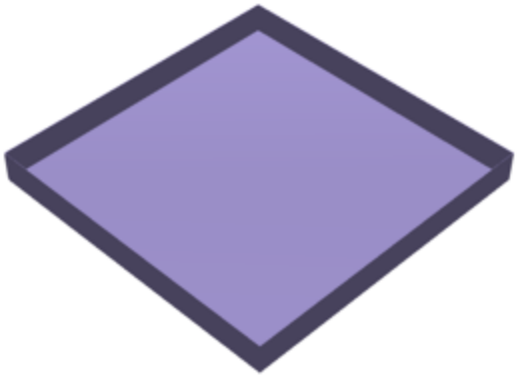
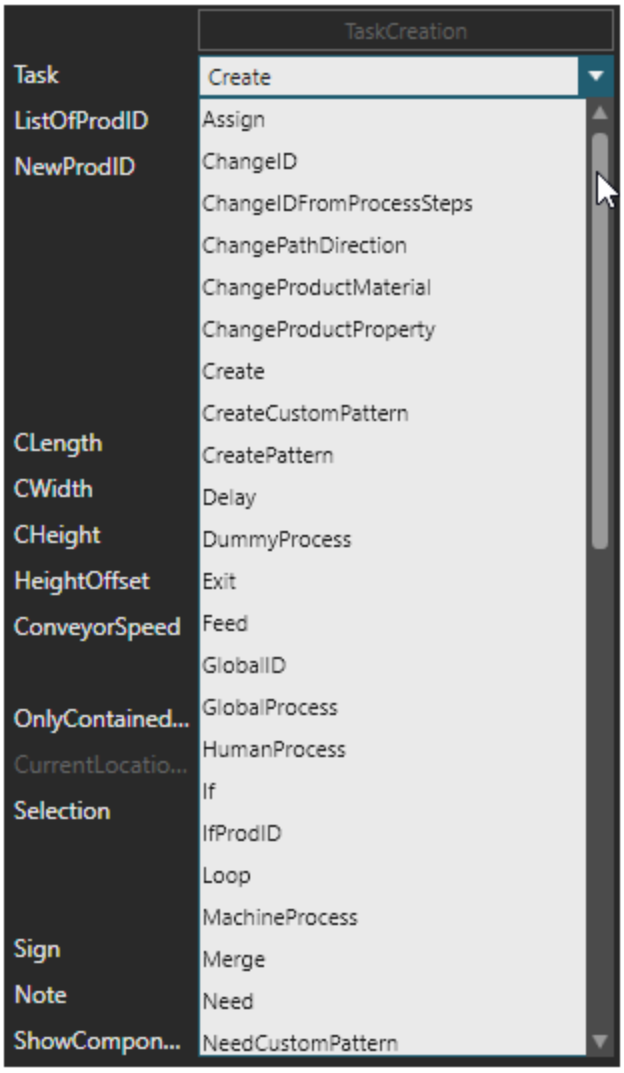
Tasks Reference Guide

A guide for using tasks in a Works Process component

Version: May 03, 2019

Disclaimer:
This document was made using Visual Components 4.1.1 and version 5.3 of the Works library. The Revision metadata property of a Works Process was 51.





Contents

Quick Reference

Demos

Layouts can be downloaded from ShareFile.

https://visualcomponents.sharefile.com/d-seeef09306a440908

Videos about the demos can be watched on YouTube.

https://www.youtube.com/playlist?list=PLSZQ7IauHhJg7bcNk-xT7IA4nXxi42gVJ

Note: If you cannot access YouTube, the video files are in the same ShareFile folder as the layouts.

Learning Pathway

You can use this pathway to help with your learning about the Works library and its tasks.

Step 1. Basics of Task Creation

This is an introductory tutorial about the Works library. It is useful for quickly learning about Works Process, Works Task Controller, and Works Robot Controller components.

The tutorial shows you how to teach locations, create different types of tasks, and use resources. The outcome of the tutorial is a layout that simulates a depalletizing operation using robots.

http://academy.visualcomponents.com/lessons/works-library-basics-task-creation/

Step 2. Human and Machine Processes

This tutorial introduces the use of human workers and machines as well as Labor Resource Location components.

The tutorial shows you to how to attach a Works Process component to a machine, execute machine processes, assign tasks to a human worker as well as its pick and place locations. The outcome of the tutorial is a machine cutting operation that is operated by a human.

http://academy.visualcomponents.com/lessons/works-library-human-machine-processes/

Step 3. Simulate a Palletizing Operation

This tutorial allows you to test your skills by building a layout with the Works library and using a robot to perform a palletizing operation.

http://academy.visualcomponents.com/lessons/simulate-palletizing-operation-works-library/

Step 4. Simulate a Depalletizing Operation

This tutorial allows you to practice your skills as well as develop ways to manipulate how a robot picks and places parts.

http://academy.visualcomponents.com/lessons/learn-build-simulate-depalletizing-operation-using-works-library/

Step 5. Simulate Process with Human, Machine and Robot

This tutorial is a good test of your basic skills and understanding of the Works library.

The tutorial shows you to how to build a layout from start to finish. New concepts include the use of a robot mounted on a track with an external axis, editing the ResourceLocation of a Works Process component, and executing subroutines in a robot program. You may also realize the logic behind changing ProdID values after each step in a global process. That is, adding more logic to a process to get the expected outcome.

http://academy.visualcomponents.com/lessons/simulate-process-human-machine-robot/

Step 6. Self-learning tasks

After you have completed the first five steps, you should take some time to review demo layouts and videos as well as tasks documented in this guide.

We recommend learning some basic tasks in this order:

1. Create
2. Delay
3. TransportOut
4. TransportIn
5. ChangeID
6. GlobalID
7. ChangeProductMaterial
8. ChangeProductProperty
9. Merge
10. Split
11. DummyProcess
12. HumanProcess
13. RobotProcess
14. GlobalProcess
15. MachineProcess
16. Feed and Need
17. Pick and Place

Note: Some users might prefer to focus first on using Presets. For example, a preset is helpful for someone in sales and marketing to quickly create and demo solutions.

We recommend learning some intermediate tasks in this order:

1. WaitProperty and WriteProperty
2. WaitSignal and WriteSignal
3. WarmUp
4. Exit
5. Order and WaitForOrder
6. StoreProdID and RestoreProdID
7. ProdIDFromList

We recommend learning some advanced tasks in this order:

1. If
2. Assign
3. IfProdID
4. Loop
5. Sync
6. SensorConveyor and ChangePathDirection
7. StoreProdID_NewFromProcessSteps, ChangeIDFromProcessSteps, and UpdateProductProcessSteps

Step 7. Create a New Task

Once you have a good understanding of how tasks are used in the Works library, you can start to create your own tasks as well as modify others.

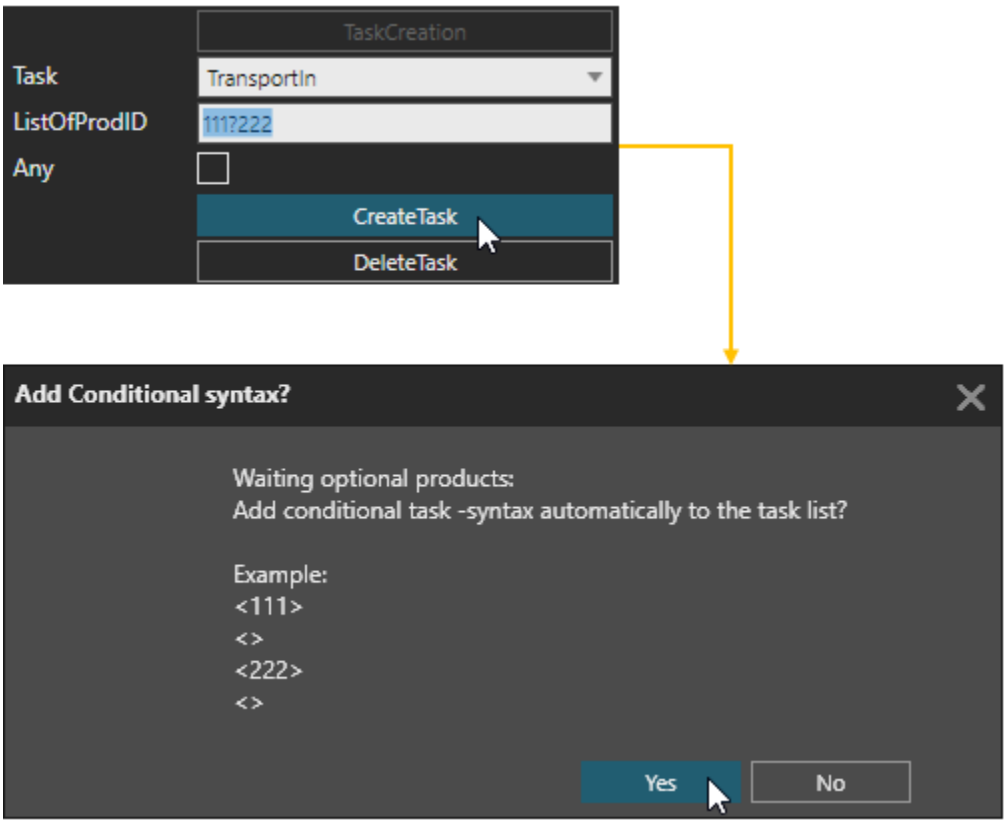
This tutorial shows you how to access, read, and edit the Python scripts used for creating and executing tasks in a Works Process component. The outcome is a new task that rotates a component.

http://academy.visualcomponents.com/lessons/create-new-task-works/

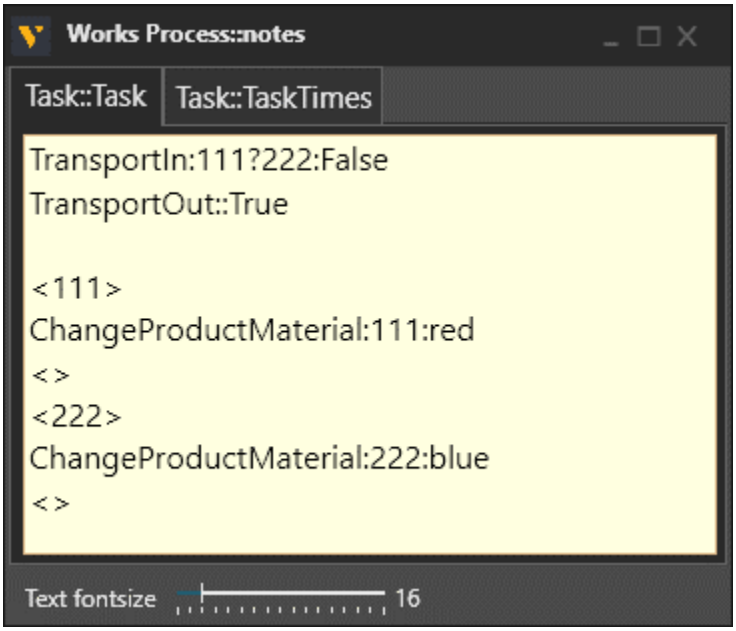
Terms

Conditional Tags

Conditional tags are a set of opening and closing tags in a process. These tags are used by tasks to execute a subprocess. For example, an If task uses conditional tags to identify what do when its expression returns a true or false value. The opening tag defines the start of the condition and its name. The closing tag marks the end of the condition. These tags can be placed immediately after a task or at the end in a Task note. In some cases, you will be prompted to add conditional tags when creating a task.



Tasks inside conditional tags are executed only when a task calls the condition by name. Every condition with that name is executed in the process. Be aware of that logic and exercise caution when using a ProdID as the name for a condition.

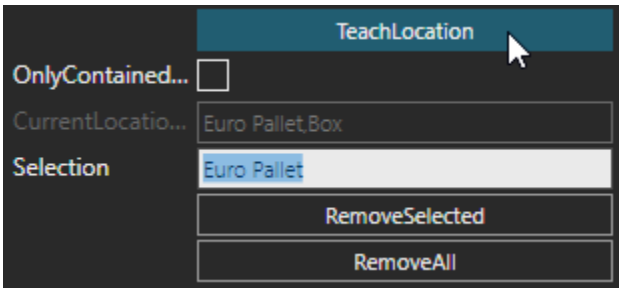


Tasks that use or support conditional tags:

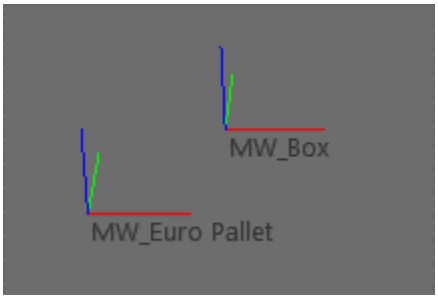
- If
- IfProdID
- Loop
- Need
- Order
- TransportIn
- WaitForOrder
- WaitProperty
- WaitSignal

Location

Location in this context is referring to a Frame feature in a Works Process component used for positioning other components. The frame can be added, removed and updated using the default properties of a Works Process component.



The location of a frame is a key-value pair: the key is a ProdID, and value is a position matrix. Each frame is labeled MW_ followed by its ProdID.



A location can refer to the global position of a static component. If you enabled the OnlyContainedComponents property, a relative location is created for components that are contained in a Works Process component. Generally, contained components are dynamic and generated during a simulation. By default, a location refers to the origin of a component.

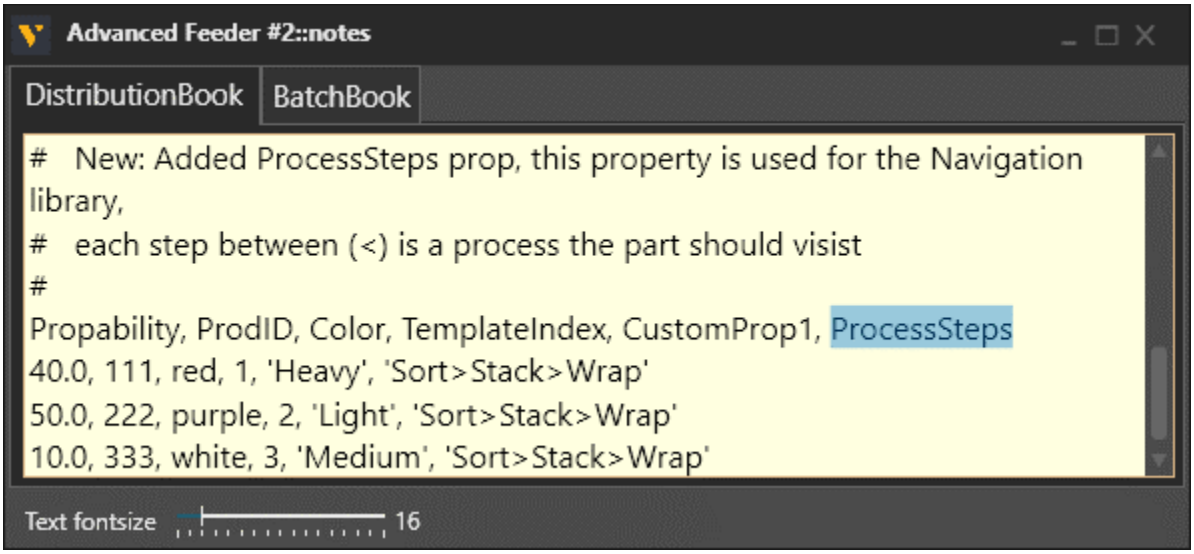
Tasks that use locations:

- Create
- CreateCustomPattern
- CreatePattern
- Need
- NeedCustomPattern
- NeedPattern
- Place
- PlacePattern
- SensorConveyor
- TransportIn
- TransportInPattern

ProcessSteps

ProcessSteps is a property in a component used for defining the steps the component should take in a process. That is, a component can locally define its process steps.

Generally, a step is used as a ProdID value. The delimiter for steps must be a greater than sign (>). Steps are useful for component navigation because they allow Works Process components and resources to refer to steps/stages defined in the parts themselves. One approach to adding a ProcessSteps property is to use an Advanced Feeder along with a template for adding properties.



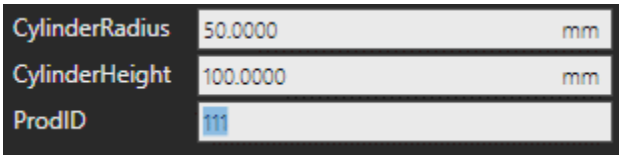
In most cases, you would first use a StoreProdID_NewFromProcessSteps task to save and update the ProdID of a component to indicate its first step. From there, a Works Process component could use a Need task to request the component. Next, a ChangeIDFromProcessSteps task would be used to update the ProdID of the component to indicate its next step. An UpdateProductProcessSteps task would then be used to remove the step from the component. This workflow of changing and updating the ProdID and ProcessSteps properties would continue until a component reaches the end of its process. From there, a RestoreProdID task could be used to reset the ProdID of the component to the value of its StoreID property.

Tasks that use ProcessSteps:

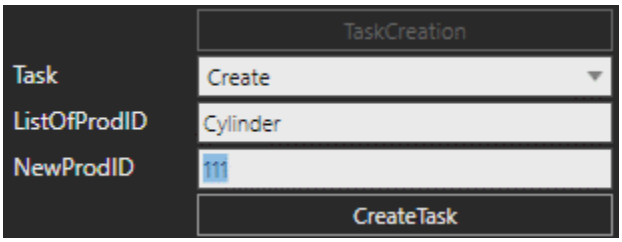
- ChangeIDFromProcessSteps
- StoreProdID_NewFromProcessSteps
- UpdateProductProcessSteps

ProdID

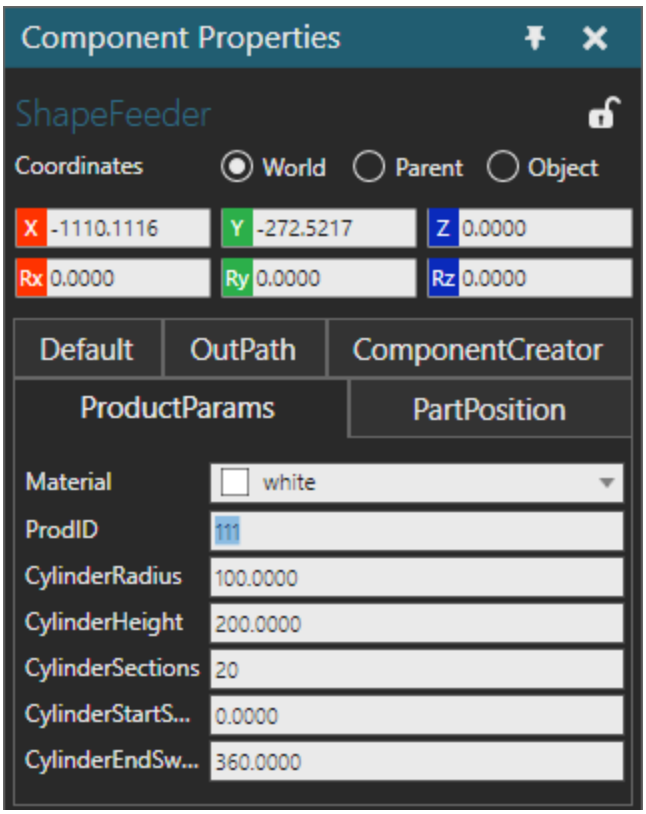
ProdID is a property in a component used for completing tasks and teaching locations. It is a way to identify the component and classify its type in a process.



If a component does not have a ProdID property, the property will be added by a Works Process component.



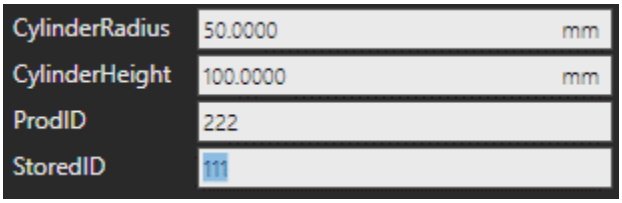
Feeders in your eCatalog panel may also create a ProdID property in a component.



Important: A ProdID should not have a hash (#), colon (:) nor an ampersand (&) in its value.

StoreID

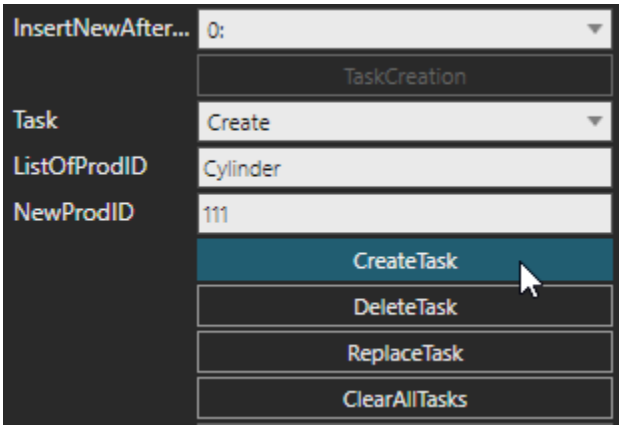
StoreID is a property in a component used for storing a previous ProdID value. It is a way to reset the ProdID of a component.



In most cases, a StoreID property is dynamic and added to a component by a StoreID or StoreProdID_NewFromProcessSteps task.

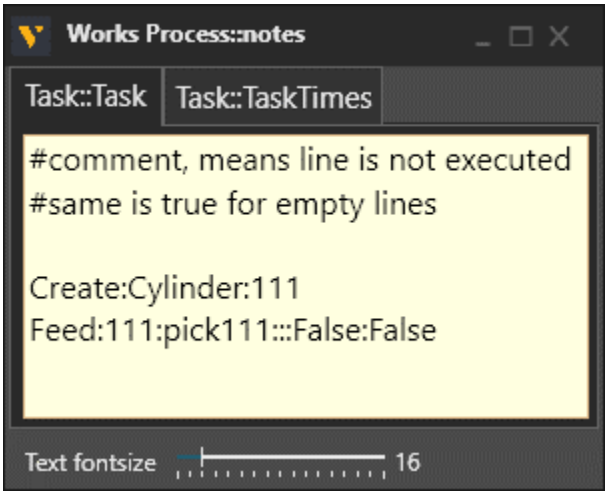
Task

Task is a command executed by a Works Process component during a simulation. Each task has its own set of properties and actions. Most tasks affect dynamic components that are contained in a Works Process component. Those components are either transported in, created, or placed there by a resource.



Some tasks require the use of resources and tools or are dependent on other tasks. A good example is Feed and Need tasks. A Need task requires a resource deliver needed components. A Feed task supplies components where they are needed by assigning the task to one or more resources. Every Feed task has a TaskName property, which allows you to define a name that identifies the task. The TaskName can then be added to a task list in a resource. That way the resource knows what tasks it can do and when to do them.

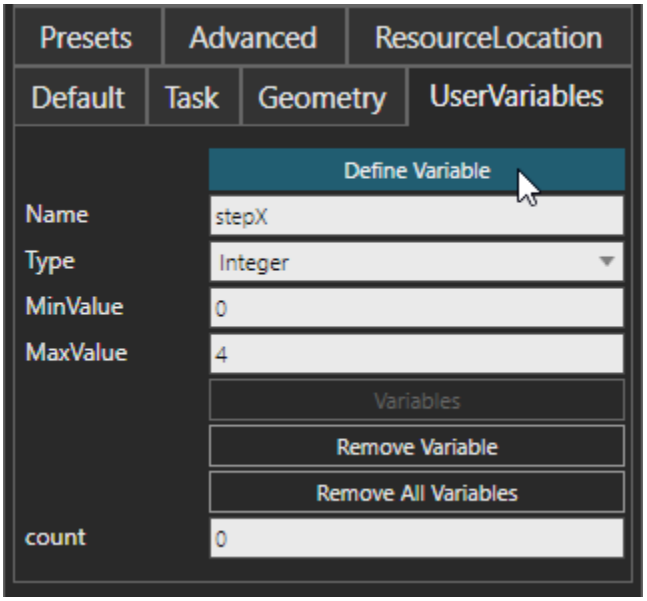
Every Works Process component has a Task note that lists the tasks in its process. Tasks are executed in a loop at the start of a simulation.



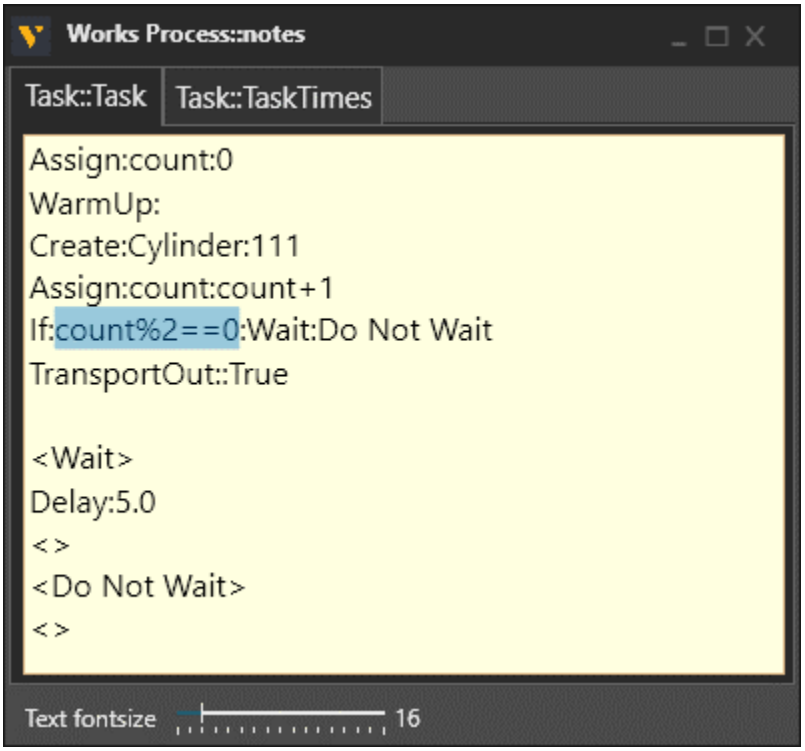
The Task:RunTaskTimes property of a Works Process component controls the number of times it executes the loop. Tasks are executed line by line in order unless they are conditional. An Exit task immediately ends the execution of the loop. A WarmUp task segments the process: tasks before a WarmUp are executed one time, whereas tasks after a WarmUp are executed in a loop.

Variable

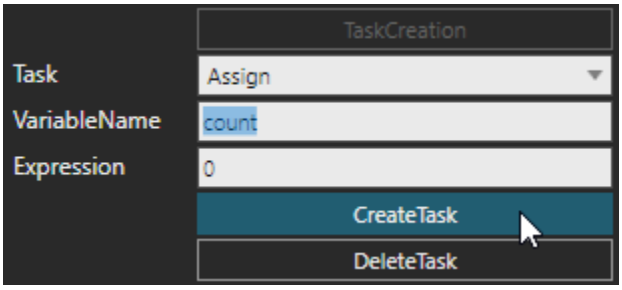
Variable is a property of a Works Process component that is listed in its UserVariables group of properties.



Variables are global in a Works Process component, so they can be referenced in expressions and used by certain types of tasks, for example an If task.



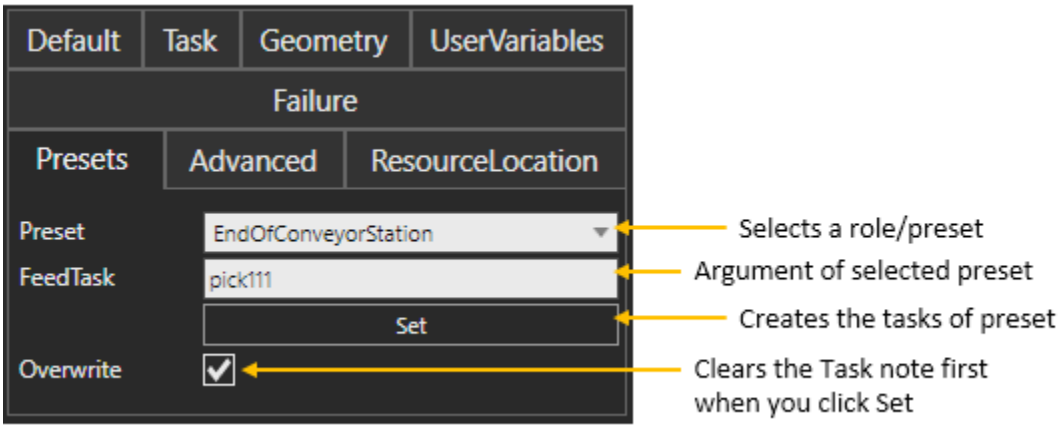
We recommend setting the value of variables at the start of a simulation. When using an Assign task, if the variable does not exist, it will be automatically created for you and its data type is based on the given expression.



Note: Each task property that is supporting the use of UserVariables is marked with [1)] superscript in the task description in this document.

Presets

A Works Process component has a group of Presets properties that allow you to generate a set of tasks based on predefined roles. This is useful for quickly adding tasks in a process.



EndOfConveyorStation

EndOfConveyorStation assumes the role of supplying other components. You can use this role to quickly transport in and feed parts.

- **FeedTask** defines the TaskName for a Feed task.

HumanProcessStation

HumanProcessStation assumes the role of working with a human. You can use this role to request parts, execute a human process, merge the parts, and then feed them.

- **PartNameIn** defines the ListOfProdID for a Need task.
- **AssembledParts** defines the ListOfProdID for another Need task.
- **ProcessTask** defines the TaskName for a HumanProcess task.
- **ProcessTime** defines the ProcessTime for a HumanProcess task.
- **PartNameOut** defines the NewProdID for a ChangeID task. PartNameIn defines the SingleProdID argument for the ChangeID task.
- **FeedTask** defines the TaskName for a Feed task.

MachineStation

MachineStation assumes the role of working with a machine. You can use this role to request parts, execute a machine process, merge the parts, and then feed them.

- **PartNameIn** defines the ListOfProdID for a Need task.
- **AssembledParts** defines the ListOfProdID for another Need task.
- **MachineName** defines the SingleCompName for a MachineProcess task.
- **ProcessTime** defines the ProcessTime for a MachineProcess task.
- **PartNameOut** defines the NewProdID for a ChangeID task. PartNameIn defines the SingleProdID argument for the ChangeID task.
- **FeedTask** defines the TaskName for a Feed task.

RobotProcessStation

RobotProcessStation assumes the role of working with a robot. You can use this role to request parts, execute a routine in a robot program, merge the parts, and then feed them.

- **PartNameIn** defines the ListOfProdID for a Need task.
- **AssembledParts** defines the ListOfProdID for another Need task.
- **ProcessTask** defines the TaskName for a RobotProcess task.
- **PartNameOut** defines the NewProdID for a ChangeID task. PartNameIn defines the SingleProdID argument for the ChangeID task.
- **FeedTask** defines the TaskName for a Feed task.

Sink

Sink assumes the role of executing a generic process that affects components. You can use this role to request parts, execute a delay, and then remove the parts.

- **PartName** defines the ListOfProdID for a Need task.
- **Interval** defines the DelayTime for a Delay task.

Source

Source assumes the role of creating components and supplying them to other components. You can use this role to create parts, and then feed them.

- **PartName** defines the ListOfProdID for a Create task.
- **NewProdID** defines the NewProdID for the same Create task.
- **FeedTask** defines the TaskName for a Feed task.
- **Interval** defines the DelayTime for a Delay task.

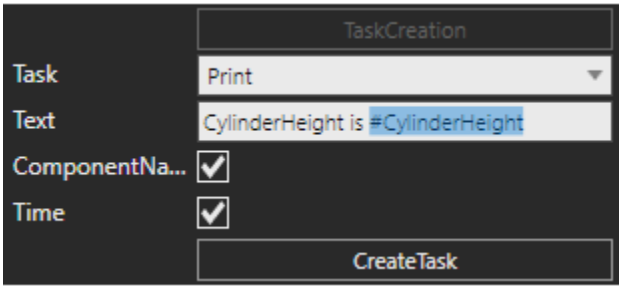
StartOfConveyorStation

StartOfConveyorStation assumes the role of demanding components. You can use this role to request parts, and then transport them to a connected conveyor.

- **PartNameIn** defines the ListOfProdID for a Need task.
- **PartNameOut** defines the NewProdID for a ChangeID task. PartNameIn defines the SingleProdID argument for the ChangeID task.

Reading Values

You can read the property value of a component contained in a Works Process or the Works Process itself by using a tag "#". This is useful for writing expressions, defining process times, and reading/writing property and signal values with tasks.



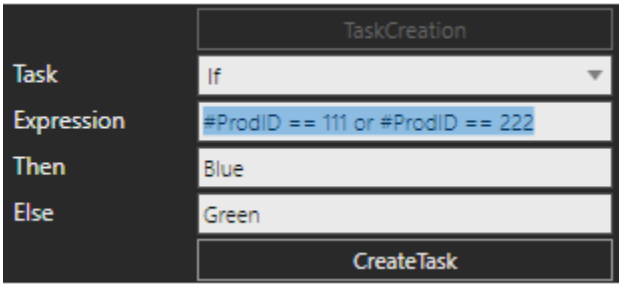
Syntax

A tag for reading a property value has short and long forms.

Short Form `#<property name>`

Long Form `#<property name>@<ProdID value>`

The long form allows you to filter properties by ProdID value. The short form allows you to simplify the tag and expand its scope to any component.



If you want to write an expression using multiple tags, use a space ( ) as a delimiter. Note that you are allowed to use both short and long forms in the same expression.

Short Form `#<prop name> <prop name>`

Long Form `#<prop name>@<ProdID value> #<prop name>@<ProdID value>`

If you are not referring to default properties, you must first indicate the tab/group name followed by two semicolons.

Short Form `#<tab name;;property name>`

Long Form `#< tab name;;property name >@<ProdID value>`

Supported Tasks

For tasks in a Works Process component, you can use tags with these properties:

- DelayTime
- Expression
- ListOfProdID (not all tasks supported, see Notes below)
- ListOfOrderNames
- NewProdID
- ProcessTime
- PropertyValue
- SignalValue
- SingleProdID (not all tasks supported, see Notes below)
- Text
- ToolName (only contained component)

Note: Each task property that is supporting the use of tag "#" syntax is marked with [2)] superscript in the task descriptions in this document.
Note: in some Tasks you can access process' UserVariables directly by just using just the UserVariable name. For more info see section Variable

Distributions

Task properties that take a time value (in seconds) support the use of normal and uniform distributions. The syntax to use is "10n1" for normal distribution with mean of 10 seconds and std. deviation of 1 second. Type "1u10" for uniform distribution with minimum value of 1 second and maximum of 10 seconds. To define a seed for the random function check Works Task Control property "WorksRandomSeed".

Supported Tasks

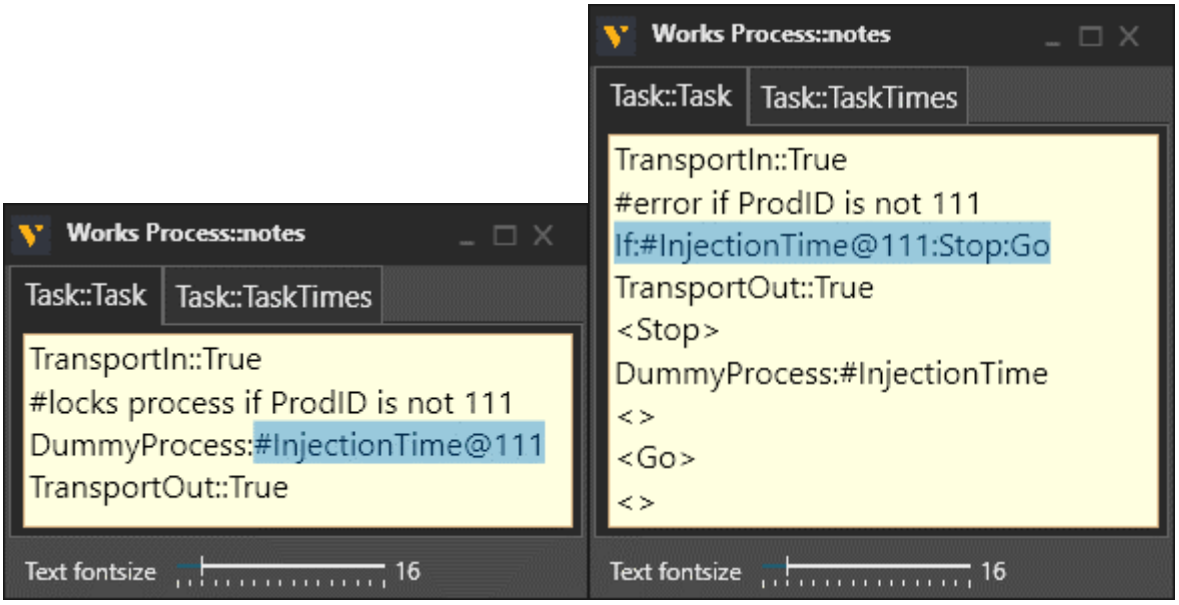
For tasks in a Works Process component, you can use distribution syntax with these properties:

- DelayTime
- ProcessTime

Note: Each task property that is supporting the use of distribution is marked with [3)] superscript in the task description in this document.

