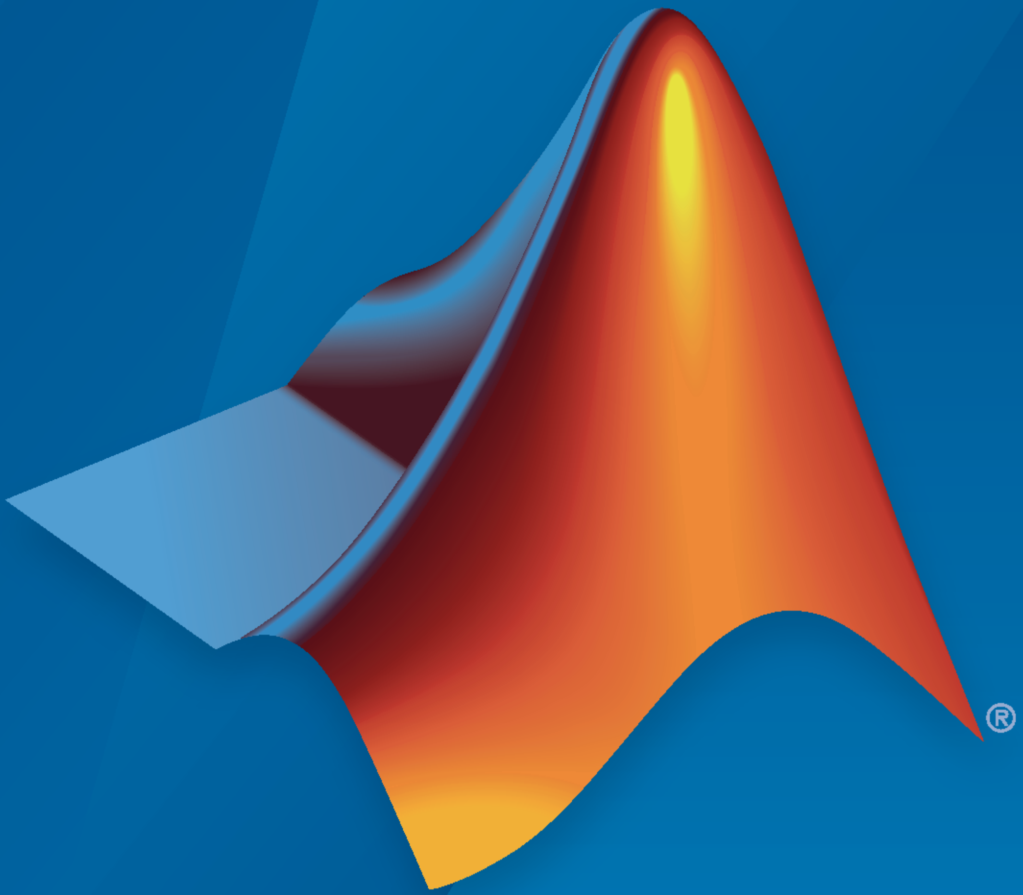
Simulink® Check™

CI/CD Automation Support Package



MATLAB®&SIMULINK®

R2022**a** and **R**2022**b**

MathWorks®

CI/CD Automation for Simulink® Check™ Support Package

Revision History

August 2022	PDF Only	Version 1.0
September 2022	PDF Only	Version 1.1
October 2022	PDF Only	Version 1.2
November 2022	PDF Only	Version 2.0
December 2022	PDF Only	Version 2.1

Contents

Get Started

The support package CI/CD Automation for Simulink® Check™ provides tools to help you integrate your model-based process into a Continuous Integration / Continuous Delivery (CI/CD) system.

The support package provides:

- A customizable process modeling system that you can use to define your build and verification process
- A build system that can efficiently execute a pipeline in your CI system
- The **Process Advisor** app for deploying and automating your prequalification process
- Integration with common CI systems, including a pipeline generator to automatically create a child pipeline configuration file for CI

You can use the support package to help you set up a model-based design (MBD) pipeline, reduce build time, reduce build failures, debug build failures, and deploy a consistent build and verification process.

For an overview of these features, see the chapter "Fundamentals".

Where to Get Started

If you are a:

- Model developer or test engineer, you may want to start with "Run Tasks Using Process Advisor".
- Process engineer, you may want to start with "Author Your Process Model" and "Run Builds".
- DevOps engineer, you may want to start with "Integrate into CI".

Note The support package only supports the MATLAB® versions:

- R2022b Update 1 and later updates
- R2022a Update 4 and later updates

Fundamentals

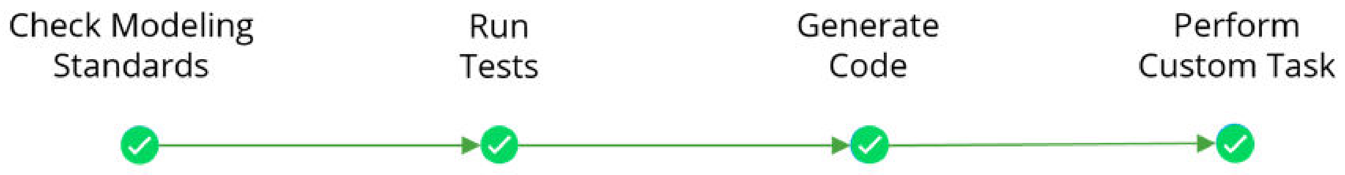
The following sections provides an overview of the:

- MBD Pipeline
- Build System
- **Process Advisor** app
- CI/CD System Integration

MBD Pipeline

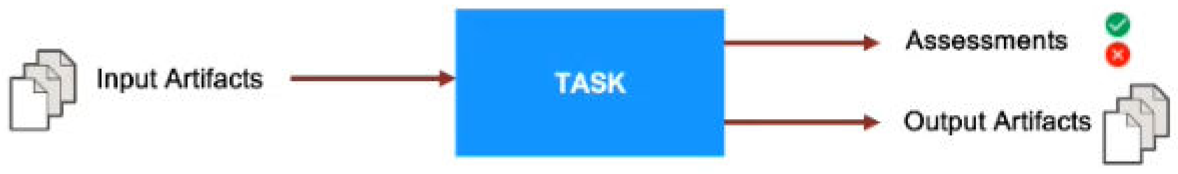
In a typical CI/CD pipeline, the CI/CD system automatically builds your source code, performs testing, packages deliverables, and deploys the packages to production. With the support package CI/CD Automation for Simulink Check, you can create a pipeline for the steps in your build and verification process, and maintain a repeatable CI/CD process for model-based design.

For example, you can create an MBD pipeline that checks modeling standards, runs tests, generates code, and performs a custom task.



You can use the customizable process modeling system to define the steps in your model-based design (MBD) pipeline. You define the steps by using a process model. A *process model* is a MATLAB script that specifies the tasks in the CI/CD process, dependencies between the tasks, and artifacts that you associate with each task.

A *task* is a single step in your process. Tasks can accept your project artifacts as inputs, perform actions, generate pass, fail, or warning assessments, and return project artifacts as outputs.



The support package contains built-in tasks for several common steps, including:

- Creating Web views for your models with Simulink Report Generator™
- Checking modeling standards with the Model Advisor
- Running tests with Simulink Test™
- Detecting design errors with Simulink Design Verifier™
- Generating a System Design Description (SDD) report with Simulink Report Generator
- Generating code with Embedded Coder®
- Checking coding standards with Polyspace® Bug Finder™
- Inspecting code with Simulink Code Inspector™
- Running tests with Simulink Test
- Generating a consolidated test results report and a merged coverage report with Simulink Test and Simulink Coverage™

The support package contains a default process model for an MBD pipeline, but you can also customize the default process model to fit your development workflow goals. For example, your custom process model might include the built-in tasks for checking modeling standards, running tests, and generating code before performing a custom task. You can customize the process model to add or remove any tasks in the MBD pipeline. You can also reconfigure the tasks in your process model to change what action a task performs or how a task performs the action.

For more information on the process modeling system, see the chapter "Author Your Process Model". For information on built-in tasks, see the chapter "Built-In Tasks — Alphabetical List".

Build System

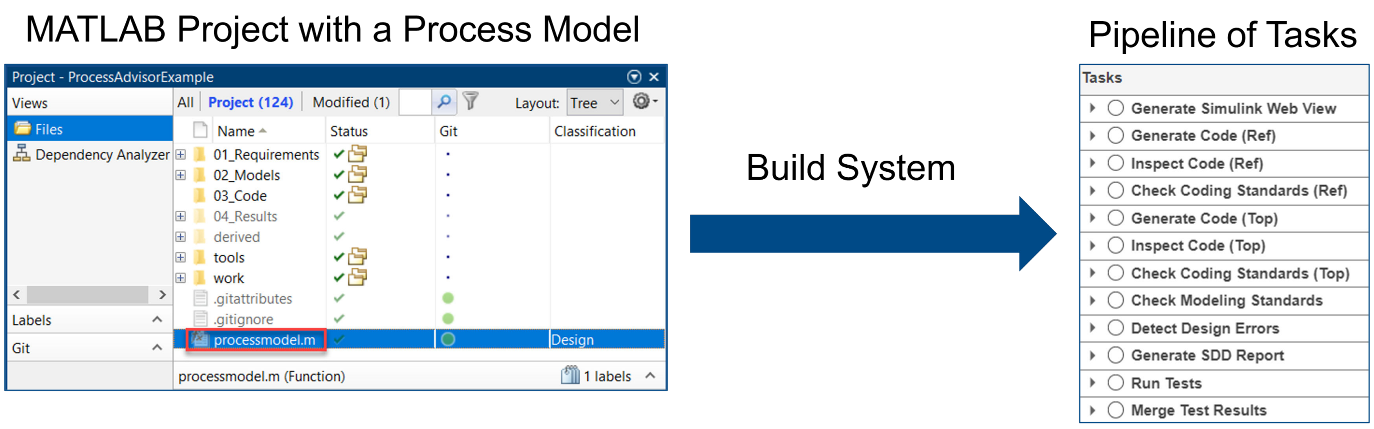
The support package CI/CD Automation for Simulink Check provides a build system that you can use to orchestrate and automate the steps in your MBD pipeline. The *build system* is software that can orchestrate tasks, efficiently execute tasks in the pipeline, and perform other actions related to the pipeline.

The build system needs:

1. A MATLAB project to analyze
2. A process model in the project that defines the tasks in the pipeline

If the project does not contain a process model, the build system copies the default process model into the project and uses the default process model to define a default MBD pipeline.

When you call the build system, the build system loads the process model, analyzes the project, and orchestrates the creation of a pipeline of tasks.


MATLAB Project with a Process Model
Build System
Pipeline of Tasks
Project - ProcessAdvisorExample
Views
Files
Dependency Analyzer
Labels
Git
All | Project (124) | Modified (1)
Layout: Tree

Name	Status	Git	Classification
01_Requirements	✓	•	
02_Models	✓	•	
03_Code	✓	•	
04_Results	✓	•	
derived	✓	•	
tools	✓	•	
work	✓	•	
.gitattributes	✓	●	
.gitignore	✓	●	
processmodel.m	✓	●	Design

processmodel.m (Function) 1 labels

Tasks
Generate Simulink Web View
Generate Code (Ref)
Inspect Code (Ref)
Check Coding Standards (Ref)
Generate Code (Top)
Inspect Code (Top)
Check Coding Standards (Top)
Check Modeling Standards
Detect Design Errors
Generate SDD Report
Run Tests
Merge Test Results

To run the tasks in the pipeline, you can call the build system using one of these approaches:

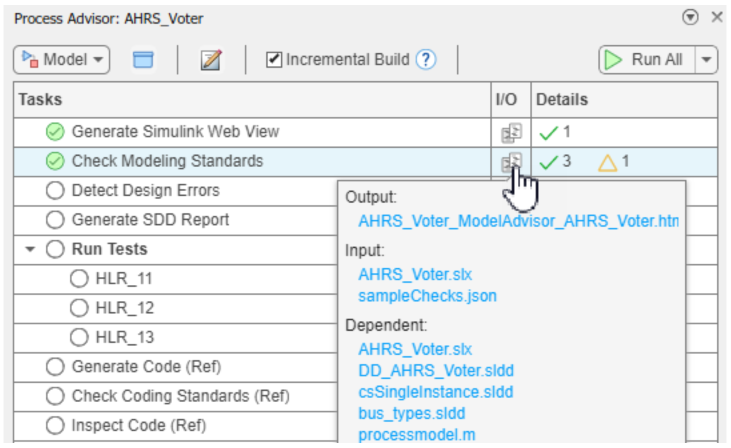
- In a CI environment by using the build system API. The build system API includes a function `runprocess` that you can use to run the tasks in a pipeline.
- Locally on your machine by using either the build system API or the **Process Advisor** app. The **Process Advisor** app is a user interface that can call the build system. The **Process Advisor** app has run buttons that you can use to run the tasks in a pipeline. If there is a failure in the CI environment, you can reproduce the issue locally by using the **Process Advisor** app.

The build system supports incremental builds. If you change an artifact in your project, the build system can detect the change and automatically determine which of the tasks in your MBD pipeline now have outdated results. In your next build, you can instruct the build system to run only the tasks with outdated results. By identifying the tasks with outdated results, the build system can help you reduce build time by reducing the number of tasks you need to re-run after making changes to your project artifacts.

Note There are limitations to the types of changes that the support package can detect. For more information, see the "Limitations on Incremental Build" section in the Appendix.

Process Advisor app

A prequalification process can help you prevent build and test failures from occurring in your CI/CD system. Use the **Process Advisor** desktop app to deploy and automate your prequalification process. You can use the app to run the tasks in your MBD pipeline and to prequalify your changes on your machine before submitting to source control. The **Process Advisor** app is a user interface that runs your tasks locally for prequalification. You can run the tasks in your MBD pipeline and to check your progress towards completing tasks in your prequalification pipeline.


Process Advisor: AHRS_Voter

Tasks	I/O	Details
Generate Simulink Web View		✓ 1
Check Modeling Standards		✓ 3 △ 1
Detect Design Errors		
Generate SDD Report		
Run Tests		
HLR_11		
HLR_12		
HLR_13		
Generate Code (Ref)		
Check Coding Standards (Ref)		
Inspect Code (Ref)		

Output:
AHRS_Voter_ModelAdvisor_AHRS_Voter.htm
Input:
AHRS_Voter.slx
sampleChecks.json
Dependent:
AHRS_Voter.slx
DD_AHRS_Voter.sldd
csSingleInstance.sldd
bus_types.sldd
processmodel.m

If you make a change to an artifact in your project, the **Process Advisor** can detect the change and automatically determine the impact of the change on your existing task results. For example, if you complete a task but then update your model, the **Process Advisor** automatically invalidates the task completion and marks the task results as outdated.

Note There are limitations to the types of changes that the **Process Advisor** app can detect. For more information, see the "Limitations on Incremental Build" section in the Appendix.

For information on **Process Advisor**, see "Run Tasks Using Process Advisor".

CI/CD System Integration

You can use the support package CI/CD Automation for Simulink Check to integrate your model-based design process into common CI/CD systems. For example, you can configure and integrate your MBD pipeline by using a YAML file to configure your pipeline for GitLab® or a Jenkinsfile for configuring your pipeline for Jenkins®.

The support package contains example pipeline configuration files:

- To open an example project that contains a GitLab pipeline file, enter this code in the MATLAB Command Window:

  ```
  processAdvisorGitLabExampleStart
  ```

  This code creates an example project that contains an example YAML file, `.gitlab-ci.yml`, in the project root. The YAML file defines a parent pipeline that uses the pipeline generator to automatically create and execute a child pipeline that runs your tasks and collects job artifacts.
- To open an example project that contains an example Jenkins pipeline file, enter this code in the MATLAB Command Window:

  ```
  processAdvisorJenkinsExampleStart
  ```

  This code creates an example project that contains an example Jenkinsfile, `Jenkinsfile`, in the project root.

For more information on CI/CD for model-based design, see https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html.

For information on CI integration, see "Integrate into CI".

Run Tasks Using Process Advisor

This chapter describes how to use the **Process Advisor** app to run tasks and prequalify your changes.

- For an example of how to run tasks and review task results, see "Prequalify Changes Before Submitting to Source Control".
- For an overview of the app, see "Quick Reference for Process Advisor App".
- For an overview of the icons that appear in the app, see "Icon Overview".

Prequalify Changes Before Submitting to Source Control

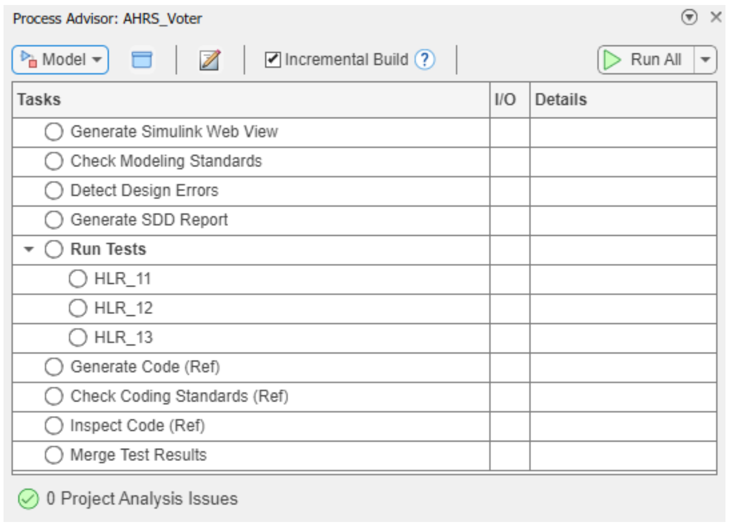
This example shows how to open the **Process Advisor** app, run tasks locally for prequalification, and review task results. The example uses an example process model to create an MBD pipeline with several common model-based design tasks. You can use the **Process Advisor** app to run each task in the MBD pipeline before submitting to source control.

1 The **Process Advisor** app runs on MATLAB projects. For this example, open the **Process Advisor** example project. In the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

This command creates a copy of the **Process Advisor** example project and opens the **Process Advisor** app for the model `AHRS_Voter`.

The **Process Advisor** pane opens to the left of the Simulink canvas. The **Process Advisor** app loads the process model, analyzes the project, and creates a pipeline of tasks. The **Tasks** column shows the pipeline of tasks associated with the current model. The tasks appears in the order that the build system will run them.



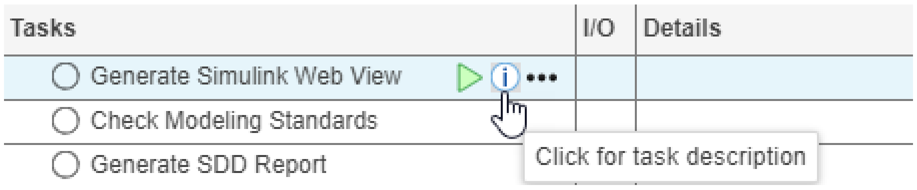
Tasks	I/O	Details
Generate Simulink Web View		
Check Modeling Standards		
Detect Design Errors		
Generate SDD Report		
Run Tests		
HLR_11		
HLR_12		
HLR_13		
Generate Code (Ref)		
Check Coding Standards (Ref)		
Inspect Code (Ref)		
Merge Test Results		

0 Project Analysis Issues

Note Each time you call `processAdvisorExampleStart`, MATLAB creates a new copy of the **Process Advisor** example project. The example project contains several models and an example process model file, `processmodel.m`, that specifies the tasks in the pipeline.

---

2 To view information about a task, point to the task in the **Tasks** column and click on the information icon ⓘ. When you click on the information icon, you can view the task description.



Tasks	I/O	Details
Generate Simulink Web View		
Check Modeling Standards		
Generate SDD Report		

3 Point to the **Generate Simulink Web View** task and click the run button ▷.

The **Generate Simulink Web View** task runs on the current model. The **Process Advisor** logs task activity in the MATLAB Command Window. When the task runs successfully, the status in the **Tasks** column shows a green circle with a check mark ✓.

4 In the top left corner of the **Process Advisor** pane, switch the filter from **Model** to **Project**.

   When the filter is set to **Project**, the Process Advisor pane shows the tasks associated with the project. By default, the **Generate Simulink Web View** task is configured to run once on each model in the project. The **Process Advisor** uses a query to find each of the models in the project and shows the names of the models as individual task iterations below the task title. The task status for **Generate Simulink Web View** shows the multiple statuses icon ● because the task passed on the `AHRS_Voter` model and was not run on the other models. For more information on icons, see "Icon Overview".

   Note You can click on an artifact name in the **Tasks** column to open the artifact.

   To open a tool associated with the task, point to the task iteration and click **...** > **Open *`Tool Name`***.

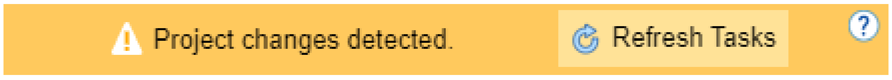
   You can also open a new window that shows the tasks associated with the project by clicking on the open project window button, to the left of the **Edit process model** icon.

   ---

5 Point to **Generate Simulink Web View** and click the run button ▷ to run the task for each model in the project.

6 In the `AHRS_Voter` model, make a change and re-save the model. For this example, you can click and drag the Model Info block to a different part of the Simulink canvas and re-save the model.

   The **Process Advisor** detects the change to the model and shows a warning banner to indicate that the app detected a change to the project and needs to refresh the task information shown in the **Process Advisor** pane.



   Note There are limitations to the types of changes that the Process Advisor can detect. For more information, see the "Limitations on Incremental Build" section in the Appendix.

   Note that sometimes the warning banner may appear while you are running tasks or after you have finished running tasks, depending on when file system events reach MATLAB.

   ---

7 Click the **Refresh Tasks** button on the warning banner to have the **Process Advisor** reanalyze the project.

   If the **Process Advisor** detects that the change caused the task to be outdated, the task status in the **Tasks** column turns gray. For example, if a task previously passed, but is now outdated, the task status in the **Tasks** column shows the Passed (Outdated) icon.

   The **Process Advisor** automatically identified that the **Generate Simulink Web View** task results are outdated for both **AHRS_Voter.slx** and **Flight_Control.slx**. The task results for **AHRS_Voter.slx** are outdated because you modified the model and directly invalidated the task results. The task results for **Flight_Control.slx** are outdated because the `AHRS_Voter` model now has outdated results and `Flight_Control` references the `AHRS_Voter`.

