



Graphical Composition

Getting Started with Signal Composition

Version: 230406

Manual



Document: Graphical Composition - Getting Started with Signal Composition



Release date: 01.06.2023



Document version: 2.0



Author: FORCAM GmbH

Contents

1	About this document	5
1.1	Target group	5
2	Concept	6
2.1	FORCE EDGE CONNECT & Graphical Composition	6
2.2	Customer benefits of the Graphical Composition	7
3	From machine signals to events	8
3.1	Procedure of signal interpretation	8
3.2	Important basic information.....	9
3.2.1	General explanation of the user interface.....	9
3.2.2	Function categories	11
3.2.3	Notation of numbers	13
3.3	General handling.....	13
3.3.1	Layout of the blocks.....	16
3.3.2	Shadow blocks	18
3.4	Error detection.....	19
4	Variables	20
4.1	Get [Variable].....	22
4.2	Set [Variable] to	23
5	Signals	24
5.1	Set [Signal] to.....	25
5.2	Get Signal.....	26
5.3	Get base / scaled value for	27
6	Events.....	28
6.1	SendImpulse	28
6.2	SendQuantity	29
6.3	SendState.....	30
6.4	SendSignalValue.....	31
6.5	SendSignalPackage	32
6.6	SendGenericInformation	33
6.7	SendState [Selection]	34
7	Logical	36

7.1	If-do.....	36
7.2	Mathematical comparison: =/≠/</>/≤/≥.....	37
7.3	Logical connective: and/or	38
7.4	Logical connective: equal/not equal	39
7.5	Rising/Falling edge	40
7.6	“Not” statement	41
7.7	True statement	42
8	Repeaters	44
8.1	Once per	44
9	Arithmetic	45
9.1	Number field.....	45
9.2	Mathematical operation.....	45
9.3	ToNumber.....	46
10	Logging	47
10.1	Logging	47
11	Text	48
11.1	String.....	48
11.2	Append String	48
11.3	ToString	49
11.4	Length.....	50
11.5	SplitString	50
11.6	FromAscii	51
11.7	Substring.....	52
12	Lists	54
12.1	ListNew	54
12.2	ListAdd.....	55
12.3	ListClear	56
12.4	ListDelete.....	57
12.5	GetList	58
13	Date and time.....	59
13.1	FormatTime	60
13.2	AtTime Do.....	61
13.3	Sleep.....	62

13.4	ConvertToTimeStamp.....	63
13.5	CurrentSystemTimestamp	64
14	Misc.....	65
14.1	HttpPost	65
14.2	Get [specific] Data.....	66
14.3	GetMachineStatus.....	67
14.4	Offline.....	67
14.5	IpAddress.....	68
14.6	HostName	69
15	Business Parameters.....	70
15.1	SetParameter	70
15.2	GetParameter	71
15.3	DeleteParameter.....	71
15.4	DeleteAllParameter.....	72
16	Glossary.....	73
17	Annex	74
17.1	Parameter overview.....	74
17.2	Ascii table	77

1 About this document

This document describes how to use the editor for easy interpretation of asset signals.

- ① 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.

1.1 Target group

The manual requires knowledge in the use of FORCE EDGE CONNECT (hereafter simply referred to as EDGE CONNECT). If you do not have any knowledge in this area, take the time to familiarize yourself with the basics.

See the **Manual - FORCE EDGE CONNECT** for detailed information. (As of version 230406.)

- ① 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 to initiate changes in the company and to use the technologies correctly.

2 Concept

To be able to evaluate the signals that have been read from a machine, sensor, etc. (called assets), these signals must first 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 determine, when data is to be sent to an MES, ERP or another third-party system.

For users that are new to the subject of signal interpretation, the way it was previously done in EDGE CONNECT sometimes turned out to be quite complex and confusing. The Graphical Composition offers an easy-to-use alternative to that. With the graphical editor, we aim to provide a beginner-friendly solution that enables any user to create and use common machine instructions. This enables any user to send and interpret the data.

This manual describes in detail the various functions and options according to topics and also provides practical examples.

2.1 FORCE EDGE CONNECT & Graphical Composition

Machine parks are often made up of many different machines from various manufacturers and of different ages. This variety leads to a number of different signal formats and machine outputs, making the interpretation of this data a complex task. EDGE CONNECT, however, standardizes the raw signals so that they become comparable and thus provide an optimal basis for subsequent analysis. The interpretation of the asset data is an essential part of this process. This is done by the Signal Composition component. Here, data received from the Southbound Link can be used to interpret and process the signals. It is also possible here to specify the time at which the data should be sent. These data packages are sent only via the Northbound Link. This way, EDGE CONNECT can be used to supply third-party systems with production data.

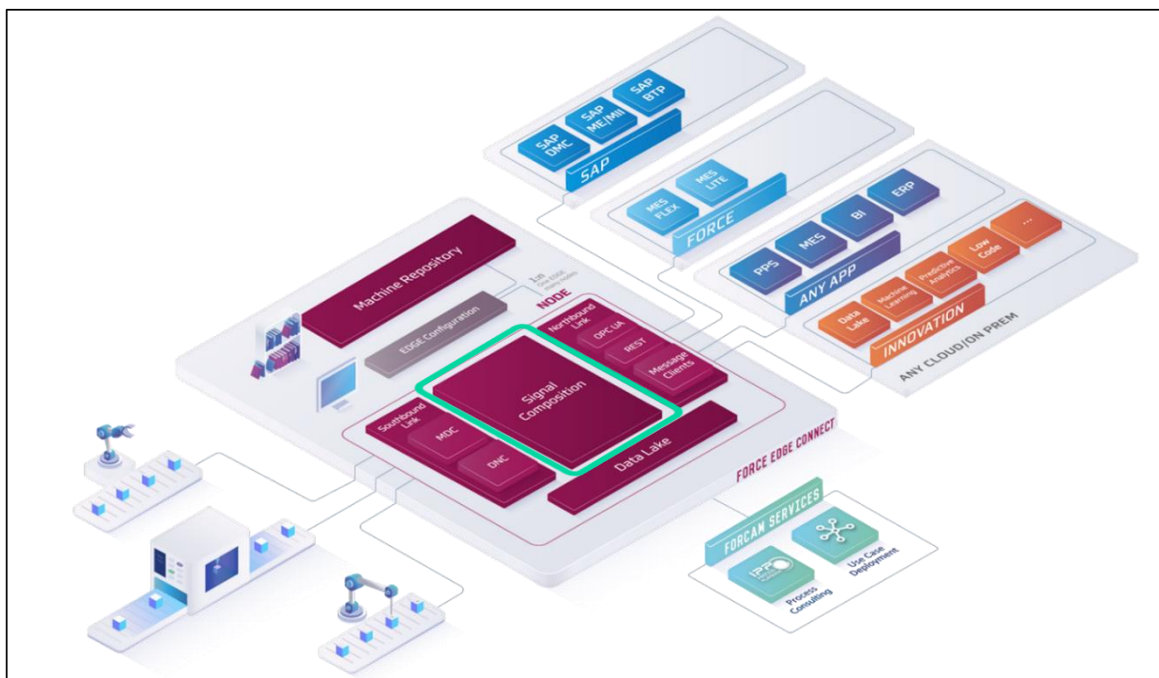


Fig. 1: FORCE EDGE CONNECT architecture overview

2.2 Customer benefits of the Graphical Composition

The essential benefit of the Graphical Composition is the approach to make the interpretation of asset signals as easy as possible even for beginners. The selected commands are visualized in a graphical way with the help of colored puzzle pieces. This way, beginners in programming can implement basic commands easily, even without prior knowledge of the subject. Knowledge in a programming language is not required. Syntax errors can be excluded. Different mechanisms make it easier to recognize other types of errors. Clearly arranged functions facilitate the handling of the editor. Templates, for example from the Machine Repository, can also be used together with the Graphical Composition.

3 From machine signals to events

The Graphical Composition provides an alternative to the classic script-based **Composition** in the EDGE CONNECT Asset Wizard. A script is a short sequence of commands that are executed by the desired program. The Composition assigns a meaning to a signal. For example, a pure numerical value (such as 0 or 1) is turned into a readable and understandable information, for example “Production” or “Stoppage”. The Graphical Composition editor is used like a building block system to make these assignments.

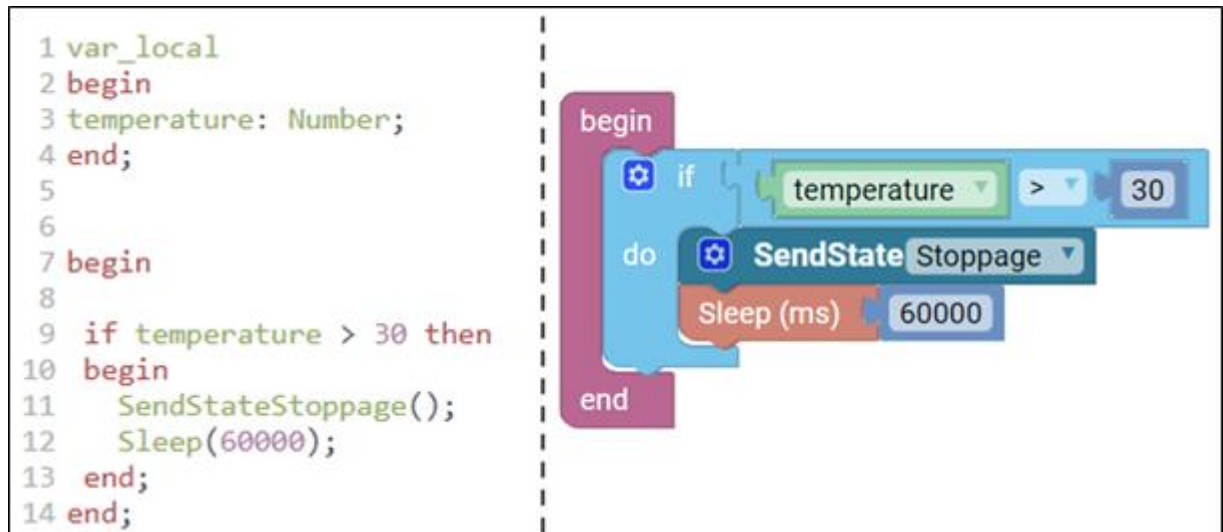


Fig. 2: Script-based vs. graphical editor

3.1 Procedure of signal interpretation



Fig. 3: Configuration Wizard

Multiple subsequent steps must be followed to interpret a signal:

Create the asset

First, the required information on the asset (machine or sensor) must be entered. This is done in steps 2 to 5 of the Configuration Wizard. New signals can be created in addition to the automatically created signals. This is done in step 5.

Prepare collectors

In step 6 **Composition**, variables are created at first. To be able to analyze these variables, they must be assigned a meaning.

Interpret signals

Signals are evaluated according to specified conditions. Depending on the condition that is used, events (incidents or actions) are executed.

Send events

In the Compositions, signals are sent to third-party systems by means of events.

Record signals

All signals can be documented.

3.2 Important basic information

3.2.1 General explanation of the user interface

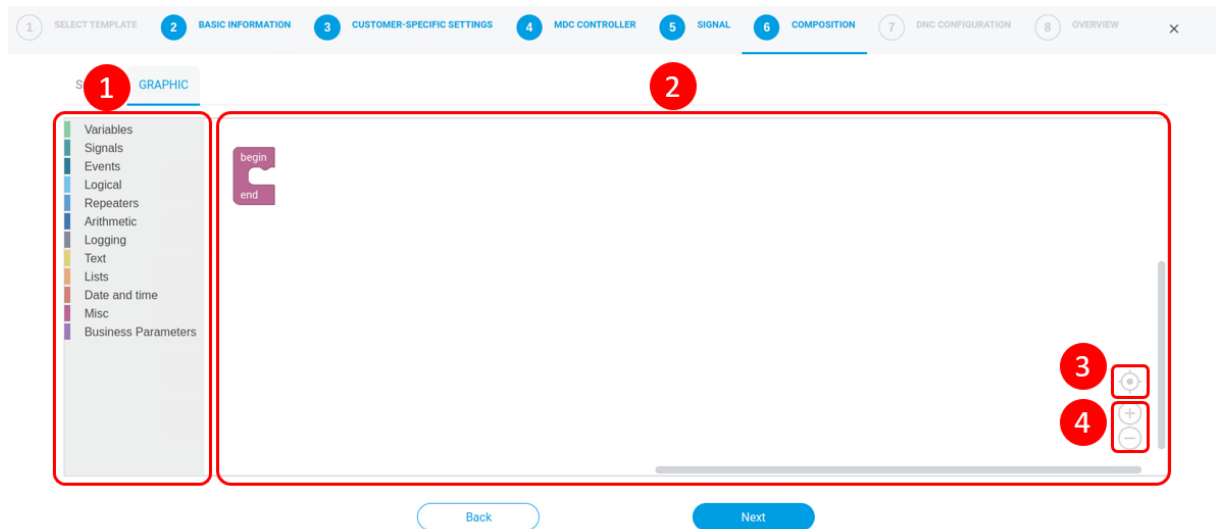


Fig. 4: Graphical editor

The conditions for the interpretation of signals are specified in step 6 **Composition** of the Configuration Wizard. They are displayed in two ways: One way is the display as graphical blocks in the graphical editor (under **GRAPHICS**). These are the programming blocks that can be assembled. The other option is to use the script editor (**SKRIPT** tab) for text-based coding.

The editor in the **GRAPHIC** tab (see Fig. 4) consists of the following elements:

- (1) Selection window with function categories and blocks
- (2) Editing area for assembling the blocks
- (3) Navigation: Center
- (4) Navigation: zoom in/out the view

Beginners are likely to work with this editor and the graphical blocks.

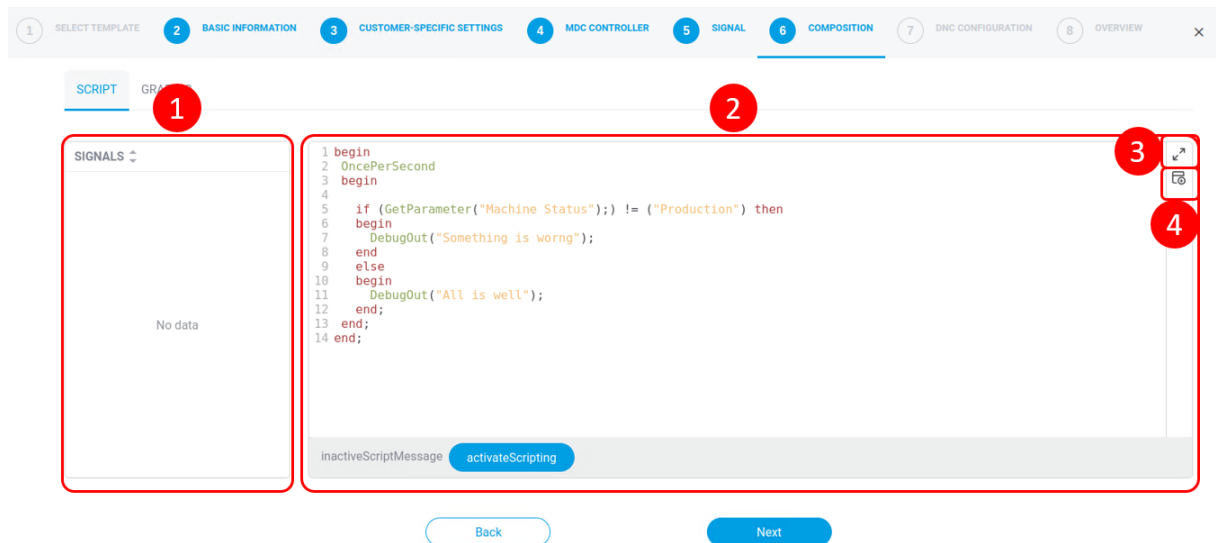



Fig. 5: Script editor

On the **script** tab, more advanced users can read the programming code that matches the block assembly. The displayed data is automatically synchronized.

 It is not possible to edit a structure in **SCRIPT** and **GRAPHIC** simultaneously. If the graphic was converted to a script, the script can be edited in the **SCRIPT** area but it cannot be reset to the block variant.

Only users that are familiar with programming should work in the **SCRIPT** editor.

The left area (1) shows the signals that were added in step 5 of the Configuration Wizard. The right area (2) displays the current script. The arrows (3) are used to enlarge this area. The script can be validated, i.e., checked for validation errors using the icon marked with (4).

3.2.2 Function categories



Fig. 6: Categories of blocks

The composition functions are divided into the following categories, which combine specific topics: Variables, Signals, Events, Logical, Repeaters, Arithmetic, Logging, Text, Lists, Date and time, Misc and Business Parameters. 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. The naming is in English. The following sections provides a short overview of the different categories:

(1) Variables

Variables are like collectors to save the data. They are used to safely store elements (usually entries or calculation results). These collectors can be of specific types. The individual types have specific restrictions regarding their content (numerals, words., etc.) and their size (how small/large, etc.). These can contain either static values (asset name, status, etc.) or calculations (temperature, pressure, time units, etc.). Changes in the variable are registered in the system. All blocks of this variable adopt the new value.

There are three types of variables and there are restrictions in combining these types during assembly: The following types of variables are available:

→ **Boolean:**

A boolean is a TRUE/FALSE value. A boolean either indicates that an event is True (1) or False (0). Or it indicates whether the event has occurred (True/1) or not (False/0). These values are also called boolean values.

Ex: The maximum value of a temperature was reached or a time period was exceeded.

→ **String:**

A string consists of an number of characters. Characters can be numbers, letters or symbols. Strings are automatically placed in quotation marks.

Ex: identification number, asset name

→ **Number:**

Numbers have numerical values.

Ex: temperature in °C, number of minutes

(2) Signals

Signals are functions that are usually measured using sensors and which carry information. In the production environment, typical signals are intervals, temperatures, machine states and pressures.

(3) Events

So-called event blocks are used to send impulses, production states or values.

(4) Logical

This category is used to relate values to each other. This enables decisions about their logical value or status.

Predefined rules decide about potential actions resulting from the value.

