# Graphical Composition

## Getting Started with Signal Composition

Version: 230721

*Manual*

Document: Manual- Graphical Composition
Getting Started with Signal Composition

Release date: 2023-07-21

Document version: 1.0

Author: FORCAM GmbH

# Contents

# 1 About this document

This document describes how to use the graphical Composition editor, which you can use to easily interpret asset signals.

ⓘ The graphical composition is part of FORCE EDGE CONNECT (hereafter simply referred to as EDGE CONNECT), it can therefore only be used for signal interpretation within this application.

The manual explains the different functions, elements and possibilities of the graphical signal interpretation based on the different topics (functional areas). Each function is explained with a practical example.

ⓘ For better readability, we generally use the generic masculine in the text. These formulations, however, are equally inclusive of all genders and intended to address all persons equally.

## Target group

In this manual, we assume that you have knowledge in the following areas:

- Knowledge of machine connection and configuration with EDGE CONNECT
  See the FORCE EDGE CONNECT manual for detailed information.
- Basic knowledge in signal processing / electronic data processing

If you do not have any knowledge in this area, take the time to familiarize yourself with the basics.

ⓘ We recommend that you use our Academy: **https://forcam.com/academie/**
The FORCAM Academy provides the knowledge to effectively use the methods for digital transformation and the technologies for the Smart Factory.
Based on lean manufacturing and TPM methods, our institute team will guide you and provide support to initiate changes in the company and to use the technologies correctly.

## Additional information

Our **Customer Area** provides all manuals, product descriptions and further information about your release.

# 2 Graphical composition in EDGE CONNECT

If you want to evaluate the signals that were read from an asset (machine, sensor, etc.), first these signals must be interpreted.
Without any programming knowledge, you can use graphical composition to define quickly and easily, which signals should be processed in which way. You can also define, when data is to be sent to an MES, ERP or another third-party system. The graphical composition is thus part of the Signal Composition, it can be used instead of the script editor during signal composition.

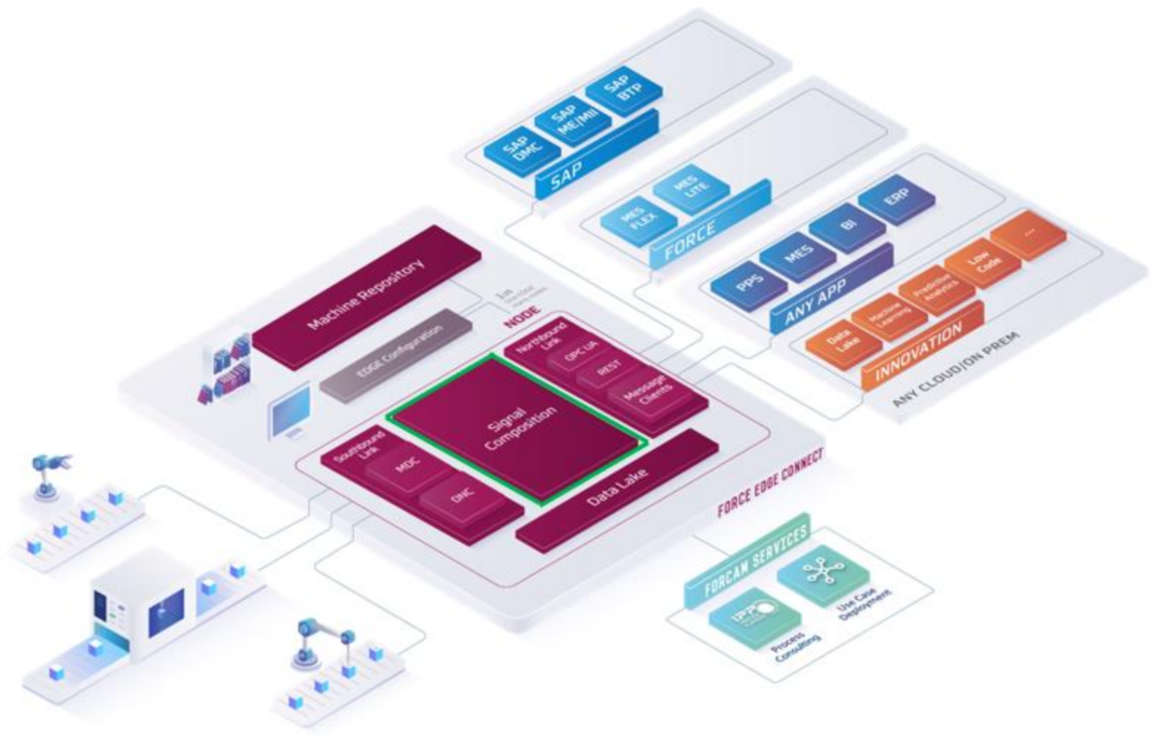
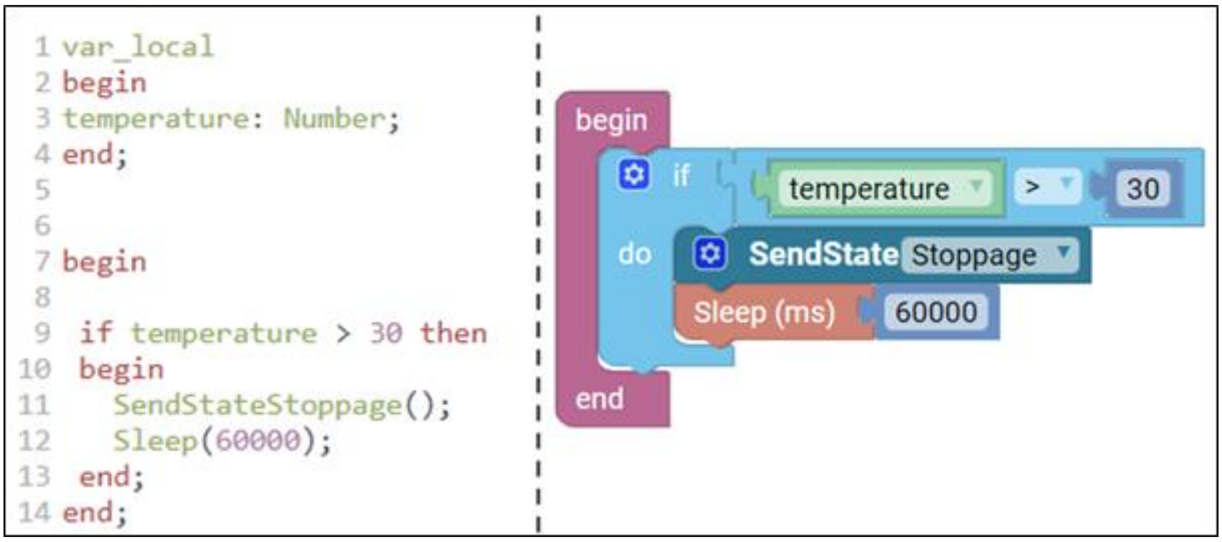


**Figure 1: Positioning of the graphical composition in EDGE CONNECT**

## Graphical vs. script-based signal interpretation

In the Signal Composition component, meanings are assigned to signals. For example, a pure numerical value (such as 0 or 1) is turned into a readable and understandable information, such as “Production” or “Stoppage”.
A script is a short sequence of commands that are executed by the program. The graphical composition provides a simple and beginner-friendly alternative to classical scripting, making signal interpretation accessible to everyone. The graphical composition is used like a modular system.



```
var_local
begin
temperature: Number;
end;


begin

 if temperature > 30 then
 begin
   SendStateStoppage();
   Sleep(60000);
 end;
end;
```

**Fig. 2: Script-based vs. graphical editor**

- The individual commands are visualized in a graphical way with the help of colored puzzle pieces.
  Beginners in programming can thus execute basic commands without any prior knowledge of a specific programming language.
- Application-supported error prevention right from the start.
  The use of graphical elements excludes syntax errors.
  Different mechanisms make it easier to recognize other types of errors.
- The clear presentation of the categories and functions visualizes the complete range of functions and facilitates the handling of the editor.
- The graphical composition can also be combined with an MR template.

For general information on signal interpretation during asset configuration, refer to the FORCE EDGE CONNECT manual.

# 3 Working with the graphical composition

## 3.1 The user interface

The graphical composition is part of the Configuration Wizard. There, conditions for the interpretation of the signals are defined in the **Composition** step.
The graphical editor is called from the **GRAPHIC** tab. The various programming blocks are shown as graphical blocks here, which can be assembled in a modular fashion.

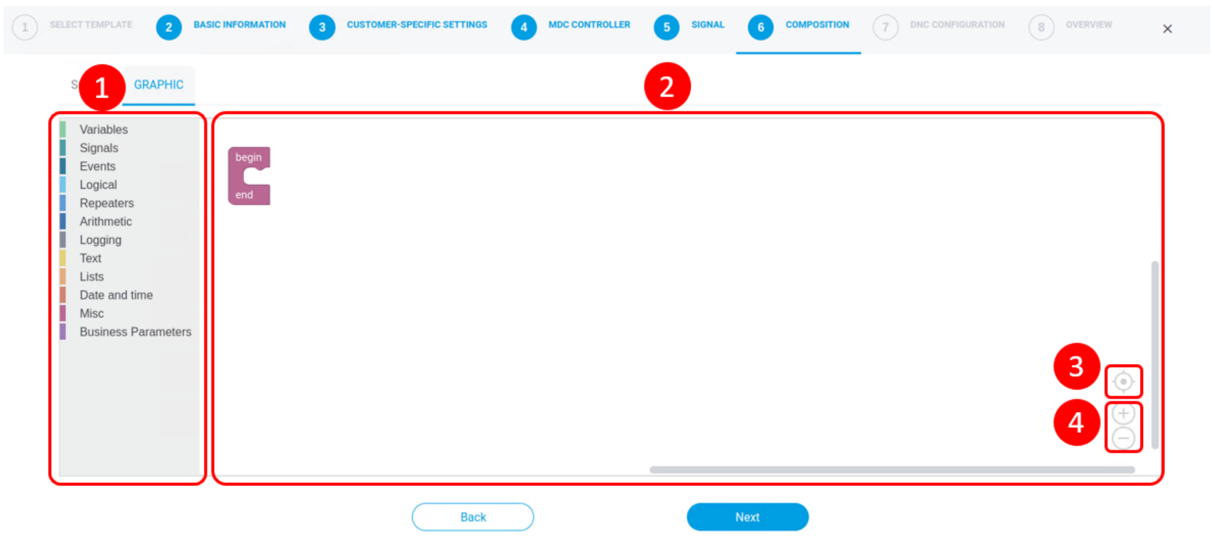


**Fig. 3: Graphical editor**

(1) Select blocks via their function categories
(2) Assemble blocks in the editing window ("Scripting")
(3) Center view
(4) Zoom in/out view

General information about the different types of blocks and categories can be found in the following sections.

## 3.2 Blocks and function categories

Each "puzzle piece" in the graphical composition is a block and usually corresponds to a function or action (function blocks). The blocks are composed in a modular way. Matching "building blocks" can be combined with each other to form an overall structure that all required signal processing commands.

Depending on their function, the blocks are grouped into function categories, e.g., all blocks for creating and managing lists are contained in the *Lists* category).
In addition to the general function blocks, there are mandatory blocks that can be used to pass on values (shadow blocks), and optional blocks that can transmit additional information.

### 3.2.1 Function categories

The function blocks are grouped into different categories according to their functions/topics. Each category has a color assigned. All blocks of a category have the same color. This way, it is easy to distinguish the individual blocks.

When you select a category, the associated function blocks are displayed to the right of it (see “

Working in the graphical editor”).

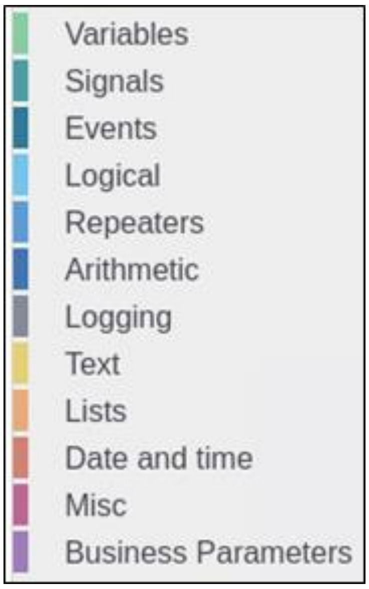


**Figure 4: Function categories**

The following table provides an overview of all available categories and their usage as well as links to the corresponding chapters in this manual.

| Category | Usage | See chapter |
|---|---|---|
| **Variables** | Create and read variables, write values to variables.<br><br>(Variables serve as containers for storing data.) | 4 "Process variables (Variables)" |
| **Signals** | Read, distinguish and convert signal values.<br><br>(Signals contain and transmit information that is usually measured by a sensor.) | 5 "Interpret signals (Signals)" |
| **Events** | Send data (impulses, production states or values) to third-party systems | 6 "Define events (Events)" |
| **Logical** | Establish correlations between values<br><br>This enables decisions about their logical value or status. | 7 Make logical connections (Logical) |
| **Repeaters** | Perform actions in defined time intervals | 8 Repeaters |
| **Arithmetic** | Perform arithmetic functions (add, subtract, multiply values)<br><br>Convert data formats | 9 Maths operations (Arithmetic) |
| **Logging** | Log and output specific values | 10 Log values (Logging) |
| **Text** | Create and process texts | 11 Create and process texts (Text) |
| **Lists** | Create, fill, empty and delete lists | 12 Create and manage lists (Lists) |
| **Date and time** | All actions related to time or date settings | 13 Managing times (Date and time) |
| **Misc** | Collection of additional commands and functions for communication with other systems | 13Managing times (Date and time) |
| **Business Parameters** | Read and process specific properties of the machine<br><br>(The corresponding data is provided in the Configuration Wizard in the previous configuration steps.) | 14 Additional actions (Misc) |

### 3.2.2 Structure and properties of the blocks

All types of blocks have basically the same structure. They differ in their color and the possibilities of docking to other blocks, depending on their function.

#### Connection points

Each block has connection points to other blocks. As with a puzzle, only matching connections can be combined.

Connection points have different functions, depending on the type of building block:

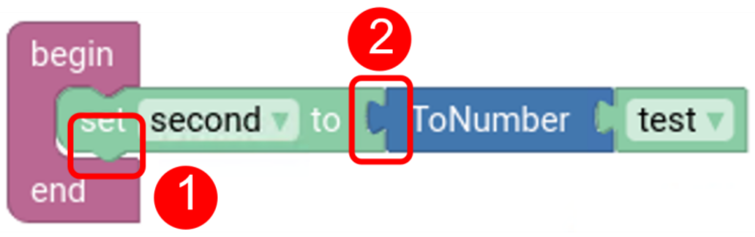


**Fig. 5: Connection points**

(1) Connection of further blocks possible
(2) Passing on input data (from right to left)

⚠ A row must be completely closed on the right (no open connection points).

## Input and output

Inputs are the contents that are required to run the program. The output is the result of processing these inputs and/or the output of a command.

Chapter 17.1 “Parameter overview” provides on overview of all rules and restrictions for the inputs and outputs of the different types of blocks.

## Data type

Each block/variable has certain restrictions about the formats (data types) for input and output values.

The following data types are possible, depending on the block/parameter:

| Data type | Description |
|---|---|
| **Boolean** | A boolean is a TRUE/FALSE value.<br><br>A boolean either indicates that an event is true (1) or false (0). It can also indicate whether an event has actually occurred (true/1) or not (false/0). |
| **String** | String of numbers, letters or symbols.<br><br>This data type is used to represent texts. |
| **Number** | Contains digits |

### Predefined input values

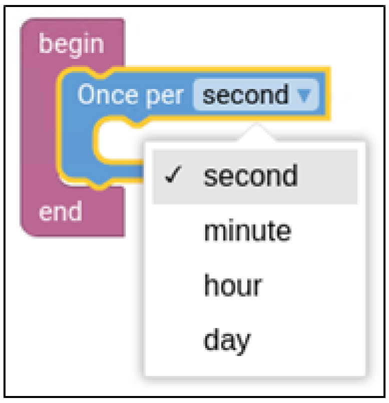


**Fig. 6: Drop-down menu**

blocks have predefined input parameters. Clicking the small triangle on the right displays the options that are available for block.

## 3.2.3 Shadow blocks

Shadow blocks serve as placeholders for inputs that are mandatory for a block to perform its function/the desired action. A shadow block is always connected to a superordinate block if this block is selected in the function category window.

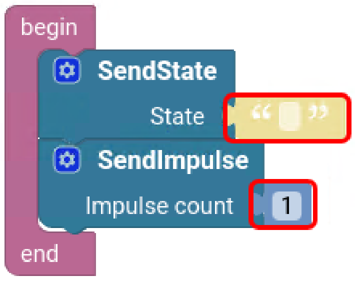


**Fig. 7: Shadow blocks**

A shadow block has a lighter coloring, it indicates that this parameter of the block must not be empty. You must then either enter a value manually (Fig. 7) or use another block to provide the required input.

A given shadow block can be replaced by another block with the same data type.

## 3.2.4 Optional blocks for extension

Some function blocks can be extended by additional blocks with optional parameters. These blocks have a blue gear to make additional settings.

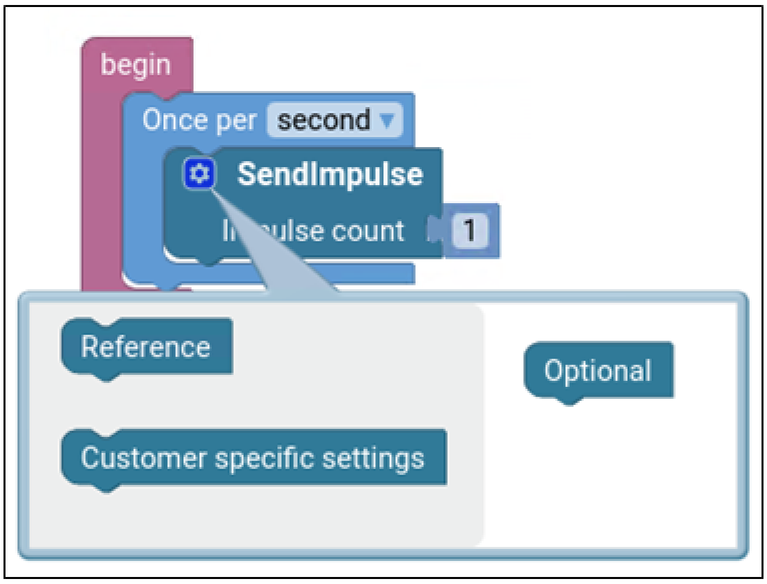


**Figure 8: Optional parameters on a block**

### Reference

This block can be used to transmit a freely selectable value, e.g., text.

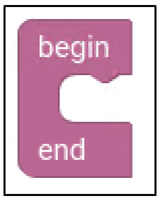
### Customer specific settings

Here you can insert the parameters created in the "Customer-specific settings" step of the Configuration Wizard. Refer to the general manual for more information on these parameters.

ⓘ The optional blocks further down can only be added together with the blocks above (i.e., if the blocks above have been inserted, too.) In the picture above, this means that the **`Customer specific settings`** can only be used after inserting the **`Reference`**.

## 3.3 Working in the graphical editor

Each composition starts and ends with the **begin...end** frame.



This block automatically appears in the editing area. You can determine the further sequence of the blocks within the structure yourself, in accordance with of block characteristics. Each block is can be used on its own and fulfills a specific task.
In order to perform the associated action, a block may require additional information (input) from other blocks that you must attach to the block. The result is a consistent structure of commands for signal processing.

### 3.3.1 Reading direction of the blocks

The blocks are read from <u>top to bottom</u> and from <u>left to right</u>.

#### Example: Reading direction from top to bottom

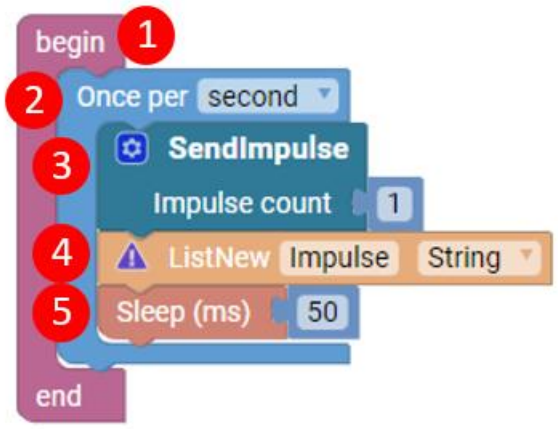


**Fig. 9Example of a top-to-bottom sequence**

The begin...end block (1) is the overall frame of each composition. The other function blocks are processed as indicated by the numbering.

