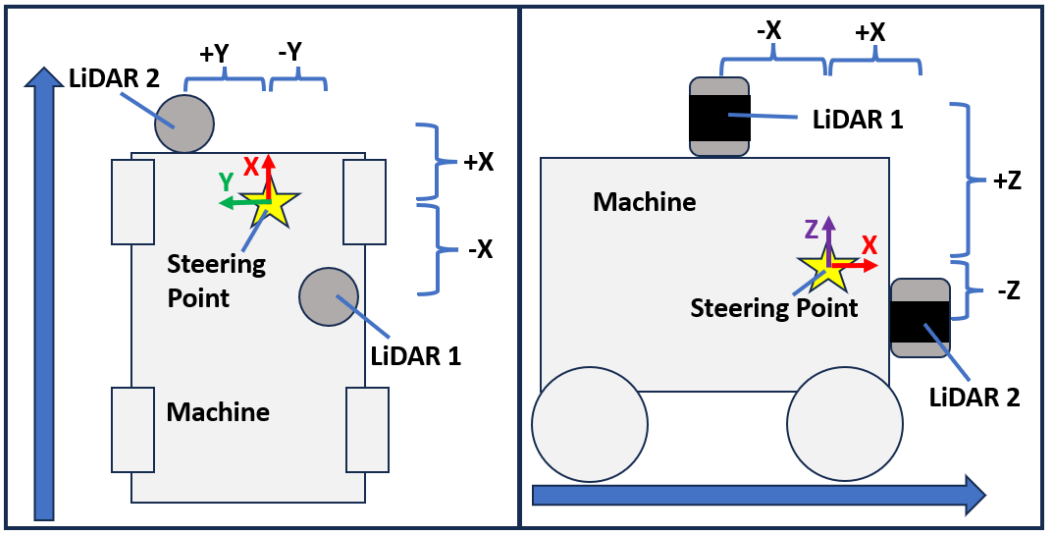
Parameters involve setting up the zone box dimensions. Zones should be wider than the machine to avoid clipping items as the machine turns. Dimensions include the distance where zones begin ahead of the machine and where they end, as well as the zone minimum and maximum height. Zones are in relation to the machine's origin, referred to as the steering point in ACL.



The image above shows a machine with the steering point near the back wheel, a LiDAR scanner on top, and five zones in front. The array of zones begin and end based on the distance from the steering point. The same minimum and maximum height applies to all five zones.

The LiDAR also needs to align with the machine's steering point as the origin. Set alignment measurements in the **Sensor_Offset_X**, **Sensor_Offset_Y**, and **Sensor_Offset_Z** parameters. See *Machine Coordinate System* on page 18.

Projected_Path_Area Function Block



The images above show the top and side view of a rear wheel steering machine with two LiDARs. The LiDARs connect based on the machine's steering point as the origin. LiDARs above, to the left, or ahead of the steering point use a positive distance parameter value. LiDARs below, to the right, or behind the steering point use a negative distance parameter value.

Besides the machine's steering point, the LiDAR mounting position is extremely important. If the LiDAR is rotated so it does not face the front of the machine, that rotation needs to be offset in the **Sensor_Orientation** parameter in order for the LiDAR to see the zones ahead of the machine. For example, if a LiDAR was mounted rotated 45 degrees from the machine front, then enter -45 degrees to offset the rotation to zero. See *Sensor Coordinate System* on page 19.

Projected_Path_Area detects the approximate cross-sectional area of items within the zones ahead of the machine, whereas the simpler **Projected_Path** function block only detects when an object is in the zone, not the size.

Area estimation depends on the number of LiDAR points landing on objects, so neither the surface area nor the exact cross-sectional area can be determined. However, the block calculations adjust for objects far away or close to the LiDAR to give a rough size estimate. For example, if a bird flies close to the LiDAR, lots of LiDAR points land on it, but the block knows it is still small. A building in the distance with a few LiDAR points still reads as a large object. If there are multiple objects within a zone, the **Areas** output combines them all. If objects overlap in zones, they are counted multiple times inside the overlap.

Reflective items around the LiDAR could reduce its ability to estimate the object's cross-sectional area.

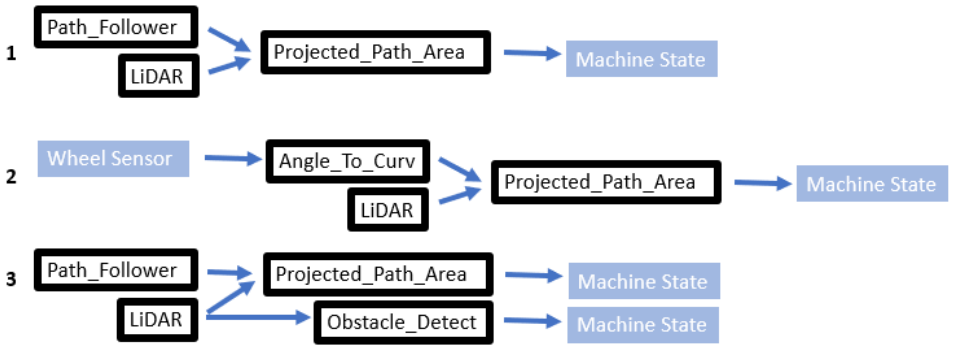
Additionally, **Projected_Path_Area** only takes in ordered point cloud data and not unordered. However, **Projected_Path** supports both ordered and unordered point cloud data.

Application Information

Common function blocks used with the **Projected_Path_Area** function block are **Data_Lockers**, **Path_Follower**, and the **Ouster_LiDAR** function block if using Ouster LiDAR hardware.

Projected_Path_Area always needs a LiDAR and **Data_Lockers**. It usually comes last in a sequence but needs custom code after it telling the machine how to react when something is detected in a zone.

Projected_Path_Area Function Block



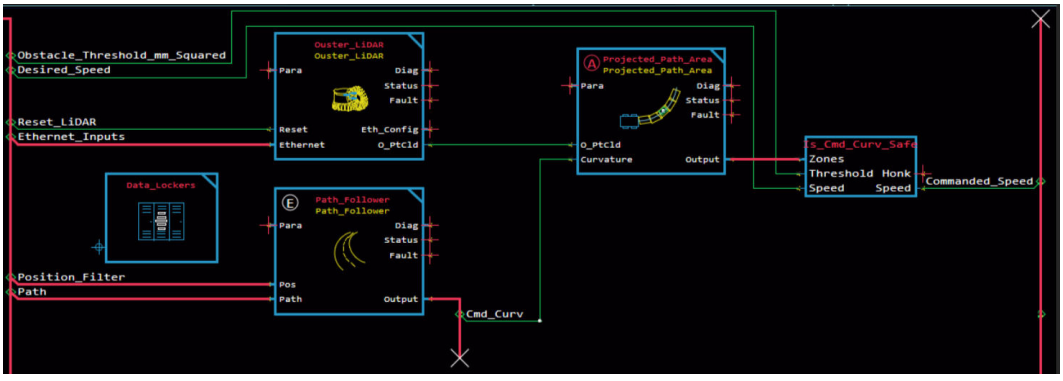
1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as Ouster LiDAR hardware and the **Ouster_LiDAR** function block. This gathers point cloud data which **Projected_Path_Area** uses to detect when something is in a zone. **Path_Follower** runs in parallel. This combination checks if the machine traveling on the path is going to hit something in the zones based on steering.
2. Scenario two shows data from the wheel sensor going into the **Angle_To_Curv** function block, which runs in parallel to LiDAR code. In an Ackermann machine, this combination checks if the machine will hit something in the zones based on where the wheels are pointing. Information comes live from a wheel sensor instead of an actual path. This works with both autonomous and manual operation. The LiDAR detects objects in the zones.
3. Scenario three shows the **Obstacle_Detect** function block, which would use information gathered from the LiDAR, but otherwise runs separately from **Projected_Path_Area**. Use **Obstacle_Detect** if setting up zones outside of where **Projected_Path_Area** covers, such as zones on the machine's side.

Additionally, place the **Projected_Path_Area** function block:

- After LiDAR data is collected, such as after the **Ouster_LiDAR** function block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.

Example

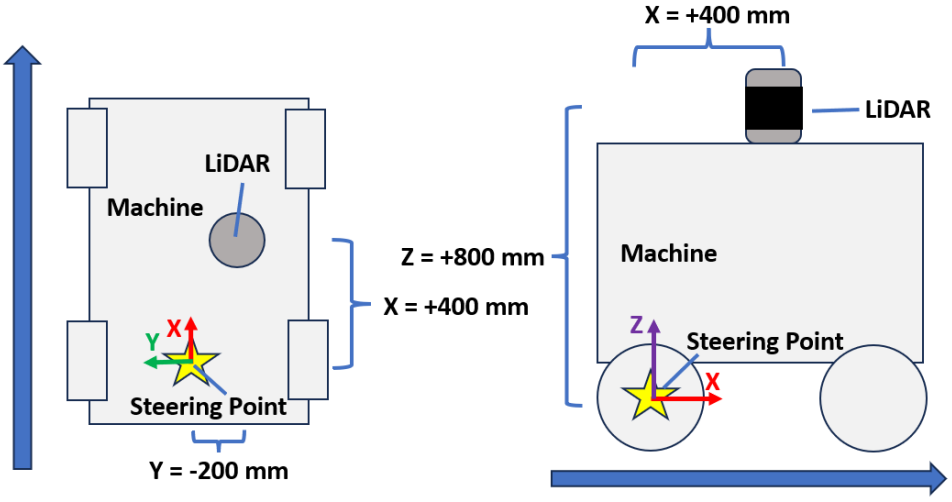
The example shows the **Projected_Path_Area** function block used as if a machine should gradually decelerate and stop when detecting something in the zones in front of it.



1. Set up the LiDAR hardware and accompanying code. This example includes the **Ouster_LiDAR** function block and Ouster LiDAR hardware. See the *Plus+1 Compliant Ouster Block User Manual* for more information. The LiDAR hardware needs to see where the expected obstacles would be detected and the code set up to include point cloud information from that area. If the LiDAR code is programmed so the range the LiDAR sees is too small, then the code for the LiDAR must be changed in order for **Projected_Path_Area** to work.
2. Add the function blocks. Here, the **Ouster_LiDAR** function block connects into **Projected_Path_Area** to send ordered point cloud information from the LiDAR. **Path_Follower** connects into the **Curvature** input to send path information.

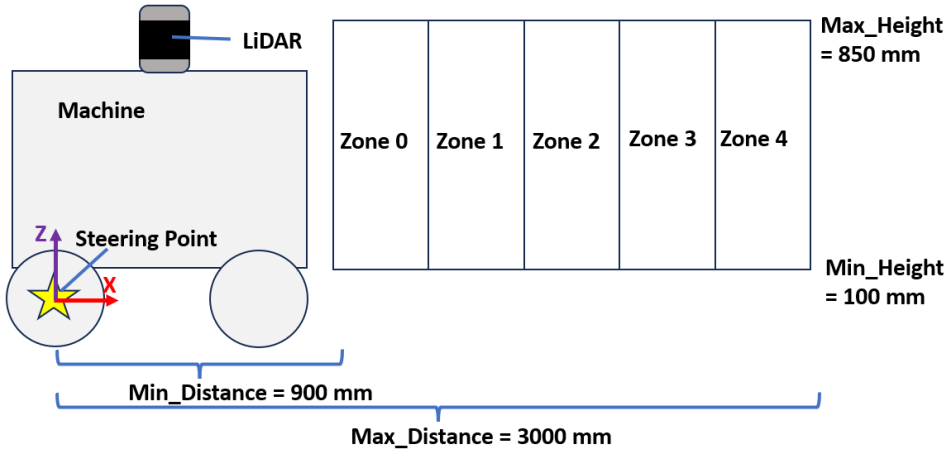
Projected_Path_Area Function Block

3. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
4. Outside of PLUS+1® GUIDE, check that the LiDAR hardware faces the front of the machine without rotation. If rotated, account for the rotation in the **Sensor_Orientation** parameter to bring the rotation degree to zero. The LiDAR should be able to see the zones in front of the machine and may need to be physically moved or have its field-of-view changed in earlier code.
5. Measure the distance from the machine's steering point to the LiDAR eye along the x-axis, y-axis, and z-axis.



The images above depict a top-down and side view of a machine driving in the direction of the arrows. It includes a LiDAR on top and the steering point between the back wheels.

6. Enter the measurements from the steering point to the LiDAR in the sensor offset parameters. Here, **Sensor_Offset_X** is 400 mm in the positive direction, **Sensor_Offset_Y** is -200 mm in the negative direction, and **Sensor_Offset_Z** is 800 mm in the positive direction.
7. Measure from the steering point to where the series of zone boxes should start and end. The zones should avoid overlapping the machine, and they should stay within the LiDAR hardware's range of view. Here, the minimum and maximum distances of the zones are 900 mm to 3000 mm.



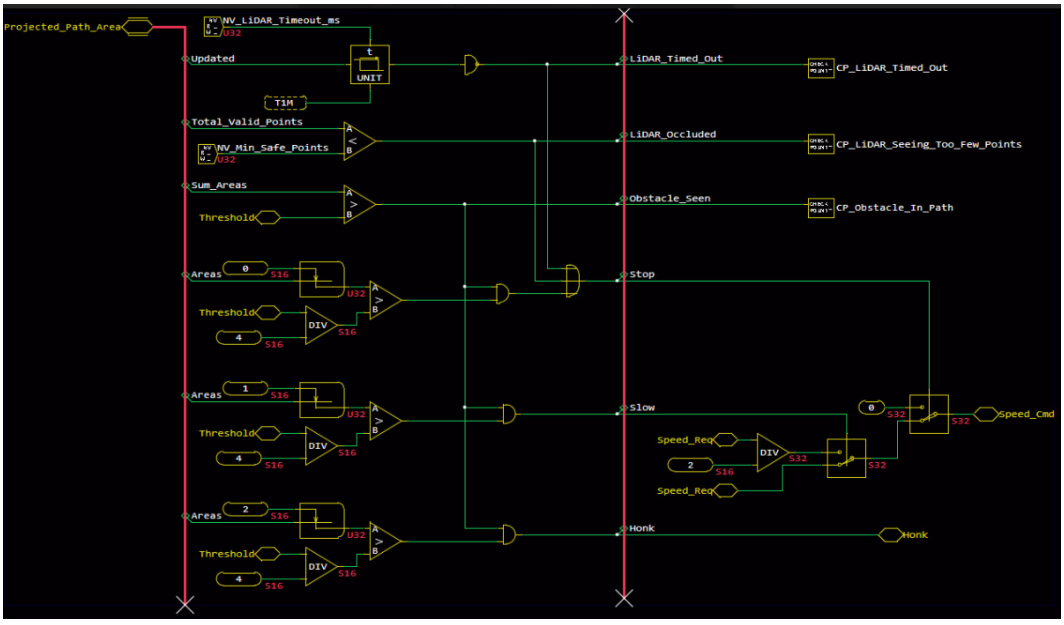
The image above shows a side view of the zones in front of the machine.