E.g., Events, Lists, Date and time, Business Parameters, Logging and many others.

(5) Repeaters

In many cases, actions are repeated in regular intervals. Repeaters trigger an action in the predefined interval.

(6) Arithmetic

These blocks implement calculation functions such as adding, subtracting or multiplying values. They also convert strings into numbers.

(7) Logging

Specific values can be logged and made available for analysis (debug out). Different warning levels are applied for this.

(8) Text

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

(9) Lists

Usually, a list is used to collect different production states. This category defines how to create, fill, empty and delete lists.

(10) 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.

(11) Misc

The category is a collection of additional commands and creates the connection to other systems. These include: integrating data from the Internet, retrieving the asset status, defining an asset as offline or outputting the IP address and host name

(12) Business Parameters

Business parameters are characteristics of a machine. This includes, for example, Description, Manufacturer, Model Number, Serial Number, Inventory Number and Location. The parameters are created in previous configuration steps.

3.2.3 Notation of numbers

⚠ The Graphical Composition uses the English notation for numbers. This changes the use of point and coma. Table 1 contains some examples to illustrate this.

Table 1: Notation of the numbers

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 General handling

Each block can be used on its own and fulfills a specific task. To fulfill this task, additional information from other blocks might be required. Blocks are moved using Drag-and-drop. Copying is also possible using the keyboard shortcuts Strg+C / Strg+V. The Del key removes the block.



Each composition starts and ends with the **begin...end** frame. This block automatically appears in the editing area. There are no other restrictions to the block sequence within the composition.

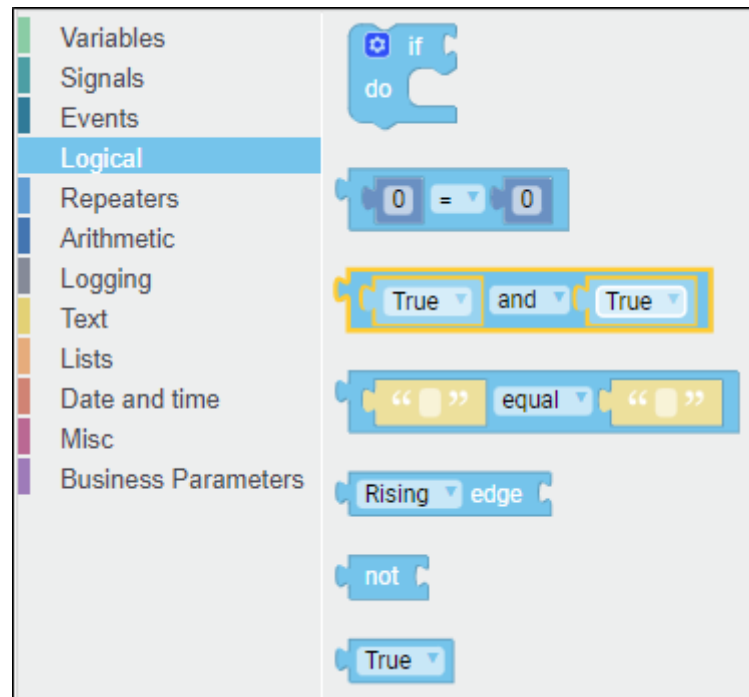


Fig. 7: Selecting functions

Clicking on a function category opens a window with the available functions.

There are functions that act as kind of bracket. They need additional blocks to complete the function. There are also blocks that need to be connected to other blocks. They contain placeholders for variables or entries.

There are specific input rules for each type of block for the use of variables or entries. In some cases, only booleans, strings or numbers are allowed.

A block that does not match the restrictions cannot be connected. It “jumps” away and is highlighted in gray.

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

The blocks are read from top to bottom and from left to right.

The input for the Graphical Composition is the content that the application needs to run the command. The output is the result or the command itself.

Top to bottom

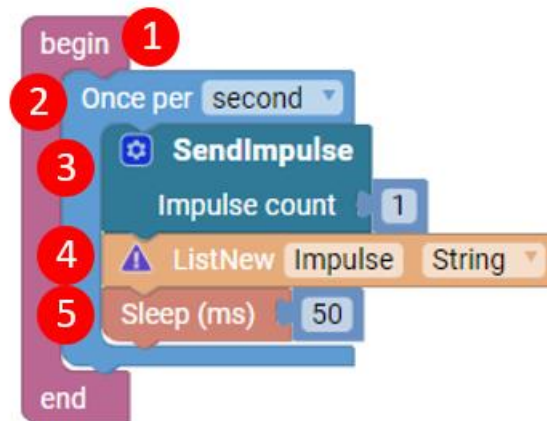


Fig. 8: Example of a top-to-bottom sequence

The **begin...end** block (1) is the overall frame of each composition. First, the program executes the block **Once per second** (2), colored in light blue. It is followed by the **SendImpulse** block in dark blue (3). After that, the orange **ListNew** block (4) is processed. The last block is the red **Sleep** block (5).

Left to right

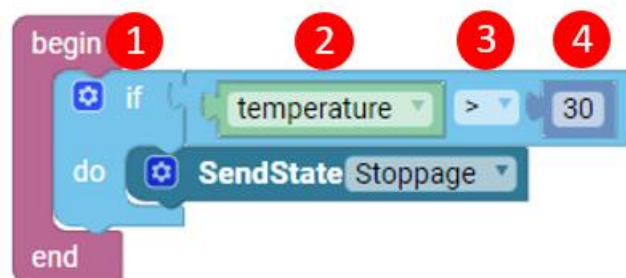


Fig. 9: Example of a left-to-right sequence

The lines within the composition are read from left to right, like a book.

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).

Then the program jumps to the next line. This line 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.

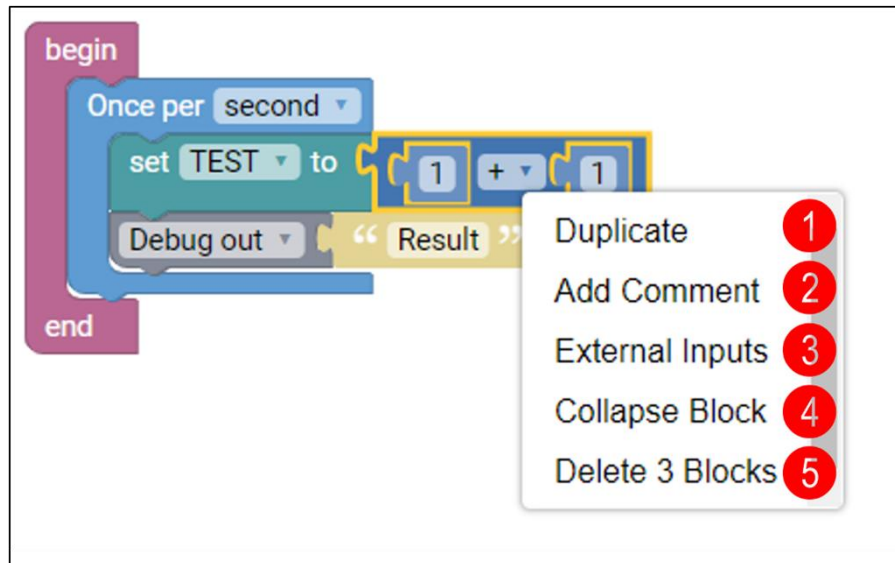


Fig. 10: Possible actions for a block

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

- (1) Duplicate block
- (2) Add comment
- (3) External Inputs/Inline Inputs: Changes the display format
- (4) Collapse block
Joined blocks can be collapsed to save space and keep the overall picture clear and readable.
- (5) Delete block groups

3.3.1 Layout of the blocks

Each block represents one action. The blocks are composed in a modular way. Matching blocks can be combined with each other to build the overall structure. This overall structure contains the signal processing commands. The following sections explain the general layout and specific features of individual blocks based on graphical examples:



Fig. 11: Connection points

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

For one, there are blocks with rounded arrows pointing downwards (1). These blocks define basic conditions and pass on the output to another block. Output is always passed on if there is an open connection point at the right side of the block. A block group must be completely closed on the right. On the other hand, there are the classic puzzle elements (2), which are put together from left to right. They pass on the input of the data.

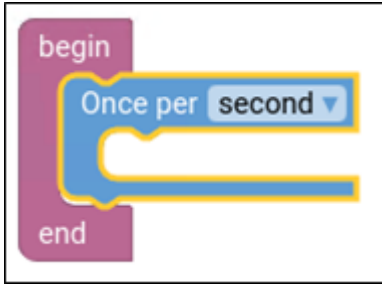


Fig. 12: Yellow frame

the yellow frame indicates the currently selected block.

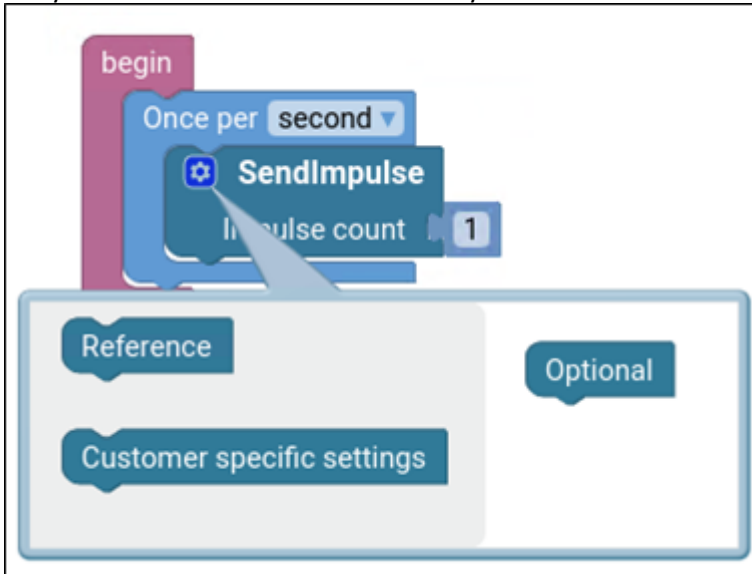


Fig. 13: Example of additional extension options

Some blocks have a blue “settings” icon. Clicking this icon opens a window with additional blocks. These provide further extension options. The blocks can be dragged from the gray area on the left to the right-hand side below **Optional**. With the **Customer specific settings** block, customized settings can be transferred, if required.

- i** 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 our example, the **Customer specific setting** block can only be used after a **Reference** has been written.

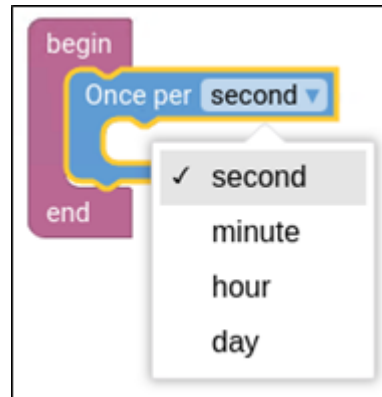


Fig. 14: Drop-down menu

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

3.3.2 Shadow blocks

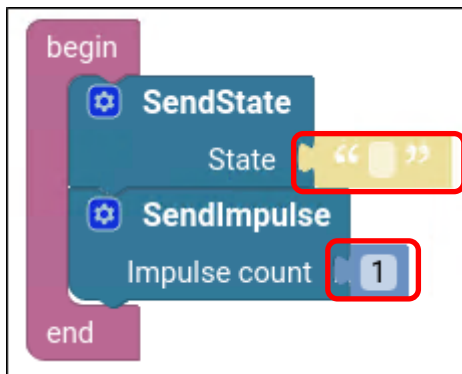


Fig. 15: Shadow blocks

Shadow blocks act as placeholders for inputs. They indicate that an input is mandatory for the block to complete its function. A shadow block is always connected to a superordinate block if this block is selected in the function category window.

A shadow block can be identified by a lighter coloring, and it indicates that this block parameter must not be empty. Either the value is entered manually (as in Fig. 15 in the yellow and blue fields), or another block is dragged there.

3.4 Error detection

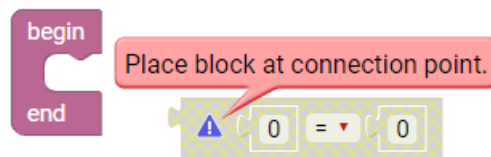


Fig. 16: Note: invalid block

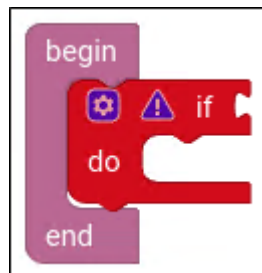


Fig. 17: Note: block not yet complete

An error in the structure is indicated in two ways.

One way is that the application does not accept the block, as shown in Fig. 16. The block is highlighted in gray. This applies if function categories or the types of variables do not match. A typical error would be to insert a Numbers block where a String would be needed.

The other way is that a block is colored in red until all information required by this block has been added to the structure. Once all required blocks are available, the block returns to its original color.

Only valid blocks are accepted. Clicking the exclamation mark displays the message cause. This makes error recognition fast and easy.

4 Variables

Variables store data. There are three types of variables: Boolean, String or Number. Each type has specific restrictions regarding the variables' size and content. Refer to chapter 3.2.2 Function categories for more information.

Handling variables

Variables must be created first and can then be used in the Graphical Composition.

Clicking the function category **Variables** opens the list of available blocks. At the very top of the field, the **Create new variable** option is available. The name and the type (number, string or boolean) must be defined for each variable.

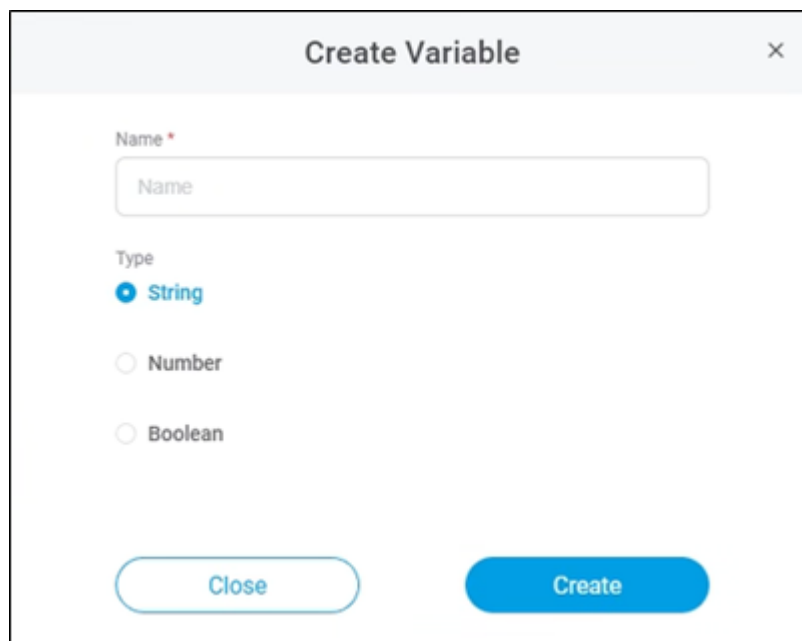
A screenshot of a 'Create Variable' dialog box. The dialog has a title bar with the text 'Create Variable' and a close button (X). Inside the dialog, there is a 'Name' field with a red asterisk indicating it is required, and a text input box containing the word 'Name'. Below the name field is a 'Type' section with three radio button options: 'String' (which is selected and highlighted in blue), 'Number', and 'Boolean'. At the bottom of the dialog, there are two buttons: 'Close' and 'Create'.

Fig. 18: Creating a new variable

Variables can be added as many times as required.

To provide a structured overview, created variables appear under the sub-headings String, Number and Boolean.

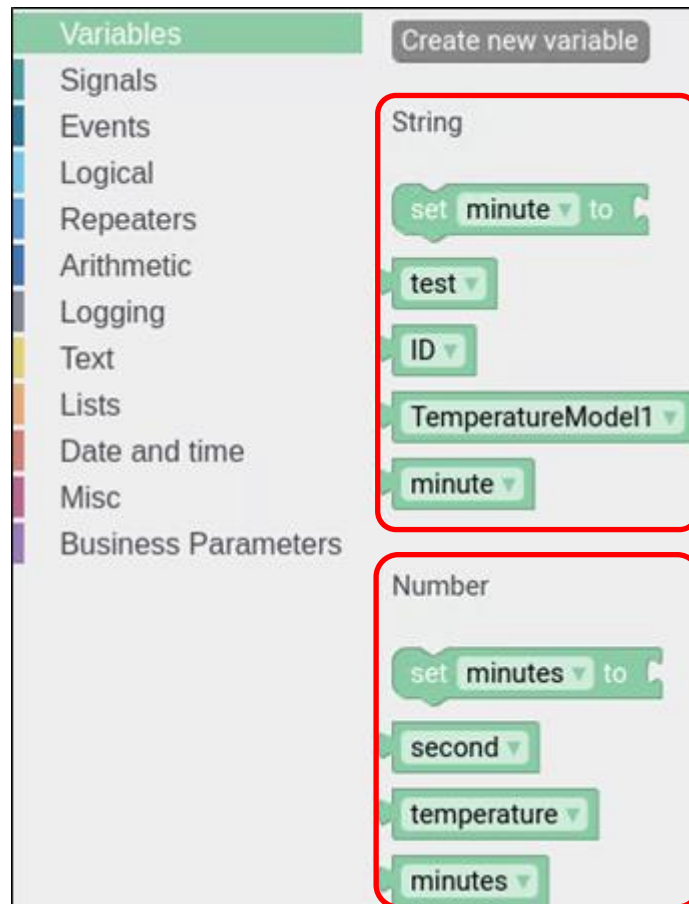


Fig. 19: Overview of variable types

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

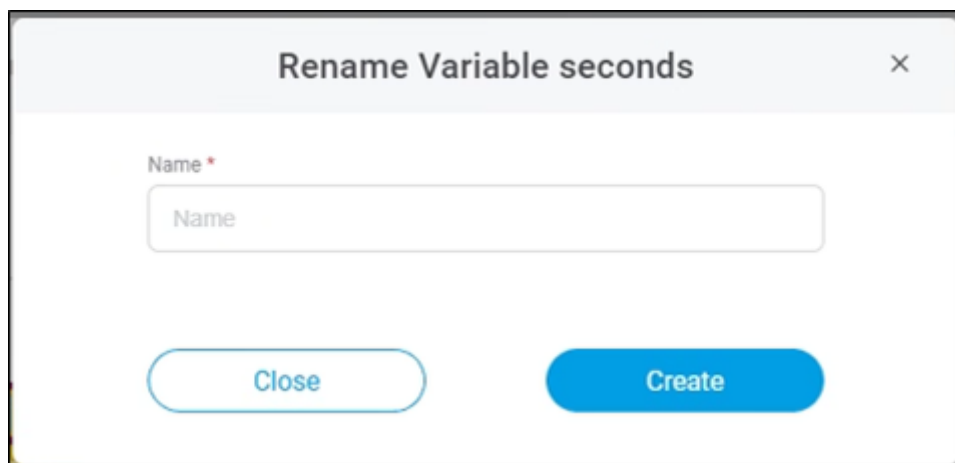


Fig. 20: Renaming a variable

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.