8 Point to the **Generate Simulink Web View** task and click the run button ▷.

The build system automatically runs an incremental build that runs only the outdated tasks and skips any tasks that already have up-to-date results.

9 For the task **Generate Simulink Web View**, point to the output files icon to view hyperlinks to the output files associated with the task.

In the column **Results**, the **Process Advisor** displays the number of passing, warning, or failing results:

- A green check mark ✓ indicates a passing result.
- An orange triangle △ indicates a warning result.
- A red "X" ✕ indicates a failing result.

The **Process Advisor** aggregates the results of each task. For this example, the **Generate Simulink Web View** task successfully created five Web views, so the column **Results** shows a value of **5** next to the green check mark for the task.

The log in the MATLAB Command Window shows the build results from running the task, including the number of task iterations that the build system was able to skip because the results were already up-to-date.

```
#### Build Status:                Pass
#### Number of tasks:             5
#### Number of tasks executed:    2
#### Number of tasks skipped:     3
```

10 Generate a PDF report with the current task results. Create a `padv.ProcessAdvisorReportGenerator` object and call `generateReport` on the object. In the MATLAB Command Window, enter:

```
rptObj = padv.ProcessAdvisorReportGenerator; % create a report object
generateReport(rptObj) % generate a report
```

The report summarizes the task statuses, task results, and other information about the task execution. For more information, see the "Generate Build Report" section of the PDF.

To run each of the tasks shown in the **Tasks** column, click **Run All**. The build system automatically skips tasks that have up-to-date results. After each task passes, you can submit your changes to source control.

For more information on the **Process Advisor** app, see "Quick Reference for Process Advisor App".

Quick Reference for Process Advisor App

Process Advisor

Automate your development workflow and prequalify changes before submitting to source control

Description

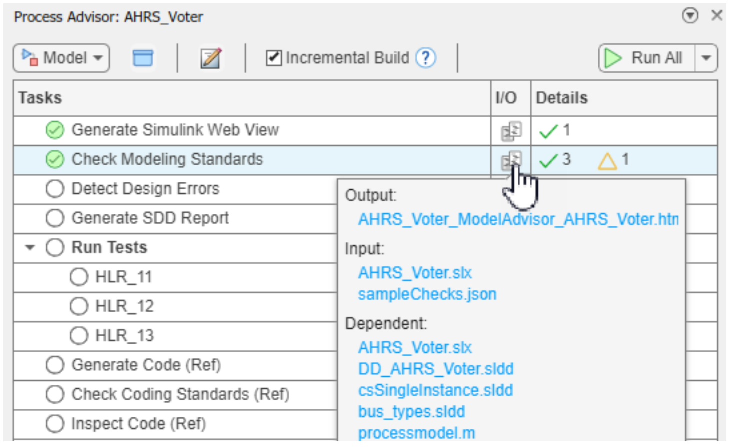
Use the **Process Advisor** app to create, deploy, and automate a consistent prequalification process for Model-Based Design (MBD). The app includes built-in tasks for performing common MBD tasks like checking modeling standards with the Model Advisor app, running tests with Simulink Test, generating code with Embedded Coder, and inspecting code with Simulink Code Inspector. You can use the customizable process modeling system to define the steps in your process and use the app to run each of the steps. As you edit and save the artifacts in your project, the app tracks changes and automatically identifies tasks and task iterations that have outdated results. The **Process Advisor** app runs your tasks locally for prequalification. The tasks run on the machine that is running MATLAB and does not use an external CI system.

To run tasks:

- Point to a task in the **Tasks** column and click the run button ▷ to run that task and any dependent tasks.
- Click **Run All** to run each of the tasks shown in the **Tasks** column.
- Click **Run All** > **Force Run All** to force the build system to run each task, even if the tasks already have up-to-date results.
- Click **Run All** > **Clean All** to clear the task results and delete task outputs for each of the tasks.
- Click **Run All** > **Refresh All** to manually refresh the list of tasks that appears in the **Tasks** column.

When the **Process Advisor** app runs tasks, a **Stop** button appears in the top-right corner. You can click the **Stop** button to stop the queued tasks from running next.

At the bottom of the **Process Advisor** app is a **Project Analysis Issues** pane. When you click on **Project Analysis Issues**, you can view any files that the app was unable to analyze. Note that the app cannot generate task iterations or detect outdated results for unanalyzed files. Fix the issues listed in the **Project Analysis Issues** pane to make sure the app can fully analyze the project, generate the expected task iterations, and detect outdated results.



Tasks	I/O	Details
✓ Generate Simulink Web View		✓ 1
✓ Check Modeling Standards		✓ 3 △ 1
Detect Design Errors		
Generate SDD Report		
Run Tests		
HLR_11		
HLR_12		
HLR_13		
Generate Code (Ref)		
Check Coding Standards (Ref)		
Inspect Code (Ref)		

Output:
AHRS_Voter_ModelAdvisor_AHRS_Voter.htn
Input:
AHRS_Voter.slx
sampleChecks.json
Dependent:
AHRS_Voter.slx
DD_AHRS_Voter.sldd
csSingleInstance.sldd
bus_types.sldd
processmodel.m

Open the Process Advisor App

- From a Simulink model: On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Process Advisor**.
- From a MATLAB Project: On the **Project** tab, in the **Tools** section, click **Process Advisor**.

Examples

Open Process Advisor For Model

Open the **Process Advisor** app for a Simulink model in a MATLAB project.

Create and open a working copy of the **Process Advisor** example project. MATLAB copies the files to an example folder so that you can edit them.

```
processAdvisorExampleStart
```

The project contains the model `OuterLoop_Control.slx`.

Open the **Process Advisor** app for the model `OuterLoop_Control.slx`.

```
processadvisor("OuterLoop_Control")
```

Open Process Advisor For Project

Open the **Process Advisor** for a MATLAB project and view the pipeline of tasks.

Create and open a working copy of an example project. MATLAB copies the files to an example folder so that you can edit them.

```
proj = Simulink.createFromTemplate("code_generation_example.sltx",...
Name="New Project");
```

Open the **Process Advisor** for the project.

```
processAdvisorWindow
```

The **Tasks** column shows the pipeline of tasks generated from the process model.

Click **Edit** 📝 to open the `processmodel.m` file that defines the process.

Programmatic Use

Note that you need to load a MATLAB project before you open the **Process Advisor**.

`processadvisor(modelName)` opens the Simulink model, `modelName`, in the current project and opens a **Process Advisor** pane to the left of the Simulink canvas.

`processAdvisorWindow()` opens the **Process Advisor** app for the current project. The app opens in a standalone window.

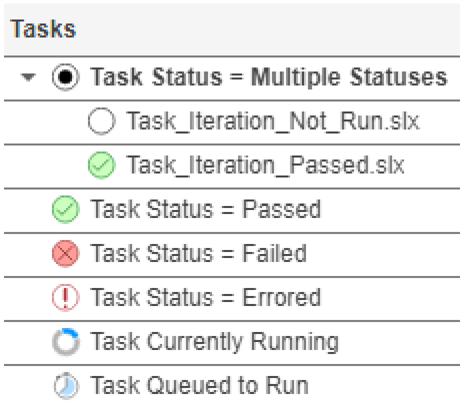
Version History

Introduced in R2022a

Icon Overview

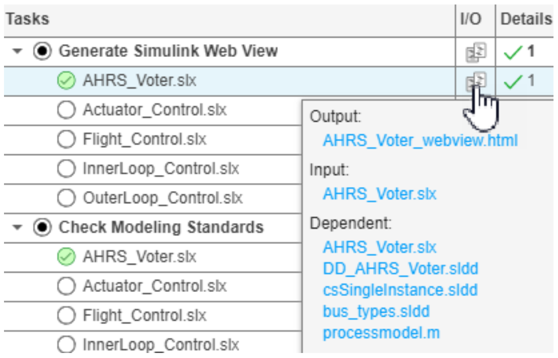
The **Process Advisor** app uses the:

- **Tasks** column to show the statuses for the tasks and task iterations.



Tasks
- Task Status = Multiple Statuses
  - Task_Iteration_Not_Run.slx
  - Task_Iteration_Passed.slx
- Task Status = Passed
- Task Status = Failed
- Task Status = Errored
- Task Currently Running
- Task Queued to Run

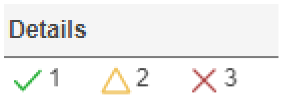
- **I/O** column to show the outputs from the tasks and task iterations.



Tasks	I/O	Details
Generate Simulink Web View		✓ 1
AHRS_Voter.slx		✓ 1
Actuator_Control.slx		
Flight_Control.slx		
InnerLoop_Control.slx		
OuterLoop_Control.slx		
Check Modeling Standards		
AHRS_Voter.slx		
Actuator_Control.slx		
Flight_Control.slx		
InnerLoop_Control.slx		

Output:
AHRS_Voter_webview.html
Input:
AHRS_Voter.slx
Dependent:
AHRS_Voter.slx
DD_AHRS_Voter.sldd
csSingleInstance.sldd
bus_types.sldd
processmodel.m

- **Details** column to show detailed results for tasks and task iterations that specify result values.



Details
✓ 1 △ 2 ✗ 3

Task Column

The status for the task or task iteration is shown on the left side of the **Tasks** column.

Statuses in the Tasks Column

Icon	Status of the Task or Task Iteration	Icon When Results Outdated	Icon When Incremental Builds Turned Off
○	Not run.	Not applicable.	Uses same icon.
◔	Currently running.	Not applicable.	Uses same icon.
⏱	Queued to run during the current build.	Not applicable.	Uses same icon.
✓	Passed.	✓	✓
⊗	Failed.	⊗	⊗
❗	Generated an error.	❗	❗
◉	Multiple statuses for different iterations of a task.	◉	Uses same icon.

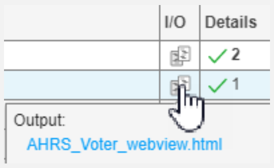
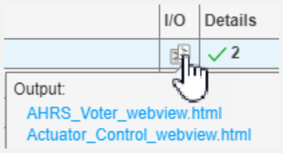
For more information on the task statuses, see the documentation for the `Status` property of the `padv.TaskResult` class in the chapter "Classes — Alphabetical List".

Note Tasks that generated an error do not rerun automatically. To rerun an errored task, point to the task and click the run button or use `runprocess` with `RerunErroredTasks` as `true`.

I/O Column

The **Process Advisor** app shows the outputs from a task or task iteration when you point to the icon in the **I/O** column.

Outputs in the I/O Column

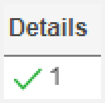
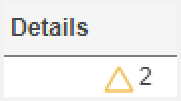
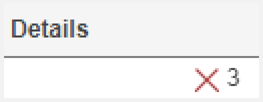
Icon	Description	Icon When Outdated
📄	The task or task iteration output a single artifact.  Output: AHRS_Voter_webview.html	📄
🗐	The task or task iteration output multiple artifacts.  Output: AHRS_Voter_webview.html Actuator_Control_webview.html	🗐

For more information on the outputs, see the documentation for the `OutputArtifacts` property of the `padv.TaskResult` class in the chapter "Classes — Alphabetical List".

Details Column

Detailed results from a task or task iteration are shown in the **Details** column.

Results in the Details Column

Icon	Result Value	Result Value for the Task or Task Iteration	Icon When Outdated
✓	Pass.	The value to the right of the icon indicates the number of result values that passed.  Details ✓ 1	✓
△	Warn.	The value to the right of the icon indicates the number of result values that generated a warning. Review the reports, outputs, or other results from the task.  Details △ 2	△
✗	Fail.	The value to the right of the icon indicates the number of result values that failed. Review any reports, outputs, or other results from the task.  Details ✗ 3	✗

For more information on the detailed results, see the documentation for the `ResultValues` property of the `padv.TaskResult` class in the chapter "Classes — Alphabetical List".

Author Your Process Model

This chapter describes how to use the customizable process modeling system to define your build and verification process.

- For an overview of the process modeling system, see "About the Process Model".
- For an example, see "Create a Custom Process Model".
- For instructions on how to use the API to author processes, see "How to Author a Process". This section includes information on how to:
  - "Create and View a Process Model"
  - "Define a Task"
  - "Add a Task"
  - "Add Inputs to a Task"
  - "Reconfigure a Task"
  - "Change Task Order and Dependencies"
- For a pseudocode example of how tasks, queries, and task iterations interact, see "How Tasks, Queries, and Task Iterations Create Results".
- For short, example process models, see "Example Process Models".

Tip You can access API help from the MATLAB Command Window by using `help` function.

For example, this code returns help information for the class `padv.Task`:

```
help padv.Task
```

The PDF also includes documentation for the API and built-ins:

- "Functions — Alphabetical List"
- "Classes — Alphabetical List"
- "Built-In Tasks — Alphabetical List"
- "Built-In Queries — Alphabetical List"

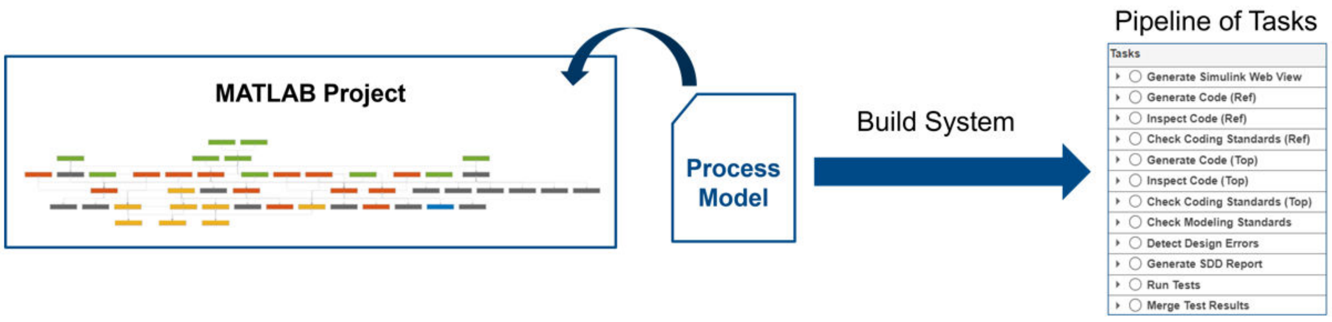
About the Process Model

There are several ways to create a process model. You can copy an empty process model into your project and add tasks to the process model. You can also copy the default process model into your project and modify that process model to fit your MBD process.

Requirements

The **Process Advisor** app requires you to have:

- Your files in a MATLAB project.
- A `processmodel.m` file on the MATLAB path. If possible, place your `processmodel.m` file in the project root folder so changes to the process model file are tracked. If your project does not have a process model and you open the Process Advisor app, the **Process Advisor** automatically creates a default process model for you at the root of the project.



You define your pipeline of tasks in the process model. The *process model* is a file, `processmodel.m`, that specifies the tasks in the process, queries that determine which artifacts to use for each task, artifacts associated with each task, and dependencies between the tasks.

Your file serves as the process model if it meets the following criteria:

- The filename is `processmodel.m`.
- The file is in the project root folder.

You do not need to manually run the process model. The process model only defines the tasks that you want to include in your pipeline. When you run tasks by using the **Process Advisor** app or the build system API, the build system automatically loads the process model to create your pipeline of tasks.

Default Process Model

The support package includes a default `processmodel.m` file that can create an MBD pipeline. You can modify the default `processmodel.m` file to fit your development process goals or you can create a new process model from an empty template.

The build system can use the default process model to create an MBD pipeline containing several common model-based design tasks. At the top of the default process model, there are several variables which you can use to control whether a task is included or excluded from the process model. For example, if you set `includeModelStandardsTask` to `false`, you can exclude the **Check Modeling Standards** task and the task does not appear in your pipeline. However, you might want to more extensively customize the process model by adding custom tasks or reconfiguring the built-in tasks to perform differently.

Custom Process Models

The support package contains several built-in tasks and built-in queries that you can use to define the steps in your process. You can use the `addTask` function to add a built-in task or a custom task to your process model.

Tip You can view the source code for the built-in tasks.

In the MATLAB Command Window, enter:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,...
"\toolbox\padv\build_service\ml\+padv\+builtin\+task")
```

MATLAB returns the path to the folder that contains the source code.

For example, the path on your machine may look like:

```
"C:\ProgramData\MATLAB\SupportPackages\R2022a_1\toolbox\padv\build_service\ml\...
+padv\+builtin\+task"
```

Create a Custom Process Model

This example shows how to create a custom process model that defines the tasks in your MBD pipeline, add built-in tasks, add dependencies between tasks and specify the task execution order, and add a custom task.

This example uses Simulink Check, Embedded Coder, and Polyspace Bug Finder.

For this example, consider a process in which you want to prequalify your changes by:

- Checking modeling standards with Model Advisor
- Generating top model code with Embedded Coder
- Analyze the generated top model code with Polyspace Bug Finder
- Run a custom `Hello, World!` task

1. If you do not already have the **Process Advisor** example project open, in the MATLAB Command Window, enter:

```
processAdvisorExampleStart
```

2. For this example, overwrite the example process model with an empty process model by entering:

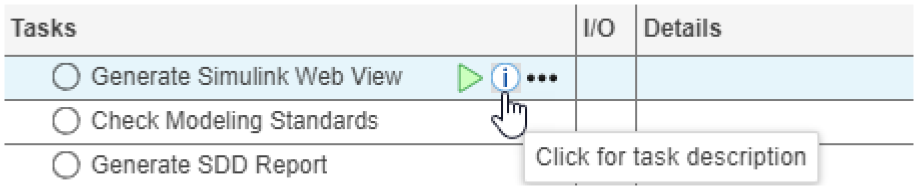
```
createprocess(Template="empty",Overwrite=true)
```

The process model at the root of the project is now empty and does not specify any tasks.

Note The support package includes a default `processmodel.m` file that can create an MBD pipeline with common, model-based design tasks. To copy the default process model into a project, enter:

```
createprocess(Template="default",Overwrite=true)
```

Note that for some default tasks, you may need to install a specific license or install the MinGW® compiler. For more information, point to a task in the **Process Advisor** app and click the information icon ⓘ. You can view the task description.



3. Open the process model for the project. In the `AHRS_Voter` model, at the top of the **Process Advisor** pane, click the edit process model icon.

The **Process Advisor** opens the process model at the root of the project. The process model is the empty process model you created by using the `createprocess` function. The empty process model contains commented out example code that shows how to specify the tasks, queries, and settings used for the pipeline.

4. Add three built-in tasks to your process model by replacing the code in your `processmodel.m` file with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    %% ADDING THREE BUILT-IN TASKS TO THE PROCESS MODEL
    % Task 1: Check Modeling Standards
        maTaskObj = addTask(pm, padv.builtin.task.RunModelStandards);
    % Task 2: Generate Top Model Code
        cgTaskObj = addTask(pm, padv.builtin.task.GenerateCodeAsTopModel);
    % Task 3: Check Coding Standards for the Top Model Code
        if exist('polyspaceroot','file') % if Polyspace installed and setup
            psTaskObj = addTask(pm, padv.builtin.task.AnalyzeTopModelCode);
        end

end
```

Note If you do not have the license for a specific task shown in an example process model, you can delete references to the task. For example, the built-in task `padv.builtin.task.AnalyzeTopModelCode` uses Polyspace Bug Finder. If you do not have a Polyspace Bug Finder license, you can delete the line that uses `addTask` to add `padv.builtin.task.AnalyzeTopModelCode` to the process.

`pm` is the `padv.ProcessModel` object for the process model.

The `addTask` function allows you to add tasks to the process model. The following table shows the connection between the task object names used in this example process model, the task instances used in the `addTask` function, and the task title shown in the **Process Advisor** app.

Task Object in Process Model	Task Instance in addTask	Task Title in Process Advisor app
`maTaskObj`	`padv.builtin.task.RunModelStandards`	**Check Modeling Standards**
`cgTaskObj`	`padv.builtin.task.GenerateCodeAsTopModel`	**Generate Code (Top)**
`psTaskObj`	`padv.builtin.task.AnalyzeTopModelCode`	**Check Coding Standards (Top)**

Note When you type `padv.builtin.task.`, you can use tab completion to see a list of the available built-in tasks.

For other example process models, see the "Example Code" section. For more information on the built-in tasks, see the Appendix of this PDF.

The output of the `addTask` function is a task object. For example, `maTaskObj` is a task object associated with the added task **Check Modeling Standards**. You can use task objects to configure task settings and add dependencies on other tasks.

5 Save the `processmodel.m` file and return to the Process Advisor pane in the window for the `AHRS_Voter` model.

   When you update the process model, the **Process Advisor** detects the change and marks any task statuses as outdated.

6 Click **Refresh Tasks** to refresh the tasks shown in the **Process Advisor** pane.

   When the filter is set to **Model**, the Process Advisor pane shows only the built-in task **Check Modeling Standards** because that is the only task associated with the `AHRS_Voter` model. The other built-in tasks only run for top models in the project.

7 In the top left corner of the **Process Advisor** pane, switch the filter from **Model** to **Project**.

   The **Process Advisor** pane shows each of the three built-in tasks. The built-in tasks **Check Coding Standards (Top)** and **Generate Code (Top)** only run for top models in the project. The **Process Advisor** found the top model, `Flight_Control`, and associated the model with the tasks.

8 Point to **Check Coding Standards (Top)** and click the run button ▷.

   The task status shows the Fails icon ⊗. The task failed to run because there is no generated code available to analyze. To run successfully, the task **Check Coding Standards (Top)** depends on having top model code to run on.