8. Enter the desired **Width** of the zones. All zones have the same width, and this should be slightly larger than the machine.
9. Enter the height of the zones, which is in relation to the steering point but also needs to be within the LiDAR scanner's field-of-view. The LiDAR sees the zones and sends information back to the steering

Projected_Path_Area Function Block

point for the machine to react. The height should be high enough off the ground to avoid detecting small items the machine runs over, like gravel. Height should avoid detecting any ceilings and anything hanging. Here, height is 100 mm to 850 mm.
Five zones automatically build after the parameters are filled out. Zones will be slightly curved because they are tied to the wheel curvature. View more detailed zone dimensions in the **Internals** section on the *Pre-Made Service Tool Screens* on page 25.

10. Create code to connect to the **Projected_Path_Area** output.



a) **Update** shows if there is new information processing within **Projected_Path_Area**. Here, the machine will stop if there is no new information for a while.
b) **Total_Valid_Points** is the number of valid LiDAR points in and close to the zones. Here, if this number is very low, then the machine will stop.
c) **Sum_Areas** includes the approximate cross-sectional area of all objects detected in all five zones added together. This helps the machine recognize if there is a large obstacle ahead because it is not cut into smaller sections of the zones. However, many small objects could add up to a big area and give a false positive. Here, the machine will stop if a large obstacle is in the path ahead.
d) **Areas** includes the approximate cross-sectional area of objects within individual zones. Here, the machine stops if it detects an object closest to it in Zone 0. The machine slows down if an object is in Zone 1, and it honks if an object is in Zone 2. The machine does not react if objects are in Zone 3 or 4.
e) Leave any unneeded outputs unconnected.

11. Monitor the **Updated** flag to ensure new information comes through. If this stops updating, then the block is not processing data.

Inputs

Inputs to the **Projected_Path_Area** function block are described.

Item	Type	Range	Description [Unit]
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Curvature	S32	-800000-800000	Curvature of the circle where the zones to be evaluated are positioned. [0.01/km]
Chkpt	BOOL	——	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Projected_Path_Area Function Block

Parameters

The following table describes parameters for the **Projected_Path_Area** function block.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Width	U16	1-20000	Width of the zones. All five zones have the same width. Default: 1000 [mm]
Min_Distance	U16	0 to **Max_Distance**-1	Distance between the steering point of the machine and the start of the first zone. Create enough distance to avoid the zone overlapping the machine. Default: 0 [mm]
Max_Distance	U16	**Min_Distance**+1 to 65535	Distance between the steering point of the machine and the end of the last zone. Default: 1000 [mm]
Min_Height	S32	-50000 to **Max_Height**-1	Minimum height of the zones with respect to the steering point. All zones have the same height. Default: 0 [mm]
Max_Height	S32	**Min_Height**+1 to 50000	Maximum height of the zones with respect to the steering point. All zones have the same height. Default: 1000 [mm]
Sensor_Orientation	S16	-18000-18000	Rotation of the LiDAR scanner around the z-axis in relation to the machine. Default: 0 [0.01 deg]
Sensor_Offset_X	S32	-2147483648-2147483647	The distance from the steering point of the machine along the x-axis to the LiDAR scanner. LiDARs in front of the steering point have positive values and in back have negative values. Default: 0 [mm]
Sensor_Offset_Y	S32	-2147483648-2147483647	The distance from the steering point of the machine along the y-axis to the LiDAR scanner. LiDARs to the left of the steering point have positive values and to the right have negative values. Default: 0 [mm]
Sensor_Offset_Z	S32	-2147483648-2147483647	The distance from the steering point of the machine along the z-axis to the LiDAR scanner. LiDARs above the steering point have positive values and below have negative values. Default: 0 [mm]

Outputs

The following table describes outputs for the **Projected_Path_Area** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.

Projected_Path_Area Function Block

Item	Type	Range	Description [Unit]
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x0001: Input value too low. 0x8002: Input value too high.
Output	BUS	——	The Output bus contains information for each zone being evaluated.
Updated	BOOL	T/F	New information is available from the block. T: New data is available. F: New data is not available.
Areas	(Array[5]U32)	0-4294967295	The approximate cross-sectional area of objects within a zone. [mm^2]
Total_Valid_Points	U32	0-4294967295	The number of valid LiDAR points. These obtain data when landing on objects. An unusually low number may indicate issues. Invalid points include LiDAR points going into the sky or dark surfaces, which are not detected by the LiDAR. For ordered point clouds, this is the number of valid LiDAR points found within and close to the zones. LiDAR points are counted multiple times if zones overlap.
Sum_Areas	U32	0-4294967295	The cross-sectional areas of all objects within all zones added together. [mm^2]
Largest_Area	U32	0-4294967295	The cross-sectional area inside of whichever zone has the largest cross-sectional area. [mm^2]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Internal Signals

The following table describes what happens internally in the **Projected_Path_Area** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Projected_Path_Area_Err	U8	0-5	Indicates when an error occurred in the block functionality. See *Projected_Path_Area Troubleshooting* on page 199
Num_Zones	U8	0-5	The number of zones. There should always be five zones. Anything other than five means data in the arrays are invalid.
Zone_X	(Array[5]S32)	-2147483648-2147483647	X component of Cartesian location of the zone's center. Use this to help visualize where the zones are located and if they are where they are supposed to be. [mm^2]
Zone_Y	(Array[5]S32)	-2147483648-2147483647	Y component of Cartesian location of the zone's center. Use this to help visualize where the zones are located and see if they are where they are supposed to be. [mm^2]
Zone_Z	(Array[5]S32)	-2147483648-2147483647	Z component of Cartesian location of the zone's center. Use this to help visualize where the zones are located and if they are where they are supposed to be. [mm^2]
Zone_Orientation	(Array[5]S16)	-18000-18000	Rotation of the zone around the LiDAR scanner's z-axis. If the LiDAR is already rotated when mounted, then this value begins at the rotated number and not zero. Each of the five zones should have a different value because of the curvature. [0.01 degree]
Zone_Length	(Array[5]U16)	0-65535	Length of an individual zone. Each of the five zones should have the same length. [mm]

Projected_Path_Area Function Block

Item	Type	Range	Description [Unit]
Zone_Width	(Array[5]U16)	1-20000	Width of an individual zone. Each of the five zones should have the same width. [mm]
Processing_Time	U32	0-4294967295	The amount of time taken for the function block to receive data, process it, and produce a new point cloud. High processing time increases the latency for downstream function blocks, and machines react slower as the processing time increases. [µs]

Projected_Path_Area Troubleshooting

The following table describes errors that could occur in the **Projected_Path_Area** function block and ways to fix them.

View the **Projected_Path_Area_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

Projected_Path_Area_Error Descriptions and Fixes

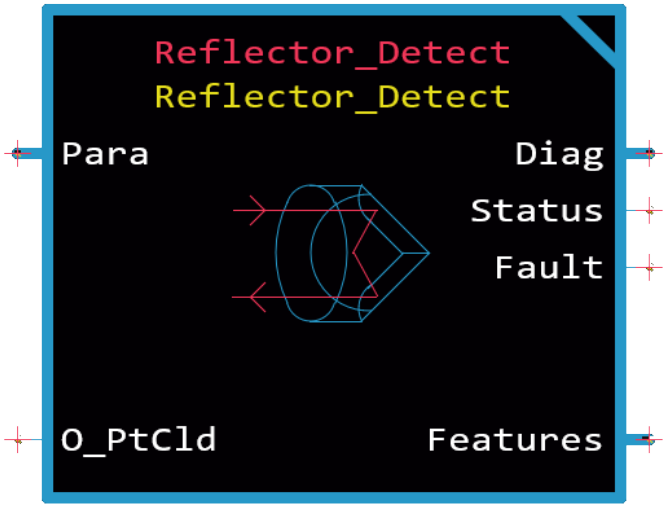
Number	Description	How to Fix
0	No errors.	Nothing needs to change.
1	Cannot create background thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
2	Not enough memory available to create thread.	This may be caused by too many Autonomous Control Library blocks. Use less than 100 ACL blocks. Turn the controller off and on, or use less code in the application.
3	Thread timeout.	There may be too much code creating a longer processing time. Reduce the LiDAR resolution or delete other processing blocks. See *Reduce Processing Time* on page 174. Turn the XM100 off, wait a bit, and restart it.
4	Point cloud is unordered. This means the point cloud data entering the block is unordered instead of ordered. Ordered LiDAR points include X, Y, and Z coordinates, whereas unordered LiDAR points have no coordinate data.	Change the point cloud data to ordered by reviewing previous code that caused it to become unordered. If unordered data is needed, use another block that does not need ordered point cloud data, such as **Projected_Path**.
5	Point cloud has invalid size. This means that the point cloud dimension is so small it may only have one row or column.	Expand the parameters in previous blocks to have a larger point cloud. Look especially in filtering blocks or the LiDAR code.

Other Errors and Fixes

Error Description	How to Fix
Zones do not pick up objects.	Check the parameters for the zones and adjust the numbers. Review the coordinates of the LiDAR. Test the edges of the zones by moving an object into and out of the zones.
The **Areas** output seems too small for the situation.	If an object has a reflector close to or on it, then this block may report a smaller cross-sectional area than it should. Move or adjust the reflector. Try different LiDARs or adjust the LiDAR settings. Or, account for the reduced **Areas** values in other code.

Reflector_Detect Function Block

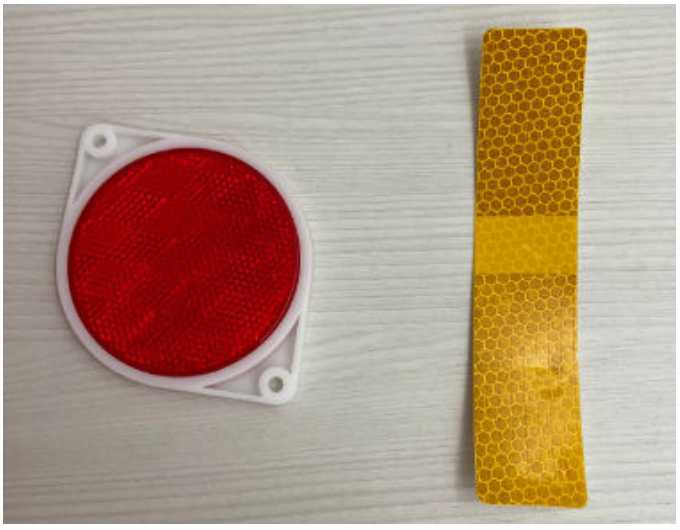
The **Reflector_Detect** function block finds reflective objects, which allows a machine to drive toward or away from the objects or use the objects as a localization base.



This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about LiDARs, the Ouster LiDAR hardware, and **Ouster_LiDAR** function block.

The **Reflector_Detect** function block works with either 2D or 3D LiDAR sensors. It searches an incoming LiDAR scan for features that meet the criteria of a reflector and outputs the center locations and properties of those reflectors. For best results, use **Reflector_Detect** to identify rectangular or circular shapes rather than other shapes.

The image below depicts reflective surfaces **Reflector_Detect** identifies, such as retroreflector lights and tape. Reflectors must be big enough for LiDAR points to land on them.

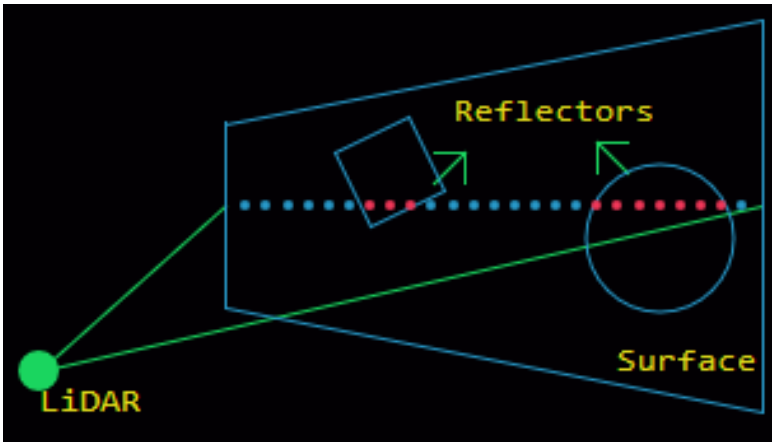


Watch out for unintentional reflective objects in the LiDAR hardware's environment that could confuse the application, such as reflective clothing from someone walking by, glare off metal or the floor, or shapes that are not circular or rectangular.

Paired with other function blocks in PLUS+1® GUIDE, **Reflector_Detect** enables a machine to navigate in an environment where reflectors are present as navigation cues. After reflective objects are detected, program a machine to drive toward or away from them, or use the reflectors as a base for localization. See *Local Coordinate System* on page 17.

Reflector_Detect Function Block

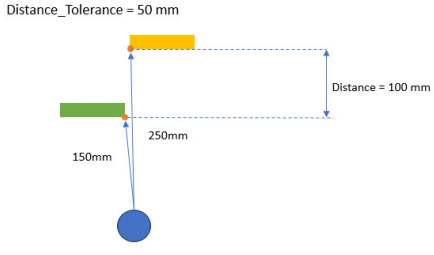
LiDAR points from a point cloud are filtered based on their locations and reflective property (intensity). Then, LiDAR points are grouped to represent individual reflectors. Locations of reflectors are defined to be at the centers of these detected point groups.



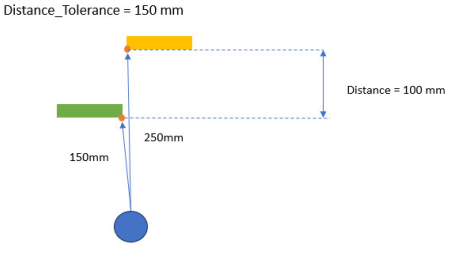
The image shows the LiDAR hardware (green dot) scanning a surface and projecting a row of LiDAR points (blue dots) in a ring row. Some of these LiDAR points detect reflectors (red dots) based on criteria in **Reflector_Detect**.

Reflectors are identified from the intensity of a reflected ray of light. The **Min_Num_Points** determines if the reflective object counts as a reflector based on the number of LiDAR points that fall on it. For example, if the **Min_Num_Points** value entered is 5 and only 3 LiDAR points land on the object, then **Reflector_Detect** does not consider the object a reflector. The **Distance_Tolerance** separates the reflectors from each other on the (X, Y) plane.