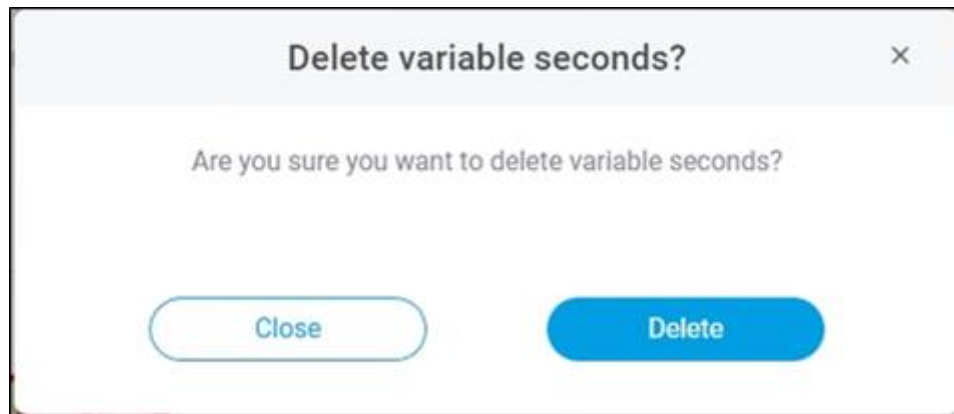


Fig. 21: Deleting a variable

4.1 Get [Variable]

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. There are no restrictions to the input. The output corresponds to the type of variable.

Example

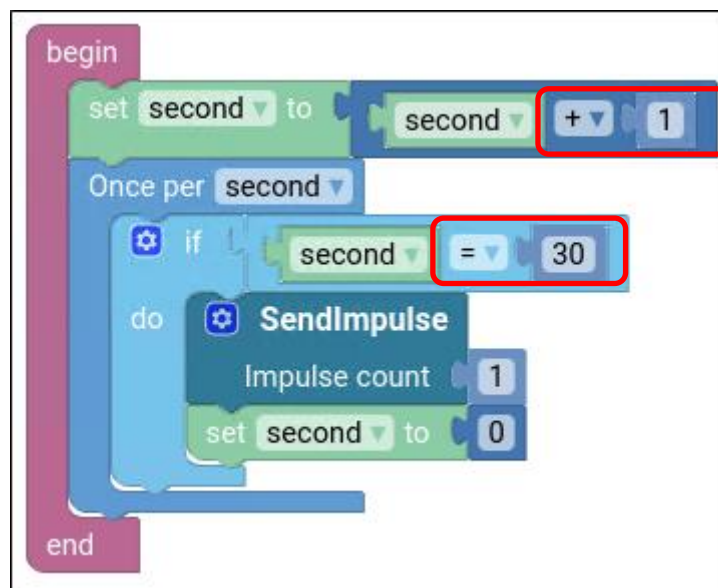


Fig.22: Example for Get [Variable]

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.

4.2 Set [Variable] to



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.

Input and output correspond to the type of variable.

Example:

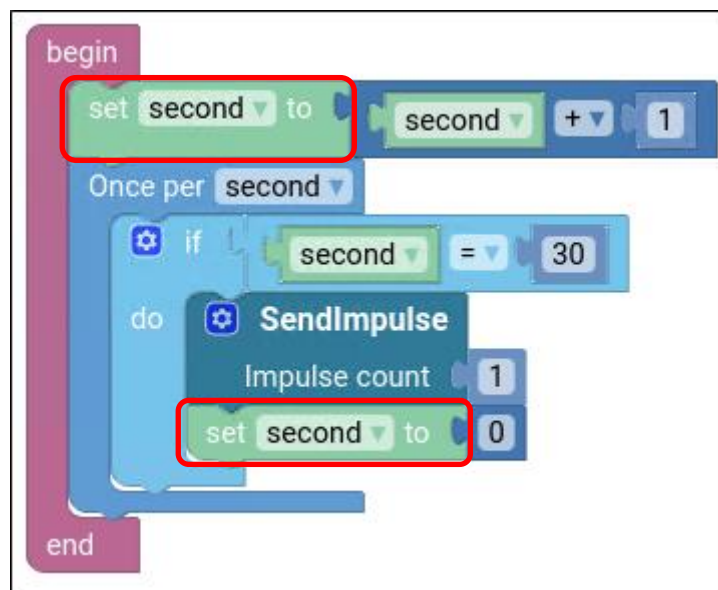


Fig.23: Example for set [Variable] to

In this case, 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 Signals

In most cases, signals are detected at the assets using sensors, they transfer the information to EDGE CONNECT.

Handling signals

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

The screenshot displays the 'Configure machine signals' window, which is part of a multi-step configuration process. The steps are: 1. SELECT TEMPLATE, 2. BASIC INFORMATION, 3. CUSTOMER-SPECIFIC SETTINGS, 4. MDC CONTROLLER, 5. SIGNAL (current step), 6. COMPOSITION, 7. DMC CONFIGURATION, and 8. OVERVIEW. The main area is divided into two panels. The left panel, titled 'Configure machine signals', contains a table with columns: TYPE, SIGNAL, ACTIVE, and DATA LAKE. There are two rows of signals: 'Binary input' with signal name 'test' and 'Binary output' with signal name 'done'. Both are active (blue toggle) and have data lake access (grey toggle). A '+' button is at the top right of this panel. The right panel, titled 'PARAMETER', contains configuration options for the selected signal. It includes 'Addressing' (Address (high) and Address (low) fields), 'Unit & Scaling' (Unit dropdown, Scale factor field, and Scale offset field), and 'Additional information' (Tags dropdown and Description field). At the bottom, there are 'Back' and 'Next' buttons.

TYPE	SIGNAL	ACTIVE	DATA LAKE
Binary input	test	ON	OFF
Binary output	done	ON	OFF

PARAMETER

Addressing

Address (high) Address (low)

Unit & Scaling

Unit Scale factor

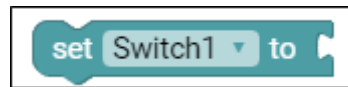
Scale offset

Additional information

Tags Description

Fig. 24: Adding a new signal

5.1 Set [Signal] to



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. There are no restrictions to the input and output values.

Example

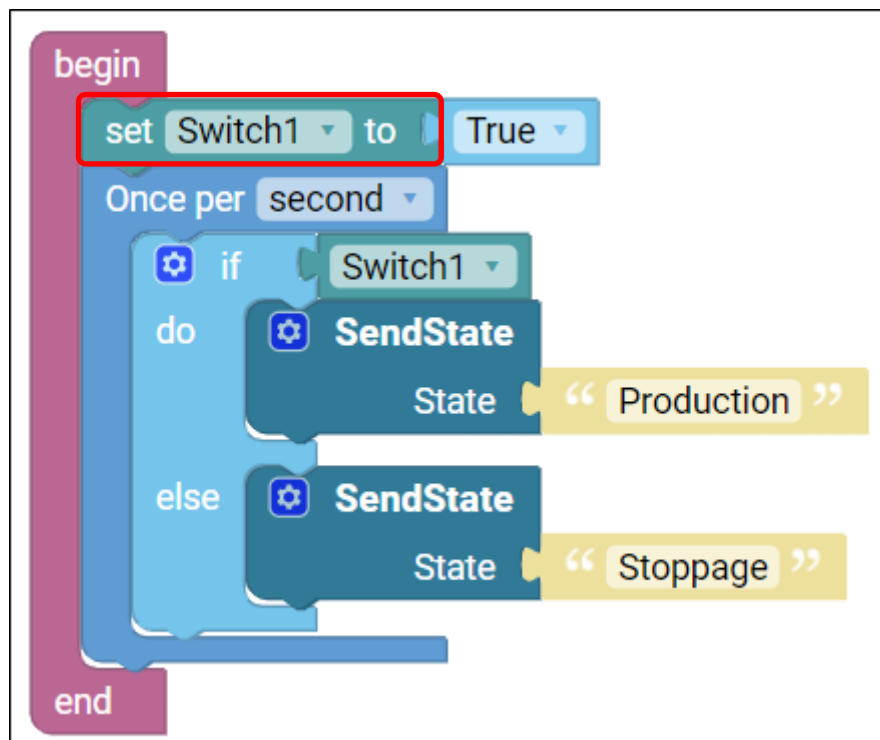
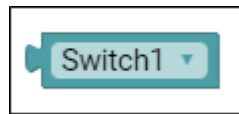


Fig.25: Example for Set [signal] to

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 Get Signal



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.

There are no restrictions to the input and output values.

Example

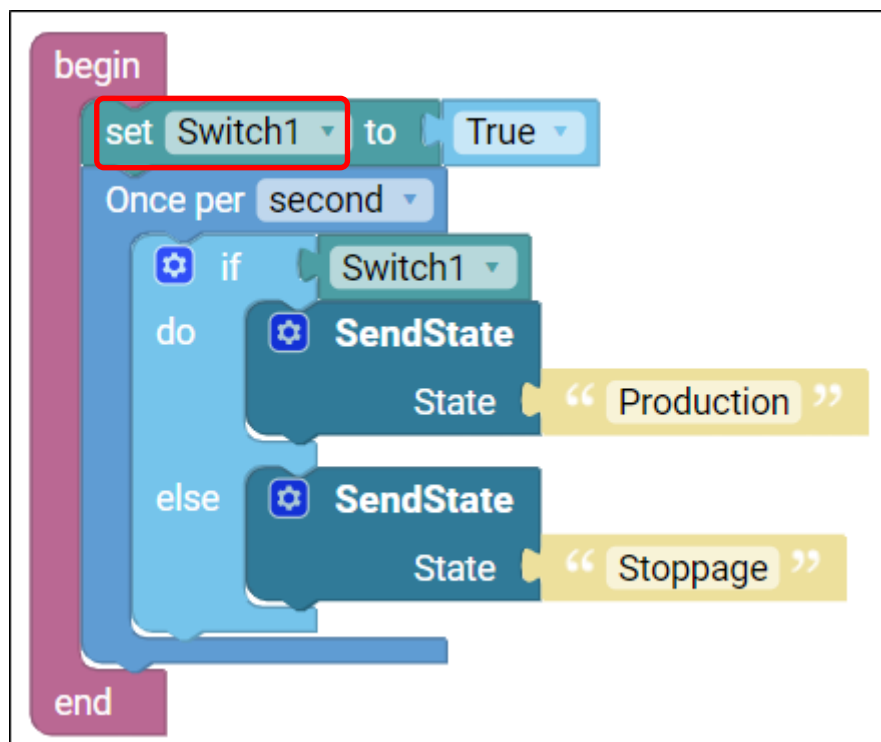
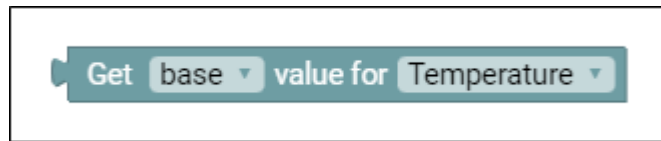


Fig. 26: Example for Get Signal

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



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. There are no restrictions to the input and output values.

Example

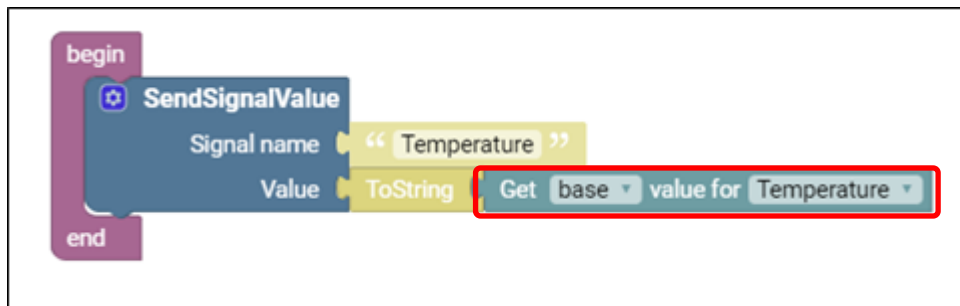


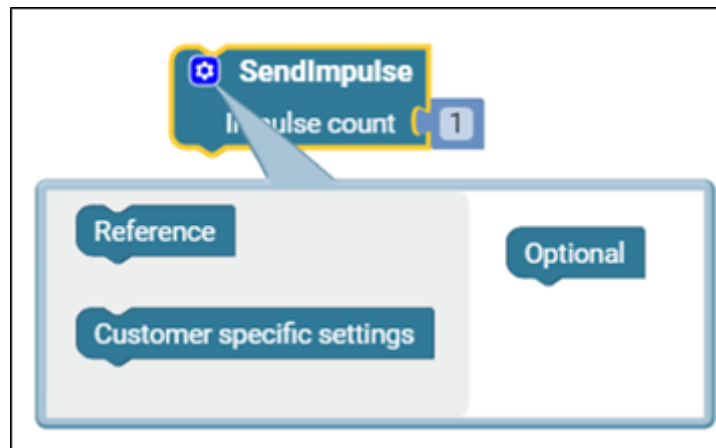
Fig. 27: Example for Get base value for

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 Events

Events send information packages to third-party systems. The content of these packages is defined in the Graphical Composition.

6.1 SendImpulse



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.

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

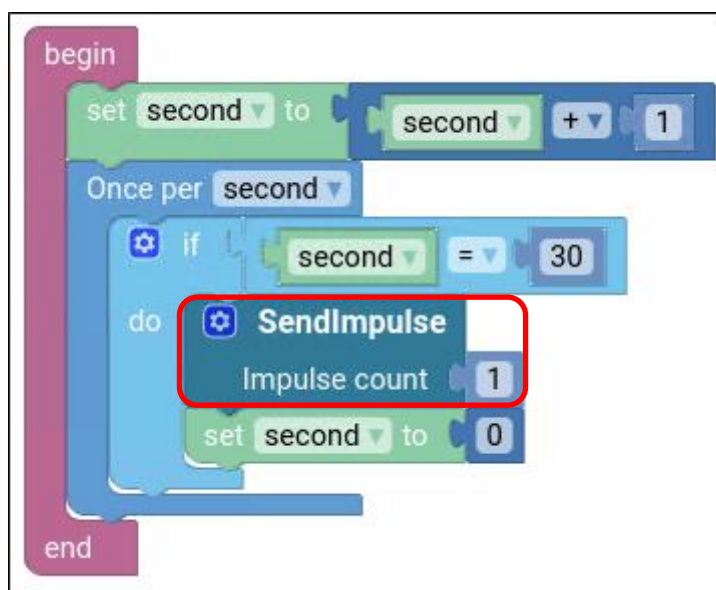
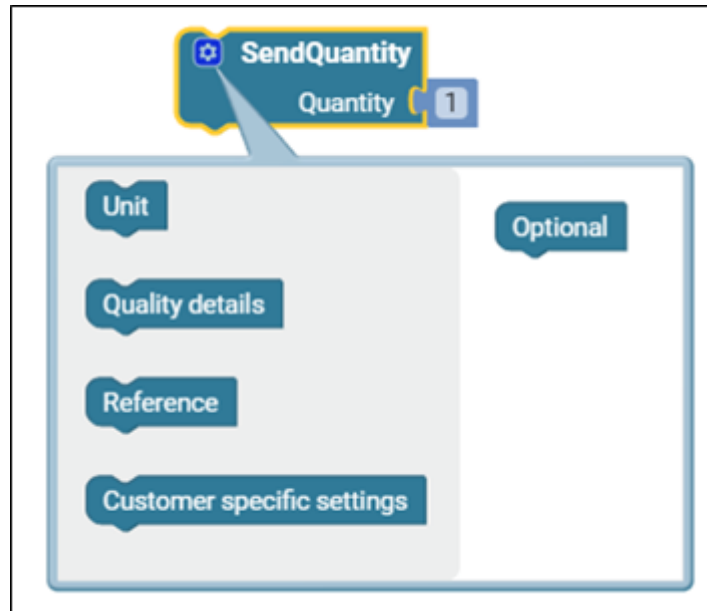


Fig. 28: Example for SendImpulse

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

6.2 SendQuantity



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

Units must first be defined as variables.

Only numbers can be used as input for **Quantity**.

All other input entries must be strings. There are no restrictions to the input and output values.