#### Example: Reading direction from left to right

The individual rows are read and processed from left to right, as in a book:

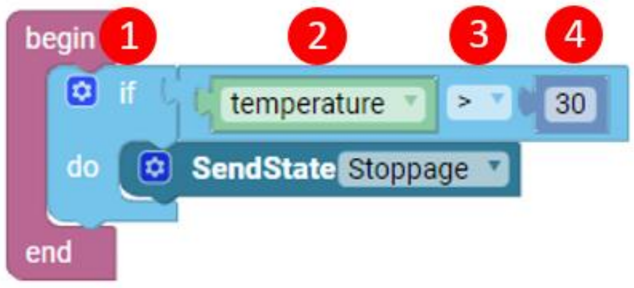


**Fig. 10: Example of a left-to-right processing sequence**

It starts with **if** (1), followed by the green **temperature** block (2), then the mathematical **>** symbol (3), and it ends with the number **30** (4).

This row starts with **do**, after that the content of the dark blue block is also read from left to right. This means that the reading sequence is not affected by the fact that this block consists of two lines.

## 3.3.2 Add and edit blocks

### Add block

To add a block, select the desired category in the left pane and then drag and drop the block into the editing area.
The currently selected block is always outlined in yellow.

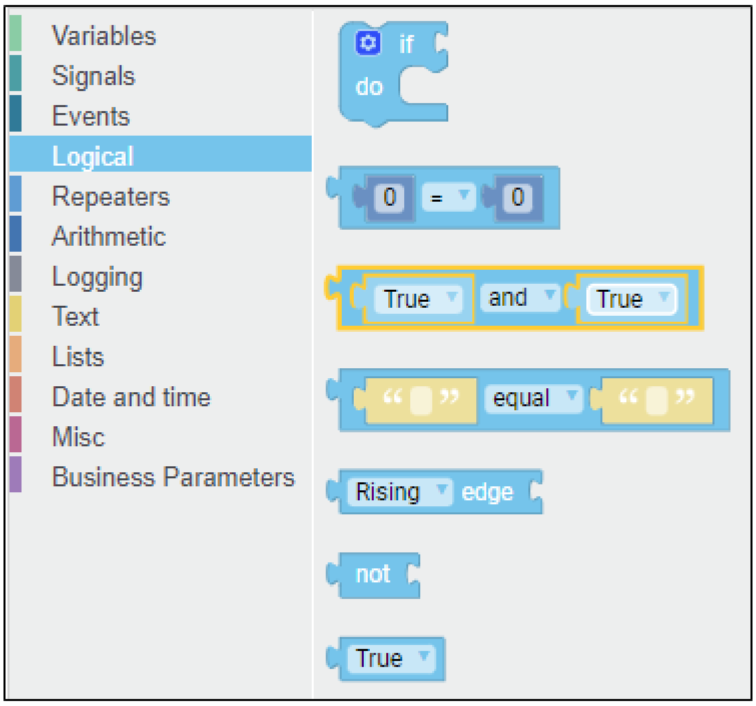


**Fig. 11: Select blocks via their function categories**

Within the editing area, you can also use Drag and Drop to move the blocks to the desired position.

Copying is also possible using the keyboard shortcuts Strg+C / Strg+V.

### Delete blocks

Blocks can be removed from the structure or the editing window by pressing the DEL key or by using the context menu of the block (see below).

### Further actions for a block

Right-clicking a block displays a list of possible actions for a block:

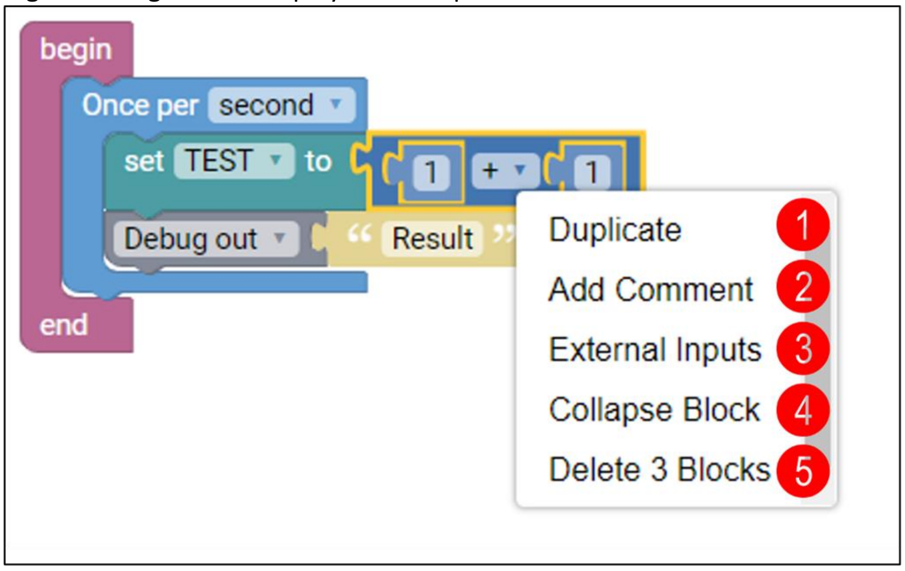


**Fig. 12: Possible actions for a block**

(1) Duplicate block
(2) Add comment
(3) External Inputs/Inline Inputs: Changes the display format
(4) Collapse block
To save space and keep the overall display clear , you can reduce the view for a block (and also the subordinate blocks, if any).
(5) Delete block groups

## 3.3.3 Notation of numbers

The graphical Composition uses the English notation for numbers. Keep this in mind when using periods and commas:

| German | In Words | English |
|---|---|---|
| 0,5 | A half | 0.5 |
| 1.000 | One thousand | 1,000 |
| -1.750,000 | Minus one million seven hundred fifty thousand | -1,750,000 |

## 3.3.4 Error detection

An error in the structure is indicated in two ways:

## Block cannot be inserted

If a block does not fit into the structure in its function category or the data type used, it is not possible to insert it at this position. A typical error would be, for example, that a string is required as input format, but the block contains data of the Number data type.

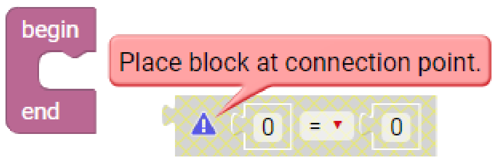


**Fig. 13: Error - invalid block**

## Block is incomplete

An inserted block remains red as long as required information (input) is still missing.
Once the input is complete (values or corresponding blocks inserted), the block returns to its original color.

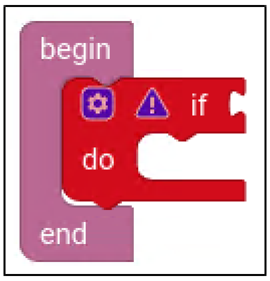


**Fig. 14: Error - incomplete block**

Only valid blocks are accepted.

⚠ By clicking on the exclamation mark, the cause of the error can be displayed and can thus be quickly identified (see Fig. 13).

# 4 Process variables (Variables)

Variables serve as containers for storing data. This can be static values (asset name, status, etc.) or calculation results (temperature, pressure, time unit, etc.).

A variable can only store one specific type of content. For example, pure digits cannot be stored in a text variable. The type of content is determined by the data type of the variable. These types have each specific restrictions regarding their content (numerals, words., etc.) and their size (how small/large, etc.). (For more information on data types, see “Structure and properties of the blocks”).

ⓘ If the content of a variable changes, this is registered throughout the system. All blocks that use this variable automatically adopt the new value.

## Handling variables

Variables must be created in the category first, and can then be used in the graphical editor. Therefore, the category is initially empty.
If you click on the **Variables** functions category, the the available blocks are displayed and/or the button for creating a new variable.

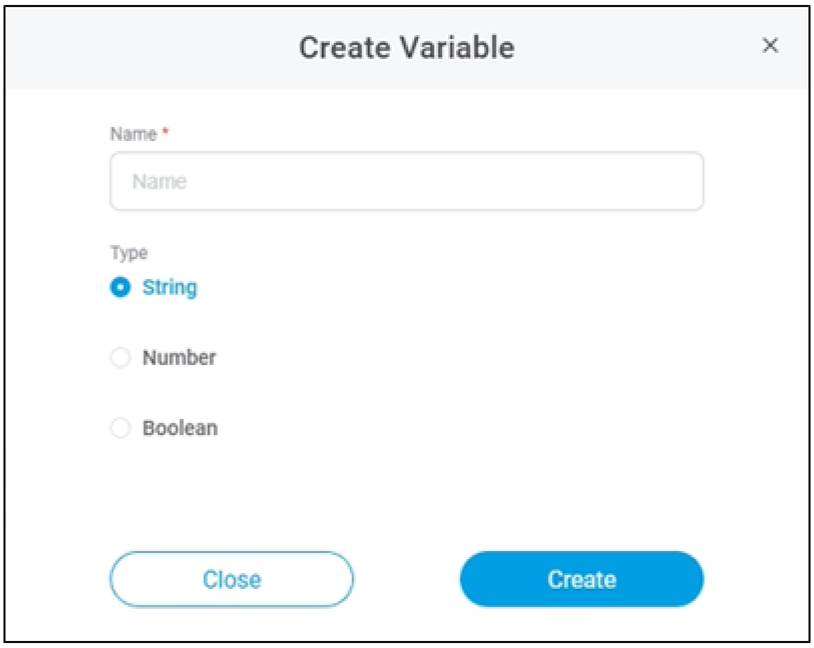


**Fig. 15: Creating a new variable**

The name and also the data type (number, string or boolean) must be defined for each variable.

ⓘ Created variables are only available in the configuration for which they were created, but within the configuration they can be used unlimited times. The name of the variable must be unique within the configuration.

For a better overview, the available variables are displayed according to their data type:

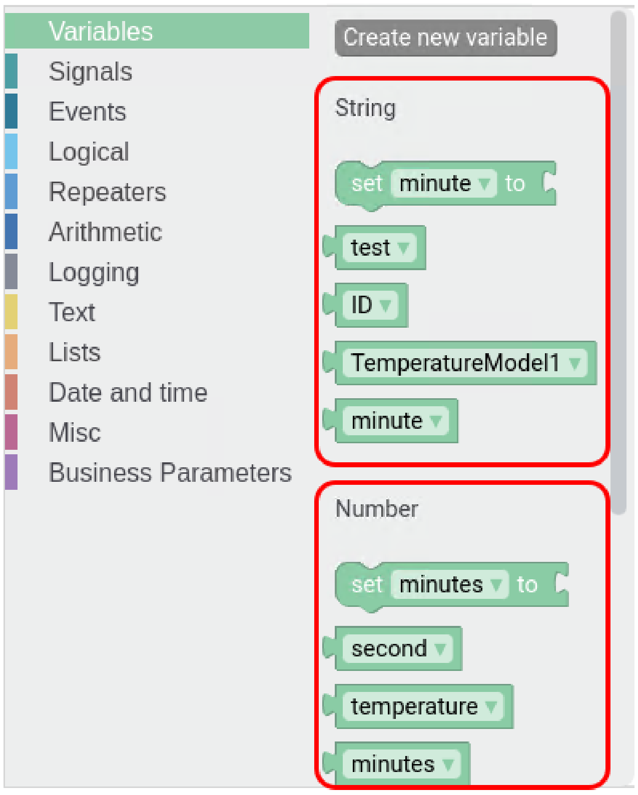


**Fig. 16: Overview of variable types**

Variables can be renamed. To do so, left-click the variable. Select **Rename** in the drop-down menu.

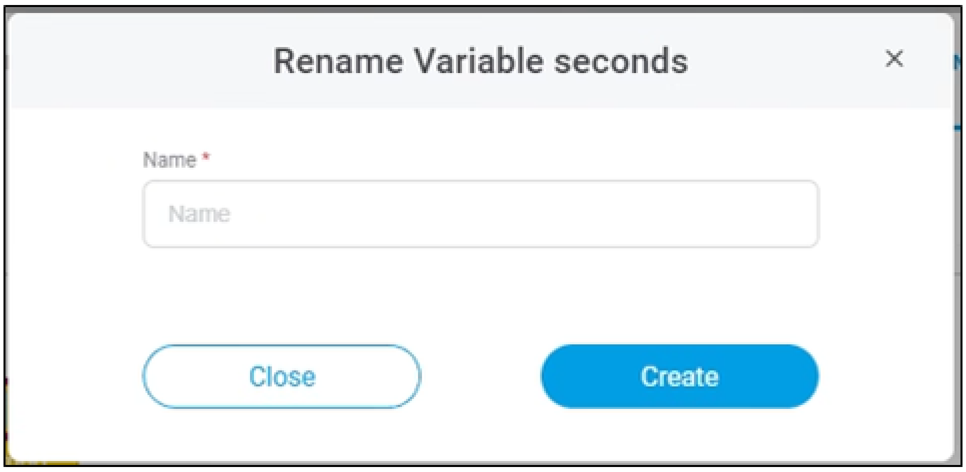
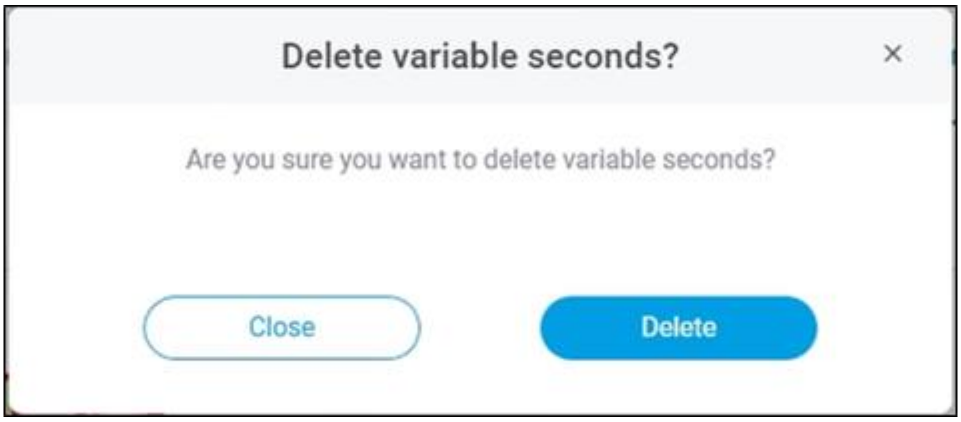


**Fig. 17: Renaming a variable**

The drop-down menu also provides the **Delete** option to delete the variable.

⚠ Each deleted variable is removed completely from the structure and also from the function category. This may result in an invalid structure.



Delete variable seconds?

Are you sure you want to delete variable seconds?

**Fig. 18: Deleting a variable**

## 4.1 [Variable] - read variable

### Usage

This block is required if you want to use a variable in the structure. For each block, all created variables of type String, Number and Boolean can be selected via the drop-down menu.

### Input/Output

| Allowed input types | Output |
|---|---|
| No restrictions | Corresponds to the type of the created variable |

### Example

In this case, once per second the number of seconds shall be increased by 1. To do so, the already created **second** is used. Once the **second** variable gets the value 30, an impulse is sent and **second** is reset to 0.

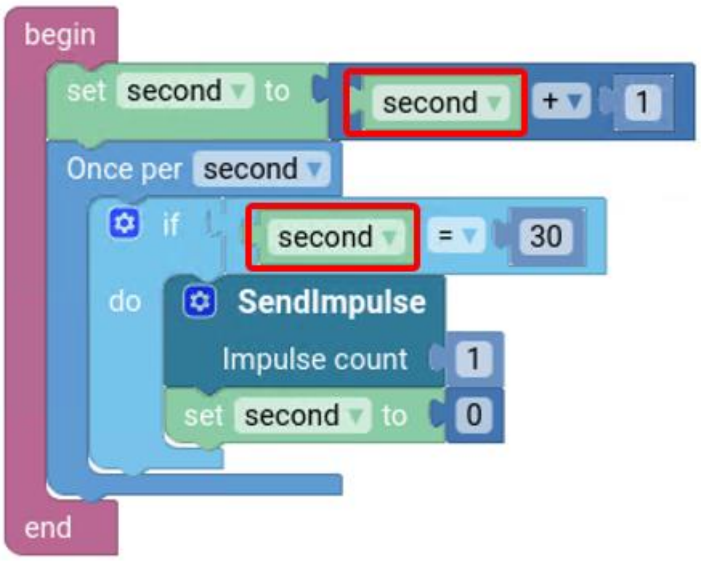
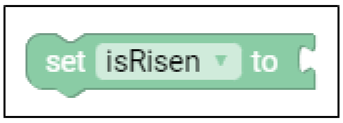


**Fig.19: Example for [Variable]**

## 4.2 Set [Variable] to



### Usage

**`Set [Variable] to`** is a connector. The block is used to assign a value to a variable. Depending on the type of variable, this might be a string, a number or a boolean value. The variable to be used is selected via the drop-down menu.

### Data type

| Allowed input types | Output |
|---|---|
| Corresponds to the type of the created variable | Corresponds to the type of the created variable |

### Example

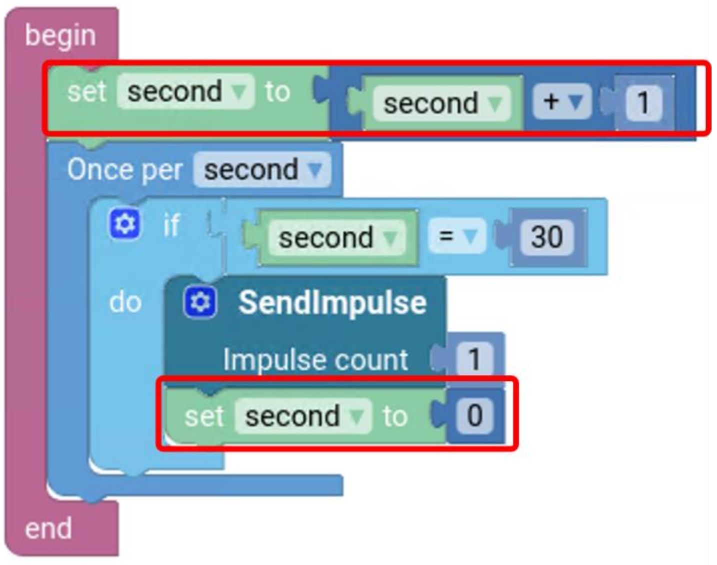


**Fig.20: Example for set [Variable] to**

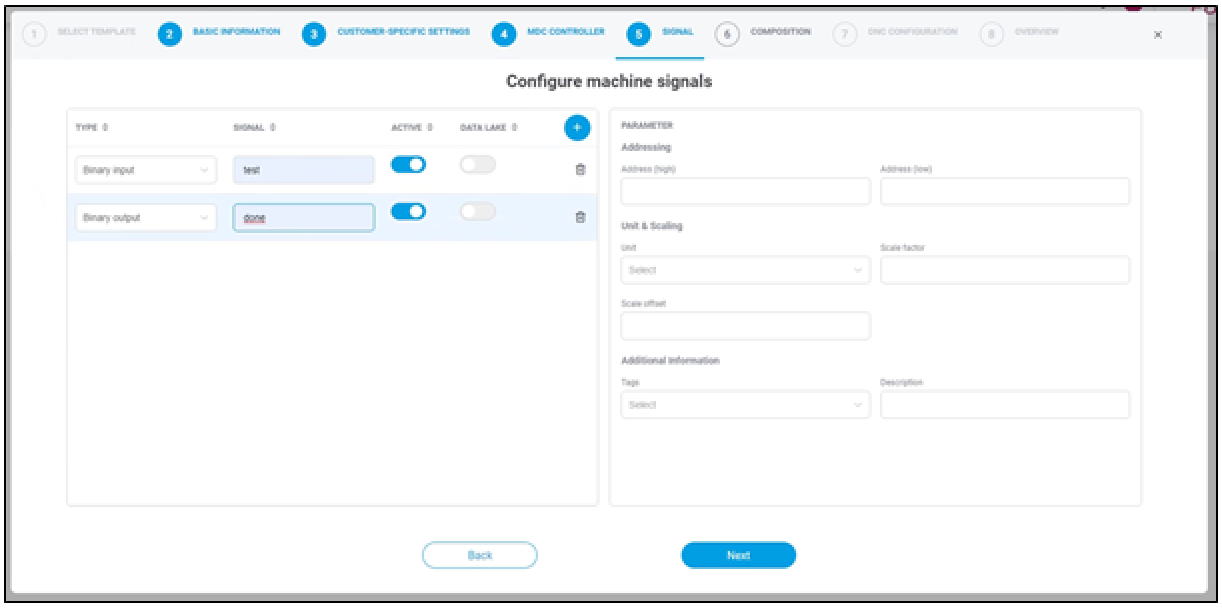
Once per second the number of seconds shall be increased by 1. To implement this requirement, the **`set second to`** block defines that the **second** variable is to be recalculated. Once the **second** variable gets the value 30, an impulse is sent. At the end, the number of seconds is again reset to 0.