When the **Distance_Tolerance** is smaller than the difference between the distance of two LiDAR points, **Reflector_Detect** distinguishes them as two different reflectors. When the **Distance_Tolerance** is larger, the two reflectors are merged into one, so the **Num_Features** detected is one.



Distance > Distance_Tolerance, two points are not considered to belong to the same reflector. Output Num_Features = 2



Distance < Distance_Tolerance, two points are considered to belong to the same reflector. Two reflectors are merged. Output Num_Features = 1

The image shows how **Num_Features** changes depending on the distance between the LiDAR and the reflective shapes it detects. **Distance_Tolerance** accounts for the difference in distance between how close or far away LiDAR points are to the hardware, not necessarily how far away reflectors are from each other or if they are in the same point cloud ring row.

Z_Tolerance involves reflectors on the z-axis, which is ground to sky in relation to the LiDAR. Specifically, **Z_Tolerance** is the distance between the centers of vertically stacked reflectors to determine if they are combined into one reflector or kept as separate reflectors, which also affects the **Num_Features** count.

Reflector_Detect Function Block

The image shows a LiDAR and three reflectors. The **Z_Tolerance** parameter is set to 55 mm. The distance between the centers of Reflector_1 and Reflector_2 is 85 mm, and 40 mm between Reflector_2 and Reflector_3. **Reflector_Detect** considers Reflector_1 and Reflector_2 as separate reflectors because they are further apart than 55 mm, but Reflector_2 and Reflector_3 are considered as one reflector.

These hard-coded criteria determine how reflectors are identified:

- A ring row from the LiDAR scan requires at least two LiDAR points.
- At least 90% of points in a detected reflector must have intensity ratings greater than the **Intensity_Threshold** value.
- At least 90% of points in a detected reflector must be within the **Distance_Tolerance** value from adjacent points.
- The maximum percentage of points allowed is 10% intensity. Avoid noise and blooming.
- If a point has distance beyond the distance tolerance from the previous point, it's invalid.
- If the center of the reflector is too far from the next ring, it reads as two reflectors.

Additionally, for best results, keep a radial distance of 1.5 meters between the reflector and LiDAR to avoid a blooming effect. To use **Reflector_Detect** as a localization source without any tracking algorithm, keep the relative motion between the LiDAR and reflector to less than 0.5 m/s.

Input data types must exactly match the indicated type to successfully compile.

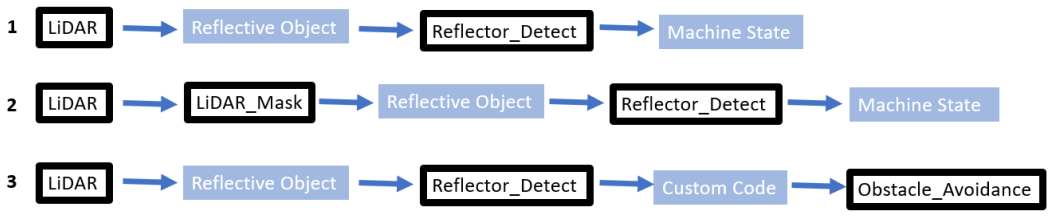
The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* on page 34 for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Application Information

Common function blocks used with the **Reflector_Detect** function block are **Data_Lockers**, **LiDAR_Mask**, and **Obstacle_Avoidance**. **Reflector_Detect** also requires LiDAR hardware with accompanying code.

Reflector_Detect uses LiDAR hardware to spot reflective surfaces, known as retroreflectors. Some basic **Reflector_Detect** combinations include:



Reflector_Detect Function Block

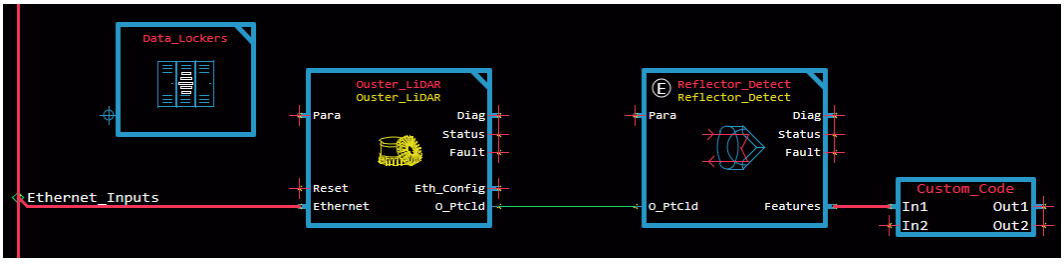
1. Scenario one shows a piece of LiDAR hardware and the accompanying code, such as Ouster LiDAR hardware and the **Ouster_LiDAR** function block. LiDARs send out laser points which detect reflective surfaces. **Reflector_Detect** determines where these reflective objects are in relation to the LiDAR hardware, and the algorithm determines the number of objects. Program a machine to react to the detected reflectors.
2. Scenario two includes the **LiDAR_Mask** function block after the LiDAR code, followed by the hardware detecting a reflective object. **LiDAR_Mask** omits certain data from the LiDAR's point cloud, saving processing time. Then, **Reflector_Detect** determines the validity of the detected objects and how many reflective features exist. Program a machine to react to the detected reflectors.
3. Scenario three includes custom code after **Reflector_Detect**, which tell the block how to process the data for the machine to avoid objects in **Obstacle_Avoidance**.

Additionally, place the **Reflector_Detect** function block:

- After LiDAR data is collected, such as after the **Ouster_LiDAR** function block.
- With one **Data_Lockers** block, version 1.11 or later, which can be on any page in the application.

Example

The example shows the **Reflector_Detect** function block used as if a machine needs to find a reflective object to drive toward.



1. Set up the LiDAR hardware and accompanying code. This example includes the **Ouster_LiDAR** function block and Ouster LiDAR hardware. See the *Plus+1 Compliant Ouster Block User Manual* for more information. The LiDAR hardware needs to see where the expected reflectors would be detected and the code set up to include point cloud information from that area. If the LiDAR code is programmed so the range the LiDAR sees is too small, then the code for the LiDAR must be changed in order for **Reflector_Detect** to work.
2. Add the **Reflector_Detect** function block. Additionally, add a **Data_Lockers** block if it does not already exist in the application. It can go on any page.
3. Outside of PLUS+1® GUIDE, determine physically in the environment where reflectors should be detected. Reflectors are detected with respect to the LiDAR hardware as the origin. See *Sensor Coordinate System* on page 19. Point the LiDAR laser toward the reflective objects to begin tuning.

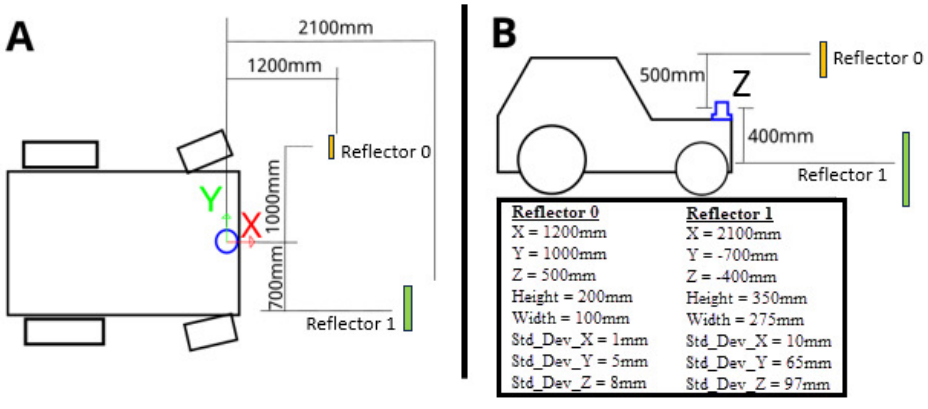


Image A shows a top down view of a machine with a LiDAR on the front. Image B shows the same machine from a side view. The LiDAR detects two reflectors, depicted as rectangles.

Reflector_Detect Function Block

4. Tune the **Intensity_Threshold** parameter in **Reflector_Detect**. LiDAR points hit a reflective surface and reflect back with a certain intensity depending on the brightness of the reflector. Set the **Intensity_Threshold** percentage to pick up the number of desired reflective surfaces. Low intensity detects more reflectors but could also detect reflectors that do not exist. High intensity makes it harder for the LiDAR to detect reflectors but could be more accurate.

   Each type of LiDAR hardware detects reflector brightness differently. Consult the intensity scale in the LiDAR hardware documentation when tuning **Reflector_Detect** to determine an appropriate intensity threshold for the application.

5. Tune the **Min_Num_Points** parameter. This number depends on the LiDAR hardware resolution. Higher values mean more LiDAR points are required to detect a reflector, so the reflectors may need to be large in size. Setting a big number reduces the chance a LiDAR detects something that is not there. A very low number detects small reflectors but could also detect reflectors that do not exist.
6. Tune the **Distance_Tolerance** parameter. This is the difference in distance between two reflective LiDAR points on the (X, Y) plane to determine if there is one or more objects. The distance is not between two reflectors but measured as points close or far from the LiDAR hardware. If this number is too small, it may separate items that should be one item. A large number may combine reflectors together that should be separate. Tune to pick up the correct number of features.
7. Tune the **Z_Tolerance** parameter. This is the distance between the centers of vertically stacked reflectors on the z-axis to determine if reflectors should be combined into one or separated. Measure the distance physically between the reflectors in the environment to decide which numbers to enter, depending on if the reflectors need to be separated or combined. Here, the distance between reflectors is 900 mm on the z-axis, so entering a number smaller than 900 mm allows the reflectors to be two.
8. Optionally, visually see the data from **Reflector_Detect** on the *Pre-Made Service Tool Screens* on page 25, or view each signal individually. In this example, the LiDAR picks up two reflectors and shows the data about them as feature outputs of the function block.
   a) The **Updated** signal pulses True to indicate new data in the arrays and processing completed. This could mean zero or many reflective features were detected.
   b) **Num_Features** tells the number of reflective objects detected and puts the coordinates into the arrays associated with the number. This example shows two **Num_Features** detected. The first number in the all the arrays in **Reflector_Detect** relate to the first reflector detected closest to the LiDAR hardware, labeled as Reflector 0 in the image because the range starts at 0. The second number in all the arrays relates to the second reflector, labeled as Reflector 1 in the image. Reflectors in arrays are sorted based on their distances closest to farthest from the LiDAR hardware.
   c) **X**, **Y**, and **Z** show the center position of each reflector. Coordinates follow the right-hand rule with respect to the LiDAR hardware as the origin. See *Sensor Coordinate System* on page 19. The example shows the **X** array as (1200, 2100) because the two reflectors are that far in front of the LiDAR in a positive x-axis direction. **Y** shows (1000, -700) because Reflector 0 is 1000 mm away from the LiDAR center in the y-axis positive direction, and Reflector 1 is 700 mm away from the LiDAR center in the y-axis negative direction. **Z** refers how far the reflectors are up and down from the LiDAR, listed as (500, -400) in the array. Reflector 0 is 500 mm in the positive z-axis direction from the LiDAR center, and Reflector 1 is 400 mm below the LiDAR center in a negative z-axis direction.
   d) **Std_Dev_X**, **Std_Dev_Y**, and **Std_Dev_Z** guess how far off the location of points are from the center of reflectors, depicting a level of confidence. High numbers indicate higher location uncertainty, and **Reflector_Detect** should be tuned more. In this example, Reflector 1 shows a high standard deviation in Y and Z fields, with **Z** possibly in the range of -300 to -500 mm on the z-axis because it could be deviated 97 mm in both directions from its center. Reflector 0 shows more confidence in the detected location in all fields. Here, **Std_Dev_Z** of 8 means the reflectors could be between 492 to 508 mm on the z-axis. These signals take into account the number of LiDAR points that land on the reflectors to find the center, but this is affected by distance and noise around the LiDAR. This example assumes the reflectors are both flat and unbent.

   If the standard deviation numbers are extremely high in the thousands, then the reflectors may be on the edge of the LiDAR's field-of-view. Either physically move the LiDAR or change the field-of-view in the LiDAR code.

Reflector_Detect Function Block

e) **Num_Points** tells how many LiDAR points were detected on the reflective feature. More points mean more confidence that it is a valid feature, and it might correlate with a larger or closer object to the LiDAR. Less points mean less confidence, a far away object, or small object.

f) **Width** of a reflector shows the total distance of the reflector shape and not just the center. This is based on where the LiDAR points fall on the largest LiDAR ring row detected, so **Width** might not exactly match the reflector's actual width.

g) **Height** of a reflector shows the total height of the reflector, calculated by measuring the distance between the LiDAR's highest and lowest point cloud rings detected on the reflector.

h) **Seq_ID** shows which LiDAR frame the data came from. This could be used to correct for machine motion during processing.

9. After tuning **Reflector_Detect**, create custom code for a machine to drive toward a reflector. Filter based on where a reflective object is expected to be located or its size. There are a lot of things in this block that determine whether an object is valid for the given use case.

Inputs

Inputs to the **Reflector_Detect** function block are described.

Item	Type	Range	Description [Unit]
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

Parameters to the **Reflector_Detect** function block are described.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Intensity_Threshold	U16	1-10000	Minimum intensity of LiDAR points required to consider it reflective. Default: 5000 [0.01%]
Min_Num_Points	U16	1-500	The minimum number of LiDAR points required to land on an object to determine if it is a reflector. This is the total number of points across all ring rows. Default: 10
Distance_Tolerance	U16	1-1000	The difference in radial distance between two LiDAR points to determine if there are one or two reflectors. The difference in distance is based on how close or far away LiDAR points are to the LiDAR hardware. Default: 50 [mm]
Z_Tolerance	U16	1-1000	Distance between the centers of vertically stacked reflectors to determine if the reflectors should be combined into one reflector or separated into two reflectors. Values higher than this number are separated and values lower are combined into one reflector. Default: 50 [mm]

Reflector_Detect Function Block

Outputs

Outputs of the **Reflector_Detect** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Processing_Time	U32	0-4294967295	Time taken to process input point cloud data. [µs]
Reflector_Detect_Err	U8	0-5	Indicates errors occurred in the function block operation. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: Point cloud is unordered. 5: Detected reflector is at the edge of the LiDAR scan.
Status	U16	——	The status of the function block. Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Features	BUS	——	This BUS contains information about the reflectors identified in the LiDAR scan. This includes reflector width, X, Y, and Z coordinates, and standard deviation in Cartesian coordinates.
Updated	BOOL	T/F	Updated signal is set to True if new point cloud data has been processed. T: New data is available. F: New data is not available.
Num_Features	U16	0-100	The number of reflectors detected. Array numbers go in order of reflectors closest to farthest from the LiDAR hardware.
X	(Array[100]S32)	-2147483648-2147483647	Point on the x-axis showing the center position of each detected reflector. X uses the Cartesian coordinate system with respect to the LiDAR scanner as origin. [mm]
Y	(Array[100]S32)	-2147483648-2147483647	Point on the y-axis showing the center position of each detected reflector. Y uses the Cartesian coordinate system with respect to the LiDAR scanner as origin. [mm]
Z	(Array[100]S32)	-2147483648-2147483647	Point on the z-axis showing the center position of each detected reflector. Z uses the Cartesian coordinate system with respect to the LiDAR scanner as origin. [mm]
Std_Dev_X	(Array[100]U16)	0-65535	Standard deviation of the reflector center along the x-axis in relation to the LiDAR. Higher numbers indicate the reflector could be located further away from what the X coordinate reading says. [mm]
Std_Dev_Y	(Array[100]U16)	0-65535	Standard deviation of the reflector center along the y-axis in relation to the LiDAR. Higher numbers indicate the reflector could be located further away from what the Y coordinate reading says. [mm]
Std_Dev_Z	(Array[100]U16)	0-65535	Standard deviation of the reflector center along the z-axis in relation to the LiDAR. Higher numbers indicate the reflector could be located further away from what the Z coordinate reading says. [mm]
Num_Points	(Array[100]U32)	0-4294967295	The number of LiDAR points detected on the reflectors. More points mean the reflector is large, close to the LiDAR, or more confidence that the feature is a reflector.