Example

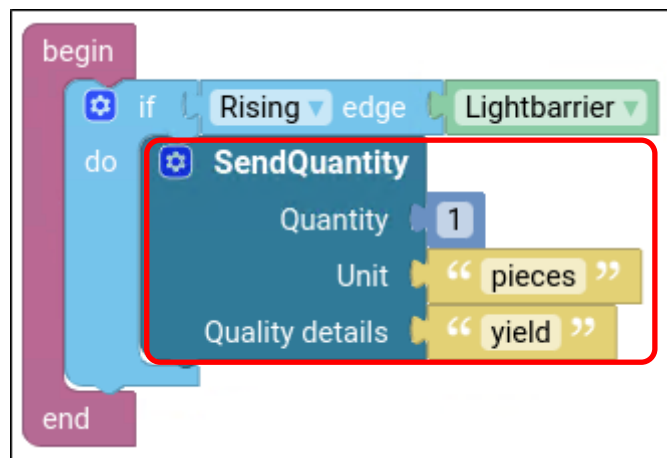
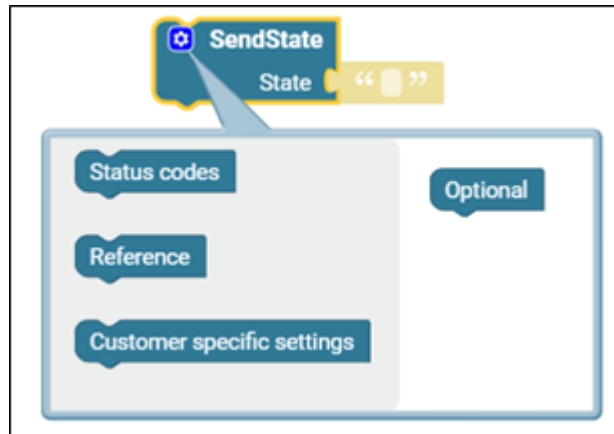


Fig. 29: Example for SendQuantity


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 detail) as yield.

6.3 SendState



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 corresponding content has been defined in step 3 of the Configuration Wizard. Refer to chapter 3.1 for more information.

-  In order to send **Status codes**, a list must be created. Refer to chapter 12 for more information.

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

Example

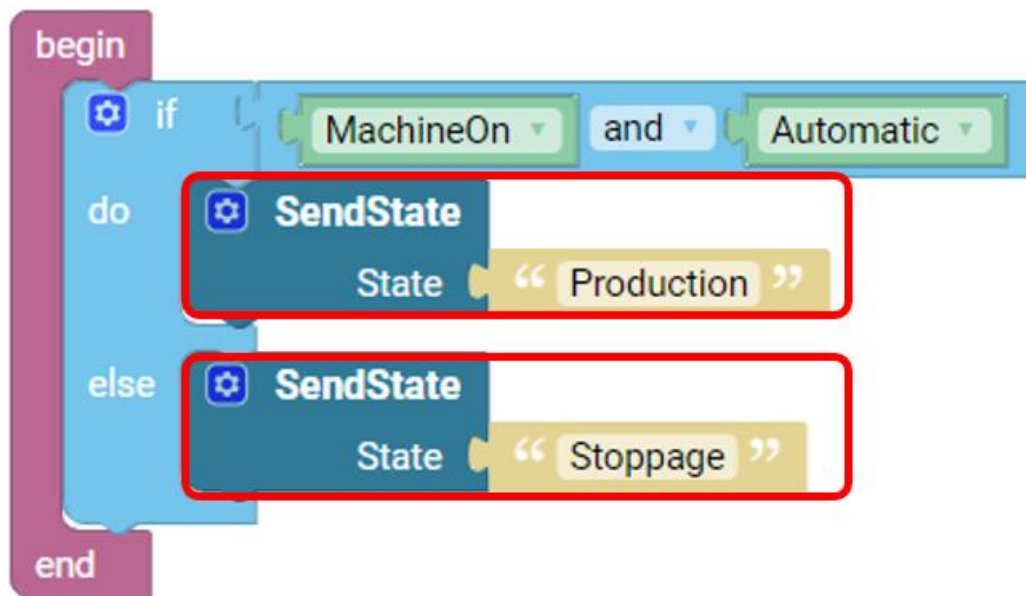
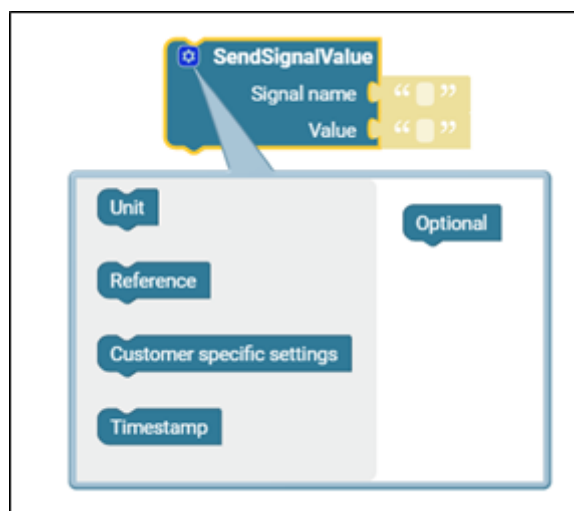


Fig. 30: Example for SendState

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



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.

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

Example

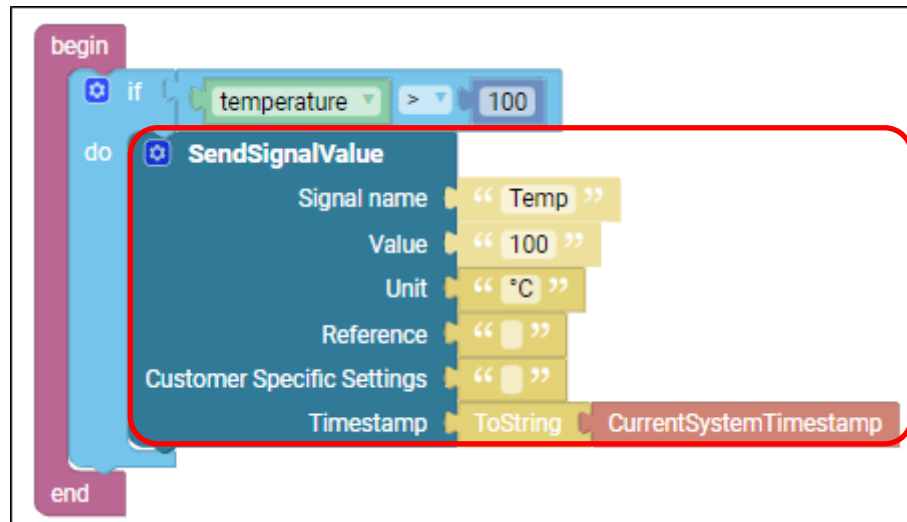


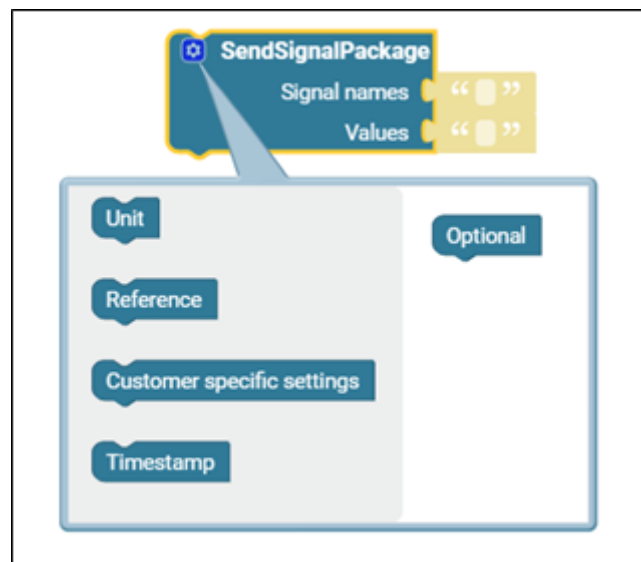
Fig. 31: Example for SendSignalValue

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



SendSignalPackage sends lists of signals. The contents originate from previously created lists.

The list of names may be extended by additional signals and matching signal values.

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

- i** In this case, the sequence must be observed:
The first entry in the list of signals must correspond to the first entry in the list of values.

Example

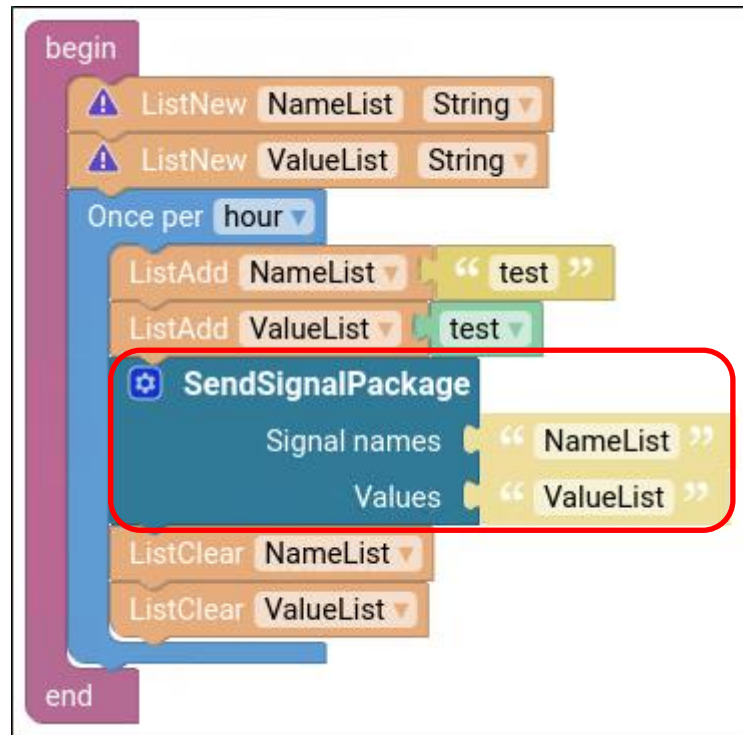
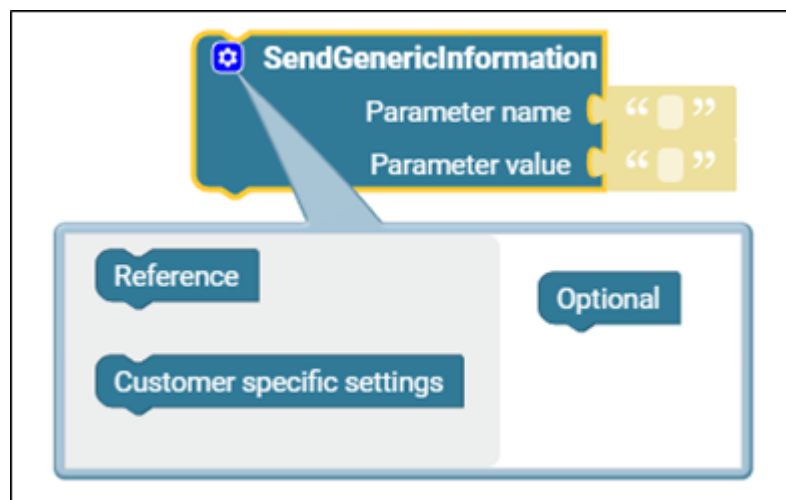


Fig. 32: Example for SendSignalPackage

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



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

Example

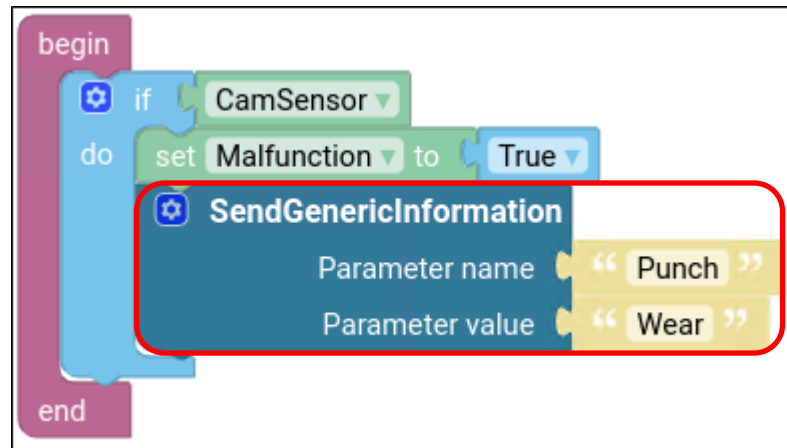
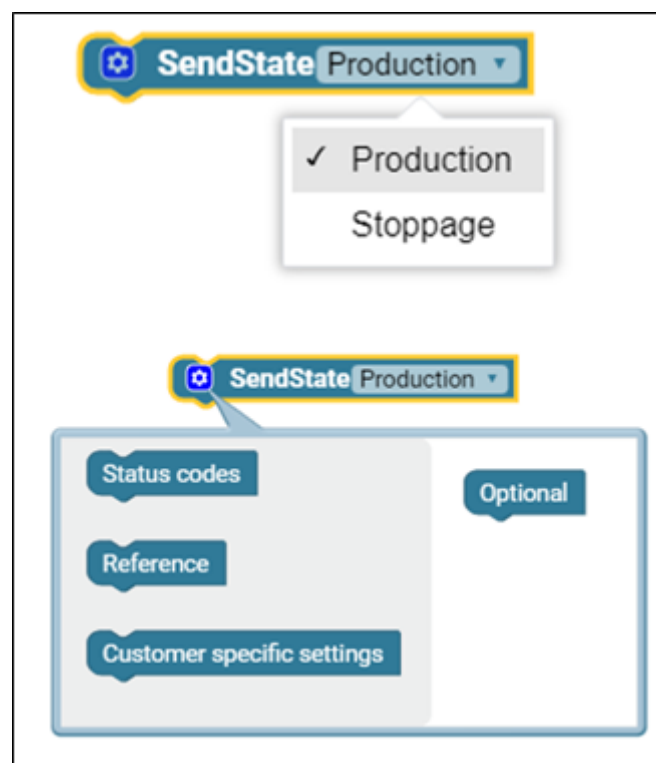


Fig. 33: Example for SendGenericInformation

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]



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. Refer to chapter 3.1 for more information.

- ❗ In order to send **Status codes**, a list must be created. Refer to chapter 12 for more information.

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

Example

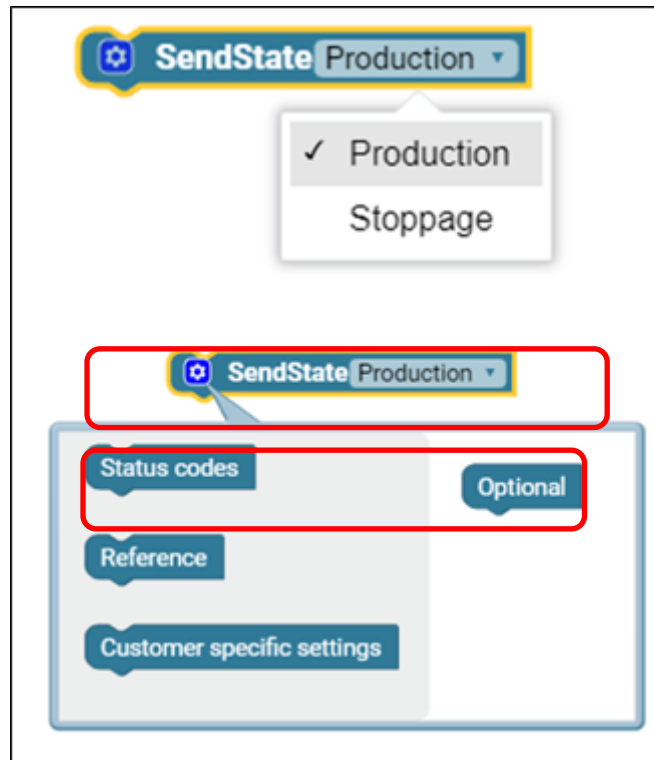
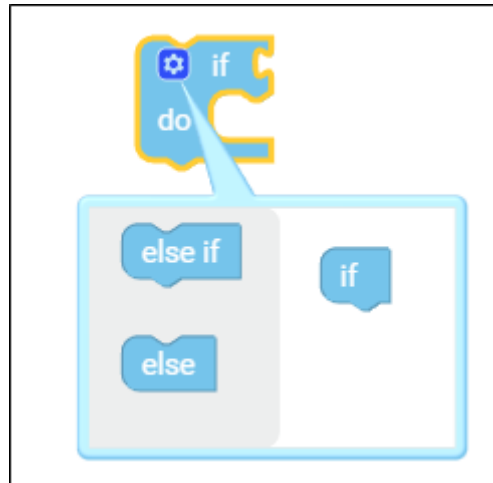


Fig. 34: Example for SendState [Selection]

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

7 Logical

7.1 If-do



This block implements the common 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. 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

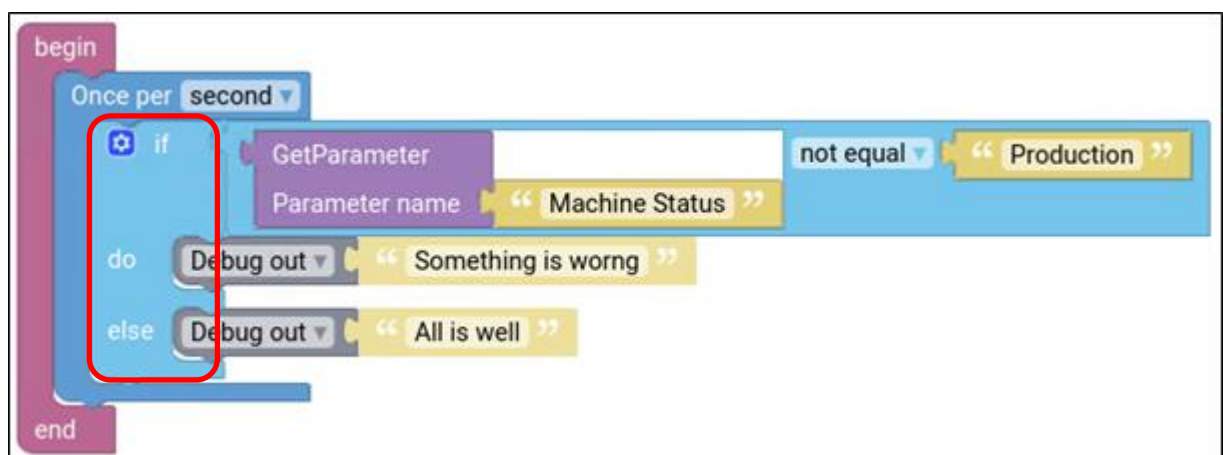


Fig. 35: Example for the If-do-Block

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: $=$ / \neq / $<$ / $>$ / \leq / \geq



Logical connectives like the $=$ 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 table contains their meanings:

$V1 = V2$	V1 equals V2
$V1 \neq V2$	V1 unequal to V2
$V1 > V2$	V1 is greater than V2
$V1 \geq V2$	V1 is greater than or equal to V2
$V1 < V2$	V1 is less than V2
$V1 \leq V2$	V1 is less than or equal to V2

Only numerical values (type Number) can be used as input. The output can only be boolean values.