9 Re-open the process model file.

10 Specify the task execution order and dependencies by replacing the code in your process model with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    %% ADDING THREE BUILT-IN TASKS TO THE PROCESS MODEL
    %% AND SPECIFYING TASK EXECUTION ORDER AND DEPENDENCIES
    % Task 1: Check Modeling Standards
        maTaskObj = addTask(pm, padv.builtin.task.RunModelStandards);
    % Task 2: Generate Top Model Code
        cgTaskObj = addTask(pm, padv.builtin.task.GenerateCodeAsTopModel);
        % Code generation should run after checking modeling standards
        runsAfter(cgTaskObj, padv.builtin.task.RunModelStandards);
    % Task 3: Check Coding Standards for the Top Model Code
        if exist('polyspaceroot','file') % if Polyspace installed and set up
            psTaskObj = addTask(pm, padv.builtin.task.AnalyzeTopModelCode);
            % Code inspection depends on the generated code
            dependsOn(psTaskObj, padv.builtin.task.GenerateCodeAsTopModel);
        end

end
```

   The process model specifies that the code generation task, `cgTaskObj`, should run after the model standards checking task, `maTaskObj`, because even though the code generation task does not require any data or inputs from the model standards checking task, you only want to generate code for models that have had model standards checking run on them. The code analysis task, `psTaskObj`, has a data dependency on the code generation task because it needs generated code to analyze.

11 Return to the **Process Advisor** pane, click **Refresh Tasks**, and confirm the new order of the tasks.

In Process Advisor, the **Tasks** column shows the tasks in the following order: **Check Modeling Standards**, **Generate Code (Top)**, **Check Coding Standards (Top)**.

12 Point to the **Check Coding Standards (Top)** task and point to the run button ▷.

The **Process Advisor** highlights the outdated tasks and dependent tasks associated with the current task. For this example, the **Check Coding Standards (Top)** task depends on the **Generate Code (Top)** task, so the **Process Advisor** highlights both tasks. The **Check Coding Standards (Top)** task is outdated because there are no task results.

If you were to run the **Check Coding Standards (Top)** task, the **Generate Code (Top)** task would run first and the **Check Coding Standards (Top)** task would show a queued icon, indicating that the **Check Coding Standards (Top)** needs to run after the **Generate Code (Top)** task.

13 Re-open the process model file.

14 By default, the **Check Modeling Standards** task runs a subset of high-integrity systems checks specified by a default Model Advisor configuration file. Reconfigure the **Check Modeling Standards** task to run a different Model Advisor configuration file by replacing the code in your process model with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    %% ADDING THREE BUILT-IN TASKS TO THE PROCESS MODEL
    %% AND SPECIFYING TASK EXECUTION ORDER AND DEPENDENCIES
    % Task 1: Check Modeling Standards
        maTaskObj = addTask(pm, padv.builtin.task.RunModelStandards);
    % Task 2: Generate Top Model Code
        cgTaskObj = addTask(pm, padv.builtin.task.GenerateCodeAsTopModel);
        % Code generation should run after checking modeling standards
        runsAfter(cgTaskObj, padv.builtin.task.RunModelStandards);
    % Task 3: Check Coding Standards for the Top Model Code
        if exist('polyspaceroot','file') % if Polyspace installed and set up
            psTaskObj = addTask(pm, padv.builtin.task.AnalyzeTopModelCode);
            % Code inspection depends on the generated code
            dependsOn(psTaskObj, padv.builtin.task.GenerateCodeAsTopModel);
        end

    %% RE-CONFIGURING A BUILT-IN TASK
    % Specify a different Model Advisor configuration file for the task
    % Create a query that looks for your Model Advisor Configuration file
        findMyConfigFile = padv.builtin.query.FindFileWithAddress(...
            'ma_config_file', fullfile('tools','sampleChecks.json'));
    % Find the configuration file and use it as an input to the task
        addInputQueries(maTaskObj,findMyConfigFile);

end
```

To reconfigure the **Check Modeling Standards** task to run a different Model Advisor configuration, the example code specifies an input query. When you specify an *input query*, you specify which queries the task uses to find input artifacts for the task. The function `addInputQueries` allows you to specify which query the task uses to identify inputs to the task. If you do not specify an input query, the **Check Modeling Standards** task runs a default Model Advisor configuration that contains a subset of high-integrity systems checks.

This process model creates query, `findMyConfigFile`, that finds the Model Advisor configuration file for the **Check Modeling Standards** task to use. `findMyConfigFile` uses the built-in query `padv.builtin.query.FindFileWithAddress` to look for a file of type `ma_config_file` (Model Advisor configuration file), named `sampleChecks.json`, in the `tools` folder of the project. You can check which artifacts a query returns by defining and running the query in the MATLAB Command Window. For example, if you enter the following code in the MATLAB Command Window:

```
findMyConfigFile = padv.builtin.query.FindFileWithAddress(...
            'ma_config_file', fullfile('tools','sampleChecks.json'))
findMyConfigFile.run % outputs files returned by the query
```

The query returns the files found.

When you specify `addInputQueries(maTaskObj,findMyConfigFile)`, the **Check Modeling Standards** task uses the specified Model Advisor configuration file instead of the default configuration file.

Note If you wanted to specify a list of check IDs instead of a configuration, you could modify the `RunOptions` of `maTaskObj`:

```
maTaskObj.RunOptions.CheckIDList = {'mathworks.jmaab.db_0032',...
'mathworks.jmaab.jc_0281'};
```

If you specify both a Model Advisor configuration file and a list of check IDs for a task, the task uses the Model Advisor configuration file.

For other examples of how to reconfigure the built-in tasks for your process, see the default process model.

15 Add a custom task by replacing the code in your process model with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    %% ADDING THREE BUILT-IN TASKS TO THE PROCESS MODEL
    %% AND SPECIFYING TASK EXECUTION ORDER AND DEPENDENCIES
    % Task 1: Check Modeling Standards
        maTaskObj = addTask(pm, padv.builtin.task.RunModelStandards);
    % Task 2: Generate Top Model Code
        cgTaskObj = addTask(pm, padv.builtin.task.GenerateCodeAsTopModel);
        % Code generation should run after checking modeling standards
        runsAfter(cgTaskObj, padv.builtin.task.RunModelStandards);
    % Task 3: Check Coding Standards for the Top Model Code
        if exist('polyspaceroot','file') % if Polyspace installed and set up
            psTaskObj = addTask(pm, padv.builtin.task.AnalyzeTopModelCode);
            % Code inspection depends on the generated code
            dependsOn(psTaskObj, padv.builtin.task.GenerateCodeAsTopModel);
        end

    %% RE-CONFIGURING A BUILT-IN TASK
    % Specify a different Model Advisor configuration file for the task
    % Create a query that looks for your Model Advisor Configuration file
        findMyConfigFile = padv.builtin.query.FindFileWithAddress(...
            'ma_config_file', fullfile('tools','sampleChecks.json'));
    % Find the configuration file and use it as an input to the task
        addInputQueries(maTaskObj,findMyConfigFile);

    %% ADD A CUSTOM TASK
    % Add a task "My Custom Task" that calls the function "SayHello"
        myTaskObj = addTask(pm, "My Custom Task",Action=@SayHello);

end

% ADD THE FUNCTION THAT DEFINES THE TASK THE CUSTOM TASK PERFORMS
function results = SayHello(~)
    disp("Hello, World!");
    results = padv.TaskResult;
    results.ResultValues.Pass = 1;
end
```

Inside the `processmodel` function, the `addTask` function adds a custom task, `My Custom Task`, which performs the action of calling the function `SayHello`. For this example, the function `SayHello` displays the string `Hello, World!` in the log in the Command Window and returns a passing result. But you can customize the contents of the custom function to run a task that is part of your development process.

By default, custom tasks run on the whole project, but you can change the `IterationQuery` to specify the list of artifacts that the task iterates over.

16 Specify the list of artifacts that the custom task iterates over by changing line 31 of the process model to:

```
        myTaskObj = addTask(pm, "My Custom Task",Action=@SayHello,...
            IterationQuery=padv.builtin.query.FindModels);
```

The built-in query `padv.builtin.query.FindModels` finds the models in the current project. The `IterationQuery` specifies that the task should run once for each artifact returned by the query. For more information, see the "Customize the Process Model" section of the PDF.

17 Save the process model, return to the **Process Advisor** pane, and click **Refresh Tasks** to see the updated list of tasks and task execution order.

The **Process Advisor** now shows a custom task **My Custom Task** that is configured to run once for each model in the project.

How to Author a Process

Create and View a Process Model

If your project does not have a process model and you open the **Process Advisor** app, the **Process Advisor** automatically creates a default process model for you at the root of the project. Alternatively, you can use the `createprocess` function to create a process model.

- You can use the `createprocess` function to copy the default process model into any project:

  ```
  createprocess(Template="default")
  ```
- You can also use the `createprocess` function to create an empty process model:

  ```
  createprocess(Template="empty")
  ```
- If a process model already exists in the project, you can overwrite the existing process model by setting `Overwrite` to `true`.

  ```
  createprocess(Template="empty",Overwrite=true)
  ```

For more information, see the documentation for the `createprocess` function in the chapter "Functions — Alphabetical List".

View the Properties of the Process Model

The `processmodel.m` file defines the process model. You can load the process model and view the properties of the process model by using the `getprocess` function.

```
pm = getprocess

pm = 

  ProcessModel with properties:

           TaskNames: ["padv.builtin.task.DetectDesignErrors"    …    ]
          QueryNames: ["padv.builtin.query.GetDependentArtifacts"    …    ]
    DefaultQueryName: "padv.builtin.query.FindProjectFile"
        RootFileName: "processmodel.m"
```

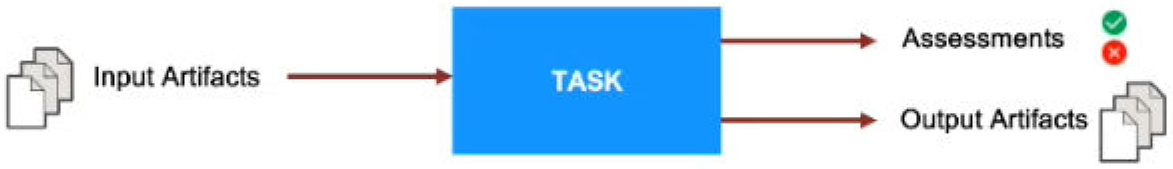
The process model, `pm`, returned by `getprocess` is a `padv.ProcessModel`. For more information, see the documentation for the `getprocess` function in the chapter "Functions — Alphabetical List".

You can use the `findTask` and `findQuery` functions on the loaded process model to find specific tasks and queries in the process.

```
findTask(pm,"padv.builtin.task.RunModelStandards")
```

Define a Task

A *task* is a single step in your process. Tasks can accept your project artifacts as inputs, perform actions, generate pass, fail, or warning assessments, and return project artifacts as outputs.



You can define a task by using either of the following approaches:

- **Function-based tasks** — Use the function `addTask` to both create and add a task. You can use the name-value arguments of the `addTask` function to define properties like the inputs to the task, what action the task performs, and the results from the task.

  For more information, see "Add a Task".
- **Class-based tasks** — Create a class that inherits from `padv.Task` and implements a `run` method.

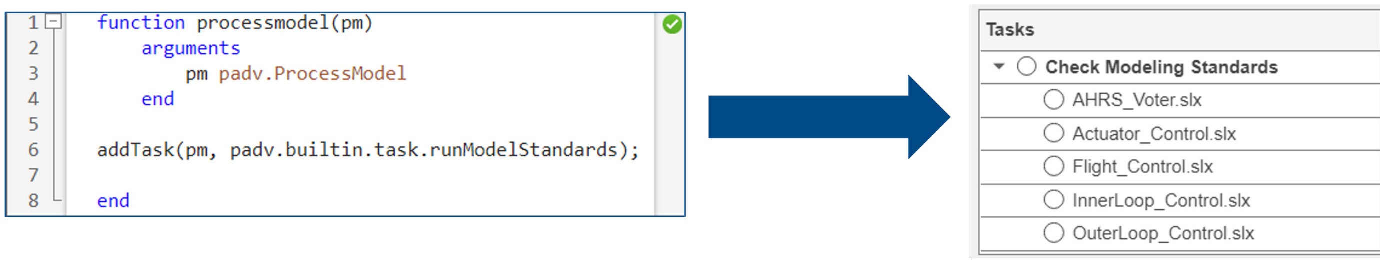
  For more information, see the documentation for the `padv.Task` class in the chapter "Classes — Alphabetical List".

You can add both function-based and class-based tasks to the process model. Class-based tasks allow you to parameterize the task using class properties, but function-based tasks are easier to implement and do not require separate class definition files.

Add a Task

You can use the function `addTask` to add a task to the process model.

The build system uses the process model to generate a pipeline of tasks.



```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

addTask(pm, padv.builtin.task.runModelStandards);

end
```

The `addTask` function requires two inputs: a process model object and a task name or task instance.

`addTask(`*`ProcessModelObject`*`, `*`TaskNameOrInstance`*`)`

Use the `addTask` function to add tasks to the process model, `pm`.

- Add a built-in task.

  For example, to add the built-in task for running model standards with the Model Advisor, `padv.builtin.task.RunModelStandards`, to a process model argument `pm`, use the following code in the process model:

  ```
  addTask(pm, padv.builtin.task.RunModelStandards);
  ```

- Add a custom task named "MyCustomTask":

  ```
  addTask(pm,"MyCustomTask")
  ```
- Specify name-value arguments. For example, specify how often a task can run by setting the `IterationQuery` argument. In this case, specify that the task runs once on each model found in the project.

  ```
  addTask(pm,"CustomTaskThatRunsForEachModel",...
          IterationQuery=padv.builtin.query.FindModels)
  ```

For more information, see the documentation for "padv.ProcessModel.addTask" in the chapter "Classes — Alphabetical List".

Add Inputs to a Task

The output of `addTask` is a configurable task object.

For certain tasks, you can use a built-in query to find specific files or types of files in your project and then use `addInputQueries` to specify the files as inputs to your task.

For example, the following code uses a query to find a Model Advisor configuration file and specifies the file as an input to the built-in task for checking modeling standards:

```
% Add task to process model
maTask = addTask(pm, padv.builtin.task.RunModelStandards());

% Find the Model Advisor configuration file
% and use the file as an input to maTask
maTask.addInputQueries(padv.builtin.query.FindFileWithAddress( ...
        'ma_config_file', fullfile('tools','sampleChecks.json')));
```

Reconfigure a Task

You can use the task object to reconfigure how a task performs an action.

For example, you can override the default output file location and specify a different location:

```
% Add task to process model
maTask = addTask(pm, padv.builtin.task.RunModelStandards());

% Specify a default report path where any output results should go
defaultResultPath = fullfile('$PROJECTROOT$', '04_Results','$ITERATIONARTIFACT$');

% Specify a subfolder 'model_standards_results'
% in the default report path as the report path for the maTask
maTask.RunOptions.ReportPath = fullfile( ...
        defaultResultPath,'model_standards_results');
```

Change Task Order and Dependencies

Specify the Task Execution Order

Use the function `runsAfter` to specify the order that tasks should run in the pipeline.

The `runsAfter` function requires two inputs:

`runsAfter(`*`TaskObject`*`,`*`Predecessors`*`)`

If one task should run before another task, even if the tasks do not depend on data from each other, use `runsAfter` to specify the order the tasks should run in the pipeline.

For example, suppose the task for checking modeling standards should run after the task that generates a Simulink Web view. Specify the desired task order by using the `runsAfter` function in the process model:

```
%% Add task to check model standards on a model
    maTask = addTask(pm, padv.builtin.task.RunModelStandards());

%% Add task to generate a Simulink WebView for a model
    slwebTask = addTask(pm, padv.builtin.task.GenerateSimulinkWebView());

%% Set Task Execution Order
    runsAfter(maTask, slwebTask);
```

Note Tasks may execute in a different order for different models in the project. For example, suppose you specify:

- `TaskC` runs after `TaskB`
- `TaskB` runs after `TaskA`

If each of the tasks runs on the current model, the task execution order is:

1. `TaskA`
2. `TaskB`
3. `TaskC`

But if `TaskB` does not run on the current model, the build system does not assume that `TaskC` should run after `TaskA`. The task execution order may be:

1. `TaskC`
2. `TaskA`

If you want to specify that predecessors need to run all task iterations or that the build system must follow a strict task order, use the name-value arguments of the `runsAfter` function: `IterationArtifactMatching` and `StrictOrdering`.

For more information, see the documentation for the `runsAfter` function in the chapter "Classes — Alphabetical List". `runsAfter` is an object function for the `padv.Task` class.

Specify a Data Dependency

Use the process model to define the tasks that the build system adds to the pipeline and the relationships between the tasks.

Use the function `dependsOn` to specify a data dependency between tasks.

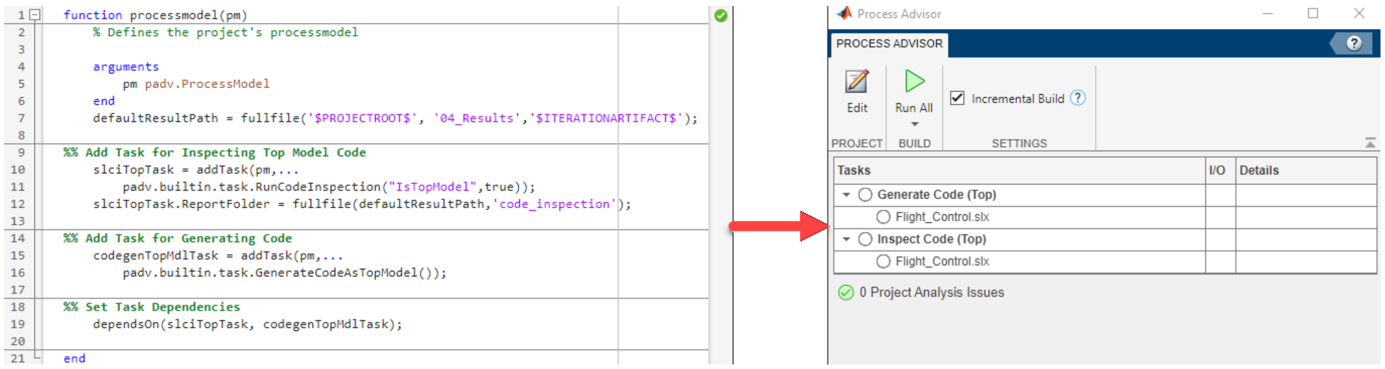
The `dependsOn` function requires two inputs:

`dependsOn(`*`TaskObject`*`,`*`Dependencies`*`)`

If the output of one task is the input to another task, there is a data dependency between the tasks.

For example, the code inspection task needs generated code to inspect, so the code inspection task depends on the code generation task. Specify the task dependency relationship by using the `dependsOn` function in the process model:

```
    defaultResultPath = fullfile('$PROJECTROOT$', '04_Results','$ITERATIONARTIFACT$');

%% Add Task for Inspecting Top Model Code
    slciTopTask = addTask(pm,...
        padv.builtin.task.RunCodeInspection("IsTopModel",true));
    slciTopTask.ReportFolder = fullfile(defaultResultPath,'code_inspection');

%% Add Task for Generating Code
    codegenTopMdlTask = addTask(pm,...
        padv.builtin.task.GenerateCodeAsTopModel());

%% Set Task Dependencies
    dependsOn(slciTopTask, codegenTopMdlTask);
```



```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end
    defaultResultPath = fullfile('$PROJECTROOT$', '04_Results','$ITERATIONARTIFACT$');

%% Add Task for Inspecting Top Model Code
    slciTopTask = addTask(pm,...
        padv.builtin.task.RunCodeInspection("IsTopModel",true));
    slciTopTask.ReportFolder = fullfile(defaultResultPath,'code_inspection');

%% Add Task for Generating Code
    codegenTopMdlTask = addTask(pm,...
        padv.builtin.task.GenerateCodeAsTopModel());

%% Set Task Dependencies
    dependsOn(slciTopTask, codegenTopMdlTask);

end
```

Tasks	I/O	Details
Generate Code (Top)		
Flight_Control.slx		
Inspect Code (Top)		
Flight_Control.slx		

0 Project Analysis Issues

If you want to specify that dependencies need to run all task iterations or that dependencies do not need to pass, use the name-value arguments of the `dependsOn` function: `IterationArtifactMatching` and `WhenStatus`.

For more information, see the documentation for the `dependsOn` function in the chapter "Classes — Alphabetical List". `dependsOn` is an object function for the `padv.Task` class.

dependsOn Versus runsAfter

In the process model, you can use the functions `dependsOn` and `runsAfter` to specify dependencies between tasks and the task execution order.

Suppose you have two tasks, `TaskA` and `TaskB`.

- If `TaskA` outputs data that `TaskB` needs, use `dependsOn` to specify the data dependency.
- If `TaskB` should not run without `TaskA` running first, use `dependsOn` to make sure that even if you only specify that run `TaskB` needs to run, `TaskA` will run first automatically (even if `TaskA` was not in the queue to run).
- If you want `TaskB` to run after `TaskA`, but only if both tasks are queued to run, use `runsAfter` to specify the desired execution order.

`dependsOn` defines a data dependency between tasks. If `TaskB` depends on `TaskA` and you run `TaskB`, `TaskA` will automatically run first (even though `TaskA` was not in the queue to run). For example, if you have the **Check Coding Standards (Ref)** task in your process, that task depends on the task that generates the code, **Generate Code (Ref)**. The task **Check Coding Standards (Ref)** depends on the code files output by the task **Generate Code (Ref)**. Additionally, **Check Coding Standards (Ref)** should not run until after **Generate Code (Ref)** runs.

`runsAfter` specifies the desired execution order for a task, without specifying a dependency between that task and the preceding task. `runsAfter` does not force the preceding task to execute, but does specify the execution order if both tasks are going to run. If `TaskB` runs after `TaskA` and you run `TaskB`, `TaskA` does not run. If you specify that you want to run both `TaskA` and `TaskB`, `runsAfter` will try to run `TaskA` before `TaskB`. For example, if you have the **Check Modeling Standards** task in your process, it may be helpful, but not a requirement for your process, that the **Check Modeling Standards** task execute before the **Generate Code (Ref)** task. In that case, you can use `runsAfter` to specify that if both **Check Modeling Standards** and **Generate Code (Ref)** are going to be run, that the system should run the **Check Modeling Standards** task first.

How Tasks, Queries, and Task Iterations Create Results

For a pseudocode example of how tasks, queries, and task iterations create results:

```
%% For each Task we can run the IterationQuery to determine what artifacts we
%% run the tasks for

IterationArtifacts=Task.IterationQuery.run();

%% You can run the Task for all or a subset of artifacts.
%% This is how we create a Task Iteration, run additional Queries and run the
%% Task, and save the Results

for IterationArtifact = IterationArtifacts

    taskIteration=TaskIteration(IterationArtifact)

%% For each Task Iteration we run the Input Queries to find the inputs for
%% for the specific task iteration

    for InputQuery = Task.InputQueries
        taskIteration.Inputs{end+1}= InputQuery.run(IterationArtifact);
    end

%% For each Input we run the Input Dependency Queries to find any additional
%% dependencies that can affect staleness

    for input = [taskIteration.Inputs{:}]
        taskIteration.AdditionalDeps{end+1}=Task.InputDependencyQuery.run(input)
    end

%% We run the Task with the inputs the iteration, and capture the
%% results

    taskIteration.Results=Task.run(taskIteration.Inputs);

%% Finally, the results of the iteration are saved

end
```

Example Process Models

Add One Built-In Task and One Custom Task

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

     % Adding a built-in task
     task1 = addTask(pm,padv.builtin.task.RunModelStandards);

     % Adding a custom task
     task2 = addTask(pm,"Custom Task",Action=@CustomAction);

     % Specify that the custom task should run after the built-in task
     runsAfter(task2,task1);

end

 function results = CustomAction(~)
     disp("Hello, world")
     results = padv.TaskResult;
 end
```

Specify a Task Execution Order

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

     %% ADD CUSTOM TASKS TO THE PROCESS MODEL
     task1 = addTask(pm,"Task 1");
     task2 = addTask(pm,"Task 2");
     task3 = addTask(pm,"Task 3");
     task4 = addTask(pm,"Task 4");
     task5 = addTask(pm,"Task 5");

     %% SPECIFY THE TASK EXECUTION ORDER
     % task2 must run after task1
         runsAfter(task2,task1,StrictOrdering=true);
     % task3 should run after task2
     % but task3 can run independently
         runsAfter(task3,task2);
     % task4 should run after task3
     % but task4 can run independently
         runsAfter(task4,task3);
     % task5 must run after task4
         runsAfter(task5,task4,StrictOrdering=true);

end
```

Include Multiple Instances of a Task

If you include duplicates of a task, the **Process Advisor** will return an error.

To include multiple instances of the same type of task, you need to specify different values of `Name` for each of the tasks. For built-in tasks, you need to override the `Name` when you create the task iteration.

For example, suppose you want to add two versions of the built-in task `padv.builtin.task.RunTestsPerTestCase`. When you create an instance of the task by using `padv.builtin.task.RunTestsPerTestCase`, you need to specify a different value for the `Name`.

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end
    taskA_v1 = addTask(pm,...
        padv.builtin.task.RunTestsPerTestCase(Name="Something else"),...
        Title="Task A - Version 1");
    taskA_v2 = addTask(pm, padv.builtin.task.RunTestsPerTestCase,...
        Title="Task A - Version 2");
end
```

You can then specify different values for the `IterationQuery` so that the tasks operate on different sets of artifacts.