Reflector_Detect Function Block

Item	Type	Range	Description [Unit]
Width	(Array[100]U 16)	0-65535	The total reflector width, calculated by using the LiDAR points in the widest detected point cloud ring. [mm]
Height	(Array[100]U 16)	0-65535	The total reflector height, calculated by using the LiDAR points between the highest and lowest point cloud rings detected on the reflector. [mm]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Reflector_Detect Troubleshooting

The following table describes errors that could occur in the **Reflector_Detect** function block and ways to fix them. View the **Reflector_Detect_Err** signal on the Service Tool screen to see if any error numbers appear. In PLUS+1® GUIDE, this signal is on the **Checkpoints** page in the **Internal Signals** column.

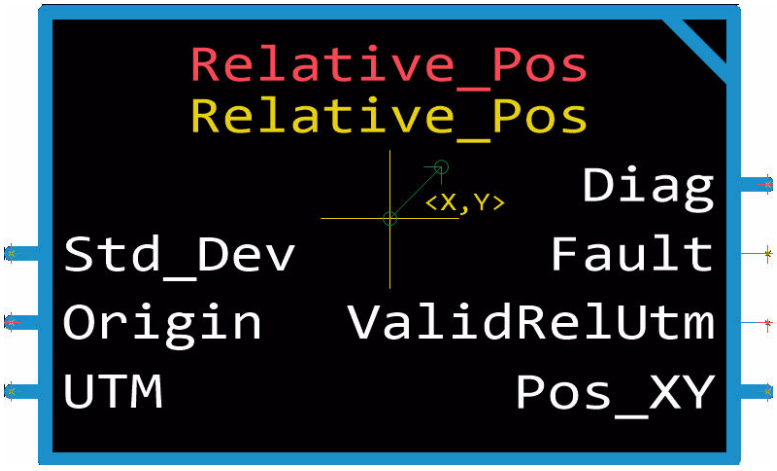
Processing_Time should be less than one microsecond.

Reflector_Detect_Err Descriptions and Fixes

Number	Description	How to Fix
0	There is no error.	Nothing needs to change.
1	Cannot create background thread.	Turn the controller off and on, or use less code in the application.
2	No memory available. This varies with the type of hardware and may happen with non-XM100 hardware.	Turn the controller off and on, or use less code in the application.
3	Thread timeout. The controller could be overloaded, which would affect many blocks.	Reduce the LiDAR resolution or delete other processing blocks.
4	Point cloud is unordered. This means the data changed from ordered to unordered so the structure of the point cloud data is lost.	Check the code to see if the data was filtered or modified to become unordered, and switch to the ordered output option.
5	The detected reflector is at the edge of the LiDAR scan. This means that when the LiDAR scan occurred, a reflector appeared at the edge of the LiDAR's field-of-view on the x-axis, y-axis, or z-axis. The LiDAR cannot determine information about the reflector because the majority of the reflector is outside what it sees.	Physically move the LiDAR hardware to see the reflector better or adjust the code to expand the LiDAR's field-of-view.

Relative_Pos Function Block

The **Relative_Pos** function block calculates the relative position between the machine's current position and the machine's origin position.



Input data types must exactly match the indicated type to successfully compile.

The Checkpoints page includes advanced checkpoints for each input, output and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name.

The Updated variable in the Origin bus and in the UTM bus must both go True at least once before the outputs to this function block are updated.

Inputs

Inputs to the **Relative_Pos** function block are described.

Item	Type	Range	Description [Unit]
Std_Dev	BUS	——	Contains UTM data for the origin.
X_Std_Dev	U32	1-4294967295	The standard deviation of X. [mm]
Y_Std_Dev	U32	1-4294967295	The standard deviation of Y. [mm]
Origin	BUS	——	The UTM data for the machine's origin point.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value.
Zone	U8	1-60	The zone that the UtmX and UtmY values are in.
Band	U8	67-72, 74-78, 80-88	The band that the UtmX and UtmY values are in. ASCII values represent the letter of the band.
Updated	BOOL	T/F	True when there is new position data ready. Update the outputs. T: Calculate relative position with the new data. Update outputs. F: Do not calculate new position. Do not update outputs.
UTM	BUS	——	UTM data for the current position.

Relative_Pos Function Block

Item	Type	Range	Description [Unit]
UtmX	U32	0-10^9	The UTM Easting (X) value of the machine's current position. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the machine's current position. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value.
Zone	U8	1-60	The zone that the UtmX and UtmY values are in.
Band	U8	67-72, 74-78, 80-88	The band that the UtmX and UtmY values are in.
Updated	BOOL	T/F	True when there is new data ready and the outputs are updated. T: Calculate the relative position of the machine. Update the outputs. F: Do not calculate the relative position. Do not update outputs.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

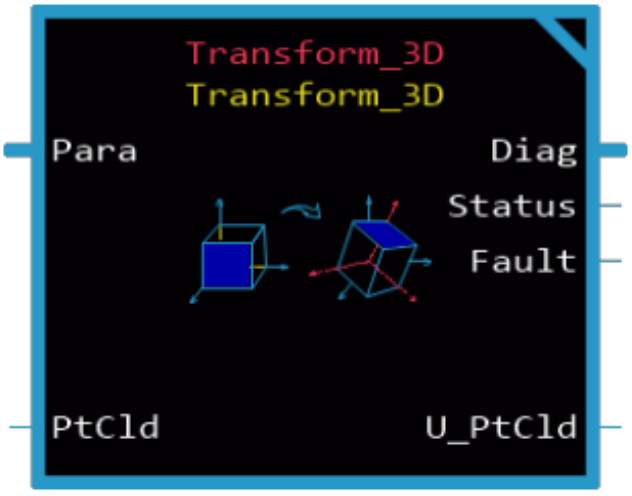
Outputs

Outputs of the **Relative_Pos** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high. 0x8010: Input value is out of range.
ValidRelUtm	BOOL	T/F	When True, the zones from the Origin and UTM buses match. Both update flags have been True at least once. There are no faults. When False, **UtmX** and **UtmY** are both set to 0. T: The zones from the Origin and UTM match. F: The zones do not match or the origin is invalid and outputs are 0.
Pos_XY	BUS	——	Data about the relative position calculated in the function block and its standard deviation.
X	S32	-2147483648-2147483647	The relative X distance between the origin and the current position. [mm]
Y	S32	-2147483648-2147483647	The relative Y distance between the origin and the current position. [mm]
Updated	BOOL	T/F	True when new data is available from the conversion. T: New data is available. F: New data is not available.
X_Std_Dev	U32	1-4294967295	The standard deviation of X. [mm]
Y_Std_Dev	U32	1-4294967295	The standard deviation of Y. [mm]

Transform_3D Function Block

The **Transform_3D** function block transforms a 2D or 3D point cloud into a format that is compatible with the coordinate system of a machine as defined by the parameters.



This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about the Ouster LiDAR scanner and block.

Each point in a point cloud contains coordinate information. This is referred to as an ordered point cloud. **Transform_3D** takes ordered point cloud data and removes the information, condensing the point cloud to save processing time. This is referred to as an unordered point cloud. After the point cloud becomes unordered, it can never become ordered again.

The function block has the following limitations:

- The function block is unable to scale the point cloud during a conversion. It performs only rigid transformations.
- During the transformation, the distance between the transformed points compared to the origin of the new coordinate system might be different than the maximum/minimum range defined in the sensor attributes. In this case, the maximum/minimum range defined in the sensor attributes is overwritten so that all points of the cloud fit into this range.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.

Inputs

The following table describes input signals in the **Transform_3D** function block.

Item	Type	Range	Description [Unit]
PtCld	S8	-1 - 99	The data locker ID of ordered or unordered point cloud data.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Transform_3D Function Block

Parameters

The following table describes parameters in the **Transform_3D** function block.

Item	Type	Range	Description [Unit]
X	S32	-2147483648 - 2147483647	Specifies the translation along the X axis. [mm] Default: 0
Y	S32	-2147483648 - 2147483647	Specifies the translation along the Y axis. [mm] Default: 0
Z	S32	-2147483648 - 2147483647	Specifies the translation along the Z axis. [mm] Default: 0
Roll	S16	-18000 - 18000	Sets the rotation around the X axis. [0.01 deg] Default: 0
Pitch	S16	-18000 - 18000	Sets the rotation around the Y axis. [0.01 deg] Default: 0
Yaw	S16	-18000 - 18000	Sets the rotation around the Z axis. [0.01 deg] Default: 0

Outputs

The following table describes outputs in the **Transform_3D** function block.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Status	U16	——	Reports the status of the function block. It is a bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range. 0x8100: Invalid ECU.
Fault	U16	——	Reports the faults of the function block. It is a bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
U_PtCld	S8	-1 - 99	The data locker ID of unordered point cloud data.

Internal Signals

The following table describes internal signal in the **Checkpoints** page in the **Transform_3D** function block.

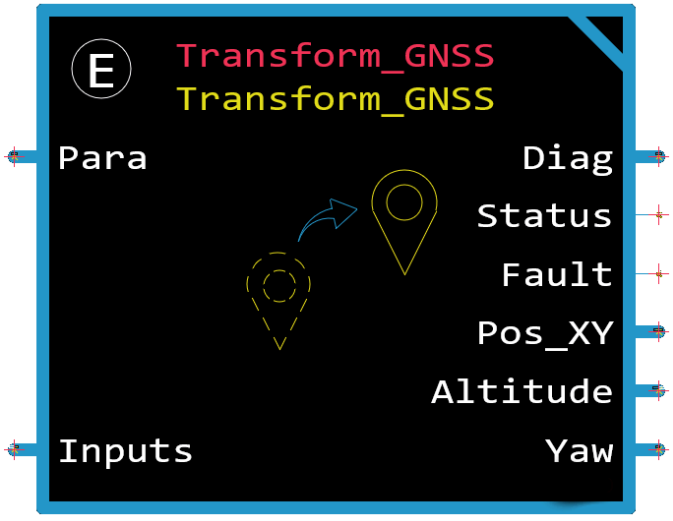
Item	Type	Range	Description [Unit]
Rot_Matrix	(ARRAY[9]S32)	-2147483648 - 2147483647	Reports elements of the rotation matrix. Element 0 of the array corresponds to the value at the first column of the first row. Element 1 corresponds to the value at the second column of the first row. It continues in that fashion. All elements are multiplied by 1e6.
Processing_Time	U32	0 - 4294967295	Time taken to process input point cloud data. [µs]

Transform_3D Function Block

Item	Type	Range	Description [Unit]
Transform_Err	U8	0 - 4	Indicates that an error occurred in the block functionality. 0: No Error 1: Unable to create thread 2: Not enough memory available to create thread 3: Thread timeout 4: The requested translation is greater than the maximum range of the sensor
Updated	BOOL	T/F	Indicates that the function block has stored a new point cloud in the data locker. T: New data is available F: No new data is available
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Transform_GNSS Function Block

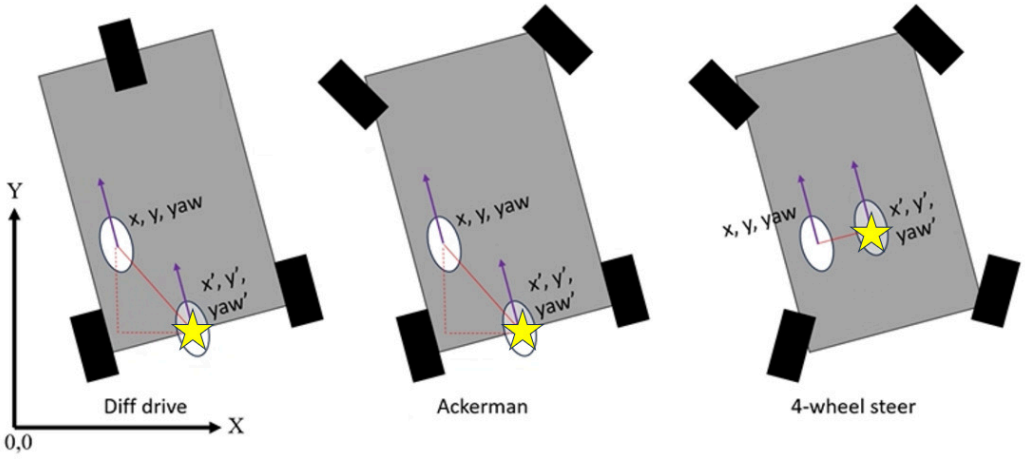
The **Transform_GNSS** function block takes coordinate data from one point and transforms it to show coordinate information from a different point. Usually, this involves coordinate data for a GNSS antenna recalculated to monitor the machine's origin.



Many machines run autonomously due to sensors calculating where the machine is in the world or in relation to its surroundings. GNSS antenna sensors measure the antenna's location with satellites, thereby getting global X and Y coordinates for the antenna. See *Global or World Coordinate System* on page 15. Some hardware sensors measure altitude, which is the height of the sensor from sea level. Other hardware sensors include a compass to measure yaw, which is rotation along the z-axis. All this data measures the sensor's location, and usually an autonomous machine needs the coordinate data to measure another section of the machine frame, such as the machine's origin on the back axle. **Transform_GNSS** allows this recalculation of the coordinates to happen, essentially transforming the data from one point to a different point.