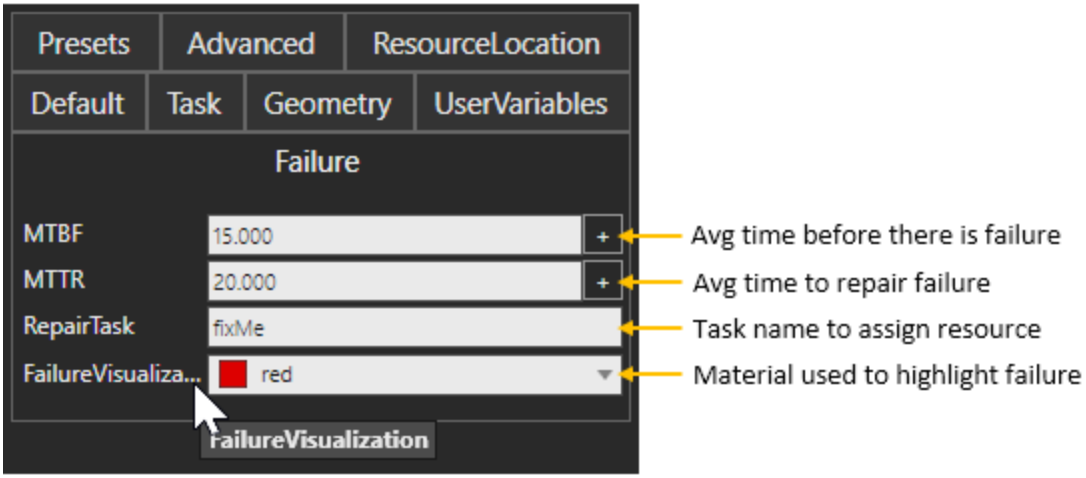
Errors and Exceptions

A tag is validated when its task is executed during a simulation. Why? Tags in this context are mostly used for reading the property values of dynamic components. You must avoid parsing and type errors by using the correct syntax and not referring to nonexistent properties in a component. You can work around deadlocks, but we recommend type safety to avoid errors.

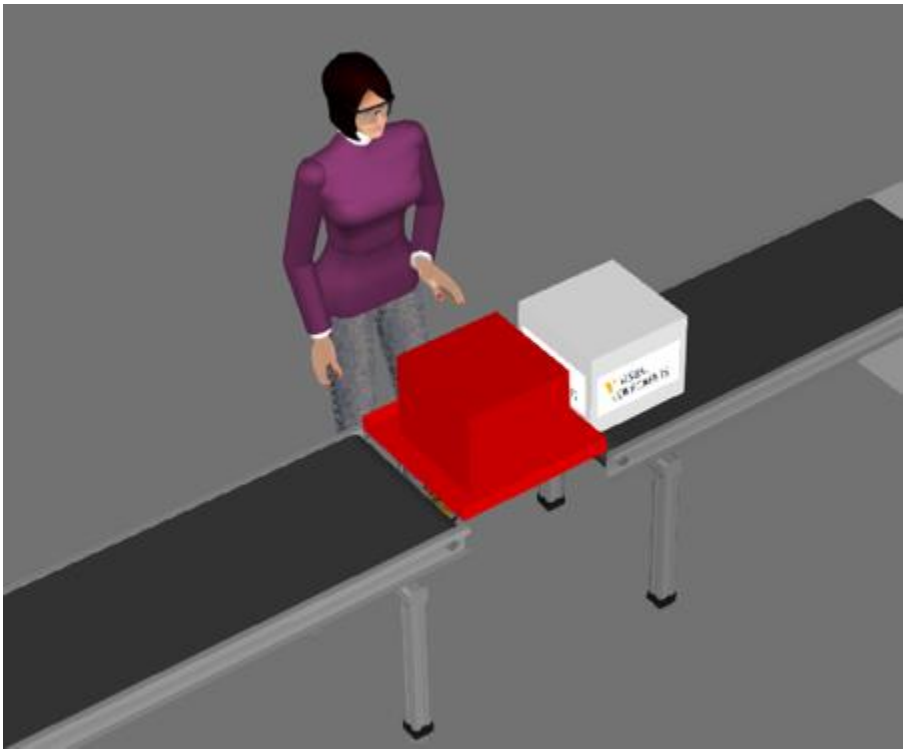


Failures and Repairs

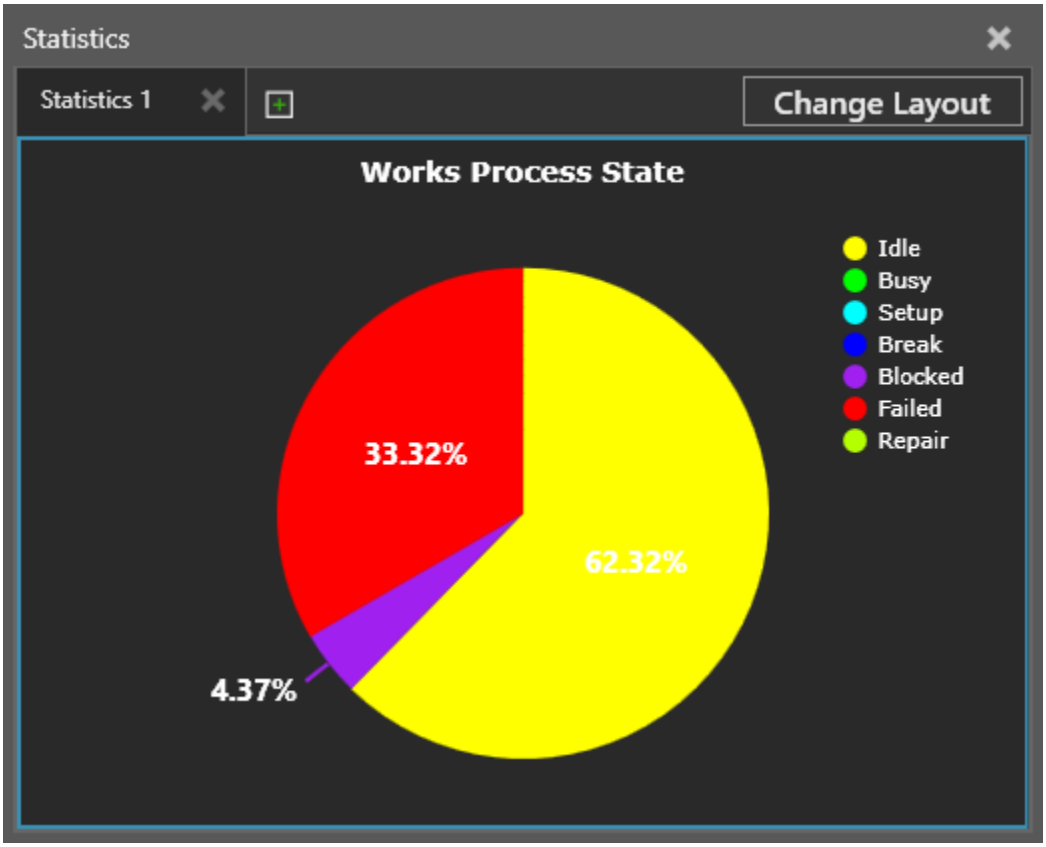
A Works Process component has a group of Failure properties for simulating failures in a process.



RepairTask allows you to require a resource to come and fix a failure at a Works Process component.



When a failure occurs, a Works Process component is automatically put into a failure state.



FAQ

Q1. What is the difference between Pick and Place and Feed and Need tasks?

Task	Action	Resource	Depends On
Pick	Robot picks up a component	Robot	None
Place	If robot has picked up component, place the component	Robot	Pick
PlacePattern	If robot has picked up component, place the component in a pattern	Robot	Pick
Feed	When component is needed, resource picks it up	Any	Need
Need	When component is available, resource first picks it up, and then places it	Any	Feed
NeedPattern	When component is available, resource first picks it up, and then places it in a pattern	Any	Feed
NeedCustomPattern	When component is available, resource first picks it up, and then places it in a pattern defined in a Works Task Controller	Any	Feed Works Task Controller

Containment - To pick up a component, tasks require a component be contained in a Works Process component.

Dependency - You cannot use a Pick task to complete a Need task. You cannot use a Feed task to complete a Place task.

Resource Availability - A resource must be available to complete each task.

Resource Type - Pick and Place tasks are done by a robot connected to a Works Robot Controller. Feed and Need tasks can be done by any resource that is compatible with the Works library, for example a robot, human, mobile robot (AGV), or drone.

Tools - Place and PlacePattern tasks allow you to specify which tool a robot should use to place a component. Need, NeedPattern and NeedCustomPattern tasks do not allow this and refer to the tool specified in a Feed task used for picking up the needed component.

Q2. How many Works Task Controllers are needed in a layout?

You only need one, and it can be invisible.

Q3. Can I use the Works library with other component libraries?

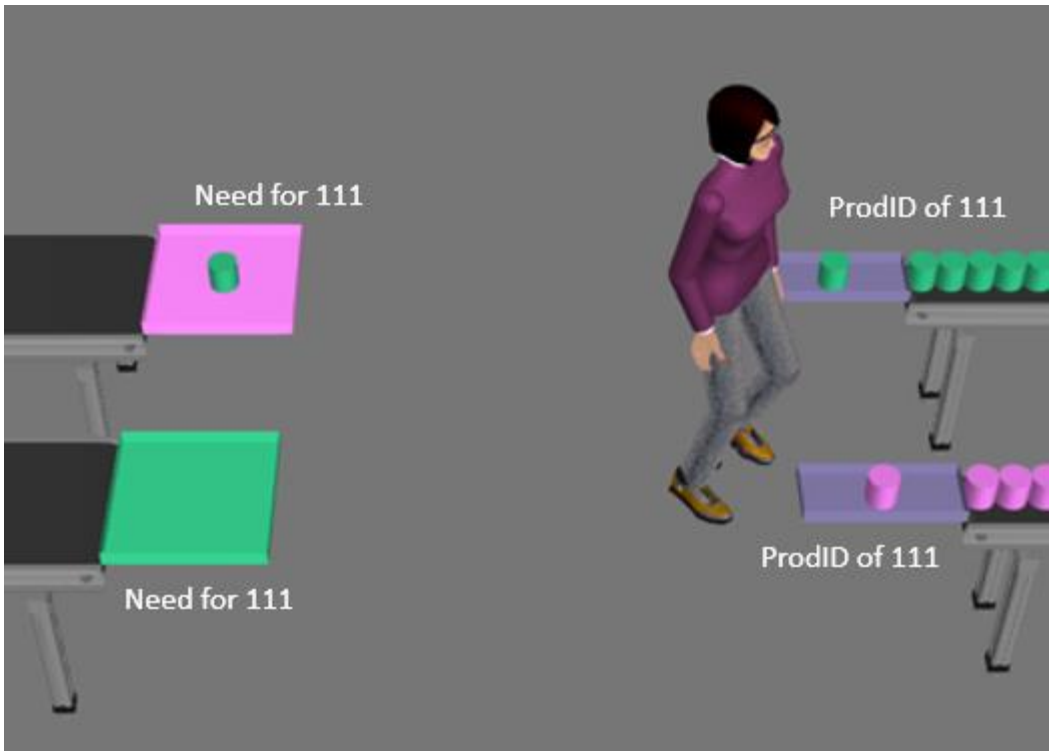
Yes. For example, you can use components in the Works library with components in the Machine Tending library. In most cases, the layout would require a Works Task Controller.

Q4. Can a human perform a Pick or Place task?

No. Only a robot can perform a Pick or Place task. A human can perform Feed and Need tasks, which involve picking and placing components.

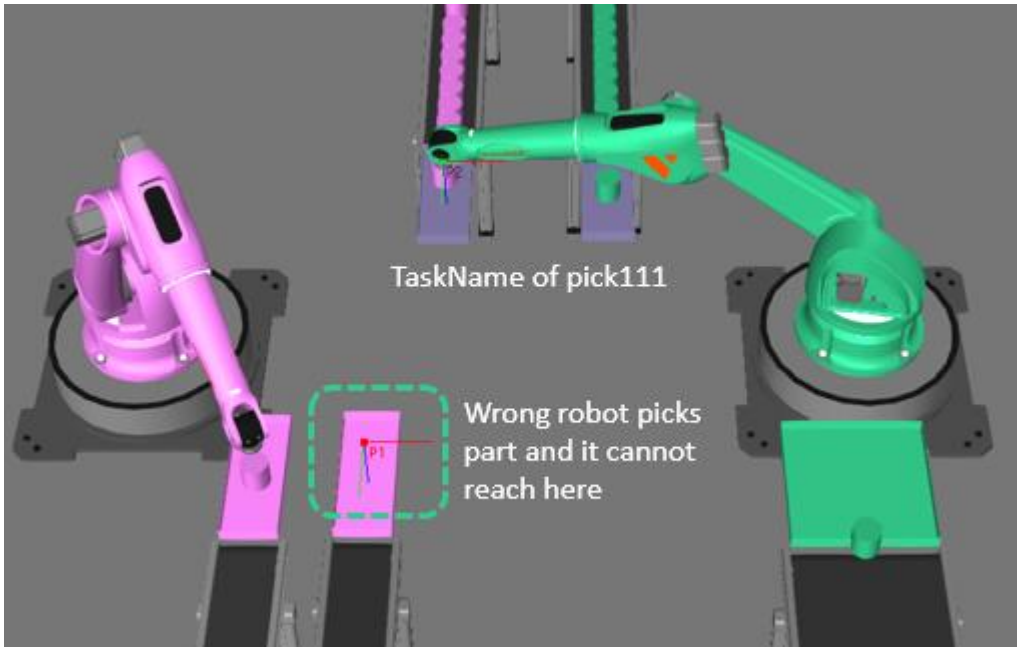
Q5. Why is a resource picking and placing components in the wrong Works Process component?

This might happen when there are multiple needs for the same ProdID.



One solution is to use different ProdID values and restore them as needed to get the expected outcome.

You might also encounter issues where the wrong resource picks up a component or there is reachability issue in robot. Generally, this happens when you use the same TaskName for multiple tasks.



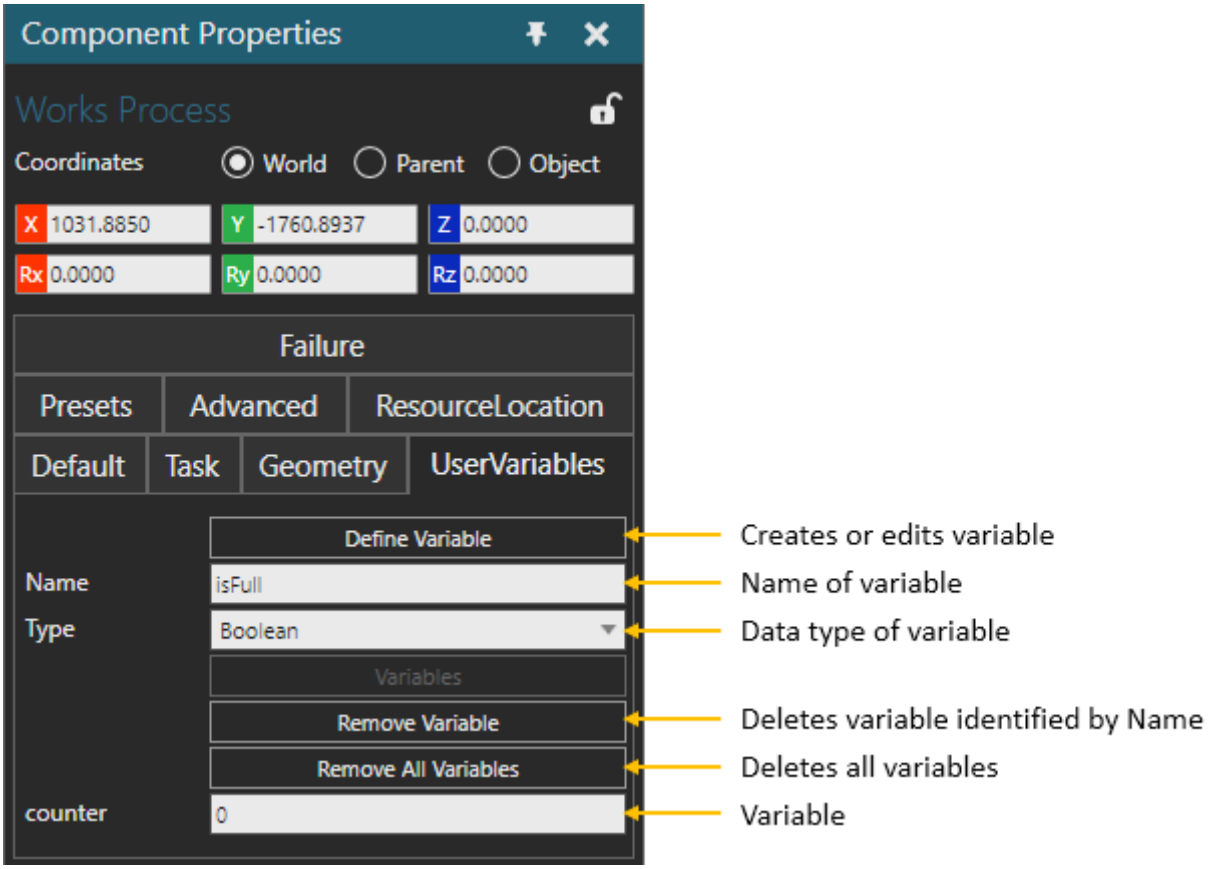
You can fix this by using different TaskName values and assigning them to the intended resources. That way a resource knows what tasks it should perform during a simulation.

Assign

An Assign task allows you to add and edit variables in a Works Process component.

Variables

Variables are grouped and managed in the UserVariables property tab of a Works Process component. Variables are component properties that can be referenced by other properties, tasks, and expressions in the component. Variable type can be either Boolean, integer, string, real number or real number distribution. Depending on the variable type, additional attributes are shown to set constraints such as min and max values.



Properties

VariableName identifies the variable by name. If the variable does not exist, the variable is created and its data type is based on the evaluation of Expression.

Expression[1,2)] defines an expression that is evaluated to return the value to assign the variable. You must use Python syntax for writing the expression. Since variables are component properties, you can refer to a variable by name to reference its current value.

For help in writing expressions, refer to Python 2.7 documentation.

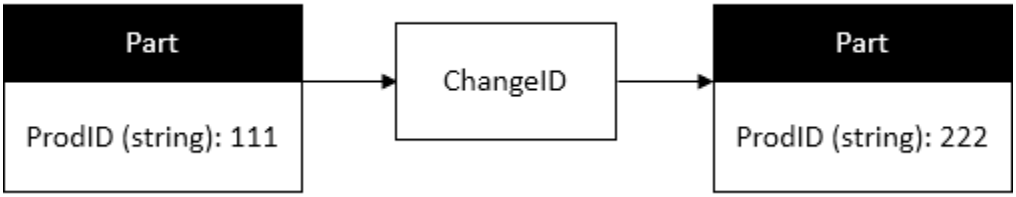
https://docs.python.org/2/reference/expressions.html

See Also

- If

ChangeID

A ChangeID task allows you to change the ProdID property of a component.



Properties

SingleProdID identifies the component by ProdID value.

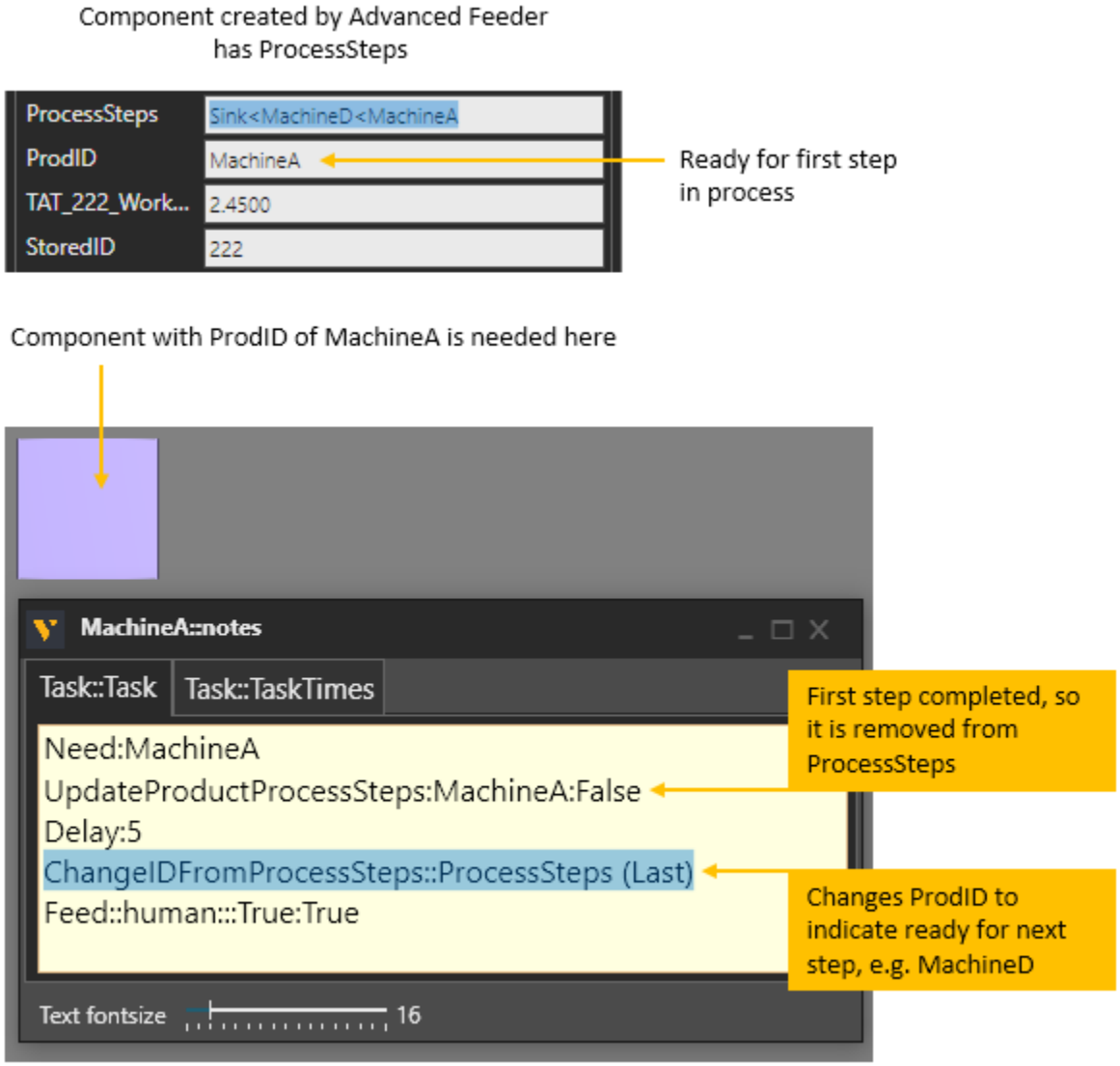
NewProdID[2)] defines the new ProdID value.

See Also

- ChangeProductMaterial
- ChangeProductProperty
- ProdIDFromList
- RestoreProdID
- StoreProdID

ChangeIDFromProcessSteps

A ChangeIDFromProcessSteps task allows you to change the ProdID property of a component by referring to its ProcessSteps property. Generally, this is done to mark the component as ready for the next step in its process.



Properties

SingleProdID identifies the component by ProdID value.

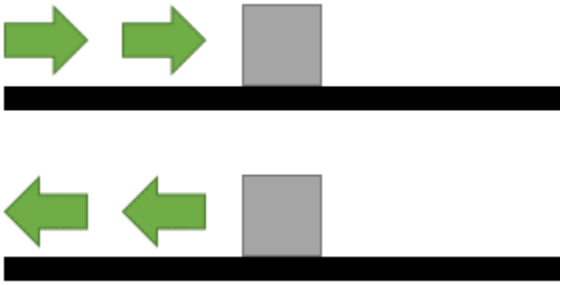
NewIDFrom selects the first or last item in the ProcessSteps property of a component. This would depend on the order in which you are executing steps.

See Also

- ProcessSteps
- StoreProdID_NewFromProcessSteps
- UpdateProductProcessSteps

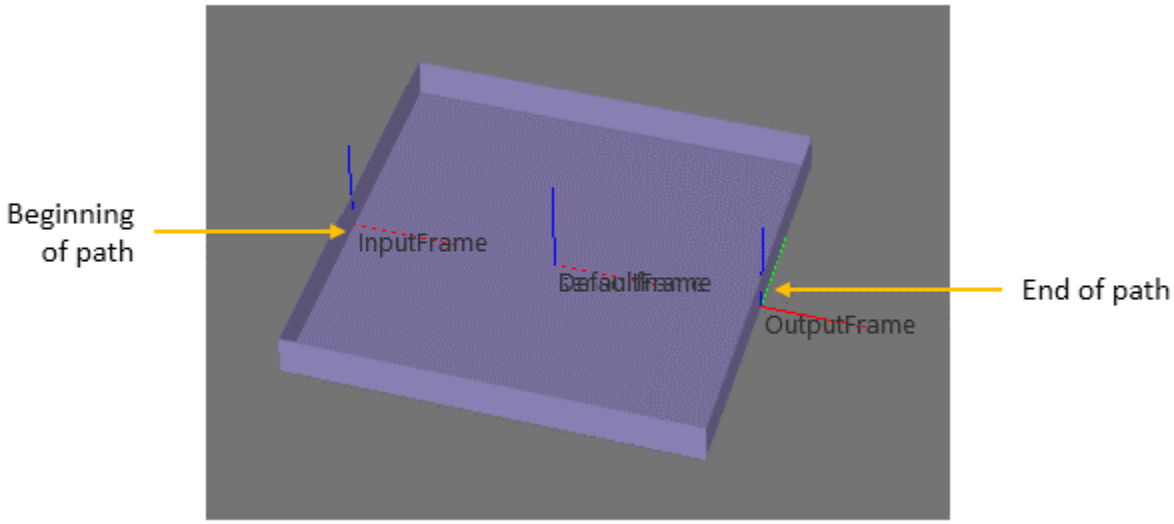
ChangePathDirection

A ChangePathDirection task allows you to change the path of a Works Process component. For example, you can move parts forward and backward after completing a set of tasks.



Path

A Works Process component has a two-way path to support this functionality. The input port of the path is located at InputFrame. The output port of the path is located at OutputFrame.



Properties

Direction defines the path direction of a Works Process component executing the task.

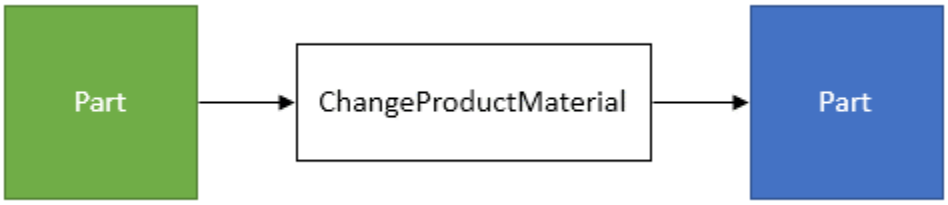
- *VC_PATH_FORWARD* moves parts in the direction of InputFrame to OutputFrame.
- *VC_PATH_BACKWARD* moves parts in the direction of OutputFrame to InputFrame.
- *VC_PATH_FORWARD_AUTO* moves parts forward. If part is transferred into path from its output port then direction automatically switches to backward.
- *VC_PATH_BACKWARD_AUTO* moves parts backward. If part is transferred into path from its input port then direction automatically switches to forward.

See Also

- SensorConveyor

ChangeProductMaterial

A ChangeProductMaterial task allows you to change the Material property of a component.



Properties

SingleProdID identifies the component by ProdID value.

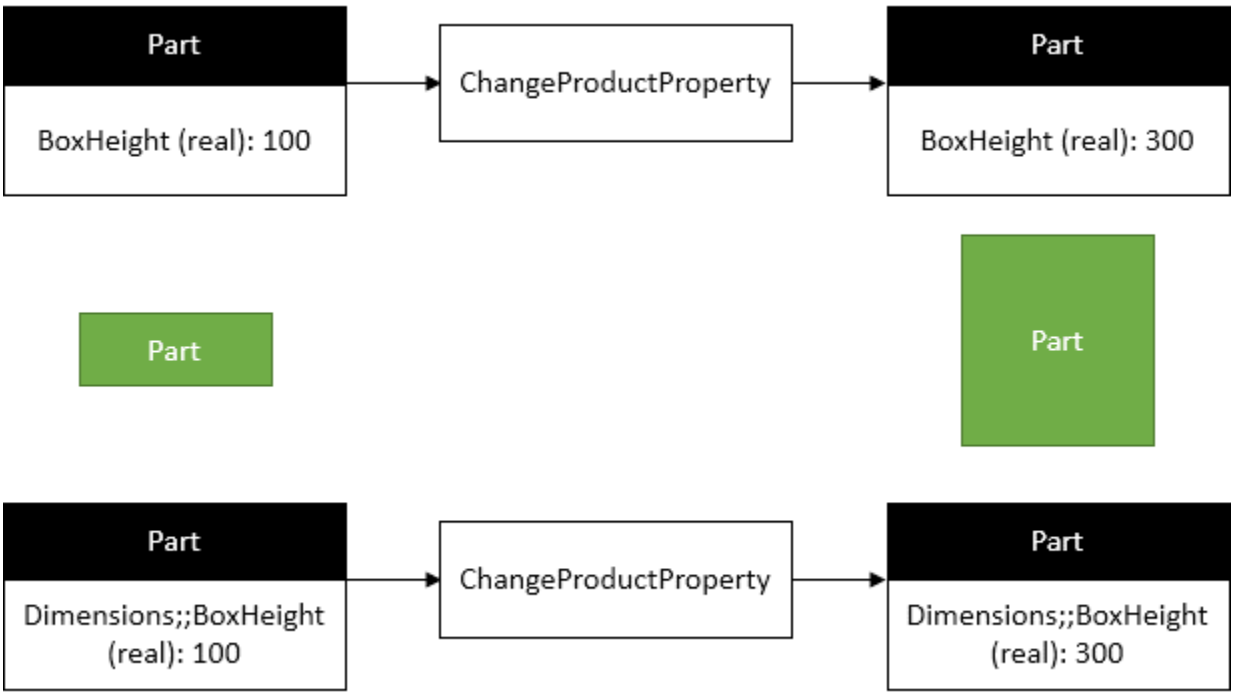
MaterialName identifies the material by name to assign the component. The material must exist in either the active layout, user library, or system library.

See Also

- ChangeID
- ChangeProductProperty

ChangeProductProperty

A ChangeProductProperty task allows you to change the property values of a component. Property type must be either Boolean, integer, real, or string.



Properties

SingleProdID identifies the component by ProdID value.

PropertyName identifies the component property by name. If not a default property, refer to the property using this syntax `<tab name>;;<property name>`.

PropertyValue[2)] defines the property value.

See Also

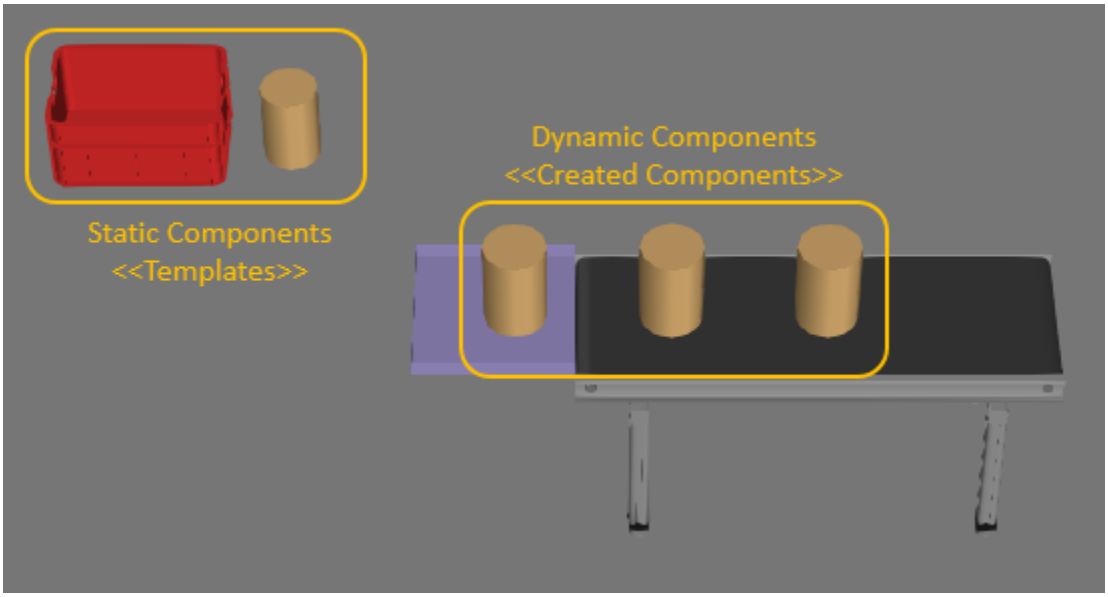
- ChangeID
- ChangeProductMaterial

Create

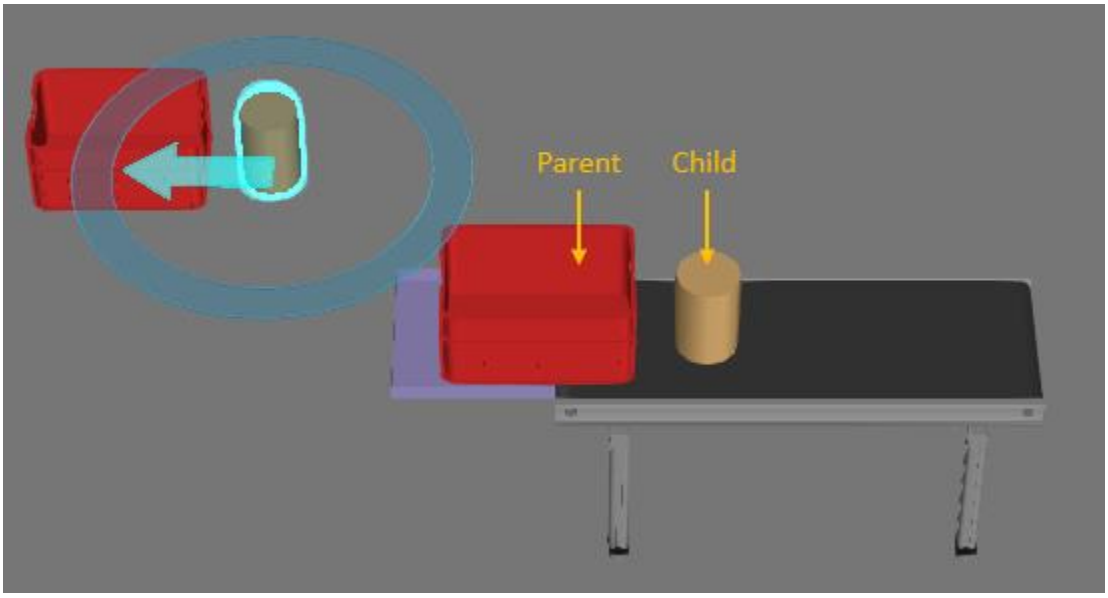
A Create task allows you to create one or more types of components. Derivatives of a Create task are CreatePattern and CreateCustomPattern tasks. Those tasks, however, can only be used to create a single type of component.

Creation

The components you want to create must exist in the 3D world. Those components are static and act as templates for creating dynamic components, which are automatically removed when you reset a simulation.

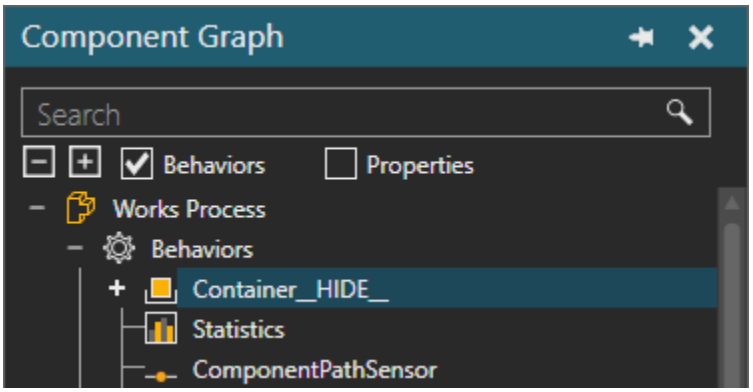


Note that a component is created along with all of its child components.