Example

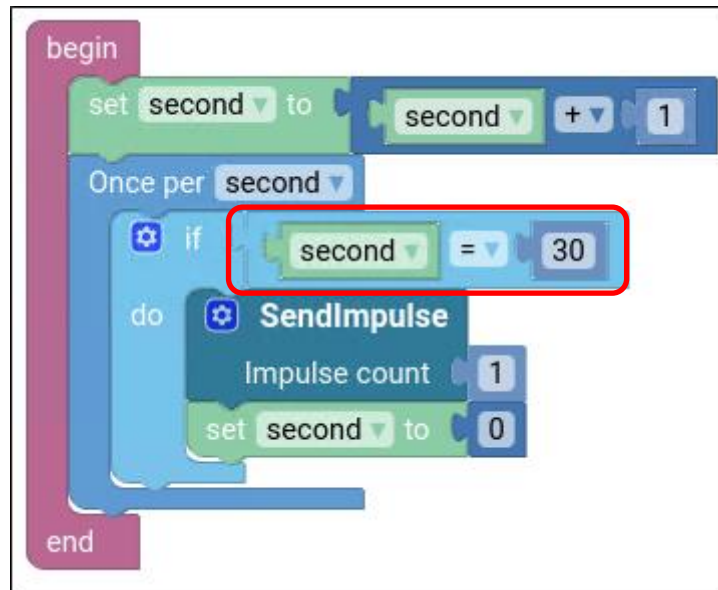


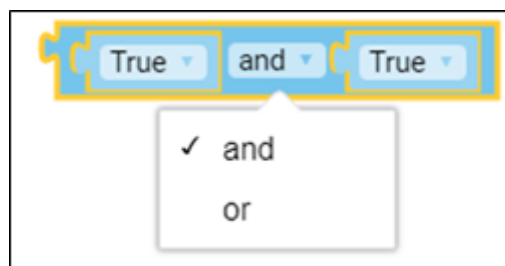
Fig. 36: Example for the = connective

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 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 Logical connective: and/or



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 and output values can only be boolean values.

Example

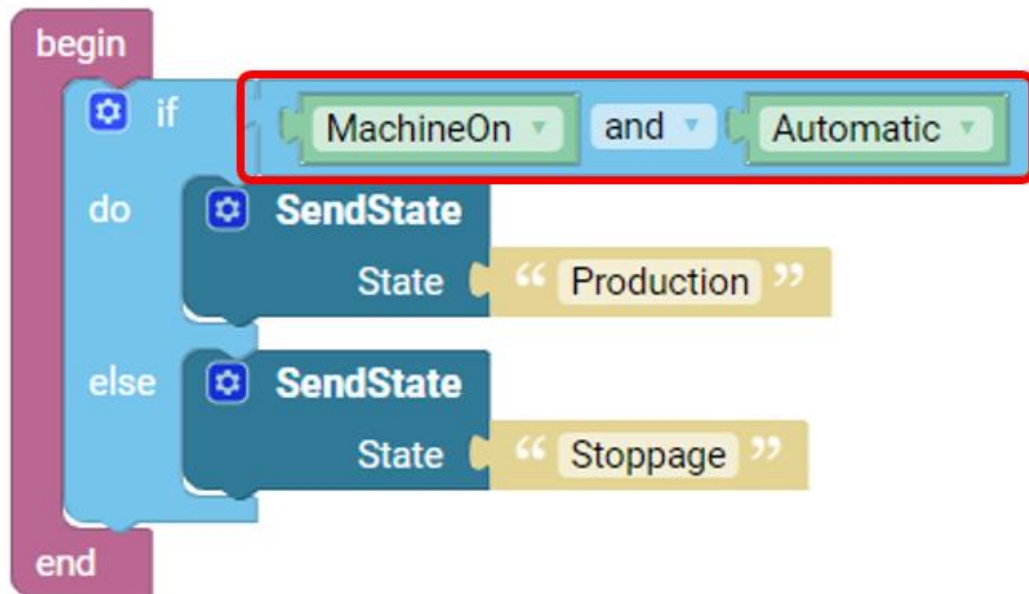
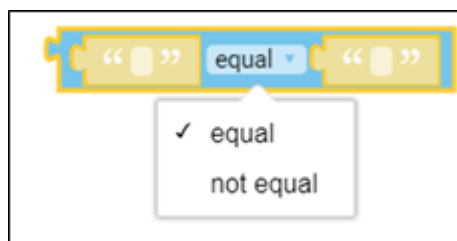


Fig. 37: Example for the “and” connective

In this example, the **SendState** block sends the status **Production** only if the machine is switched on (**MachineOn**) and is running in automatic mode (**Automatic**).

If only one of the two prerequisites applies, status **Stoppage** is sent.

7.4 Logical connective: equal/not equal



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.

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

Example

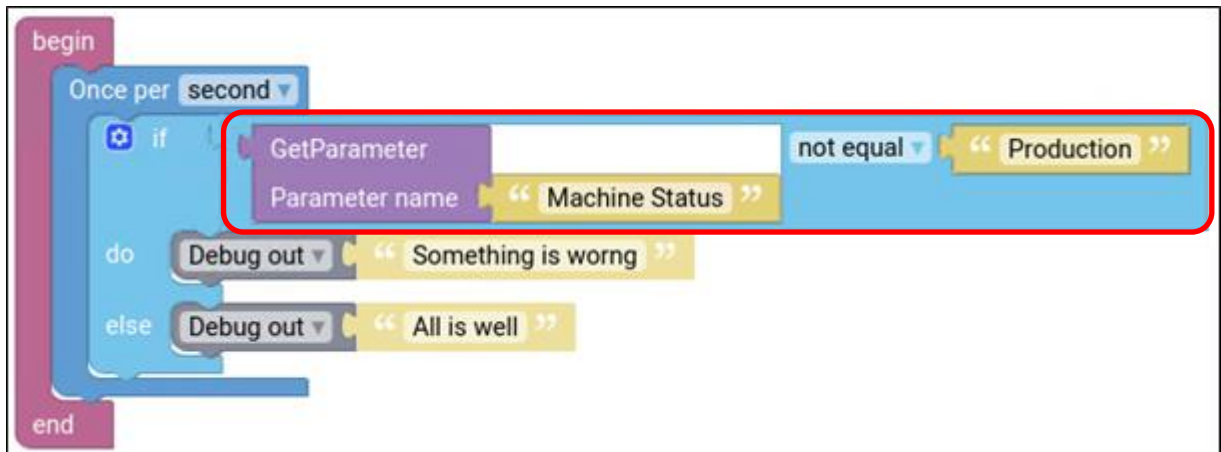
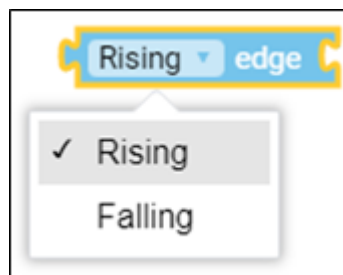


Fig. 38: Example for “not equal”

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 does, the message shall be **All is well**.

7.5 Rising/Falling edge



This block indicates that a variable or signal has changed from true (1) to false (0) or vice versa. The input can only be a boolean value.

Rising edge: At the beginning, the boolean value is false (0). **Rising edge** 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** 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.

Example



Fig. 39: 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



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 and output values can only be boolean values.

Example

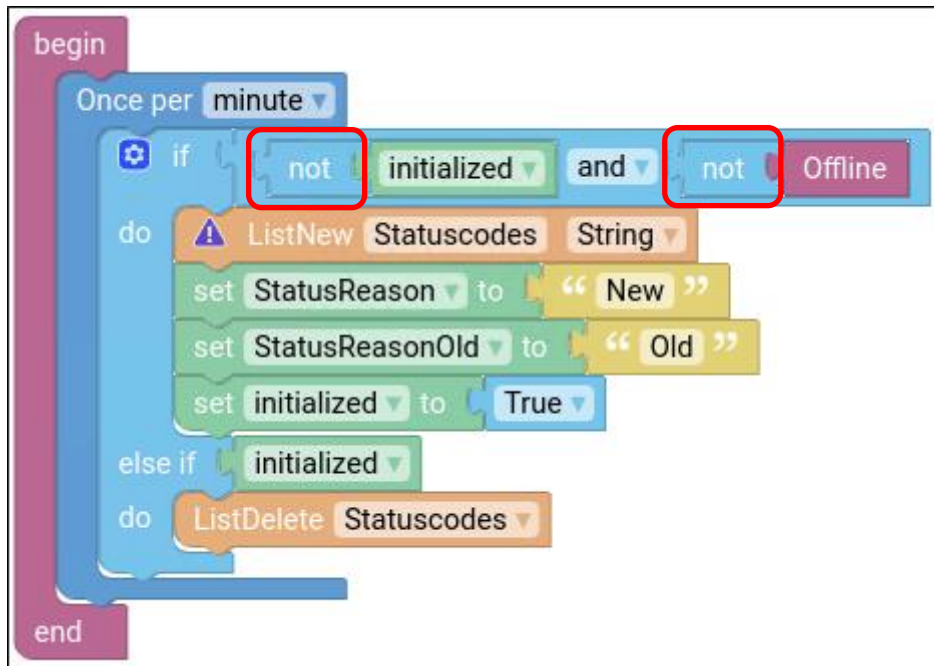


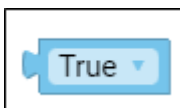
Fig. 40: Example for a “not” statement

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 statement



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.

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

Example

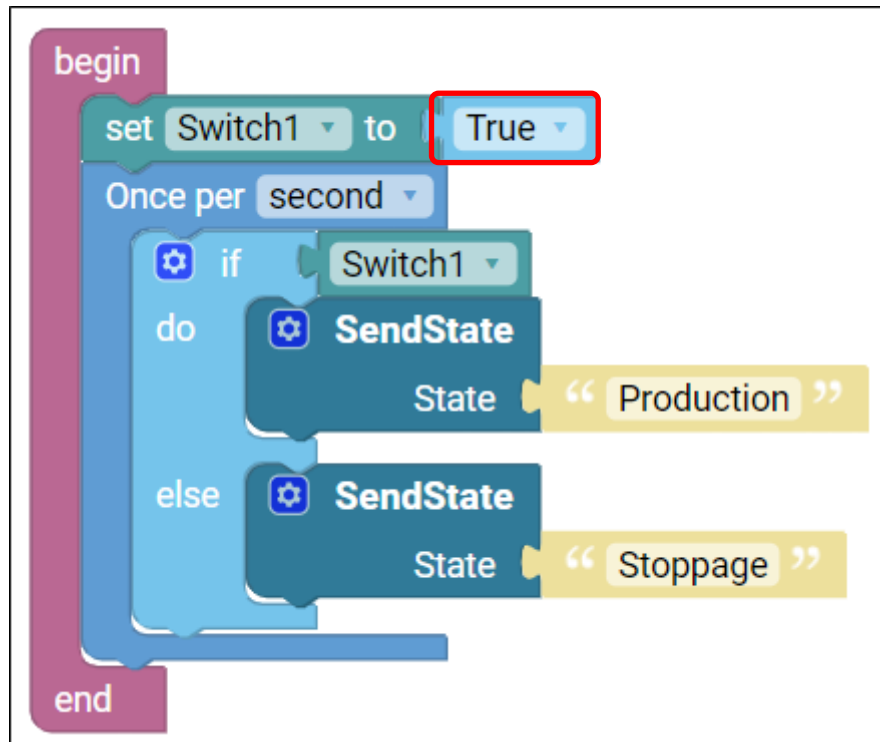
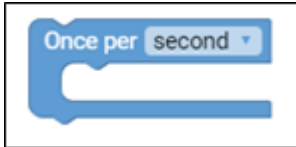


Fig. 41: Example for “true”

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

8.1 Once per



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

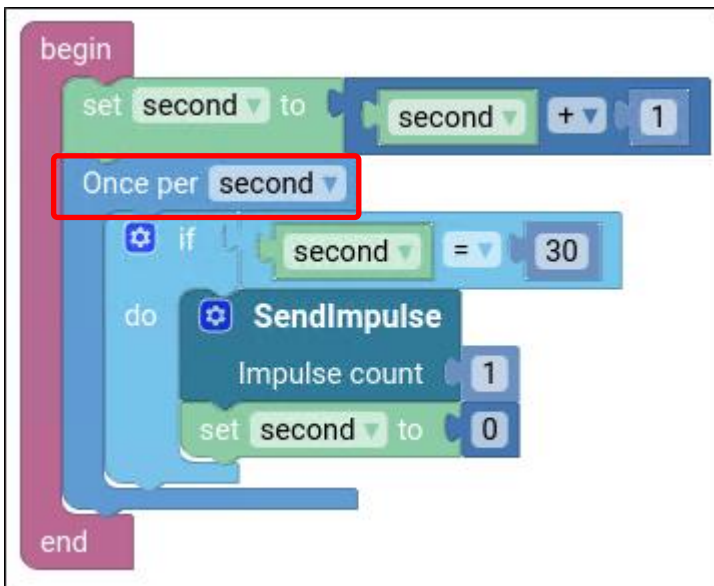


Fig. 42: Example for once per

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.

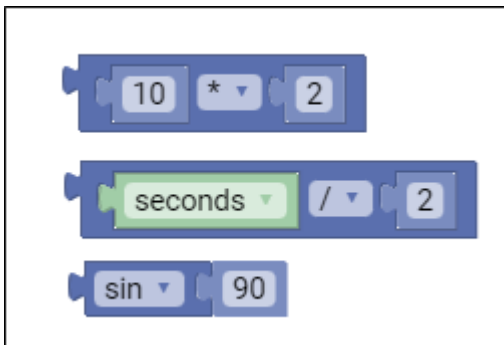
9 Arithmetic

9.1 Number field



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



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 and output values can only be numbers.

Example

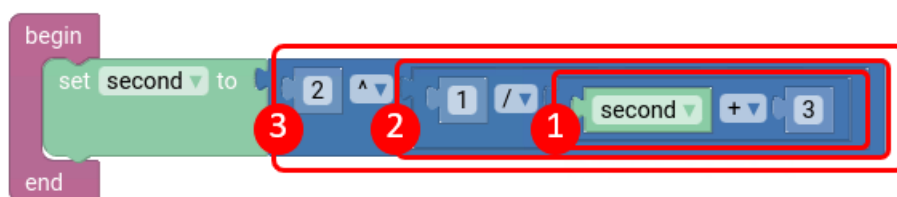


Fig. 43: Example for maths operations

A nested calculation indicates the factor. For understanding the calculation method it is important to follow an "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



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

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

Example

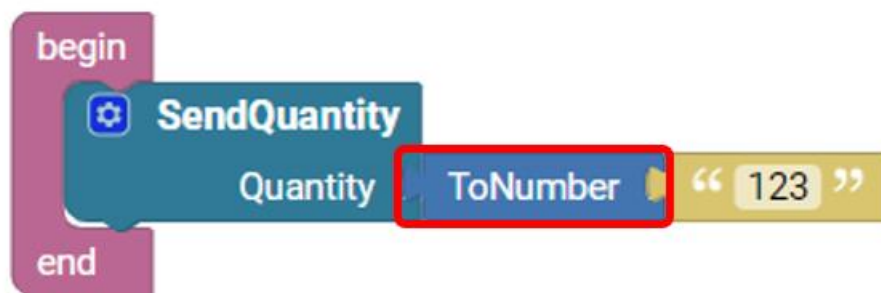
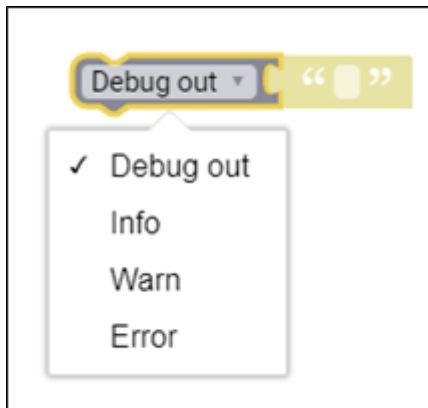


Fig. 44: Example for ToNumber

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 **SendQuantity** 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 Logging

10.1 Logging



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

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

Example

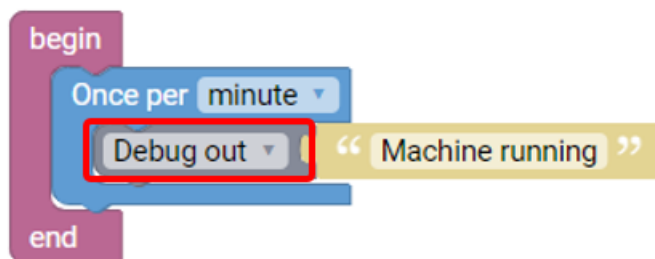


Fig. 45: Example for Debug out

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

11 Text

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

11.1String



Using these blocks, strings can be added by typing them in the quotes. There are no restrictions to the input. The output can only be string values.

Example

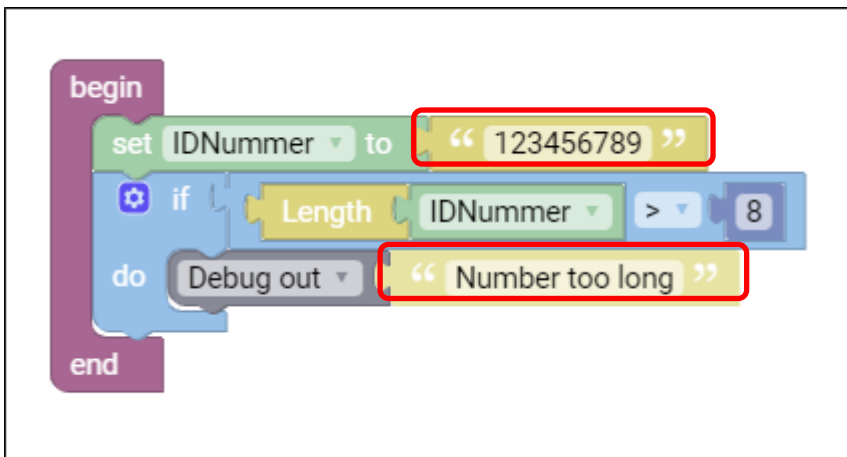
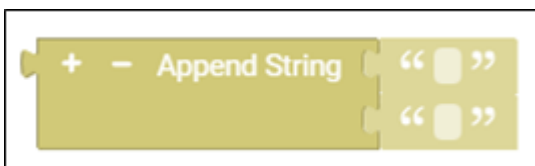


Fig. 46: Example for Length

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



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 and output values can only be strings.