# 5 Interpret signals (Signals)

In most cases, signals are detected at the assets using sensors, they transfer the information to EDGE CONNECT. In the production environment, typical signals are intervals, temperatures, machine states and pressures.

## Handling signals

Unlike variables, the signals used come from the assets themselves. Signals can be configured in step 5 and used later in the script.



Configure machine signals

| TYPE | SIGNAL | ACTIVE | DATA LAKE |
|---|---|---|---|
| Binary input | test | ✓ | |
| Binary output | done | ✓ | |

PARAMETER

Addressing

Address (high)

Address (low)

Unit & Scaling

Unit: Select

Scale factor

Scale offset

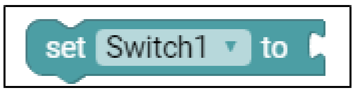
Additional information

Tags: Select

Description

Fig. 21: Adding a new signal

## 5.1 Set [Signal] to



### Usage

**`Set [Signal] to`** is a connector. The block is used to assign a number, string or boolean value to a signal. The signal to be used is selected via the drop-down menu.

### Data type

| Allowed input types | Output |
|---|---|
| No restrictions | No restrictions |

### Example

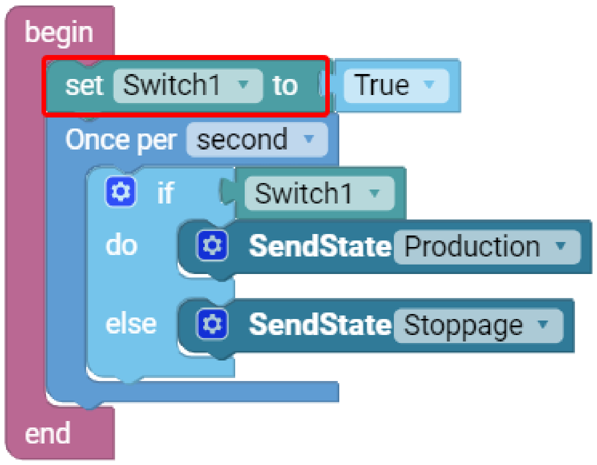
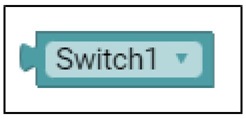


**Fig.22: Example for Set [signal] to**

At first, **`Switch1`**is set to **`True`** (value 1). Once per second a repeater is processed. If **`Switch1`**is switched, the production status is sent to **`Production`**. If not, status **`Stoppage`** is output.

## 5.2 [Signal] - Read signal value



### Usage

This block is required if you want to use signals in the structure. It reads the signal value. The signal to be used is selected via the drop-down menu.

### Data type

| Allowed input types | Output |
|---|---|
| No restrictions | No restrictions |

### Example

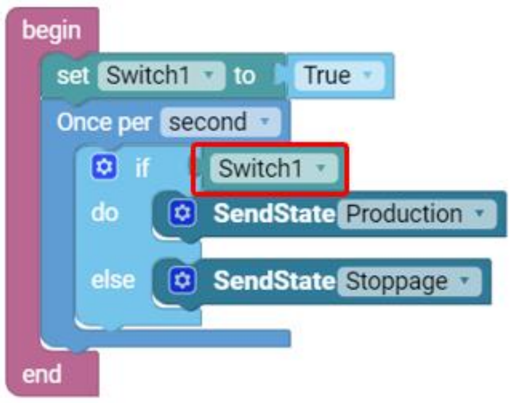
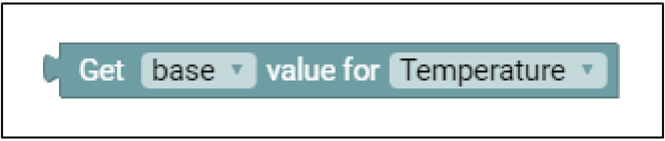


**Fig. 23**: **Example for [Signal]**

If **`Switch1`** switches to **`True`** (value =1), a repeater is called once per second. The repeater checks whether **`Switch1`** was switched. If yes, the production state is set to **`Production`**. If not, status **`Stoppage`** is output.

## 5.3 Get base / scaled value for



### Usage

The **`Get base value`** block converts a signal value into another unit and outputs this value.
The **`Get scaled Value`** block outputs the value that is calculated from scaling and offset.

In step 5 of the Configuration Wizard, numerical signals were entered together with the assigned unit, scaling factor and scaling offset.

The **`base value`** indicates that value in the defined SI base unit.
Scaling factor and scaling offset are defined during signal configuration.
With a defined scaling factor and offset of 0, for example, 0 °C is output as 273,15 °Kelvin.
The **`scaled value`** is the input value multiplied by the scaled factor and the scaled offset.

### Data type

| Allowed input types | Output |
|---|---|
| No restrictions | No restrictions |

### Example

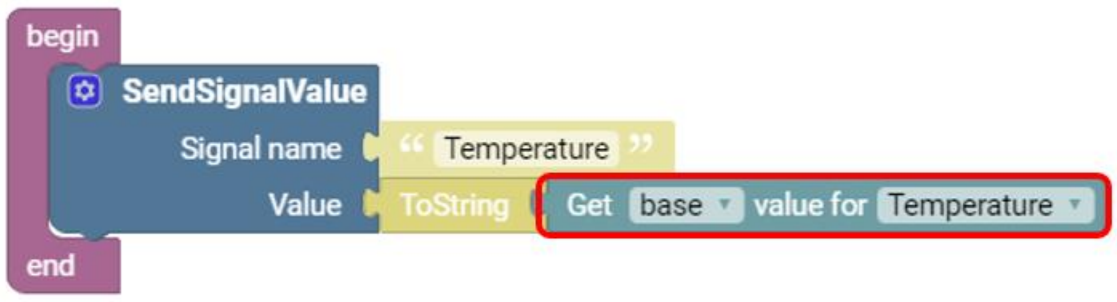


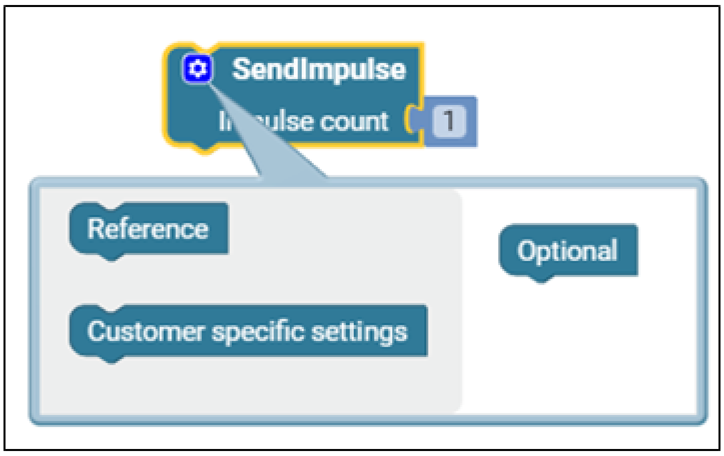
**Fig. 24: Example for Get base value for**

In this example, the temperature value is converted to a different measurement unit and passed on to a third-party system. **`Temperature`** is the **`Signal name`**, which is entered in a text block. The corresponding value is added to the message by the **`Get base value for Temperature`** block. This value must be converted as the **`SendSignalValue`** event only accepts strings as input values. See the following chapters for details.

# 6 Define events (Events)

Events send information packages to third-party systems. The content of these packages (impulses, production states or values) is defined in the graphical composition.

## 6.1 SendImpulse



### Usage

The **`SendImpulse`** block is used whenever a specific impulse is to be sent. The **`Impulse Count`** value defines number of impulses to be sent. Additional blocks (**`Reference`** and **`Customer specific settings`**) can optionally be included.

### Input/Output

Only numbers can be used as input for **`Impulse count`**.
All other input entries must be strings. There are no restrictions to the output.

### Example

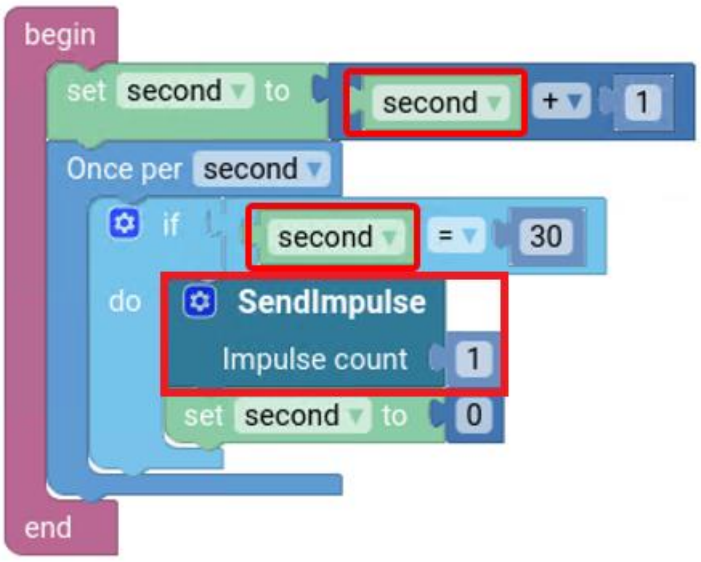
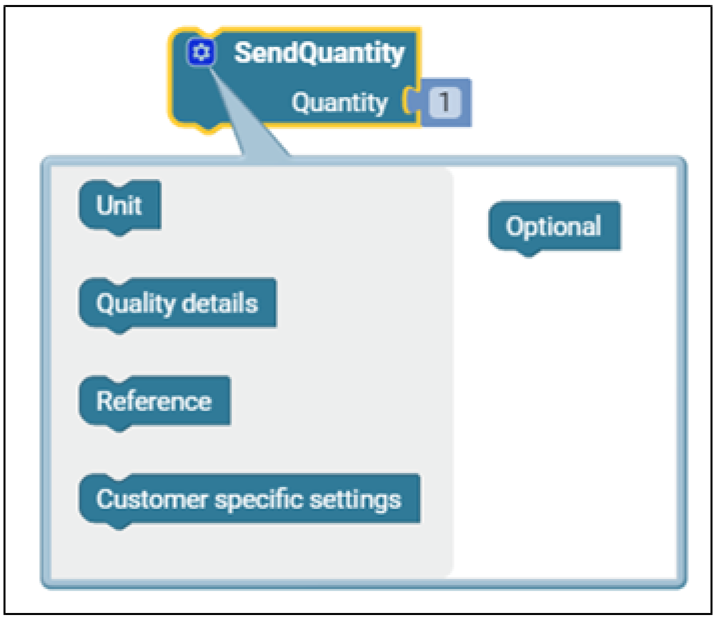


**Fig. 25**: **Example for SendImpulse**

In this case, once per second the number of seconds shall be increased by 1. Once the number of seconds reaches 30, the **`SendImpuls`** triggers a message, and the **`second`** variable is set to 0.

## 6.2 SendQuantity



### Usage

The **`SendQuantity`** block sends a defined quantity to third-party systems. The required quantity entered as number for **`Quantity`**. Optionally, **`Unit`**, **`Quality details`**, **`Reference`** and **`Customer specific settings`** can be included in the message.
**`Units`** must first be defined as variables.

### Input/Output

Only numbers can be used as input for **`Quantity`**.
All other input entries must be strings.

### Example

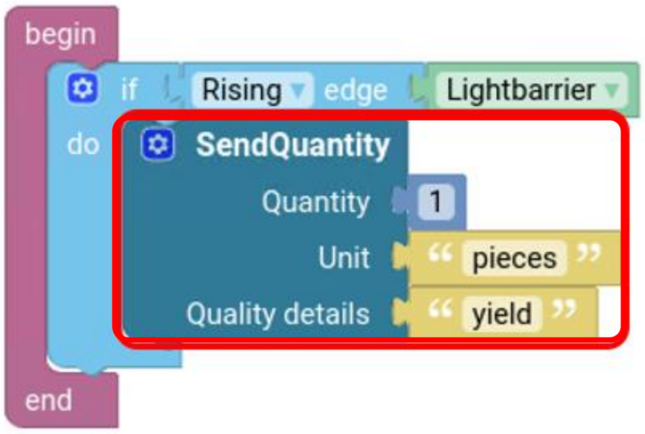
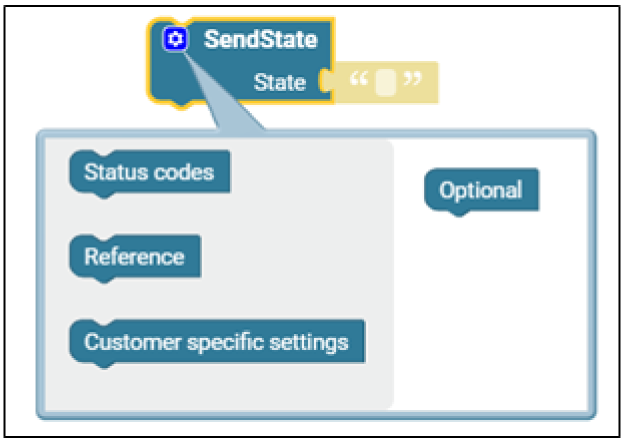


**Fig. 26: Example for SendQuantity**

The **`SendQuantity`** block shall send a message whenever a light barrier is activated. The message contains the information, that a **`quantity`** of 1 with the **`unit`** "pieces" has been produced, and that this quantity has been qualified (**`quality details`**) as **`yield`**.

## 6.3 SendState



### Usage

The **`SendState`** block sends the asset status as defined in the **`State`** field. The status values can be freely defined here.
Optionally, the list of **`Status codes`**, a **`Reference`** and **`Customer specific settings`** can be included in the message. The input comes from the entries in the Configuration Wizard (see Optional blocks for extension).

ⓘ In order to send **`Status codes`**, a list muss be created.
Refer to chapter 12 on list administration for more information.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

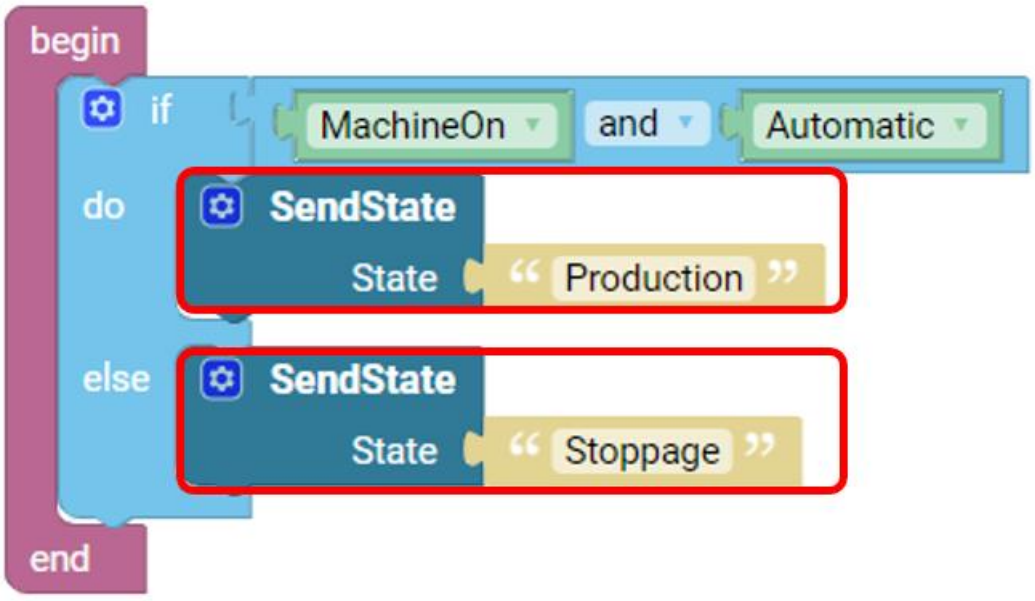
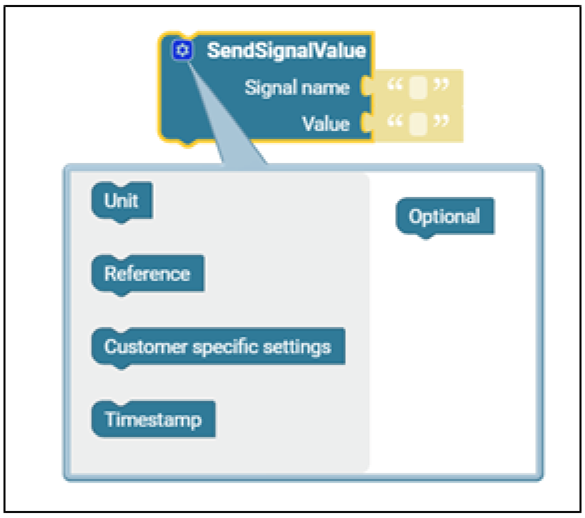


**Fig. 27: Example for SendState**

In this example, one of two statuses is transmitted. If the machine is switched on (**`MachineOn`**) and working in (**`Automatic`**) mode, the **`SendState`** blocks outputs the status **`Production`**. If not, the **`Stoppage`** status is output.

## 6.4 SendSignalValue



### Usage

The **`SendSignalValue`** block is used to send signal values.
The **`Signal name`** takes the name of the signal.
The corresponding value is entered in the **`Value`** field, the **`Unit`** contains the signal unit.
If one of the optional blocks further down is to be used, all other blocks above must be inserted first.
However, these blocks can remain empty, if not required.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

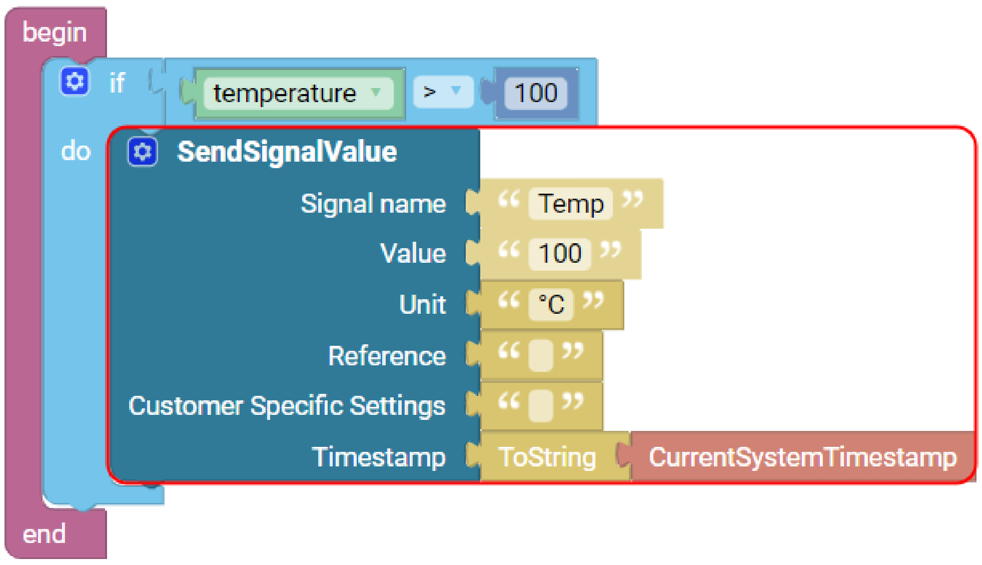
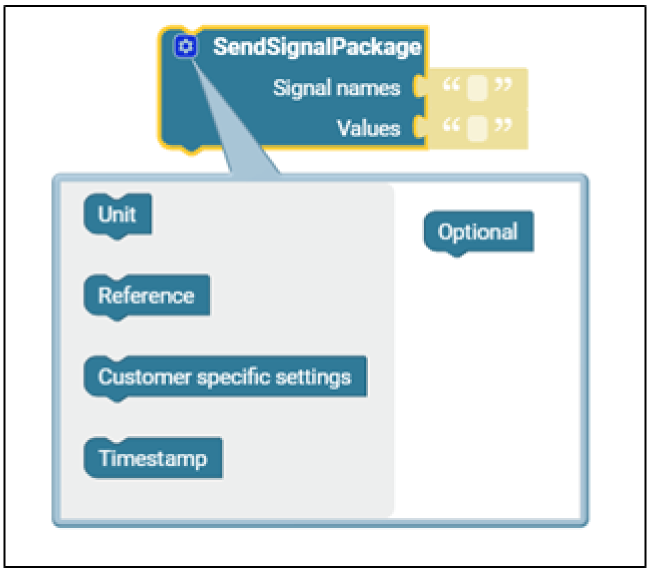


**Fig. 28: Example for SendSignalValue**

In this example, a warning message shall be sent whenever the temperature gets too high.
A signal value is sent if the **`temperature`** signal exceeds the value **`100`**.
The transmitted signal contains the signal name (**`Temp`**), the value (**`100`**), the unit of the value (**`°C`**) and the time when the limit was exceeded **`CurrentSystemTimestamp`**). Information for **`Reference`** or **`Customer specific settings`** is optional, these may remain empty.