Run a Custom Task on Each Model in the Project

You can use the `IterationQuery` and `InputQueries` arguments to specify the artifacts that your task runs on.

For example, you could have a custom task that analyzes each models in the project and returns the maximum cyclomatic complexity returned by the metric API:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    maTaskObj = addTask(pm, padv.builtin.task.RunModelStandards);
    cgTaskObj = addTask(pm, padv.builtin.task.GenerateCodeAsTopModel);
    cgTaskObj.dependsOn(padv.builtin.task.RunModelStandards);

    % Custom Task
    myTaskObj = addTask(pm,"Run Custom Task",Action=@MyCustomTask,...
        IterationQuery=padv.builtin.query.FindModels,...
        InputQueries=padv.builtin.query.GetIterationArtifact);

end

function results = MyCustomTask(inputs)
    % identify model name
    model = inputs{1};
    [~,modelName,~] = fileparts(model.Address);

    % Load model
    load_system(modelName)
    % Collect model metrics
    metric_engine = slmetric.Engine();
    setAnalysisRoot(metric_engine,'Root',modelName,'RootType','Model');
    metricID = 'mathworks.metrics.CyclomaticComplexity';
    execute(metric_engine,metricID);

    % Access Results
    res_col = getMetrics(metric_engine, metricID);
    maxValSeen = 0;
    metricData = {'Model','Aggregated Value'};
    for n=1:length(res_col)
        results = res_col(n).Results;
        for m=1:length(results)
            maxValSeen = max(maxValSeen,results(m).AggregatedValue);
        end
    end

    % Export Data
    metricData{2,1} = modelName;
    metricData{2,2} = maxValSeen;
    sys = char(modelName);
    filename = ['cyclomaticMetric_', sys, '.xlsx'];
    T = table(metricData);
    writetable(T,filename);

    % Determine pass / fail task results
    results = padv.TaskResult;
    if (strcmp(res_col.Category,'Compliant'))
        results.ResultValues.Pass = maxValSeen;
        results.OutputPaths = string(fullfile(pwd,filename));
    else
        results.ResultValues.Fail = maxValSeen;
    end
end
```

Control Builds

This chapter describes how to run builds and customize build execution.

- For an overview of the build system, see "Run Tasks in MBD Pipeline Using Build System".
- For information on incremental builds and full builds, see "Incremental Builds".
- For an overview of the API for running builds, see "Build System API". This section includes information on how to:
  - "Run Tasks in Pipeline"
  - "View Available Tasks in Pipeline"
  - "Generate Build Report"
- For guidance on when and how to execute builds, see "Best Practices for Effective Builds".

Run Tasks in MBD Pipeline Using Build System

You can run tasks programmatically by using the `runprocess` function.

- To run each of the tasks associated with the current project, enter:

  ```
  runprocess()
  ```

- To run a specific set of tasks, specify a list of tasks by using the `Tasks` argument. For example, you can specify the relative path to a model, use the `generateProcessTasks` function to list the tasks, and then specify the `Tasks` argument.

  ```
  % specify the relative path to the model AHRS_Voter
  model = padv.Artifact("sl_model_file", "\02_Models\AHRS_Voter\specification\AHRS_Voter.slx");
  % find the tasks associated with the model AHRS_Voter
  ahrsVoterTasks = generateProcessTasks(FilterArtifact=model)
  % run only the ahrsVoterTasks
  runprocess(Tasks=ahrsVoterTasks)
  ```

  For more information, see the documentation for the `runprocess` function in the chapter "Functions — Alphabetical List".

Incremental Builds

By default, the build system and the **Process Advisor** app perform incremental builds. Incremental builds can help you reduce the number of task iterations that you need to re-run by identifying and running only the task iterations with outdated results. If the task iteration results are up-to-date, the build system and the **Process Advisor** app skip the task iteration.

How to Disable Incremental Builds

If you want to force the build system and the **Process Advisor** app to re-run task iterations, you can disable incremental builds for the project. When you disable incremental builds, the build system and the **Process Advisor** app do not identify any results as up-to-date or outdated, and effectively force run task iterations in the project. In the **Process Advisor** app, in the **Tasks** column, the statuses for tasks and task appear in black because the app is no longer identifying up-to-date or outdated results. The statuses only indicate whether the task or task iteration passed, failed, generated an error, or did not run.

You can disable incremental builds by using one of the following approaches:

- In the **Process Advisor** app, in the toolstrip, clear the check box for the **Incremental Build** option.
- Create a `padv.Preferences` object and specify the property `IncrementalBuild` as `false`. For example:

  ```
  PREF = padv.Preferences;
  PREF.IncrementalBuild = false;
  ```

  Note that `padv.Preferences` do not persist if you restart your MATLAB session or if you run `clear classes`. To create preferences that the **Process Advisor** app and build system will use each time they run on your project, create a project startup script that specifies the properties for `padv.Preferences`.

Build System API

Run Tasks in Pipeline

You can run tasks programmatically by using the `runprocess` function.

- To run each of the tasks associated with the current project, enter:

  ```
  runprocess()
  ```

- To run a specific set of tasks, specify a list of tasks by using the `Tasks` argument. For example, you can specify the relative path to a model, use the `generateProcessTasks` function to list the tasks, and then specify the `Tasks` argument.

  ```
  % specify the relative path to the model AHRS_Voter
  model = padv.Artifact("sl_model_file", "\02_Models\AHRS_Voter\specification\AHRS_Voter.slx");
  % find the tasks associated with the model AHRS_Voter
  ahrsVoterTasks = generateProcessTasks(FilterArtifact=model)
  % run only the ahrsVoterTasks
  runprocess(Tasks=ahrsVoterTasks)
  ```

View Available Tasks in Pipeline

- Use the `generateProcessTasks` function to return a list of the available tasks in the current process model.

  ```
  generateProcessTasks
  ```

- List a set of specific tasks by using the `FilterArtifact` argument. For example, you can specify the relative path to a model and list the associated tasks.

  ```
  % specify the relative path to the model AHRS_Voter
  model = padv.Artifact("sl_model_file", "\02_Models\AHRS_Voter\specification\AHRS_Voter.slx");
  % find the tasks associated with the model AHRS_Voter
  ahrsVoterTasks = generateProcessTasks(FilterArtifact=model)
  ```

Generate Build Report

After you run the tasks in your pipeline, you can generate a report that summarizes the build results for each task in your pipeline. The report includes a:

- Summary of task statuses
- Summary of task results
- Details about the task configuration and execution

After you run a task, create a `padv.ProcessAdvisorReportGenerator` report object.

```
rptObj = padv.ProcessAdvisorReportGenerator;
```

Run `generateReport` on the report object to generate a build report in the current directory.

```
generateReport(rptObj)
```

For example, if you run the tasks in the default MBD pipeline, the report provides an overview of the:

- Model Advisor analysis, including the number of passing, warning, and failing checks
- Test results, organized by iteration
- Generated code files
- Coding standards checks

By default, the report generator generates a PDF. To generate an HTML report, specify the `Format` of the `ProcessAdvisorReportGenerator` object as `html-file`.

```
htmlReport=padv.ProcessAdvisorReportGenerator(Format="html-file");
generateReport(htmlReport);
```

Note If you want to run tasks and generate a report in batch mode, you need to specify the `runprocess` argument `ExitInBatchMode` as `false` and use the `exitCode` returned by `runprocess` to exit:

```
[buildResult, exitCode] = runprocess(ExitInBatchMode=false);
rptObj = padv.ProcessAdvisorReportGenerator();
generateReport(rptObj);
exit(exitCode);
```

Otherwise, the function `runprocess` automatically exits MATLAB before the report can generate.

Best Practices for Effective Builds

The following are best practices for an effective build schedule:

- For builds that you perform on a daily or more frequent basis, use incremental builds. Incremental builds are faster and more efficient, but incremental builds skip tasks that the build system considers up to date.

  By default, the function `runprocess` performs an incremental build:

  ```
  runprocess()
  ```

  If you use a pull request workflow, incremental builds are helpful for efficiently prequalifying changes before merging with the main repository.
- Outside of the normal build schedule, you should run a full (non-incremental) build at least one time per week and anytime you are qualifying software for a release. When you run a full build, the build system force runs each of the tasks in the pipeline. The full build makes sure that each task in the pipeline executes and that the output artifacts reflect the latest changes.

  To run a full build, use the function `runprocess` with the argument `Force` specified as `True`:

  ```
  runprocess(Force=true)
  ```

  The `Force` argument forces tasks in the pipeline to execute, even if the tasks already have up to date results.

For more information, see "Incremental Builds" and the documentation for the `runprocess` function in the chapter "Functions — Alphabetical List".

Integrate into CI

This chapter describes how to integrate MathWorks® tools into a CI system using the support package CI/CD Automation for Simulink Check.

- For an overview of system requirements and setup, see "Prerequisites".
- For information on how to integrate into GitLab, see "Integrate into a GitLab CI System".

Tip The support package includes example pipeline configuration files for GitLab and Jenkins systems.

- **For GitLab —**

  In the MATLAB Command Window, enter:

  ```
  processAdvisorGitLabExampleStart
  ```

  This code creates an example project that contains an example YAML file, `.gitlab-ci.yml`, in the project root. The YAML file defines a parent pipeline that uses the pipeline generator to automatically create and execute a pipeline that runs your tasks and collects job artifacts.

- **For Jenkins —**

  In the MATLAB Command Window, enter:

  ```
  processAdvisorJenkinsExampleStart
  ```

  This code opens a MATLAB project and an example Jenkinsfile, `Jenkinsfile`.

  Before you use the example Jenkinsfile, edit the file to specify the appropriate Git `'branch'`, `'credentialsId'`, and `'url'` for your repository.

Prerequisites

Before integrating into a CI/CD system:

1. Check that the CI system can run MATLAB. For information on the supported platforms, see https://www.mathworks.com/help/matlab/matlab_prog/continuous-integration-with-matlab-on-ci-platforms.html.

   Note License Considerations for CI: If you plan to perform CI on many hosts or on the cloud, contact MathWorks (continuous-integration@mathworks.com) for help. Transformational products such as MathWorks coder and compiler products may require client access licenses (CAL).

2. Install the support package CI/CD Automation for Simulink Check for the MATLAB instance or instances that run in your CI system.

For information on how CI/CD can apply to model-based design, see https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html.

Integrate into a GitLab CI System

A *pipeline* is a collection of automated procedures and tools that execute in a specific order to enable a streamlined software delivery process. CI systems allow you to define and configure a pipeline by using a pipeline configuration file. In GitLab, you can configure your pipeline by using a `.yml` file that you store in your project. The `.yml` file can configure different parts of your CI/CD jobs including the stages of the job, the tag for your GitLab Runner, the script that the Runner executes, and artifacts you want to attach to a successful job.

The support package CI/CD Automation for Simulink Check comes with an example `.yml` file, `.gitlab-ci.yml`, that you can add to your project to automatically run pipelines in GitLab. The example `.gitlab-ci.yml` file generates and executes pipelines for you so that you do not need to manually update any pipeline files when you change the tasks and artifacts in your project.

Integrate Using Default Options

1. Connect your MATLAB project to GitLab by following the instructions in Appendix 1 of https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models-using-gitlab.html.

2. Open the GitLab example project that contains the example `.gitlab-ci.yml` file. In the MATLAB Command Window, enter:

   ```
   processAdvisorGitLabExampleStart
   ```

   This command creates a copy of the example project and opens the example `.gitlab-ci.yml` file from the root of the project.

3. Copy the example `.gitlab-ci.yml` file into your MATLAB project and then add the file to your MATLAB project.

   Note The example `.gitlab-ci.yml` file is generic and can work with any MATLAB project that contains a `processmodel.m` file.

4. Open and inspect the `.gitlab-ci.yml` file in your project.

   The file `.gitlab-ci.yml` defines a parent pipeline. The parent pipeline uses the pipeline generator, `padv.pipeline.generatePipeline`, to automatically generate and execute a child pipeline for your MATLAB project. The options for the child pipeline are specified by the object `padv.pipeline.GitLabOptions`. For more information about parent-child pipelines, see https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html.

5. In your `.gitlab-ci.yml` file, replace `padv_demo_ci` with the CI/CD tag associated with your GitLab Runner.

   For example, if your Runner is associated with the tag `highMemory`, change the `tags` field to:

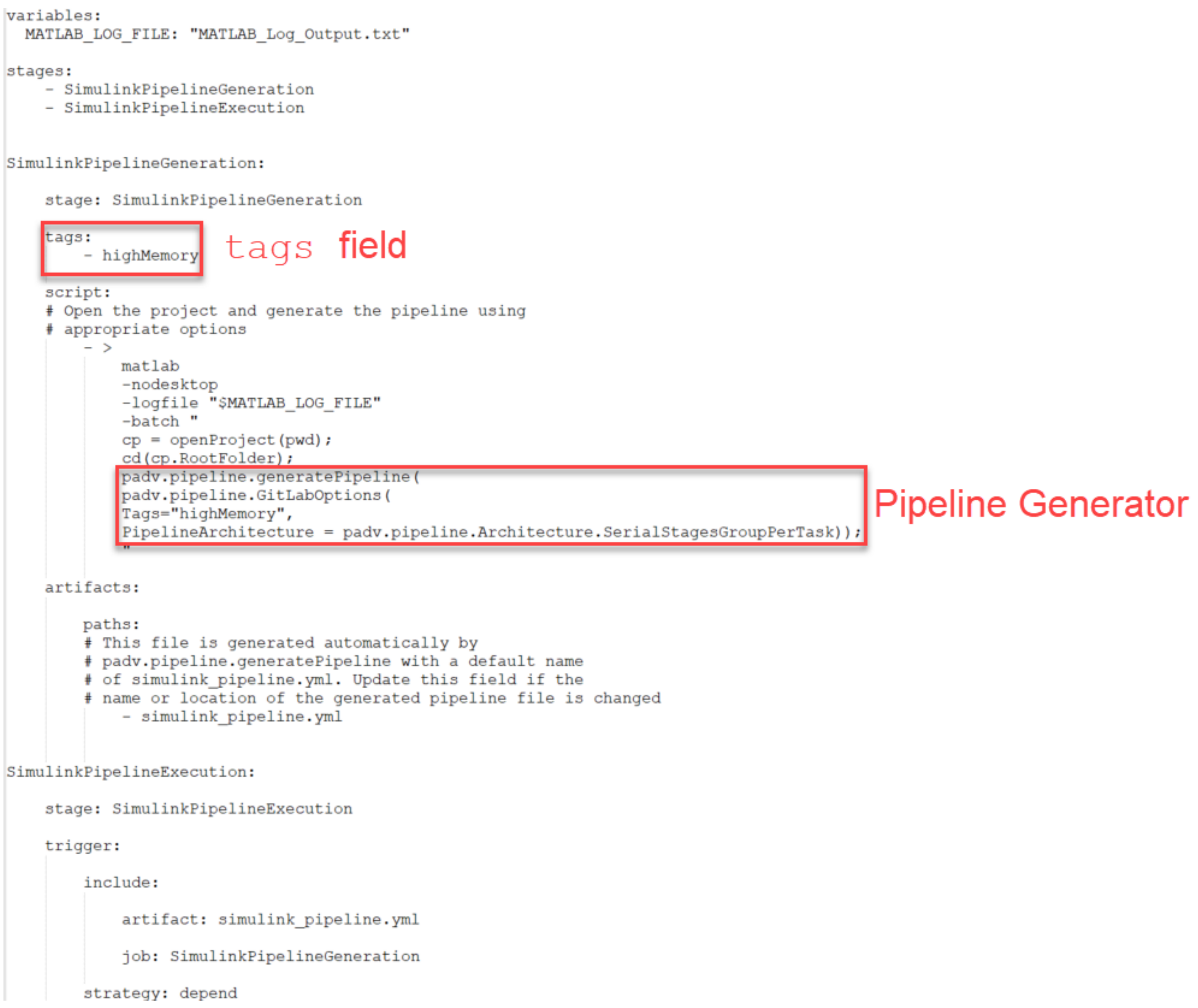
   ```
       tags:
           - highMemory
   ```

6. Modify the object `padv.pipeline.GitLabOptions` to specify the CI/CD tag associated with your GitLab Runner. `.gitlab-ci.yml` passes the tag to the child pipeline.

For example, if your Runner is associated with the tag `highMemory`, you would specify:

```
padv.pipeline.generatePipeline(
padv.pipeline.GitLabOptions(
Tags="highMemory",
PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask));
```

Now your `.gitlab-ci.yml` file will have your GitLab Runner tag specified in the `tags` field and in your `padv.pipeline.GitLabOptions` in the call to the pipeline generator.



```
variables:
  MATLAB_LOG_FILE: "MATLAB_Log_Output.txt"

stages:
    - SimulinkPipelineGeneration
    - SimulinkPipelineExecution


SimulinkPipelineGeneration:

    stage: SimulinkPipelineGeneration

    tags:
        - highMemory

    script:
    # Open the project and generate the pipeline using
    # appropriate options
        - >
            matlab
            -nodesktop
            -logfile "$MATLAB_LOG_FILE"
            -batch "
            cp = openProject(pwd);
            cd(cp.RootFolder);
            padv.pipeline.generatePipeline(
            padv.pipeline.GitLabOptions(
            Tags="highMemory",
            PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask));
            "

    artifacts:

        paths:
        # This file is generated automatically by
        # padv.pipeline.generatePipeline with a default name
        # of simulink_pipeline.yml. Update this field if the
        # name or location of the generated pipeline file is changed
            - simulink_pipeline.yml


SimulinkPipelineExecution:

    stage: SimulinkPipelineExecution

    trigger:

        include:

            artifact: simulink_pipeline.yml

            job: SimulinkPipelineGeneration

        strategy: depend
```

7 Push the changes to your GitLab repository.

By default, a GitLab project automatically considers any file named `.gitlab-ci.yml` as the CI/CD configuration file for the repository. Your GitLab Runner can now automatically generate and execute a custom pipeline for your project each time that you submit changes.

Note You do not need to update the `.gitlab-ci.yml` file if you make changes to your projects or process model. The pipeline generator generates the child pipeline using the latest project and process model. You only need to update the `.gitlab-ci.yml` file if you want to change how the pipeline generator organizes and executes the pipeline.

In GitLab, your pipeline will contain several jobs:

- Two upstream jobs:
  - **SimulinkPipelineGeneration** — Generates a child pipeline configuration file
  - **SimulinkPipelineExecution** — Executes the child pipeline configuration file
- Downstream jobs in the child pipeline:
  - One job for each task defined in the `processmodel.m` file
  - One job that collects job artifacts

If you want to change how the downstream jobs get organized and executed, you can modify the properties of the `padv.pipeline.GitLabOptions`. For example, you can modify the `PipelineArchitecture` property to change the number of stages in your child pipeline. For more information, see "Customize Child Pipeline" or enter this code in the MATLAB Command Window:

```
help padv.pipeline.GitLabOptions
```

Customize Child Pipeline

You can use the properties of `padv.pipeline.GitLabOptions` to control which GitLab Runner tags to associate with the child pipeline, the stages in the pipeline (the pipeline architecture), how tasks execute, MATLAB startup options in CI, and artifact collection for CI jobs.

For example, in your `.gitlab-ci.yml` file you can change the `script` field to specify different values for the `Tags`, `RerunFailedTasks`, and `PipelineArchitecture` properties in `padv.pipeline.GitLabOptions`:

```
script:
# Open the project and generate the pipeline using
# appropriate options
    - >
        matlab
        -nodesktop
        -logfile "$MATLAB_LOG_FILE"
        -batch "
        cp = openProject(pwd);
        cd(cp.RootFolder);
        padv.pipeline.generatePipeline(
        padv.pipeline.GitLabOptions(
        Tags="highMemory",
        RerunFailedTasks = true,
        PipelineArchitecture = padv.pipeline.Architecture.SerialStages));
        "
```

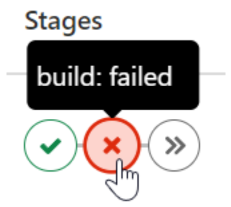
This code specifies that the pipeline should be associated with the GitLab Runner tag `highMemory`, should try to rerun failed tasks, and should use a serial stage pipeline architecture that creates a job for each task iteration.

To see a list of the available properties in the MATLAB Command Window, enter:

```
help padv.pipeline.GitLabOptions
```

Customize Pipeline Architecture

After you run a pipeline, GitLab shows the overall status of the pipeline and the status of each stage in the pipeline. For example, the **Stages** column can show a pipeline mini graph that shows the first stage passed, the second stage failed, and the third stage was skipped.



If you want to group the information that appears in your pipeline results, you can specify a pipeline architecture that defines more stages. If a pipeline has more stages, you can more easily identify where any failures occurred, but the pipeline execution may not be as efficient.

If you specify the pipeline architecture as:

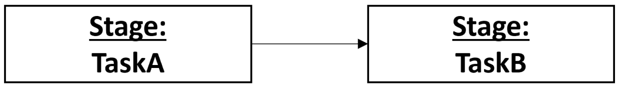
- `SingleStage` — The generated pipeline contains a single stage, **Runprocess**, that runs all tasks.