Common scenarios include transforming the coordinate data from the GNSS sensor to the machine's origin, or transforming the data to collect coordinates on the edges of the machine. For example, the coordinates of the four corners of a machine could be calculated to determine if a section of the machine crosses a boundary.

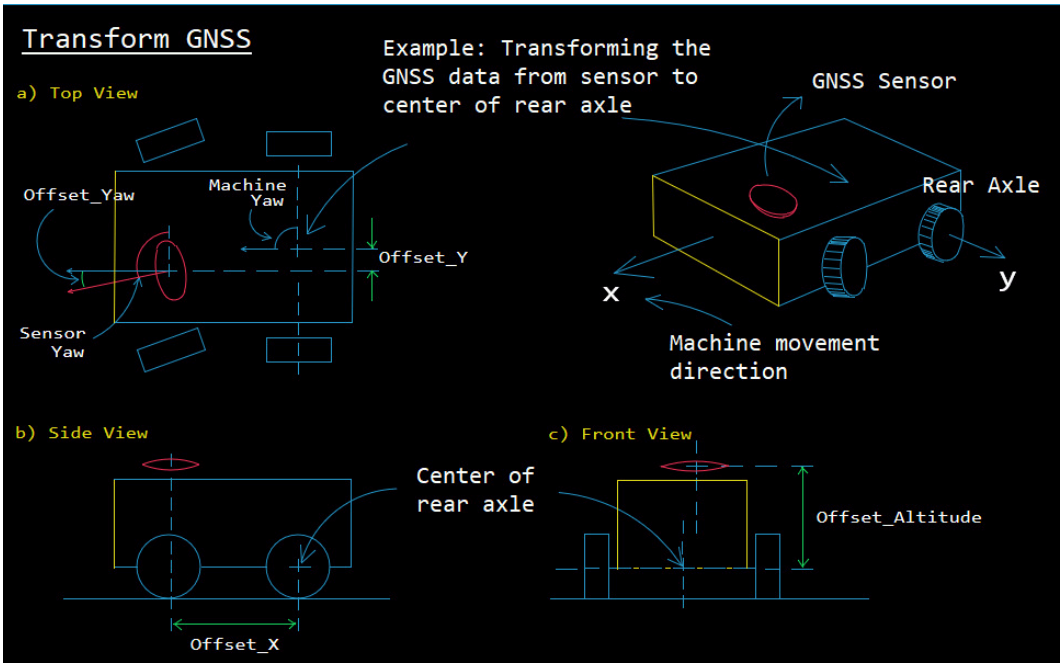
When recalculating the coordinates from the GNSS antenna to the machine's origin, determine the machine's origin based on the type of machine.



The image above shows the machine's origin (yellow star) in different places based on the type of machine. The GNSS coordinates move from the antenna (purple circle) to the machine's origin. The antenna includes a yaw sensor.

Transform_GNSS Function Block

Measure from the machine's origin to the GNSS antenna, and put these distances into **Offset_X** and **Offset_Y** parameters in **Transform_GNSS**. Antennas in front or to the left of the machine's origin have positive values, and behind or to the right have negative values.

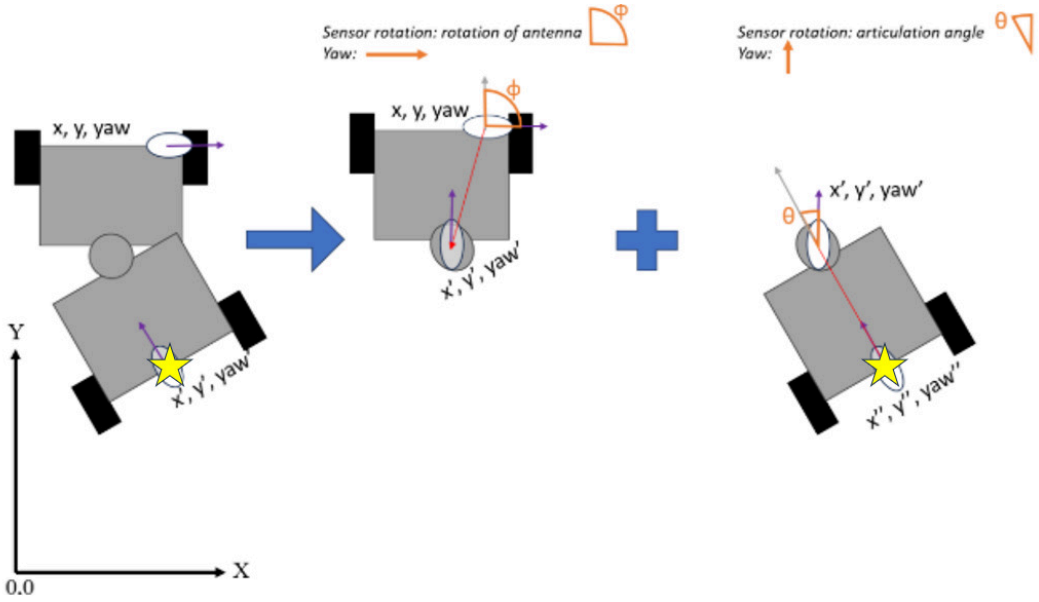


The image above shows three viewpoints of a machine with a dual GNSS and yaw antenna sensor on top. The sensor gets coordinates for the sensor's origin, and that information is transformed to show the machine's origin coordinates on the center rear axle. The distances between the sensor's and machine's origins are entered as offsets in **Transform_GNSS** parameters.

When mounting either a dual GNSS antenna or a yaw sensor, line up the sensor rotation with the machine's rotation so that yaw is zero. Get the sensor's yaw from its instruction manual or use the **Yaw_Estimate** function block. The GNSS antenna's information goes into the inputs of **Transform_GNSS**, which could include roll, pitch, altitude, yaw, X, Y, and standard deviations.

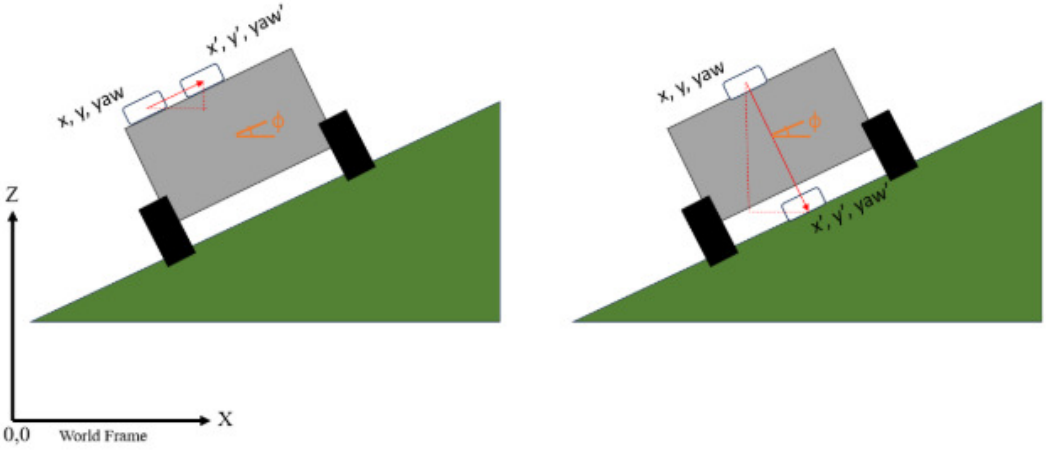
If using an articulated machine, **Transform_GNSS** calculates the pivot point where two machine sections connect, and then a second **Transform_GNSS** calculates from the pivot point to the machine's origin. Ideally, use zero for the sensor rotation **Yaw_Offset** in the first block, and use the articulation angle as the **Yaw_Offset** in the second block.

Transform_GNSS Function Block



The image above shows an articulated machine with a dual GNSS and yaw sensor in the top right corner. Coordinates move from the sensor to the articulated point, and then from the articulated point to the back of the machine. Use the articulation angle for a yaw offset in the second block.

If a machine will go on a slope, measure from the ground to the GNSS sensor's origin, entered as parameter **Altitude_Offset**. The machine frame also requires pitch and roll data to identify a slope, which could be gathered from an Inertial Measurement Unit (IMU) sensor. On a slope, **Transform_GNSS** automatically adjusts the coordinates to read from the ground rather than the machine's origin. The **Altitude** output shows the distance above sea level either from the ground for a slope or the machine's origin for flat surfaces. However, this feature is turned off if **Altitude_Offset** is zero.



The image above shows a machine on a slope. The coordinates move from the GNSS sensor (top left blue rectangle) to the center of the machine, and then to the ground, rather than the machine's origin.

Transform_GNSS does not specifically need hardware sensors. Custom code could determine where the machine is in relation to its surroundings. In that case, any mention of hardware sensor would mean the first coordinate point, before the information is transformed to a second coordinate point. Another **Transform_GNSS** function block is needed each time coordinate data moves to another location.

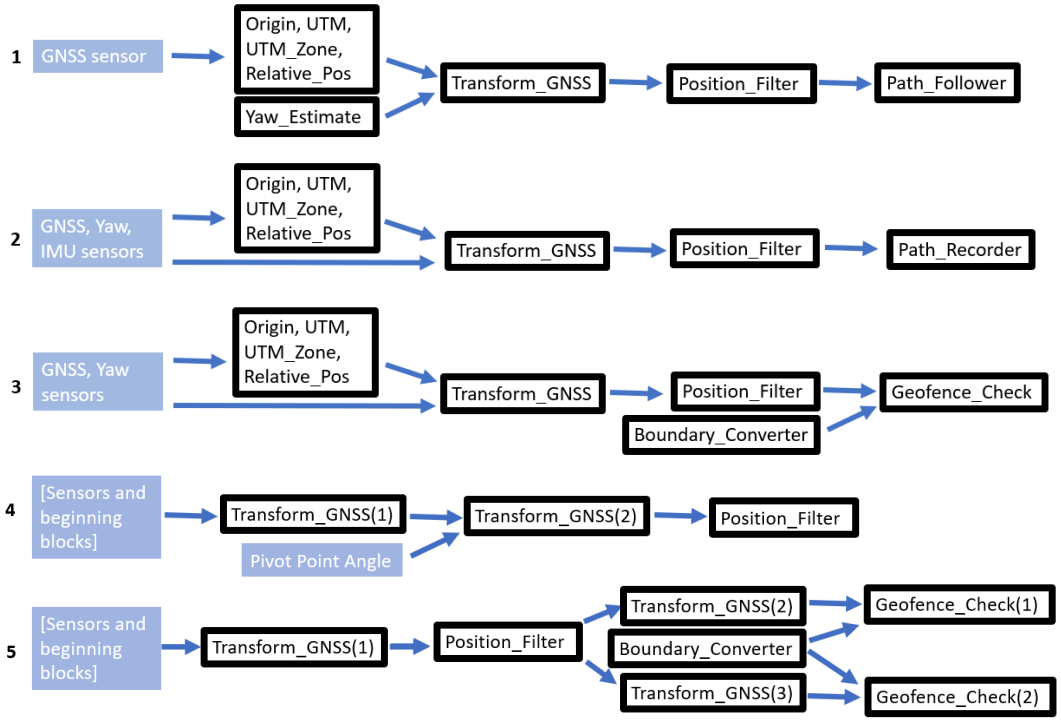
Transform_GNSS outputs show the coordinates of the transformed point, which is usually the machine's origin. Compare the input and output data to check that the coordinates are different from each other. Additionally:

Transform_GNSS Function Block

- Avoid very low standard deviations, such as less than 1 mm or 1 degree. This prevents numerical issues and small deviations occurring downstream in the code, especially in **Position_Filter**.
- Identify the inputs before using **Transform_GNSS** or **Position_Filter** in the field.
- Mount all hardware sensors aligned with the machine coordinates to avoid extra calculations.
- **Transform_GNSS** assumes the GNSS antenna and machine have a rigid body transformation with no moving parts between the antenna and machine chassis.
- A yaw sensor could be used to get the machine's origin. The **Yaw_Offset** parameter aligns the yaw sensor to the machine's frame.
- Dual GNSS antennas are not required. **Transform_GNSS** assumes that **Roll** and **Pitch** inputs are for the machine frame, and the **Yaw** input is for the GNSS antenna frame.
- **Roll** and **Pitch** inputs must be sensed by the machine frame, and are usually gathered from an Inertial Measurement Unit (IMU) sensor. Do any required rotations before using these inputs.
- Record the transformed position first if using **Transform_GNSS** with **Path_Follower_Adv** or **Path_Recorder**, rather than recording the path first.

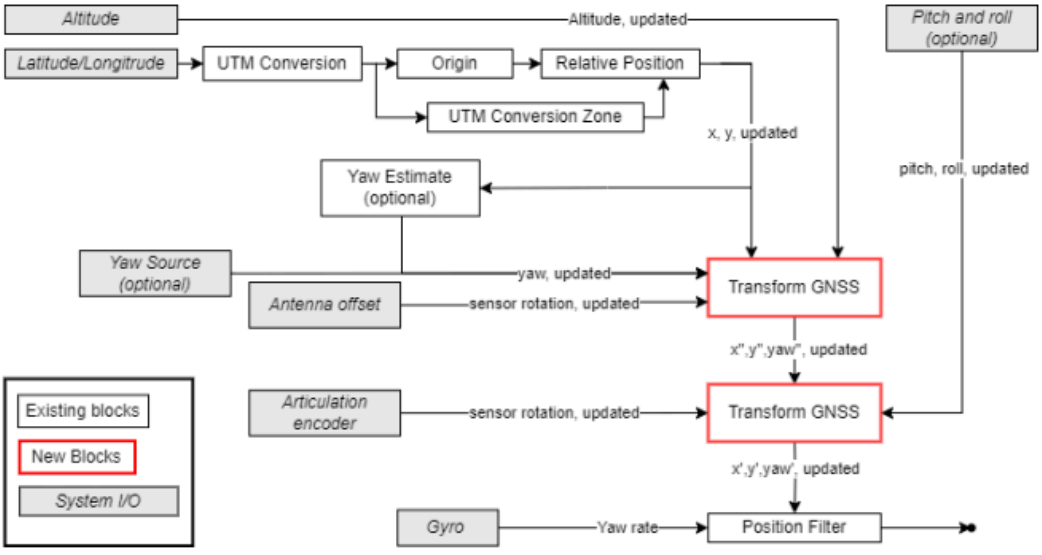
Application Information

Common function blocks that work with **Transform_GNSS** are **Position_Filter**, as well as GNSS function blocks **Origin**, **Relative_Pos**, **UTM_Conv**, and **UTM_Conv_Zone**.