## 6.5 SendSignalPackage



### Usage

**`SendSignalPackage`** sends lists of signals. The contents originate from previously created lists (see Create and manage lists (Lists)).
The list of names may be extended by additional signals and matching signal values.

Observe sequence:
The first entry in the list of signals must correspond to the first entry in the list of values.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

## Example

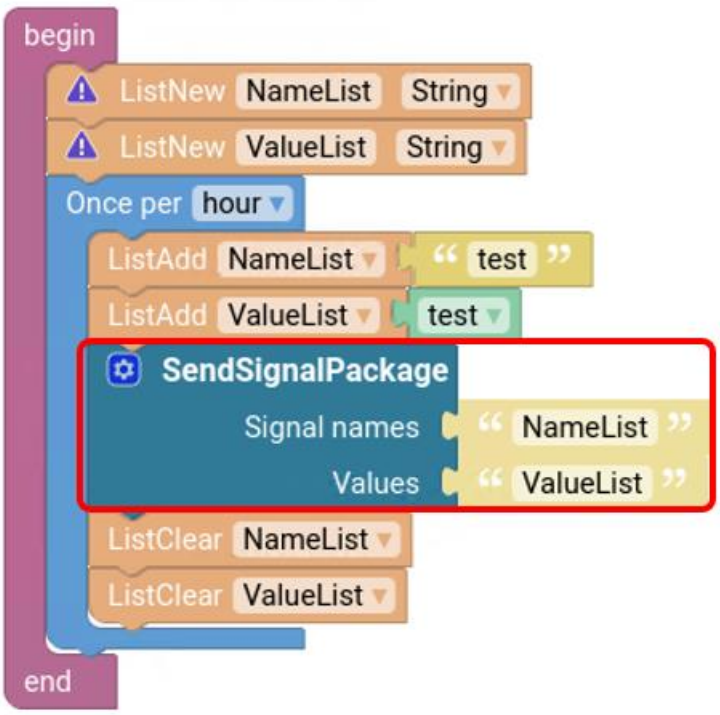
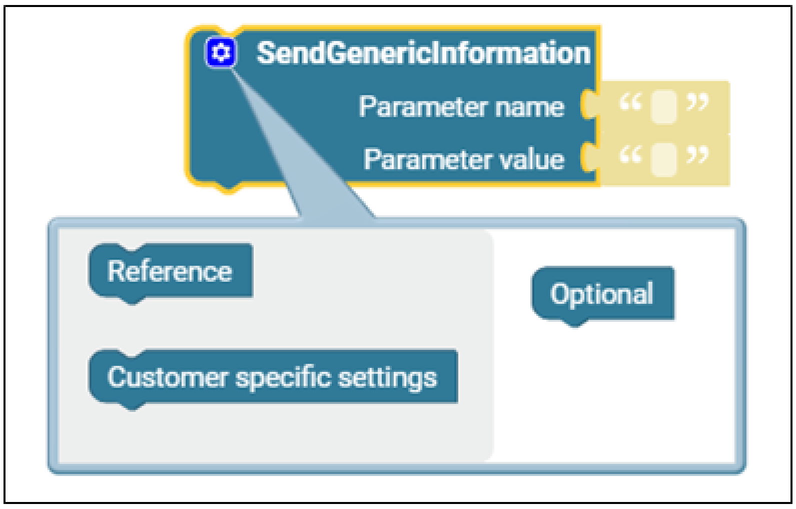


**Fig. 29: Example for SendSignalPackage**

First, two new lists were created: one list of names (**`NameList`**) and one list of values (**`ValueList`**).
In a **`Once per hour`** repeater, the signal called **`test`** is added to the **`NameList`**. The corresponding value (**`test`**) is written into the **`ValueList`**.
In this example, the **`SendSignalPackage`** block sends the lists once per hour.
After that, the lists are emptied.

## 6.6 SendGenericInformation



### Usage

The **`SendGenericInformation`** block sends an event with the current machine (asset) information.
The entries **`Parameter name`** and **`Parameter value`**
Additional **`Reference`** and **`Customer specific settings`**) can optionally be included.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

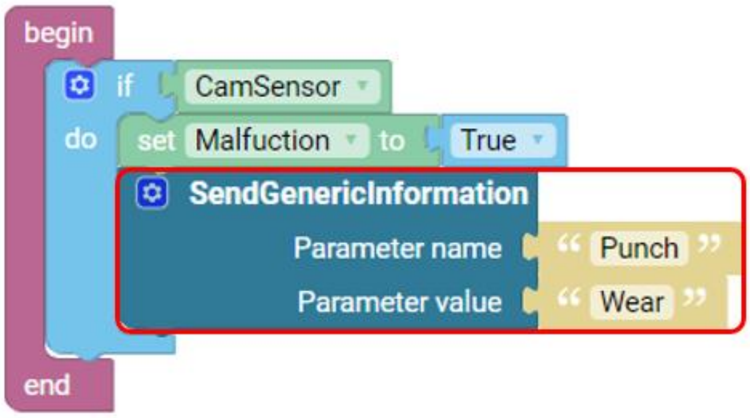
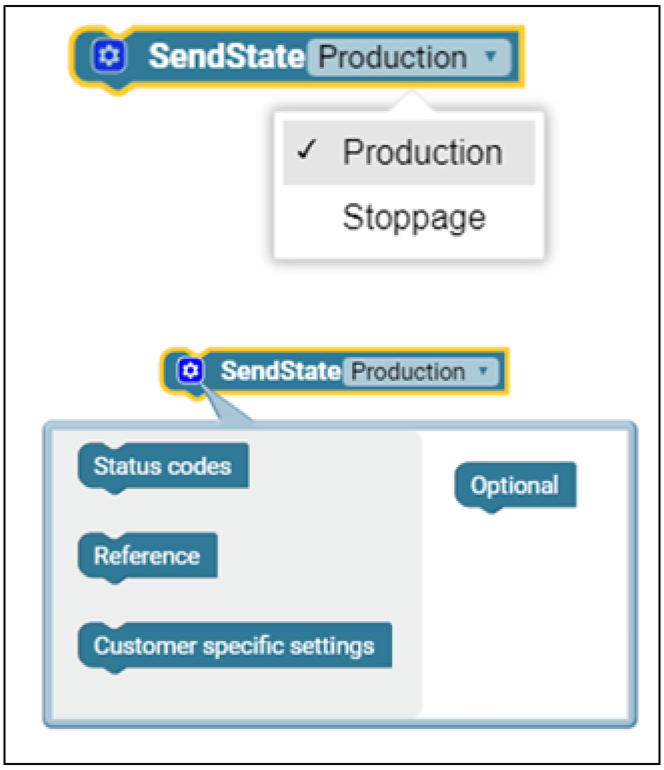


**Fig. 30: Example for SendSignalValue**

If the camera sensor is activated, the **`Malfunction`** status is set to **`True`** (1). This indicates a malfunction.
The **`SendGenericInformation`** block sends the error message of the punch as **`wear`**.

## 6.7 SendState [Selection]



### Usage

The **`SendState[Selection]`** block sends an asset status. There are two options: **`Production`** and **`Stoppage`**. Optionally, the list of **`Status codes`**, a **`Reference`** and **`Customer specific settings`** can be included in the message. The corresponding content has been defined in step 3 of the Configuration Wizard.

ⓘ In order to send **`Status codes`**, a list muss be created.
Refer to chapter 12 for more information.

### Input/Output

Input entries for **`SendState [Selection]`** must be strings. There are no restrictions to the output.

## Example

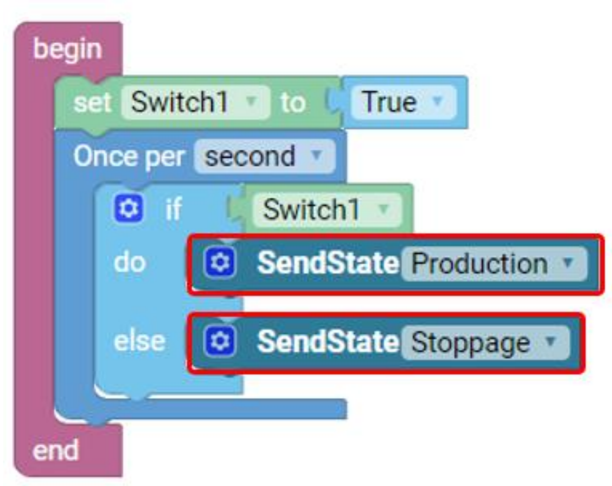
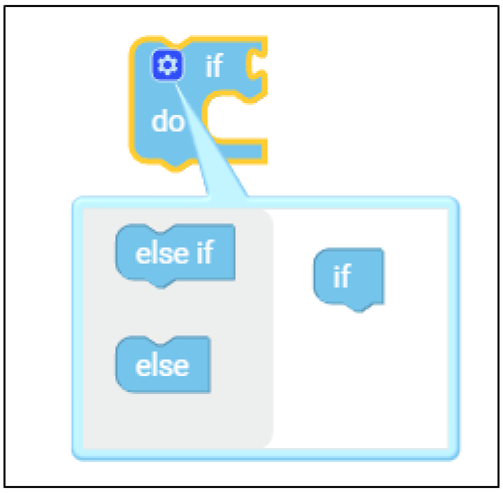


**Fig. 31: Example for SendState [Selection]**

At first, the Switch1 signals is set from **`False`** (0) to **`True`** (1).
The the repeater starts. If **`Switch1`** is switched, the production status **`Production`** is sent. If not, the **`Stoppage`** status is output.

# 7 Make logical connections (Logical)

## 7.1 if-do



### Usage

This block implements the if-do logic.
**If** represents a condition that must be fulfilled in order to process the subsequent command (**do**). If a condition is not fulfilled, **else** can be used to trigger a different command.
The **else if** block is optional. The command is processed whenever the related condition is regarded as **True**(1). The dark blue settings icon can be used to select additional parameters.

### Input/Output

The input type for **if**, **else if** and **else** is boolean.
There are no restrictions to the input for **do**.
There are no restrictions to the output.

### Example

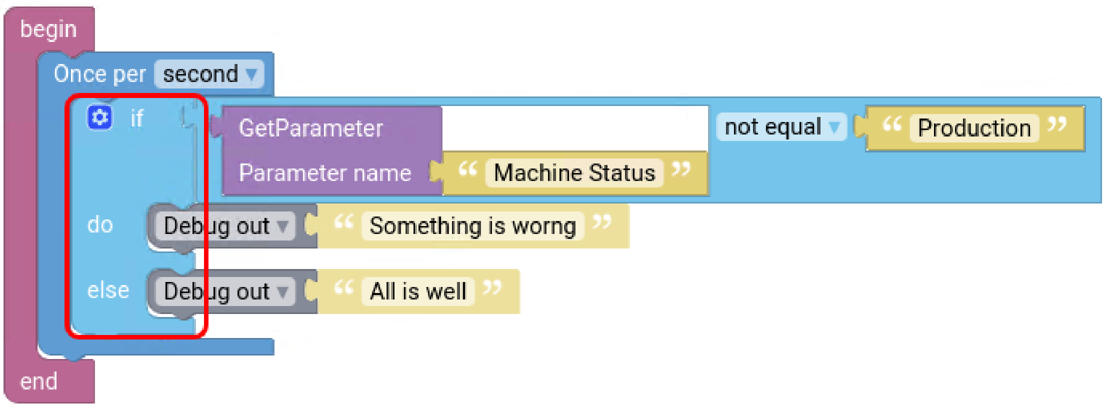
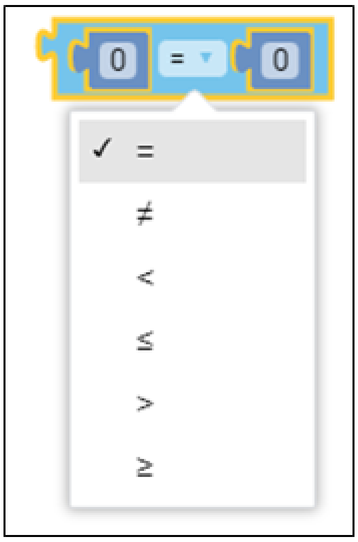


**Fig. 32**: **Example for the If-do-Block**

In this example, the machine status is requested once per second.

If the machine status is not **`Production`** the message **`Something is wrong`** shall be output. If it is, the message shall be **`All is well`**.

## 7.2 Mathematical comparison (=/≠/</>/≤/)



### Usage

Logical connectives like “**=**” link two variables of the *Number* data type.
The output is always a boolean value, i.e., True (1) or False (0).
The mathematical symbol can be replaced by other symbols.

The following tables contains their meanings:

| | |
|---|---|
| V1 = V2 | V1 equals V2 |
| V1 ≠ V2 | V1 unequal to V2 |
| V1 > V2 | V1 is greater than V2 |
| V1 ≥ V2 | V1 is greater than or equal to V2 |
| V1 < V2 | V1 is less than V2 |
| V1 ≤ V2 | V1 is less than or equal to V2 |

### Input/Output

Only numerical values (type Number) can be used as input. Output are Boolean values only.

### Example

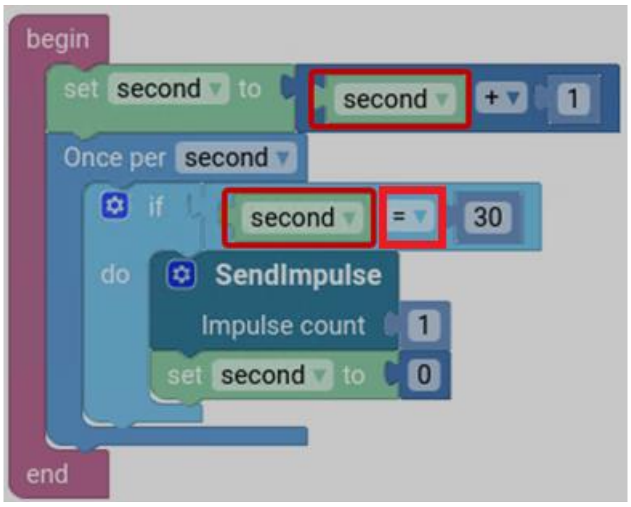
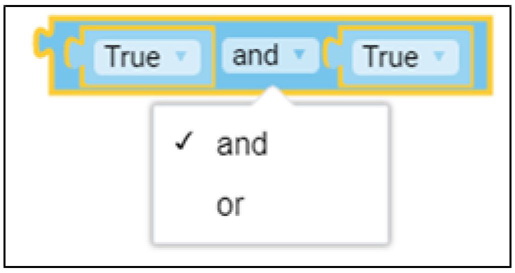


**Fig. 33**: **Example for the = connective**

In this example, 30 seconds shall be counted down. After that, the value is reset to 0.
Once per second, the **second** variable is increased by 1.
A check is performed each second to detect how many seconds have already passed. If the number of seconds equals (=) 30 an impulse is sent. This impulse resets the counter to the original value 0.

## 7.3 ‚And/or' (logical connective)



### Usage

The **and** connective is a basic connective (operator). If the states or statements before or after it apply, the result is **True**(1). The sequence of input states is not fixed. The output is always a boolean value, i.e., True (1) or False (0).
In the drop-down menu, the **or** connective can be selected. For this operator, only one of the statements must apply in order to regard the result as True (1).

### Input/Output

Input and output values can only be boolean values.

### Example

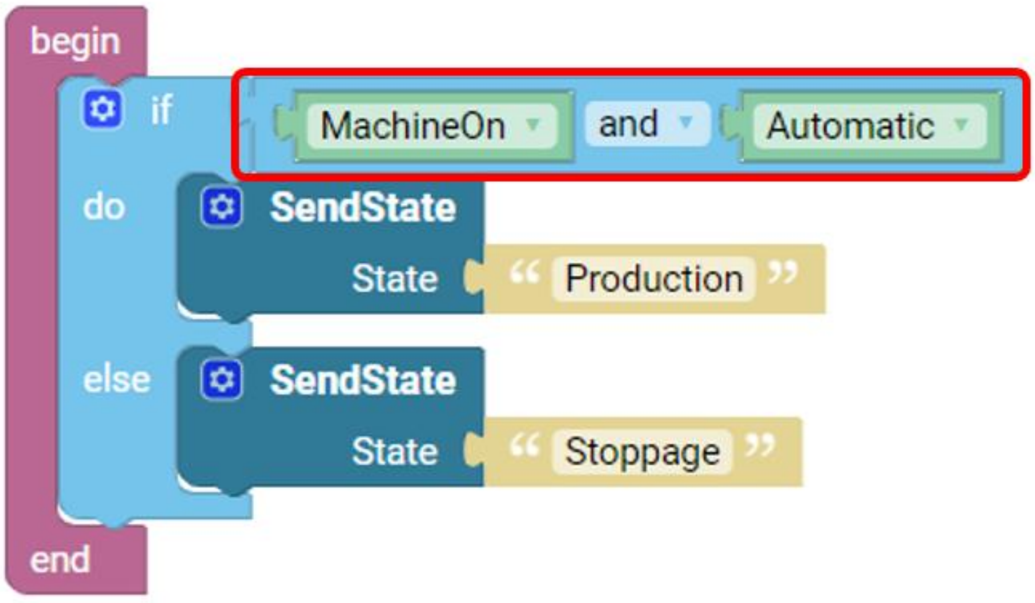
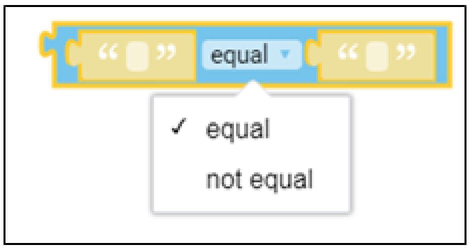


**Fig. 34**: **Example for the "and" connective**

In this example, the **`SendState`** block sends the status **`Production`** only if the machine is switched on (**`MachineOn`**) and (**and** connective) is running in automatic mode (**`Automatic`**).
If only one of the two prerequisites applies, status **`Stoppage`** is sent.

## 7.4 equal/not equal (logical connective)



### Usage

**`Equal`** is a basic connective (operator). If two states or statements are equal, the result (output) is **`True`**(1).
The sequence of input states is not fixed. The input is a string value, the output is boolean, i.e.,.**`True`** (1) or **`False`** (0).
In the drop-down menu, the opposite (**`not equal`**) can be selected.
The difference between the connectives „**`=`**" and **`equal`** is that **`equal`** is used to compare string values.

### Input/Output

Only strings are possible as input values. The output can only be boolean values.

### Example

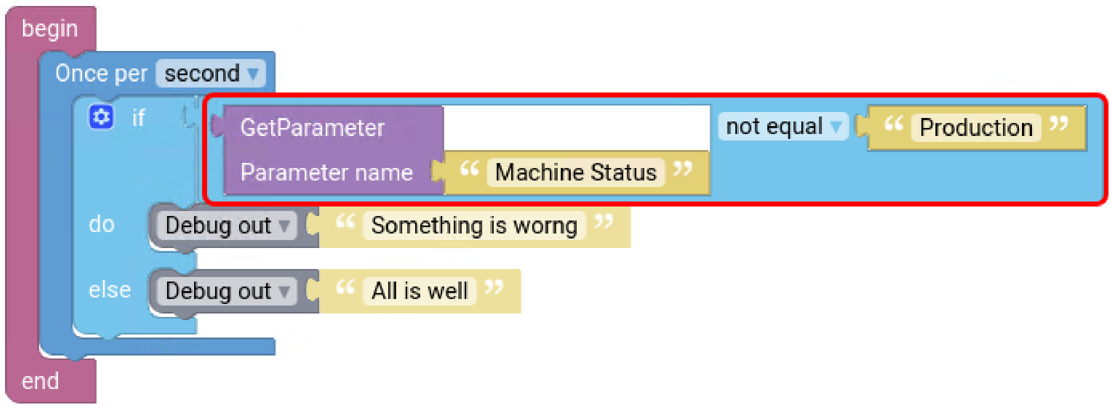
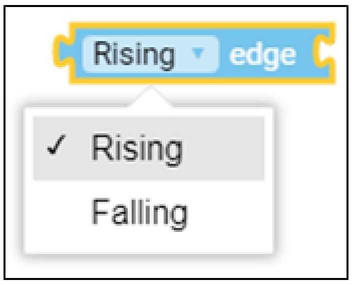


**Fig. 35: Example for "not equal"**

In this example, the machine status is requested once per second.
If the machine status does not match the **`Production`** status (**`not equal`**), the message **`Something is wrong`** shall be output. If it is, the message shall be **`All is well`**.