  ```
  padv.pipeline.GitLabOptions(...
  PipelineArchitecture = padv.pipeline.Architecture.SingleStage)
  ```

Stage:
Runprocess

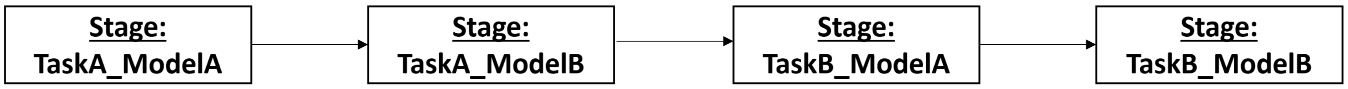
- `SerialStagesGroupByTask` — The generated pipeline contains one stage for each type of task.

  ```
  padv.pipeline.GitLabOptions(...
  PipelineArchitecture = padv.pipeline.Architecture.SerialStagesGroupPerTask)
  ```



- `SerialStages` — The generated pipeline contains one stage for each task iteration.

  ```
  padv.pipeline.GitLabOptions(...
  PipelineArchitecture = padv.pipeline.Architecture.SerialStages)
  ```



Comparison of Pipeline Architectures

The following table compares the different pipeline architectures.

Pipeline Architecture	Description	Pros	Cons
`SingleStage`	One stage for all tasks	Efficient execution since the CI system only launches MATLAB and the MATLAB project one time	Difficult to identify where a failure occurred. If the pipeline fails, you must investigate the logs, build report, or other output files to identify which specified task or task iteration failed.
`SerialStagesGroupByTask`	One stage for each task. The stages run in series, not in parallel.	If the pipeline fails, you can see which task failed, directly in the pipeline results.	Less efficient execution because the CI system has to close and reopen MATLAB and the MATLAB Project one time for each stage
`SerialStages`	One stage for each task iteration. The stages run in series, not in parallel.	If the pipeline fails, you can see which task iteration failed, directly in the pipeline results.	Inefficient execution because the CI system has to close and reopen MATLAB and the MATLAB Project one time for each stage

Troubleshooting and Limitations

Troubleshooting Missing Tasks or Artifacts

When you use CI/CD Automation for Simulink Check, the support package creates a digital thread to capture the attributes and unique identifiers of the artifacts in your project. The digital thread is a set of metadata information about the artifacts in a project, the artifact structure, and the traceability relationships between artifacts. The **Process Advisor** app and build system monitor and analyze the digital thread to identify artifacts, detect changes to project files, generate task iterations, and identify outdated task results.

See the next sections for troubleshooting steps and limitations.

Artifact Issues

Before you begin troubleshooting the **Process Advisor** app or build system, check that:

- Artifacts are saved in the project.
- Artifacts are not in a referenced project. Project references are not fully supported.
- Artifacts are on the MATLAB search path before you open the **Process Advisor** app.
- You used the **Process Advisor** app or build system to run your tasks and to collect task results.
- Artifacts are not saved to a prohibited output folder. Prohibited output folders include the simulation cache, project resources folder, and `.SimulinkProject`.
- You have a compiler configured. You should use the same compiler that you use in the target development environment. If you only have the MinGW compiler installed on your system, the `mex` command automatically chooses MinGW.

Resolve Path Issues

If an artifact is not on the MATLAB search path, add the artifact to your MATLAB project, then close and re-open the project. When you re-open the project, the MATLAB search path reflects the updated search path.

Note If a test harness is saved inside a model file, the **Process Advisor** and build system return an incorrect warning that the internal test harness is not on the MATLAB search path. Ignore the warning, and, if possible, convert your internal test harnesses to external test harnesses so that the support package can differentiate between changes to the test harness and changes to the main model.

To convert a test harness, open Simulink Test for the main model and, on the **Tests** tab, click **Manage Test Harnesses > Convert to External Harnesses**. Click **Yes** to convert the affected test harnesses.

Unsupported Modeling Constructs

If there are issues with an artifact, check that the artifact does not use the following unsupported modeling constructs:

Affected Artifact	**Unsupported Construct**
Library	Library forwarding table
	Self-modifiable masks
Model	Saved in release R2012a or earlier
	Model loading callbacks
	Model shadowing
Test case	MATLAB-based Simulink test

Other Limitations

There are known limitations in the **Process Advisor** app and build system:

- If a top model and at least one referenced model have unsaved changes, the **Process Advisor** is unable to save the top model and generates the error:`The following files were not able to be saved: <Path to top model>`
- If a test harness is saved inside a model file, the **Process Advisor** and build system return an incorrect warning that the internal test harness is not on the MATLAB search path. Ignore the warning, and, if possible, convert your internal test harnesses to external test harnesses so that the support package can differentiate between changes to the test harness and changes to the main model.
- This issue may affect Linux® users: If you point to a task and try to click on more options in the **...**menu, the standalone **Process Advisor** app may miss the mouse click. As a workaround, use the arrows on the keyboard and press **Enter** to interact with the options in the menu.

Limitations on Incremental Build

There are changes that incremental build does not detect. Tasks depending on those changes will remain up-to-date and will not execute with **Run All**. If incremental build does not detect changes to a file that a task depends on, the file is an *untracked dependency*.

The table in this section lists the known untracked dependencies.

- The **Artifact** column lists the artifacts with known untracked dependencies.
- The **Untracked Dependency** column lists the files that incremental build does not detect changes to. Changes to these files do not cause tasks associated with the artifact to become outdated.

For example, if you have a model that uses a referenced global workspace variable and you make a change to the variable, the task results associated with the model will not become outdated. The table shows:

- **Artifact**: Model
- **Untracked Dependency**: Referenced global workspace variable

Artifact	Untracked Dependency
Model	Model callbacks
	Referenced global workspace variables*
	Global enumeration definitions*
	Externally-saved model workspace variables (if auto-initialized)
	Data or functions referenced in masks or callbacks inside the model
	Known dependencies specified in the model reference rebuild options of a configuration set
	Simulation inputs and simulation outputs specified in model configuration sets
	Signal Editor scenarios
	C code referenced in C Caller blocks
	Code inside SIL (software-in-the-loop) blocks
	Files associated with S-Functions
	Code replacement libraries
	Custom code
	System Composer™ profiles or stereotypes
Test case	MATLAB code in: • Pre-load, post-load, clean-up, and assessment callbacks • Custom criteria
	External configurations
	MATLAB test files

*If possible, use a Simulink Data Dictionary file instead. The digital thread tracks changes to data dictionaries.

Note If you do not want the build system or the **Process Advisor** app to run incremental builds, you can disable incremental builds for a project. For more information, see the section "How to Disable Incremental Builds".

You can also force up-to-date tasks to execute by using one of these approaches:

- In the **Process Advisor** app, either point to a task and click the run button ▷ or click **Run All** > **Force Run All**.
- For the `runprocess` function, specify `Force` as `true`.

Note The build system and **Process Advisor** app are able to track the following test case dependencies:

- Baseline files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, `.xls`, and `.mldatx` format.
- Input files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, and `.xls` format.
- Parameter override files in `.mat`, `.xlsm`, `.xlsb`, `.xlsx`, `.xls`, and `.m` format.

Functions — Alphabetical List

The API includes the following functions:

Create, Access, and Run Process Model

Function	Description
`createprocess`	Create a process model
`getprocess`	Get process model object for process model in project
`runprocess`	Run task iterations defined by the process model

Get Individual Task Iterations and Results from Process Model

Function	Description
`createProcessTaskID`	Generate an ID for a specific task iteration defined by the process model
`generateProcessTasks`	Generate a list of the IDs for the task iterations defined by the process model
`getProcessTaskResults`	Get available results and result details for task iterations defined by the process model

Open Process Advisor App

Function	Description
`processadvisor`	Open the **Process Advisor** app for a specific Simulink model
`processAdvisorWindow`	Open the **Process Advisor** app for a MATLAB project

The function reference pages are listed alphabetically on the following pages.

Tip You can also access API help from the MATLAB Command Window by using `help` function.

For example, this code returns help information for the function `runprocess`:

```
help runprocess
```

createprocess

Create process model

Syntax

```
processModelPath = createprocess()
processModelPath = createprocess(Name=Value)
```

Description

`processModelPath = createprocess()` creates a process model at the project root and returns the path to the created process model. The process model is saved as `processmodel.m`.

By default, the process model is a default process model that can create a model-based design pipeline. You can only call `createprocess` if you have a MATLAB project open.

`processModelPath = createprocess(Name=Value)` specifies the output process model using one or more `Name=Value` arguments.

Examples

Create Process Model

Open a project that does not have a process model and copy the default process into the project.

Open an example MATLAB project, `dashboardCCProjectStart`, that does not have a process model.

```
dashboardCCProjectStart
```

Create a process model for the project.

```
processModelPath = createprocess
```

`createprocess` copies the default process model into the project root and saves the path to the process model to `processModelPath`.

Create a project object for the currently loaded project.

```
myProject = currentProject;
```

Add the process model file to the current project.

```
addFile(myProject,processModelPath)
```

Open the Process Advisor app in a standalone window to view the tasks associated with the project and project artifacts.

```
processAdvisorWindow
```

Overwrite Process Model with Empty Process

Open a project and overwrite the process model with an empty process model.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Use `createprocess` to overwrite the existing process model with an empty process model.

```
processModelPath = createprocess(Template="empty",Overwrite=true)
```

Open the created process model to view the commented-out example code.

```
open(processModelPath)
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `processModelPath = createprocess(Overwrite=true)`

`Template` — Name of predefined process model template
`"default"` (default) | `"empty"`

Name of predefined process model template, specified as either:

- `"default"` — Process model file that includes several built-in tasks
- `"empty"` — Process model file that contains commented-out example code for adding built-in and custom tasks

Example: `"empty"`

Data Types: `char` | `string`

`Overwrite` — Setting to overwrite existing process model
`false` or `0` (default) | `true` or `1`

Setting to overwrite existing process model, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Example: `true`

Data Types: `logical`

Output Arguments

`processModelPath` — Path to created process model
character vector

Path to created process model, returned as a character vector.

By default, `createprocess` creates a process model at the project root.

Alternative Functionality

App

If a project does not have a process model, you can use the **Process Advisor** app to create the default process model. To open the **Process Advisor** app for a project, in the MATLAB Command Window, enter:

```
processAdvisorWindow
```

When you open the **Process Advisor** app on a project that does not have a process model, the app automatically creates a copy of the default process model at the root of the project.

Version History

Introduced in R2022a

createProcessTaskID

Generate ID for specific task iteration defined by process model

Syntax

```
ID = createProcessTaskID(task,artifact)
```

Description

`ID = createProcessTaskID(task,artifact)` generates the identifier, `ID`, for an individual task iteration defined by the process model. A *task iteration* is the pairing of a task, `task`, to a specific project artifact, `artifact`.

Examples

Run One Task on One Artifact

Suppose you have a process model with several tasks, but right now you only want to run the task `padv.builtin.task.RunModelStandards` on the model `AHRS_Voter.slx`. Use the function `createProcessTaskID` to generate the ID for a specific task iteration, then use the function `runprocess` to run only that specific task iteration.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Specify a task that exists in the process model. For this example, specify the built-in task for running Model Advisor checks, `padv.builtin.task.RunModelStandards`.

```
task = padv.builtin.task.RunModelStandards;
```

Use `padv.Artifact` to specify the project artifact that you want the task to run on. For this example, the artifact type is `sl_model_file` because the artifact is a Simulink model and the address is the path to `AHRS_Voter.slx`, relative to the project root.

```
artifactType = "sl_model_file";
address = "02_Models/AHRS_Voter/specification/AHRS_Voter.slx";
artifact = padv.Artifact(artifactType,address);
```

Use the task instance and artifact to generate the ID for the specific task iteration.

```
runModelStandards_for_AHRS_Voter = createProcessTaskID(task,artifact)
```

```
runModelStandards_for_AHRS_Voter =

"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter
```

Use the function `runprocess` to run the task iteration.

```
runprocess(Tasks = runModelStandards_for_AHRS_Voter)
```

When you specify the `Tasks` value as the ID for a single task iteration, the function `runprocess` runs only the specified task iteration. For this example, `runprocess` runs only the task iteration associated with the task `padv.builtin.task.RunModelStandards` and the artifact `AHRS_Voter.slx`.

Note You can only run task iterations that are already defined by the process model. For each task iteration, the task must be a task that you added to the process model and the artifact must be an artifact that you specified the task runs on.

For example, if the task `myCustomTask` is a task that runs once for each model in the project, you cannot run using the ID `"myCustomTask|project|ProcessAdvisorExample.prj"` until you specify, in the process model, that `myCustomTask` is a task that runs once for the project.

Input Arguments

`task` — Task name or subclass of padv.Task
string | character vector | `padv.Task` object

Either:

- Name of task, specified as a string or character vector. The name of a task is stored in the `Name` property of the task. For example, `"name_of_my_custom_task"`.
- Subclass of `padv.Task`, specified as a `padv.Task` object. Built-in tasks are subclasses of `padv.Task`. For example, you can specify the `padv.Task` object `padv.builtin.task.RunModelStandards` for the `task` argument.

Example: `"name_of_my_custom_task"`

Example: `"padv.builtin.task.RunModelStandards"`

Example: `padv.builtin.task.RunModelStandards`

Data Types: `char` | `string`

`artifact` — File in project
`padv.Artifact` object

File in project, specified as a `padv.Artifact` object.

Example: `padv.Artifact("project","ProcessAdvisorExample.prj")`

Example: `padv.Artifact("sl_model_file", "02_Models/AHRS_Voter/specification/AHRS_Voter.slx")`

Output Arguments

`ID` — Identifier for task iteration defined by process model
string

Identifier for task iteration defined by the process model, returned as a string.

IDs take the form: "*`taskNameOrObject|fileType|relativePath`*", where `relativePath` is the path relative to the project root.

Example IDs:

- `"myCustomProjectTask|project|ProcessAdvisorExample.prj"`
- `"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`
- `"padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"`

Alternative Functionality

App

You can also use the **Process Advisor** app to run individual task iterations in the process. To open the **Process Advisor** app for a project, in the MATLAB Command Window, enter:

```
processAdvisorWindow
```

Version History

Introduced in R2022a

generateProcessTasks

Get list of IDs for task iterations in MBD pipeline

Syntax

```
IDs = generateProcessTasks()
IDs = generateProcessTasks(FilterArtifact=artifact)
```

Description

`IDs = generateProcessTasks()` returns identifiers, `IDs`, for each of the task iterations in the model-based design (MBD) pipeline.

By default, `generateProcessTasks` returns an ID for each combination of tasks and associated project artifacts in the MBD pipeline.

`IDs = generateProcessTasks(FilterArtifact=artifact)` filters the list of IDs in the MBD pipeline to show only IDs for task iterations associated with a specific artifact, `artifact`.

Examples

List IDs for Each Task Iteration in MBD Pipeline

Suppose you have a process model that adds several tasks to the process. Use the function `generateProcessTasks` to list the IDs for each task iteration in the MBD pipeline.

Open the Process Advisor example project, which contains an example process model.

```
processAdvisorExampleStart
```

List the IDs for each task iteration in the MBD pipeline.

```
IDs = generateProcessTasks()
```

Run Each Task Associated with an Artifact

Suppose you have a process model that adds several tasks to the process, but right now you only want to run the tasks associated with one specific artifact. Use the function `generateProcessTasks`, but filter the list of IDs to only include task iterations associated with a specific model in the project, `AHRS_Voter.slx`.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Use `padv.Artifact` to specify the project artifact that you want the task to run on. For this example, the artifact type is `sl_model_file` because the artifact is a Simulink model and the address is the path to `AHRS_Voter.slx`, relative to the project root.

```
artifactType = "sl_model_file";
address = "02_Models/AHRS_Voter/specification/AHRS_Voter.slx";
artifact = padv.Artifact(artifactType,address);
```

Get a list of the IDs for the task iterations in the MBD pipeline, but filter the list to include only task iterations associated with the artifact `AHRS_Voter.slx`.

```
IDs_AHRS_Voter = generateProcessTasks(FilterArtifact=artifact);
```

Use the function `runprocess` to run only the task iterations associated with the artifact `AHRS_Voter.slx`.

```
runprocess(Tasks=IDs_AHRS_Voter)
```

When you specify the `Tasks` value as a list of IDs for task iterations, the function `runprocess` runs only the specified task iterations. For this example, `runprocess` runs only the task iterations associated with the artifact `AHRS_Voter.slx`.

Note You can only run task iterations that are already defined in the process model. For each task iteration, the task must be a task that you added to the process model and the artifact must be an artifact that you specified the task runs on.

For example, if the task `myCustomTask` is a task that runs once for each model in the project, you cannot run using the ID `"myCustomTaskTask|project|ProcessAdvisorExample.prj"` until you specify, in the process model, that `myCustomTask` is a task that runs once for the project.

Input Arguments

`artifact` — File in project

`padv.Artifact` object

File in project, specified as a `padv.Artifact` object.

Example: `padv.Artifact("project","ProcessAdvisorExample.prj")`

Example: `padv.Artifact("sl_model_file", "02_Models/AHRS_Voter/specification/AHRS_Voter.slx")`

Output Arguments

`IDs` — Identifiers for task iterations defined by process model

string

Identifiers for task iterations in the MBD pipeline, returned as a string.

IDs take the form: "*`taskNameOrObject|fileType|relativePath`*", where `relativePath` is the path relative to the project root.

Example IDs:

- `"myCustomProjectTask|project|ProcessAdvisorExample.prj"`
- `"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`

- `"padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"`

Alternative Functionality

App

You can also use the **Process Advisor** app to run individual task iterations in the process or to view task iterations for a specific model.

- To open the **Process Advisor** app for a project, in the MATLAB Command Window, enter:

  ```
  processAdvisorWindow
  ```
- To open the **Process Advisor** app for a specific model, provide the name of the model, *`modelName`*, to the function `processadvisor`:

  ```
  processadvisor(modelName)
  ```

Version History

Introduced in R2022a

getprocess

Get process model object for process model in project

Syntax

```
processModelObject = getprocess()
```

Description

`processModelObject = getprocess()` returns a process model object, `processModelObject`, for the process model in the project. You can use the process model object to view the properties of the process model in the project. For more information, see the documentation for "padv.ProcessModel" in the chapter "Classes — Alphabetical List".

If the current project does not have a process model, the function `getprocess` automatically creates a new process model at the root of the project.

Examples

Find the Default Query for the Current Process

Use `getprocess` to find the default query that the current process model uses. If you have a task that does not specify an iteration query, the default query defines which artifacts the process iterates over. By default, custom tasks run once per project because the default query is `"padv.builtin.query.FindProjectFile"`.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Get the properties of the current process model.

```
currentProcessModelProperties = getprocess()
```

```
currentProcessModelProperties = 

  ProcessModel with properties:

           TaskNames: ["padv.builtin.task.AnalyzeRefModelCode"    …    ]
          QueryNames: ["padv.builtin.query.FindModels"    …    ]
    DefaultQueryName: "padv.builtin.query.FindProjectFile"
        RootFileName: "processmodel.m"
```

Get the default query for the current process model.

```
defaultQuery = currentProcessModelProperties.DefaultQueryName
```

```
defaultQuery = 

    "padv.builtin.query.FindProjectFile"
```

Suppose you want to override the default query for the current process model. Open the process model and use the `padv.ProcessModel` object `pm` to specify a different default query. For this example, change the default query to `padv.builtin.query.FindModels` by adding the following line of code to the process model:

```
pm.DefaultQueryName = "padv.builtin.query.FindModels";
```

Now if you add a new custom task to the process model and do not specify an iteration query, the custom task runs once for each model in the project.

Output Arguments

`processModelObject` — Properties of process model
`padv.ProcessModel` object

Properties of process model, returned as a `padv.ProcessModel` object.

The `padv.ProcessModel` object returns the names of the tasks, queries, default query, and root process model file for the process.

Version History
Introduced in R2022a

getProcessTaskResults

Get available task results and result details for task iterations in MBD pipeline

Syntax

```
[IDsWithTaskResults,taskResults,taskResultsOutdated] =
getProcessTaskResults()
```

Description

`[IDsWithTaskResults,taskResults,taskResultsOutdated] = getProcessTaskResults()` returns available task results and result details for the task iterations in the MBD pipeline. The function returns the identifiers for task iterations that have task results, `IDsWithTaskResults`, the current task results, `taskResults`, and a logical value that indicates if the task results are outdated, `taskResultsOutdated`.

If you do not have task results, use the function `runprocess` to run tasks and generate results. The function `getProcessTaskResults` only returns information related to task iterations that are defined in the process model. If you have task results from a task iteration that is not in the process model, the function does not return information related to those task results.

Examples

Get Output Artifacts from Task Results

Get the available task results for a task iteration and use the result details to find information about the output artifacts of the task iteration.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

List the IDs for each task iteration in the MBD pipeline.

```
IDs = generateProcessTasks();
```

Run the first task iteration in the list.

```
runprocess(Tasks=IDs(1))
```

For this example, the build system runs the task `padv.builtin.task.GenerateSimulinkWebView` for the model `AHRS_Voter.slx`.