Containment

A created component is contained in the Component Container of a Works Process component. The name of the behavior is Container__HIDE__ and it is located in the root node of a Works Process component.

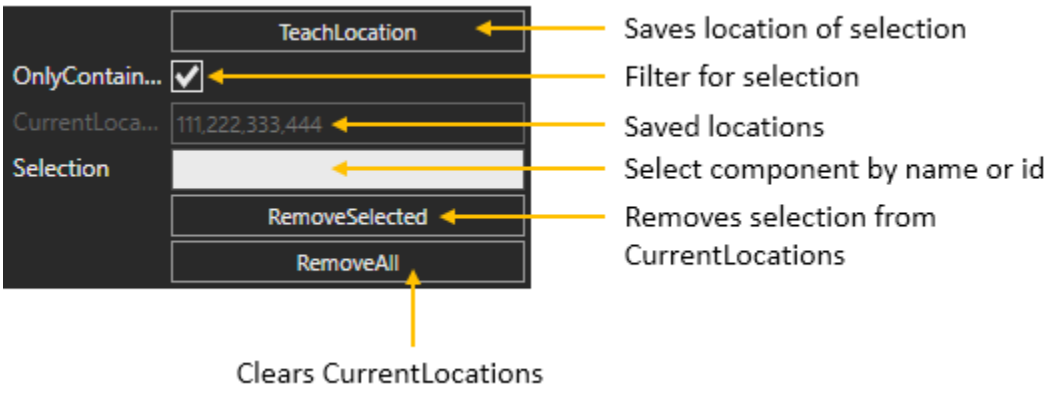


Location

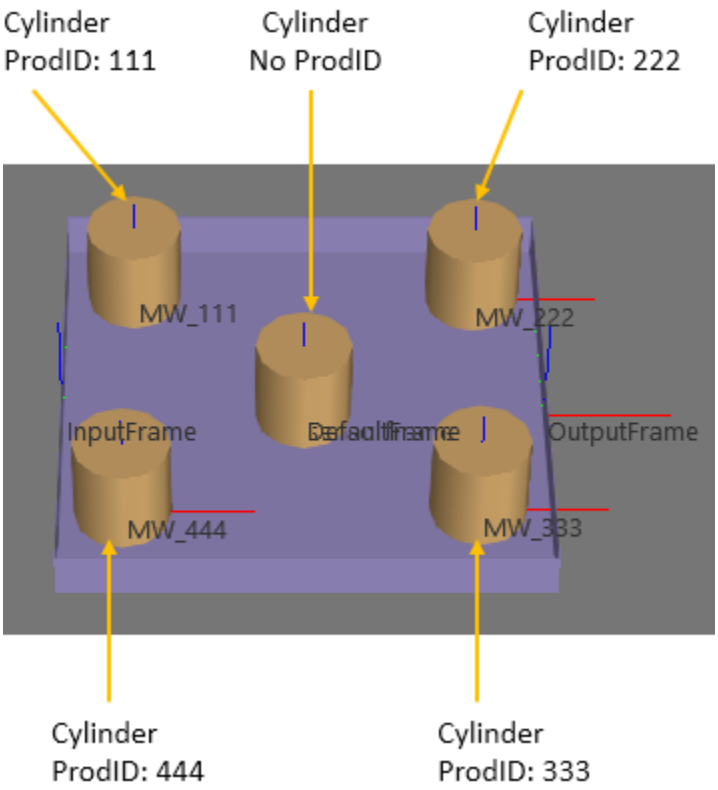
By default, a component is created at the DefaultFrame in a Works Process component.

You can create a component at a specific location by teaching it to a Works Process component. A location is a key-value pair:

- **Key** – The ProdID value of a component. If there is no ProdID property, the name of the component is used as the key.
- **Value** – A location referencing the coordinate system of a Works Process component. That is, if you move the Works Process component, the location moves with it.



Locations are used by a Works Process component to position other components. This applies to created components, components transported in, and components placed by a resource. The ProdID of a component takes precedence over its name. As a result, you can teach different locations for the same component.



Every location is a Frame feature in a Works Process component prefixed with MW_ followed by its key. Saving a location adds a new frame or updates an existing one. Removing a location deletes its associated frame.

Properties

ListOfProdID[1,2)] identifies the components you want to create during a simulation. If a component does not have a ProdID property, refer to the component by name.

NewProdID defines the ProdID value to assign a created component. If empty, the ProdID value would be the name of the component. If there is no ProdID property in the component, the property is added to the component.

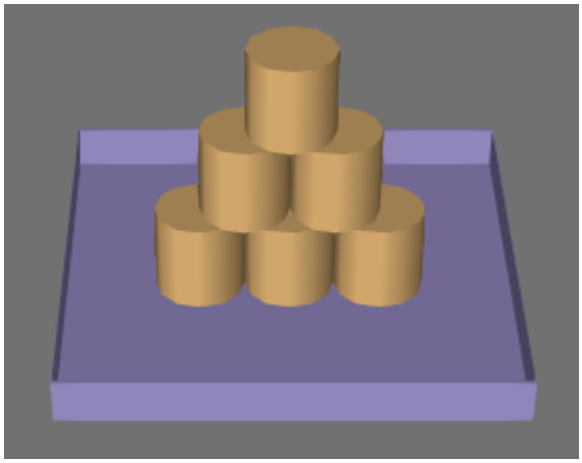
See Also

- CreateCustomPattern
- CreatePattern

CreateCustomPattern

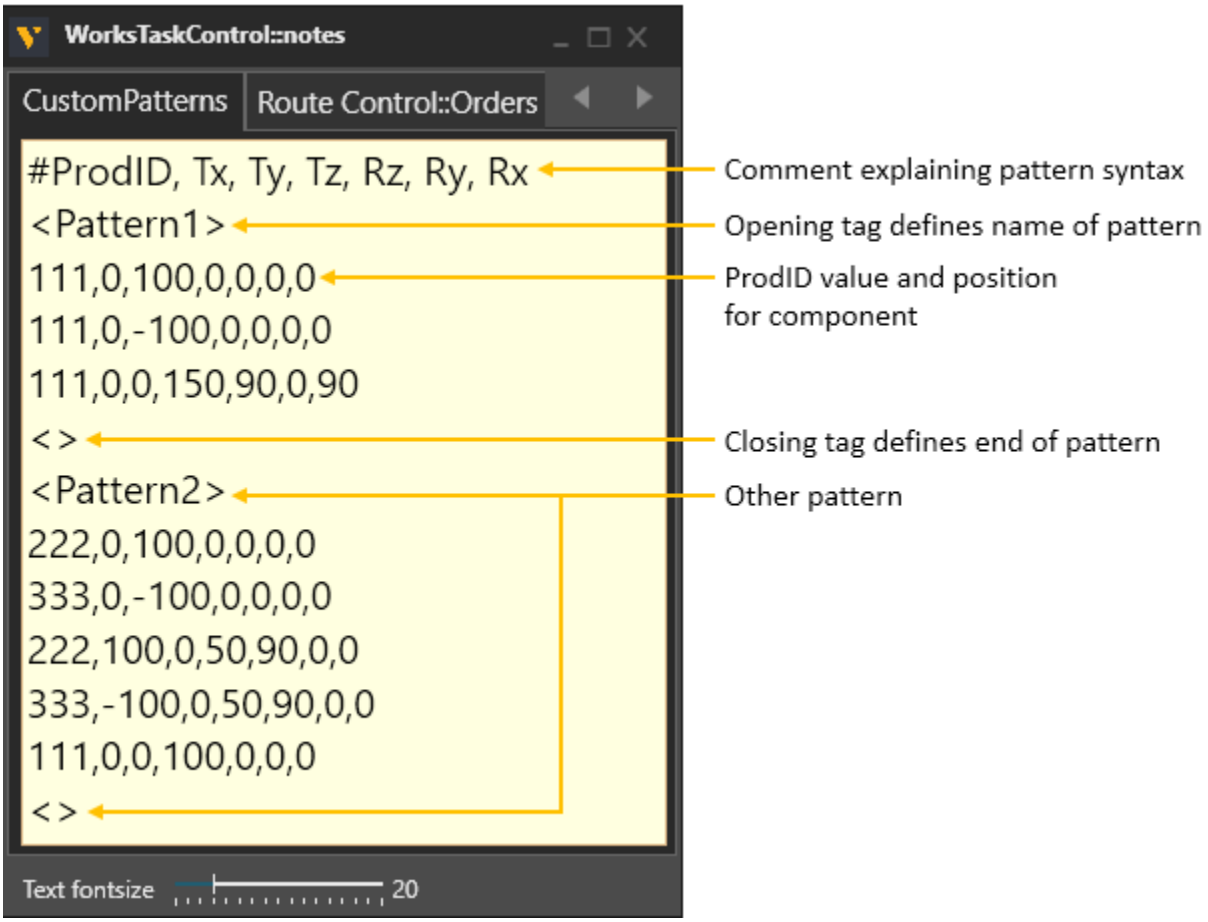
A CreateCustomPattern task allows you to create your own pattern of components.

```
<Pyramid>
#first layer
111,0,0,0,0,0
111,0,100,0,0,0
111,0,200,0,0,0
#second layer
111,0,50,100,0,0
111,0,150,100,0,0
#third layer
111,0,100,200,0,0
<>
```



Pattern

The pattern is defined in the CustomPatterns note of a Works Task Controller component. Note that in this case the ProdID argument in the pattern is just a placeholder.



Properties

SingleCompName identifies the component you want to create by name.

PatternName identifies the pattern you want to use by name.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

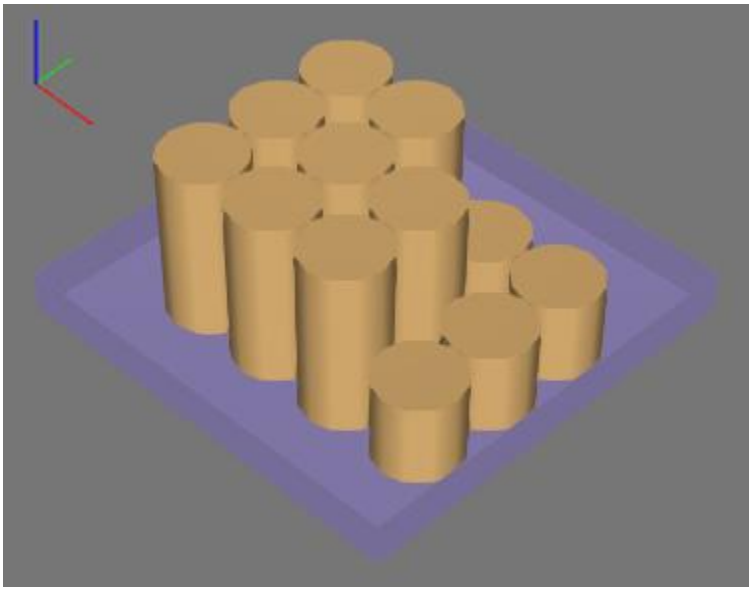
Tip: Generally, a range is modified for testing the position of components in the pattern.

See Also

- Create
- CreatePattern

CreatePattern

A CreatePattern task allows you to create a pattern of components.



Properties

SingleCompName identifies the component you want to create by name.

AmountX defines the number of components to create along X-axis.

AmountY defines the number of components to create along Y-axis.

AmountZ defines the number of components to create along Z-axis.

StepX defines the spacing of components along the X-axis.

StepY defines the spacing of components along the Y-axis.

StepZ defines the spacing of components along the Z-axis.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

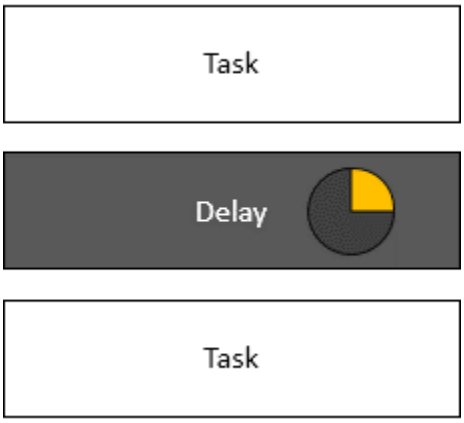
Tip: Generally, a range is modified for testing the position of components in the pattern.

See Also

- Create
- CreateCustomPattern

Delay

A Delay task allows you to delay the execution of tasks.



Properties

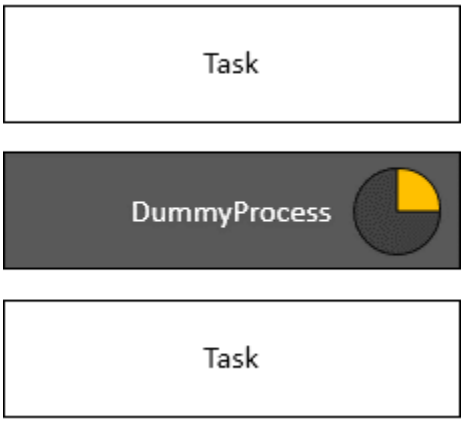
DelayTime[1,2,3)] defines the duration of delay (in seconds).

See Also

- DummyProcess
- GlobalProcess
- WaitForOrder
- WaitProperty
- WaitSignal

DummyProcess

A DummyProcess task allows you to execute a time-consuming process.



Properties

ProcessTime[1,2,3)] defines the duration of process (in seconds).

See Also

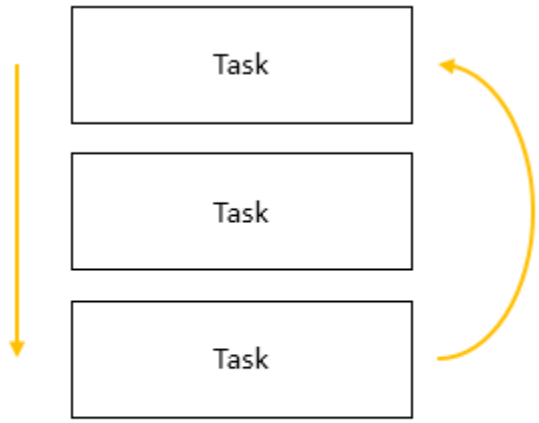
- Delay
- GlobalProcess
- WaitForOrder
- WaitProperty
- WaitSignal

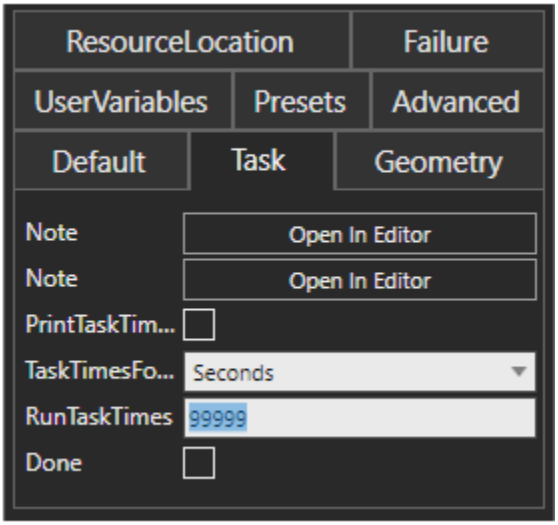
Exit

An Exit task allows you to end the execution of tasks in a Works Process component.

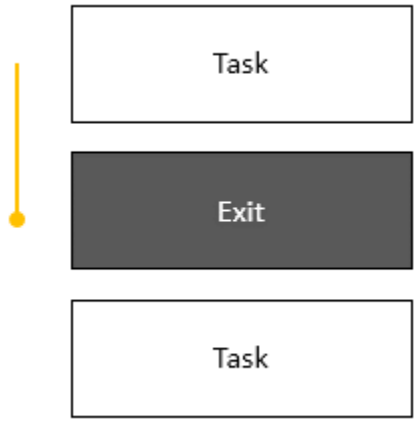
Execution

Tasks in a Works Process component are executed in a loop. The Tasks::RunTaskTimes property defines the number of times to execute that loop during a simulation.





An Exit task breaks that loop and immediately ends the process.



Properties

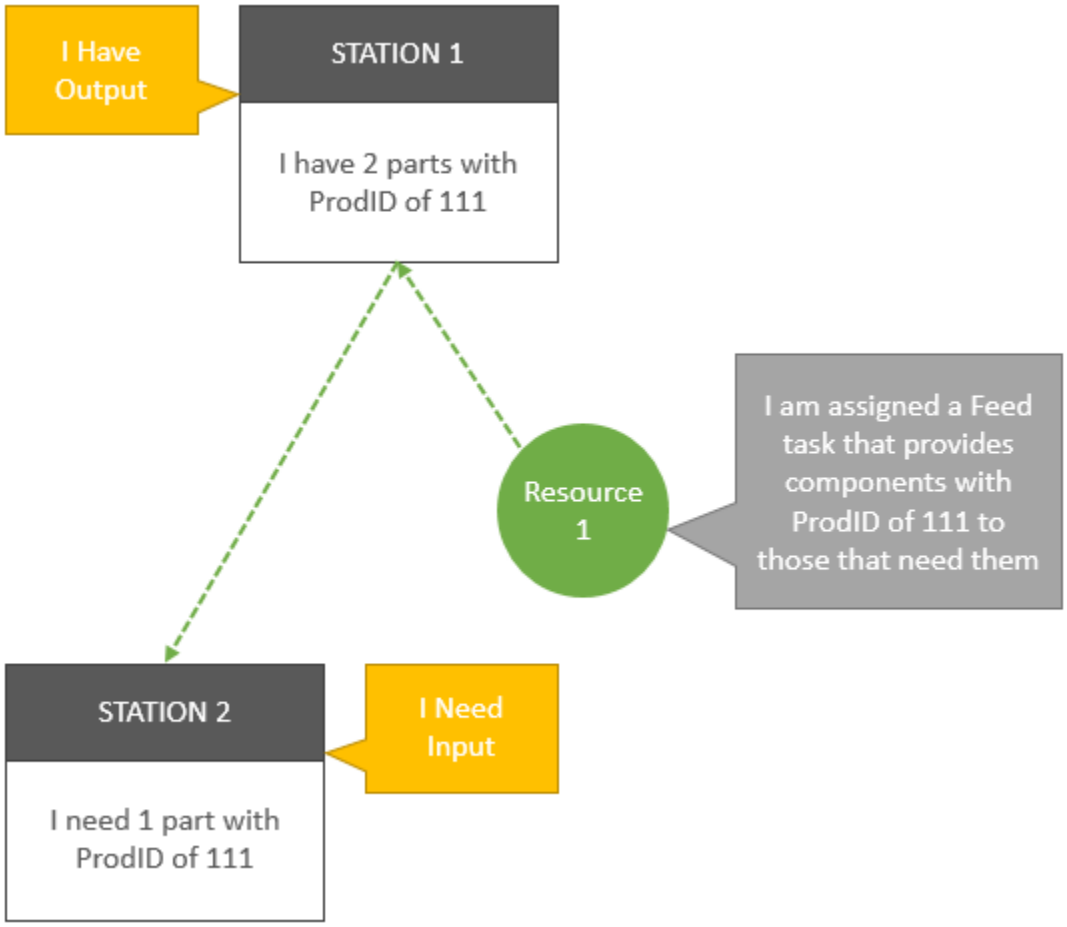
No additional properties.

See Also

- WarmUp

Feed

A Feed task allows you to request a resource to pick up and delivery components to other Works Process components.

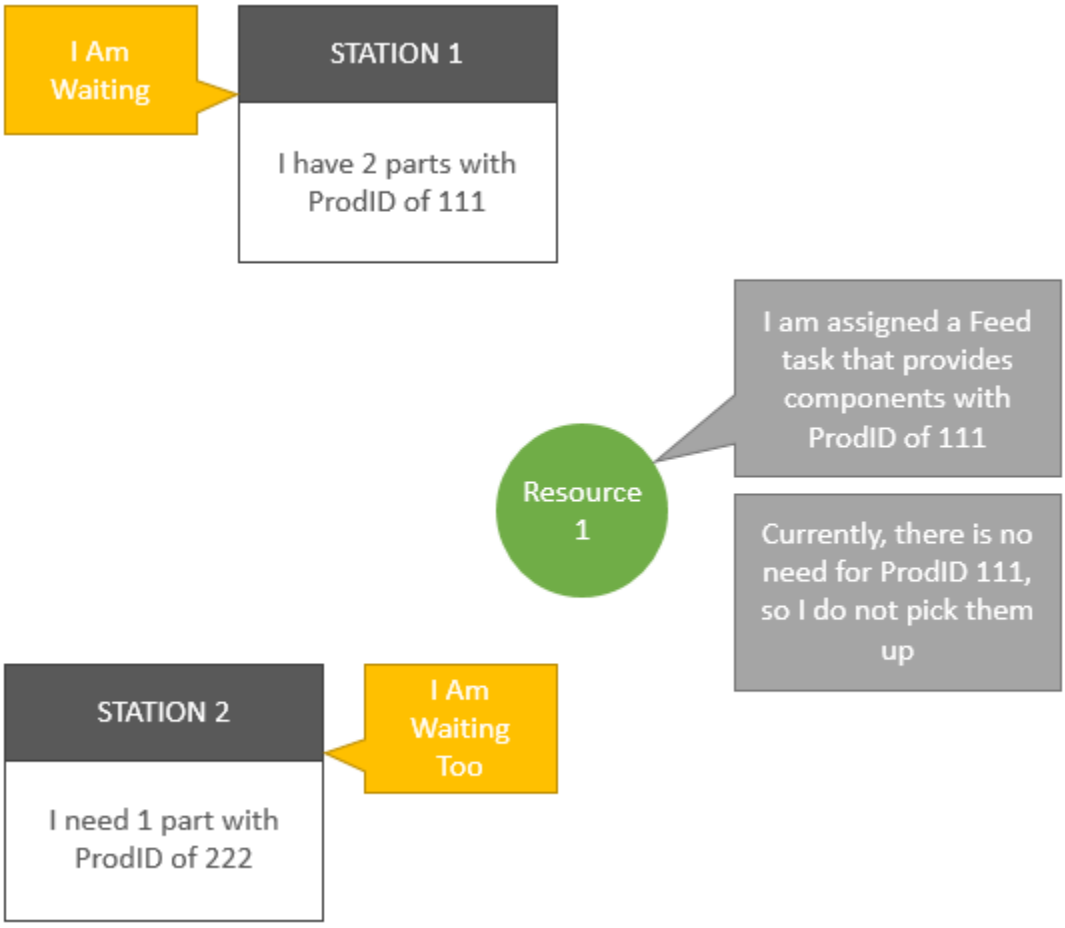


The execution of Feed task depends on several factors, which include:

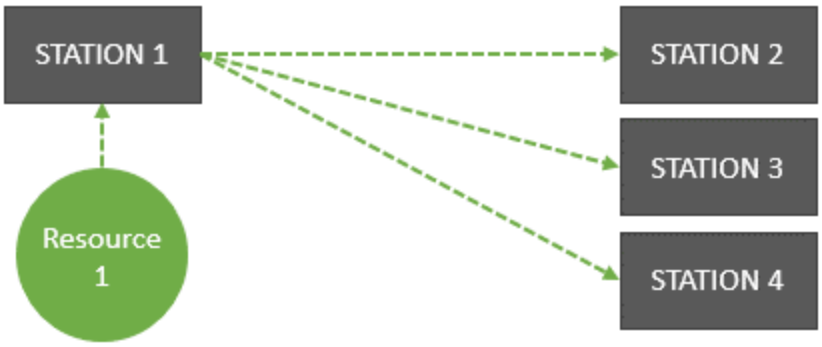
- Demand
- Task assignment
- Supply
- Timing
- Priority

Demand

A resource will not pick up components if there is no need for them. As a result, a Feed task is idle until a different Works Process component executes a Need task or one of its derivatives for those components.

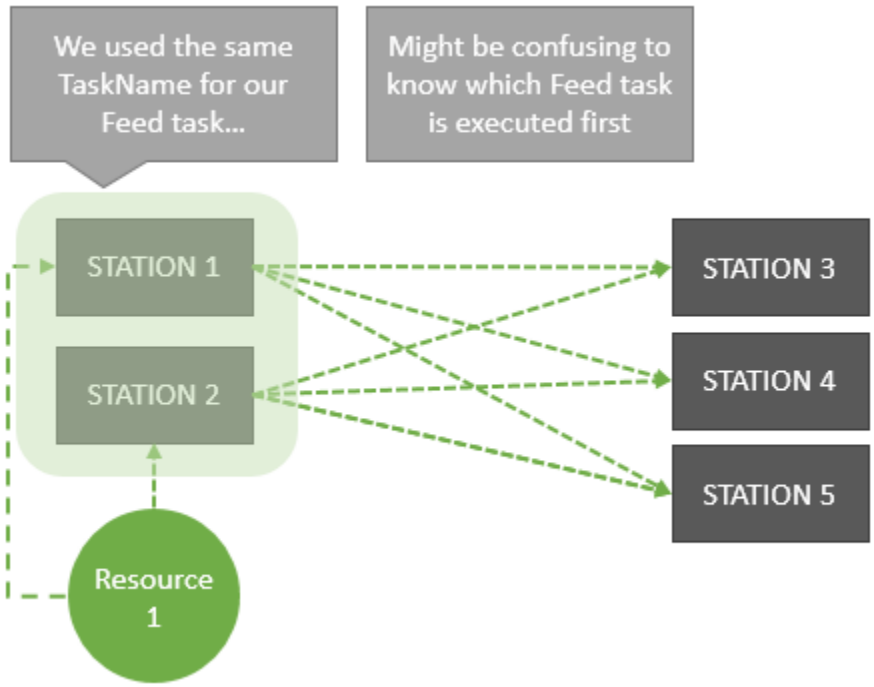


A Feed task is one-to-many meaning it can service multiple Need tasks at a time.

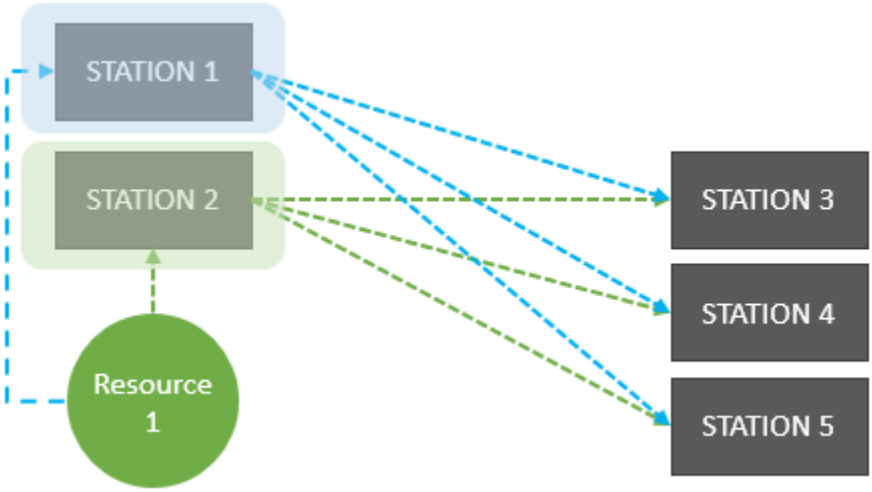


Task Assignment

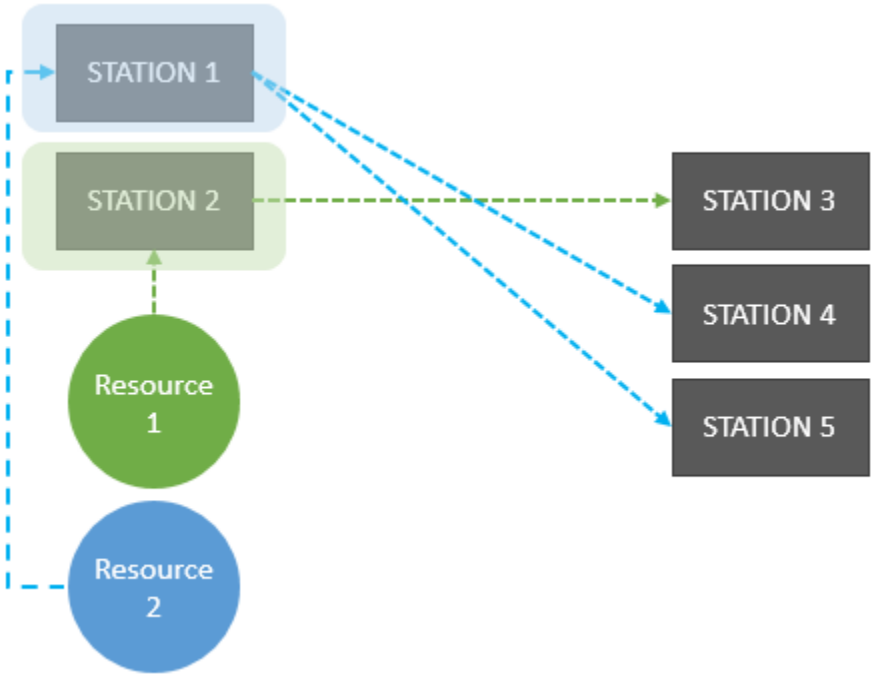
A Feed task can be executed by a robot or human resource. The task itself must be assigned to a resource using a name defined by the TaskName argument of that task. The TaskName is not a key rather a task added to a backlog of work managed by a Works Task Controller. The task informs the controller what and where components are available. That means a TaskName does not have to be unique.



However, we recommend using a TaskName that indicates its use or is easy to implement and manage. Examples of names are pickBox, pick111, pick222, and input333.



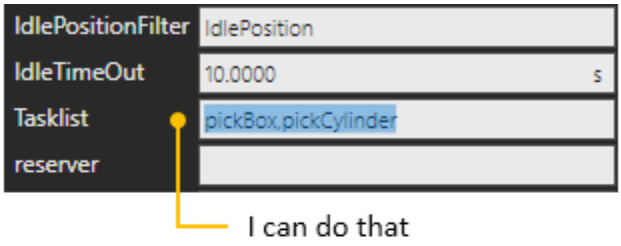
Generally, most resources have a Tasklist property that identifies its assigned tasks.

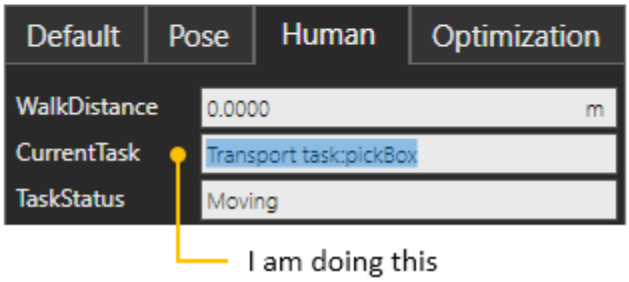


Tasks in a Tasklist are not sequential and are executed when the resource is available.

Human

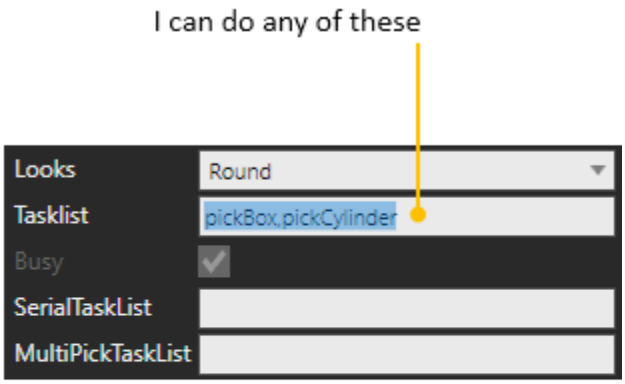
You can directly assign a Feed task to a human resource.





Robot

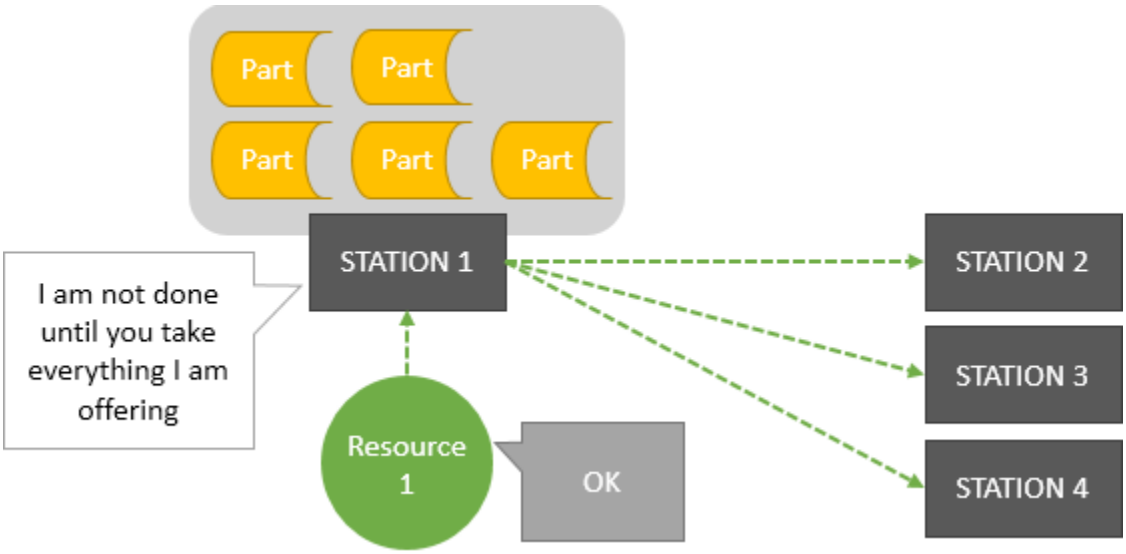
You cannot directly assign a Feed task to a robot resource. Instead, the robot must be connected to a Works Robot Controller, and then you assign the task to the controller of that robot.



If you want a robot to try to execute tasks in a certain order, use the SerialTaskList property. That list of tasks is evaluated in order from first to last. If a task in the list cannot be done, the controller will try to execute the next task. This process repeats itself during a simulation.

Supply and Demand

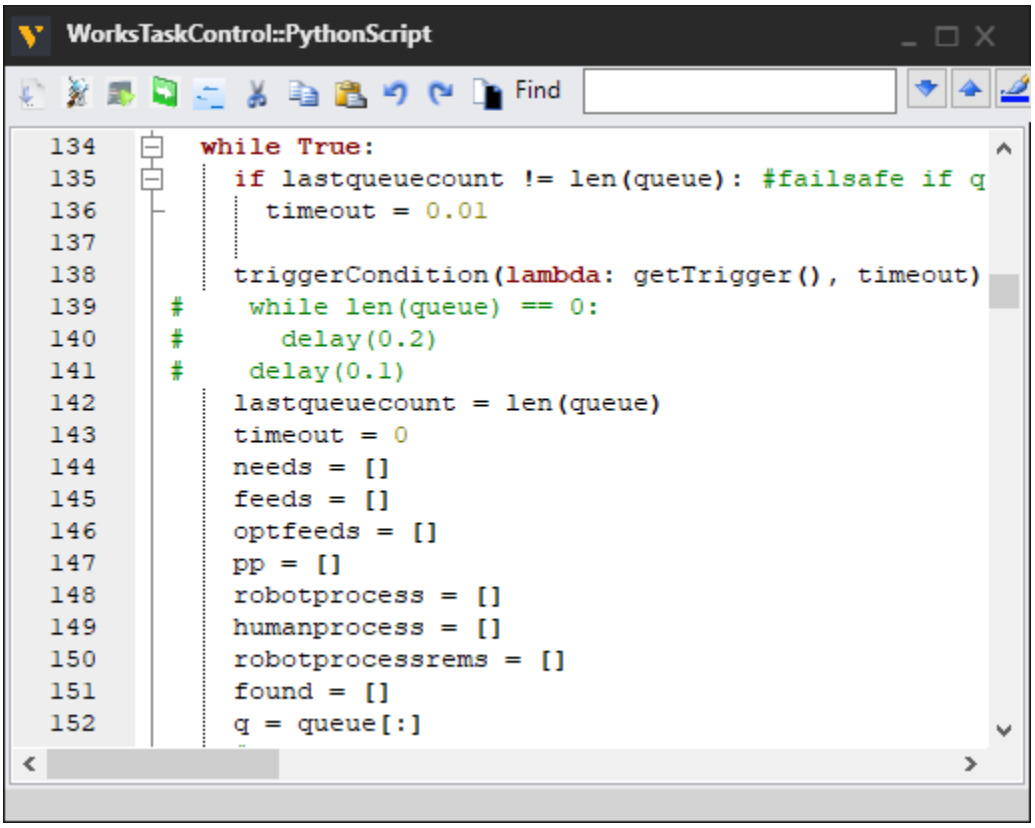
A Feed task is not completed until all of its listed components are removed from the container of its Works Process component. You have the option to feed specific components by ProdID or all of them.