## 7.5 Rising/Falling edge (detect edges)



### Usage

This block indicates that a variable or signal has changed from true (1) to false (O) or vice versa.

**`Rising edge`**: At the beginning, the boolean value is false (0). **`Rising edge`** no checks whether the value is now true (1). This would mean, that the value has changed from 0 to 1. In this case, the corresponding command is processed.

**`Falling edge`**: At the beginning, the boolean value is true (1). **`Falling edge`** no checks whether the value is now false (0). This would mean that the value has changed from 1 to 0. In this case, the corresponding command is processed.

### Input/Output

Input and output values can only be boolean values.

### Example

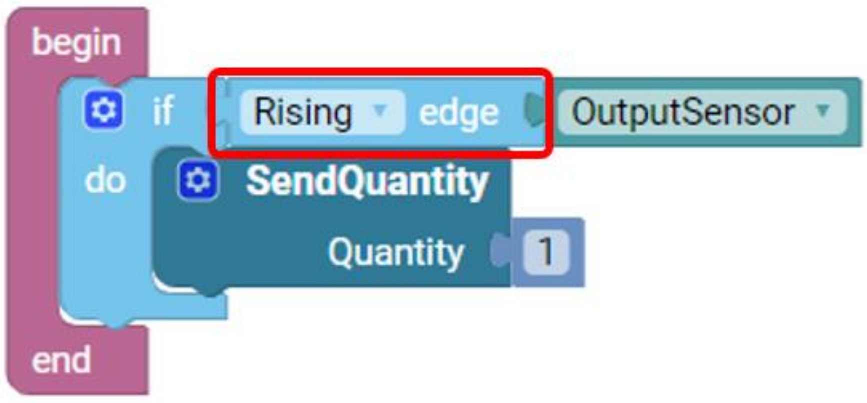
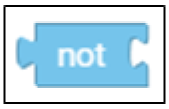


**Fig. 36**: **Example for rising edge**

In this example, an **`OutputSensor`** is used. Each time a piece is produced, the sensor triggers a signal change. This means, the boolean value of the signal changes from false (0) to true (1). Consequently, the **`Rising edge`** block is true (1). This triggers the subsequent command and the **`SendQuantity`** block reports one produced piece.

## 7.6 “not” statement (negation)



### Usage

The result of a **not** statement is true if the input value is false.
This means that the original state is the opposite of the output state.

### Input/Output

Input and output values can only be boolean values.

### Example

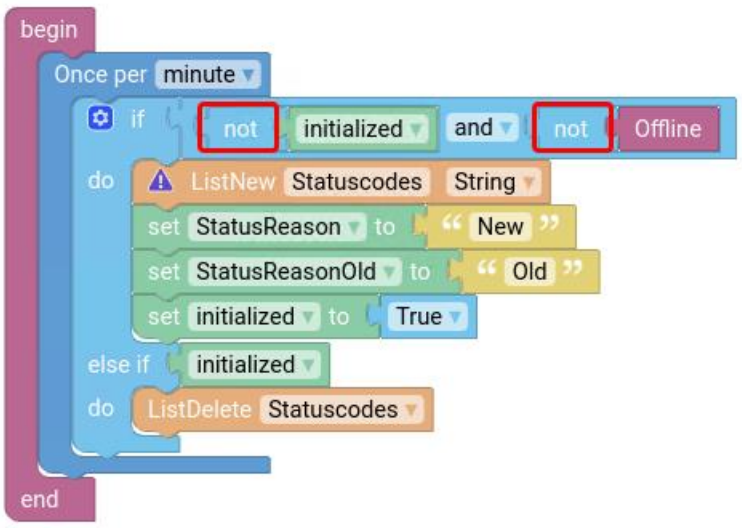
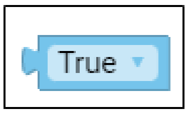

begin
Once per minute
if not initialized and not Offline
do ListNew Statuscodes String
set StatusReason to "New"
set StatusReasonOld to "Old"
set initialized to True
else if initialized
do ListDelete Statuscodes
end

**Fig. 37: Example for a "not" statement**

In this case, once a minute a check is performed to detect whether the machine is running for the first time.
The asset is considered running if the program is not processed (`not initialized`) and the asset is not offline (`not offline`). Therefore, lists are created with current and previous reasons for a status. The creation of the lists triggers the execution of the program (`initialized`). This switches the variable to `True` (1).
After that, the list of status reasons is deleted.

## 7.7 True/False



### Usage

This block is placed at the end and used to define whether the result is `True` (1) or `False` (0). To do so, `True` or `False` can be selected from the drop-down menu.

### Input/Output

There are no restrictions to the input. The output can only be boolean values.

## Example

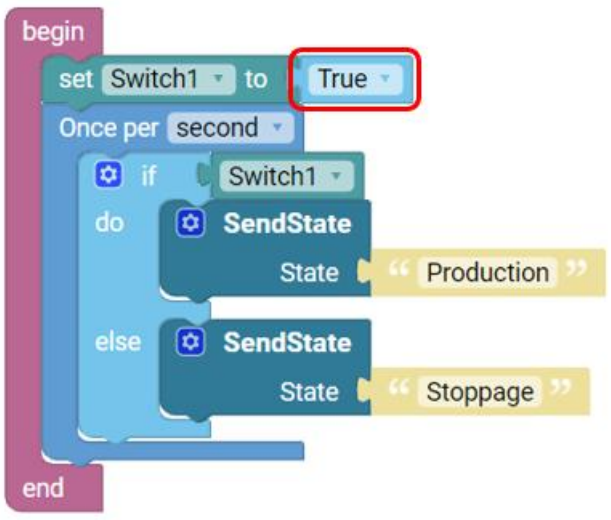


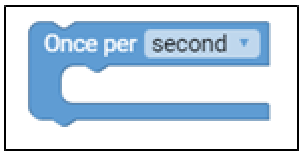
**Fig. 38: Example for “true”**

At first, **`Switch1`** is activated, which triggers the signal and therefore changes to **`True`** (1). After that, a repeater is called once per second to check whether **`Switch1`** was switched. If yes, the production status is set to **`Production`**. If not, status **`Stoppage`** is output.

# 8 Repeaters

In many cases, actions are repeated in regular intervals. Repeaters trigger actions in defined intervals.

## 8.1 Once per



### Usage

Repeaters are used to repeat an action at regular intervals. The required interval can be select in the drop-down menu.

```
Once per
      second
      minute
      hour
      day
```

### Example

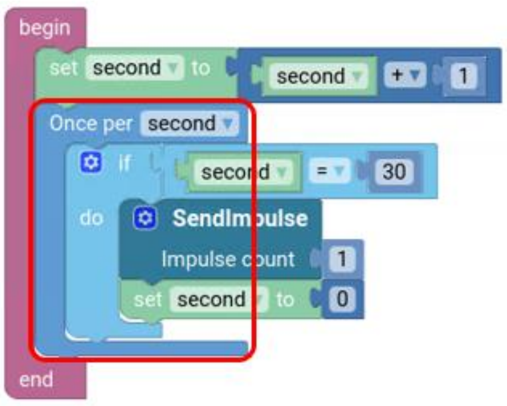


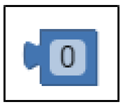
**Fig. 39**: **Example for once per**

In this example, 30 seconds shall be counted down. After that, the value shall be reset to 0.
Once per second, the **second** variable is increased by 1. A check is performed each second (**Once per second**) to detect how many seconds have already passed. If the number of seconds equals (=) 30 an impulse is sent. The impulse resets the counter for the **second** variable to the original value 0.

# 9 Maths operations (Arithmetic)

These blocks implement calculation functions such as adding, subtracting or multiplying values, and they convert data formats.

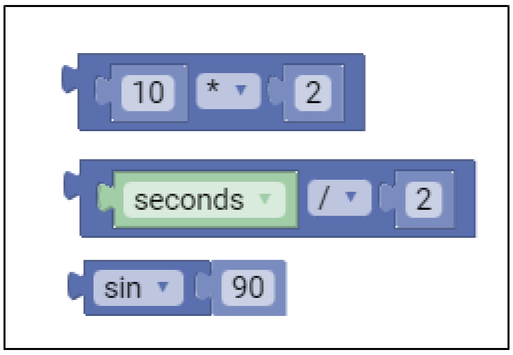
## 9.1 Number field



### Usage

In this block, a numerical value is inserted to connect it to a task.
Input and output values can only be numbers.

## 9.2 Mathematical operation



### Usage

The block can be used for various math operations like addition, subtraction, multiplication, division, exponentiation or calculating the sine value. Besides numbers, variables can also be included for calculation.

### Input/Output

Input and output values can only be numbers.

### Example

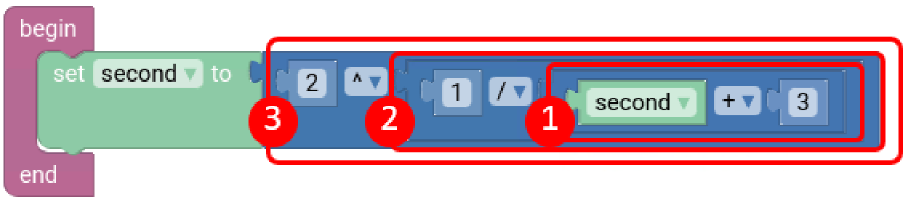


**Fig. 40**: **Example for maths operations**

A nested calculation indicates the factor. For understanding the calculation method it is important to follow the“ from inside out” calculation rule. This principle is used to place the parenthesis and defines the calculation order.
In this example, three is added to the **second** variable first (1). The result is used as denominator of the fraction(2). This result is then used as exponent to two in the last math operation (3).

## 9.3 ToNumber

### Usage

The **ToNumber**block changes the data type from string to a numerical value (number).  The string to be converted must consist of numbers only.

### Input/Output

The input must be a numerical value of data type string. The output can only be numbers.

### Example

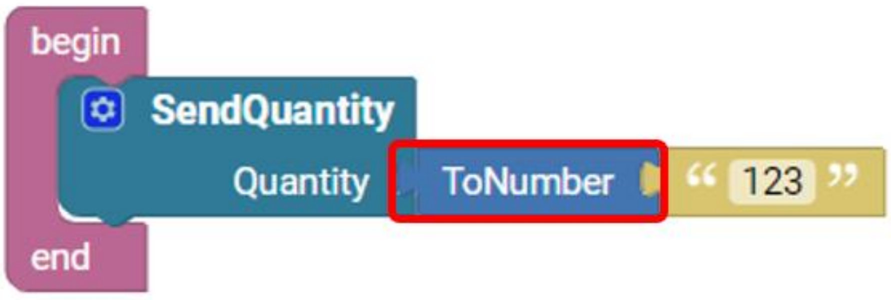


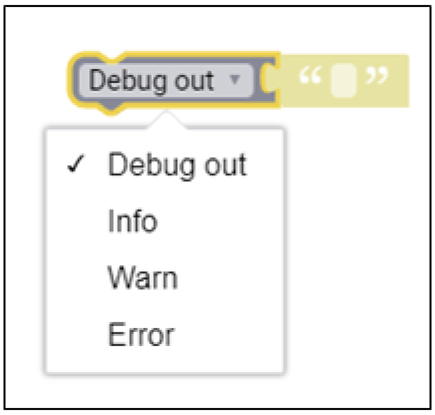
**Fig. 41: Example for ToNumber**

The **SendQuantity** block shall report a quantity. However, the input is a string value in our case. Although it only consists of numbers, the string is not a valid input data type for the **SendQuanity** block. Therefore, the **ToNumber** block is used to convert the string data type into a number. Only this way the **SendQuantity** can be processed.

# 10 Log values (Logging)

The blocks in this category can be used to log specific values and make them available for analysis. Different warning levels are applied for this.

## 10.1Debug out



### Usage

Raw signals and variables are logged to get the desired values.
Different types of log entries can be selected.

`Debug out`: Information that can be helpful during issue diagnosis
`Info`: General log for all types of activities
`Warn`: Issues or malfunctions that do not prevent processing
`Error`: Issue that stops/prevents several functions

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

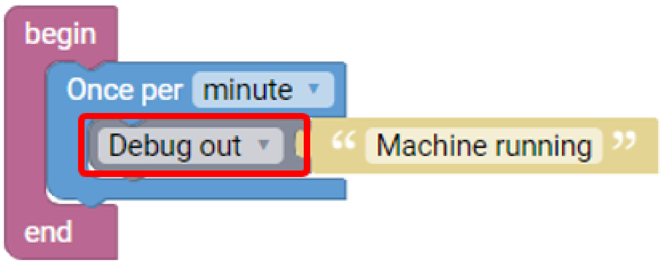


**Fig. 42: Example for Debug out**

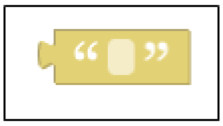
In this example, the `Debug out` block is used to write the string `Machine running` to the log file once per minute.

# 11 Create and process texts (Text)

The graphical/modular composition also needs words and sentences to make the values understandable. This category can be used to create texts and use additional processing functions.

In graphical composition, text is regarded as a string. As with a string, text can consist of letters, numbers and characters.

## 11.1String



### Usage

Using these blocks, strings can be added by typing them in the quotes.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

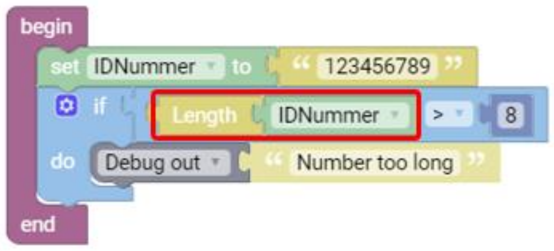
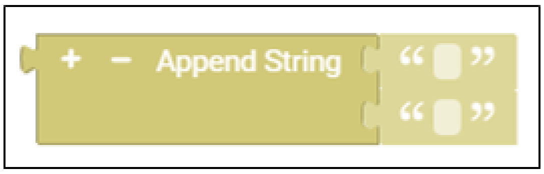


**Fig. 43: Example for String**

The **`set IDNumber to`** block defines the ID number of an asset with the string “123456789”. After that, the **`if-do`** block checks whether the Id number has more than 8 characters. If yes, a message is written to the log file. This message is entered in the string. In this case the message is “Number too long”.

## 11.2Append String



### Usage

As an extension to the simple string, **`Append String`** puts several strings together. Strings are added or deleted by clicking the plus or the minus sign.

### Input/Output

Input and output values can only be strings.

### Example

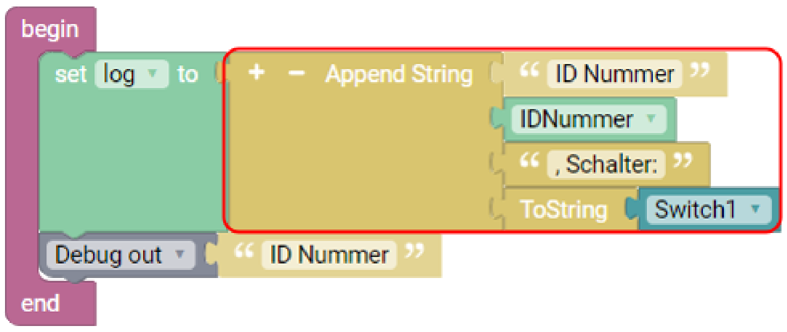


**Fig. 44: Example for Append String**

This example is about logging the ID number and the switch status. The **`set log to`** block makes it more readable. The **`Append string`** block is read from top to bottom. Therefore, first the text "ID number" is displayed, then the value of the variable **`IDNumber`** is added. Then the text “, Schalter:” is displayed and the signal of the switch **`Switch1`** is added. At the end, the entire string is written to the log file.

## 11.3ToString

### Usage

**`ToString`** is used to convert numbers, or variables representing numbers, into a string.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

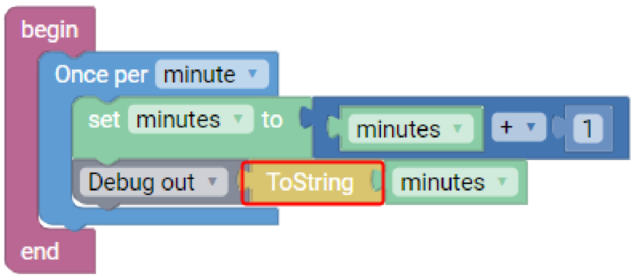
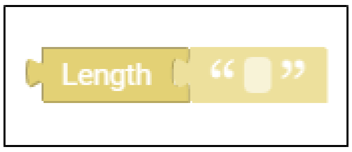


**Fig. 45: Example for ToString**

The goal is to output the number of minutes. **`Once per minute`** the variable **`minutes`** is increased by one. The **`Debug out`** block is then used to write the new value to the log file. However, **`Debug out`** can only have strings as input values. Therefore, **`ToString`** converts the **`minutes`** variable into a text.

## 11.4Length



### Usage

**`Length`** counts the number of characters in a string. The desired string is entered in the quotation marks. It is also possible to attach a variable. The counted number of string characters is output as the result. The result is a number. Counting starts with 1.

### Input/Output

Only strings are possible as input values. The output can only be numbers.

### Example

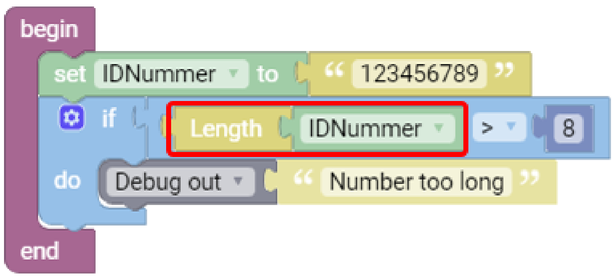
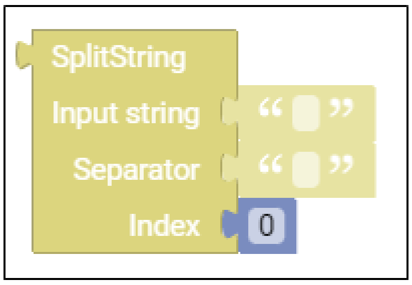


**Fig. 46: Example for Length**

As an example, the length of the order number is to be counted to make sure it does not exceed a defined threshold of eight characters.
If the **`IDNummer`** is more than 8 characters long (**`Length`** > 8), the **`Debug out`**block should write the message “Number too long” to the log file.

## 11.5 SplitString



### Usage

In the **SplitString** block, a value from a self-defined selection of categories can be output. **Input string** is used to define the different categories.
They are separated by a predefined character. This character is defined under **Separator**, typically a comma or an underscore is used as separator.
The **index** indicates which of the **Input string** entries is to be selected. Only one value can be output. The **Input strings** are counted from left to right. Counting starts with 0.

### Input/Output

Only strings can be used as input and output for **Input string** and **Separator**.
For **Index**, input and output values can only be numbers.

### Example

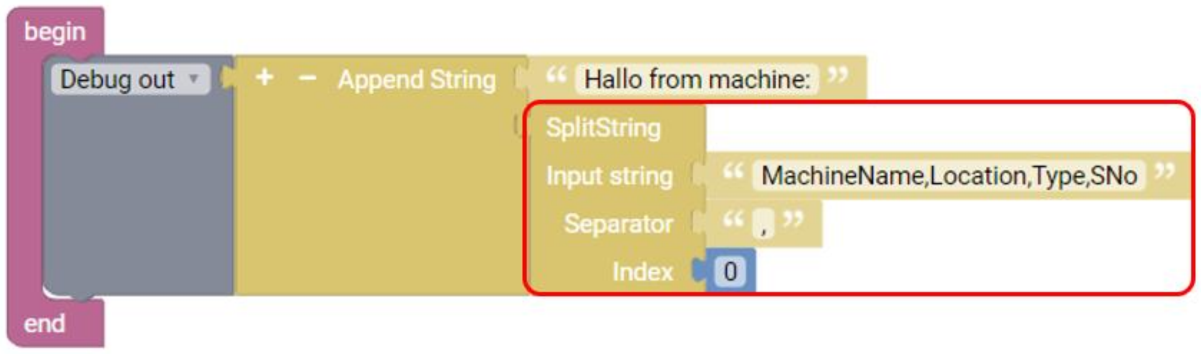
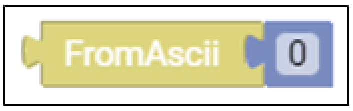


**Fig. 47: Example for SplitString**

In this example, the machine name should be output. It starts with the text “Hello from machine:”. The possible categories are listed under **Input string** and are separated by commas (**Separator**). The **Index** is specified with 0. As a consequence, the MachineName is output. If the index were “2”, the type (“Type”) would be output.