Get the available task results and result details.

```
[IDsWithResults,results,outdated] = getProcessTaskResults()
```

```
IDsWithResults =

    "padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_Voter/specification/
```

```
results = 

  TaskResult with properties:

             Status: Pass
    OutputArtifacts: [1×1 padv.Artifact]
            Details: [1×1 struct]
       ResultValues: [1×1 struct]


outdated =

  logical

   0
```

Get the output artifacts from the result. For this example, the result is a Simulink Web View for the model `AHRS_Voter.slx`.

```
webView = results.OutputArtifacts
```

```
webView = 

  Artifact with properties:

              Type: "padv_output_file"
            Parent: [0×0 padv.Artifact]
           Address: "04_Results/AHRS_Voter/webview/AHRS_Voter_webview/AHRS_Voter_webview.html"
              UUID: "6b37eb48-d694-4daf-a5dd-024a4bf2348c"
             Label: [0×0 string]
    StorageAddress: [0×0 string]
```

Output Arguments

`IDsWithTaskResults` — Identifiers for task iterations that have task results and are defined in process model
string | string array

Identifiers for task iterations that have task results and are defined in the process model, returned as a string or string array.

- If you do not have task results for task iterations in your process model, `IDsWithTaskResults` returns an empty array, `[]`. You can use the function `runprocess` to run tasks and generate results.
- If you have task results for task iterations that are not in your process model, `IDsWithTaskResults` returns an empty array, `[]`.
- If you have task results for task iterations that are in your process model, `IDsWithTaskResults` returns the IDs for the task iterations that have task results.

IDs take the form: "*`taskNameOrObject|fileType|relativePath`*", where `relativePath` is the path relative to the project root.

Example IDs:

- `"myCustomProjectTask|project|ProcessAdvisorExample.prj"`
- `"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`
- `"padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"`

`taskResults` — Results for task iterations
`padv.TaskResult` | `padv.TaskResult` array

Results for task iterations, returned as a `padv.TaskResult` or `padv.TaskResult` array.

- If you do not have task results for task iterations in your process model, `taskResults` returns an empty array, `[]`.
- If you have task results for task iterations that are not in your process model, `taskResults` returns an empty array, `[]`.
- If you have task results for task iterations that are in your process model, `taskResults` returns a `padv.TaskResult` or `padv.TaskResult` array.

`padv.TaskResult` objects contain properties for the result status, output artifacts, details, and result values for the number of passing, warning, and failing results for task iterations.

`taskResultsOutdated` — Whether task results are outdated or up-to-date
logical | logical array

Status of task results, returned as a logical value or logical array. Values of `1` indicate that the results for the task iteration are outdated and may not reflect the current state of the project or task. Values of `0` indicate that the results for the task iteration are up-to-date. The result is an empty array, `[]`, when there are not task results.

Version History

Introduced in R2022a

processadvisor

Open **Process Advisor** app for Simulink model

Syntax

`processadvisor(modelName)`

Description

`processadvisor(modelName)` opens the Simulink model, `modelName`, in the current project and opens a **Process Advisor** pane to the left of the Simulink canvas.

You need to load a MATLAB project to use the function `processadvisor`.

Examples

Open Process Advisor for Model in Project

Open the **Process Advisor** app for a specific model in a project.

Open the Process Advisor example project, which contains an example model `AHRS_Voter.slx`.

```
processAdvisorExampleStart
```

Open the **Process Advisor** app for the model `AHRS_Voter.slx`.

```
processadvisor("AHRS_Voter")
```

The `AHRS_Voter` model opens in Simulink and the **Process Advisor** app opens in a pane to the left of the Simulink canvas. You can use the **Process Advisor** app to run the tasks in your process.

Input Arguments

`modelName` — Model name

character vector | string

Model name, specified as a character vector or string.

Do not include the model extension (`.slx` or `.mdl`) in the model name.

Example: `"AHRS_Voter"`

Data Types: `char` | `string`

Alternative Functionality

App

You can also open the **Process Advisor** app for a model by using the Apps Gallery.

1. Open a Simulink model in your project.
2. Click the **Apps** tab.
3. In the **Model Verification, Validation, and Test** section, click **Process Advisor**.

Version History

Introduced in R2022a

processAdvisorWindow

Open **Process Advisor** app for project

Syntax

```
processAdvisorWindow()
```

Description

`processAdvisorWindow()` opens the **Process Advisor** app for the current project. The app opens in a standalone window.

Examples

Open Process Advisor app for Project

Open the **Process Advisor** app for a MATLAB project.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Open the **Process Advisor** app for the project.

```
processAdvisorWindow()
```

The standalone **Process Advisor** window shows each of the task iterations in the project, organized by task. In the **Task** column, the table shows each task and the artifacts that the task iterates over. You can double-click on an artifact name to open the artifact. For example, if you double-click on the name of a test case, the test case opens in Test Manager.

Alternative Functionality

App

You can also open the **Process Advisor** app for a project directly from the **Project** tab in MATLAB.

On the **Project** tab, in the **Tools** gallery, click **Process Advisor**.

Version History

Introduced in R2022a

runprocess

Generate and run model-based design (MBD) pipeline using build system

Syntax

```
[buildResult,exitCode] = runprocess()
[buildResult,exitCode] = runprocess(Name=Value)
```

Description

`[buildResult,exitCode] = runprocess()` generate a model-based design (MBD) pipeline and run the pipeline using the build system. The process model, `processmodel.m`, in the project defines the tasks for the pipeline.

`[buildResult,exitCode] = runprocess(Name=Value)` specifies how the MBD pipeline runs using one or more `Name=Value` arguments.

Examples

Run MBD Pipeline

Open a project and use `runprocess` to generate and run the MBD pipeline using the build system.

Open the **Process Advisor** example project, which contains an example process model. The process model defines the tasks for the pipeline.

```
processAdvisorExampleStart
```

Generate and run the MBD pipeline and store the results in the variable `results`.

```
results = runprocess()
```

Run Specific Task Iteration, Clean Task Results, and Delete Task Outputs

Open a project and run one specific task iteration in the pipeline.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Get a list of the task iterations in the MBD pipeline.

```
tasks = generateProcessTasks;
```

Force `runprocess` to run one of the task iterations by specifying `Force` as `true` and `Tasks` as one of the tasks in `tasks`.

```
runprocess(Force=true,Tasks=tasks(1))
```

When `Force` is `true`, `runprocess` runs the pipeline, even if the pipeline already had results that were marked as up-to-date.

Clean task results and delete task outputs.

```
runprocess(Clean=true,DeleteOutputs=true)
```

When you clean task results and delete task outputs, it is as if the tasks were not run.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[buildResult,exitCode] = runprocess(Force=true)`

Tasks — List of task iteration IDs
`{}` (default) | character vector | cell array of character vectors | string | string array

List of task iteration IDs that you want to call `runprocess` on, specified as a character vector, cell array of character vectors, string, or string array. A *task iteration* is the pairing of a task to a specific project artifact. By default, `runprocess` acts on each task iteration in the project.

You can find task iteration IDs by using one of the following approaches:

- Call the function `generateProcessTasks` to create a list of IDs for each task iteration in the pipeline.

  ```
  taskIterationIDs = generateProcessTasks
  ```
- Use the function `createProcessTaskID` to create the ID for a specified task and project artifact. For example, suppose you want the ID for running the built-in task `padv.builtin.task.GenerateSimulinkWebView` on a model, `modelName.slx`, in the folder `modelsFolder` in the project.

  ```
  taskName = "padv.builtin.task.GenerateSimulinkWebView";
  artifactType = "sl_model_file";
  relativePath = "modelsFolder/modelName.slx"
  artifact = padv.Artifact(artifactType,relativePath);
  taskIterationID = createProcessTaskID(taskName, artifact)
  ```

  ```
  taskIterationID = 
  "padv.builtin.task.GenerateSimulinkWebView|sl_model_file|modelsFolder/modelName.slx"
  ```

  IDs take the form: "`taskName|fileType|relativePath`", where `relativePath` is the path relative to the project root.

  Example IDs:

  - `"myCustomProjectTask|project|ProcessAdvisorExample.prj"`
  - `"padv.builtin.task.RunModelStandards|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`
  - `"padv.builtin.task.RunTestsPerTestCase|sl_test_case|ced877ff-cfb8-4fa8-9bbf-aaa29b1d926b"`

Note You can only run task iterations that are already defined in the process model. For each task iteration, the task must be a task that you added to the process model and the artifact must be an artifact that you specified the task runs on.

For example, if the task `myCustomTask` is a task that runs once for each model in the project, you cannot run using the ID `"myCustomTaskTask|project|ProcessAdvisorExample.prj"` until you specify, in the process model, that `myCustomTask` is a task that runs once for the project.

Example: `"padv.builtin.task.GenerateSimulinkWebView|sl_model_file|modelsFolder/modelName.slx"`

Example: `["padv.builtin.task.GenerateSimulinkWebView|sl_model_file|modelsFolder/modelName.slx","padv.builtin.task.GenerateSimulinkWebView|sl_model_file|modelsFolder/otherModel.slx"]`

Data Types: `char`

`Force` — Setting to skip or run up-to-date task iterations
`false` or `0` (default) | `true` or `1`

Setting to skip or run up-to-date tasks, specified as a numeric or logical `0` (`false`) or `1` (`true`). By default, `runprocess` does not run task iterations that have up-to-date results.

Example: `true`

Data Types: `logical`

`Isolation` — Setting to include or ignore task dependencies
`false` or `0` (default) | `true` or `1`

Setting to include or ignore task dependencies, specified as a numeric or logical `0` (`false`) or `1` (`true`). By default, `runprocess` includes task dependencies when running a task. Specify `Isolation` as `true` if you want to run a task in isolation, without running any task dependencies.

Note that you define task dependencies in the process model by using the function `dependsOn`.

Example: `true`

Data Types: `logical`

`Clean` — Clear task results and delete outputs
`false` or `0` (default) | `true` or `1`

Setting to clean task results and outputs, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Note that if you specify `Clean` as `true`, `runprocess` ignores other name-value arguments and cleans the task results and output.

Note If you specify `Clean` as `true`, then you cannot specify `MarkStale` as `true`. The arguments are mutually exclusive.

Example: `true`

Data Types: `logical`

`DeleteOutputs` — Setting to delete task outputs

`false` or `0` (default) | `true` or `1`

Setting to delete task outputs, specified as a numeric or logical `0` (`false`) or `1` (`true`).

Note To delete task outputs with `DeleteOutputs`, you must specify `Clean` as `true`.

Example: `true`

Data Types: `logical`

`MarkStale` — Setting to mark task as outdated

`false` or `0` (default) | `true` or `1`

Setting to mark task as outdated, specified as a numeric or logical `0` (`false`) or `1` (`true`). When you mark a task as stale, the results appear outdated in the **Process Advisor** app.

Note If you specify `MarkStale` as `true`, then you cannot specify `Clean` as `true`. The arguments are mutually exclusive.

Example: `true`

Data Types: `logical`

`ExitInBatchMode` — Setting to exit MATLAB when running in batch mode

`true` or `1` (default) | `false` or `0`

Setting to exit MATLAB when running in batch mode, specified as a numeric or logical `1` (`true`) or `0` (`false`). By default, if you are running MATLAB in batch mode and `runprocess` finishes running, `runprocess` exits MATLAB.

The process exit codes are:

- `0` if the `Status` of `buildResult` is `PASS`
- `1` if the `Status` of `buildResult` is `ERROR`
- `2` if the `Status` of `buildResult` is `FAIL`

For example, suppose you want to run tasks and generate a report in batch mode. You would need to specify `ExitInBatchMode` as `false` and use the `exitCode` returned by `runprocess` to exit:

```
[buildResult, exitCode] = runprocess(ExitInBatchMode=false);
rptObj = padv.ProcessAdvisorReportGenerator();
generateReport(rptObj);
exit(exitCode);
```

Otherwise, the function `runprocess` would automatically exit MATLAB before the report can generate.

Example: `false`

Data Types: `logical`

`RerunFailedTasks` — Setting to ignore or rerun failed task iterations
`false` or `0` (default) | `true` or `1`

Setting to ignore or rerun failed task iterations, specified as a numeric or logical `0` (`false`) or `1` (`true`). `runprocess` considers failed task iterations as outdated and reruns the task iterations.

Example: `true`

Data Types: `logical`

`RerunErroredTasks` — Setting to ignore or rerun errored task iterations
`false` or `0` (default) | `true` or `1`

Setting to ignore or rerun errored task iterations, specified as a numeric or logical `0` (`false`) or `1` (`true`). `runprocess` considers task iterations with errors as outdated and reruns the task iterations.

Example: `true`

Data Types: `logical`

`RefreshProcessModel` — Setting to automatically refresh before running tasks
`true` or `1` (default) | `false` or `0`

Setting to automatically refresh before running tasks, specified as a numeric or logical `1` (`true`) or `0` (`false`). By default, `runprocess` refreshes before running tasks so that the run uses the current state of the process model and project. If you specify `RefreshProcessModel` as `false`, `runprocess` does not refresh before running, but the run may not include the latest changes to tasks in the process model or artifacts in the project.

Example: `false`

Data Types: `logical`

`ReanalyzeProjectAnalysisIssues` — Automatically reanalyze project analysis issues that have severity level of error
`true` or `1` (default) | `false` or `0`

Automatically reanalyze project analysis issues that have a severity level of error, specified as a numeric or logical `1` (`true`) or `0` (`false`).

If you are using R2022b Update 1 or later, you can specify `ReanalyzeProjectAnalysisIssues` as `false` to prevent the build system from reanalyzing project analysis issues that have a severity level of error. This may reduce the execution time for `runprocess`, but the build system may not generate the expected task iterations or detect outdated results.

Fix the issues listed in the **Project Analysis Issues** pane of the **Process Advisor** app to make sure the build system can fully analyze the project, generate the expected task iterations, and detect outdated results.

Example: `false`

Data Types: `logical`

Output Arguments

`buildResult` — Results of run
`padv.BuildResult`

Results of run, returned as a `padv.BuildResult` object.

The `padv.BuildResult` object includes:

- The start time and end time of the run
- The status of the run (`Pass`,`Error`,`Fail`)
- Lists of the tasks that the passed, errored, were skipped, or failed during the run
- Input arguments to the run

`exitCode` — Exit code from run
`0` | `1` | `2`

Exit code from run, returned as a `double` representing the process error code.

- `0` if the `Status` of `buildResult` is `Pass`
- `1` if the `Status` of `buildResult` is `Error`
- `2` if the `Status` of `buildResult` is `Fail`

Alternative Functionality

App

You can also use the **Process Advisor** app to run each task or individual task iterations in the process. To open the **Process Advisor** app for a project, in the MATLAB Command Window, enter:

```
processAdvisorWindow
```

Version History

Introduced in R2022a

Classes — Alphabetical List

The API includes the following classes:

Class	Object Functions	Description
`padv.Artifact`	None	Store artifact information
`padv.BuildResult`	None	Result from build system build
`padv.Preferences`	None	Set `runprocess` function settings
`padv.ProcessModel`	• `reset` • `reload` • `addTask` • `addQuery` • `findQuery` • `findTask` • `exists`	Define tasks and process for project
`padv.Query`	• `run`	Select set of artifacts from project
`padv.Task`	• `run` • `dependsOn` • `runsAfter` • `addInputQueries`	Single step in process
`padv.TaskResult`	• `pass` • `fail` • `error` • `applyStatus`	Create and access results from task

The class reference pages are listed alphabetically on the following pages.

Tip You can also access API help from the MATLAB Command Window by using `help` function.

For example, this code returns help information for the class `padv.Task`:

```
help padv.Task
```

padv.Artifact

Store artifact information

Description

A `padv.Artifact` object represents an artifact that you can run a task on in the process that you define in your process model. You can use a `padv.Artifact` to specify a specific project artifact that you want a task to run on. Use a `padv.Artifact` object as an input to the function `createProcessTaskID` when you want to get the ID for a specific task iteration. A task iteration is the pairing of a task to a specific project artifact.

Creation

Syntax

Description

`artifactObject = padv.Artifact(artifactType,relativePath)` stores artifact information in a `padv.Artifact` object, `artifactObject`. You can use the artifact information when you want to get the ID for a specific task iteration.

`artifactObject = padv.Artifact( ___ ,Name=Value)` specifies the artifact using one or more `Name=Value` arguments.

Input Arguments

`artifactType` — Type of artifact
string

Type of artifact, specified as a string.

Valid artifact types include:

- `"sl_model_file"` — Simulink model file
- `"sl_test_case"` — Simulink Test test case
- `"project"` — MATLAB project

Example: `"project"`

Data Types: `string`

`relativePath` — Address of artifact
string

Address of artifact, specified as a string. The address of the artifact is the path to the artifact, relative to the project root.

Example: `"02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`

Data Types: `string`

Properties

`Type` — Type of artifact
string

Type of artifact, specified as a string.

Valid artifact types include:

- `"sl_model_file"` — Simulink model file
- `"sl_test_case"` — Simulink Test test case
- `"project"` — MATLAB project

Example: `"project"`

Data Types: `string`

`Parent` — Reference to parent artifact
empty `padv.Artifact` object (default) | `padv.Artifact` object

Reference to parent artifact, specified as a `padv.Artifact` object.

`Address` — Address of artifact
string

Address of artifact, specified as a string. The address of the artifact is the path to the artifact, relative to the project root.

Example: `"02_Models/AHRS_Voter/specification/AHRS_Voter.slx"`

Data Types: `string`

`UUID` — Universal unique identifier
empty string (default) | string

Universal unique identifier, specified as a string.

`Label` — Human-readable name for artifact
empty string (default) | string

Human-readable name for the artifact, specified as a string.

`StorageAddress` — Address for sub-file artifact
empty string (default) | string

Address for a sub-file artifact, specified as a string.

Examples

Run One Task on One Artifact

Suppose you have a process model with several tasks, but right now you only want to run the task `padv.builtin.task.RunModelStandards` on the model `AHRS_Voter.slx`. Use the function `createProcessTaskID` to get the ID for a specific task iteration, then use the function `runprocess` to run only that specific task iteration.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Use `padv.Artifact` to specify the project artifact that you want the task to run on. For this example, the artifact type is `sl_model_file` because the artifact is a Simulink model and the address is the path to `AHRS_Voter.slx`, relative to the project root.

```
artifactType = "sl_model_file";
address = "02_Models/AHRS_Voter/specification/AHRS_Voter.slx";
artifact = padv.Artifact(artifactType,address);
```

You can use the `padv.Artifact` object, `artifact`, as an input to functions like:

- `generateProcessTasks` — Find the IDs for each task iteration associated with an artifact

  ```
  IDs_AHRS_Voter = generateProcessTasks(FilterArtifact=artifact);
  ```
- `createProcessTaskID` — Find the ID for a specific task iteration

  ```
  task = padv.builtin.task.RunModelStandards;
  runModelStandards_for_AHRS_Voter = createProcessTaskID(task,artifact)
  ```

You can then use the function `runprocess` to run the task iterations.

- `runprocess(Tasks=IDs_AHRS_Voter)`
- `runprocess(Tasks = runModelStandards_for_AHRS_Voter)`

Version History

Introduced in R2022a

padv.BuildResult

Result from build system build

Description

Use the build result, `padv.BuildResult`, to find the properties of the build system build, including a list of the tasks that the build system ran and the settings the build system used.

Creation

Syntax

Description

`buildResultObj = padv.BuildResult()` stores the results from a build system build.

Properties

`StartTime` — Start time of build
`[0×0 datetime]` (default) | `datetime`

Start time of build, returned as `datetime`.

Example: `09-Aug-2022 14:32:05`

Data Types: `datetime`

`EndTime` — End time of build
`[0×0 datetime]` (default) | `datetime`

End time of build, returned as `datetime`.

Example: `09-Aug-2022 14:32:37`

Data Types: `datetime`

`Status` — Overall status for build
`Pass` (default) | `Fail` | `Error`

Overall status for build, returned as the `padv.TaskStatus` enumeration value:

- `Error` if any task iteration in the build returns an error.
- `Fail` if no task iterations in the build return an error, but at least one task iteration fails.
- `Pass` if no task iterations were run, or if no task iterations in the build return an error or fail.

Example: `Pass`

`ResultValues` — Task iteration result values
`[1×1 struct]` (default) |

Task iteration result values, returned as a structure array with fields:

- `Pass`
- `Warn`
- `Fail`

For example, if the build runs one task iteration and the task iteration returns one passing result and five warning results, the structure array contains:

```
  struct with fields:

    Pass: 1
    Warn: 5
    Fail: 0
```

Data Types: `struct`

`PassTasks` — IDs for task iterations that passed during the build
[ ] (default) | cell array

IDs for task iterations that passed during the build, returned as a cell array.

If the build system runs one task iteration and the task iteration passes, `PassTasks` returns a one-dimensional cell array. For example, if the build system only ran the task `padv.builtin.task.GenerateCodeAsRefModel` on the model `AHRS_Voter.slx` and the task iteration passed, `PassTasks` returns:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
```

If multiple task iterations pass, `PassTasks` returns one cell for each task iteration that passed. For example:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Actuator_Control/specificatio
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Flight_Control/specification/
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/InnerLoop_Control/specificati
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/OuterLoop_Control/specificati
```

Data Types: `cell`

`ErrorTasks` — IDs for task iterations that returned an error during the build
[ ] (default) | cell array

IDs for task iterations that returned an error during the build, returned as a cell array.

If the build system runs one task iteration and the task iteration returns an error, `ErrorTasks` returns a one-dimensional cell array. For example, if the build system tried to run a custom task, `customTask`, on the model `AHRS_Voter.slx`, but the task iteration returned an error, `ErrorTasks` returns:

```
{'customTask|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'}
```

If multiple task iterations error, `ErrorTasks` returns one cell for each task iteration that returned an error. For example:

```
{'customTask|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_Voter.slx'              }
{'customTask|sl_model_file|02_Models/Actuator_Control/specification/Actuator_Control.slx'  }
```

```
{'customTask|sl_model_file|02_Models/Flight_Control/specification/Flight_Control.slx'      }
{'customTask|sl_model_file|02_Models/InnerLoop_Control/specification/InnerLoop_Control.slx'}
{'customTask|sl_model_file|02_Models/OuterLoop_Control/specification/OuterLoop_Control.slx'}
```

Data Types: `cell`

`SkippedTasks` — IDs for task iterations that the build system skipped
[ ] (default) | cell array

IDs for task iterations that the build system skipped, returned as a cell array. The build system skips task iterations that already have up-to-date results, unless you specify `Force` as `true` when you call the function `runprocess`.