Timing

The order of execution for Feed tasks is affected by the moment when a Works Process component executes a Feed task.

This is combined with the time it takes a Works Task Controller to evaluate the task.

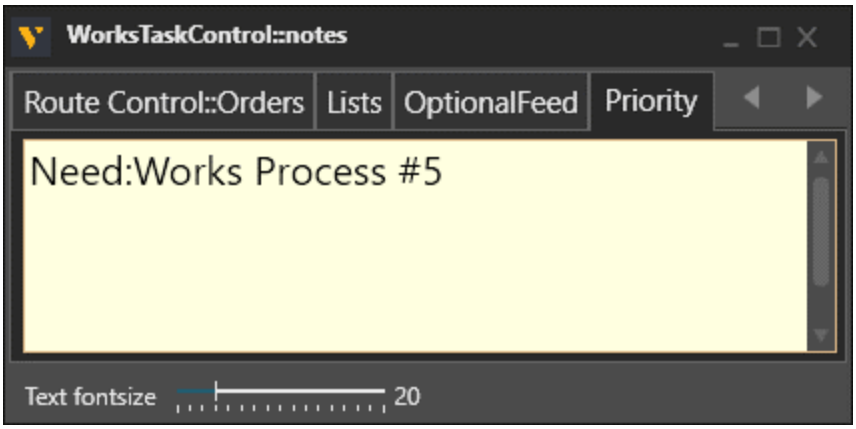


Line 134 A loop in the OnRun event that monitors and collects tasks requiring resources.

A Feed task might be executed at several stations at once, but one will always come before the other.

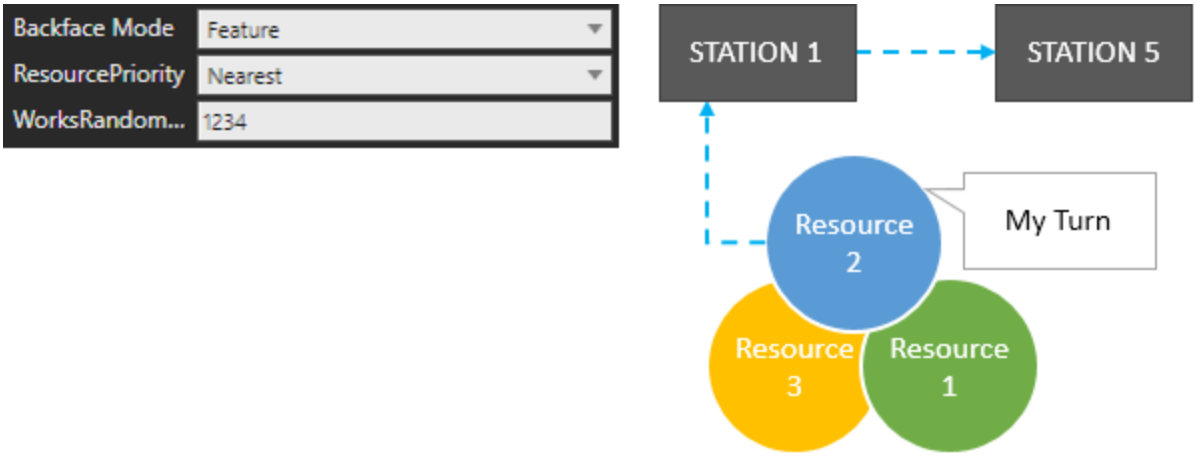
Prioritization

A Works Task Controller allows you to prioritize Feed, Need and other types of tasks using its Priority note.



For Feed and Need tasks, you can specify either a ProdID or the name of a Works Process component.

For resources, you can use the ResourcePriority property of a Works Task Controller to affect which resource is used to execute a task.



Properties

ListOfProdID[1,2)] identifies the components by ProdID value.

TaskName defines a name that allows you to reference the Feed task and assign it to a resource.

ToolName[2*)] identifies a tool component by name that should be used to execute the task. For example, you could reference the name of a pallet jack, forklift, or end-of-arm tool for a robot.
2*) property value given with "#" is read only from the product that is going to be picked.

TCPName identifies a tool frame by name in the tool component used for the task.

All ignores ListOfProdID and feeds all components contained (use LimitCount to limit the number of components to be fed).

Simultaneous allows resources to pick up listed components at the same time. By default, a call is sent for one resource to come and pick up a component. If you enable Simultaneous, there will be concurrent calls, thereby allowing resources to pick up components without waiting on one another to complete the task.

LimitCount[1)] limit the number of components to be fed. When left empty, feed all matching components by ProdID or All.

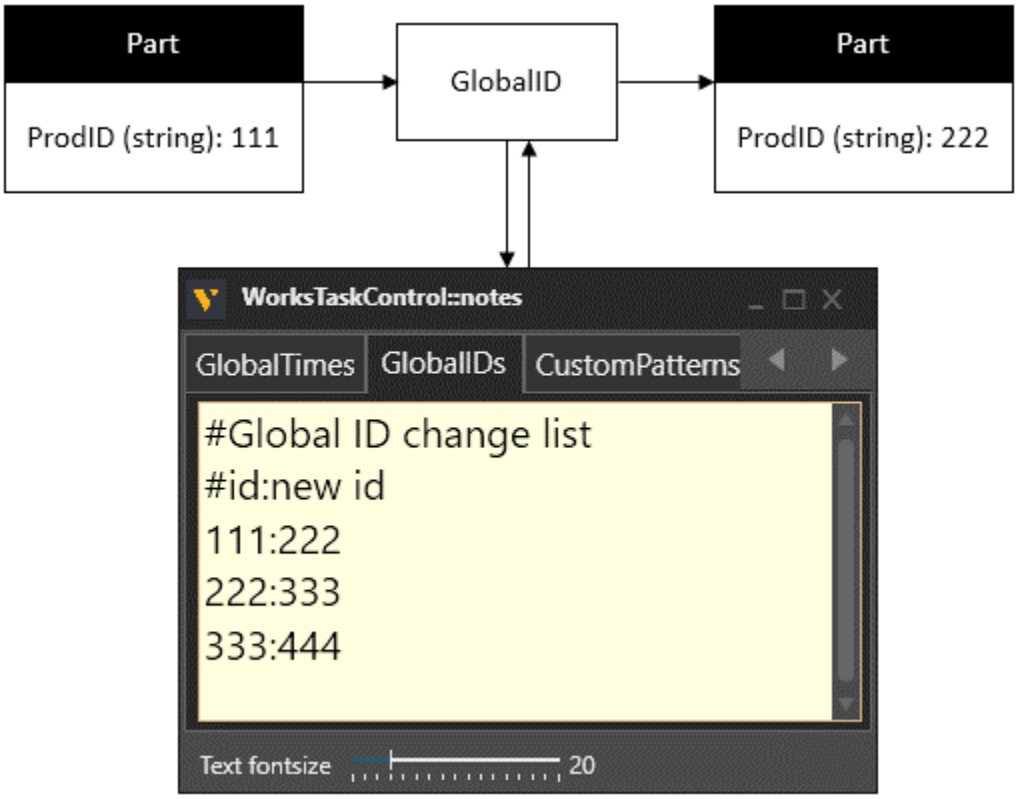
See Also

- Need
- NeedCustomPattern
- NeedPattern
- Pick
- Place

GlobalID

A GlobalID task allows you to change the ProdID value of a component by referring to the GlobalIDs note of a Works Task Controller.

The syntax for changing an id is `<ProdId>:<New ProdID>`.



Properties

No additional properties.

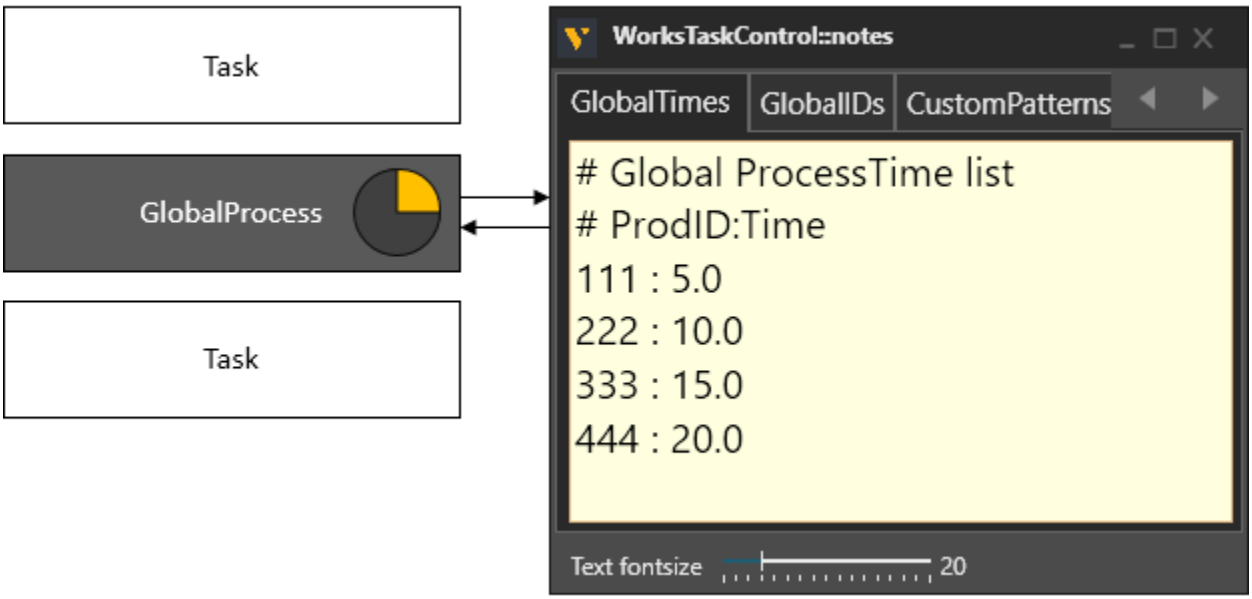
See Also

- ChangeID
- GlobalProcess
- ProdIDFromList

GlobalProcess

A GlobalProcess task allows you to execute a time-consuming process by referring to the GlobalTimes note of a Works Task Controller.

The syntax for a process time is `<ProdId>:<seconds>`.



Properties

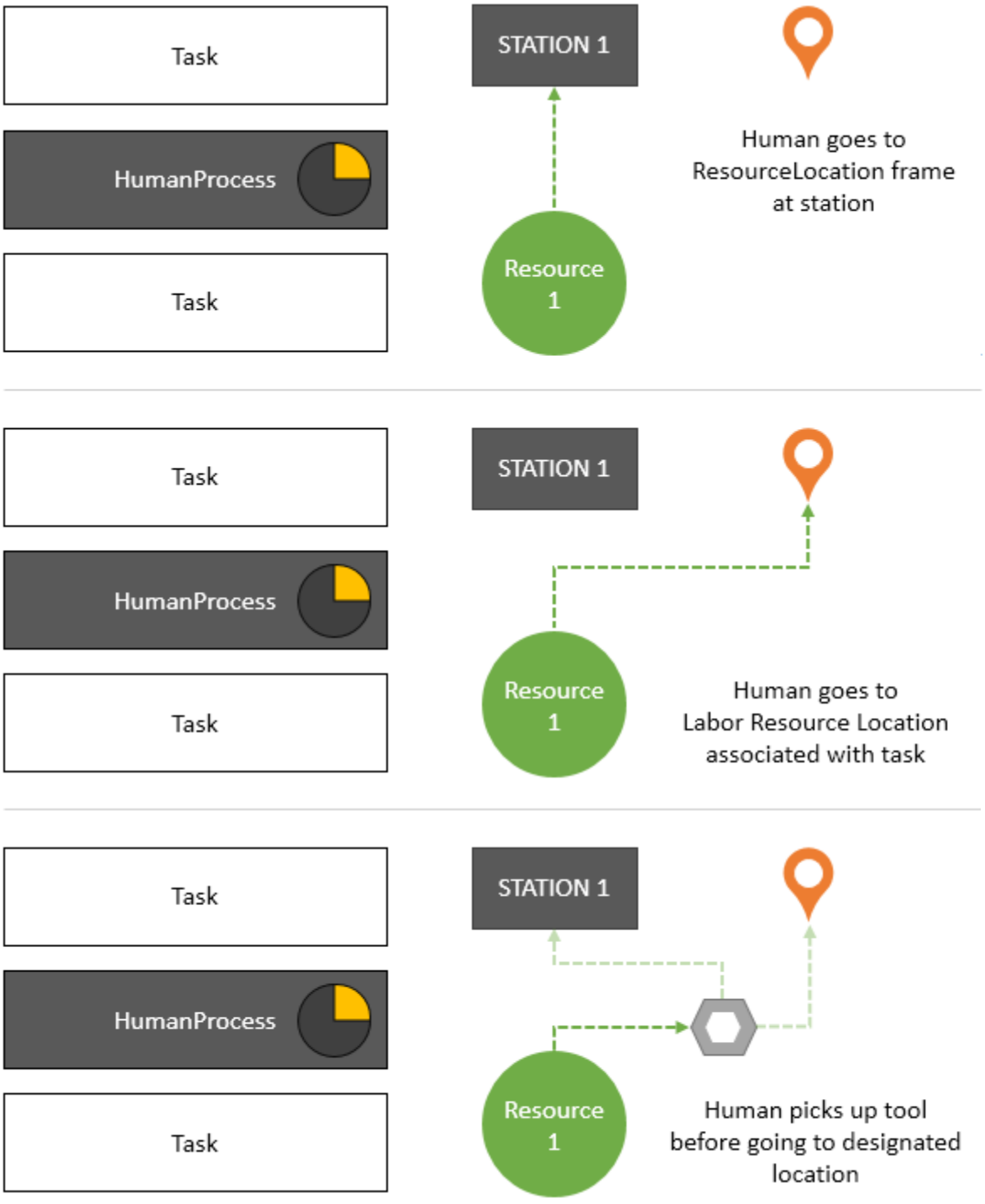
No additional properties.

See Also

- ChangeID
- GlobalID
- DummyProcess
- HumanProcess
- RobotProcess

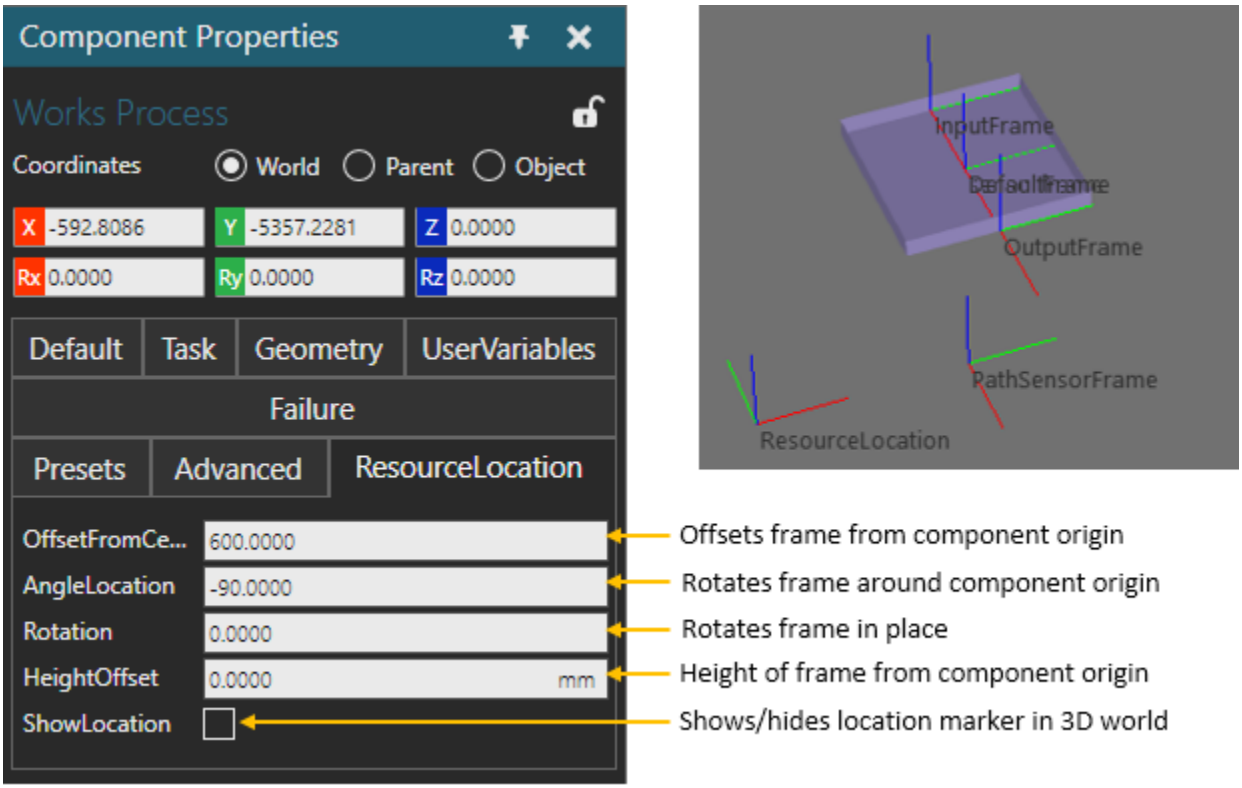
HumanProcess

A HumanProcess task allows you to call a human resource to execute a time-consuming process.

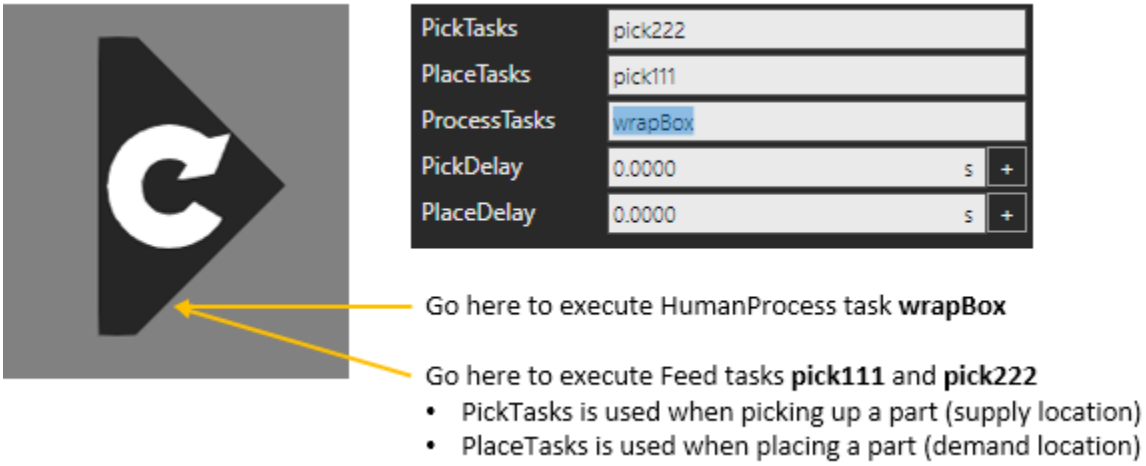


Location

By default, a HumanProcess task is executed at the ResourceLocation frame of a Works Process component. You can use the ResourceLocation properties of a Works Process component to edit the position of that frame as well as its visibility.



You have the option of using a Labor Resource Location component to specify where a human executes an assigned process. You also have the option to dictate pick and place locations for Feed tasks executed by a human.



Properties

ProcessTime[1,2,3)] defines the duration of process (in seconds).

TaskName defines a name that allows you to reference the HumanProcess task and assign it to a resource as well as a Labor resource location.

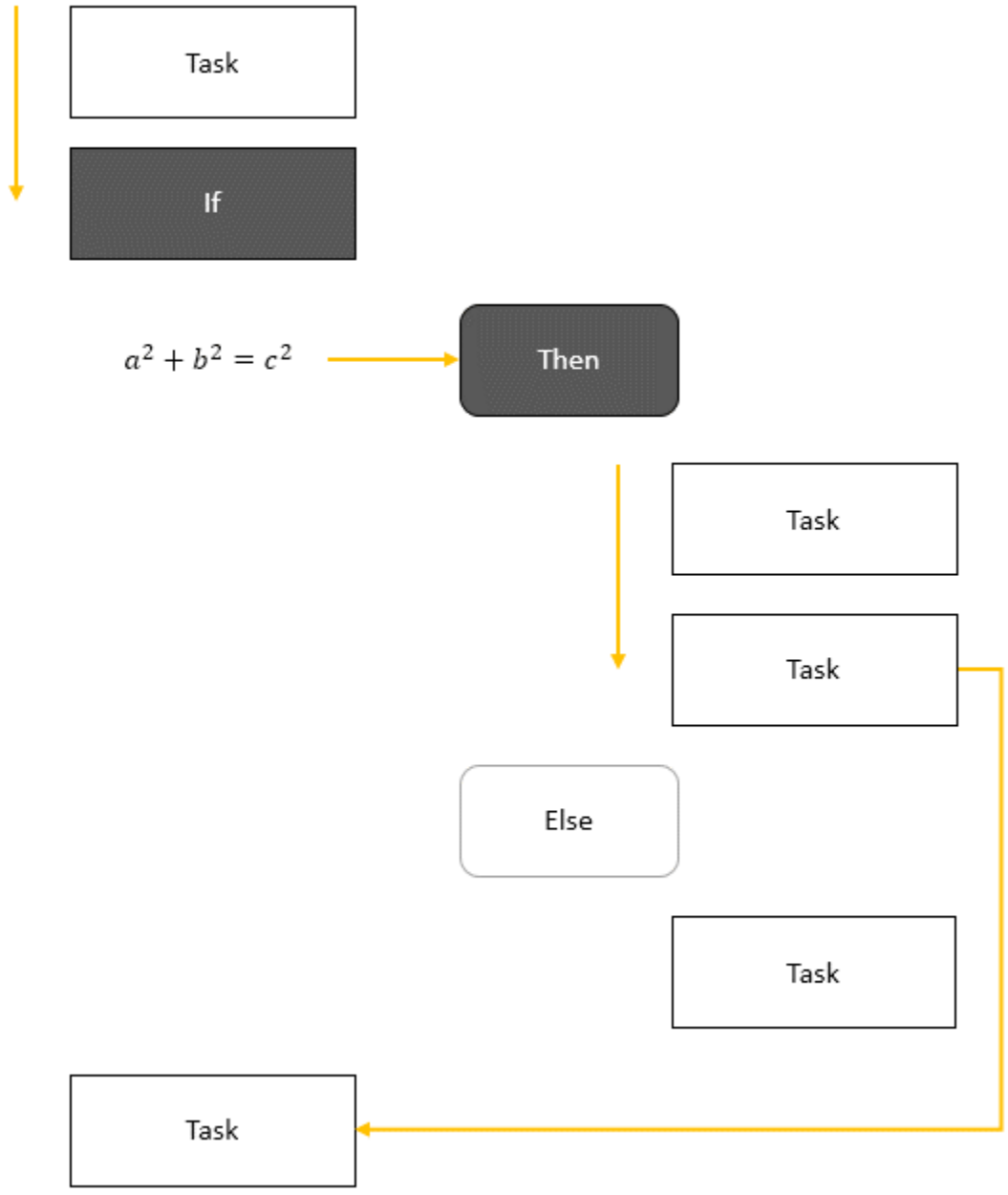
ToolName identifies a tool component by name that should be used to execute the task. For example, you could reference the name of a pallet jack or forklift.

See Also

- DummyProcess
- Feed
- MachineProcess
- Need
- RobotProcess

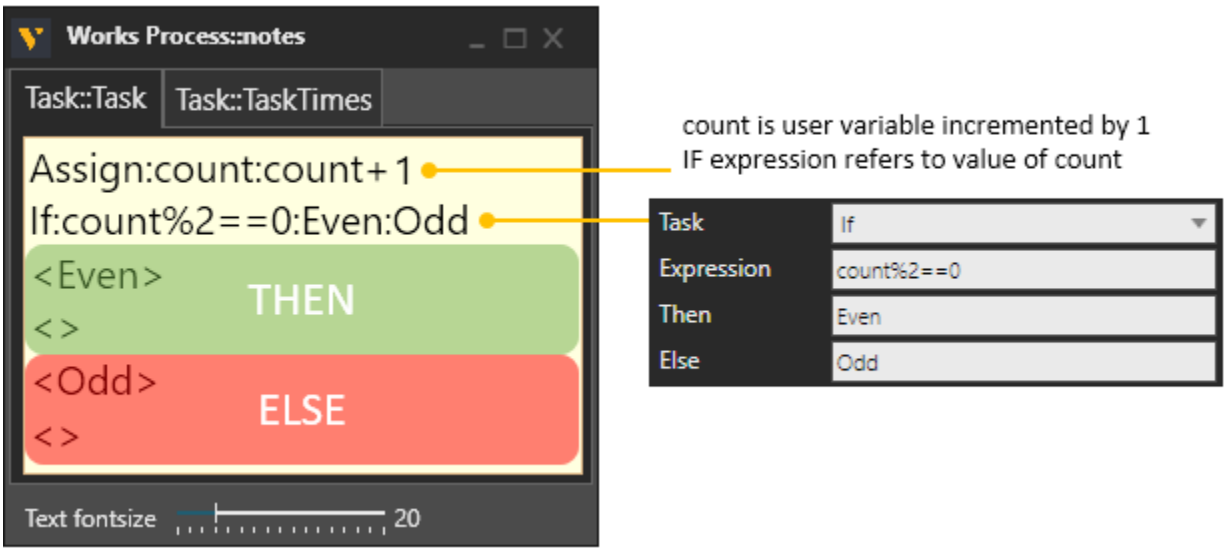
If

An If task allows you to execute one of two conditional sets of tasks based on a given expression. That is, the If task will execute one set of tasks and ignore the other set. A derivative of an If task is an IfProdID task, which is used to execute one conditional set of tasks based on a ProdID value.



Syntax

When you create an If task, the task is listed with its expression and the names for its Then and Else tags in the Task note of a Works Process component. Immediately following that line are the conditional tags for the Then and Else properties.



Conditional Tags

Then and Else are defined by a set of conditional tags. The start tag defines the condition name, and the end tag marks the end of that condition. Tasks inserted inside these tags are executed by the If task depending on the result of its expression.

If the expression returns a true value, the If task executes tasks inside its Then tags. If the expression returns a false value, the If task executes tasks inside its Else tags.

You can use "Then" and "Else" as conditional names. Generally, you would use names of subprocesses, ProdID values, or something self-explanatory. Remember that changing the name of a condition also requires updating the names referenced by the If task.

The position of tags in the Task note of a Works Process component does not matter since the Then and Else arguments of an If task, along with other types of conditional tasks, indicate the conditions to execute by name. Some users prefer to keep conditions at the bottom to improve readability and troubleshooting. Others prefer WYSIWYG.

```
#original
Assign:count:0
WarmUp:
TransportIn:111?222:True
<111>
ChangeProductMaterial:111:green
<>
<222>
ChangeProductMaterial:222:red
<>
Assign:count:count+1
TransportOut::True
```

```
#same process as original
Assign:count:0
WarmUp:
TransportIn:111?222:True
Assign:count:count+1
TransportOut::True

<111>
ChangeProductMaterial:111:green
<>
<222>
ChangeProductMaterial:222:red
<>
```

Tip: Empty lines and lines starting with a hash (#) are not executed in a process.

Potential Conflicts

Be aware that using ProdID values for naming conditions might conflict with conditional tags used by other tasks and tasks that return a ProdID value in the script executing those tasks.

```
#original
Assign:count:0
WarmUp:
TransportIn:111?222:True
<111>
ChangeProductMaterial:111:green
<>
<222>
ChangeProductMaterial:222:red
<>
Assign:count:count+1
If:count%2==0:111:222
<111>
Remove:111:False
<>
<222>
Remove:222:False
<>
TransportOut::True
```

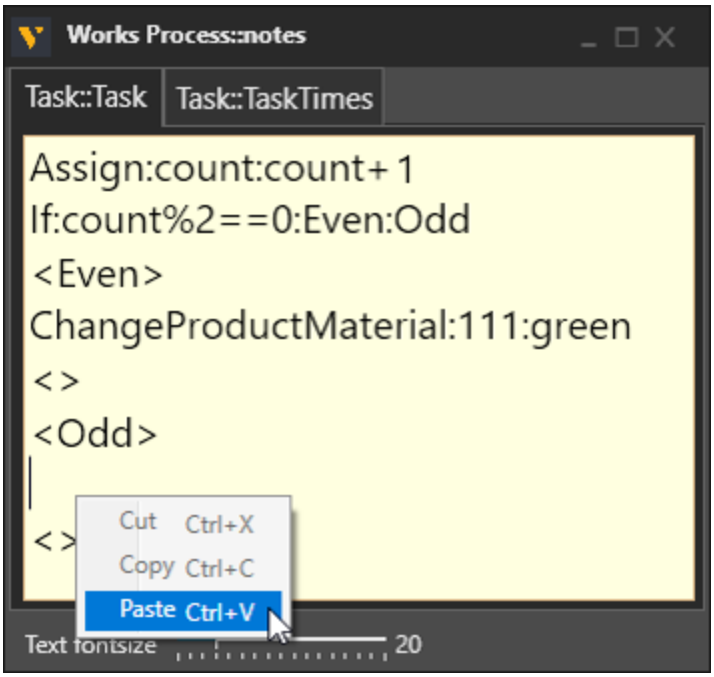
```
#what is really happening
Assign:count:0
WarmUp:
TransportIn:111?222:True
Assign:count:count+1
If:count%2==0:111:222
TransportOut::True

#both TransportIn and If call these
<111>
ChangeProductMaterial:111:green
Remove:111:False
<>
<222>
ChangeProductMaterial:222:red
Remove:222:False
<>
```

```
#fix by renaming If conditional tags
Assign:count:0
WarmUp:
TransportIn:111?222:True
<111>
ChangeProductMaterial:111:green
<>
<222>
ChangeProductMaterial:222:red
<>
Assign:count:count+1
If:count%2==0:X:Y
<X>
Remove:111:False
<>
<Y>
Remove:222:False
<>
TransportOut::True
```

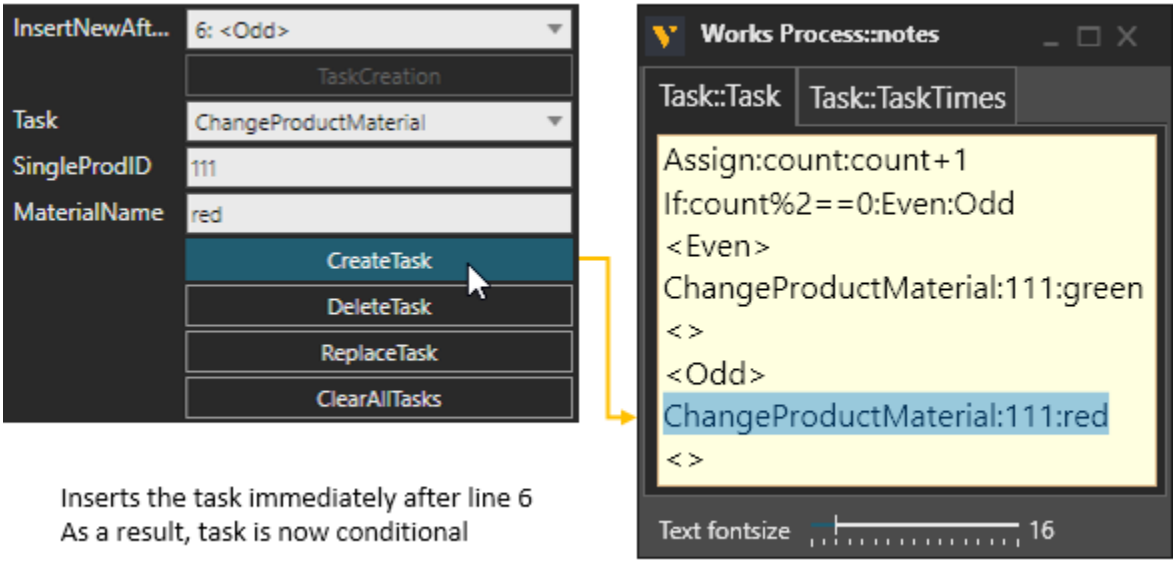
```
#same fix with conditions at end
Assign:count:0
WarmUp:
TransportIn:111?222:True
Assign:count:count+1
If:count%2==0:X:Y
TransportOut::True
<111>
ChangeProductMaterial:111:green
<>
<222>
ChangeProductMaterial:222:red
<>
<X>
Remove:111:False
<>
<Y>
Remove:222:False
<>
```

Nesting

To insert tasks inside Then and Else tags, you can use the Task note editor. Generally, this approach is used when you are very familiar with the syntax of tasks and their execution.



Another option is to use the InsertNewAfterLine property when you are creating tasks.



Tip: In most cases, you would combine both approaches. Be aware that you cannot nest an If task inside another If task.

Properties

Expression[1,2)] defines an expression that is evaluated to return a true or false value.

You must use Python syntax for writing the expression. You can refer to properties of a Works Process component by name as well as its variables to reference their values. For help in writing expressions, refer to Python 2.7 documentation.

https://docs.python.org/2/reference/expressions.html

Then defines the name of the conditional tags referred to when Expression is true.

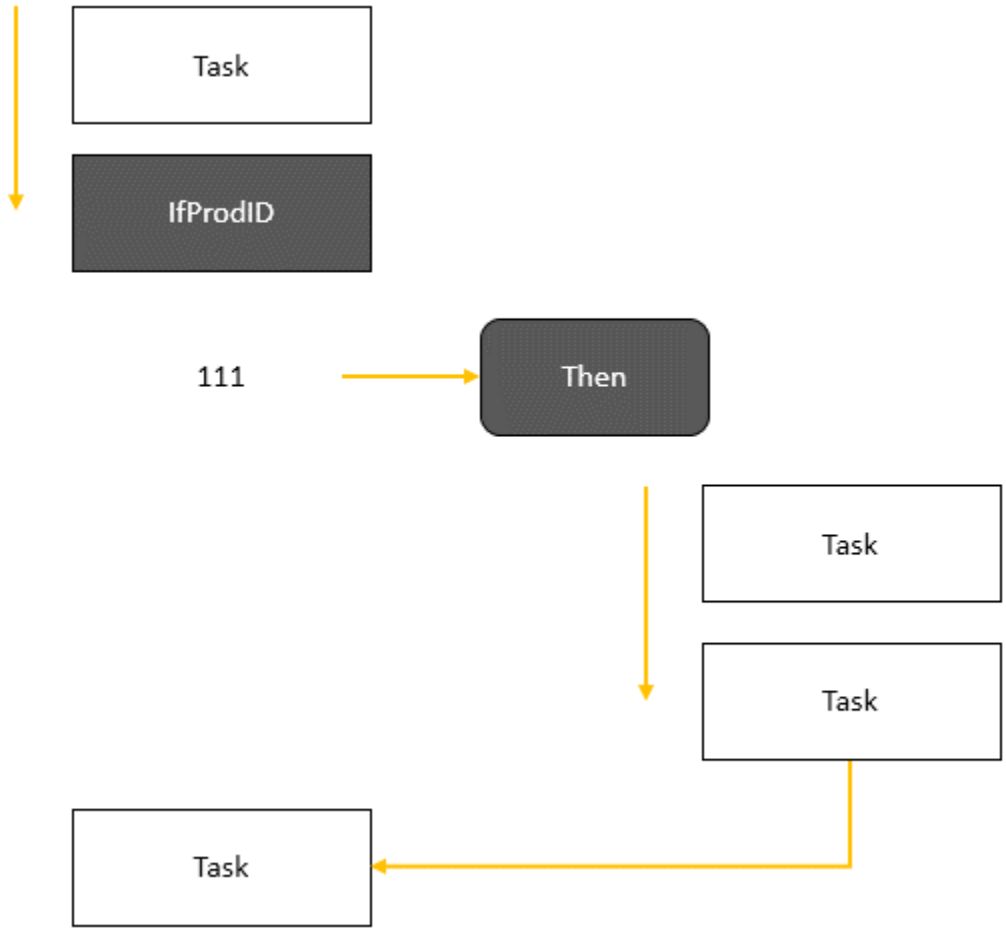
Else defines the name of the conditional tags referred to when Expression is false.