## 11.6FromAscii



### Usage

The **`FromAscii`** block refers to a specified table of values with instructions and characters. The block accesses a value from this table. The number indicates which value of the ASCII table is to be selected.

### Input/Output

Only numbers can be used as input. The output can only be string values.

i The ASCII table can be found in chapter 17.2 ASCII table, page 79.

### Example

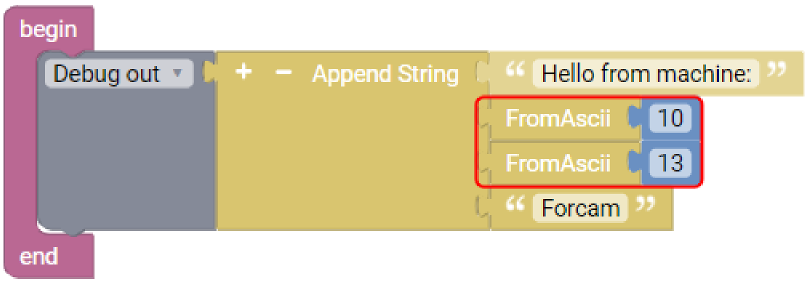
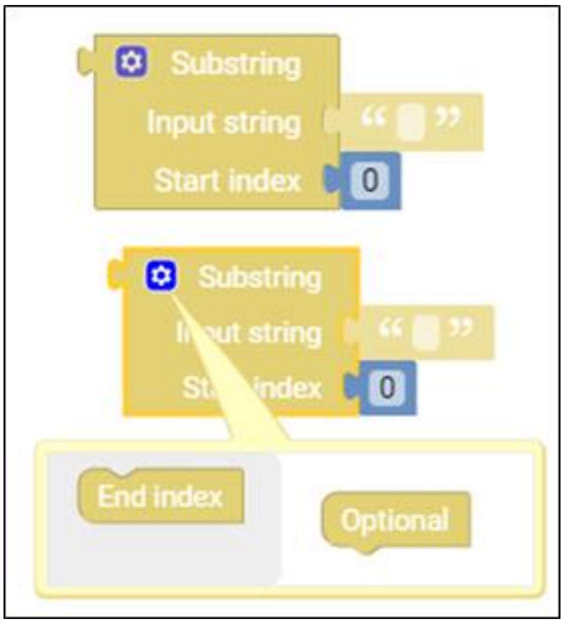


**Fig. 48: Example for FromAscii**

In this example, the text "Hello from machine:" shall be output followed by a paragraph mark and the text "Forcam".
The **`Append String`** block lists strings one after another. After the first text string "Hello from machine:" is inserted, the block **`FromAscii`** reads and processes the tenth command from the ASCII table. This is LF for line feed (new line). Then a second **`FromAscii`** block fetches command 13 from the ASCII table. This is CR, i.e., carriage return (same as pressing the Enter key). This places the cursor at the beginning of a line.
The result looks like this:

Hello from machine:
Forcam

## 11.7Substring



### Usage

The **substring** block outputs only a part of a string. The entire string entered under **Input string**. **Start index** and **End index** are entered below as numbers. As typical for index handling, characters are counted starting from 0. The **End Index** is excluded.

### Input/Output

Input and output for **Input string** are strings.
Only numbers are possible as input for **Start index** and **End index**. The output can only be string values.

### Example

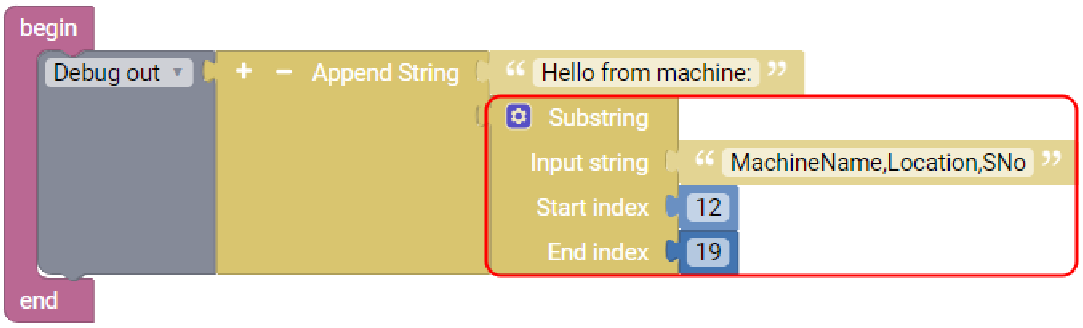


**Fig. 49: Example for Substring**

In this example, the location of the machine shall be output.
The **Append String** block first sets the text "Hello from machine:". **Input string** provides a list of asset properties. **Start index** specifies that the output starts at character 12. **End index** indicates that the output ends and includes character 19.
Because counting starts with 0 from the left, the **Location** property is output.

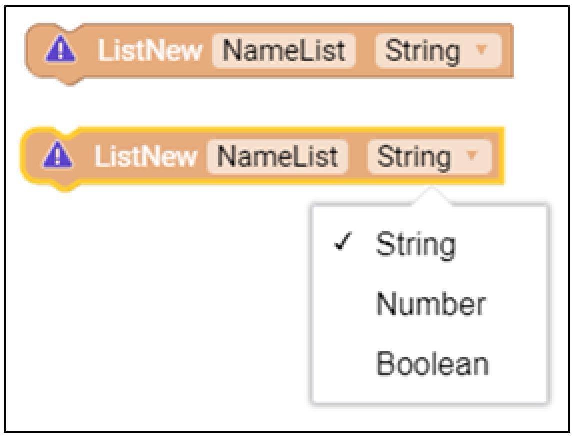
# 12 Create and manage lists (Lists)

Usually, a list is used to collect different production states. The blocks of this category create, fill, empty and delete lists.

ⓘ A list must be created first.
Only then more blocks are available for use with the list.

⚠ Always empty a list after using it (see function ListClear).

## 12.1ListNew



### Usage

The **`ListNew`** block creates a new list. The name of the list can be entered in the first field. The type of list input (string, number or boolean values) is selected from the drop-down menu.

### Input/Output

Restrictions for the input are made via the selection.

### Example

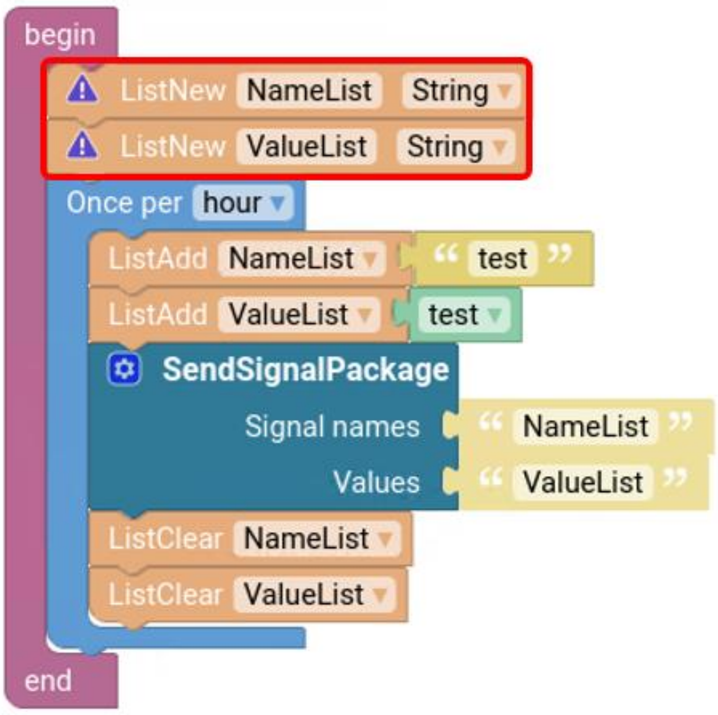
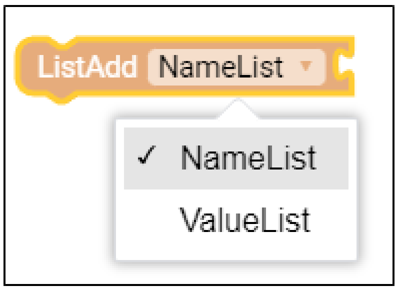


**Fig. 50: Example for ListNew**

First, the **`ListNew`** blocks create two new lists, a list of names and a list of values. The exclamation marks remind you to empty or delete the list at the end. A repeater adds the signal name **`test`** to the **`NameList`** list. The corresponding value is inserted in the **`ValueList`**.
Then the **`SendSignalPackage`** block sends both lists. The **`ListClear`** blocks clear the contents of the assigned list.

## 12.2ListAdd



### Usage

The **`ListAdd`** block adds values to a list. As a prerequisite, the list must already have been created using the **`ListNew`** block. The desired list is selected via the drop-down menu.

### Input/Output

The input for the block is always a previously created list. This list is selected from the drop-down menu. There are no restrictions to the output.

### Example

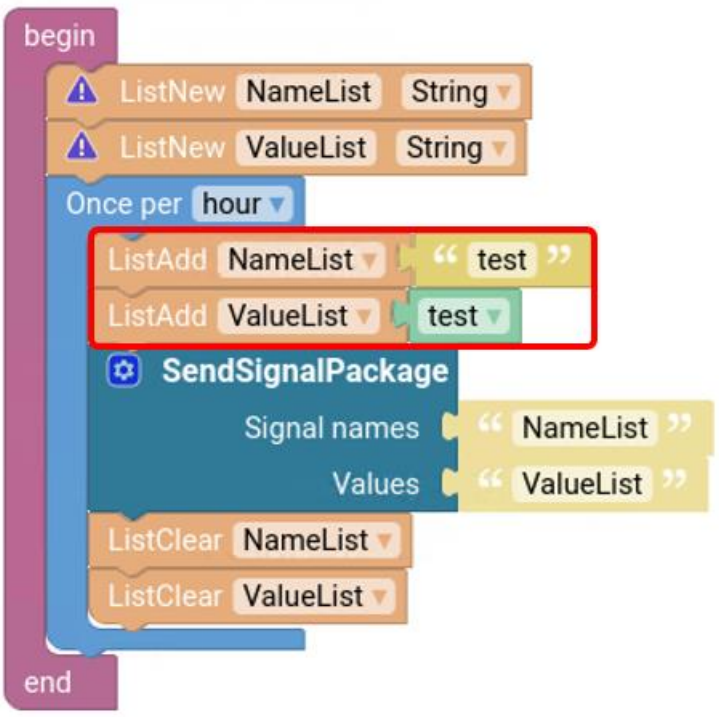
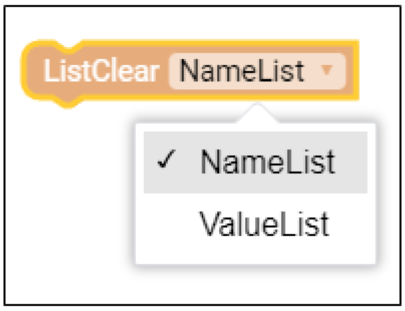


**Fig. 51: Example for ListAdd**

First, two new lists are created: one list of names and one list of values. One of the **`ListAdd`** blocks adds the signal name test to the **`NameList`** once per hour; the other block inserts the corresponding value into the **`ValueList`**. Then the **`SendSignalPackage`** block sends both lists. The **`ListClear`** blocks clear the contents of the assigned list.

## 12.3 ListClear



### Usage

**`ListClear`** deletes the contents of a list.

ⓘ It is important to run the **`ListClear`** command regularly after creating a new list to keep free memory.

⚠ **`ListClear`** deletes only the contents of a list.
**`ListDelete`** completely deletes a previously created list.

### Input/Output

The input for the block is always a previously created list. This list is selected from the drop-down menu. There are no restrictions to the output.

### Example

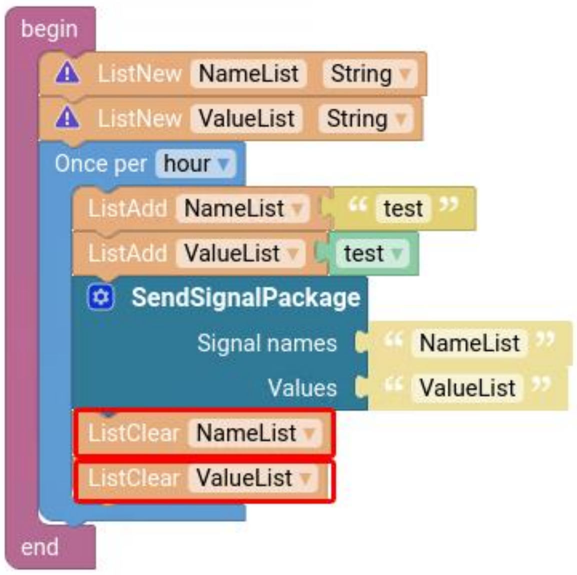
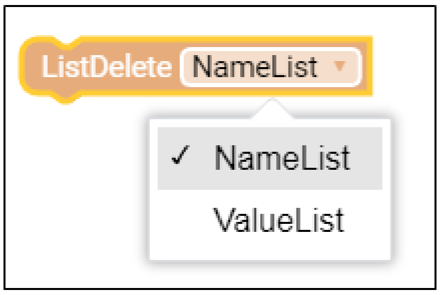


**Fig. 52: Example for ListClear**

First, two new lists are created: one list of names and one list of values. One of the **`ListAdd`** blocks adds the signal name **`test`** to the **`NameList`** once per hour; the other block inserts the corresponding value into the **`ValueList`**. Then the **`SendSignalPackage`** block sends both lists. The **`ListClear`** blocks clear the contents of the assigned list.

## 12.4ListDelete



### Usage

The **`ListDelete`** block deletes an existing list. The drop-down menu is used to select the list to be deleted.

⚠ **`ListDelete`** completely deletes a previously created list.
**`ListClear`** deletes only the contents of a list.

### Input/Output

The input for the block is always a previously created list. This list is selected from the drop-down menu. There are no restrictions to the output.

### Example

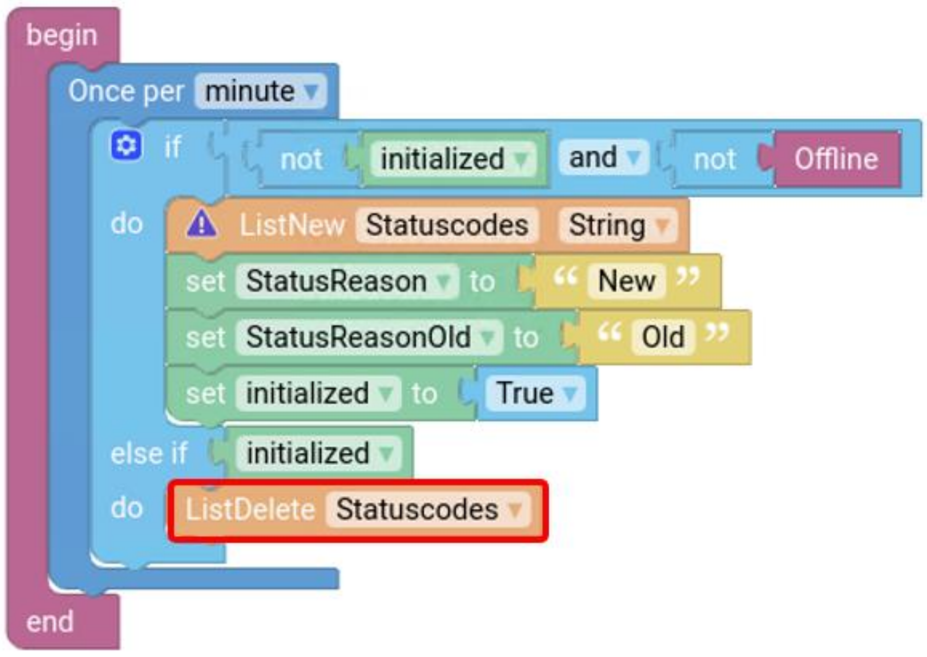
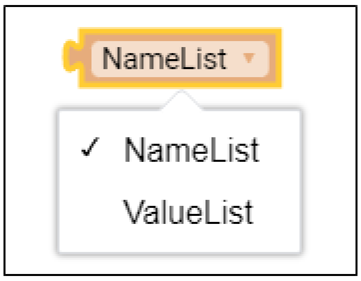


**Fig. 53: Example for ListDelete**

In this case, once a minute a check is performed to detect whether the machine is running for the first time.
The asset is considered running if the program is not processed (**`not initialized`**) and the asset is not offline (**`not offline`**). Therefore, lists are created with current and previous reasons for a status.
The creation of the lists triggers the execution of the program (**`initialized`**). This switches the variable to **`True`** (1).
The list with status reasons is deleted by the **`ListDelete`** block.

## 12.5 [List] - Insert list



### Usage

The block inserts a list into the structure. The (already created) list is selected in the drop-down menu.

## Input/Output

The input for the block is always a previously created list. This list is selected from the drop-down menu. There are no restrictions to the output.

## Example

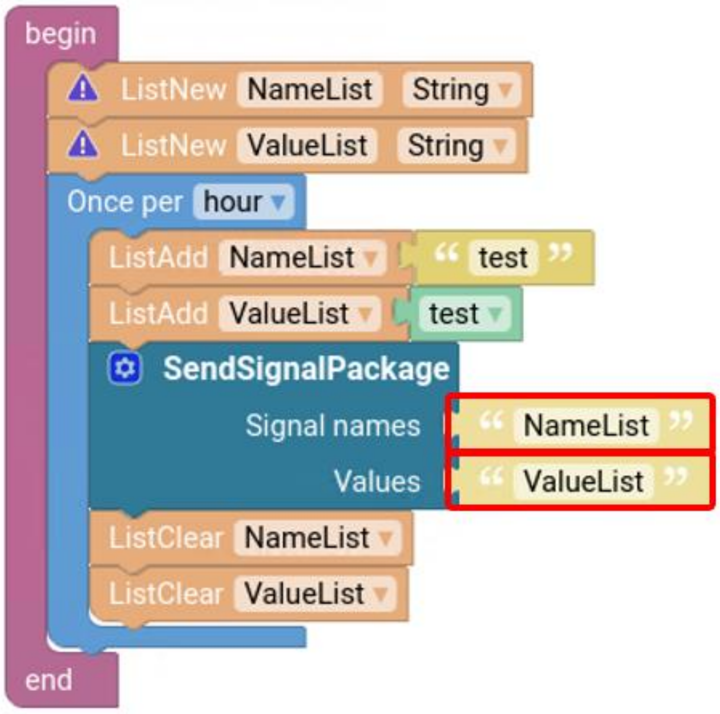


**Fig. 54: Example for [List]**

In this example, two lists of temperature values shall be created, filled with values, sent and, at the end, emptied again.
After the lists are created and the temperature values inserted once an hour, they are sent with using the `SendSignalPackage` block. The signal name and the corresponding signal values are taken from the name list and the value list.

# 13 Managing times (Date and time)

Date and time must be defined in order to trigger an action at a specific point in time. The current time of an event and pauses are also stored.

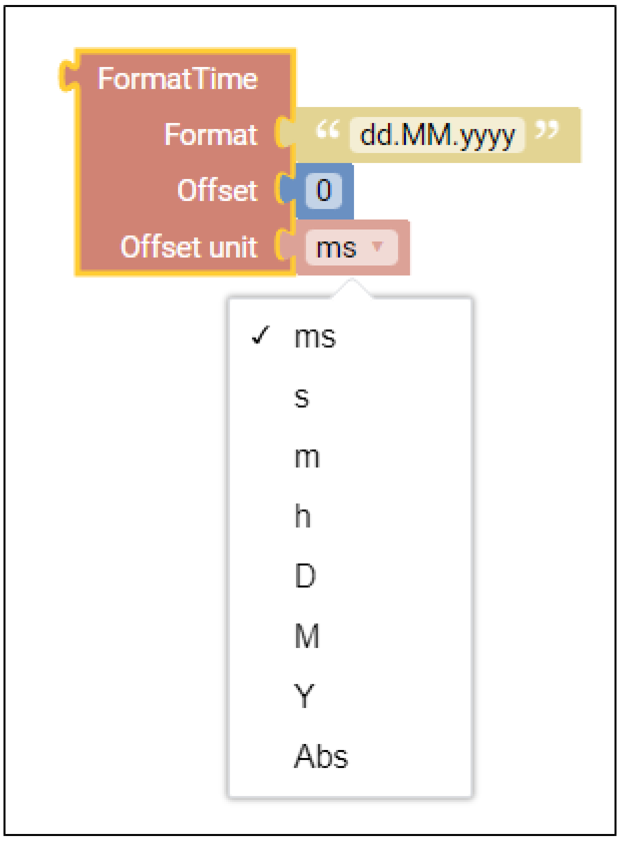
This function category contains all actions related to time or date settings. UTC time is used throughout the category.