Transform_GNSS Function Block

1. Scenario one shows coordinate information from a GNSS sensor transformed to a machine's origin. A GNSS hardware sensor gets global coordinate information, such as longitude and latitude numbers. Information passes through the four GNSS function blocks to give the hardware sensor's relative position, which are smaller numbers. The **Yaw_Estimate** function block guesses the hardware sensor's rotation around the z-axis. All this hardware coordinate information goes into **Transform_GNSS**, which transforms the coordinate data to monitor the machine's origin. The machine's relative coordinate information enters **Position_Filter** to get a more precise location reading. Then **Path_Follower** uses the coordinate information for the machine to follow a path.
2. Scenario two includes more hardware sensors that measure yaw and Inertial Measurement Unit (IMU). These extra sensors help compensate when a machine tilts on a hill. Because the hardware gives yaw information, the **Yaw_Estimate** function block is not needed. **Transform_GNSS** transforms the relative hardware coordinates to read another section of the machine. The relative machine coordinates go into **Position_Filter** to give more precise readings. Then, **Path_Recorder** records a path for a machine to follow, and it compensates for the tilted hardware sensor to give more accurate readings on a different machine.
3. Scenario three uses **Transform_GNSS** to determine when a section of the machine crosses a boundary in **Geofence_Check**. In this case, the coordinate information moves from the GNSS and yaw sensors to another part of the machine, such as the machine's origin. Boundary information flows from **Boundary_Converter** into **Geofence_Check**. Further code tells the machine how to react after the machine's origin crosses the boundary.
4. Scenario four uses two **Transform_GNSS** to determine an articulated machine's coordinates. The first transformed point occurs at the pivot point angle, and then those coordinates with the angle are transformed to the machine's origin with the second **Transform_GNSS**.
5. Scenario five adds more **Transform_GNSS** after **Position_Filter** determined the machine's origin. Two more **Transform_GNSS** locate two other coordinate points on or around the machine, such as the front and back of the machine. In parallel, **Boundary_Converter** gives boundary data to **Geofence_Check**, which reads if the two areas of the machine pass over the boundary.



The image above shows two **Transform_GNSS** used with an articulated machine and where the data originates.

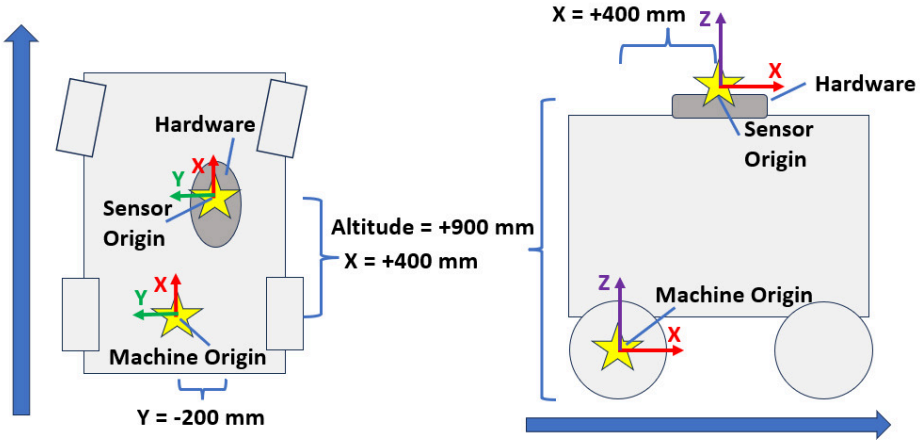
Additionally, place the **Transform_GNSS** function block:

- After position data is collected from hardware sensors. Sometimes, the machine's position can be established by locating itself to nearby items without using GNSS hardware.
- After GNSS function blocks **Origin**, **Relative_Pos**, **UTM_Conv**, and **UTM_Conv_Zone**.
- Before the **Position_Filter** function block.

Transform_GNSS Function Block

Example

The example shows the **Transform_GNSS** function block used to move coordinates from a GNSS antenna to the machine's origin.

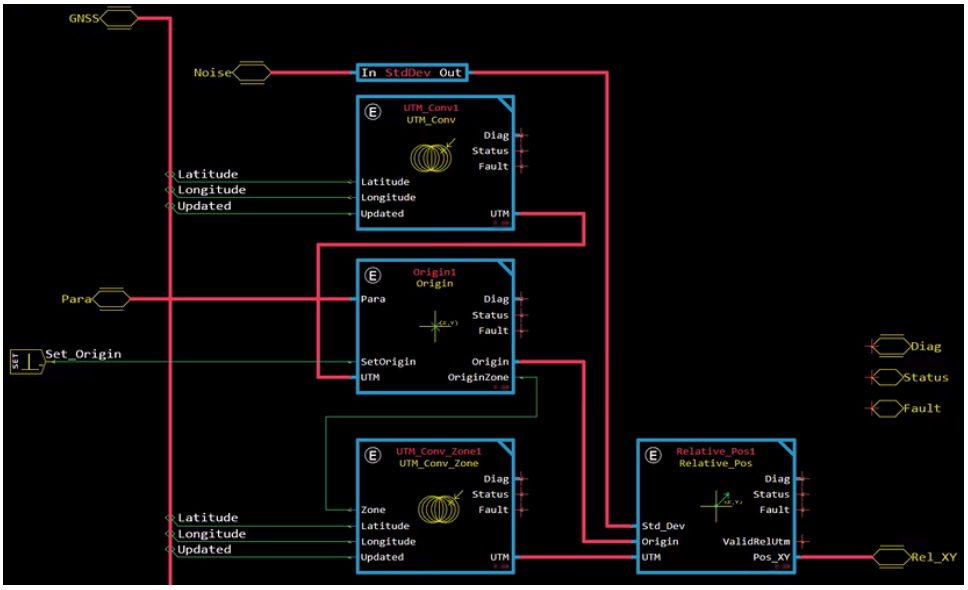


The image above shows a top down and side views of a machine with a GNSS antenna sensor on top, labeled as **Hardware**. The machine and sensor origins appear as yellow stars.

1. Determine where the machine's origin should be based on the type of machine. Here, it is an Ackermann machine, so the machine's origin is in the center back axle.
2. Mount the GNSS antenna hardware on the highest place on the machine, as close as possible to the machine's origin. The antenna should be high enough to have visibility of open sky and not under a hood.
3. Look at the antenna hardware instructions to determine the antenna's yaw. Ideally, match the antenna yaw rotation to the machine's yaw at 0 degrees. Optionally, use the **Yaw_Estimate** function block to get the machine's offset from the antenna.
4. Make sure the Inertial Measurement Unit (IMU) is aligned to the machine. To do that, rotate it. Put the machine on a flat surface, where X and Y are 0,0. Anything seen in the IMU is a bias. Put in custom code to average the values. Then, remove that average from the signal.

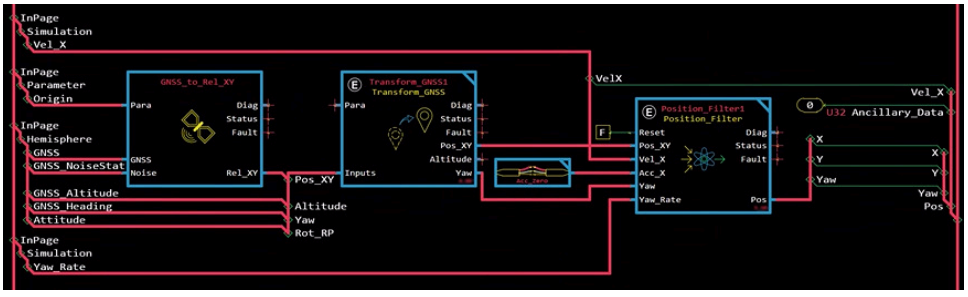
Transform_GNSS Function Block

5. Set up the **UTM_Conv**, **Origin**, **UTM_Conv_Zone**, and **Relative_Pos** function blocks, which establish the machine's location. Here, the noise from the hardware GNSS antenna sensor goes into **Relative_Pos**.



The image above shows a common set-up to establish the machine's location.

6. Connect **Relative_Pos** to **Transform_GNSS**. Here, the four beginning blocks are combined in the **GNSS_to_Rel_XY** block with the output **Rel_XY** going into **Inputs**.



Hemisphere refers to the GNSS hardware, whose coordinate information goes into **UTM_Conv** and **UTM_Conv_Zone**. The noise of the GNSS hardware goes into **Relative_Pos**, and the hardware altitude goes into **Transform_GNSS**. Roll and pitch machine information goes into **Transform_GNSS**.

7. Connect the outputs of **Transform_GNSS Pos_XY** to **Position_Filter**, as well as **Yaw**. The machine's yaw rate goes into **Position_Filter**.
8. Go into **Position_Filter** to make **Acc_X** zero. Here, this code is pulled into another box outside of **Position_Filter**.
   a) Enter 0 for **AccX**.
   b) Enter 1 for **AccX_Std_Dev**.
   c) Enter F for **Updated**.

Transform_GNSS Function Block

9. Measure from the machine's origin to the hardware antenna's origin. Here, that is the back axle to the top center of the antenna. Roll and pitch are relative to the machine body and not to the antenna. Fill out the parameters in **Transform_GNSS**.
   a) From the machine's origin up to the hardware sensor is 400 mm. In the parameters, enter 400 into **Offset_X**.
   b) From the machine's origin to the hardware sensor is 200 mm to the right. In the parameters, enter -200 into **Offset_Y**.
   c) From the ground to the hardware sensor is 900 mm. In the parameters, enter 900 into **Offset_Altitude**. Altitude comes from the antenna hardware. This shows the sensor above sea level, so numbers in the output will likely be higher than the sensor to the ground.
   d) The rotation of the hardware sensor and the machine were mounted to match each other. In the parameters, enter 0 in **Offset_Yaw**.
10. Connect the **Position_Filter** coordinates to other function blocks. For example, they can go into the path blocks for the machine to follow a path.
11. Look at the **Transform_GNSS** input and output numbers side by side to see that they are different from each other, and check there are no error codes. Visually see the data on their *Pre-Made Service Tool Screens* on page 25, or view each signal individually.
12. Plot on a graph in Excel the X and Y positions to see that the machine is going in the correct location and points are in a smooth line. If the plots seem off or points are not in a good line, that means the sensors might have issues reading data. This problem happens if **Position_Filter** was not set up correctly. If the IMU is not aligned with machine, then the position of the machine is wrong and the wrong spot will be calculated.

Inputs

The following table describes inputs required for the **Transform_GNSS** function block. Usually, the input location and source information is for a hardware sensor, and the output information is for the transformed, new location on the machine.

Item	Type	Range	Description [Unit]
Pos_XY	BUS	——	A bus that contains location information, including standard deviation and updated conditions. This information usually originates from a hardware sensor that measures GNSS coordinates or the **Relative_Pos** function block.
X	S32	-2147483648-2147483647	The X position before it transforms to another X position. Usually, this is the hardware sensor's relative X position. [mm]
Y	S32	-2147483648-2147483647	The Y position before it transforms to another Y position. Usually, this is the hardware sensor's relative Y position. [mm]
X_Std_Dev	U32	1-4294967295	The standard deviation of the hardware sensor's location along the x-axis. Smaller numbers indicate more confidence in the location. [mm]
Y_Std_Dev	U32	1-4294967295	The standard deviation of the hardware sensor's location along the y-axis. Smaller numbers indicate more confidence in the location. [mm]
Updated	BOOL	T/F	Indicates whether the hardware sensor's X and Y position data is updated. T: New data is available. F: New data is not available.
Altitude	BUS	——	A bus that contains altitude information, including standard deviation and updated conditions. This information usually comes from a hardware sensor that measures altitude.
Altitude	S32	-2147483648-2147483647	The altitude of the hardware sensor's location, which is the distance from sea level up to the hardware sensor. [mm]

Transform_GNSS Function Block

Item	Type	Range	Description [Unit]
Altitude_Std_Dev	U32	1-4294967295	The standard deviation of the hardware sensor location along the z-axis, referring to the distance above sea level. Smaller numbers indicate more confidence in the location. [mm]
Updated	BOOL	T/F	Indicates whether the hardware sensor's altitude data is updated. T: New data is available. F: New data is not available.
Yaw	BUS	——	A bus that contains yaw information, including standard deviation and updated conditions. Yaw refers to the machine's rotation around the z-axis. This information usually comes from a hardware sensor or **Yaw_Estimate** function block.
Yaw	S32	-72000 - 72000	The angle of the hardware sensor's rotation around the z-axis using the ENU (East-North-Up) reference frame. [0.01 degree]
Yaw_Std_Dev	U32	1-4294967295	The standard deviation of the hardware sensor's rotation around the z-axis. Smaller numbers indicate more confidence in the machine's rotation. [0.01 degree]
Updated	BOOL	T/F	Indicates whether the hardware sensor's yaw data is updated. T: New data is available. F: New data is not available.
Rot_RP	BUS	——	A bus that contains information about a machine's rotation around the x-axis and y-axis, including updated conditions. This information usually comes from a hardware sensor.
Pitch	S16	-18000 - 18000	The angle of the machine frame's rotation around the y-axis. This data usually comes from an Inertial Measurement Unit (IMU) sensor. [0.01 degree]
Roll	S16	-18000 - 18000	The angle of the machine frame's rotation around the x-axis. This data usually comes from an Inertial Measurement Unit (IMU) sensor. [0.01 degree]
Updated	BOOL	T/F	Indicates whether the machine frame's roll and pitch data is updated. T: New data is available. F: New data is not available.

Parameters

The following table describes the parameters for the **Transform_GNSS** function block. Usually, the input location and source information is for a hardware sensor, and the output information is for the transformed, new location on the machine.

Item	Type	Range	Description [Unit]
Offset_X	S32	-2147483648-2147483647	The distance from the machine's transformed point along the x-axis to the hardware sensor which measures GNSS. Hardware sensors in front of the transformed point have positive values and in back have negative values. Default: 0 [mm]
Offset_Y	S32	-2147483648-2147483647	The distance from the machine's transformed point along the y-axis to the hardware sensor which measures GNSS. Hardware sensors to the left of the transformed point have positive values and to the right have negative values. Default: 0 [mm]

Transform_GNSS Function Block

Item	Type	Range	Description [Unit]
Offset_Altitude	S32	-2147483648-2147483647	The distance between the ground along the z-axis to the hardware sensor which measures altitude. Although altitude reads from sea level, this parameter only requires the distance from the ground to the hardware sensor. Default: 0 [mm]
Offset_Yaw	S16	-18000 - 18000	The difference between the hardware sensor's yaw angle and the machine's yaw angle. Yaw refers to rotation around the z-axis. Default: 0 [0.01 degree]

Outputs

The following table describes outputs for the **Transform_GNSS** function block. Usually, the input location and source information is for a hardware sensor, and the output information is for the transformed, new location on the machine.

Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x0001: Input value too low. 0x8002: Input value too high.
Output	BUS	——	A bus that contains information about the machine's position after applying the offsets and rotation to the input values from the hardware sensor.
Pos_XY	BUS	——	A bus that contains machine location information adjusted to a new position, including standard deviation and updated conditions.
X	S32	-2147483648-2147483647	The X position after it transforms to another X position. Usually, this X position is the machine's origin or another part of the machine frame. [mm]
Y	S32	-2147483648-2147483647	The Y position after it transforms to another Y position. Usually, this Y position is the machine's origin or another part of the machine frame. [mm]
X_Std_Dev	U32	1-4294967295	The standard deviation of the machine location along the x-axis. Smaller numbers indicate more confidence in the machine location. [mm]
Y_Std_Dev	U32	1-4294967295	The standard deviation of the machine location along the y-axis. Smaller numbers indicate more confidence in the machine location. [mm]
Updated	BOOL	T/F	Indicates whether the machine's X and Y position data is updated. T: New data is available. F: New data is not available.
Altitude	BUS	——	A bus that contains machine altitude information after transformation, including standard deviation and updated conditions.
Altitude	S32	-2147483648-2147483647	The altitude of the machine's location, which is the distance from sea level up to the machine. [mm]
Altitude_Std_Dev	U32	1-4294967295	The standard deviation of the machine location along the z-axis, referring to the machine's distance above sea level. Smaller numbers indicate more confidence in the machine location. [mm]

Transform_GNSS Function Block

Item	Type	Range	Description [Unit]
Updated	BOOL	T/F	Indicates whether the machine's altitude data is updated. T: New data is available. F: New data is not available.
Yaw	BUS	——	A bus that contains machine yaw information after transformation, including standard deviation and updated conditions. Yaw refers to the machine's rotation around the z-axis.
Yaw	S32	-72000 - 72000	The angle of the machine's rotation around the z-axis using the ENU (East-North-Up) reference frame. [0.01 degree]
Yaw_Std_Dev	U32	1-4294967295	The standard deviation of the machine's rotation around the z-axis. Smaller numbers indicate more confidence in the machine's rotation. [0.01 degree]
Updated	BOOL	T/F	Indicates whether the machine's yaw is updated. T: New data is available. F: New data is not available.

Internal Signals

The following table describes what happens internally in the **Transform_GNSS** function block.

View the internal signals on the Service Tool screen. In PLUS+1® GUIDE, these signals are in the **Checkpoints** page in the **Internal Signals** column.

Item	Type	Range	Description [Unit]
Rot_Matrix	ARRAY[9]S32	-2147483648-2147483647	Includes all elements of the rotation matrix for troubleshooting purposes.

Transform_GNSS Troubleshooting

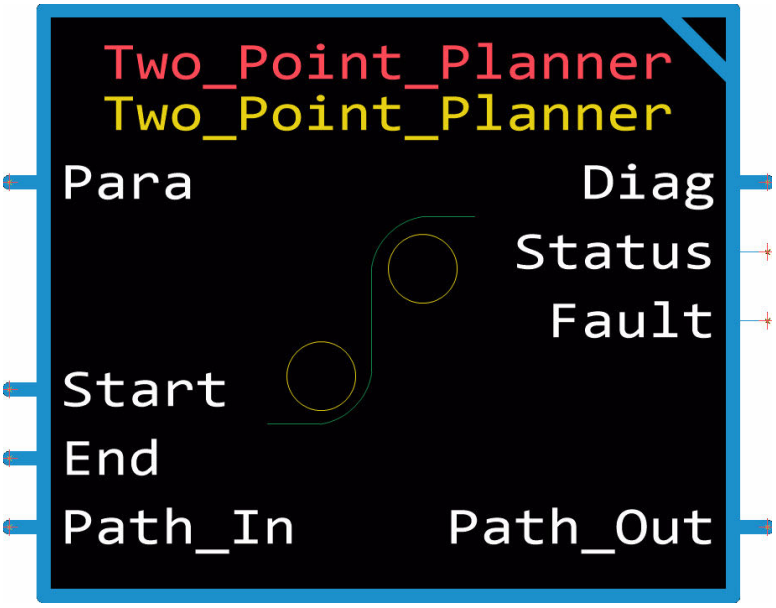
The following table describes errors that could occur in the **Transform_GNSS** function block and ways to fix them. No error codes exist for **Transform_GNSS**.

Other Errors and Fixes

Error Description	How to Fix
There is no information in the outputs.	Check the hardware sensors. Information should come through there into the inputs. All the **Updated** signals should show that information is coming in through the sensors. If information is entering correctly, then check the parameters are valid.

Two_Point_Planner Function Block

The **Two_Point_Planner** function block simplifies the creation of path curves by constraining the machine to fixed radius curves.



Instead of inputting a full path curve, the **Two_Point_Planner** function block only requires a starting and ending position, as well as the desired turning radius. The **Two_Point_Planner** function block then calculates the shortest path between these two points by evaluating the six possible turning combinations. The output of the **Two_Point_Planner** function block is a complete path between the two points which is directly compatible with the **Path_Follower** function block.

The following table describes risks when using the function block.

Risk	Mitigation
Obstacle Collision	Use this function block with Obstacle Avoidance.
Tire Damage	This function block plans the same path for the same points every time, which is the shortest path between those points. Add more waypoints.
Sharp Turns	This function block is developed for all machines. Input the correct turning radius to avoid sharp turns and provide the appropriate propel speeds for the turns involved.

Inputs

The following table describes inputs to the **Two_Point_Planner** function block.

Item	Type	Range	Description [Unit]
Start	BUS	——	Bus containing the machine's start location and driving direction.
X	S32	-2000000000-2000000000	X coordinate of the starting position. [mm]
Y	S32	-2000000000-2000000000	Y coordinate of the starting position. [mm]
Yaw	S32	-72000-72000	The driving direction of the machine's bearing at the starting point. [0.01 degree]
End	BUS	——	Bus containing the coordinates of the machine's destination and driving direction.

Two_Point_Planner Function Block

Item	Type	Range	Description [Unit]
X	S32	-2000000000-2000000000	X coordinate of the machine end position. [mm]
Y	S32	-2000000000-2000000000	Y coordinate of the machine end position. [mm]
Yaw	S32	-72000-72000	The driving direction of the machine at the end position. [0.01 degree]
Path_In	BUS	——	Input bus that contains the path of the machine onto which the estimated path is appended.
Waypoint_X	(ARRAY[X]S32)	-2147483648-2147483647	The X position of the waypoint. [mm]
Waypoint_Y	(ARRAY[X]S32)	-2147483648-2147483647	The Y position of the waypoint. [mm]
Bearing	(ARRAY[X]S32)	-72000-72000	Angle at which the machine goes through the waypoint. This uses the ENU convention and the right-hand rule. [0.01 degree]
Forward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the forward control. Smaller radii yield sharper turns. [mm]
Backward_Radius	(ARRAY[X]U32)	0-4294967295	Distance from the waypoint to the backward control point. Smaller radii yield sharper turns. [mm]
NumOfWaypoints	U8	0-50	The number of waypoints to add this loop.
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Parameters

Parameters to the **Two_Point_Planner** function block are described.

Item	Type	Range	Description [Unit]
Para	BUS	——	Bus containing configuration parameters for the function block.
Radius	U16	1-65535	Sets the turning radius of the machine. Default: 2000 [mm]

Outputs

The following table describes outputs of the **Two_Point_Planner** function block.

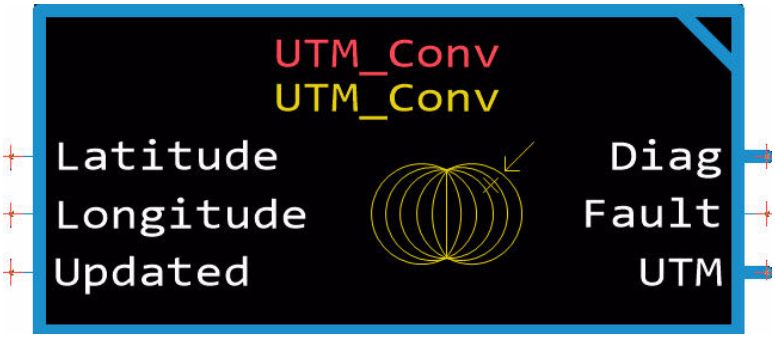
Item	Type	Range	Description [Unit]
Diag	BUS	——	This bus provides diagnostic values for troubleshooting. In addition, this bus contains all inputs, parameters, and output signals.
Status	U16	——	The status of the function block. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. *Non-Standard 0x0000: No fault. 0x0001: Input value too low. 0x0002: Input value too high. 0x0008: NumOfWaypoints_Out overflows.
Path_Out	BUS	——	A path with the shortest curve appended.

Two_Point_Planner Function Block

Item	Type	Range	Description [Unit]
Waypoint_X	(ARRAY[X]S32)	-2147483648-2147483647	X position of the waypoint. [mm]
Waypoint_Y	(ARRAY[X]S32)	-2147483648-2147483647	Y position of the waypoint. [mm]
Bearing	(ARRAY[X]S32)	-72000-72000	Angle at which the machine goes through the waypoint. [0.01 degree]
Forward_Radius	(ARRAY[X]S32)	0-4294967295	Distance from the waypoint to the forward control. Smaller radii yield sharper turns. [mm]
Backward_Radius	(ARRAY[X]S32)	0-4294967295	Distance from the waypoint to the backward control point. Smaller radii yield sharper turns. [mm]
NumOfWaypoints	(ARRAY[X]S32)	0-50	The desired number of waypoints to add this loop.

UTM_Conv Function Block

The **UTM_Conv** function block converts latitude and longitude data of the machine into UTM coordinates.



This function block is likely to be used in conjunction with a **Relative_Pos** function block to achieve relative Cartesian position estimates from GNSS.

The conversion outputs:

- UtmX
- UtmY
- UTM zone
- UTM band

Use the **UTM_Conv** function block with the **Origin** and **Relative_Pos** function blocks.

Input data types must exactly match the indicated type for a successful compile.

The function block does not support polar (UTM) zones A, B, Y or Z.

Inputs

Inputs to the **UTM_Conv** function block are described.

Item	Type	Range	Description [Unit]
Latitude	S32	-800000000-840000000	The latitude value. [0.0000001 degree]
Longitude	S32	-1800000000-1800000000	The longitude value. [0.0000001 degree]
Updated	BOOL	T/F	True when there is new data to convert. Outputs are updated. T: Convert latitude and longitude. Update outputs. F: Do not convert data. Do not update outputs.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

Outputs

Outputs of the **UTM_Conv** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8100: Invalid ECU.

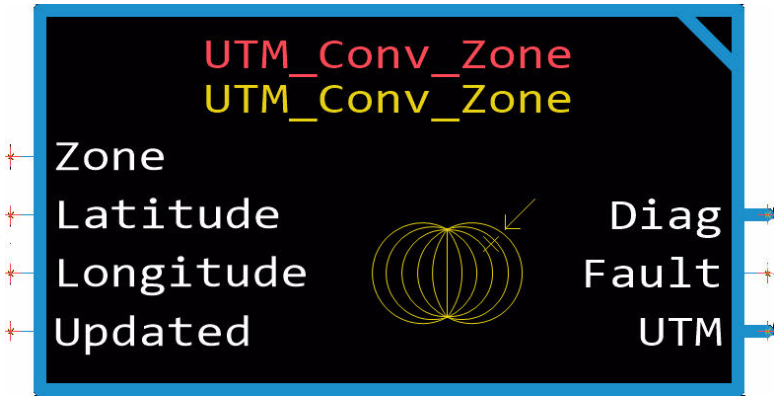
UTM_Conv Function Block

Item	Type	Range	Description [Unit]
Fault	U16	——	Bitwise code where multiple items can be reported at a time. *Non-standard 0x0000: No fault. 0x0001: Input value too low. 0x0002: Input value too high. 0x0004: Latitude is in the UPS Zone range.
UTM	BUS	——	Latitude and longitude data converted into Cartesian units based on the Universal Transverse Mercator projection.
UtmX	U32	0-10^9	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	0-10^{10}	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	——	The 32 most significant bits of UtmY as stored in a U64 value.
UtmY_Lower	U32	——	The 32 least significant bits of UtmY as stored in a U64 value.
Band	U8	0, 67-72, 74-78, 80-88	The band where the UtmX and UtmY values are. 0: (NULL) The latitude value is outside of the conversion limits.
Zone	U8	1-60	The zone that the UtmX and UtmY values are in.
Updated	BOOL	T/F	True When new data is available from the conversion. T: New data is available. F: New data is not available.

Band ASCII Values			
67	C	78	N
68	D	80	P
69	E	81	Q
70	F	82	R
71	G	83	S
72	H	84	T
74	J	85	U
75	K	86	V
76	L	87	W
77	M	88	X

UTM_Conv_Zone Function Block

The **UTM_Conv_Zone** function block receives the machine's latitude and longitude information and converts this information into UTM coordinates.



The conversion outputs:

- UtmX
- UtmY
- UTM zone
- UTM band

Unlike the **UTM_Conv** function block, the **UTM_Conv_Zone** function block provides the ability to manually input zone information. This is useful if the autonomous machine is operating close to the boundary of two different zones.

It is also used with the **Origin** and **Relative_Pos** function block to ensure that the machine position is calculated in the same UTM Zone as the origin.

The function block does not support polar (UTM) zones A, B, Y or Z.

Inputs

Inputs to the **UTM_Conv_Zone** function block are described.

Item	Type	Range	Description [Unit]
Zone	U8	1-60	The UTM zone into which the latitude and longitude values are converted. When a Zone value is provided for an Input, the value is the Input zone.
Latitude	S32	-800000000-840000000	The latitude value. [0.0000001 degree]
Longitude	S32	-1800000000-1800000000	The longitude value. [0.0000001 degree]
Updated	BOOL	T/F	True when there is new data to convert. Outputs are updated. T: Convert latitude and longitude. Update outputs. F: Do not convert data. Do not update outputs.
Chkpt	BOOL	T/F	Enables advanced checkpoints with namespace for each Diag signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.