See Also

- Assign
- IfProdID
- Loop
- Need
- TransportIn
- WaitForOrder
- WaitProperty
- WaitSignal

IfProdID

An IfProdID task allows you to execute a conditional set of tasks for a ProdID value.



Properties

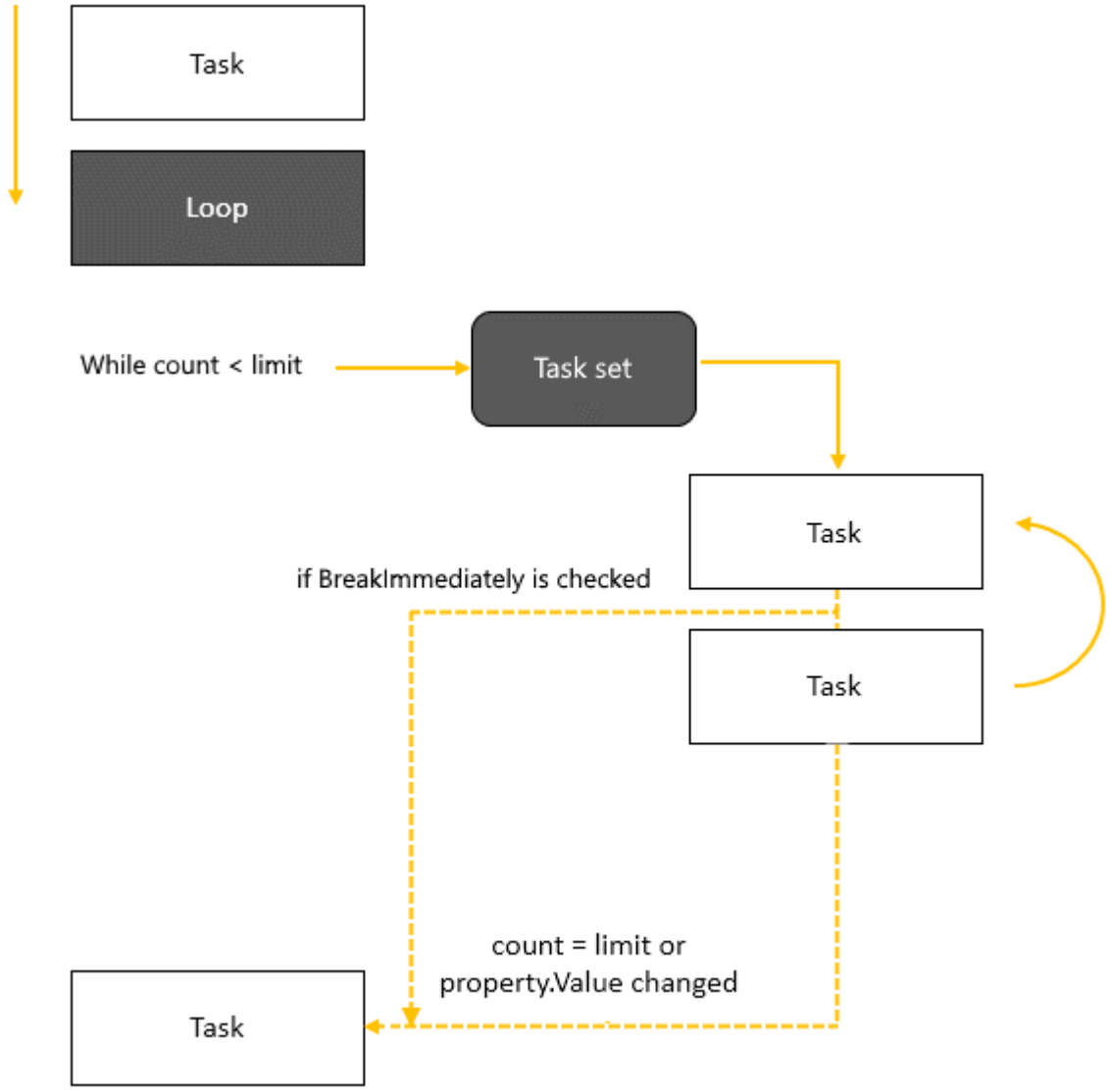
SingleProdID identifies the component by ProdID value.

See Also

- If
- Loop

Loop

A Loop task allows you to execute a conditional set of tasks repeatedly based on a given count and property for exiting the loop (optional). In programming context, a Loop task is the equivalent of a While statement in which the break property value change is tested.



Properties

SingleProdID defines the name of the conditional tags referred to by the loop. Each iteration of the loop will execute tasks inside those tags.

You will need to add the tags yourself using the Task note editor of a Works Process component.

```
#makes 1 cylinder and transports it out... does this 5 times
Loop:makeBatch:5:stopBatching
<makeBatch>
Create:Cylinder:
TransportOut::True
<>
Delay:20.0
```

LoopCount[1)] defines the number of times to execute the loop.

LoopBreakPropertyName identifies a property in a Works Process component whose change in value will break the loop.

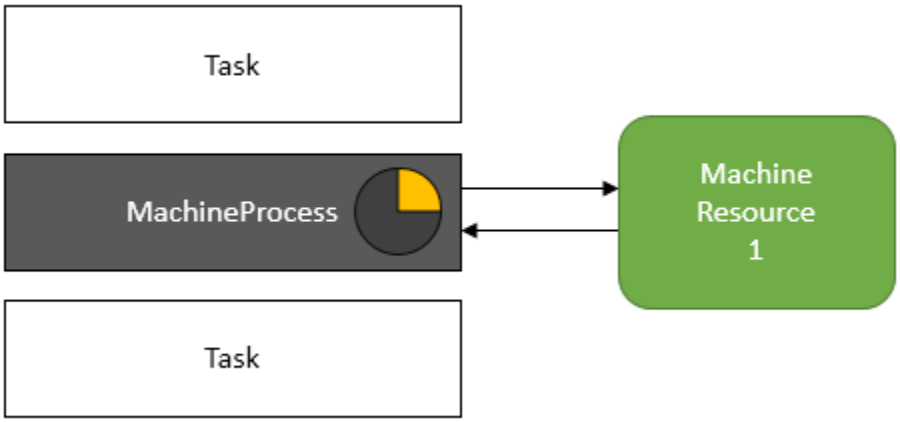
BreakImmediately if checked, the loop will exit instantly when the break property value changes instead of continuing to run set of conditional tasks (loop block) until to the end and then exit the loop.

See Also

- If
- IfProdID
- WaitSignal
- WaitProperty
- WriteProperty
- WriteSignal

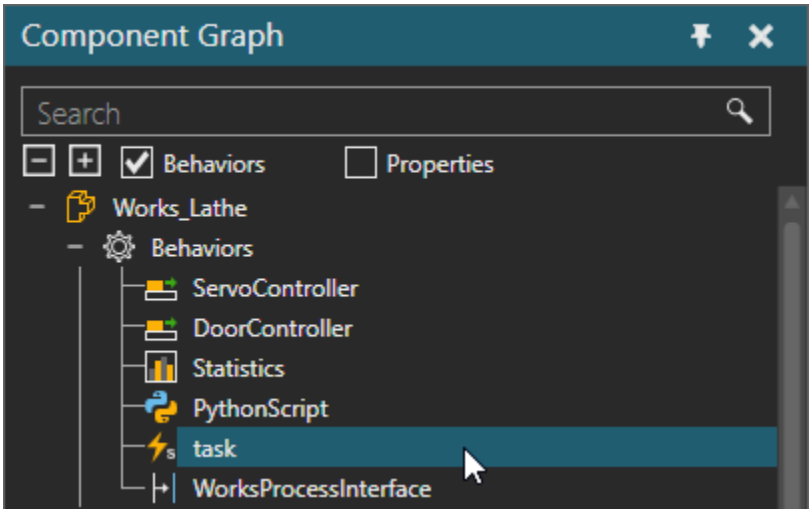
MachineProcess

A MachineProcess task allows you to execute a process in a component modeled as a machine. For example, you can execute a machine tending process in a lathe machine.

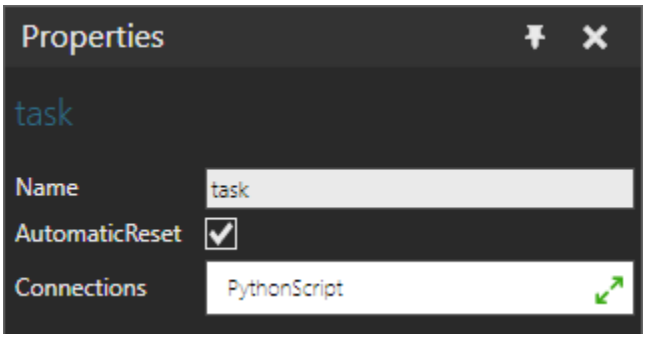


Requesting

The machine must have a String Signal named "task" in order to receive the request.

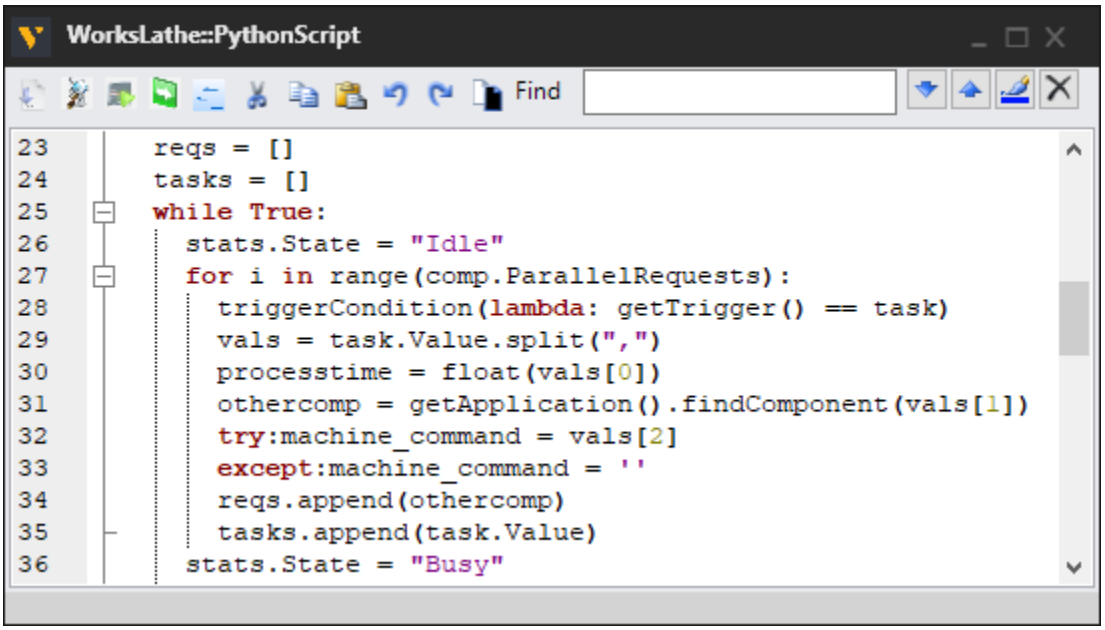


Generally, the task signal is connected to a Python script and acts as a trigger in that script.



Signal Value

The value of the task signal is a set of arguments for completing a MachineProcess task.



Line 23 List of components requesting machine process.

Line 24 List of machine process tasks. Each task is the value of the task signal.

Line 28 The task signal is used as a trigger in OnRun event.

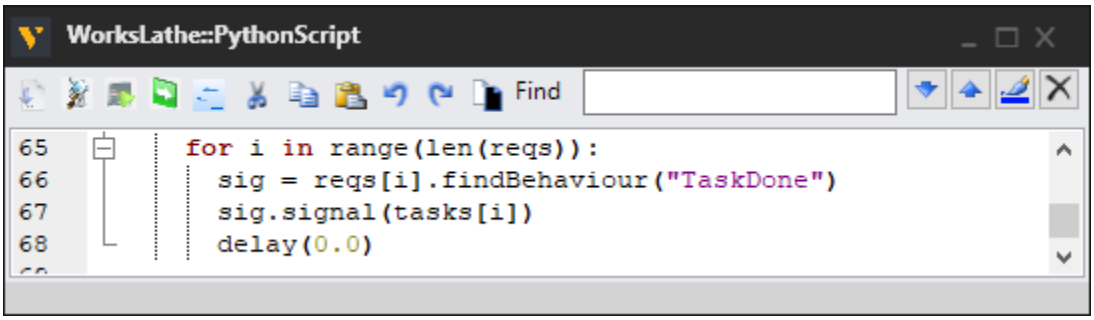
Line 29 – 33 Value of task signal is parsed to get arguments for process time, component requesting the task, and a command for executing a specific process. If no machine command is given, argument is empty string.

Line 34 Records the component that requested the task. This component must be told by the machine when it completes the task.

Line 35 Records the value of task. This value must be used to communicate the completion of that task.

Completion

A MachineProcess task is not completed in a Works Process component until it signals the task using its TaskDone signal.



Line 65 – 67 After tasks are done, loop through list of requests, find TaskDone signal in requesting component, and then signal the value of the task it sent the machine.

Template

Here is a snippet for implementing a machine process in a component using a Python script.

```
from vcScript import *

#get or create task signal and connect it to script
app = getApplication()
comp = getComponent()
task = comp.getBehaviour("task")
if not task:
  task = comp.createBehaviour(VC_STRINGSIGNAL, "task")
  script = comp.getBehaviour("PythonScript")
  task.Connections = [script]

#create task list
tasks = []

def getTaskData(task):
  '''Returns the task arguments for machine process.
  Value is <process time>,<component requesting task>,<machine command>
  '''
  vals = task.split(",")
  pTime = float(vals[0])
  req = app.findComponent(vals[1])
  m_cmd = vals[2]
  return pTime, req, m_cmd
...
def signalTaskDone(c,task):
  '''Signals the completion of machine process task.
  Component requesting task is used to signal task completion
  '''
  signal = c.findBehaviour("TaskDone")
  signal.signal(task)

#this example uses signal events to schedule and execute tasks
#some other examples in eCat use triggerCondition in OnRun
def OnSignal(signal):
  if signal == task and task.Value != "":
    tasks.append(task.Value)
    resumeRun()



...
```

```
...
#loop is suspended until there is a scheduled task
def OnRun():
  tasks[:]
  while app.Simulation.IsRunning:
    suspendRun()
    if tasks:
      for t in tasks:
        pTime, req, m_cmd = getTaskData(t)

        #simple example of using MachineCommand property for customization
        if m_cmd == "Hello":
          print "Hello\t", app.Simulation.SimTime
        elif m_cmd == "World":
          print "World\t", app.Simulation.SimTime
        else:
          print "Default\t", app.Simulation.SimTime

        #refer to Works Lathe in eCat for example on scaling process time
        delay(pTime)

        #mark completion of task by removing it from list and signal task is done
        tasks.remove(t)
        signalTaskDone(req,t)
```

Properties

SingleCompName identifies the component by name.

ProcessTime[1,2,3)] defines the duration of the process (in seconds).

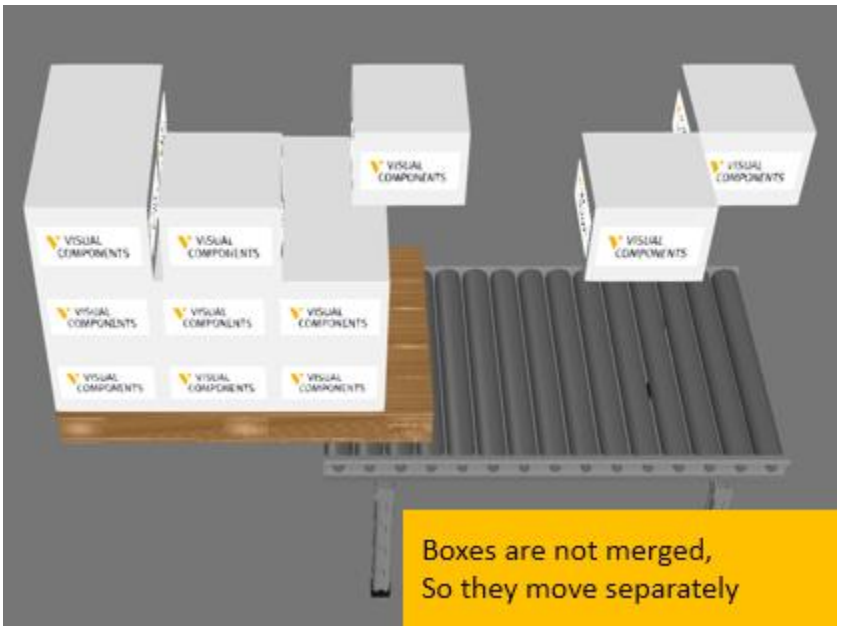
MachineCommand allows you to pass additional arguments for executing the process.

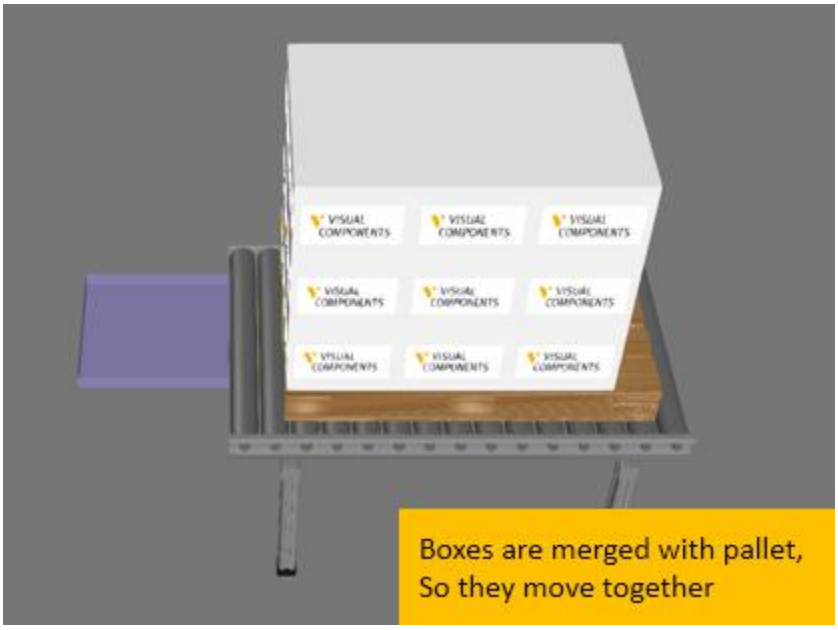
See Also

- ChangeID
- Create
- Feed
- Delay
- DummyProcess
- HumanProcess
- Need
- Pick
- Place
- ProcessSteps
- Remove
- RobotProcess

Merge

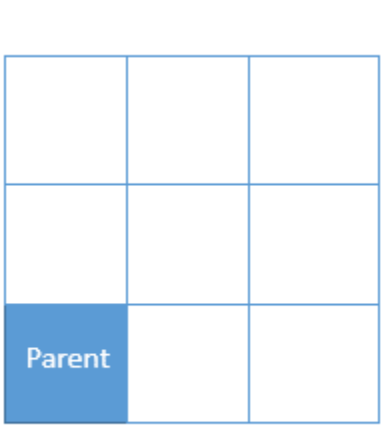
A Merge task allows you to attach components contained in a Works Process component to one another. Generally, a Merge task is used to bundle components and make them move together. For example, you can merge a stack of parts with a pallet, and then transport out the pallet to move the entire stack with it. Afterward, you could use a Split task to detach parts in the stack to pick them one by one.



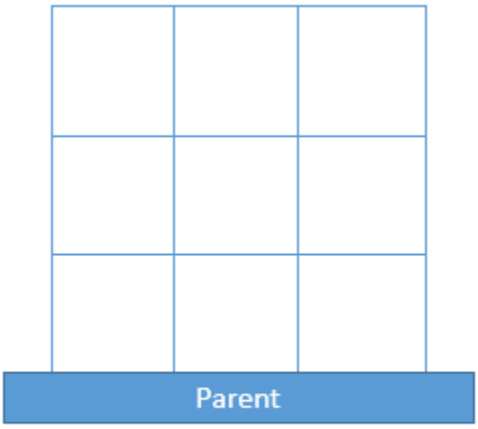


Properties

ParentProdID identifies the parent component by ProdID value. Components will be attached to the root node of that component. If you are attaching components with the same ProdID, the first component of that type in the Works Process component will be the parent, for example the first component in a pattern.



Stack



Stack on Pallet

ListOfProdID allows you to filter which components are merged with ParentProdID.

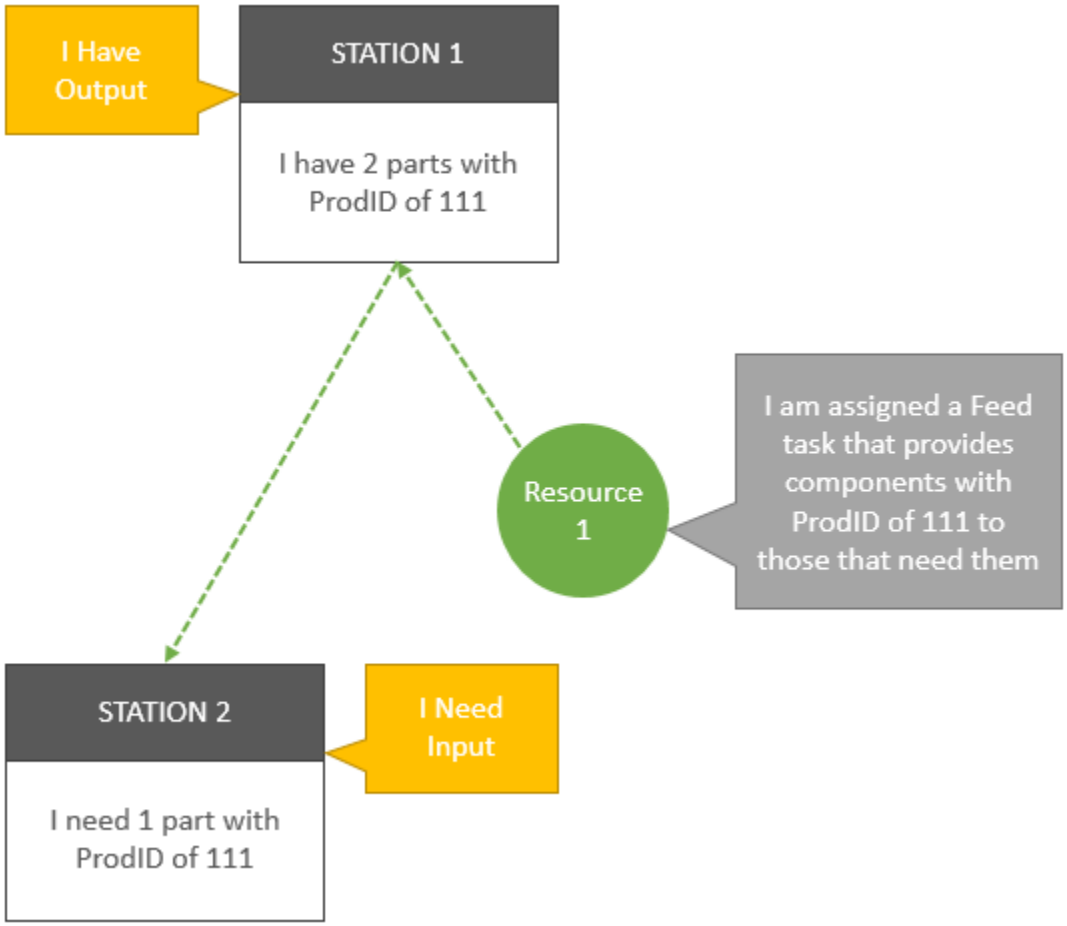
All means every component contained in the process will be merged with ParentProdID.

See Also

- Split
- TransportOut

Need

A Need task allows you to demand one or more types of components be delivered to a Works Process component by resources assigned Feed tasks. A Feed task is supply, and a Need task is demand. A Feed task must be executed at a different Works Process component to complete a Need task.



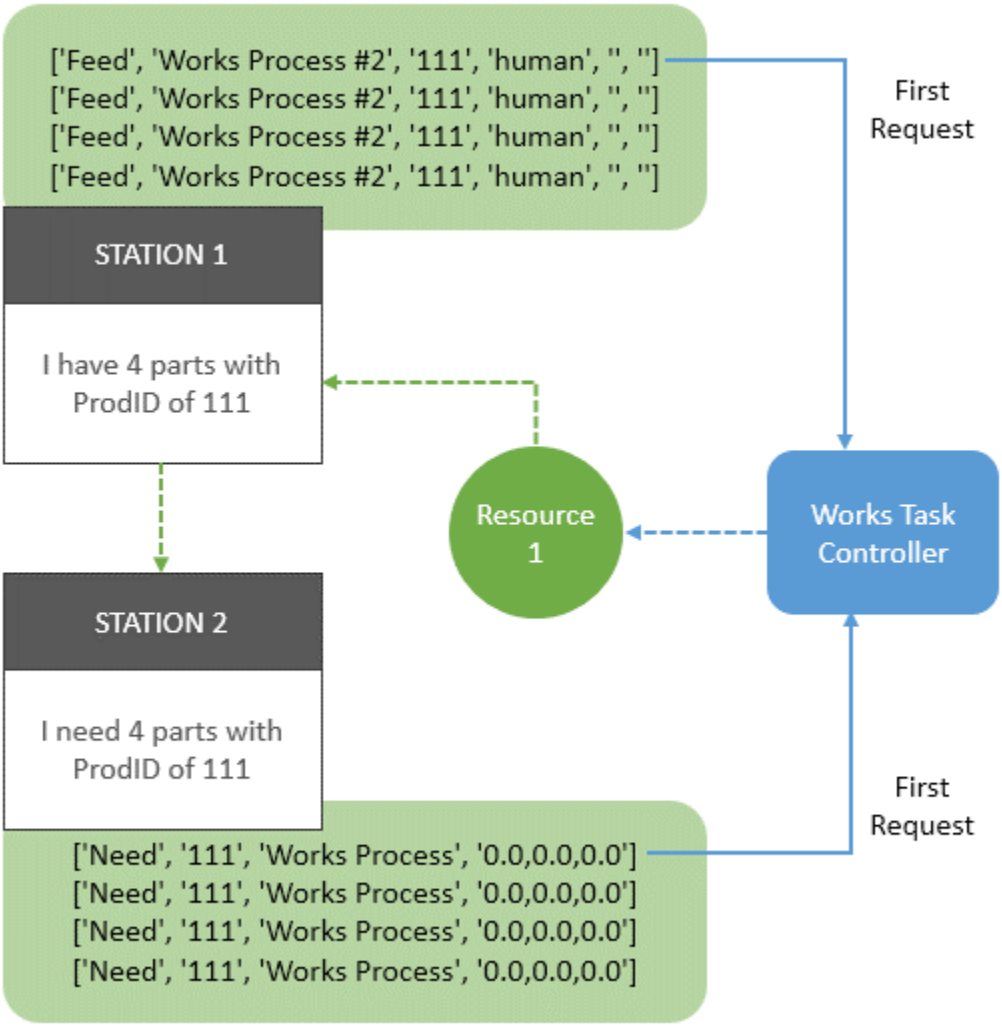
Derivatives of a Need task are NeedPattern and NeedCustomPattern tasks.

Request

A Need task along with its derivatives can request multiple components. Components are not requested at once rather one-by-one. Each request is sent to a Works Task Controller that identifies the ProdID and XYZ step values for placing the needed component.

```
Output
[['Need', '111', 'Works Process', '0.0,0.0,0.0']]
[['Need', '111', 'Works Process', '0.0,100.0,0.0']]
[['Need', '111', 'Works Process', '100.0,0.0,0.0']]
```

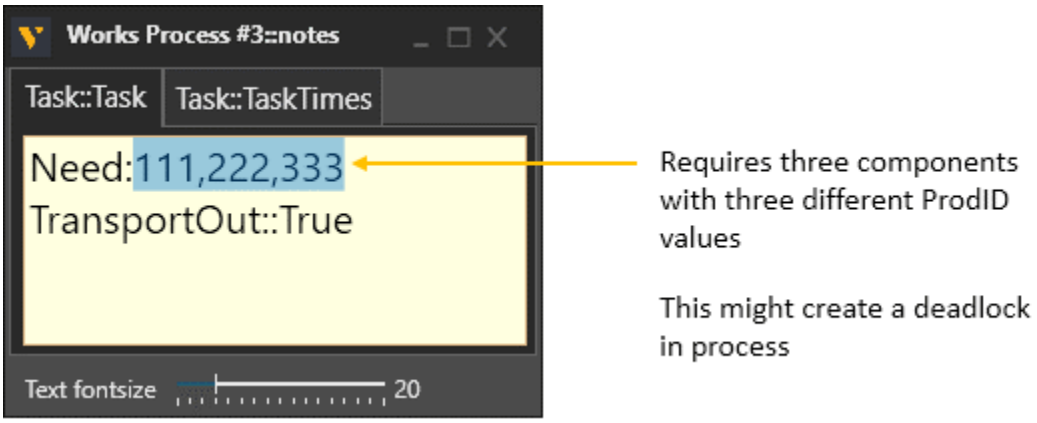
The Works Task Controller considers each request a Need task, so the request/task is added to a queue and executed when there is a corresponding Feed task to complete the request. This process of requesting a component continues until all the needed components are delivered by a resource executing Feed tasks. For example, a NeedPattern of 2x2 is considered four Need tasks in the scripts responsible for completing the pattern.



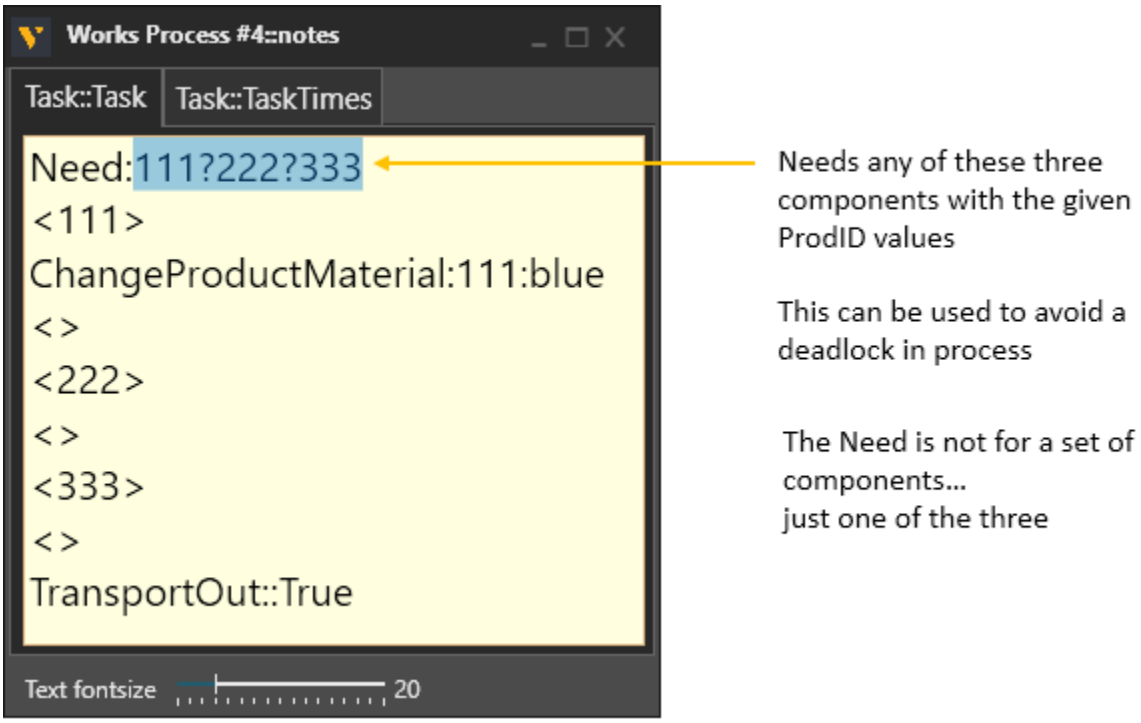
Properties

ListOfProdID[1,2)] identifies the needed components by ProdID value.

Use a comma (,) as a delimiter to request a set of components. For example, a Need task would not be completed until each needed component is received at its Works Process component.



Use a question mark (?) as a delimiter to specify conditions for more than one type of component. That will generate conditional tags for each ProdID, thereby allowing you to run different sets of tasks for needed components.



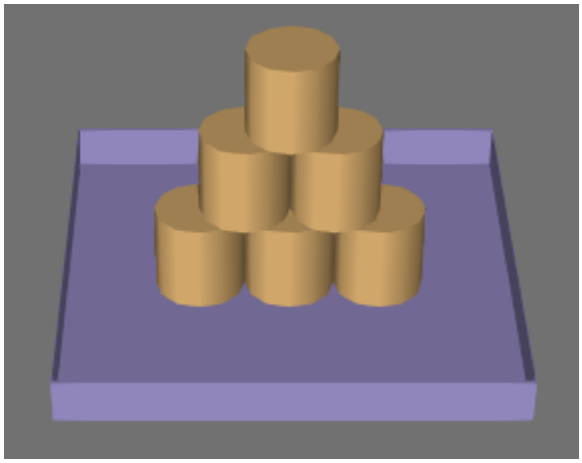
See Also

- Create
- Feed
- If
- NeedCustomPattern
- NeedPattern
- Pick
- Place
- ProcessSteps
- TransportIn

NeedCustomPattern

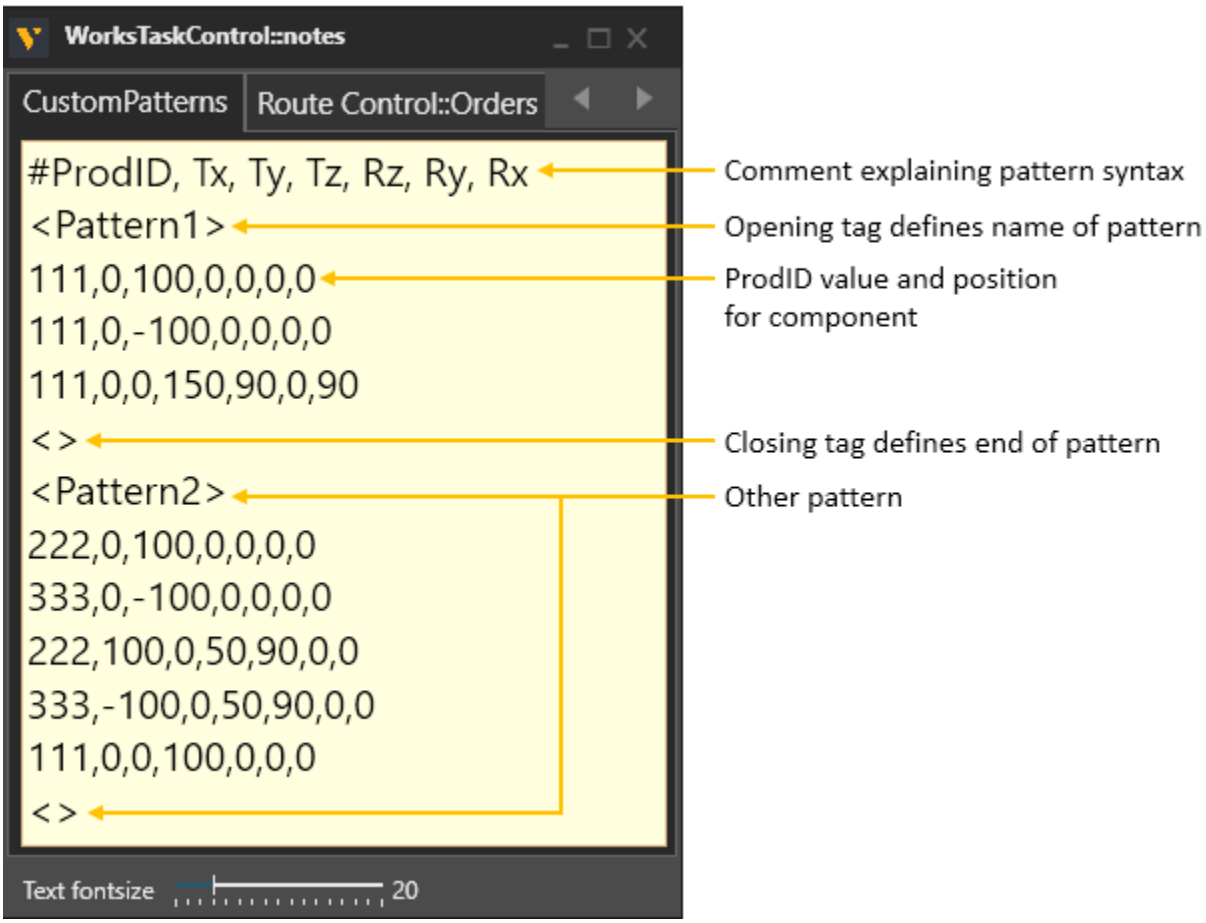
A NeedCustomPattern task allows you to demand your own pattern of components.

```
<Pyramid>
#first layer
111,0,0,0,0,0,0
111,0,100,0,0,0,0
111,0,200,0,0,0,0
#second layer
111,0,50,100,0,0,0
111,0,150,100,0,0,0
#third layer
111,0,100,200,0,0,0
<>
```



Pattern

The pattern is defined in the CustomPatterns note of a Works Task Controller component.