Example

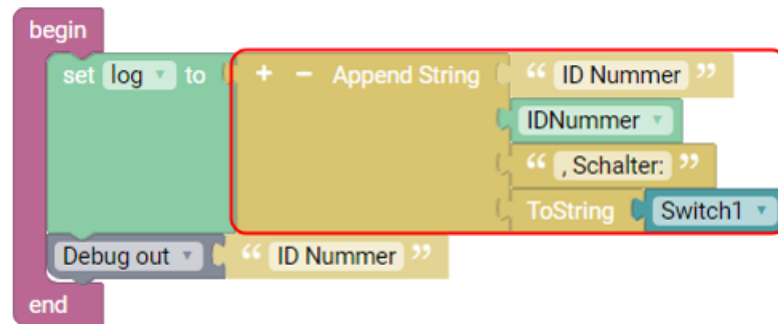
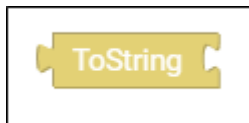


Fig. 47: Example for Append String

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.

11.3 ToString



ToString is used to convert numbers, or variables representing numbers, into a string. There are no restrictions to the input. The output can only be string values.

Example

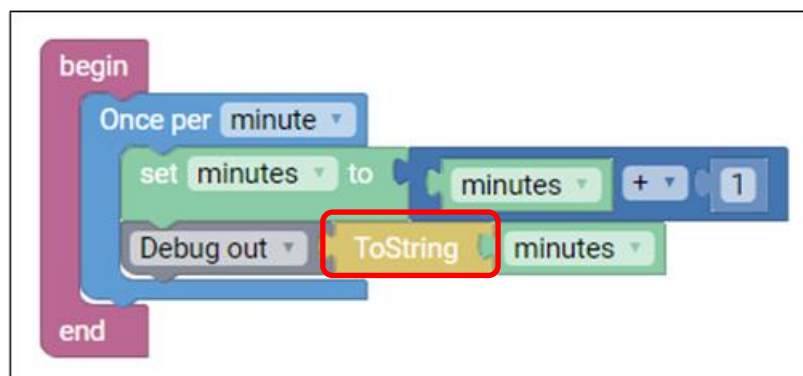
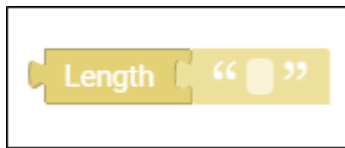


Fig. 48: Example for ToString

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.

11.4 Length



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.

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

Example

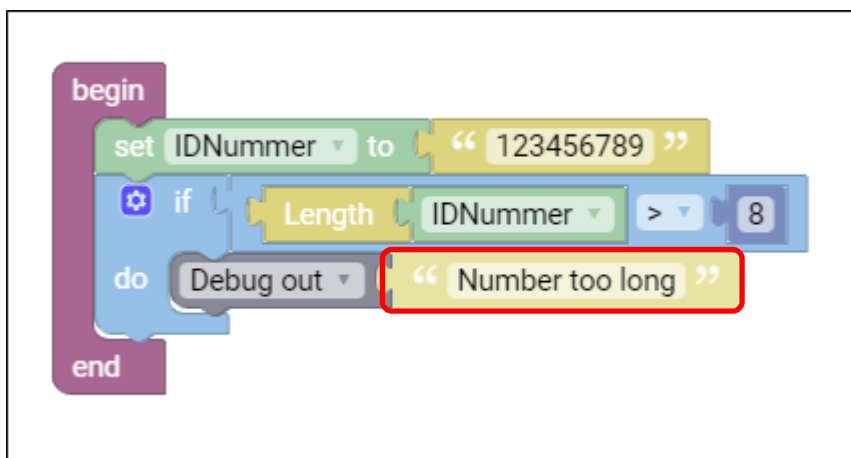
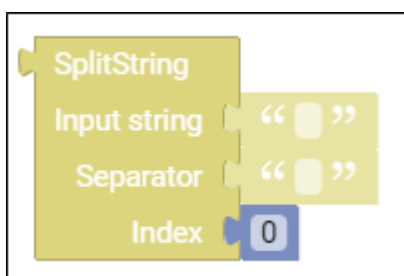


Fig. 49: Example for Length

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



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 **Seperator**, 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.

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. 50: 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.

11.6 FromAscii



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.

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

The ASCII table can be found in chapter 17.2 Ascii table, page 77.

Example

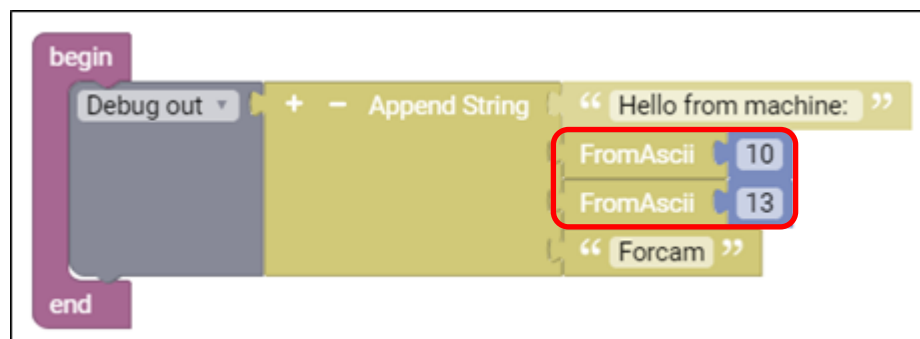


Fig. 51: Example for FromAscii

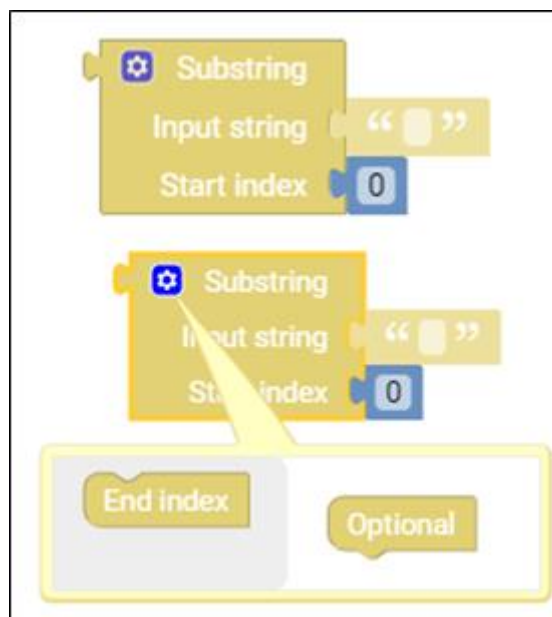
In this example, the text "Hello from machine:" shall be output followed by a paragraph mark and the text "Forcam".

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

11.7 Substring



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 indices handling, characters are counted starting from 0. The **End Index** is excluded.

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

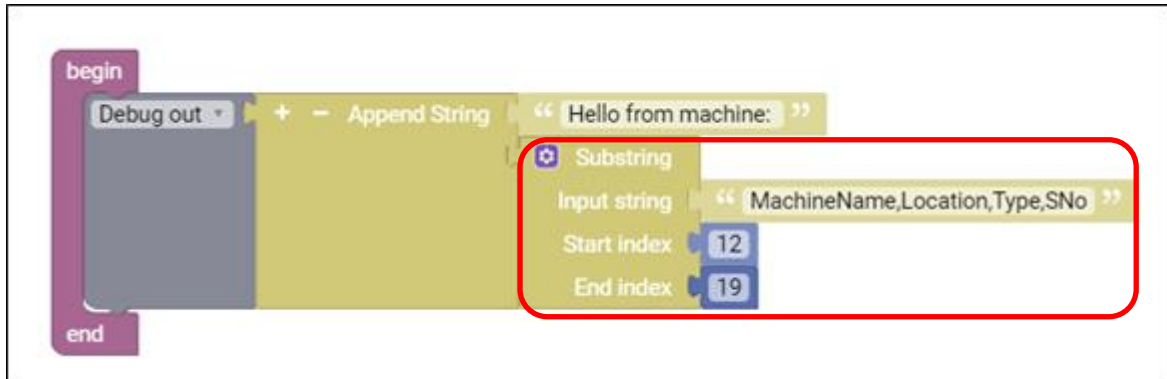


Fig. 52: Example for Substring

In this example, the location of the machine shall be output.

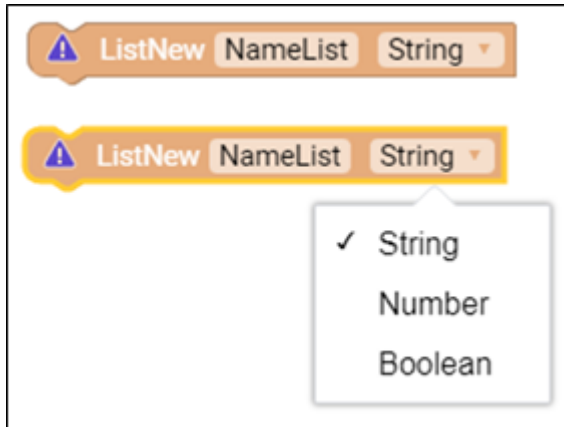
The **Append String** block first sets the text "Hello from machine:". **Input string** provides a list of asset properties. **Start index** specifies that the output starts at character 12. **End index** indicates that the output ends and includes character 19.

Because counting starts with 0 from the left, the **Location** property is output.

12 Lists

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

12.1 ListNew



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. Restrictions for the input are made via the selection.

Example

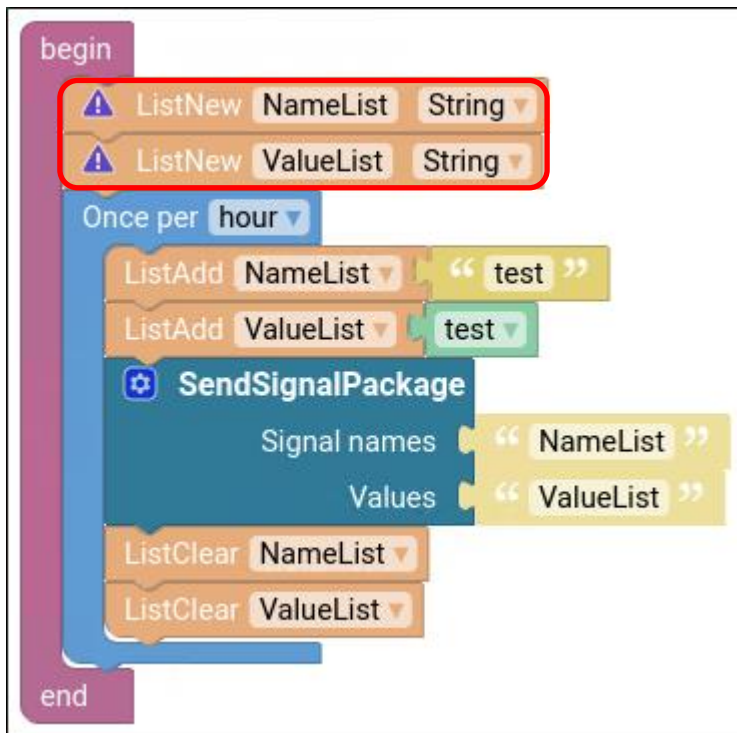
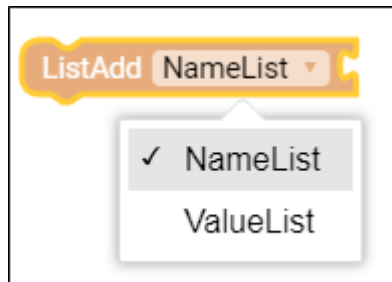


Fig. 53: Example for ListNew

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.

12.2 ListAdd



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. 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

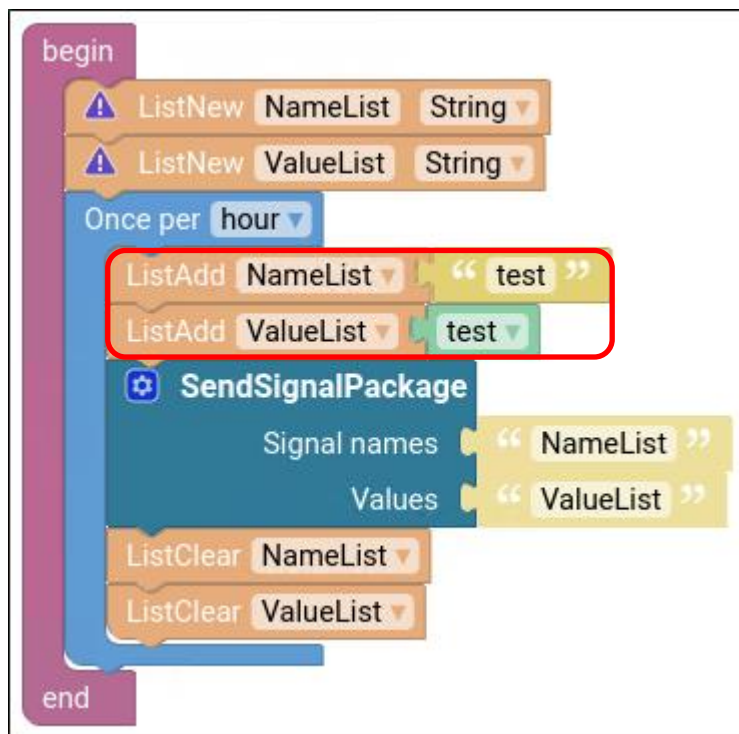
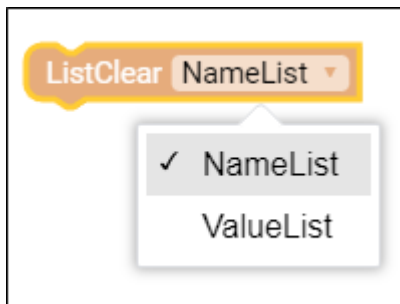


Fig. 54: Example for ListAdd

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



ListClear deletes the contents of a list.

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.

i 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.

Example

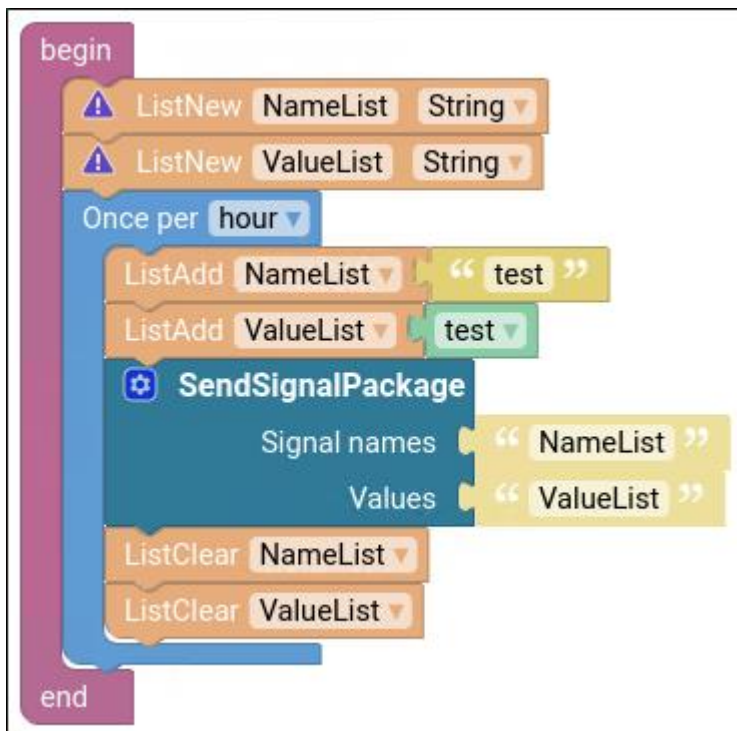
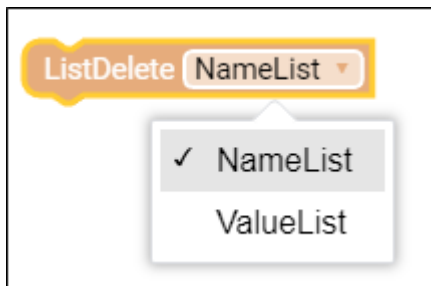


Fig. 55: Example for ListClear

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.

12.4 ListDelete



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

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.