UTM_Conv_Zone Function Block

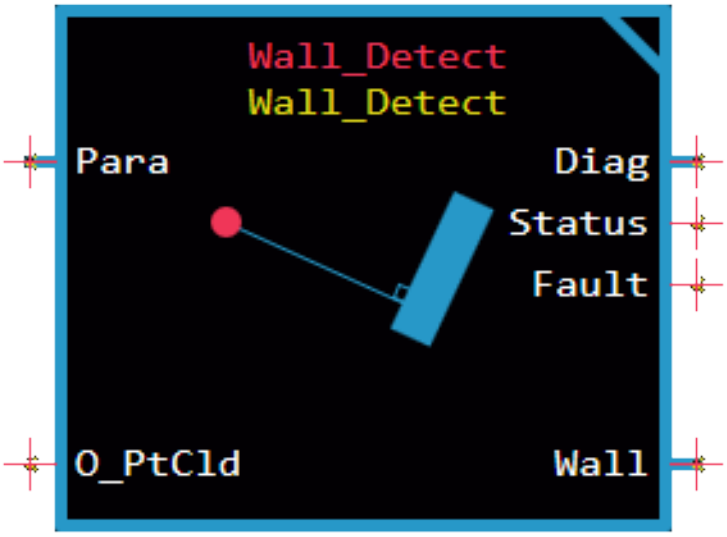
Outputs

Outputs of the **UTM_Conv_Zone** function block are described.

Item	Type	Range	Description [Unit]
Diag	BUS	——	Provides diagnostic values for troubleshooting.
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x0001: Input value too low. 0x0002: Input value too high. 0x0004: Latitude is in the UPS Zone range. 0x0008: Out of conversion range.
UTM	BUS	——	The output bus contains the results of the conversion.
UtmX	U32	$0\text{-}10^{9}$	The UTM Easting (X) value of the origin. [mm]
UtmY	U32 This uses two U32 types, equivalent to a U64.	$0\text{-}10^{10}$	The UTM Northing (Y) value of the origin. [mm]
UtmY_Upper	U32	$0\text{-}10^{10}$ This is the range of the full U64 bit number.	The 32 most significant bits of UtmY as stored in a U64 value. [mm]
UtmY_Lower	U32	$0\text{-}10^{10}$ This is the range of the full U64 bit number.	The 32 least significant bits of UtmY as stored in a U64 value. [mm]
Band	U8	0, 67-72, 74-78, 80-88	The latitude band where the UtmX and UtmY values are. Values are represented in ASCII, not letters. 0: (NULL) The latitude value is outside of the conversion limits.
Zone	U8	1-60	The UTM zone that the UtmX and UtmY values are in. This is also the same zone specified on the input.
Updated	BOOL	T/F	True when new data is available from the conversion. T: New data is available. F: New data is not available.

Wall_Detect Function Block

The **Wall_Detect** function block parses a LiDAR scan, fits a line to the scan looking for a continuous surface, then reports the angle and distance information.



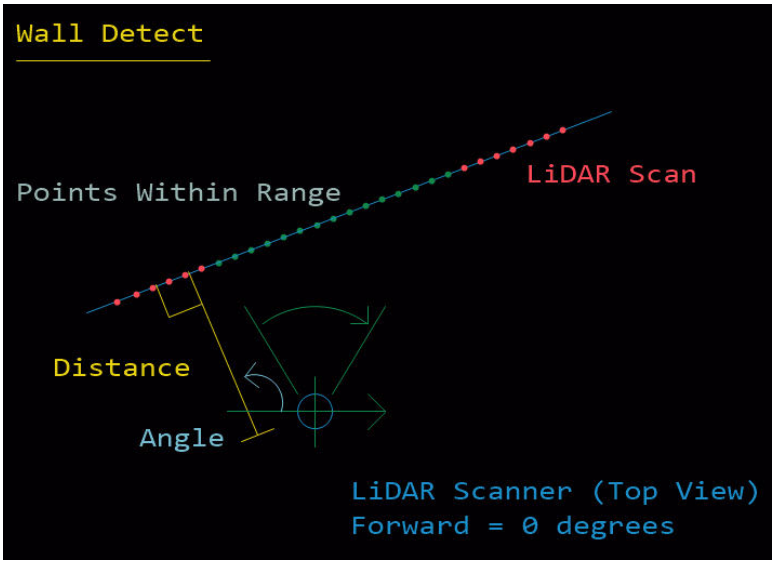
This block requires a LiDAR scanner and the accompanying code. See the *Plus+1 Compliant Ouster Block User Manual* for information about the Ouster LiDAR scanner and block.

This can be used to find any generic smooth feature, such as a wall. This function block can limit the range of the scan to perform a better fit and provide more accurate results.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* for more information about creating unique namespaces.

This function block requires the 'Data_Lockers' block to compile and function correctly. Place the 'Data_Lockers' block, only once, anywhere in the application from the 'Utility' category of the latest version of Autonomous Control Library.



Wall_Detect Function Block

Inputs

Inputs to the **Wall_Detect** function block are described.

Item	Type	Range	Description
Chkpt	BOOL	T/F	Enables Advanced Checkpoints with Namespace for each **Diag** signal. T: Include checkpoints when compiled. F: Do not include checkpoints when compiled.
O_PtCld	S8	-1-99	The data locker ID of an ordered point cloud data.

Parameters

The **Wall_Detect** function block's operating characteristics are set by para bus input signals.

Item	Type	Range	Description [Unit]
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Start_Angle	S16	-18000-18000	This parameter specifies which beam of the LiDAR scan to use to start parsing for an edge. Default: -18000 [0.01 deg]
Stop_Angle	S16	-18000-18000	This parameter specifies which beam of the LiDAR scan to use to stop parsing for an edge. Default: 18000 [0.01 deg]
Max_Distance	U32	0-100000	This parameter determines how far away a data point can be and still be used in the least-squares fit calculation. This can be used to ensure that the function block is not fitting features that are too far away. Default: 10000 [mm]
Ring	U16	0-65535	The horizontal ring row of the 3D LiDAR scan. Set to zero for a 2D LiDAR. Zero is the bottom ring row. Default: 0

Outputs

Outputs of the **Wall_Detect** function block are described.

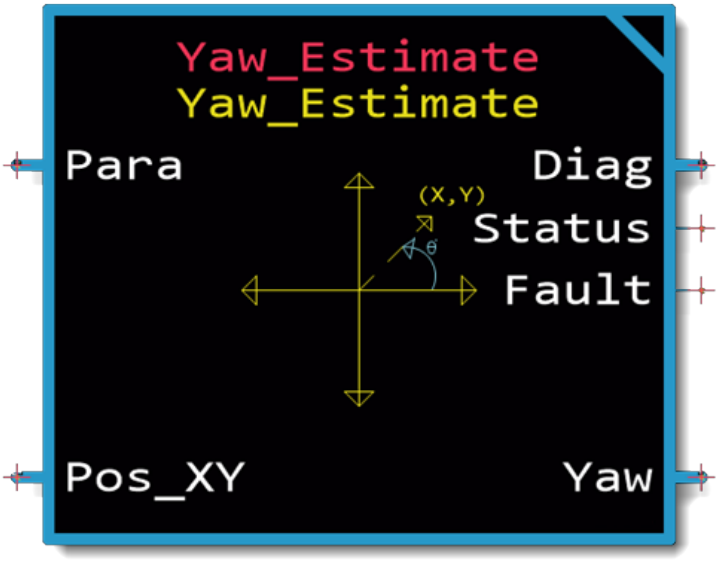
Item	Type	Range	Description [Unit]
Status	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Diag	BUS	——	This bus provides diagnostic values for troubleshooting and information about the current status of the function.
Mid_Distance	U32	0-4294967295	Distance to the midpoint of the line. [mm]
Mid_Bearing	S16	-18000-18000	[0.01 deg]
Wall_Detect_Err	U8	0-4	Indicates errors occurred in the function block operation. 0: No error. 1: Unable to create thread. 2: Not enough memory available to create thread. 3: Thread timeout. 4: Point cloud is unordered.

Wall_Detect Function Block

Item	Type	Range	Description [Unit]
Processing_Time	U32	0-4294967295	The time it took to fit a line to the data. [µs]
Wall	BUS	——	The **Wall** bus contains the updated information about the location of the detected wall.
Updated	BOOL	T/F	New information is available from the block. T: New data is available. F: New data is not available.
Angle	S16	-18000-18000	Angle to the edge in radial coordinates. [0.01 deg]
Distance	U32	0-4294967295	Distance to the edge in radial coordinates. [mm]
Angle_Std_Dev	U32	0-4294967295	Angle to the edge in radial coordinates. [0.01 deg]
Std_Dev	U32	0-4294967295	The standard deviation of the line fit. A lower value indicates a better fit, a higher value indicates a poor fit. [mm]
Seq_ID	U32	0-4294967295	The unique identifier of the point cloud data frame that the function block most recently processed. This number updates every loop and should increase every time a new point cloud scan occurs. The ID could be tracked through different function blocks.

Yaw_Estimate Function Block

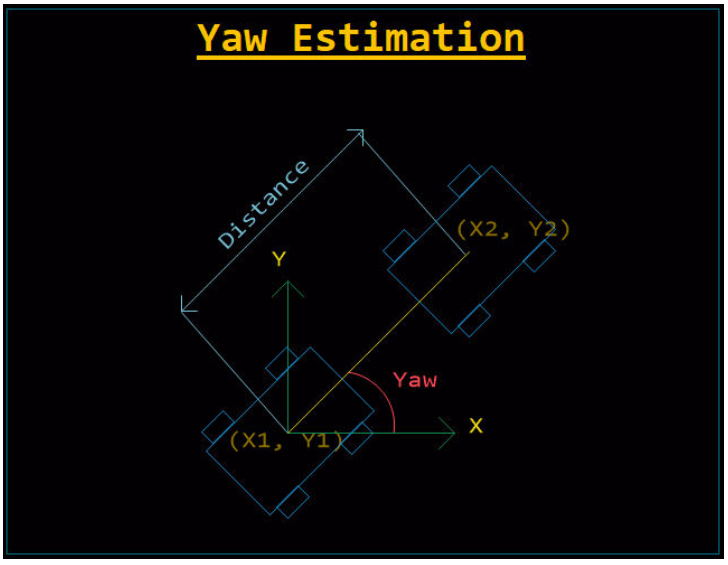
The **Yaw_Estimate** function block estimates the machine's current yaw value by calculating the angle of the line between a previous position and the current position of the machine when it travels a certain distance.



Use this function block when you can not get yaw from an IMU or directly from your GNSS unit.

Input data types must exactly match the indicated type to successfully compile.

The checkpoints page includes advanced checkpoints for each input, output, and internal signal. These require a unique namespace to prevent multiple checkpoints with the same name. See the topic *Change Namespace Value* for more information about creating unique namespaces.



Use the **Yaw_Estimate** function block to calculate the angle of the machine with respect to the X axis, traveling in an XY plane. Whenever the machine displaces from a previous position by set distance, yaw is estimated using a trigonometric function and the machine position is updated.

If the machine drives in reverse and does not have a smart antenna, program logic to flip the yaw between the **Yaw_Estimate** function block output and **Position_Filter** input.

Yaw_Estimate Function Block

Inputs

The table below describes inputs to the **Yaw_Estimate** function block.

Item	Type	Range	Description
Pos_XY	BUS	——	The primary input to the block is the Cartesian coordinates of the machine location and respective standard deviations.
X	S32	-2147483648-2147483647	Cartesian X component of machine location. [mm]
Y	S32	-2147483648-2147483647	Cartesian Y component of machine location. [mm]
X_Std_Dev	U32	1-4294967295	Standard deviation of X. [mm]
Y_Std_Dev	U32	1-4294967295	Standard deviation of Y. [mm]
Updated	BOOL	T/F	True if new X and Y values are available for yaw estimate. T: Use the new X and Y values. F: Do not update the X and Y value being used.
Chkpt	BOOL	T/F	TRUE when there is new X and Y values to be used for the Yaw calculation. T: Use the new X and Y values. F: Do not update the X and Y values being used.

Parameters

Parameter to the **Yaw_Estimate** function block are described.

Item	Type	Range	Description
Para	BUS	——	Adjust configuration values here, or replace them with signals routed from the application through the **Para** bus.
Distance	U16	100-5000	Minimum distance traveled before Yaw is recalculated. Default: 500 [mm]

Outputs

Outputs of the **Yaw_Estimate** are described.

Item	Type	Range	Description
Diag	BUS	——	Provides diagnostic values for troubleshooting and information about the current status of the machine positions.
Status	U16	——	Bitwise code for distance. 0x0000: Status OK. 0x8008: At least one parameter is out of range or in the wrong order. 0x8100: Invalid ECU.
Fault	U16	——	Bitwise code where multiple items can be reported at a time. 0x0000: No fault. 0x8001: Input value too low. 0x8002: Input value too high.
Yaw	BUS	——	The **Yaw** bus contains estimated information of the machine Yaw.
Yaw	S32	-72000-72000	The estimated yaw value. [0.01 deg]

Yaw_Estimate Function Block

Item	Type	Range	Description
Yaw_Std_Dev	U32	1-4294967295	Standard deviation of the estimated yaw. [0.01 deg]
Updated	BOOL	T/F	True when there is new data. T: New data is available. F: No new data was converted.

Third Party Licenses

Included below are third party licenses Autonomous Control Library uses.

cJSON License

GitHub - DaveGamble/cJSON: Ultralightweight JSON parser in ANSI C

TinyEKF License

ENGINEERING
TOMORROW

Products we offer:

- Cylinders
- Electric converters, machines, and systems
- Electronic controls, HMI, and IoT
- Hoses and fittings
- Hydraulic power units and packaged systems
- Hydraulic valves
- Industrial clutches and brakes
- Motors
- PLUS+1® software
- Pumps
- Steering
- Transmissions

Hydro-Gear
www.hydro-gear.com

Daikin-Sauer-Danfoss
www.daikin-sauer-danfoss.com

Danfoss Power Solutions designs and manufactures a complete range of engineered components and systems. From hydraulics and electrification to fluid conveyance, electronic controls, and software, our solutions are engineered with an uncompromising focus on quality, reliability, and safety.

Our innovative products makes increased productivity and reduced emissions a possibility, but it's our people who turn those possibilities into reality. Leveraging our unsurpassed application know-how, we partner with customers around the world to solve their greatest machine challenges. Our aspiration is to help our customers achieve their vision — and to earn our place as their preferred and trusted partner.

Go to www.danfoss.com or scan the QR code for further product information.

Danfoss
Power Solutions (US) Company
2800 East 13th Street
Ames, IA 50010, USA
Phone: +1 515 239 6000

Danfoss
Power Solutions GmbH & Co. OHG
Krokamp 35
D-24539 Neumünster, Germany
Phone: +49 4321 871 0

Danfoss
Power Solutions ApS
Nordborgvej 81
DK-6430 Nordborg, Denmark
Phone: +45 7488 2222

Danfoss
Power Solutions Trading (Shanghai) Co., Ltd.
Building #22, No. 1000 Jin Hai Rd
Jin Qiao, Pudong New District
Shanghai, China 201206
Phone: +86 21 2080 6201