Properties

PatternName[1,2)] identifies the pattern you want to use by name.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

Tip: Generally, a range is modified for testing the position of components in the pattern.

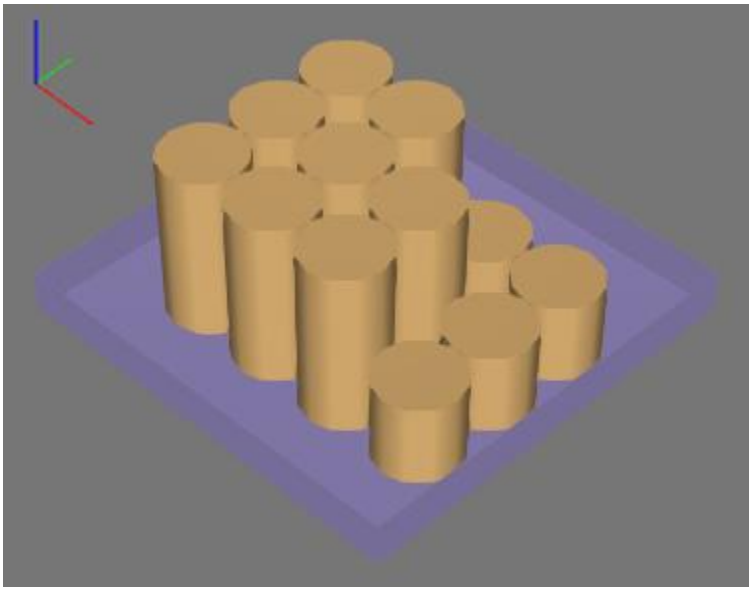
Simultaneous allows resources to place needed components at the same time. By default, a call is sent for one resource to come and place a component. If you enable Simultaneous, there will be concurrent calls, thereby allowing resources to place components without waiting on one another to execute the task.

See Also

- Create
- Feed
- If
- Need
- NeedPattern
- Pick
- Place
- PlacePattern
- ProcessSteps
- TransportIn

NeedPattern

A NeedPattern task allows you to demand a pattern of components.



Properties

SingleProdID[1,2)] identifies the component you need by ProdID value.

AmountX[1)] defines the number of components to create along X-axis.

AmountY[1)] defines the number of components to create along Y-axis.

AmountZ[1)] defines the number of components to create along Z-axis.

StepX[1)] defines the spacing of components along the X-axis.

StepY[1)] defines the spacing of components along the Y-axis.

StepZ[1)] defines the spacing of components along the Z-axis.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

Tip: Generally, a range is modified for testing the position of components in the pattern.

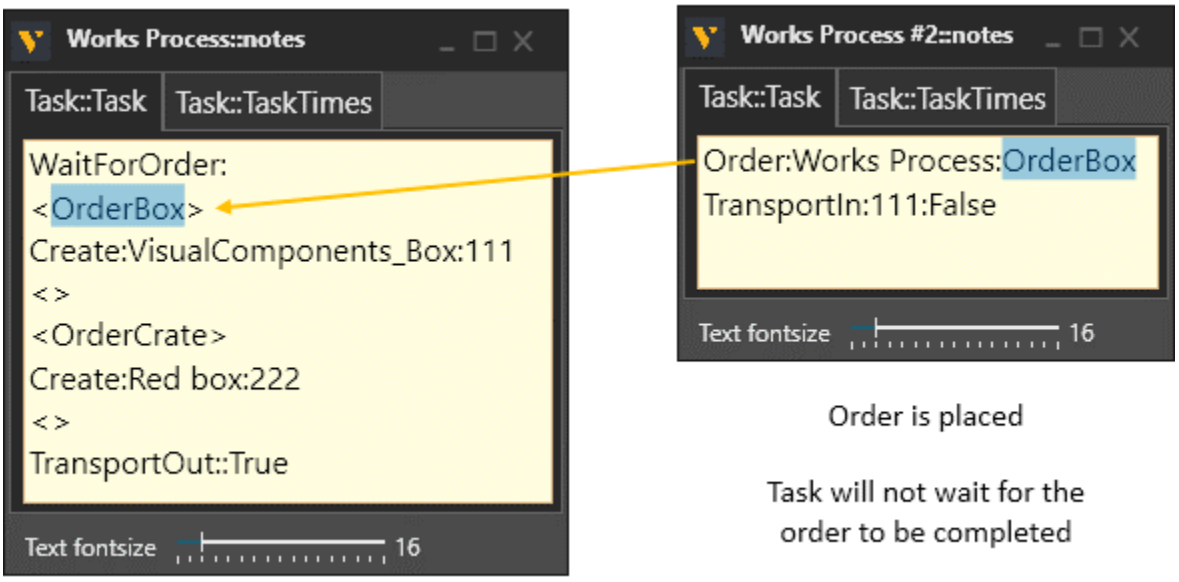
Simultaneous allows resources to place needed components at the same time. By default, a call is sent for one resource to come and place a component. If you enable Simultaneous, there will be concurrent calls, thereby allowing resources to place components without waiting on one another to execute the task.

See Also

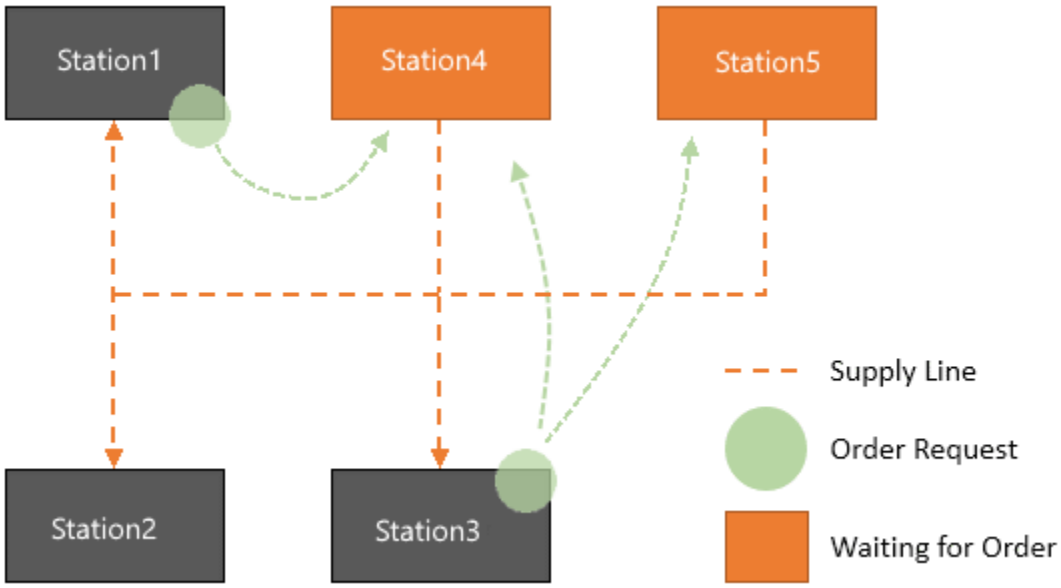
- Create
- Feed
- Need
- NeedCustomPattern
- Pick
- Place
- PlacePattern
- ProcessSteps

Order

An Order task allows you to place a work order / part demand at one or more Works Process components. A Works Process component receiving the order must be referenced by name and have a WaitForOrder defined in its task list to handle the order. The name of the order is used by the receiving Works Process and the order is placed in its order queue.



Generally, the order is placed by a Works Process that is downstream in a production line to a Works Process component that is upstream. This is also known as “pull” for parts.



Properties

ListOfCompNames identifies the Works Process component(s) by name. The order will be sent only to these components (Works Processes).

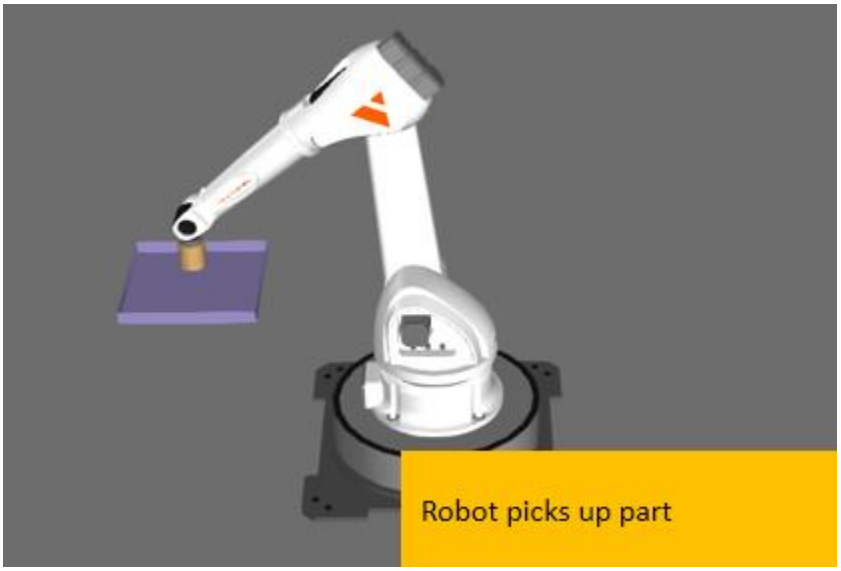
ListOfOrderNames[2)] defines the name of the order. To send multiple orders at once to the queue, separate the orders with a comma ",".

See Also

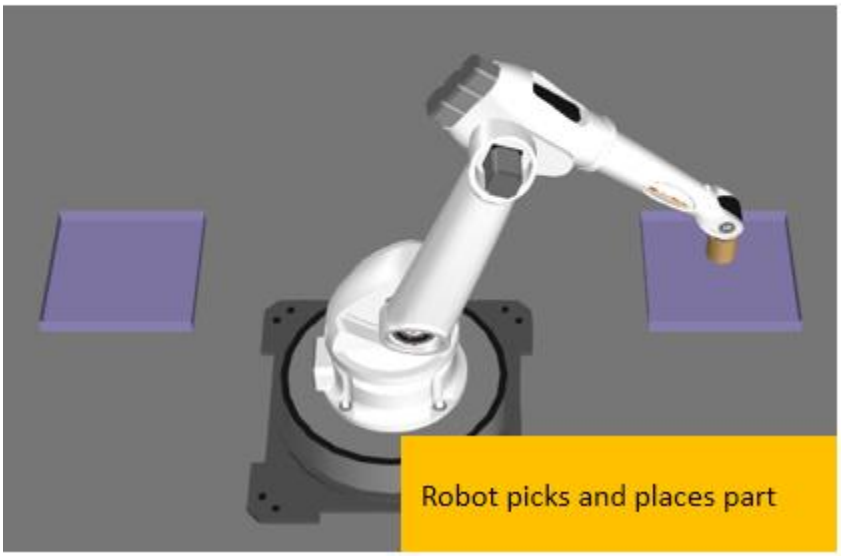
- WaitForOrder
- Sync
- Feed

Pick

A Pick task allows you to request a robot to pick up a component.

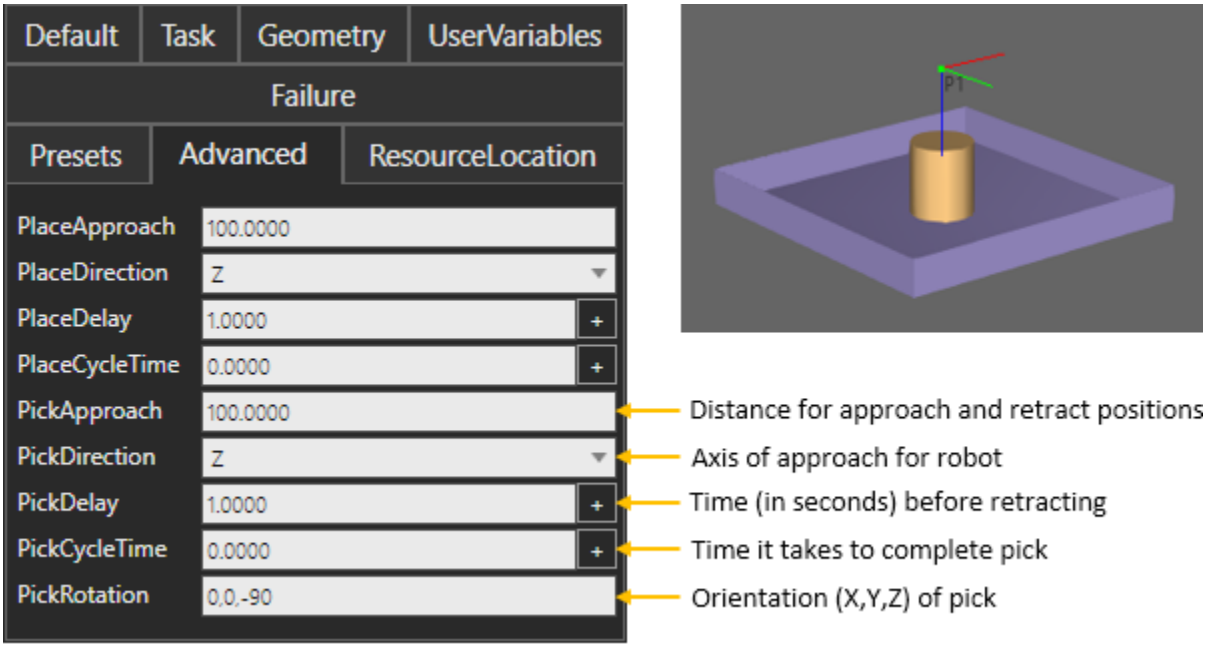


In all cases, a Pick task must be used to complete a Place task executed by the same robot. That requires Pick and Place tasks to be assigned using the SerialTaskList property of a Works Robot Controller.

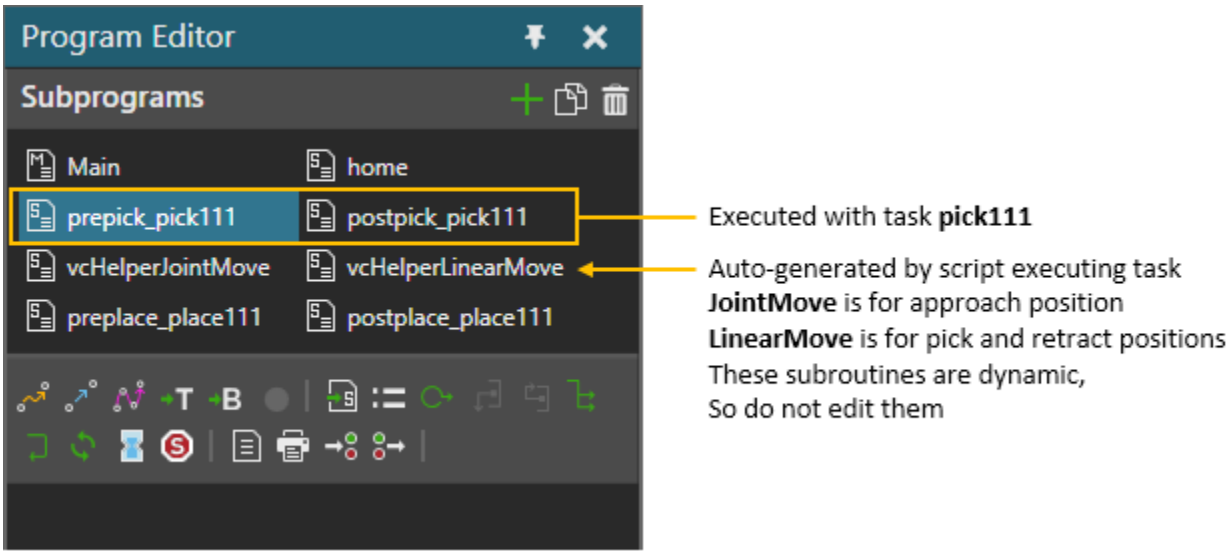


Customization

A Works Process component has a group of Advanced properties defining how resources execute pick type tasks, such as Feed and Pick.



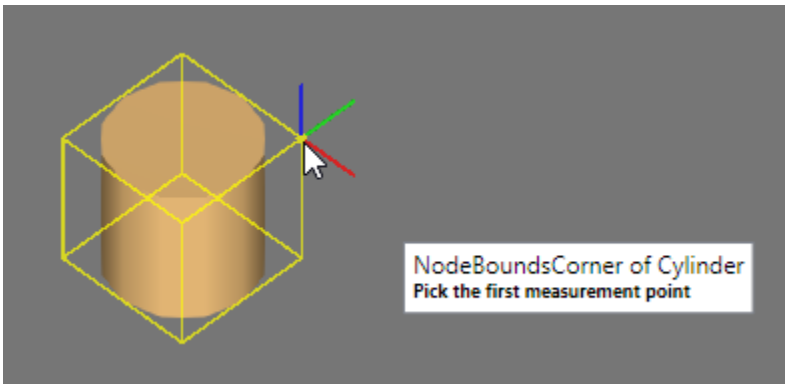
A robot executing a Feed or Pick task will have prepick and postpick subroutines in its program. A prepick sequence is executed before a robot picks a component. A postpick sequence is executed after a robot picks a component. Both sequences are executed automatically by the task itself, thereby allowing you to customize the execution of that task.



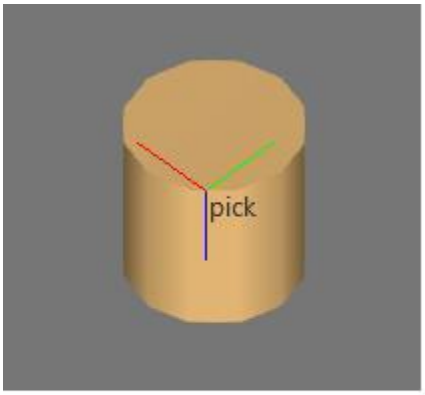
The name of a task is used to indicate its prepick and postpick sequences. If they do not exist in a robot program, they will be generated when the robot executes that task.

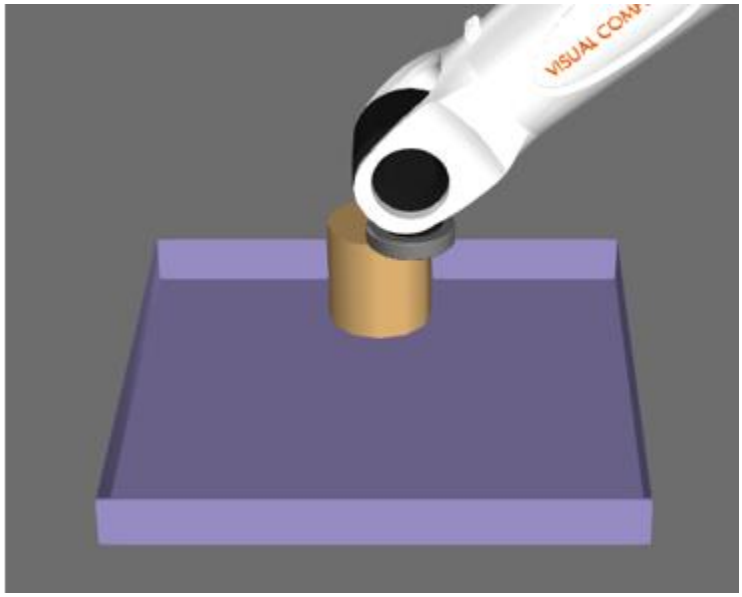
Pick Matrix

By default, a target matrix for Feed and Pick tasks is calculated using the bounding box of a target component. The Advanced::PickDirection property of a Works Process component affects the result. Generally, a robot would pick a component from its top face center.



In a component, you can create a Frame feature and name it "pick" to define its pick location.





Properties

SingleProdID identifies the component by ProdID value.

TaskName defines a name that allows you to reference the Pick task and assign it to a robot.

ToolName[2*)] identifies a tool component by name that should be used to execute the task. For example, you could reference the name of an end-of-arm tool for a robot.

2*) property value given with "#" is read only from the product that is going to be picked.

TCPName identifies a tool frame by name in the tool component used for the task.

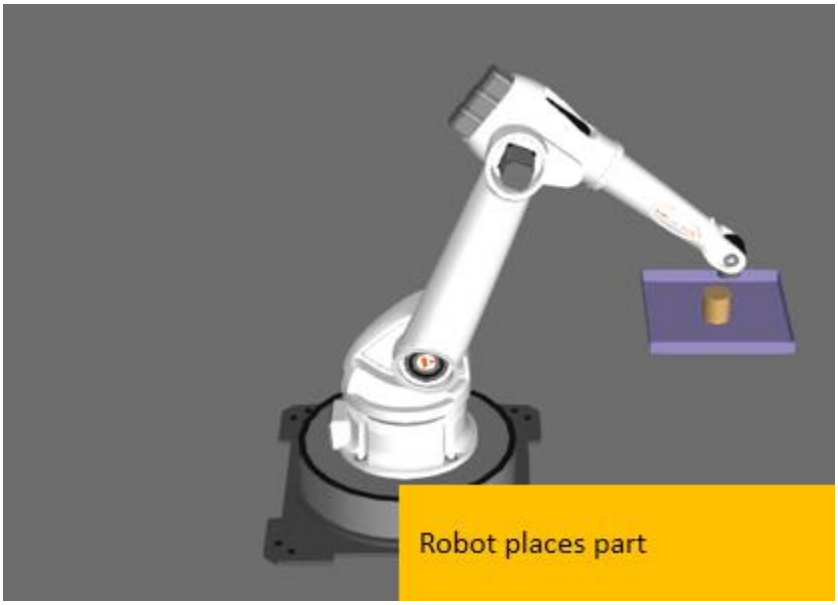
All requires every component matching SingleProdID to be picked in order to complete the task.

See Also

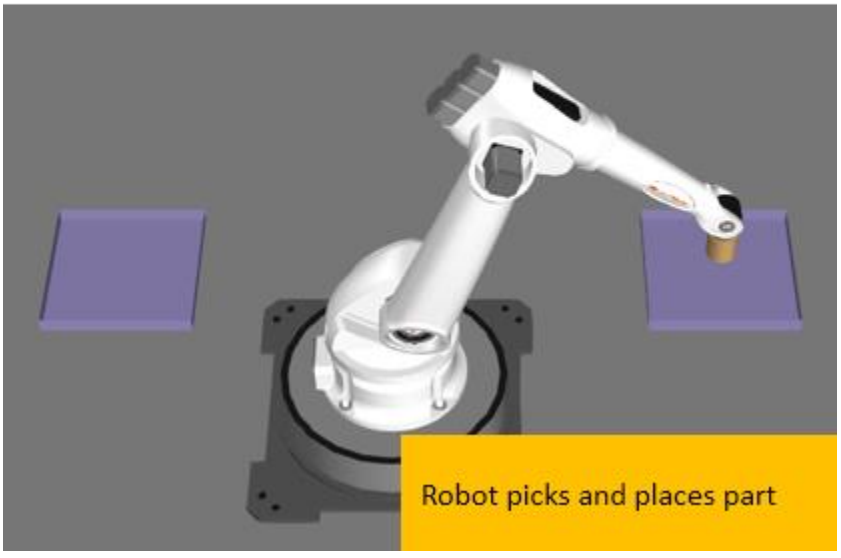
- Create
- Feed
- Need
- NeedCustomPattern
- Place
- PlacePattern
- ProcessSteps

Place

A Place task allows you to request a robot to place a component. A derivative of a Place task is a PlacePattern task, which allows you to place components in a pattern.

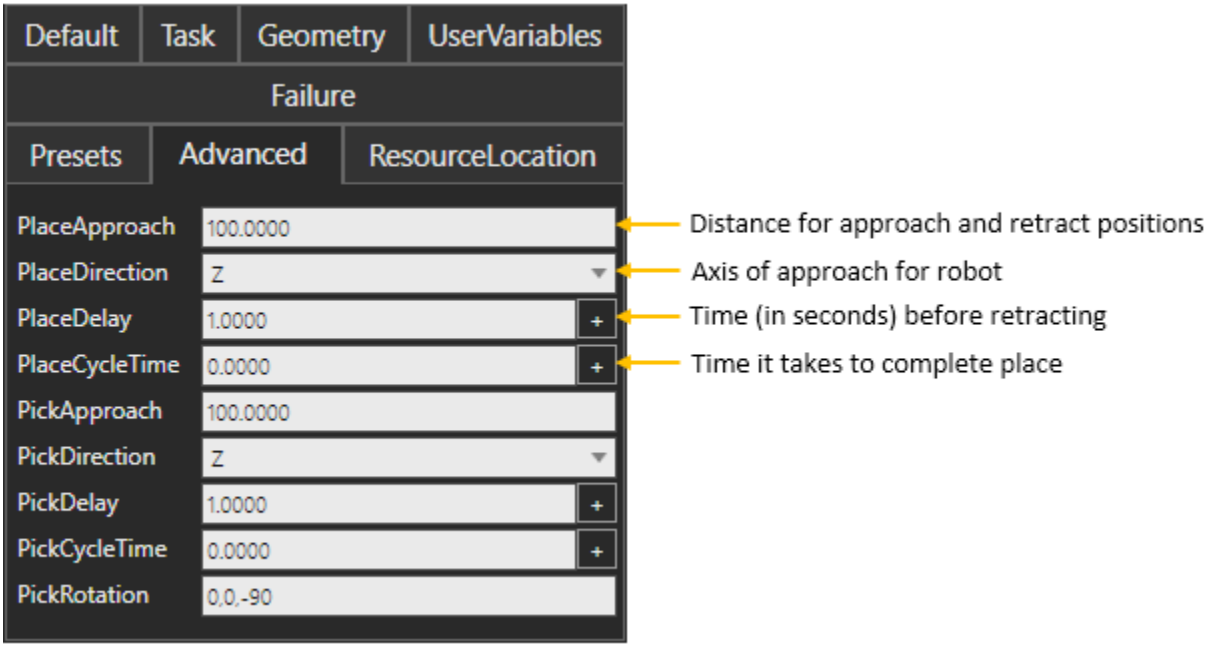


In all cases, a robot must first execute a Pick task for the component required by a Place task. This is why Pick and Place tasks must be assigned using the SerialTaskList property of a Works Robot Controller.

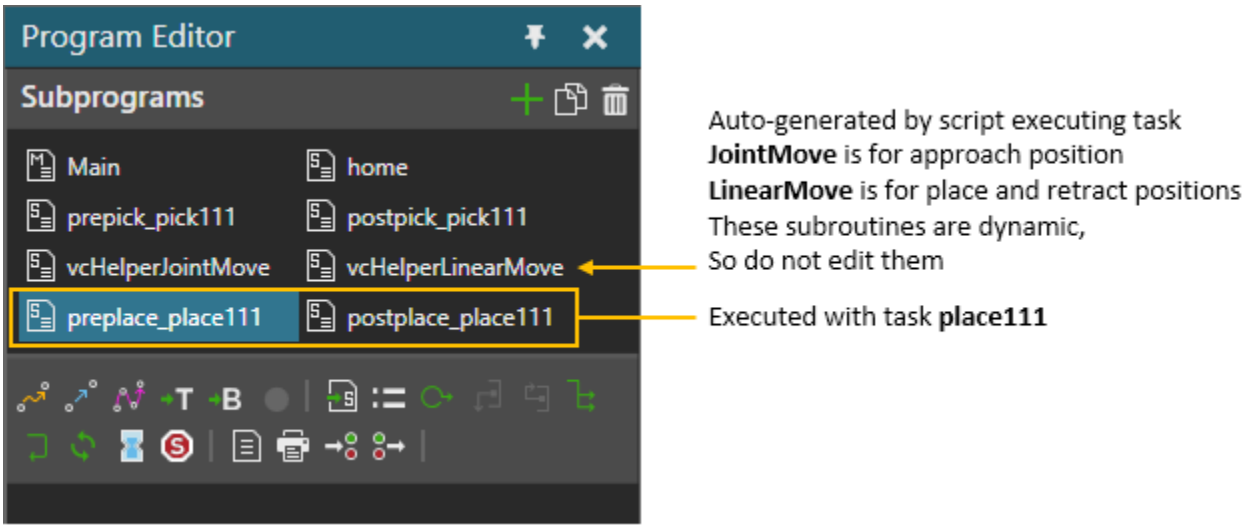


Customization

A Works Process component has a group of Advanced properties defining how resources execute place type tasks, such as Need and Place.

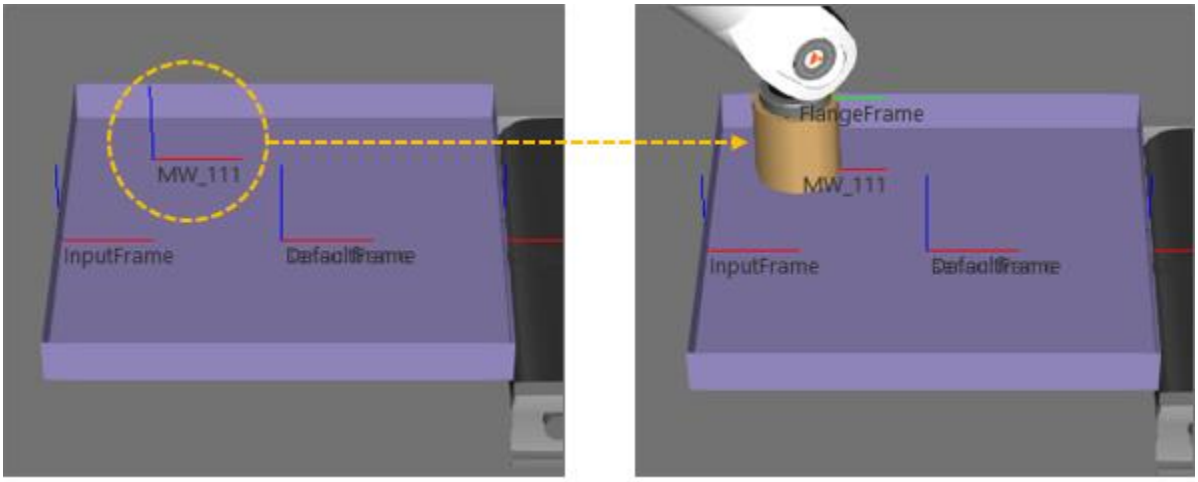


A robot executing a Need or Place task will have preplace and postplace subroutines in its program. A preplace sequence is executed before a robot places a component. A postplace sequence is executed after a robot places a component. Both sequences are executed automatically by the task itself, thereby allowing you to customize the execution of that task.



The name of a task is used to indicate its preplace and postplace sequences. If they do not exist in a robot program, they will be generated when the robot executes that task.

Note that Need and Place type tasks do respect the locations of a Works Process component. For example, you can teach a location for a component to define where it is placed by a robot.



Properties

SingleProdID identifies the component by ProdID value.

TaskName defines a name that allows you to reference the Place task and assign it to a robot.

ToolName identifies a tool component by name that should be used to execute the task. For example, you could reference the name of an end-of-arm tool for a robot.

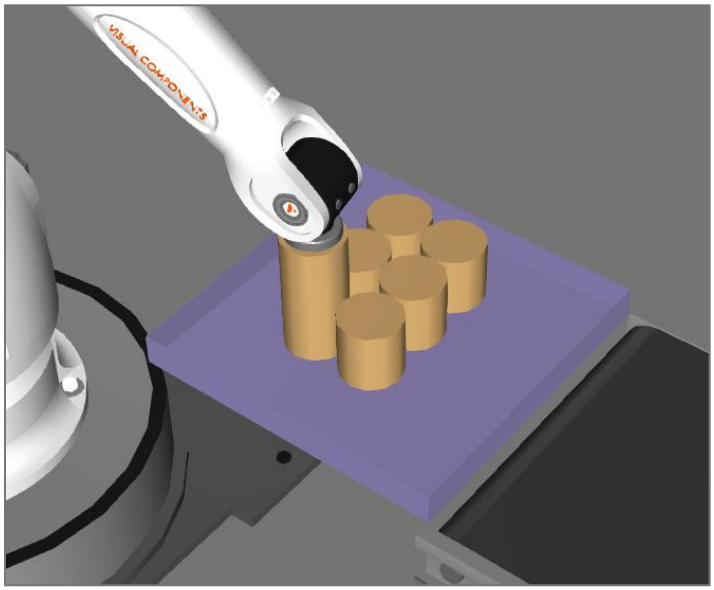
TCPName identifies a tool frame by name in the tool component used for the task.

See Also

- Create
- Feed
- Need
- NeedCustomPattern
- Pick
- PlacePattern
- ProcessSteps

PlacePattern

A PlacePattern task allows you to request a robot to place a pattern of components.



Properties

SingleProdID identifies the component you need by ProdID value.

AmountX[1)] defines the number of components to create along X-axis.

AmountY[1)] defines the number of components to create along Y-axis.

AmountZ[1)] defines the number of components to create along Z-axis.

StepX[1)] defines the spacing of components along the X-axis.

StepY[1)] defines the spacing of components along the Y-axis.

StepZ[1)] defines the spacing of components along the Z-axis.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

TaskName defines a name that allows you to reference the PlacePattern task and assign it to a robot.

ToolName identifies a tool component by name that should be used to execute the task. For example, you could reference the name of an end-of-arm tool for a robot.

TCPName identifies a tool frame by name in the tool component used for the task.

See Also

- NeedCustomPattern
- Pick
- Place

Print

A Print task allows you to print a message in the Output panel.

Properties

Text[2)] defines the content of message.

ComponentName prepends the name of the Works Process component printing the message.

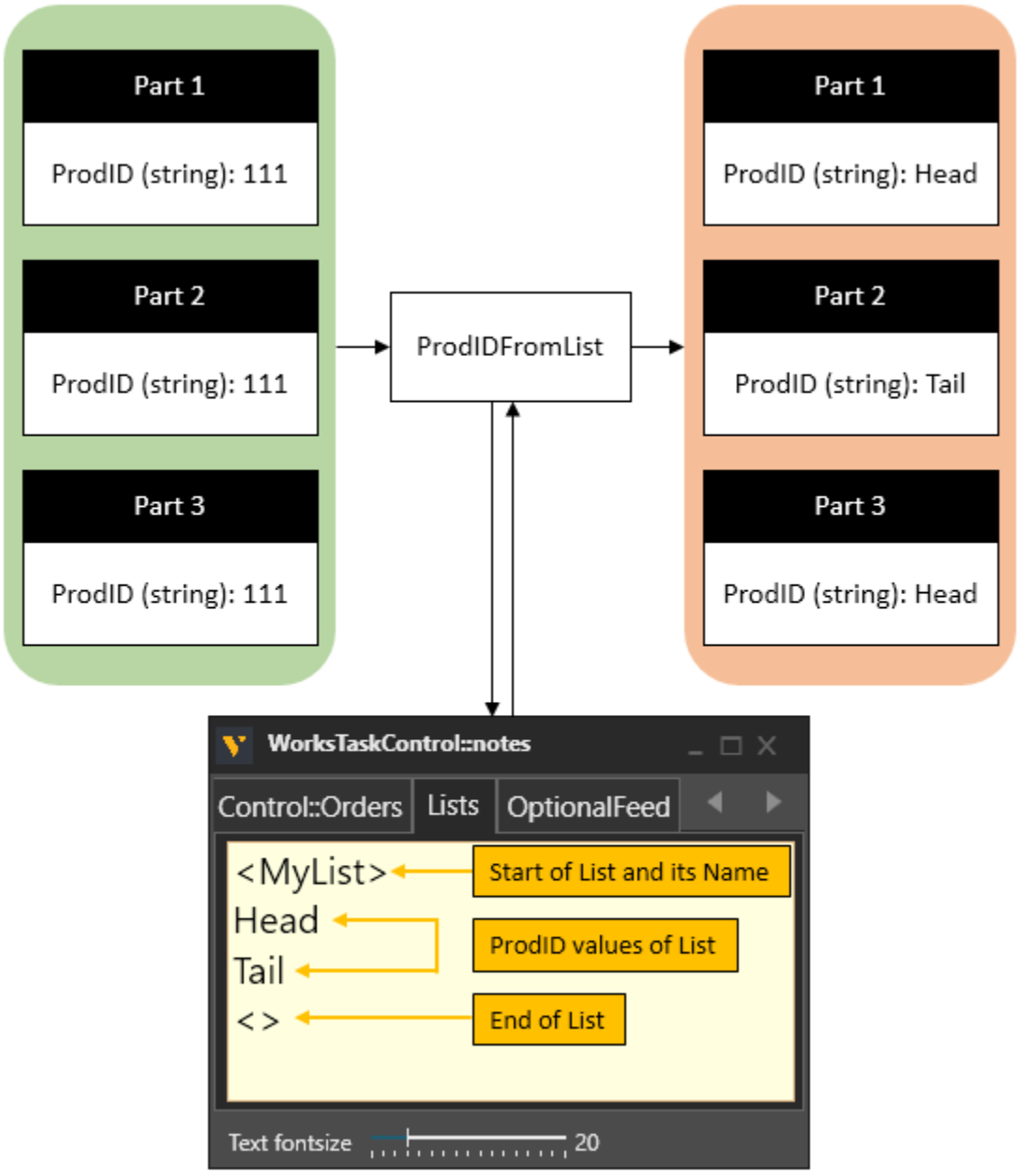
Time prepends the time the task is executed during a simulation to the message. The format is hours/minutes/seconds.