The following table lists the abbreviations used in the graphical composition for the various time units.

| Letter | Date or time | Example |
|---|---|---|
| **G** | Calendar system era | AD |
| **Y** | Year | 2018 (yyy), 18 (yy) |
| **M** | Month of the year | July (MMMM), Jul (MMM), 07 (MM) |
| **w** | Week of the year | 16 |
| **W** | Week of a month | 3 |
| **D** | Day in a year | 266 |
| **d** | Day in a month | 4 |
| **F** | Week in a month | 4 |
| **E** | Day of the week | Tuesday, Tue |
| **u** | Number of the weekday, where 1 stands for Monday, 2 for Tuesday, etc | 2 |
| **a** | AM or PM | AM |
| **h** | Hour of the day with am/pm (1-12) | 12 |
| **H** | Hour of the day (0-23) | 12 |
| **k** | Hour of the day (1-24) | 23 |
| **K** | Hour of the day with am/pm (0-11) | 2 |
| **m** | Minute per hour | 59 |
| **s** | Second per minute | 35 |
| **S** | Millisecond per minute | 978 |

| | | |
|---|---|---|
| **z** | Time zone | GMT-08:00 |
| **Z** | Time zone offset in hours (RFC pattern) | -0800 |
| **X** | Time zone offset in ISO format | -08;-08:00 |
| **E, dd MMM yyyy HH:mm:ss** | Example | Tue, 02 Jan 2023 11:22:35 |

## 13.1FormatTime



### Usage

The **FormatTime** block creates the desired time unit of the current time/a date based on the current time stamp.
The format specifies the unit of the **Offset**, e.g., dd.MM.yyyy or MM.dd.yyyy.
The current time is indicated as an **Offset** of 0 (zero).
The **Offset unit** determines the counting unit. Possible counting units are milliseconds, seconds, minutes, hours, days, months or years. For example, the result of an **Offset** of 10 and milliseconds (ms) as the unit would be the current time plus 10 milliseconds.

ⓘ **Abs** is used to convert Unix time stamps (e.g., time stamps that are received directly from the asset). In this case, the reference time (offset = 0) for conversion is not the current time but January 1st, 1970, 00:00 o'clock. If **Abs** is selected, the offset value is therefore the time difference (in ms) to this (reference) date. This value is converted to the desired format.

### Input/Output

The input for **`Format`** is a string value. The output can only be string values.
The input for **`Offset`** is a number. The output can only be string values.
The input for **`Offset unit`** is a drop-down menu. The output can only be string values.

### Example

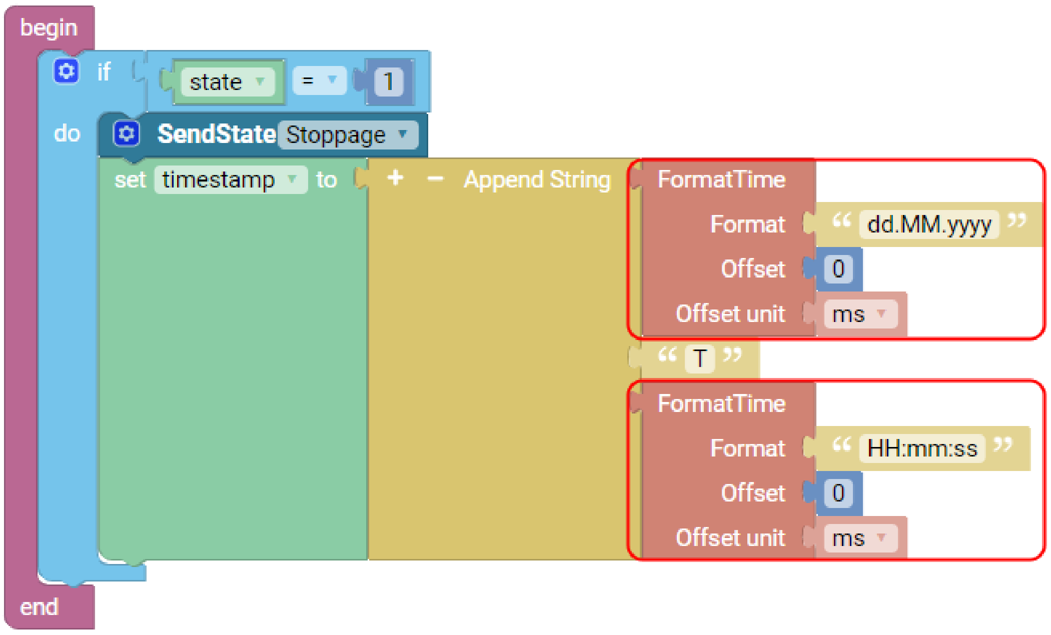
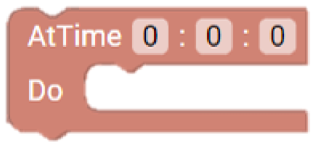


**Fig. 55: Graphical example for FormatTime**

In this example, a time stamp shall be to be recorded for each stoppage.
If the status is one (1), the **`SendState`** block shall send the status **`Stoppage`**. At the same time, the following string shall be written to the timestamp variable: First the date in the order day.month.year, then the text string **`T`** for time, then the time in the order hour:minute:second.

## 13.2AtTime Do



### Usage

The **`AtTime Do`** block executes a specific action at a defined time.
The time is specified in the following format: HH : mm: ss. The number range of the hours is from 0 to 23, that of minutes and seconds from 0 to 59.

### Input/Output

Only numbers can be used as input.

### Example

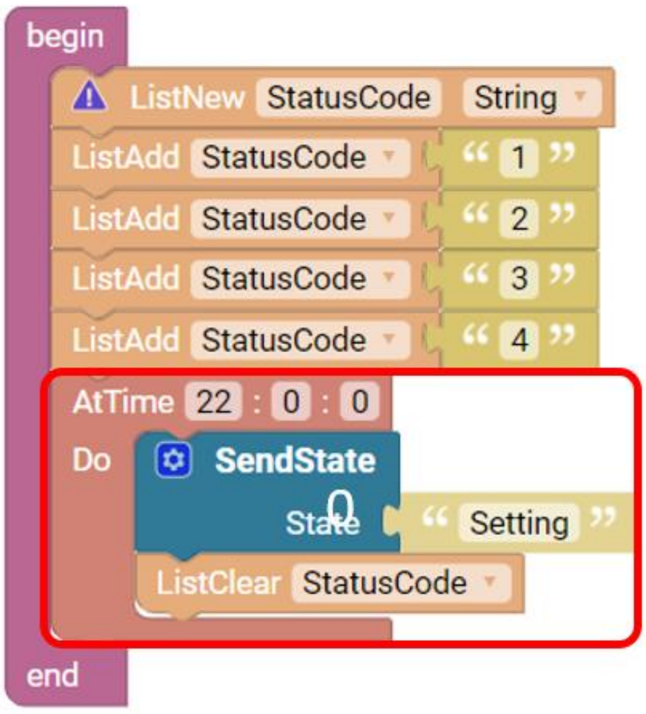
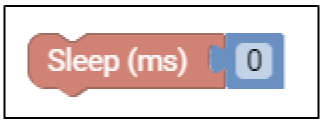


**Fig. 56: Example for AtTime Do**

This example shows, a status shall always be sent at exactly the same time. To do so, the **`ListNew`** block is used to create a **`StatusCode`** list. This list contains strings. In the **`AtTime Do`** block, the time 22:0:0 is defined. At this time, the **`SendState`** action will be executed.
The list is then cleared again.

## 13.3Sleep



### Usage

The **`Sleep`** block waits for a certain period of time. The numeric value indicates the period of time (in milliseconds) for which there shall be no action performed. After that, the next block is executed. This is especially helpful for actions that take longer to execute. This way it will not be "overtaken" by subsequent tasks.

### Input/Output

Only numbers can be used as input. There are no restrictions to the output.

### Example

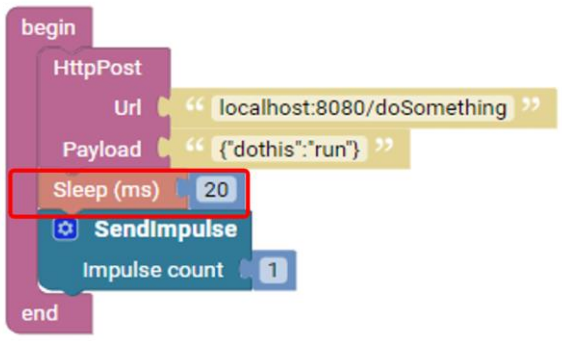
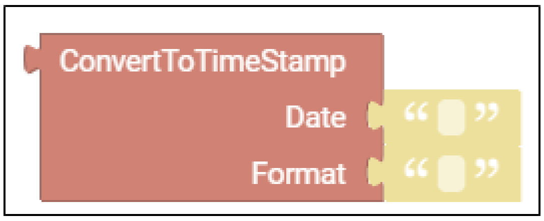


**Fig. 57: Example for Sleep0**

In this example, **`Sleep`** is used as a time buffer. Without a rest period of 20 millisecond, sending a pulse **(`SendImpulse`**) would be faster than calling the endpoint on a server. This would trigger an error.

## 13.4ConvertToTimeStamp



### Usage

The **ConvertToTimeStamp** block outputs a time stamp. **`Date`** contains the date to be converted, the **`Format`** string below defines the format of this date. The output is a UNIX value, i.e., the time in milliseconds after 01/01/1970 at 0:00.

### Input/Output

Input and output values can only be strings.

### Example

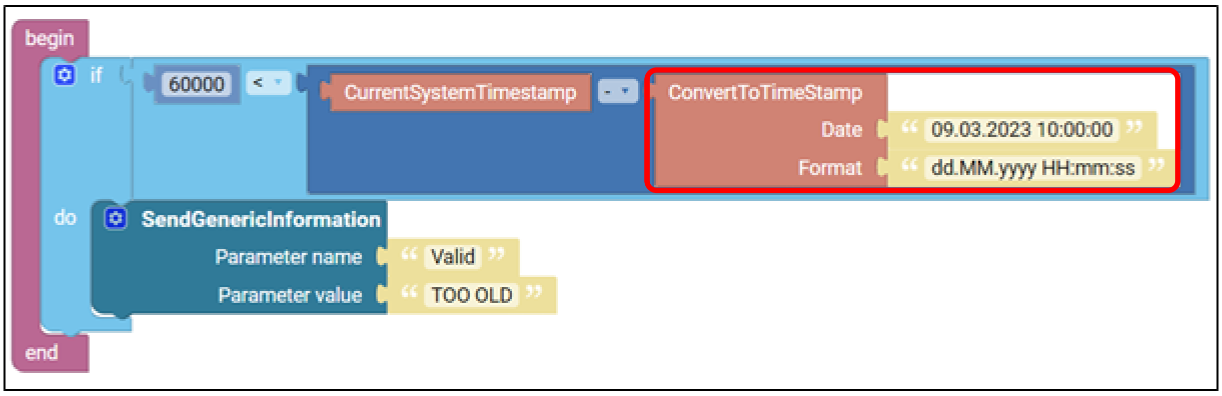
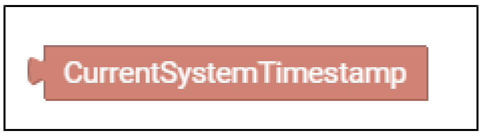


**Fig. 58: Example for ConvertToTimeStamp**

In this example, two different points in time shall be compared.
If the difference between the received time stamp (**`ConvertToTimeStamp`**) and the current time (**`CurrentSystemTimestamp`**) is more than 60,000 ms (i.e., one hour), a message is sent using the block. This message contains the information that the received time stamp is outdated.

## 13.5CurrentSystemTimestamp



### Usage

The **`CurrentSystemTimestamp`** block always enters the current Unix time. It indicates how many seconds have passed since 01.01.1970.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

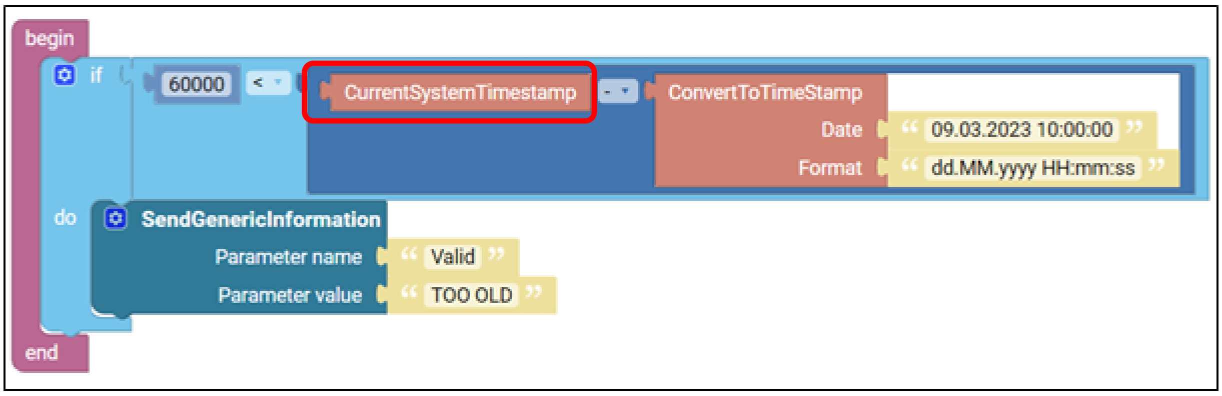


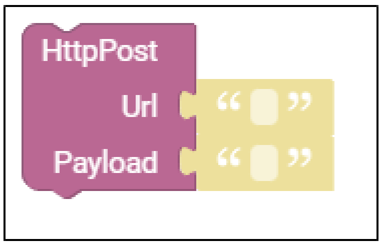
**Fig. 59: Example for CurrentSystemTimestamp**

In this example, two different points in time shall be compared.
If the difference between the received time stamp **(`ConvertToTimeStamp`**) and the current time **(`CurrentSystemTimestamp`**) is more than 60,000 ms (i.e., one hour), a message is sent using the **`SendGenericInformation`** block. This message contains the information that the received time stamp is outdated.

# 14 Additional actions (Misc)

This category is a collection of additional commands and blocks that create the connection to other systems. The related functions include, for example, integrating data from the Internet, retrieving the asset status, defining an asset as offline or outputting the IP address and host name.

## 14.1HttpPost



### Usage

Block **`HttpPost`** block sends a message to a third-party system. The Internet address (destination) is entered in **`Url`**. The **`payload`** refers to the actual data to be transmitted with the message.
We recommend to use the notation with two primes (superscript quotation marks, e.g., “k”).

### Input/Output

Inputs are strings. There are no restrictions to the output.

### Example

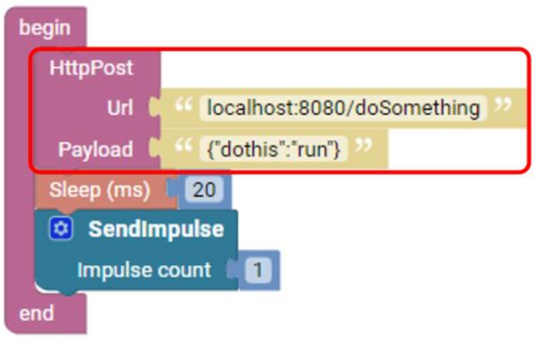
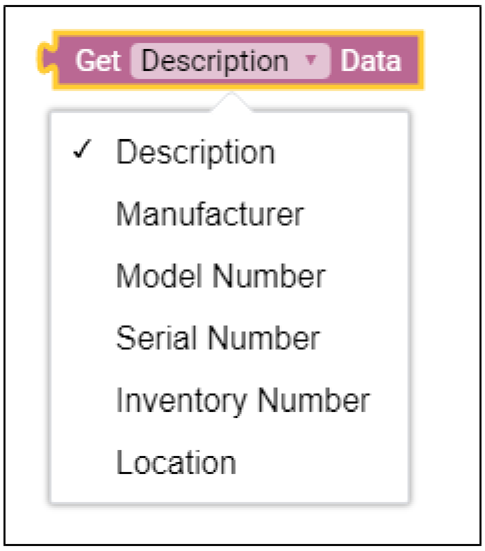


**Fig. 60: Example for HttpPost**

In this example, a server communication endpoint shall be called. The `url` and `payload` to be used fort the call are entered.
The program then waits for 20 ms **(`sleep`** block). This provides the time to call the page. Then the **`SendImpulse`** block sends the value 1.

## 14.2Get [specific] Data



### Usage

The **`Get [specific] Data`** block outputs specific information. Predefined data includes **`Description`**, **`Manufacturer`**, **`Model Number`**, **`Serial Number`**, **`Inventory Number`** and **`Location`**.
In the Configuration Wizard, parameters have already been determined in step 2 and step 3.
These parameters are automatically added to the drop-down menu.

### Input/Output

The input is selected from the drop-down menu. The output can only be string values.

### Example

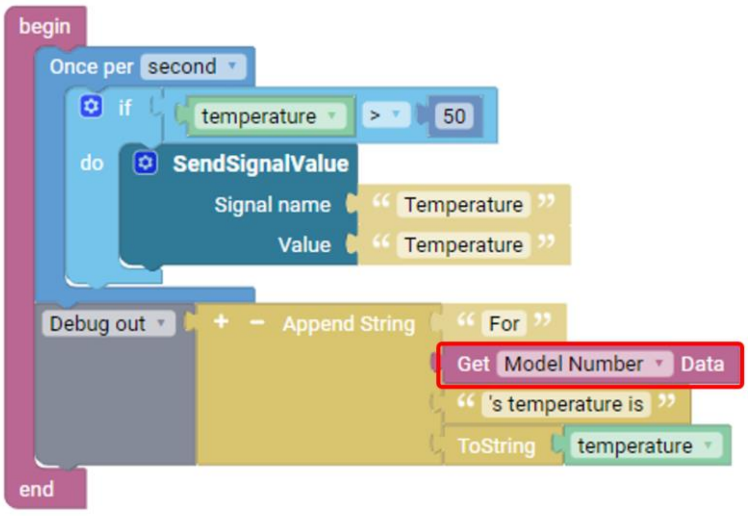
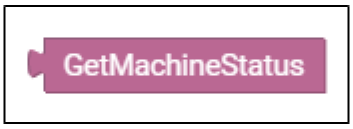


**Fig. 61: Example for Get [specific] Data**

If the temperature is higher than 50°C, the **`SendSignalValue`** block transmits “ Temperature” as the **`signal name`** together with the related temperature value **`(Value`**).
An entry is then made in the log file. The entry contains the number of the asset **`(Get [Model Number] Data`**), the text "'s temperature is" and the current value of the “temperature” variable.

## 14.3GetMachineStatus



### Usage

**`GetMachineStatus`** outputs the current machine status.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

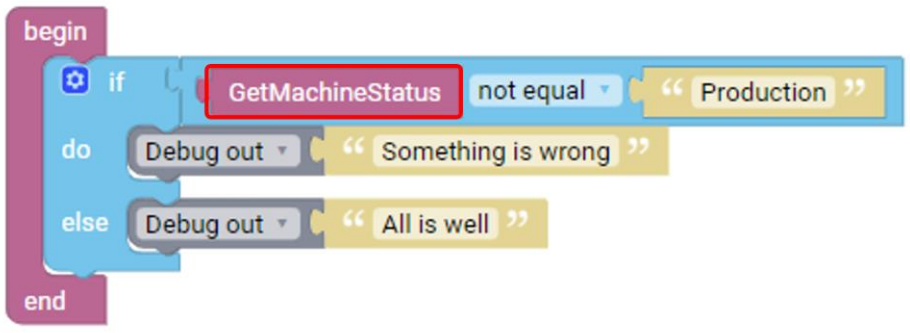
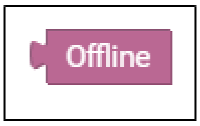


**Fig. 62**: **Example for GetMachineStatus**

In the example, **`GetMachineStatus`** is used to query the machine status. If this is **`not equal`** to the status **`Production`**,the entry **`Something is wrong`** is written to the log file via **`(Debug out`**). If not, the message **`All is well`** is written to the log.

## 14.4Offline



### Usage

If a system or machine is not in operation, the status query **`Offline`** can be used.

### Input/Output

There are no restrictions to the input. The output can only be boolean values.