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

Example

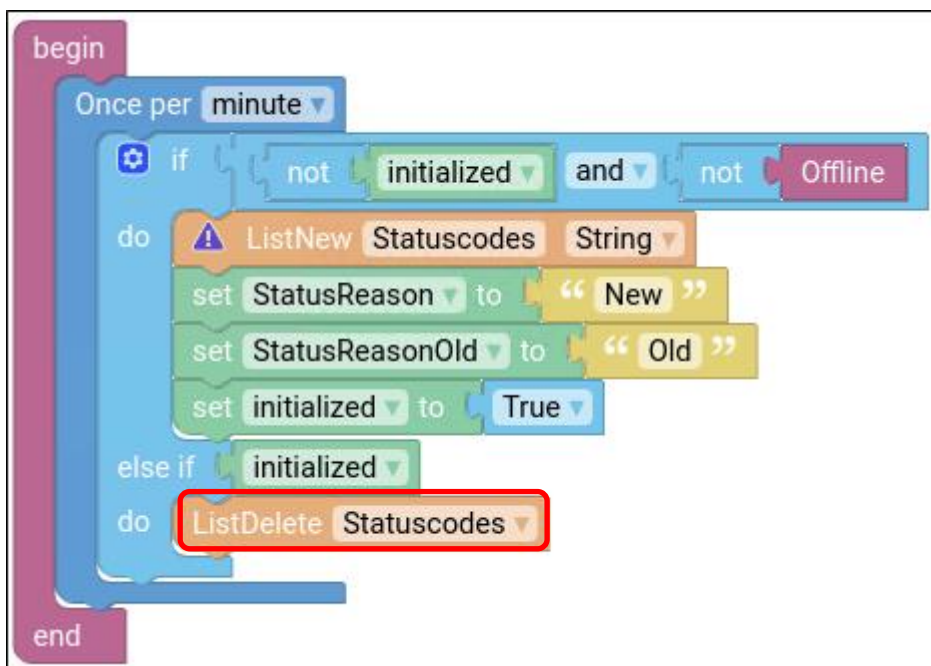


Fig. 56: Example for ListDelete

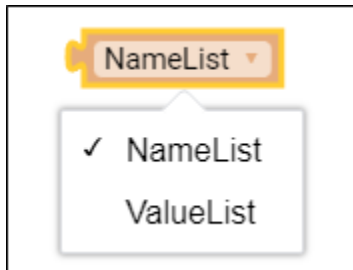
In this case, once a minute a check is performed to detect whether the machine is running for the first time.

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 GetList



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

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

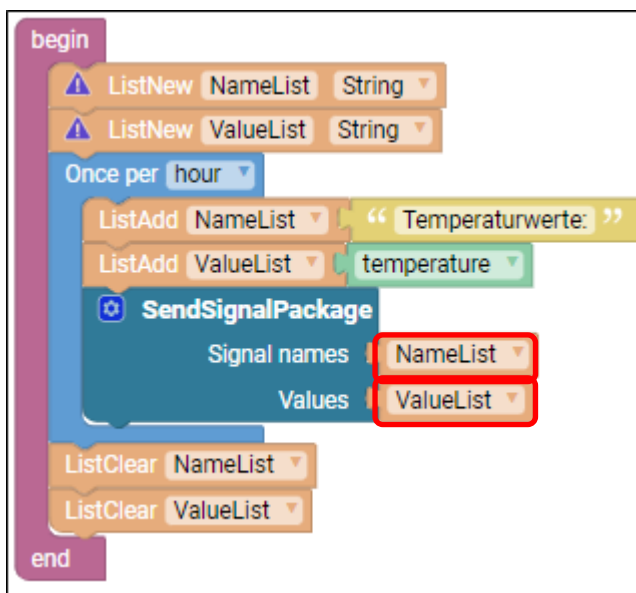


Fig. 57: Example for GetList

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 Date and time

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

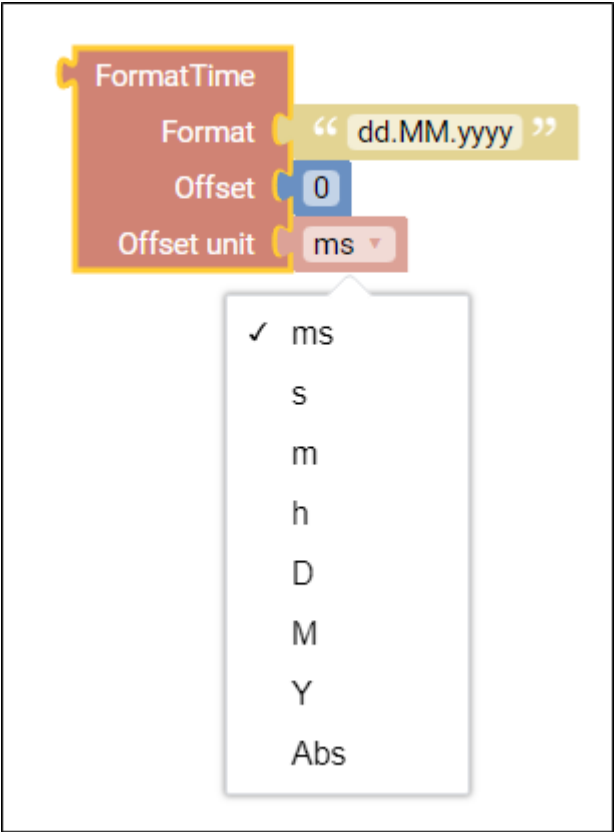
Since there are different abbreviations of time units, Table 2 lists the abbreviations used in the Graphical Composition.

Table 2: Time units used in Graphical Composition

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



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.

t **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.

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

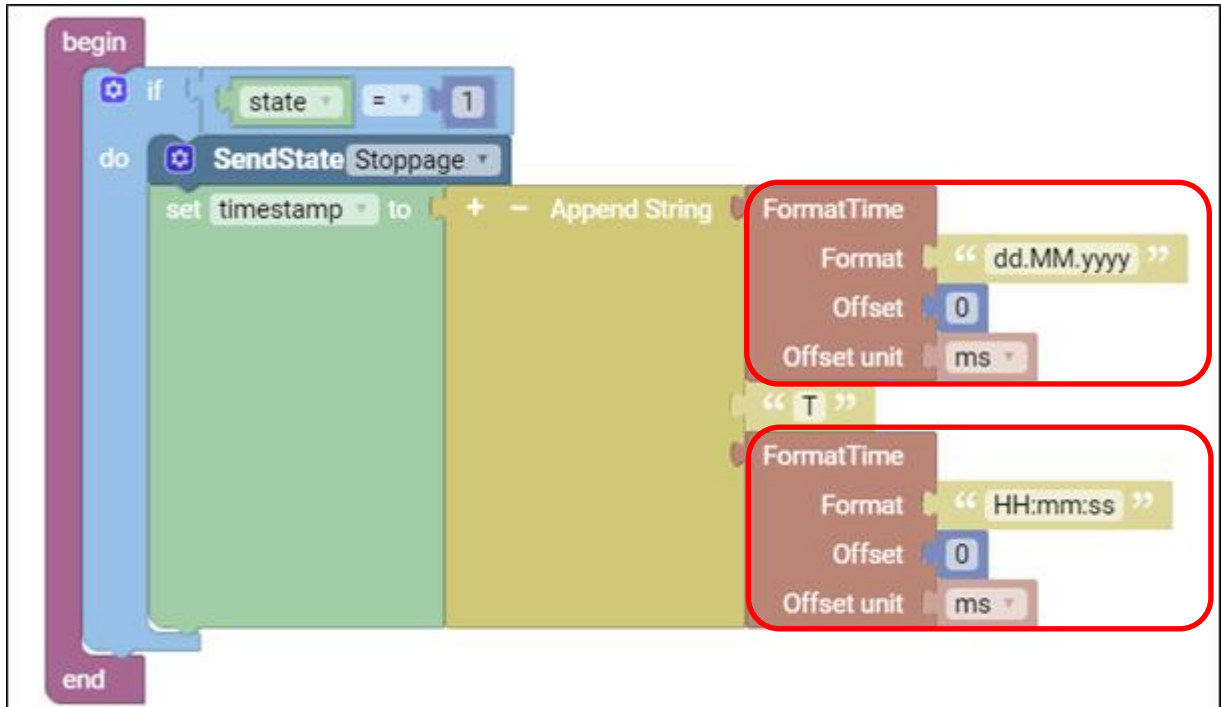


Fig. 58: Graphical example for FormatTime

In this example, a time stamp shall 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.

13.2 AtTime Do



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.

Only numbers can be used as input.

Example

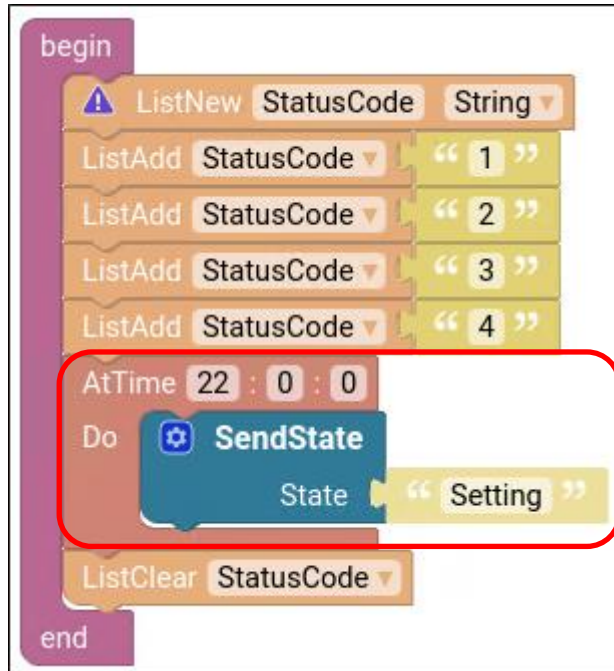
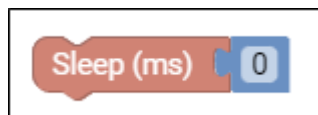


Fig. 59: Example for AtTime Do

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.

13.3 Sleep



The **Sleep** block waits for a certain period of time. The numeric value indicates the period (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. Only numbers can be used as input. There are no restrictions to the output.

Example

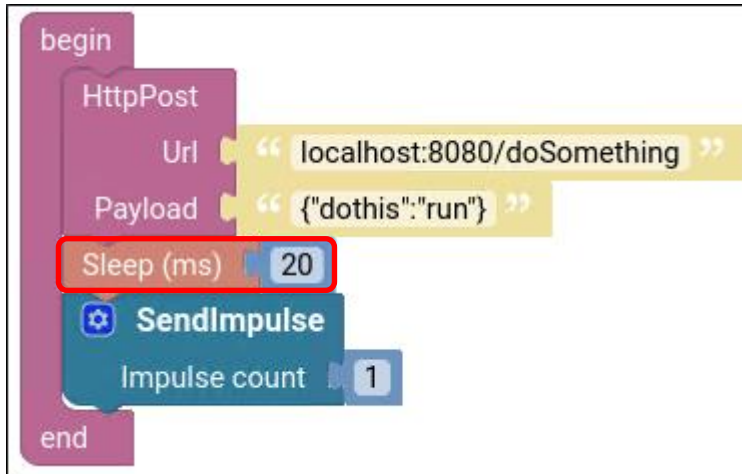
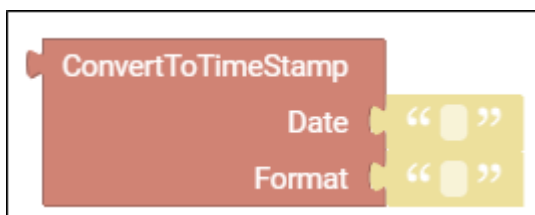


Fig. 60: Example for Sleep

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.

13.4 ConvertToTimeStamp



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 and output values can only be strings.

Example

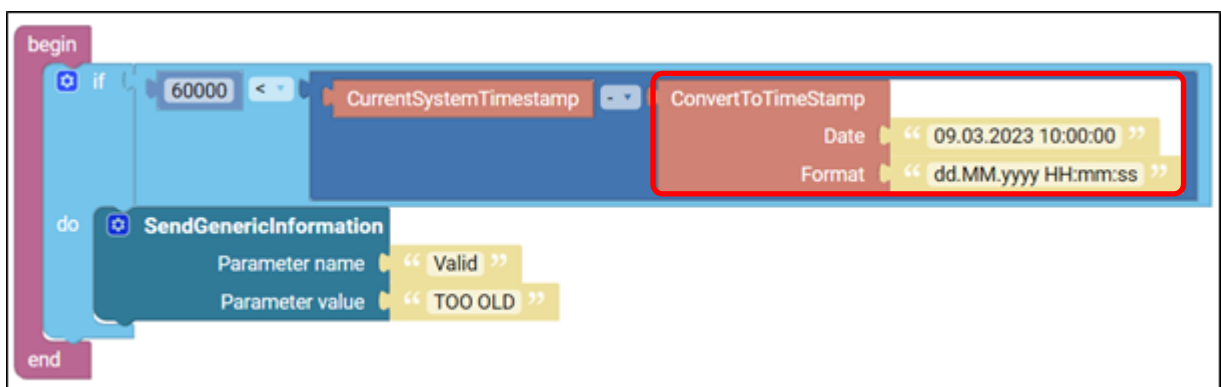
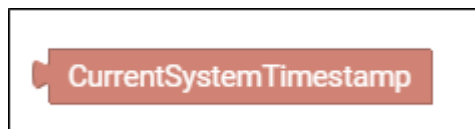


Fig. 61: Example for ConvertToTimeStamp

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 **SendGenericInformation** block. This message contains the information that the received time stamp is outdated.

13.5 CurrentSystemTimestamp



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

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

Example

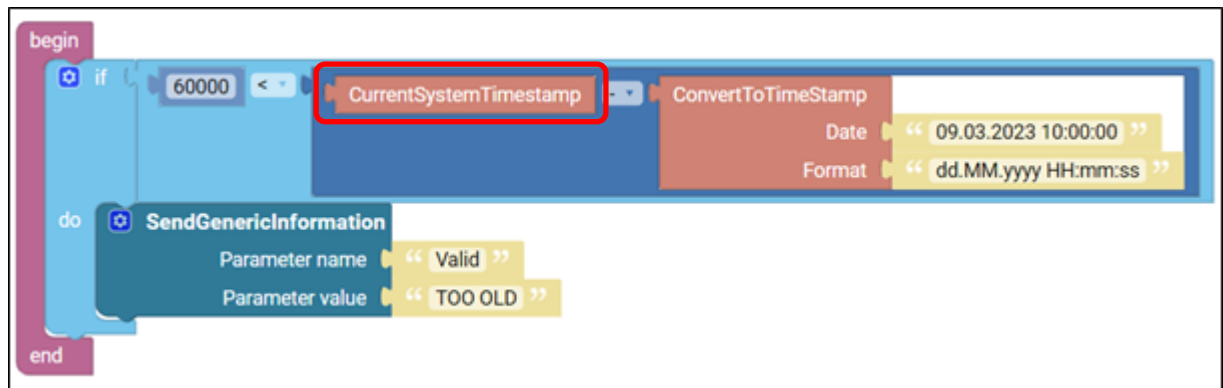


Fig. 62: Example for CurrentSystemTimestamp

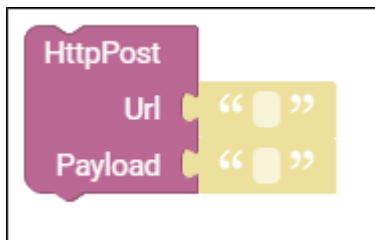
In this example, two different points in time shall be compared.

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 Misc

Misc means miscellaneous. This chapter summarizes important blocks with various functions.

14.1 HttpPost



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

Example

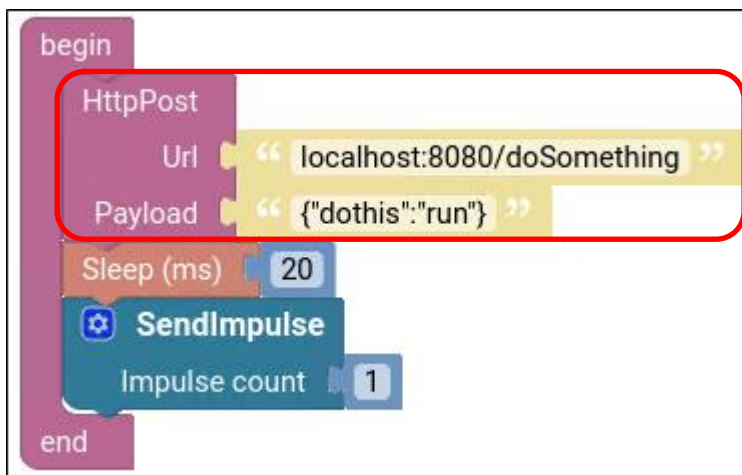
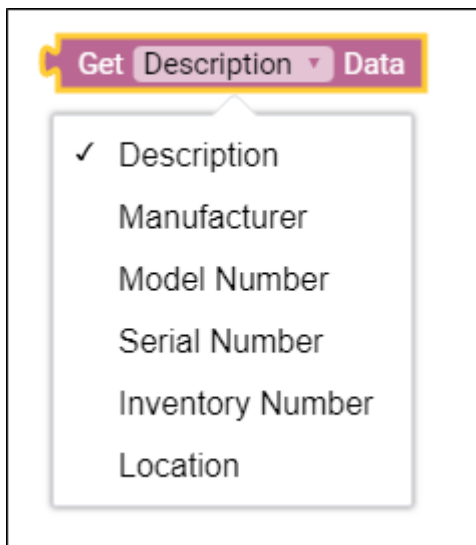


Fig. 63: Example for HttpPost

In this example, a server communication endpoint shall be called. The `url` and `payload` to be used for 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.

14.2 Get [specific] Data



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. Refer to chapter 3.1 for more information.

These parameters are automatically added to the drop-down menu.

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

Example

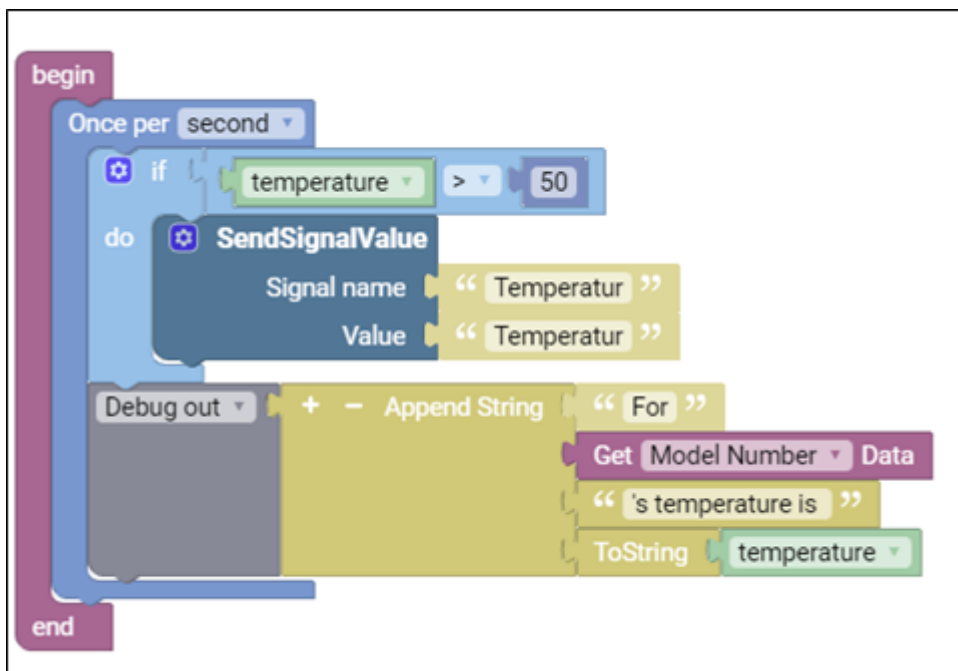
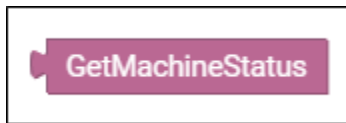


Fig. 64: Example for Get [specific] Data

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.