See Also

- Exit
- WarmUp

ProdIDFromList

A ProdIDFromList task allows you to change the ProdID property of a component by referring to the Lists note of a Works Task Controller.



Properties

SingleProdID allows you to filter which component is assigned a new ProdID.

Any ignores SingleProdID and changes the ProdID value of any component.

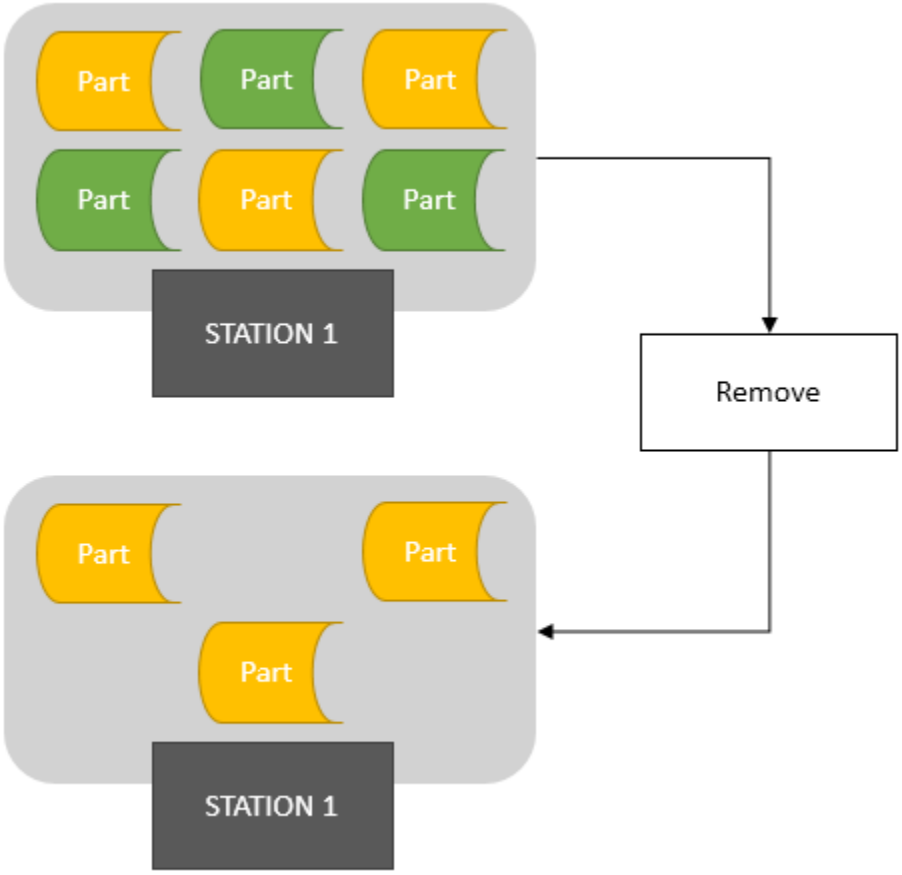
ListName identifies the list by name to refer to in the Lists note of a Works Task Controller.

See Also

- ChangeID
- GlobalID
- GlobalProcess
- NeedCustomPattern
- RestoreProdID
- StoreProdID

Remove

A Remove task allows you to delete components contained in a Works Process component. For example, you can delete components placed there by a resource.



Properties

ListOfProdID allows you to filter which components are deleted.

All ignores ListOfProdID and deletes all components.

See Also

- Delay
- DummyProcess
- Split

ReleaseResource

Once a resource is reserved by the Works Process, Releaseresource must be used to release the resource so that it can be used to carry out tasks for other processes.

Properties

ResourceName allows you to specify the resource manually by the name of the resource component.

CurrentlyReserved releases the resource that is currently reserved by the process (if any).

See Also

- ReserveResource

ReserveResource

Reserves a single resource to the Works Process until ReleaseResource is called. A resource to be reserved can be selected automatically (first arriving resource, Recommended) or named, see Properties.

Reserving overrides any active/previous reserver, release can be done only by the active reserver (Works Process). Reserved resource can't be used by any other process and has highest priority (e.g. over the Nearest) in Task Control.

Reserving is applied to all tasks requiring resource interaction that follow ReserveResource task: Feed, Need, HumanProcess, RobotProcess, Pick, Place.

Typically, the task is used to reserve a resource to a process after bringing component to it and then wait for the process to complete and then pickup the processed component. This is accomplished by calling ReserveResource before a Need and after the following Feed calling ReleaseResource.

Properties

ResourceName allows you to specify the resource manually by the name of the resource component.

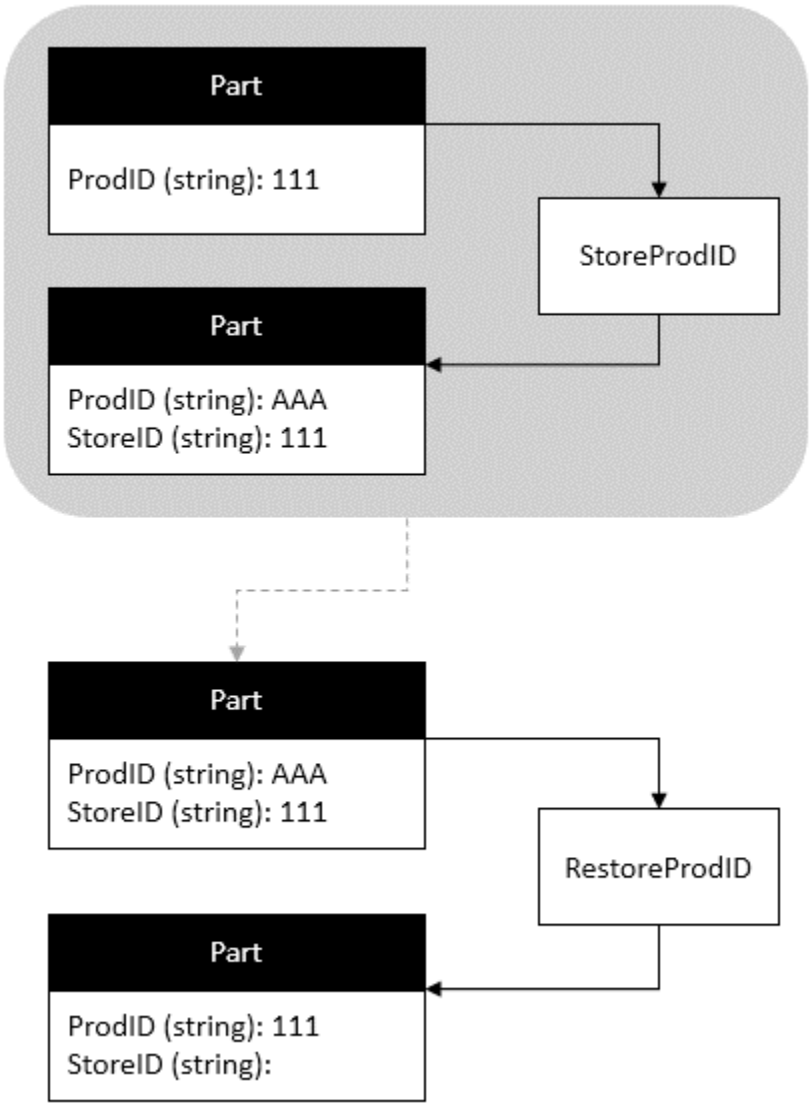
ReserveFirst the first resource arriving the process (decided by the Task Control) will be automatically reserved.

See Also

- ReleaseResource

RestoreProdID

A RestoreProdID task allows you to change the ProdID value of a component to its StoreID value. Generally, a StoreID task is used to stamp a component with a StoreID property. This records the old ProdID value and replaces it with a new one. If you restore a ProdID value, the recorded value is used and the StoreID value becomes an empty string.



Properties

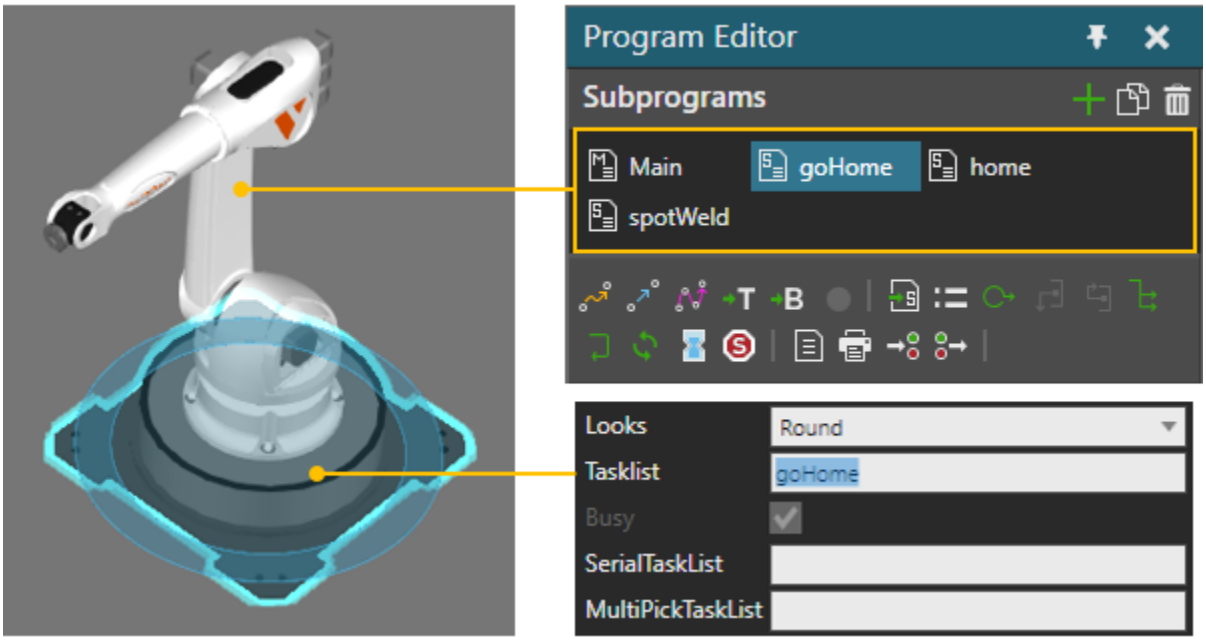
No additional properties.

See Also

- ChangeID
- StoreProdID

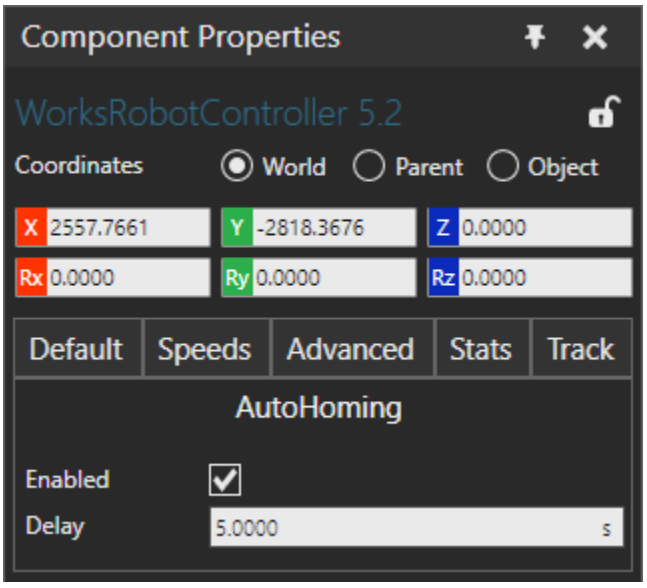
RobotProcess

A RobotProcess allows you to execute a routine in a robot program. The robot must be connected to a Works Robot Controller, and then you assign the task to the controller of that robot.

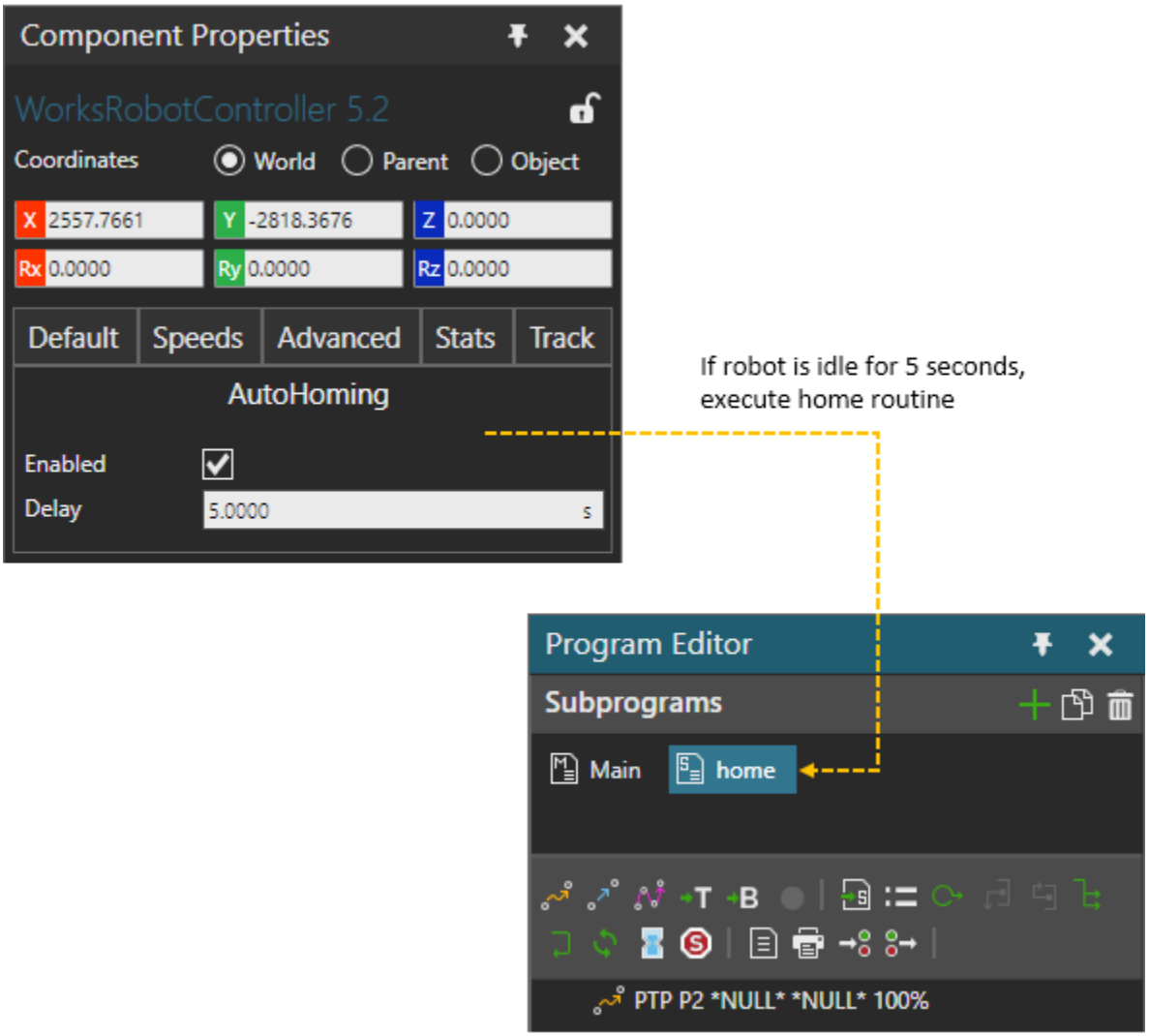


Home

A Works Robot Controller has a group of AutoHoming properties for managing the idleness of a robot.



By default, a robot resource will have a home routine in its program. The home routine is automatically called if AutoHoming is enabled and the robot is idle for an amount of time.



Tip: If there is not a home routine in a robot program, make sure the robot is connected to a Works Robot Controller and run the simulation at least once.

Properties

TaskName defines a name that allows you to reference the RobotProcess task and assign it to a robot. You must use the name of the routine you want to call in the robot's program.

ToolName identifies a tool component by name that should be used to execute the task. For example, you could reference the name of an end-of-arm tool for a robot.

TCPName identifies a tool frame by name in the tool component used for the task.

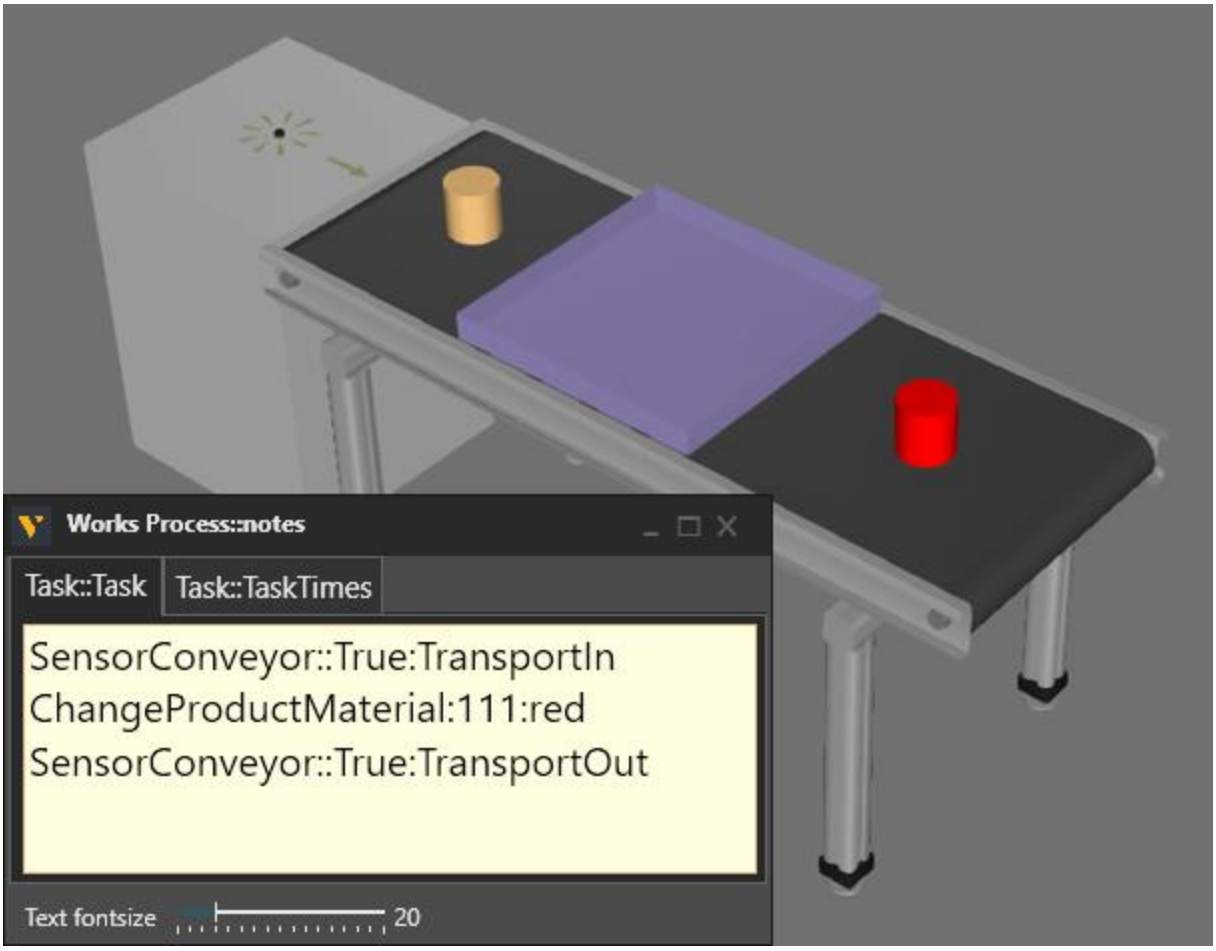
See Also

- GlobalProcess
- HumanProcess
- MachineProcess
- Pick
- Place

SensorConveyor

A SensorConveyor task allows you to use a Works Process component as an external path sensor. This means a Works Process component must be connected to a path, for example the path of a conveyor. From there, you could use a SensorConveyor task to manipulate the conveyor and execute an inline process.

Important: The path sensor is designed to detect the origin of a component.



Basic Workflow

1. Select a **Works Process** component, and then set its **HeightOffset** property to zero.
2. Use the PnP command to connect the Works Process component to the path of a conveyor.
3. To change the distance of the Works Process component on the path, do any of the following:
    a. Use the PnP command to drag the Works Process component along the path.
    b. Edit the **PathSensor::Distance** property.
4. Create one or more SensorConveyor tasks and other types of tasks as needed.

Properties

ListOfProdID allows you to filter which components are detected by the path sensor.

Any ignores ListOfProdID and detects any component.

SensorConveyor defines a task to execute in association with the connected conveyor.

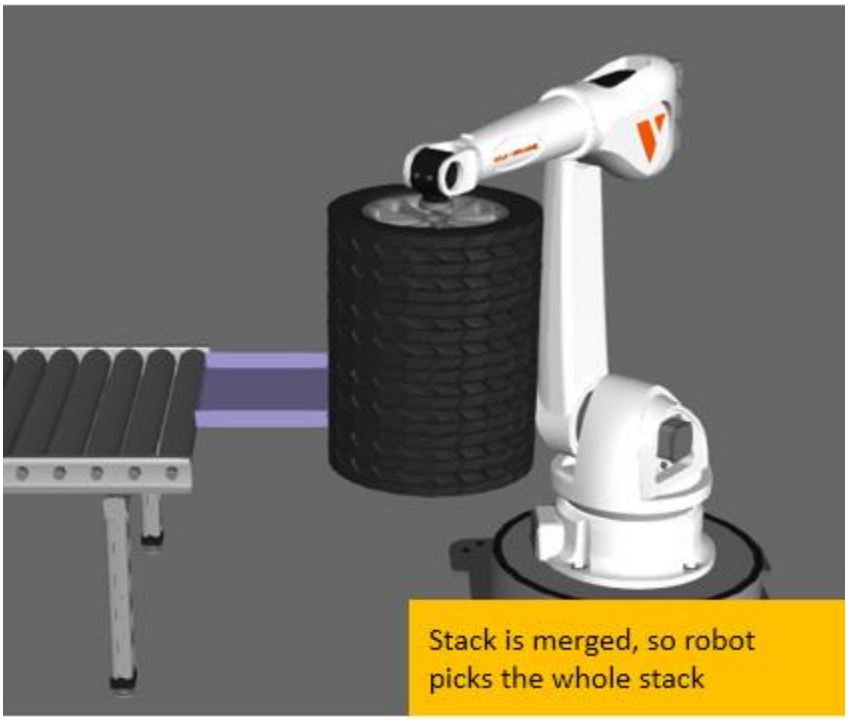
- *TransportIn* moves the detected component from the conveyor to the Works Process component. This will stop the detected component and transfer it into the container of the Works Process component. Generally, this is done to allow other tasks in a process to affect the detected component.
- *TransportOut* moves the detected component from the Works Process component to the conveyor.
- *Start* turns on the path of the conveyor.
- *Stop* turns off the path of the conveyor.
- *CapacityAvailableTrue* allows the conveyor to accept components on its path. A capacity test for the conveyor will return a true value.
- *CapacityAvailableFalse* denies the conveyor from accepting components on its path. A capacity test for the conveyor will return a false value. Be careful with this task because the effect is not canceled when you reset a simulation.
- *ChangePathDirectionForward* makes the path of the conveyor flow forward.
- *ChangePathDirectionBackward* makes the path of the conveyor flow backward.

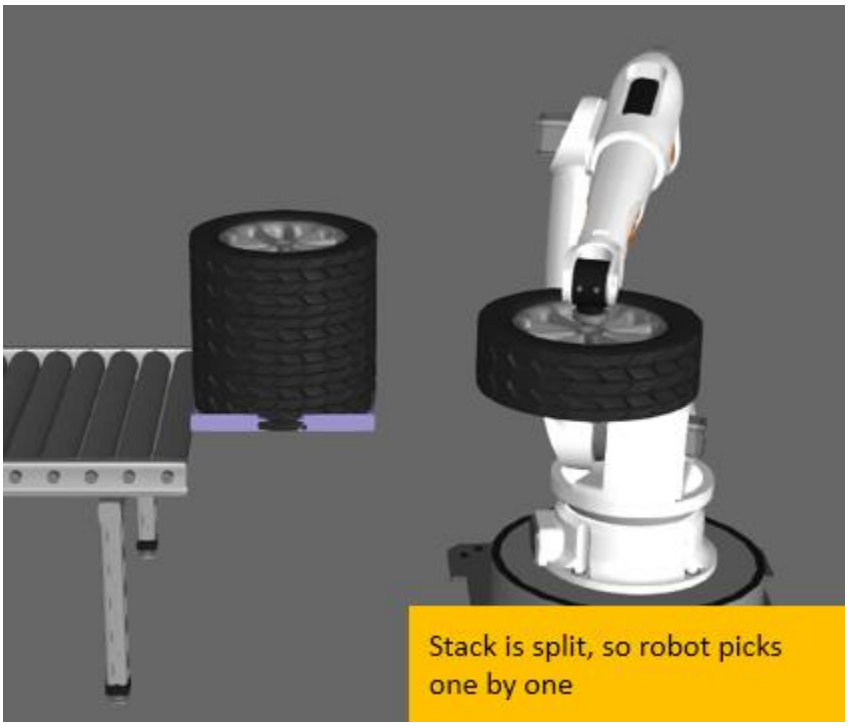
See Also

- ChangePathDirection
- TransportIn
- TransportOut
- WriteProperty

Split

A Split task allows you to detach components contained in a Works Process component from one another. Generally, this is done for a bundle of merged components that need to be picked separately by resources.





Properties

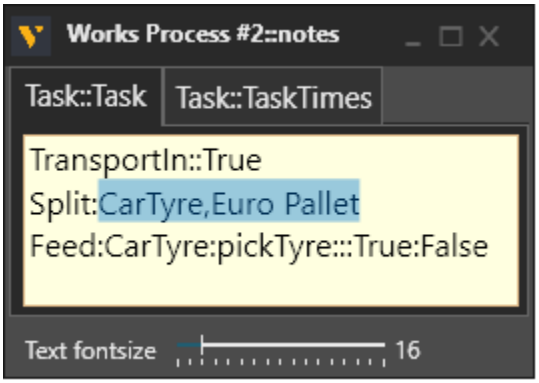
ListOfProdID identifies the components by ProdID value that will be detached from one another, and then attached to the Works Process component.

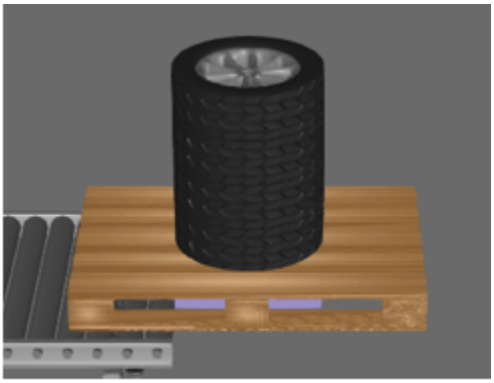
Troubleshooting

You might encounter the following issues after splitting up components and then requesting a resource pick them up.

Issue 1. Nothing is Picked Up

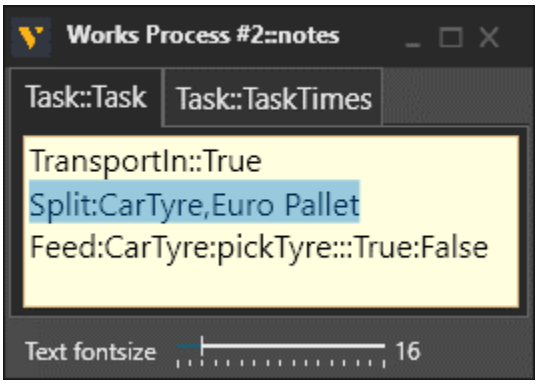
This might happen when you do not split all components in a bundle. For example, you split parts on a pallet but not the parts and the pallet. A quick solution is to list every ProdID value in the bundle, including the ProdID of the parent component, in the Split task.





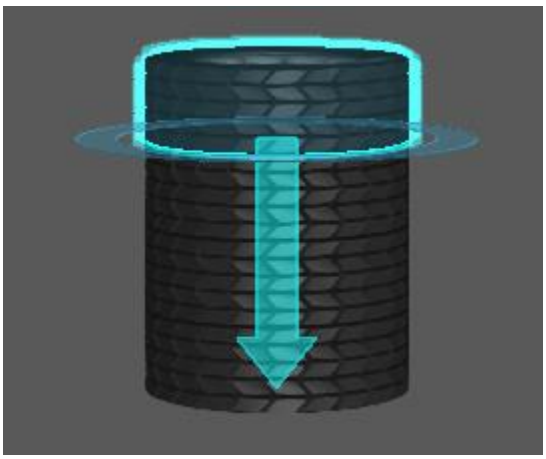
Issue 2. Entire Stack is Picked Up

This happens when components in a bundle are not detached from one another. The solution is to insert a Split task before components in the bundle are picked by a resource.

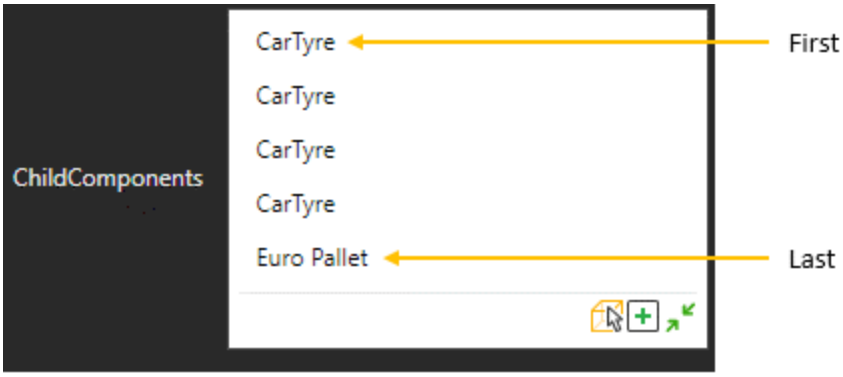


Issue 3. Resource Picks in Wrong Order

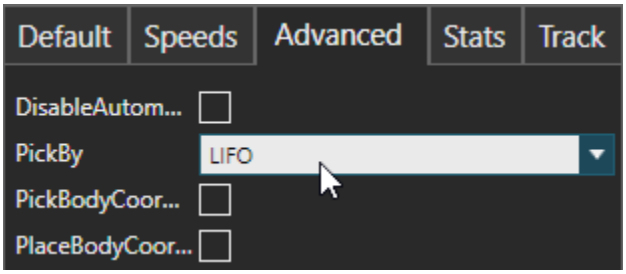
One factor is the parent-child hierarchy of a bundle. That is, how components in a bundle were attached to one another when you merged them.



Another factor is the list of child components in a Works Process component. That is, the order in which components were attached to a Works Process component when you split them.



Another factor is the pick order of a resource. The default pick order for a human is LIFO. A Works Robot Controller has an Advanced::PickBy property that allows you to use LIFO or FIFO.

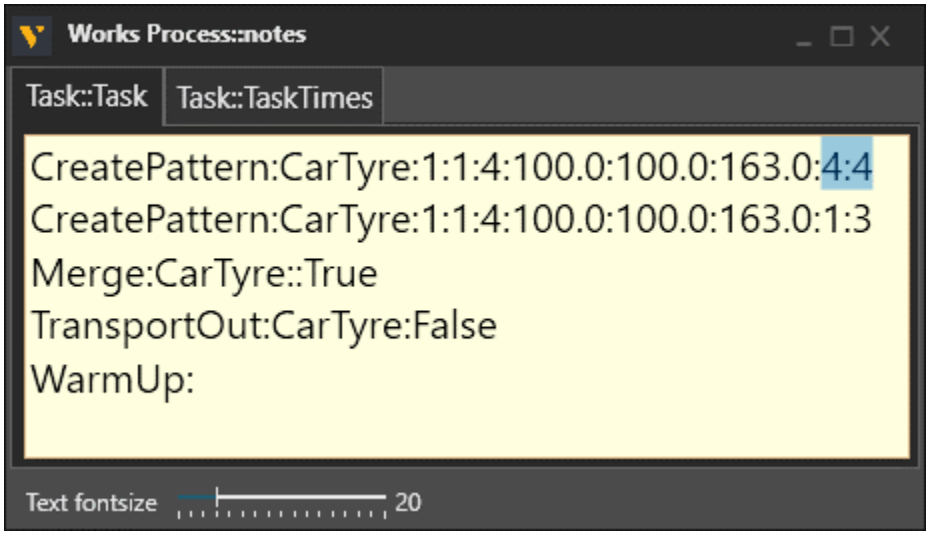


If a robot is picking in the wrong order, for example bottom to top not top to bottom:

- Set its Advanced::PickBy property to LIFO/FIFO and test.

If a resource is picking components out of order, for example one from top, one from bottom, and then one from middle:

- This might be a combination of two or more factors. One workaround is to create a duplicate CreatePattern task. The duplicate should come first and create only the last component in the pattern. The original should come second and create the rest of the components in the pattern.



Why would this work? The last component in the pattern would be added first and become the parent component if the pattern of components were merged together. When you split them up, the parent component is the last detached component. By default, the last component is picked first, for example top of stack.

See Also

- Merge
- ChangeID
- CreatePattern
- Feed
- TransportOut

StoreProdID

A StoreProdID task allows you to change the ProdID value of a component and keep the old value in a StoreID property.

Properties

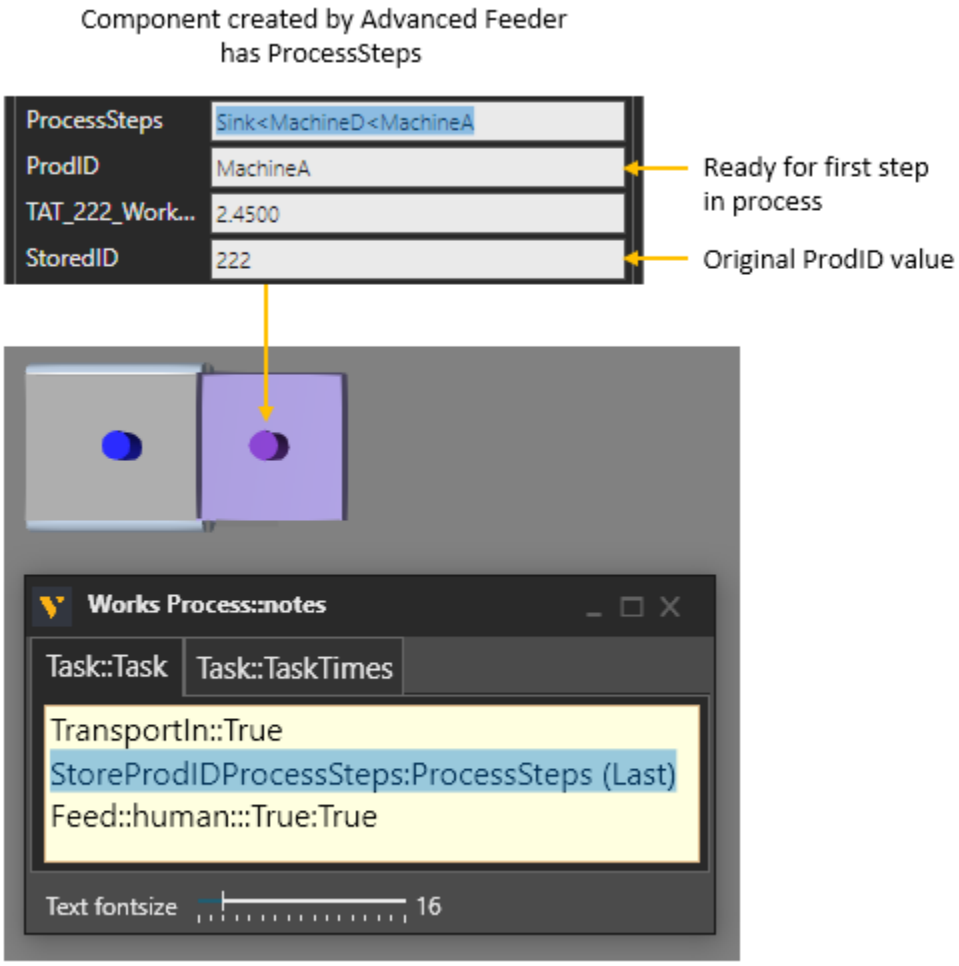
NewProdID[2)] defines the new ProdID value for all components contained in the Works Process component.