If the build system skips one task iteration, `SkippedTasks` returns a one-dimensional cell array. For example, if you instructed the build system to run the task `padv.builtin.task.GenerateCodeAsRefModel` on the model `AHRS_Voter.slx`, but the task iteration already had up-to-date results, `SkippedTasks` returns:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
```

If the build system skips multiple task iterations, `SkippedTasks` returns one cell for each task iteration that the build system skipped. For example:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Actuator_Control/specification
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Flight_Control/specification/
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/InnerLoop_Control/specificatio
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/OuterLoop_Control/specificatio
```

Data Types: `cell`

`FailedTasks` — IDs for task iterations that failed during the build
[ ] (default) | cell array

IDs for task iterations that failed during the build, returned as a cell array.

If the build system runs only one task iteration and the task iteration fails, `FailedTasks` returns a one-dimensional cell array. For example, if the build system ran the task `padv.builtin.task.GenerateCodeAsRefModel` on the model `AHRS_Voter.slx` and the task iteration failed, `FailedTasks` returns:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
```

If multiple task iterations fail, `FailedTasks` returns one cell for each task iteration that failed. For example:

```
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/AHRS_Voter/specification/AHRS_
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Actuator_Control/specification
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/Flight_Control/specification/
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/InnerLoop_Control/specificatio
{'padv.builtin.task.GenerateCodeAsRefModel|sl_model_file|02_Models/OuterLoop_Control/specificatio
```

Data Types: `cell`

`InputArgs` — Input arguments that defined how the build system ran the build
`[1×1 struct]` (default) | structure array

Input arguments that defined how the build system ran the build, returned as a structure array with fields:

- `TasksToBuild` — List of task iteration IDs that you want the build system to run
- `Isolation` — Setting to include or ignore task dependencies
- `Force` — Setting to skip or run up-to-date task iterations
- `RerunFailedTasks` — Setting to ignore or rerun failed task iterations
- `RerunErroredTasks` — Setting to ignore or rerun task iterations that returned an error

For example, the `InputArgs` for a build result could return:

```
  struct with fields:

         TasksToBuild: [1×5 string]
            Isolation: 0
                Force: 0
     RerunFailedTasks: 0
    RerunErroredTasks: 0
```

For more information, see the function `runprocess`.

Data Types: `struct`

Examples

Get List of Passed Task Iterations and Build Settings

Open a project, run a build, and use the build result, `padv.BuildResult`, to get a list of the passed task iterations and the settings that the build system used when running the build.

Open the **Process Advisor** example project, which contains an example process model.

```
processAdvisorExampleStart
```

Generate a list of the tasks defined by the process model.

```
tasks = generateProcessTasks;
```

Run the first five task iterations in `tasks` and specify `Force` as `true`.

```
buildResult = runprocess(Force=true,Tasks=tasks(1:5))
```

Use the build result, `buildResult`, to get a list of the task iterations that passed.

```
passed = buildResult.PassTasks'
```

```
passed =

  5×1 cell array

    {'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_Voter/specification/
    {'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/Actuator_Control/specific
    {'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/Flight_Control/specificat
    {'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/InnerLoop_Control/specifi
    {'padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/OuterLoop_Control/specifi
```

When you used the function `runprocess`, you specified `Force` as `true`. You can see that information in the `InputArgs` property of the build result, `buildResult`.

```
runprocessInputs = buildResult.InputArgs
```

```
runprocessInputs = 

  struct with fields:

         TasksToBuild: ["padv.builtin.task.GenerateSimulinkWebView|sl_model_file|02_Models/AHRS_V
            Isolation: 0
                Force: 1
     RerunFailedTasks: 0
    RerunErroredTasks: 0
```

The build result shows that the `Force` setting was `1` (`true`) when the build system ran.

Version History

Introduced in R2022a

padv.Preferences

Set `runprocess` function settings

Description

Use the preferences, `padv.Preferences`, to specify how the function `runprocess` runs. To specify the preferences for a specific project, create a startup script for the project and specify the property values for the global preferences object.

Creation

Syntax

Description

`P = padv.Preferences()` gets the handle to the global preferences object, `P`. There is only one set of preference properties.

Preferences are not persistent. If you restart MATLAB or call `clear classes`, the preference properties reset to the default values.

Properties

`GarbageCollectTaskOutputs` — Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project
`true` or `1` (default) | `false` or `0`

Setting for automatically cleaning task results for tasks and artifacts that do not match current process model or project, specified as a numeric or logical `1` (`true`) or `0` (`false`).

By default, when you use the build system, the build system cleans task results that are no longer relevant for the current process model or project. For example, if you had task results from a specific task and then you remove that task from the process model, the build system automatically deletes the task results associated with the task. If you had task results associated with a specific project artifact and then you removed that artifact from the project, the build system automatically deletes the task results associated with the artifact. Note that the build system does not delete generated artifacts like generated code.

If you specify `GarbageCollectTaskOutputs` as `false`, the build system does not automatically clean task results associated with tasks and artifacts that are not in the current process model or project.

Example: `false`

Data Types: `logical`

`ShowDetailedErrorMessages` — Setting to show more information in error messages
`false` or `0` (default) | `true` or `1`

Setting to show more information in error messages, specified as a numeric or logical `0` (`false`) or `1` (`true`).

By default, error messages from the build system are not verbose.

If you specify `ShowDetailedErrorMessages` as `true`, the build system shows full stack traces in error messages. You may want to see full stack traces when you are debugging a process model.

Example: `true`

Data Types: `logical`

`TrackProcessModel` — Setting for tracking changes to process model
`true` or `1` (default) | `false` or `0`

Setting for tracking changes to process model, specified as a numeric or logical `1` (`true`) or `0` (`false`).

By default, if you make a change to the process model file, `processmodel.m`, the build system marks each task status and task result as outdated because the tasks in the updated process model might not match the tasks that generated the task results from the previous version of the process model. For example, if you ran the built-in task `padv.builtin.task.RunModelStandards` with the default Model Advisor configuration, updated the process model to specify a different Model Advisor configuration file for the task, and then ran the task again, the task results are now outdated because they are the task results from the default configuration.

If you specify `TrackProcessModel` as `false` and make a change to the process model, the build system will not mark the task statuses and task results as outdated.

Example: `false`

Data Types: `logical`

Examples

Specify Preferences for Builds

Use `padv.Preferences` to specify preferences for the **Process Advisor** app and build system.

Create a `padv.Preferences` object.

```
PREF = padv.Preferences
```

```
PREF = 

  Preferences with properties:

    GarbageCollectTaskOutputs: 1
    ShowDetailedErrorMessages: 0
            TrackProcessModel: 1
             IncrementalBuild: 1
```

Specify `IncrementalBuild` as `0`.

```
PREF.IncrementalBuild = 0;
```

Now, when you run tasks in the current MATLAB session, incremental builds are disabled and the build system forces tasks to run, even if the tasks have up to date results.

Version History

Introduced in R2022a

padv.ProcessModel

Define tasks and process for project

Description

A `padv.ProcessModel` object represents the process model that defines the tasks and process for a project. A *task* performs an action and is a single step in your process. A *process* is a series of tasks that run in a specific order. The process model defines the tasks that you can perform on the project, and the order and relationships between tasks in the process. You can use tasks and queries to dynamically perform actions and find artifacts in the project. Use the `addTask` object function to add tasks to the process model. You can use the function `runprocess` to run the tasks defined in the process model.

Creation

Syntax

```
pm = padv.ProcessModel()
```

Description

`pm = padv.ProcessModel()` creates an empty process model object, `pm`.

Properties

`TaskNames` — Tasks added to process model object
`[1×0 string]` (default) | string array

Tasks added to process model object, returned as string array.

Use the object function `addTask` to add a task instance to a process model.

Example: `["padv.builtin.task.GenerateSimulinkWebView" "padv.builtin.task.RunModelStandards"]`

Data Types: `string`

`QueryNames` — Queries added to process model object
`[1×0 string]` (default) | string array

Queries added to process model object, returned as string array.

Use the object function `addQuery` to add a query instance to a process model.

Example: `["padv.builtin.query.FindModels" "padv.builtin.query.FindProjectFile"]`

Data Types: `string`

`DefaultQueryName` — Default query for tasks added to process model object
`"padv.builtin.query.FindProjectFile"` (default) | name of `padv.Query` query

Default query for tasks added to process model, specified as the name of a `padv.Query` query.

Example: `"padv.builtin.query.FindModels"`

Data Types: `string`

`RootFileName` — Name of process model file
`"processmodel.m"` (default) | string

Name of process model file, specified as a string.

Data Types: `string`

Object Functions

`reset`	Removes tasks and queries from process model
	
	`pm = padv.ProcessModel();`
	`reset(pm);`

`reload`	Load process model by executing `processmodel.m` file for project
	
	`pm = padv.ProcessModel();`
	`reload(pm);`

`addTask`	Add task instance to process model For information, see "padv.ProcessModel.addTask".
`addQuery`	Add query instance to process model For information, see "padv.ProcessModel.addQuery".

`findQuery`	Find query instance by name
	
	`pm = padv.ProcessModel();`
	`QUERY = findQuery(pm,...`
	`"padv.builtin.query.FindModels")`

`findTask`	Find task instance by name
	
	`pm = padv.ProcessModel();`
	`TASK = findTask(pm,...`
	`"padv.builtin.task.RunModelStandards");`

`exists`	Check if process model exists for project `[FOUND, PATH] = padv.ProcessModel.exists()`

Examples

Add Tasks to Process Model Object

You can use the object function `addTask` to add the tasks to a `padv.ProcessModel` object.

Open the **Process Advisor** example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the **Process Advisor** pane to the left of the Simulink canvas.

In the **Process Advisor** pane, click the **Edit process model** button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    addTask(pm,"taskA");
    addTask(pm,"taskB");

end
```

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Use the function `getprocess` to get the process model object for the project.

```
pm = getprocess;
```

Get the task object for `"taskA"` defined in the current process model.

```
taskAObj = findTask(pm, "taskA");
```

`taskAObj` is a `padv.Task` object that you can use to view the properties of the task `"taskA"`.

Version History

Introduced in R2022a

padv.ProcessModel.addQuery

Package: `padv`

Add query instance to process model

Syntax

```
queryObj = addQuery(pm,queryNameOrInstance)
queryObj = addTask( ___ ,Name=Value)
```

Description

`queryObj = addQuery(pm,queryNameOrInstance)` adds the query specified by `queryNameOrInstance` to the process model. You can access the query using the query object `queryObj`.

`queryObj = addTask( ___ ,Name=Value)` specifies the properties of the query using one or more `Name=Value` arguments.

Examples

Input Arguments

pm — Process for project
`padv.ProcessModel` object (default) |

Process for project, specified as a `padv.ProcessModel` object.

Example: `pm = padv.ProcessModel`

`queryNameOrInstance` — Name or instance of query
string | `padv.Query` object

Name or instance of a query, specified as a string or `padv.Query` object.

Example: `"NameOfMyQuery"`

Example: `padv.builtin.query.FindModels`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

`DefaultArtifactType` — Artifact type returned by query
`"padv_output_file"` (default) | valid value for the `Type` property of a `padv.Artifact` object

Artifact type returned by the query, specified as a valid value for the `Type` property of a `padv.Artifact` object.

Example: `DefaultArtifactType = "sl_model_file"`

`Title` — Human readable name
`Name` property of query (default) | string

Human readable name for the query, specified as a string. By default, the `Title` property of the query is the same as the `Name`.

Example: `"My Query"`

Data Types: `string`

`FunctionHandle` — Handle to function that runs when you run query object
`[]` (default) | `function_handle`

Handle to function that runs when you run query object, specified as a `function_handle`.

When you call the `run` function on a query object, `run` runs the function specified by the `function_handle`.

Example: `FunctionHandle = @FunctionForQuery`

Data Types: `function_handle`

`Parent` — Initial query run before iteration query
`[0×0 string]` (default) | `padv.Query` object | `Name` of `padv.Query` object

Initial query run before iteration query, specified as either a `padv.Query` object or the `Name` of a `padv.Query` object. When you specify a `padv.Query` object as the iteration query for a task, the `Parent` query is the initial query that the build system runs before running the specified iteration query.

For example, the built-in query`padv.builtin.query.FindModelsWithTestCases` has the `Parent` query `padv.builtin.query.FindModels`. If you specify `padv.builtin.query.FindModelsWithTestCases` as the iteration query for a task, you are specifying that you want the task to run once for each model with a test case. The build system runs the `Parent` query `padv.builtin.query.FindModels` first, to find the models in the project, and then the build system runs the iteration query `padv.builtin.query.FindModelsWithTestCases` to find the models with test cases.

The build system ignores the `Parent` query when you specify a query as an input query or dependency query for a task.

Example: `"padv.builtin.query.FindModels"`

`SortArtifacts` — Setting for automatically sorting artifacts by address
`true` or `1` (default) | `false` or `0`

Setting for automatically sorting artifacts by address, specified as a numeric or logical `1` (`true`) or `0` (`false`). When a query returns artifacts, the artifacts should be in a consistent order. By default, the build system sorts artifacts by the artifact address.

Alternatively, you can sort artifacts in a different order by overriding the internal `sortArtifacts` method in a subclass that defines a custom sort behavior. The build system automatically calls the `sortArtifacts` method when using the process model. The `sortArtifacts` method expects two input arguments: a `padv.Query` object and a list of `padv.Artifact` objects returned by the `run` function. The `sortArtifacts` method should return a list of sorted `padv.Artifact` objects.

Example: `SortArtifacts = false`

Data Types: `logical`

Output Arguments

`queryObj` — Query object
`padv.Query` object

Query object, returned as a `padv.Query` object.

For more information, see the documentation for "padv.Query" in the chapter "Classes — Alphabetical List".

Version History

Introduced in R2022a

padv.ProcessModel.addTask

Package: padv

Add task instance to process model

Syntax

```
taskObj = addTask(pm,taskNameOrInstance)
taskObj = addTask(___,Name=Value)
```

Description

`taskObj = addTask(pm,taskNameOrInstance)` adds the task specified by `taskNameOrInstance` to the process model. You can access the task using the task object `taskObj`.

`taskObj = addTask(___,Name=Value)` specifies the properties of the task using one or more `Name=Value` arguments.

Examples

Add Tasks to Process Model

You can use the `addTask` function to create function-based tasks directly in the process model.

Open the **Process Advisor** example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the **Process Advisor** pane to the left of the Simulink canvas.

In the **Process Advisor** pane, click the **Edit process model** button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    addTask(pm,"MyCustomTask",Action=@SayHello,...
        IterationQuery=padv.builtin.query.FindModels);

end

function results = SayHello(~)
    disp("Hello, World!");
    results = padv.TaskResult;
    results.ResultValues.Pass = 1;
end
```

This code adds a task, `MyCustomTask` to the process model while specifying that the task runs the function `SayHello` one time for each model found in the project. The function `SayHello` also specifies the results returned by the task.

Input Arguments

`pm` — Process for project

`padv.ProcessModel` object (default)

Process for project, specified as a `padv.ProcessModel` object.

Example: `pm = padv.ProcessModel`

`taskNameOrInstance` — Name or instance of task

string | `padv.Task` object

Name or instance of a task, specified as a string or `padv.Task` object.

Example: `"NameOfMyTask"`

Example: `padv.builtin.task.RunModelStandards`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:
`addTask(pm,"RunOnceForEachModel",IterationQuery=padv.builtin.query.FindModels)`

`Title` — Human readable name that appears in Process Advisor app

`Name` property of task (default) | string

Human readable name that appears in the **Tasks** column of the **Process Advisor** app, specified as a string. By default, the **Process Advisor** app uses the `Name` property of the task as the `Title`.

Example: `"My Task"`

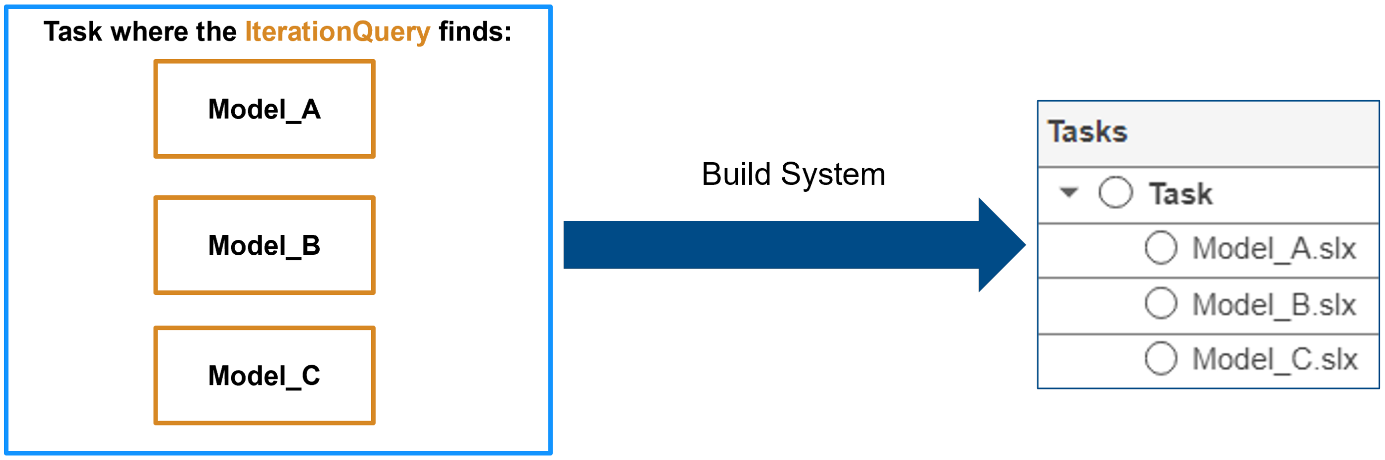
Data Types: `string`

`IterationQuery` — Artifacts that task iterates over

`[1×1 padv.internal.QueryReference]` (default) | `padv.Query` object | name of `padv.Query` object

Artifacts that task iterates over, specified as a `padv.Query` object or the name of a `padv.Query` object. By default, task objects run one time and are associated with the project. When you specify `IterationQuery`, the task runs one time *for each* artifact specified by the `padv.Query`. In the **Process Advisor** app, the artifacts specified by `IterationQuery` appear under task title.

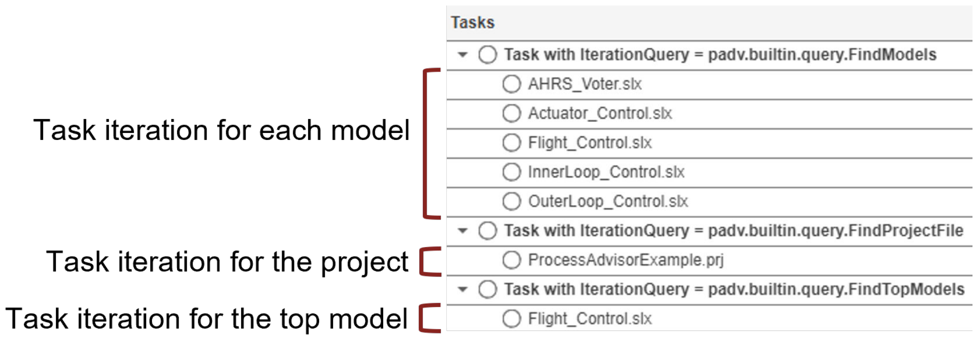
For example, if the `IterationQuery` for a task finds three models, `Model_A`, `Model_B`, and `Model_C`, the build system creates three task iterations under the title of the task in the **Tasks** column.



Each of the artifacts under the task title represents a *task iteration*.

For an example of the effect of different `IterationQuery` values:

- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindModels` to find each of the models in the project, the build system creates a task iteration for each model.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindProjectFile` to find the project file, the build system creates a task iteration for the project file.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindTopModels` to find top models in the project, the build system creates a task iteration for each top model.


Task iteration for each model
Task iteration for the project
Task iteration for the top model
Tasks
Task with IterationQuery = padv.builtin.query.FindModels
AHRS_Voter.slx
Actuator_Control.slx
Flight_Control.slx
InnerLoop_Control.slx
OuterLoop_Control.slx
Task with IterationQuery = padv.builtin.query.FindProjectFile
ProcessAdvisorExample.prj
Task with IterationQuery = padv.builtin.query.FindTopModels
Flight_Control.slx

Example: `IterationQuery = padv.builtin.query.FindModels`

Data Types: `string`

`InputQueries` — Inputs to task
empty array of `padv.Query` objects (default) | `padv.Query` object | name of `padv.Query` object | array of `padv.Query` objects

Inputs to the task, specified as:

- a `padv.Query` object
- the name of `padv.Query` object
- an array of `padv.Query` objects
- an array of names of `padv.Query` objects

By default, the task does not specify any artifacts as inputs. When you specify `InputQueries`, the task tasks the artifacts specified by the specified query or queries as an input.

Suppose a task runs once for each model in the project and you want to provide the models as inputs to the task. If you specify `InputQueries` as the built-in query `padv.builtin.query.GetIterationArtifact`, the query returns each artifact that the tasks iterates over, which in this example is each of the models in the project.

Example: `InputQueries = padv.builtin.query.GetIterationArtifact`

`InputDependencyQuery` — Artifact dependencies for task inputs
`[1×1 padv.internal.QueryReference]` (default) | `padv.Query` object | name of `padv.Query` object

Artifact dependencies for task inputs, specified as a `padv.Query` object or the name of a `padv.Query` object.

`Action` — Function that task runs
`[]` (default) | function handle

Function that the task runs, specified as the function handle. When you run the task, the task runs the function specified by the function handle.

For example, if you want the task to run the function `myFunction`, specify `Action` as `@myFunction`.

Example: `Action = @myFunction`

Data Types: `function_handle`

`RequiredIterationArtifactType` — Artifact type that task can run on
`""` (default) | string

Artifact type that the task can run on, specified by a string. The required iteration artifact type must be the artifact type supported by the `IterationQuery` property of the task.

Example: `RequiredIterationArtifactType = "sl_model_file"`

Data Types: `string`

`Licenses` — List of licenses that task requires
empty string (default) | string array

List of licenses that the task requires, specified as a string array.

Example: `Licenses = ["matlab_report_gen" "simulink_report_gen"]`

Data Types: `string`

`AllLicenseRequired` — Setting to require all licenses for task
`true` or `1` (default) | `false` or `0`

Setting to require all licenses for task, specified as a numeric or logical `1` (`true`) or `0` (`false`). By default, all licenses in the `Licenses` property of the task are required for the task to run. Specify `0` (`false`) if the task can run without all licenses listed in the `Licenses` property.

Example: `Licenses = ["matlab_report_gen" "simulink_report_gen"]`

Data Types: `logical`

`DescriptionText` — Task description
empty string (default) | string

Task description, specified as a string.

Example: "This task runs myScript."

Data Types: `string`

`DescriptionCSH` — Path to task documentation
empty string (default) | string

Path to task documentation, specified as a string.

Example: `DescriptionCSH = fullfile(pwd,"taskHelpFiles","myTaskDocumentation.pdf")`

Data Types: `string`

Output Arguments

`taskObj` — Task object
`padv.Task` object

Task object, returned as a `padv.Task` object.

For more information, see the documentation for "padv.Task" in the chapter "Classes — Alphabetical List".

Version History

Introduced in R2022a

padv.Query

Select set of artifacts from project

Description

A `padv.Query` object represents a query that you can use to select a set of artifacts from a project. Use the input arguments to define the set of artifacts that the query selects. Queries can either be function-based or class-based. Use `FunctionHandle` to specify a function for a function-based query or use inheritance for a class-based query.

Creation

Syntax