### Example

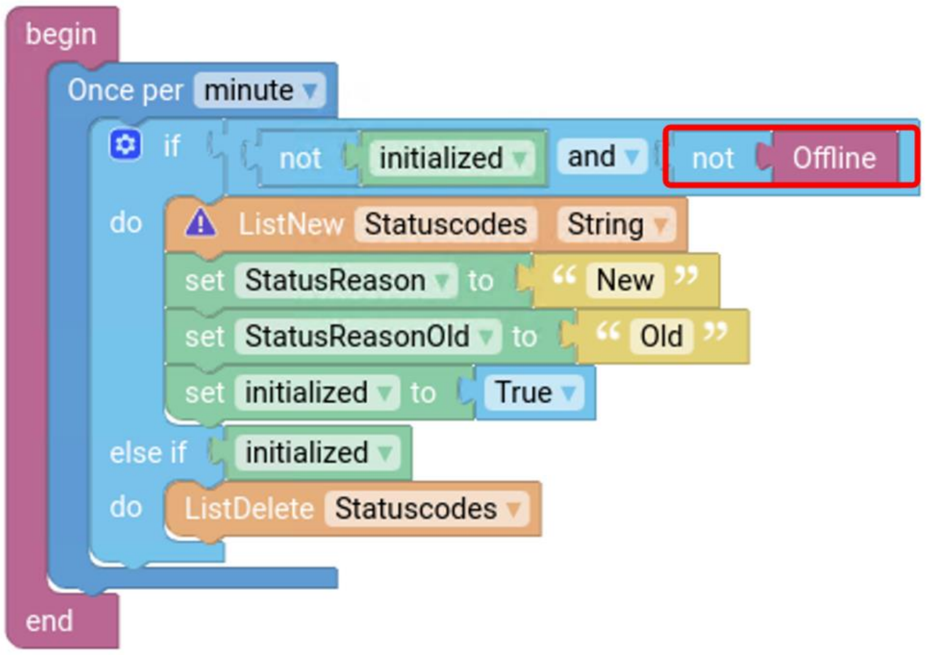


**Fig. 63: Example for Offline**

In the example, the program checks once per minute for the following status:

- The program has not just been initialized ((not initialized)
  and
- the asset is not offline (not Offline)

If this status applies, the asset is running. In this case, lists are created with the current and with previous reasons for a status. Afterwards, `True` is used to confirm that the program has just been started (`initialized`). This prevents the program from processing the upper part of the list again. The `ListDelete` block then deletes the list of status codes.

## 14.5IpAddress

### Usage

The `IPAdress` block outputs the IP address of an asset. The IP address is an individual address that identifies a device on the Internet or within a local network.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

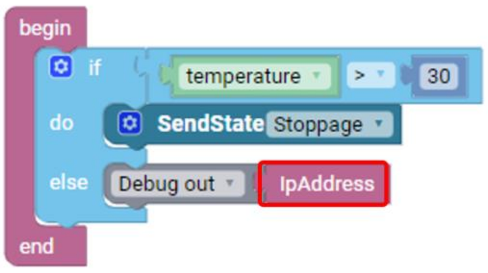


**Fig. 64: Example for IpAddress**

If the temperature is greater than 30, the **`SendState`** block sends the asset status **`Stoppage`**. In addition, the IP address (**`IPAdress`**) is written to the log file.

## 14.6HostName

### Usage

**`HostName`** enters the name of the host of an asset.
A host is a computer and the operating system running on it, that is part of a network and makes its services available to other network stations.

### Input/Output

There are no restrictions to the input. The output can only be string values.

### Example

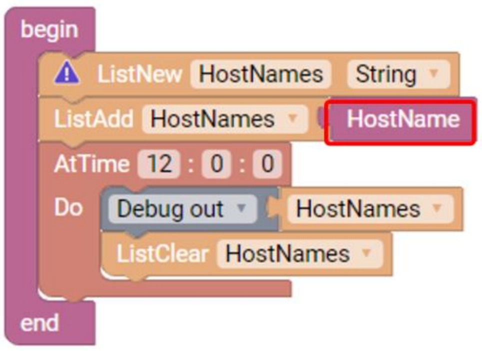


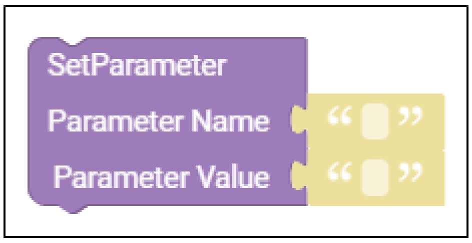
**Fig. 65: Example for HostName**

In this example, a new list (**`ListNew`**) is created. All **`HostName`** values are added to this list using the **`ListAdd`** block. At 12 o'clock, this list is written to the log file and the list is emptied afterwards.

# 15 Process asset properties (Business Parameters)

Business parameters are characteristics of a machine, such as a description, the manufacturer, model or serial number, or the machine's location. The corresponding data is provided in the Configuration Wizard in the previous configuration steps. (See the EDGE CONNECT manual.)

## 15.1SetParameter



### Usage

The **`SetParameter`** block specifies a new parameter and assigns a value to it. Name and the value of this parameter are entered in a string.
Parameters have also been defined in step 2 and 3 of the Configuration Wizard already. (See the EDGE CONNECT manual.)
If an already defined parameter is to be used, the **`GetParameter`** block (chapter 15.2) is used.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

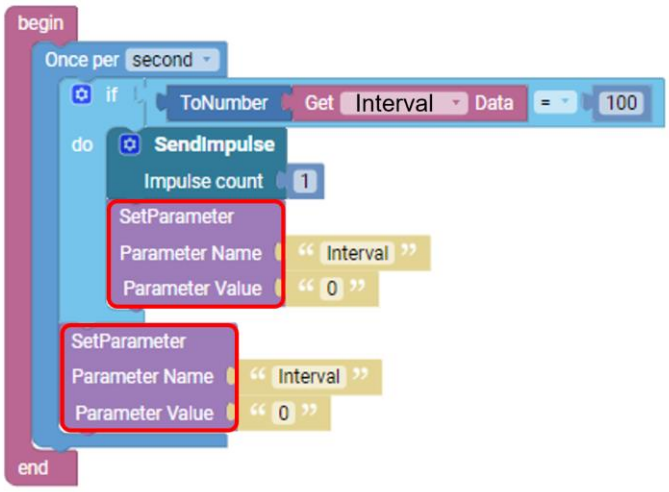
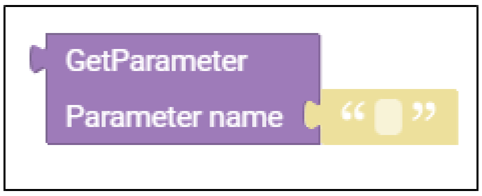


**Fig. 66: Example for SetParameter**

In this example, once a second a check is performed to determine whether the interval equals 100. If this is the case, an impulse is sent. After that, the **`Parameter Name`** "Interval" is reset to 0 (**`Parameter Value`**) using the **`SetParameter`** block. Otherwise, the program continues to increment the interval by 1.

## 15.2GetParameter



### Usage

The **`GetParameter`** block pulls the value of a parameter.

### Input/Output

Input and output values can only be strings.

### Example

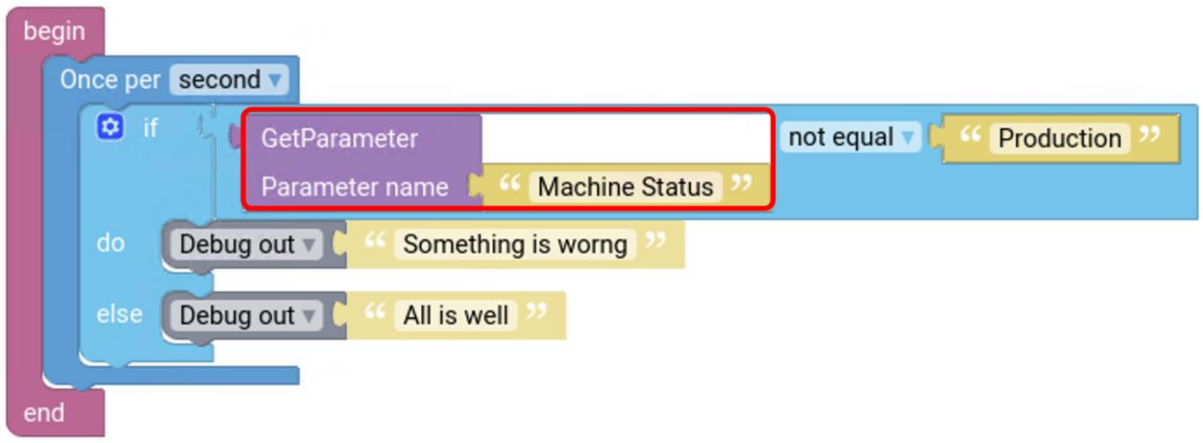
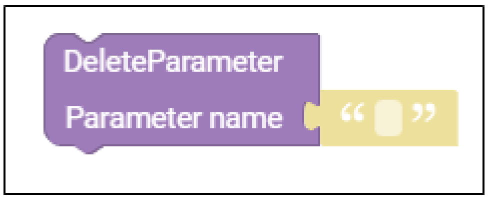


**Fig. 67: Example for GetParameter**

In this example, the machine status is requested once per second. This is done using the **`GetParameter`**block. The name of the parameter **`(Parameter name`**) is **`Machine Status`**.
If this name does **`not equal`** the status **`Production`**, the entry **`Something is wrong`** shall be written to the log file using the **`Debug out`** block. In any other case, the message **`All is well`** is written to the log ( **`Debug out`**).

## 15.3DeleteParameter



### Usage

The **`DeleteParameter`** block resets the parameter value in the database to 0.

### Input/Output

Only strings are possible as input values. There are no restrictions to the output.

### Example

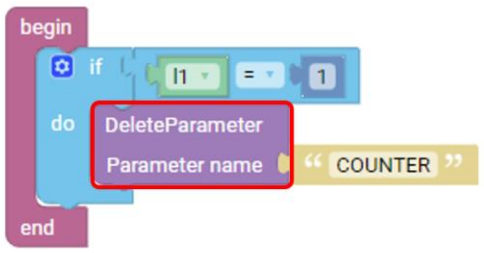


**Fig. 68: Example for DeleteParameter**

If the signal I1 is equal to(=) 1, the statement of the mathematical comparison = True (1). In this case, the **`DeleteParameter`** block resets the **`Parameter name`** COUNTER to 0.

## 15.4 DeleteAllParameter

### Usage

The **`DeleteAllParameter`** block deletes all parameters. It is used in the same way as the **`DeleteParameter`** block.

⚠ All parameters already used will also be reset to zero.

### Input/Output

There are no restrictions to the input and output values.

# 16 Glossary

| Abbreviations and terms used | Description |
|---|---|
| **Bit** | The smallest memory unit in a computer: 0 or 1 |
| **ERP** | Enterprise Resource Planning (a software solution for resource planning within companies) |
| **Hexadecimal number** | A number system that consists of 16 possible digit symbols and is used to facilitate the readability of large numbers or long bit sequences, e.g., in the ASCII table |
| **IoT** | Internet of Things |
| **MES** | Manufacturing Execution System |
| **SFT** | Shopfloor Terminal |
| **UTC** | Coordinated Universal Time |
| **°C** | Degree Celsius |

# 17 Annex

## 17.1Parameter overview

| Blocks | Other | Input | Output |
|---|---|---|---|
| **Variables** | | | |
| Get [Variable] | | N/A | Depends on the selection of String, Number or Boolean |
| Set [Variable] to | | Depends on the selection of String, Number or Boolean | N/A |
| **Signals** | | | |
| Set [Signal] to | | N/A | N/A |
| Get Signal | | N/A | N/A |
| Get base / scaled value for | | N/A | Number |
| **Events** | | | |
| SendImpulse<br>Impulse count<br>Reference<br>Customer specific settings | <br><br>Optional<br>Optional | <br>Number<br>String<br>String | <br>N/A<br>N/A<br>N/A |
| SendQuantity<br>Quantity<br>Unit<br>Quality details<br>Reference<br>Customer specific settings | <br><br>Optional<br>Optional<br>Optional<br>Optional | <br>Number<br>String<br>String<br>String<br>String | <br>N/A<br>N/A<br>N/A<br>N/A<br>N/A |
| SendState<br>State<br>Status codes<br>Reference<br>Customer specific settings | <br><br>Optional<br>Optional<br>Optional | <br>String<br>String<br>String<br>String | <br>N/A<br>N/A<br>N/A<br>N/A |
| SendSignalValue<br>Signal name<br>Value<br>Unit<br>Reference<br>Customer specific settings<br>Timestamp | <br><br><br>Optional<br>Optional<br>Optional<br>Optional | <br>String<br>String<br>String<br>String<br>String<br>String | <br>N/A<br>N/A<br>N/A<br>N/A<br>N/A<br>N/A |
| SendSignalPackage<br>Signal name | | String<br>String | N/A<br>N/A |

| Blocks | Other | Input | Output |
|---|---|---|---|
| Value<br>Unit<br>Reference<br>Customer specific settings | <br>Optional<br>Optional<br>Optional | String<br>String<br>String<br>String | N/A<br>N/A<br>N/A<br>N/A |
| SendGenericInformation<br>Parameter name<br>Parameter value<br>Reference<br>Customer specific settings | <br><br><br>Optional<br>Optional | <br>String<br>String<br>String<br>String | <br>N/A<br>N/A<br>N/A<br>N/A |
| SendState<br>Status codes<br>Reference<br>Customer specific settings | <br>Optional<br>Optional<br>Optional | <br>String<br>String<br>String | <br>N/A<br>N/A<br>N/A |
| **Logical** | | | |
| If-do<br>If<br>Else if<br>Else<br>Do | <br><br>Optional<br>Optional | <br>Boolean<br>Boolean<br>Boolean<br>Any | <br>N/A<br>N/A<br>N/A<br>N/A |
| Mathematical comparison =/≠/</>/≤/ | | Number | Boolean |
| Logical connective AND/OR | | Boolean | Boolean |
| Logical connective equal/not equal | | String | Boolean |
| Rising/Falling edge | | Boolean | Boolean |
| “NOT” statement | | Boolean | Boolean |
| Truth statement | | N/A | Boolean |
| **Repeaters** | | | |
| Once per | | Drop-down menu | N/A |
| **Arithmetic** | | | |
| Number field | | Number | Number |
| Math operation +/-/*/:/sin/cos/tan/sqrt | | Number | Number |
| ToNumber | | N/A | Number |
| **Logging** | | | |
| Logging | | String | N/A |
| **Text** | | | |
| String | | N/A | String |
| Append String | | String | String |
| ToString | | N/A | String |

| Blocks | Other | Input | Output |
|---|---|---|---|
| Length | | String | Number |
| SplitString<br>Input string<br>Separator<br>Index | | <br>String<br>String<br>Number | <br>String<br>String<br>Number |
| FromAscii | | Number | String |
| Substring<br>Input string<br>Start index<br>End index | <br><br><br>Optional | <br>String<br>Number<br>Number | String<br>N/A<br>N/A<br>N/A |
| **Lists** | | | |
| ListNew | | String | Drop-down menu |
| ListAdd | | String | N/A |
| ListClear | | Drop-down menu | N/A |
| ListDelete | | Drop-down menu | N/A |
| GetList | | N/A | String |
| **Date and time** | | | |
| FormatTime<br>Format<br>Offset<br>Offset unit | | <br>String<br>Number<br>Drop-down menu | String<br>String<br>String<br>String |
| AtTime Do | | Number | N/A |
| Sleep | | Number | N/A |
| ConvertToTimeStamp<br>Date<br>Format | | <br>String<br>String | Long<br>String<br>String |
| CurrentSystemTimestamp | | N/A | Long |
| **Misc** | | | |
| HttpPost<br>Url<br>Payload | | <br>String<br>String | <br>N/A<br>N/A |
| Get [specific] Data | | Drop-down menu | String |
| GetMachineStatus | | N/A | String |
| Offline | | N/A | Boolean |
| IpAddress | | N/A | String |
| Host | | N/A | String |
| **Business Parameters** | | | |
| SetParameter<br>Parameter name<br>Parameter value | | <br>String<br>String | <br>N/A<br>N/A |
| GetParameter | | String | String |
| DeleteParameter | | String | N/A |
| DeleteAllParameter | | N/A | N/A |

## 17.2ASCII table

| Dec | Char | Description |
|---|---|---|
| 0 | NUL | No input |
| 1 | SOH<br>Start of heading | Beginning of the header |
| 2 | STX<br>Start of Text | Beginning of a text part |
| 3 | ETX<br>End of text | End of a text part |
| 4 | EOT<br>End of transmission | Completion of a transmission |
| 5 | ENQ<br>Enquiry | A request for a response from the receiving station |
| 6 | ACK<br>Acknowledge | Confirmation |
| 7 | BEL<br>Bell | Generates an audible signal |
| 8 | BS<br>Backspace | Moves the cursor one position to the left and removes the character at this position |
| 9 | TAB<br>Horizontal tab | Tabulator for horizontal indentation of the next text character |
| 10 | LF<br>Line feed | Line break |
| 11 | VT<br>Vertical tab | Tabulator for horizontal indentation of the next text character |
| 12 | FF<br>Form feed | Page jump |
| 13 | CR<br>Carriage return | Positions the cursor at the beginning of a line |
| 14 | Shift out | Moves the cursor out |
| 15 | SI<br>Shift in | Moves the cursor inside |
| 16 | DLE<br>Data link escape | Shift character |
| 17 | DC1<br>Device control 1 | Device-specific function - often used as XON (continue transmission) |
| 18 | DC2<br>Device control 2 | Device-specific function |
| 19 | DC3<br>Device control 3 | Device-specific function - often used as XOFF (pause transmission) |
| 20 | DC4<br>Device control 4 | Device-specific function |
| 21 | NAC<br>Negative acknowledge | Negative confirmation |
| 22 | SYN<br>Synchronous idle | In synchronous data transmissions, enables synchronization even in the absence of signals to be transmitted |
| 23 | ETB<br>End of trans. Block | Indicates the end of a data block |
| 24 | CAN<br>Cancel | Cancel |
| 25 | EM<br>End of medium | Indicates the end of a medium. |
| 26 | SUB<br>Substitute | Replace |
| 27 | ESC<br>Escape | Cancels an activity |
| 28 | FS<br>File separator | Separation of main groups |
| 29 | GS<br>Group separator | Group separation |
| 30 | RS<br>Record separator | Subgroup separation |
| 31 | US<br>Unit separator | Separation of parts of a group |

| Dec | Char | Description |
|---|---|---|
| 32 | Space | Blank character |
| 33 | ! | |
| 34 | ” | |
| 35 | # | |
| 36 | $ | |
| 37 | % | |
| 38 | & | |
| 39 | ` | |
| 40 | ( | |
| 41 | ) | |
| 42 | * | |
| 43 | + | |
| 44 | , | |
| 45 | - | |
| 46 | . | |
| 47 | / | |
| 48 | 0 | |
| 49 | 1 | |
| 50 | 2 | |
| 51 | 3 | |
| 52 | 4 | |
| 53 | 5 | |
| 54 | 6 | |
| 55 | 7 | |
| 56 | 8 | |
| 57 | 9 | |
| 58 | : | |
| 59 | ; | |
| 60 | < | |
| 61 | = | |
| 62 | > | |
| 63 | ? | |
| 64 | @ | |
| 65 | A | |
| 66 | b | |
| 67 | C | |
| 68 | D | |
| 69 | E | |
| 70 | F | |
| 71 | G | |
| 72 | H | |
| 73 | I | |
| 74 | J | |
| 75 | K | |
| 76 | L | |
| 77 | M | |
| 78 | N | |
| 79 | O | |
| 80 | P | |
| 81 | Q | |
| 82 | R | |
| 83 | S | |
| 84 | T | |
| 85 | U | |
| 86 | V | |
| 87 | W | |
| 88 | X | |
| 89 | Y | |
| 90 | Z | |
| 91 | [ | |
| 92 | \ | |
| 93 | ] | |
| 94 | ^ | |
| 95 | _ | |
| 96 | ` | |
| 97 | a | |
| 98 | b | |
| 99 | c | |
| 100 | d | |
| 101 | e | |
| 102 | f | |
| 103 | g | |
| 104 | h | |
| 105 | i | |
| 106 | j | |
| 107 | k | |
| 108 | l | |
| 109 | m | |
| 110 | n | |
| 111 | o | |
| 112 | p | |
| 113 | q | |
| 114 | r | |
| 115 | s | |
| 116 | t | |
| 117 | u | |
| 118 | v | |
| 119 | w | |
| 120 | x | |
| 121 | y | |
| 122 | z | |
| 123 | { | |
| 124 | \| | |
| 125 | } | |
| 126 | ~ | |

| Dec | Char | Description |
|---|---|---|
| 127 | DEL<br>Delete | Delete the last character |