See Also

- RestoreProdID
- ChangeID

StoreProdID_NewFromProcessSteps

A StoreProdID_NewFromProcessSteps task allows you to change the ProdID value of a component and keep the old value in a StoreID property. The new value is taken from the ProcessSteps property of a component. Generally, this is done to mark the component as ready to begin the steps in its process. At the end of the process, you could use a RestoreProdID task to retrieve the old value.



Properties

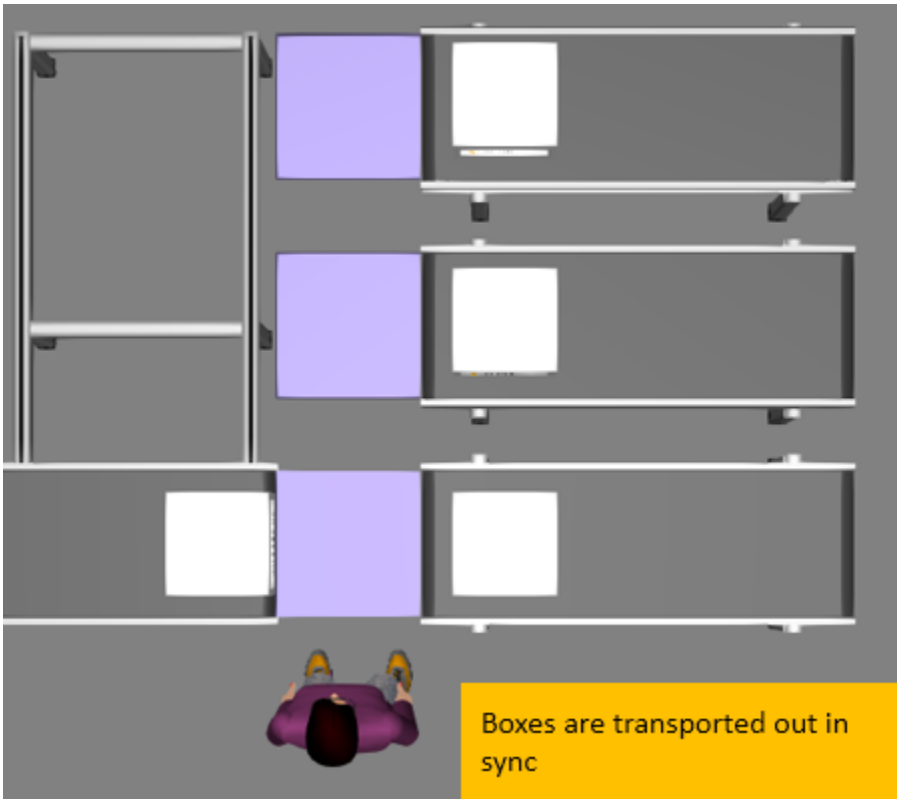
NewIDFrom[2)] selects the first or last item in the ProcessSteps property of a component. This would depend on the order in which you are executing steps.

See Also

- ProcessSteps
- ChangeIDFromProcessSteps
- StoreProdID

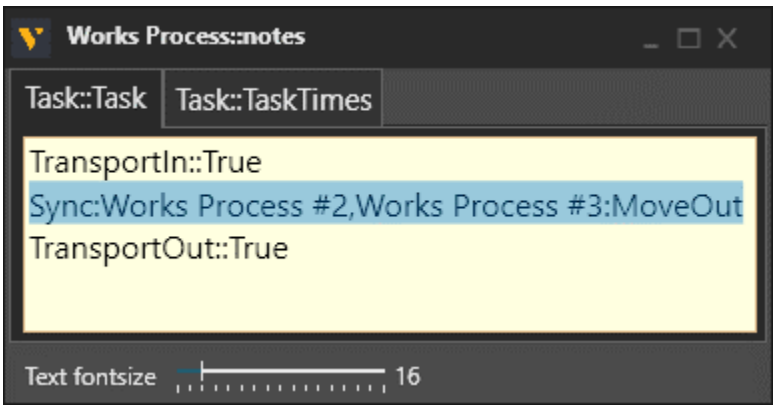
Sync

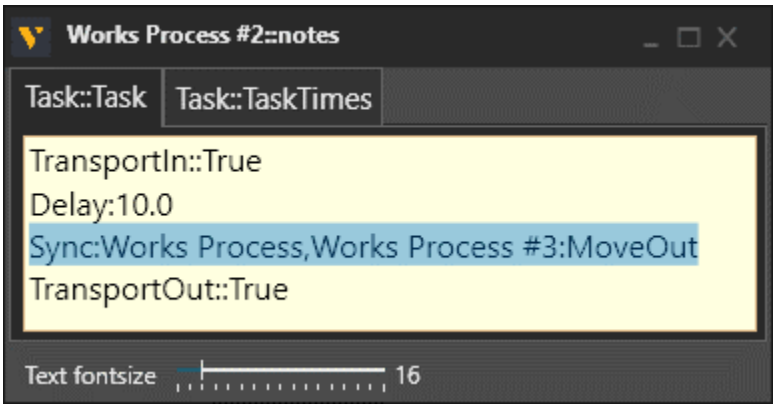
A Sync task allows you to synchronize the execution of tasks in a Works Process component with other Works Process components. For example, you can make Works Process components wait for one another before executing a critical task.

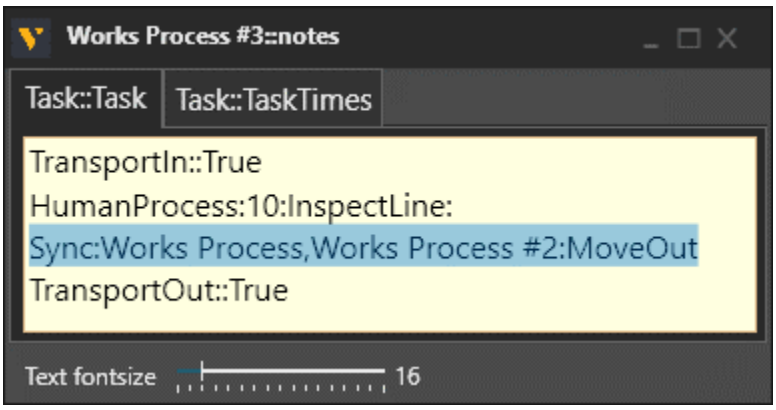


Logic

A sync can involve multiple Works Process components. Each component must execute its own Sync task and refer to the other components. The message of the sync must match for each component.







Sync tasks can be executed out of order. The message of a Sync task is sent to a Works Task Controller, which listens for each message. Once all the messages are received from the required components, the Works Task Controller signals each of the components using their SyncOK signal. This marks the completion of that Sync task.

Properties

ListOfCompNames identifies the other Works Process components by name. These components must execute their own Sync task and refer to one another.

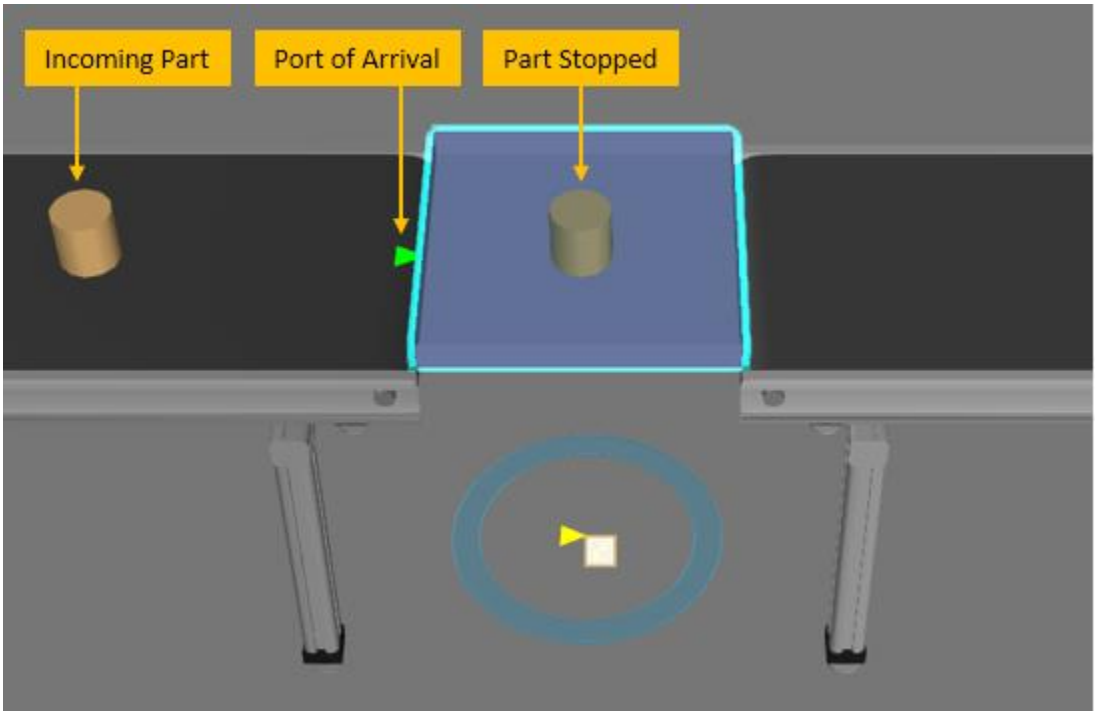
SyncMessage defines the message that must be passed by each Works Process component to complete the task.

See Also

- WaitProperty
- WaitSignal
- WaitForOrder
- WriteProperty
- WriteSignal

TransportIn

A TransportIn task allows you to move one or more types of components into a Works Process component and stop them there. Generally, this is done to allow other tasks in the process to affect the components.



By default, a Works Process component acts like a conveyor. It will flow components in and out of its path without stopping them. In order to stop them, components must arrive from an interface connected at the beginning of the path. The components will then stop at the default frame of a Works Process component or a taught location. From there, the components are contained in the Works Process component.

Properties

ListOfProdID allows you to filter which components are stopped by the task. Note that the task does not wait for a set of components. The task is completed when a component with one of the listed ProdID values is transported into the Works Process component.

```
#transports in 111 or 222, but only changes material for 111
TransportIn:111,222:False
ChangeProductMaterial:111:red
TransportOut::True
```

You can use a question mark (?) as a delimiter to create conditional tags for each ProdID value. That allows you to create conditional sets of tasks for different components.

```
#transports in 111 or 222, but only changes material for 111
TransportIn:111?222:False
<111>
ChangeProductMaterial:111:red
<>
<222>
<>
TransportOut::True
```

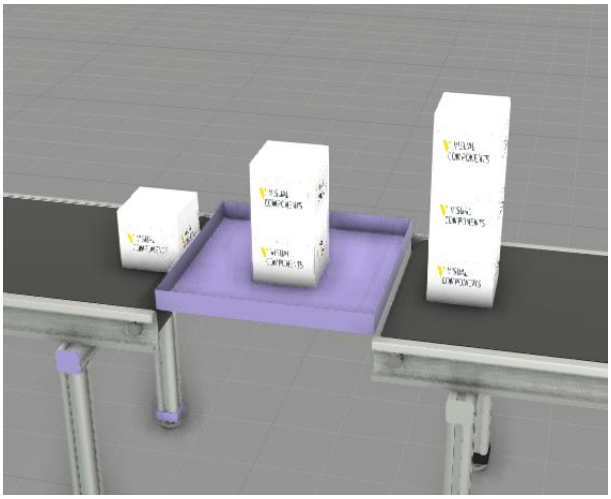
Any ignores ListOfProdID and stops any component.

See Also

- ChangePathDirection
- SensorConveyor
- TransportOut

TransportInPattern

TransportInPattern allows you to transport components in to the process via connected component e.g. a conveyor for stacking to a pattern, similarly as in NeedPattern.



```
# Stacks boxes (111) to pillar of 3 and bundles that to one with Merge
TransportInPattern:111:1:1:3:150.0:150.0:150.0:1:999999
Merge:111::True
TransportOut::True
```

Properties

SingleProdID identifies the component you need by ProdID value.

AmountX[1)] defines the number of components to create along X-axis.

AmountY[1)] defines the number of components to create along Y-axis.

AmountZ[1)] defines the number of components to create along Z-axis.

StepX[1)] defines the spacing of components along the X-axis.

StepY[1)] defines the spacing of components along the Y-axis.

StepZ[1)] defines the spacing of components along the Z-axis.

StartRange[1)] identifies the beginning element of pattern. By default, the pattern begins with its first element.

EndRange[1)] identifies the final element of pattern. By default, all elements in the pattern are included by passing a value greater than the number of elements in the pattern, e.g. 999999.

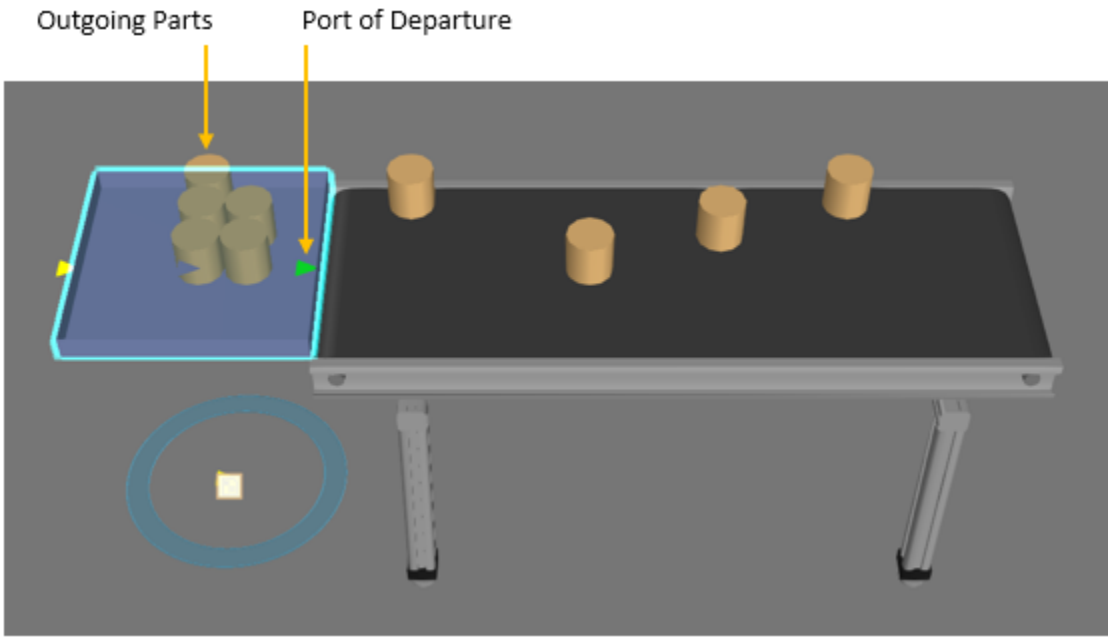
Tip: Generally, a range is modified for testing the position of components in the pattern.

See Also

- NeedPattern

TransportOut

A TransportOut task allows you to move components out of a Works Process component. The components are moved along the path of a Works Process component. In all cases, there must a connected interface at the end of that path. Otherwise, components will not depart the Works Process component.



A TransportOut task is completed when a Works Process component no longer contains components that should be moved out of its container.

Properties

ListOfProdID allows you to filter which components are moved out by the task.

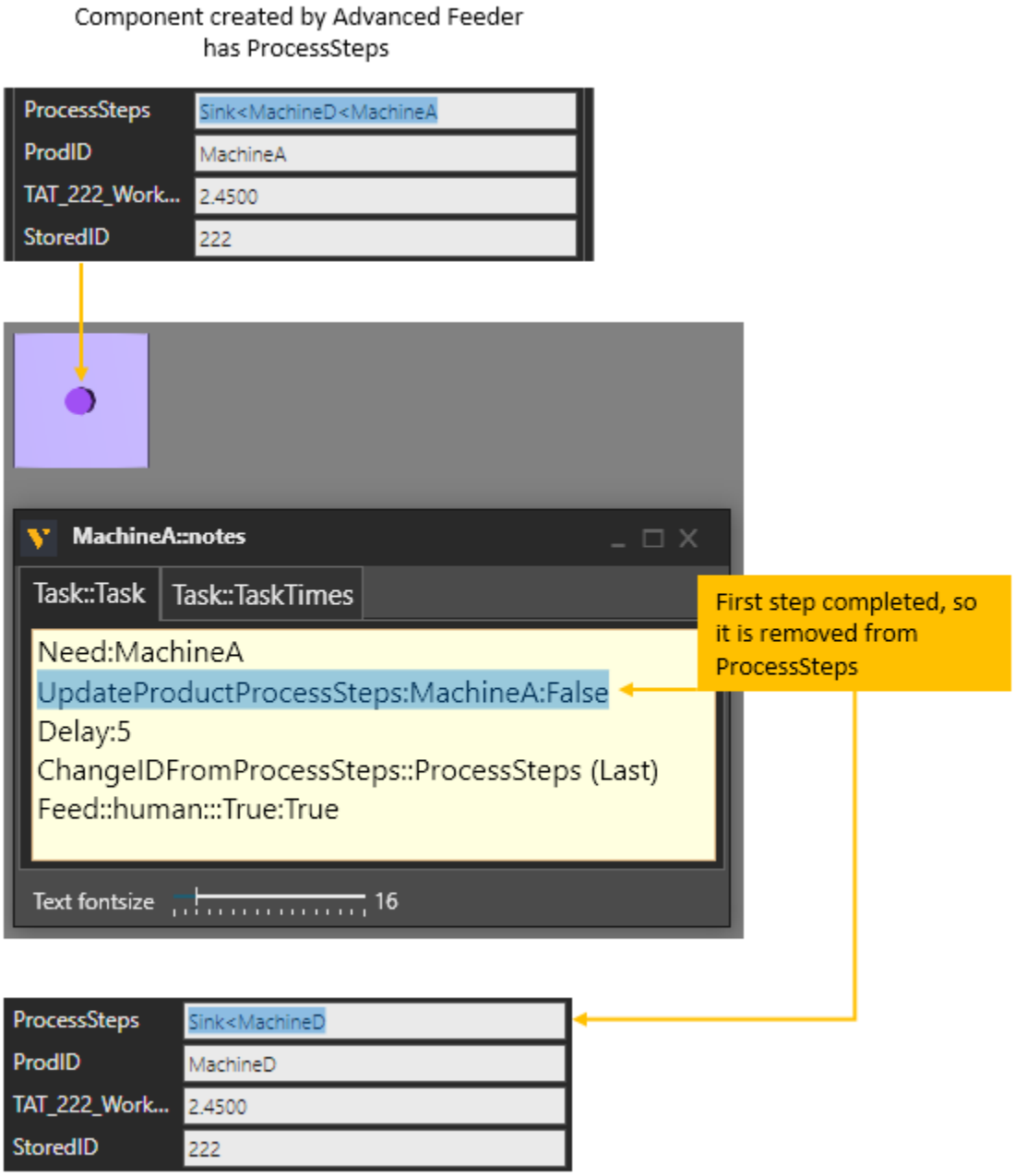
All ignores ListOfProdID and moves out every component.

See Also

- ChangePathDirection
- SensorConveyor
- TransportOut

UpdateProductProcessSteps

An UpdateProductProcessSteps task allows you to update the ProcessSteps property of a component. Generally, this is done to indicate when a component completes a step in its process. That is, you are removing one or more steps from the process of the component.



Properties

ListOfProcessStepsPerformed allows you to filter which steps to remove from the ProcessSteps property of components contained in the Works Process component.

AllProcessSteps ignores ListOfProcessStepsPerformed and removes all steps.

See Also

- ProcessSteps
- ChangeIDFromProcessSteps
- StoreProdID_NewFromProcessSteps

WaitForProductPick

WaitProductPick waits until a product is picked from the works process. As with WaitForProductPlace this task doesn't define that how the product is picked/placed and by whom. For example, a robot program could be used to pick & place the products with grasp and release action signals.

Properties

No additional properties.

See Also

- WaitForProductPlace

WaitForProductPlace

WaitProductPick waits until a product is placed to the works process. As with WaitForProductPick this task doesn't define that how the product is picked/placed and by whom. For example, a robot program could be used to pick & place the products with grasp and release action signals.

Properties

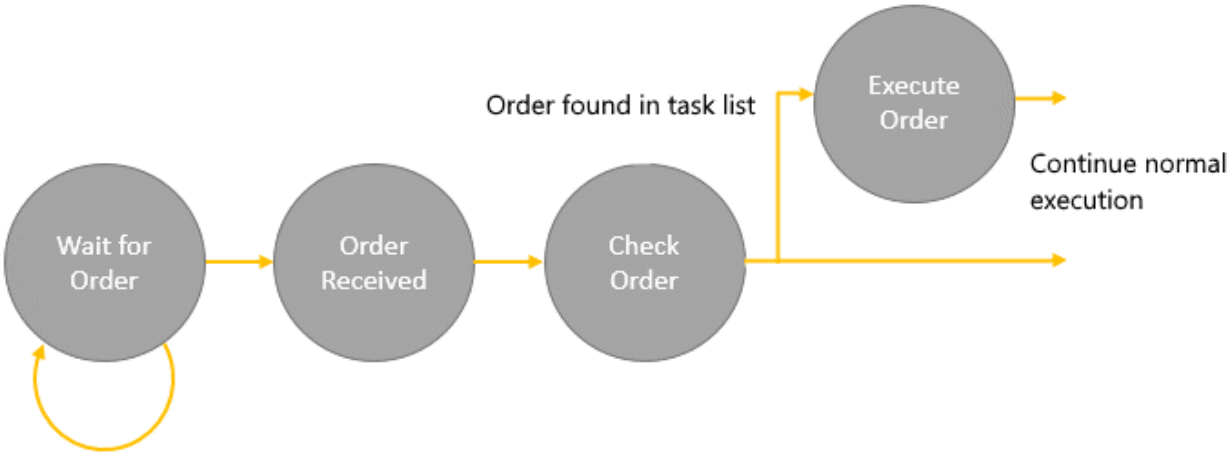
No additional properties.

See Also

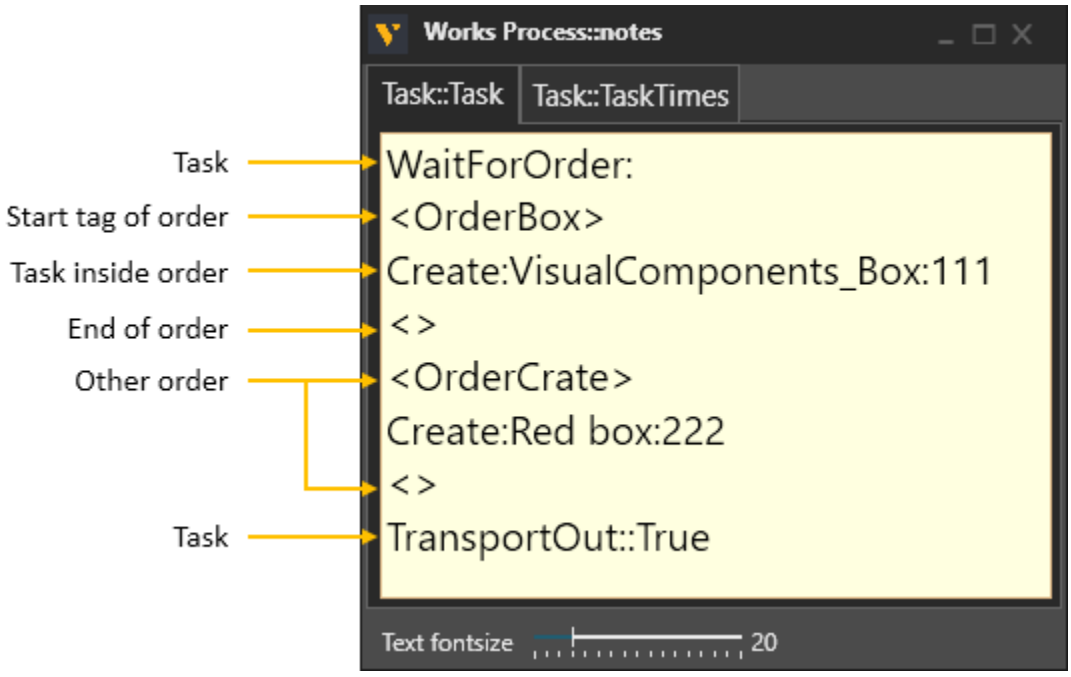
- WaitForProductPick

WaitForOrder

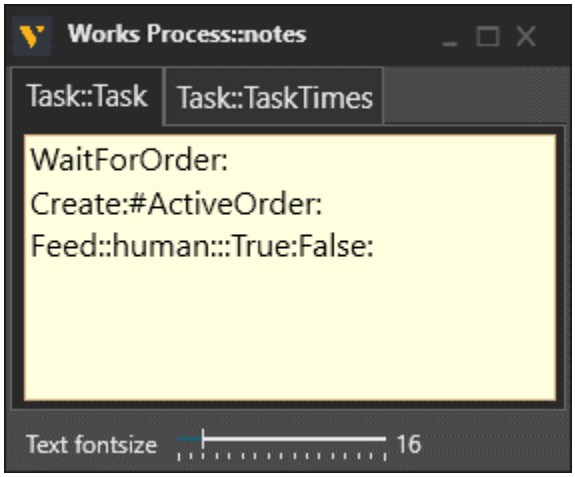
A WaitForOrder task allows you to wait for an order from a Works Process component or handle one if there's already order in a queue, and then execute a conditional set of tasks or just proceed executing following tasks normally ignoring name of the order. The currently handled (active) order is stored to the Works Process as a property value "ActiveOrder". Generally, the order is placed by a Works Process component downstream in a production line to a Works Process component that is upstream.



When a WaitForOrder task is created, a series of conditional tags can be generated in the task list to predefine tasks for certain orders. The start tag defines the order name, and the end tag marks the end of the order. Tasks inserted inside these tags are executed by the order. Access the Task note of a Works Process component to edit order tags and add new ones.



Alternatively, the conditional tags can be left out and the active order name can be read from the process property “ActiveOrder”. This way a product can be created and fed based directly on the order value which can be the needed ProdID for example:



All received orders are placed on a queue in the process where those are consumed by first come first serve or FIFO principle.

Properties

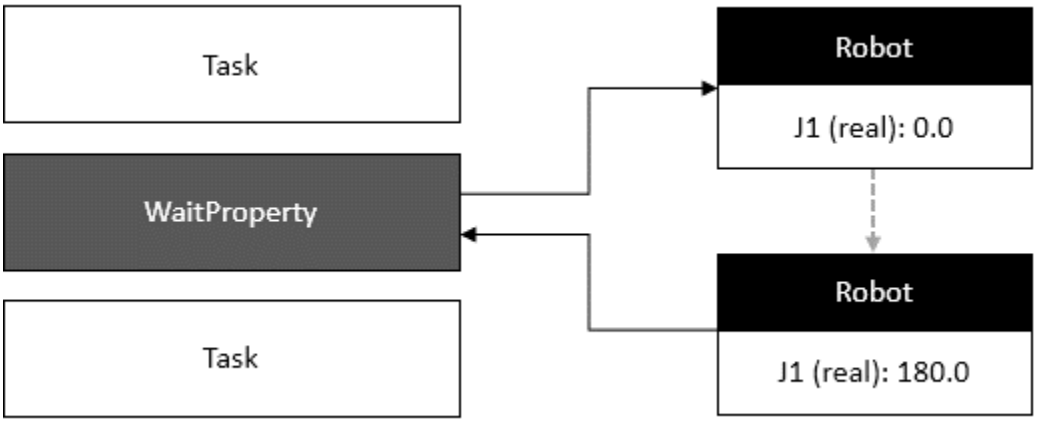
No additional properties.

See Also

- Order
- WaitProperty
- WaitSignal

WaitProperty

A WaitProperty task allows you to wait for a property value in a static component. Property type must be either Boolean, integer, real, or string. The value is checked by polling it every 10 ms.



Properties

SingleCompName identifies the component by name.

PropertyName identifies the component property by name. If not a default property, refer to the property using this syntax `<tab name>;;<property name>`.

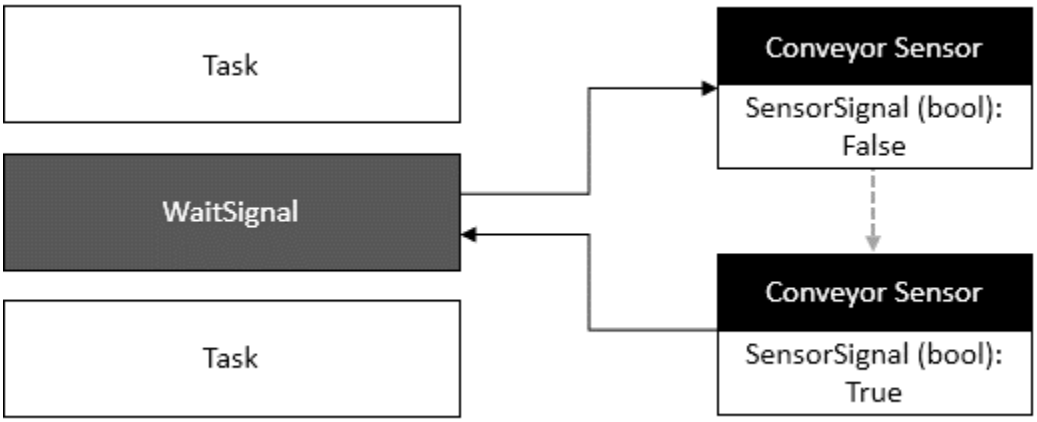
PropertyValue defines the property value that is used for checking and to proceed.

See Also

- WaitForOrder
- WaitSignal
- WriteProperty
- WriteSignal

WaitSignal

A WaitSignal task allows you to wait for a signal value in a static component. Signal type must be either Boolean, integer, real, or string. You cannot use a WaitSignal task with IO ports of a robot but can refer to signals connected to those ports.




```
# evaluates the "SensorBooleanSignal" value and
# executes conditional tasks based on the evaluated (current) value
WaitSignal:Sensor:SensorBooleanSignal:True?False:False
<True>
# do tasks...
<>
<False>
# do something else...
<>
```

Properties

SingleCompName identifies the component by name.

SignalName selects the signal. You must first set SingleCompName to populate this list.

SignalValue defines the value of the selected signal. Supports the use of "?" conditional syntax.

WaitTrigger defines if the signal is used as a trigger instead of testing the value immediately i.e. if True, the task is tested and triggered only on the "rising edge" of the signal (when its value changes)

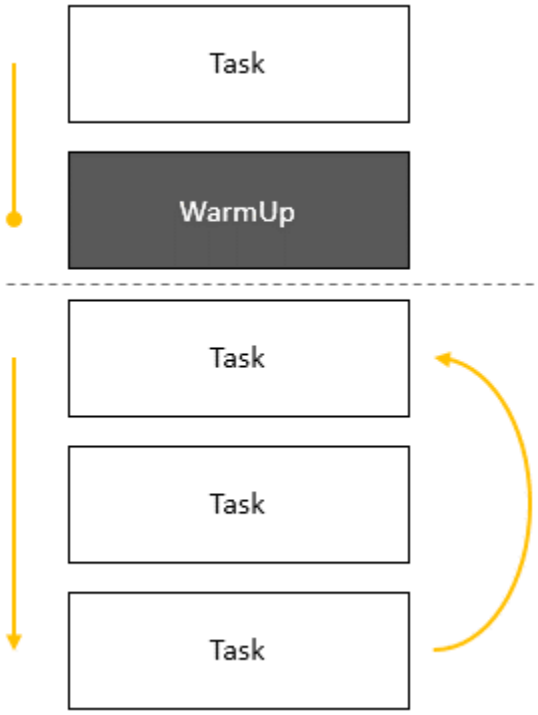
See Also

- WaitForOrder
- WaitProperty
- WriteProperty

- WriteSignal

WarmUp

A WarmUp task allows you to execute preceding tasks only once during a simulation.



Properties

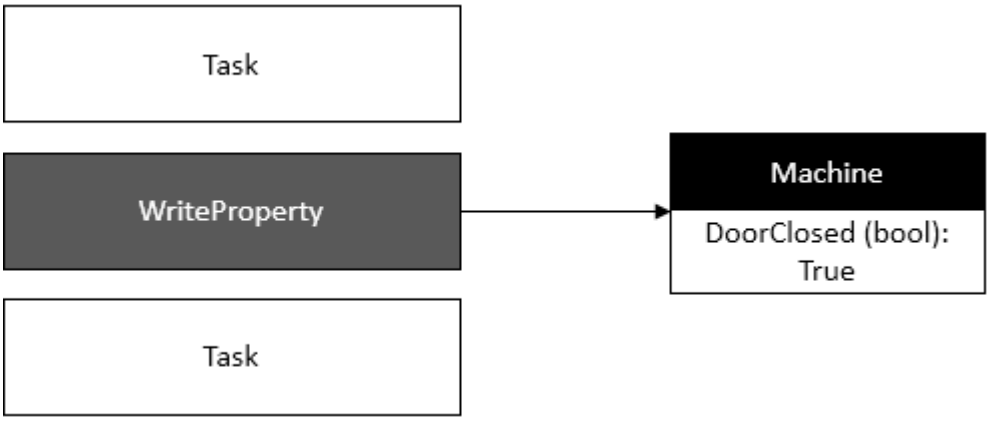
No additional properties.

See Also

- Exit
- Loop

WriteProperty

A WriteProperty task allows you to set the value of a property in a static component. Property type must be either Boolean, integer, real, or string.



Properties

SingleCompName identifies the component by name.

PropertyName identifies the component property by name. If not a default property, refer to the property using this syntax `<tab name>;;<property name>`.

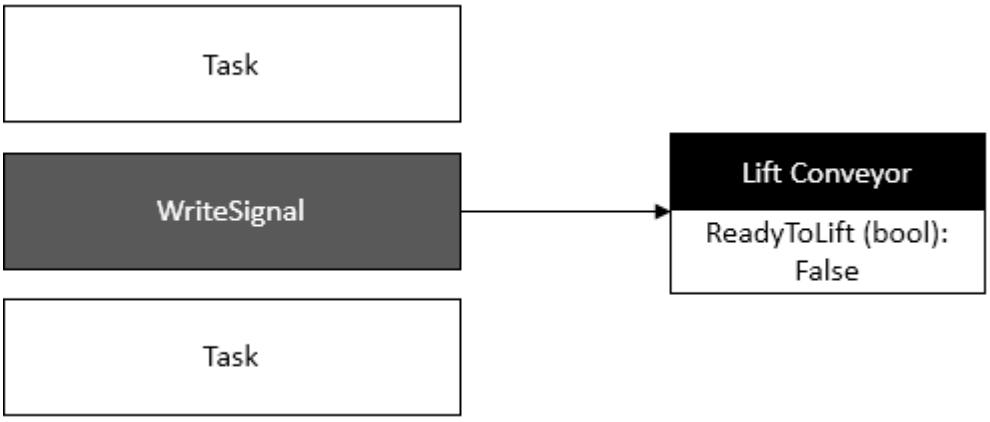
PropertyValue defines the property value.

See Also

- WaitForOrder
- WaitSignal
- WriteProperty
- WriteSignal

WriteSignal

A WriteSignal task allows you to set the value of a signal in a static component. Signal type must be either Boolean, integer, real, or string. You cannot use a WriteSignal task with IO ports of a robot but can refer to signals connected to those ports.



Properties

SingleCompName identifies the component by name.

SignalName selects the signal. You must first set SingleCompName to populate this list.

SignalValue defines the value of the selected signal.

See Also

- WaitForOrder
- WaitSignal
- WriteProperty
- RobotProcess
- Sync