14.3 GetMachineStatus



GetMachineStatus outputs the current machine status.

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

Example

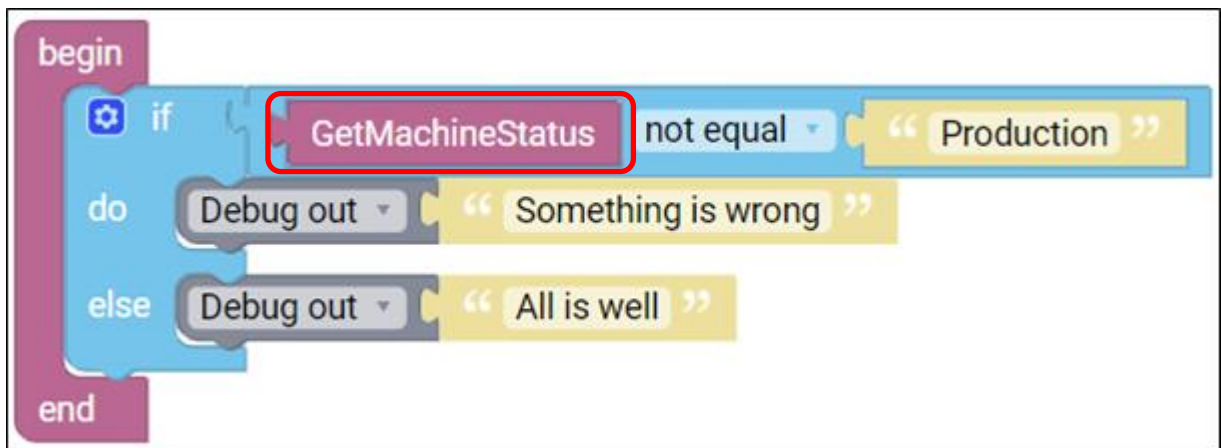


Fig. 65: Example for GetMachineStatus

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.

14.4 Offline



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

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

Example

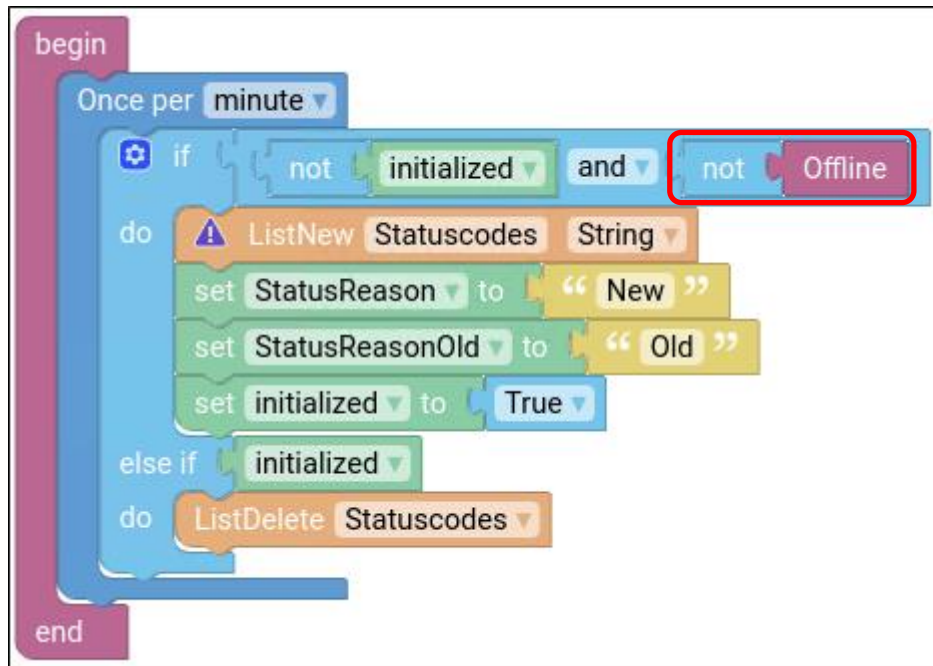


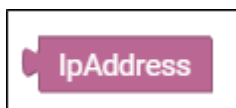
Fig. 66: Example for Offline

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



The **IPAddress** block noutputs 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.

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

Example

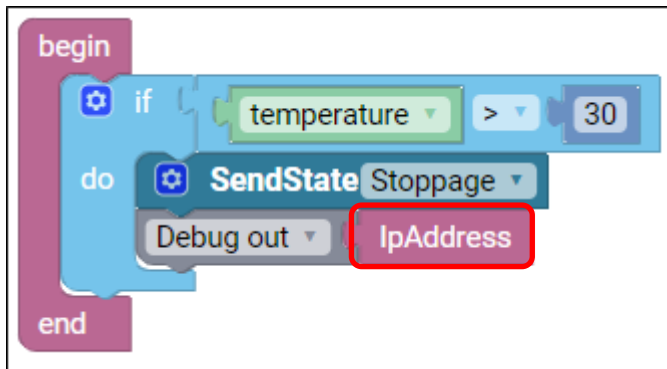
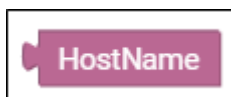


Fig. 67: Example for IPAddress

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



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.

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

Example

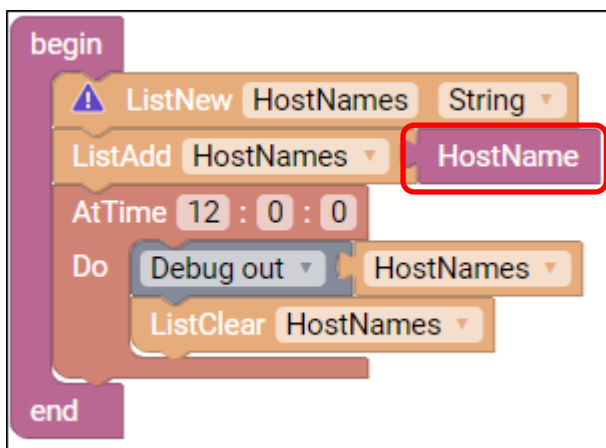
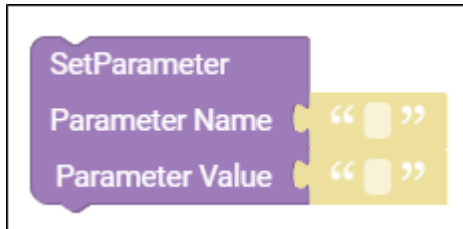


Fig. 68: Example for HostName

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 Business Parameters

15.1 SetParameter



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. Refer to chapter 3.1 for more information.

If an already defined parameter is to be used, the **GetParameter** block is used (see chapter 15.2). Only strings are possible as input values. There are no restrictions to the output.

Example

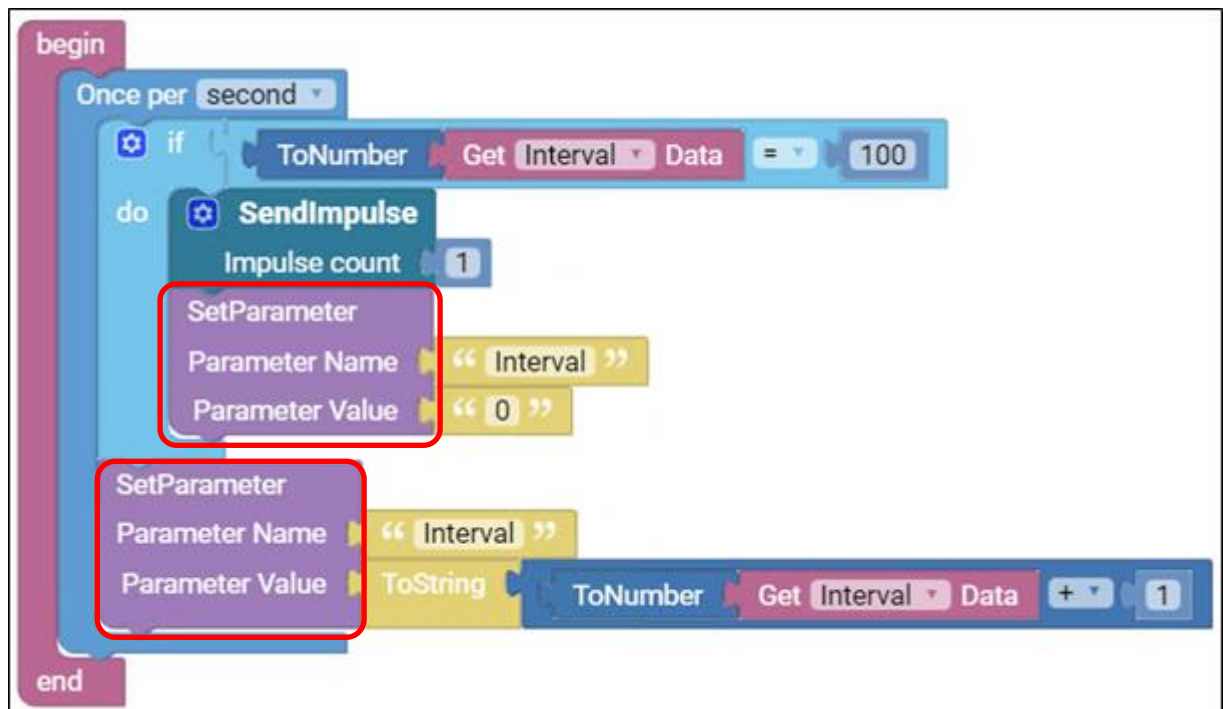
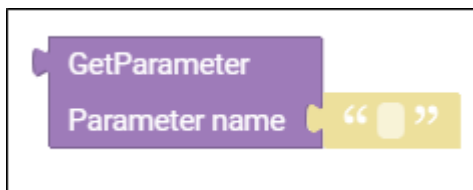


Fig. 69: Example for SetParameter

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.

15.2 GetParameter



The **GetParameter** block pulls the value of a parameter. Input and output values can only be strings.

Example

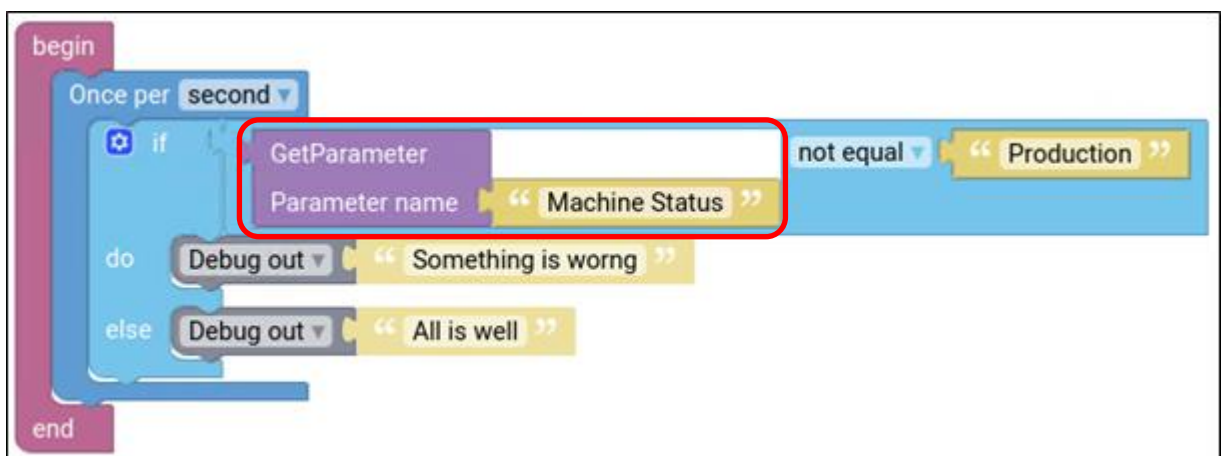
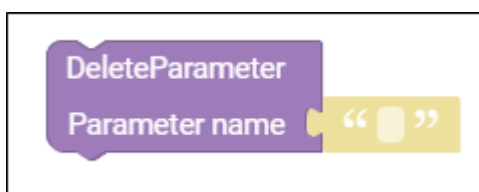


Fig. 70: Example for GetParameter

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**).

15.3 DeleteParameter



The **DeleteParameter** block resets the parameter value in the database to 0. Only strings are possible as input values. There are no restrictions to the output.

Example

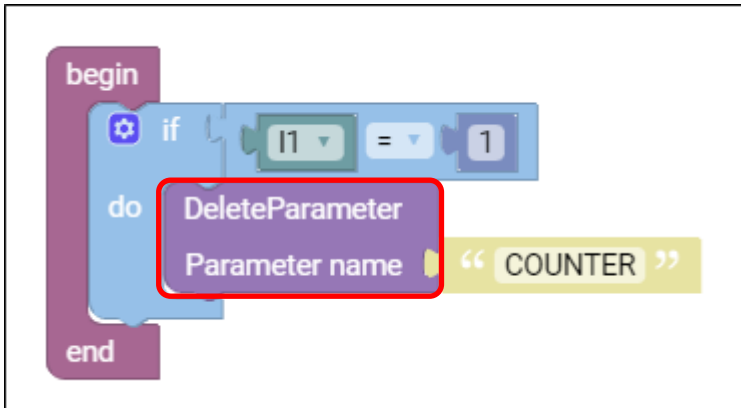
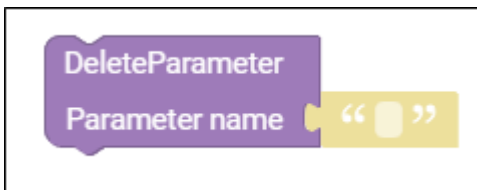


Fig. 71: Example for DeleteParameter


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



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

There are no restrictions to the input and output values.

 All parameters already used will also be reset to zero.

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

17.1 Parameter 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 Impulse count Reference Customer specific settings	 Optional Optional	Number String String	N/A N/A N/A
SendQuantity Quantity Unit Quality details Reference Customer specific settings	 Optional Optional Optional Optional	Number String String String String	N/A N/A N/A N/A N/A
SendState State Status codes Reference Customer specific settings	 Optional Optional Optional	String String String String	N/A N/A N/A N/A
SendSignalValue Signal name Value Unit Reference Customer specific settings Timestamp	 Optional Optional Optional Optional	String String String String String String	N/A N/A N/A N/A N/A N/A
SendSignalPackage Signal name Value Unit Reference Customer specific settings	 Optional Optional Optional	String String String String String String	N/A N/A N/A N/A N/A N/A
SendGenericInformation Parameter name Parameter value Reference	 Optional	String String String	N/A N/A N/A

Customer specific settings	Optional	String	N/A
SendState			
Status codes	Optional	String	N/A
Reference	Optional	String	N/A
Customer specific settings	Optional	String	N/A
Logical			
If-do			
If		Boolean	N/A
Else if	Optional	Boolean	N/A
Else	Optional	Boolean	N/A
Do		Any	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
Length		String	Number
SplitString			
Input string		String	String
Separator		String	String
Index		Number	Number
FromAscii		Number	String
Substring			
Input string		String	String
Start index		Number	N/A
End index	Optional	Number	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		String	String
Format		String	String
Offset		Number	String
Offset unit		Drop-down menu	String
AtTime Do		Number	N/A
Sleep		Number	N/A
ConvertToTimeStamp			Long
Date		String	String
Format		String	String
CurrentSystemTimestamp		N/A	Long
Misc			
HttpPost			
Url		String	N/A
Payload		String	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			
Paramter name		String	N/A
Parameter value		String	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 Start of heading	Beginning of the header
2	STX Start of Text	Beginning of a text part
3	ETX End of text	End of a text part
4	EOT End of transmission	Completion of a transmission
5	ENQ Enquiry	A request for a response from the receiving station
6	ACK Acknowledge	Confirmation
7	BEL Bell	Generates an audible signal
8	BS Backspace	Moves the cursor one position to the left and removes the character at this position
9	TAB Horizontal tab	Tabulator for horizontal indentation of the next text character
10	LF Line feed	Line break
11	VT Vertical tab	Tabulator for horizontal indentation of the next text character
12	FF Form feed	Page jump
13	CR Carriage return	Positions the cursor at the beginning of a line
14	Shift out	Moves the cursor out
15	SI Shift in	Moves the cursor inside
16	DLE Data link escape	Shift character
17	DC1 Device control 1	Device-specific function - often used as XON (continue transmission)
18	DC2 Device control 2	Device-specific function
19	DC3 Device control 3	Device-specific function - often used as XOFF (pause transmission)
20	DC4 Device control 4	Device-specific function

Dec	Char	Description
21	NAC Negative acknowledge	Negative confirmation
22	SYN Synchronous idle	In synchronous data transmissions, enables synchronization even in the absence of signals to be transmitted
23	ETB End of trans. block	Indicates the end of a data block
24	CAN Cancel	Cancel
25	EM End of medium	Indicates the end of a medium.
26	SUB Substitute	Replace
27	ESC Escape	Cancels an activity
28	FS File separator	Separation of main groups
29	GS Group separator	Group separation
30	RS Record separator	Subgroup separation
31	US Unit separator	Separation of parts of a group
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	

Dec	Char	Description
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	

Dec	Char	Description
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	~	
127	DEL Delete	Delete the last character