```
Q = padv.Query(Name)
Q = padv.Query(Name,Name,Value)
```

Description

`Q = padv.Query(Name)` creates a query object with the name `Name`.

`Q = padv.Query(Name,Name,Value)` specifies query properties using one or more name-value arguments. For example, `DefaultArtifactType = "sl_model_file"` changes the default artifact type for the query from a generic output file, `"padv_output_file"`, to a model file, `"sl_model_file"`.

Input Arguments

`Name` — Unique identifier for query
character vector | string

Unique identifier for query, specified as character vector or string. You can only specify a query name when you create a query object. You cannot change the query name after you create the query object.

Each query in the process model must have a unique name.

Example: `"CustomQueryForArtifacts"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `DefaultArtifactType = "sl_model_file"`

`Title` — Human-readable name for query
`Name` (default) | character vector | string

Human-readable name for query, specified as character vector or string.

Example: `Title = "Custom Query for Artifacts"`

Data Types: `char` | `string`

`DefaultArtifactType` — Expected artifact type
`"padv_output_file"` (default) | valid value for the `Type` property of a `padv.Artifact` object

Expected artifact type, specified as a valid value for the `Type` property of a `padv.Artifact` object. `padv.Task` objects use the `DefaultArtifactType` to confirm that the artifacts output by the query are the types of artifacts required by the `padv.Task` object.

When you use the `run` function on a query object, the `DefaultArtifactType` is the default value for artifacts returned by the function.

Example: `DefaultArtifactType = "sl_model_file"`

`Parent` — Initial query run before iteration query
`[0×0 string]` (default) | `padv.Query` object | `Name` of `padv.Query` object

Initial query run before iteration query, specified as either a `padv.Query` object or the `Name` of a `padv.Query` object. When you specify a `padv.Query` object as the iteration query for a task, the `Parent` query is the initial query that the build system runs before running the specified iteration query.

For example, the built-in query`padv.builtin.query.FindModelsWithTestCases` has the `Parent` query `padv.builtin.query.FindModels`. If you specify `padv.builtin.query.FindModelsWithTestCases` as the iteration query for a task, you are specifying that you want the task to run once for each model with a test case. The build system runs the `Parent` query `padv.builtin.query.FindModels` first, to find the models in the project, and then the build system runs the iteration query `padv.builtin.query.FindModelsWithTestCases` to find the models with test cases.

The build system ignores the `Parent` query when you specify a query as an input query or dependency query for a task.

Example: `"padv.builtin.query.FindModels"`

`SortArtifacts` — Setting for automatically sorting artifacts by address
`true` or `1` (default) | `false` or `0`

Setting for automatically sorting artifacts by address, specified as a numeric or logical `1` (`true`) or `0` (`false`). When a query returns artifacts, the artifacts should be in a consistent order. By default, the build system sorts artifacts by the artifact address.

Alternatively, you can sort artifacts in a different order by overriding the internal `sortArtifacts` method in a subclass that defines a custom sort behavior. The build system automatically calls the `sortArtifacts` method when using the process model. The `sortArtifacts` method expects two input arguments: a `padv.Query` object and a list of `padv.Artifact` objects returned by the `run` function. The `sortArtifacts` method should return a list of sorted `padv.Artifact` objects.

Example: `SortArtifacts = false`

Data Types: `logical`

`FunctionHandle` — Handle to function that runs when you run query object
`[]` (default) | `function_handle`

Handle to function that runs when you run query object, specified as a `function_handle`.

When you call the `run` function on a query object, `run` runs the function specified by the `function_handle`.

Example: `FunctionHandle = @FunctionForQuery`

Data Types: `function_handle`

Version History

Introduced in R2022a

padv.Task

Single step in process

Description

A `padv.Task` object represents a single step in a `padv.ProcessModel` process. For example, a `padv.Task` object could represent a step like checking modeling standards, running tests, generating code, or performing a custom action. `padv.Task` objects can accept project artifacts as inputs, perform actions, generate assessments, and return project artifacts as outputs. In your process model, use the object functions `addInputQueries`, `dependsOn`, and `runsAfter` to specify the inputs, dependencies, and desired execution order for a task. You can execute tasks as part of a pipeline. Use the `runprocess` function to generate and run a pipeline of tasks.

Creation

Syntax

```
taskObject = padv.Task(Name)
taskObject = padv.Task( ___ ,Name=Value)
```

Description

`taskObject = padv.Task(Name)` represents a task, named `Name`, in a `padv.ProcessModel` process. Each task object in a process must have a unique `Name`.

`taskObject = padv.Task( ___ ,Name=Value)` sets properties using one or more name-value arguments. For example, `padv.Task("myTask",IterationQuery=padv.builtin.query.FindModels)` creates a task object named `myTask` that runs once for each model.

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Properties

Name — Unique identifier for task in process
string

Unique identifier for task in process, returned as a string. When you specify the `Name`, you specify the `Name` property of the task object.

Each task in the process model must have a unique `Name`. After you specify a `Name` for a `padv.Task` object, you cannot change the `Name`.

Example: `"myTask"`

Data Types: `string`

`Title` — Human readable name that appears in Process Advisor app
`Name` property of task (default) | string

Human readable name that appears in the **Tasks** column of the **Process Advisor** app, returned as a string. By default, the **Process Advisor** app uses the `Name` property of the task as the `Title`.

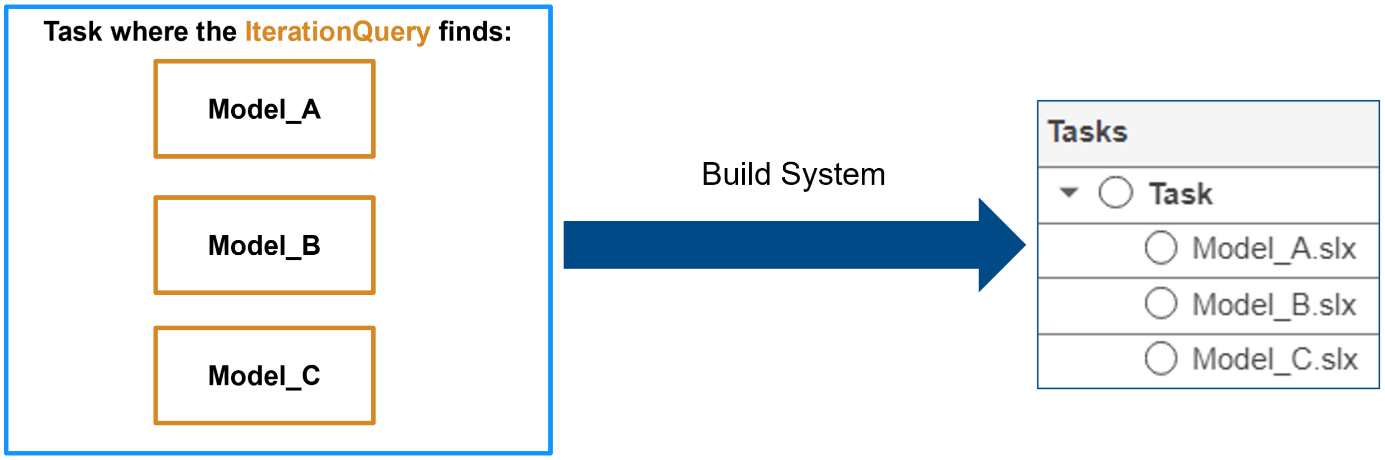
Example: `"My Task"`

Data Types: `string`

`IterationQuery` — Artifacts that task iterates over
`[1×1 padv.internal.QueryReference]` (default) | `padv.Query` object | name of `padv.Query` object

Artifacts that task iterates over, returned as a `padv.Query` object or the name of a `padv.Query` object. By default, task objects run one time and are associated with the project. When you specify `IterationQuery`, the task runs one time *for each* artifact returned by the `padv.Query`. In the **Process Advisor** app, the artifacts returned by `IterationQuery` appear under task title.

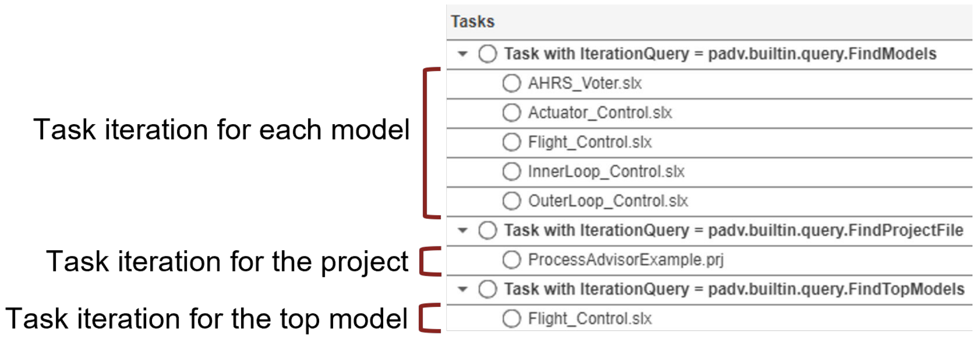
For example, if the `IterationQuery` for a task finds three models, `Model_A`, `Model_B`, and `Model_C`, the build system creates three task iterations under the title of the task in the **Tasks** column.



Each of the artifacts under the task title represents a *task iteration*.

For an example of the effect of different `IterationQuery` values:

- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindModels` to find each of the models in the project, the build system creates a task iteration for each model.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindProjectFile` to find the project file, the build system creates a task iteration for the project file.
- If you have a task where the `IterationQuery` uses `padv.builtin.query.FindTopModels` to find top models in the project, the build system creates a task iteration for each top model.


Task iteration for each model
Task iteration for the project
Task iteration for the top model

Tasks
Task with IterationQuery = padv.builtin.query.FindModels
AHRS_Voter.slx
Actuator_Control.slx
Flight_Control.slx
InnerLoop_Control.slx
OuterLoop_Control.slx
Task with IterationQuery = padv.builtin.query.FindProjectFile
ProcessAdvisorExample.prj
Task with IterationQuery = padv.builtin.query.FindTopModels
Flight_Control.slx

Example: `IterationQuery = padv.builtin.query.FindModels`

Data Types: `string`

`InputQueries` — Inputs to task
empty array of `padv.Query` objects (default) | `padv.Query` object | name of `padv.Query` object | array of `padv.Query` objects

Inputs to the task, returned as:

- a `padv.Query` object
- the name of `padv.Query` object
- an array of `padv.Query` objects
- an array of names of `padv.Query` objects

By default, the task does not specify any artifacts as inputs. When you specify `InputQueries`, the task tasks the artifacts returned by the specified query or queries as an input.

Suppose a task runs once for each model in the project and you want to provide the models as inputs to the task. If you specify `InputQueries` as the built-in query `padv.builtin.query.GetIterationArtifact`, the query returns each artifact that the tasks iterates over, which in this example is each of the models in the project.

Example: `InputQueries = padv.builtin.query.GetIterationArtifact`

`Action` — Function that task runs
`[]` (default) | function handle

Function that the task runs, returned as the function handle. When you run the task, the task runs the function specified by the function handle.

For example, if you want the task to run the function `myFunction`, specify `Action` as `@myFunction`.

Example: `Action = @myFunction`

Data Types: `function_handle`

`RequiredIterationArtifactType` — Artifact type that task can run on
`""` (default) | string

Artifact type that the task can run on, returned by a string. The required iteration artifact type must be the artifact type supported by the `IterationQuery` property of the task.

Example: `RequiredIterationArtifactType = "sl_model_file"`

Data Types: `string`

`DescriptionText` — Task description
empty string (default) | string

Task description, returned as a string.

Example: "This task runs myScript."

Data Types: `string`

`DescriptionCSH` — Path to task documentation
empty string (default) | string

Path to task documentation, returned as a string.

Example: `DescriptionCSH = fullfile(pwd,"taskHelpFiles","myTaskDocumentation.pdf")`

Data Types: `string`

`Licenses` — List of licenses that task requires
empty string (default) | string array

List of licenses that the task requires, returned as a string array.

Example: `Licenses = ["matlab_report_gen" "simulink_report_gen"]`

Data Types: `string`

`Products` — List of products that must be installed to run task
empty string (default) | string array

List of products that must be installed to run the task, returned as a string array.

Data Types: `string`

`AllLicenseRequired` — Setting to require all licenses for task
`true` or `1` (default) | `false` or `0`

Setting to require all licenses for task, returned as a numeric or logical `1` (`true`) or `0` (`false`). By default, all licenses in the `Licenses` property of the task are required for the task to run. Specify `0` (`false`) if the task can run without all licenses listed in the `Licenses` property.

Example: `Licenses = ["matlab_report_gen" "simulink_report_gen"]`

Data Types: `logical`

`InputDependencyQuery` — Artifact dependencies for task inputs
`[1×1 padv.internal.QueryReference]` (default) | `padv.Query` object | name of `padv.Query` object

Artifact dependencies for task inputs, returned as a `padv.Query` object or the name of a `padv.Query` object.

`LaunchToolAction` — Function that launches a tool
`[]` (default) | function handle

Function that launches a tool, returned as the function handle.

When the property `LaunchToolAction` is specified, you can point to the task in the **Process Advisor** app and click **...** > **Open *`Tool Name`*** to open the tool associated with the task.

For tasks that are not built-in tasks, the task options show **...** > **Launch Tool**.

Example: `LaunchToolAction = @myFunction`

Data Types: `function_handle`

Object Functions

- `addInputQueries`
- `dependsOn`
- `run`
- `runsAfter`

Examples

Create Task Objects and Add Tasks to Process Model

You can use `padv.Task` to create task objects and then use the `addTask` function to add the task objects to the `padv.ProcessModel` object.

Open the **Process Advisor** example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the **Process Advisor** pane to the left of the Simulink canvas.

In the **Process Advisor** pane, click the **Edit process model** button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    taskA = padv.Task("taskA");
    taskB = padv.Task("taskB");

    runsAfter(taskB,taskA);

    addTask(pm,taskA);
    addTask(pm,taskB);

end
```

This code uses `padv.Task` to create two task objects: `taskA` and `taskB`.

The object function `runsAfter` specifies that `taskB` should run after `taskA`.

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Version History

Introduced in R2022a

padv.Task.addInputQueries

Package: padv

Add input artifacts as inputs to task

Syntax

`addInputQueries(taskObj,inputQueries)`

Description

`addInputQueries(taskObj,inputQueries)` adds the input artifacts returned by `inputQueries` as inputs to the task represented by `taskObj`.

If the task already has input queries specified, `addInputQueries` adds `inputQueries` to the list of input queries in the `InputQueries` property.

Examples

Add Inputs to Task

Use `addInputQueries` to specify the models in the project as inputs to a task.

Create a new `padv.Task` object `myTaskObj` that represents a task named `runForEachModel`.

```
myTaskObj = padv.Task("runForEachModel");
```

By default, the task does not have any inputs.

Use the function `addInputQueries` to add the built-in query `padv.builtin.query.FindModels` as the input query for the task.

```
addInputQueries(myTaskObj,padv.builtin.query.FindModels);
```

When you run the task defined by `myTaskObj`, the query `padv.builtin.query.FindModels` finds each model in the project and provides the models as the input artifacts for the task.

Input Arguments

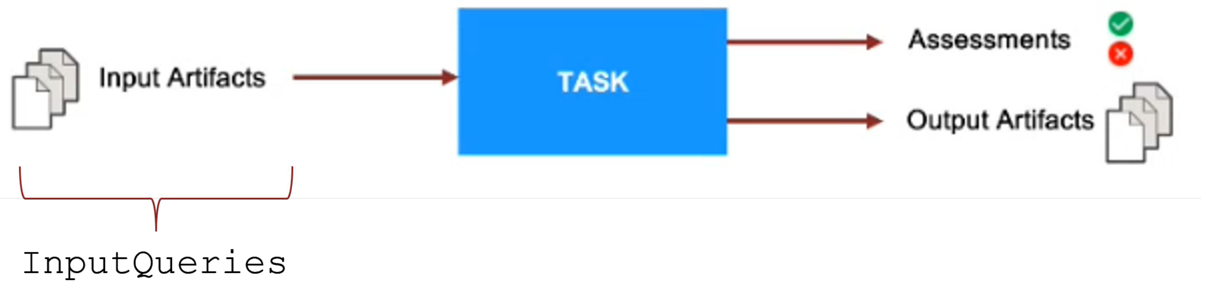
`taskObj` — Task object that represents task
`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

`inputQueries` — Queries that get input artifacts for task
`padv.Query` object | array of `padv.Query` object

A query or queries that get the input artifacts for a task, specified as a `padv.Query` object or an array of `padv.Query` objects. Each artifact that the query or queries return becomes an input to the task.



For example, if you specify the `InputQuery` property for a task as the query `padv.builtin.query.FindModels`, the query returns each model and the models become input artifacts for the task.

Note You can only specify the following queries for the `inputQueries` argument:

- `padv.builtin.query.FindArtifacts`
- `padv.builtin.query.FindFileWithAddress`
- `padv.builtin.query.FindModels`
- `padv.builtin.query.FindProjectFile`
- `padv.builtin.query.FindRequirements`
- `padv.builtin.query.FindRequirementsForModel`
- `padv.builtin.query.FindTestCasesForModel`
- `padv.builtin.query.FindTopModels`
- `padv.builtin.query.GetDependentArtifacts`
- `padv.builtin.query.GetIterationArtifact`
- `padv.builtin.query.GetOutputsOfDependentTask`

You cannot specify the following queries for `inputQueries`:

- `padv.builtin.query.FindFilesWithLabel`
- `padv.builtin.query.FindModelsWithLabel`
- `padv.builtin.query.FindModelsWithTestCases`
- `padv.builtin.query.FindRefModels`

Example: `addInputQueries(myTaskObj,padv.builtin.query.FindModels)`

Example: `addInputQueries(myTaskObj,[padv.builtin.query.GetIterationArtifact,padv.builtin.query.GetDependentArtifacts])`

Version History

Introduced in R2022a

padv.Task.dependsOn

Package: padv

Create dependency between tasks

Syntax

```
dependsOn(taskObj,dependencies)
dependsOn( ___ ,Name=Value)
```

Description

`dependsOn(taskObj,dependencies)` creates a dependency between `taskObj` and `dependencies`. `taskObj` runs only after the tasks specified by `dependencies` run and return a task status.

`dependsOn( ___ ,Name=Value)` specifies how the build system handles dependencies using one or more `Name=Value` arguments.

Examples

Create Dependency Between Two Tasks

Use the `dependsOn` function to create a dependency between two tasks in a process model.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

Open the `processmodel.m` file. The file is at the root of the project.

Replace the contents of the `processmodel.m` file with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    TaskA = padv.Task("TaskA");
    TaskB = padv.Task("TaskB");

    dependsOn(TaskB,TaskA);

    addTask(pm,TaskA);
    addTask(pm,TaskB);

end
```

This code uses `padv.Task` to create two task objects: `TaskA` and `TaskB`.

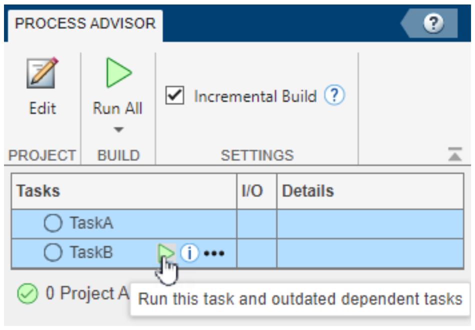
The object function `dependsOn` specifies that `TaskB` depends on `TaskA`.

The function addTask adds the task objects to the padv.ProcessModel object.

Open the **Process Advisor** app. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

In the **Tasks** column, point to the run button for **TaskB**. The **Process Advisor** app automatically highlights both tasks since **TaskA** is a dependent task. If you click the run button for **TaskB**, **TaskA** will run before **TaskB**.



Tasks	I/O	Details
TaskA		
TaskB		

0 Project A

Run this task and outdated dependent tasks

Input Arguments

taskObj — Task object that represents task
padv.Task object

Task object that represents a task, specified as a padv.Task object.

Example: myTaskObj = padv.Task("myTask");

dependencies — Tasks that need to run before taskObj runs
string | character vector | padv.Task object

Tasks that need to run before taskObj runs, specified as either:

- The name of a task, specified as a string or character vector.
- A padv.Task object.

Example: dependsOn(TaskB,"TaskA")

Example: dependsOn(TaskB,TaskA)

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: dependsOn(TaskB,TaskA,WhenStatus=["Pass","Fail"])

IterationArtifactMatching — Setting that controls which dependent task iterations run
true or 1 (default) | false or 0

Setting that controls which dependent task iterations run, specified as a numeric or logical `1` (`true`) or `0` (`false`):

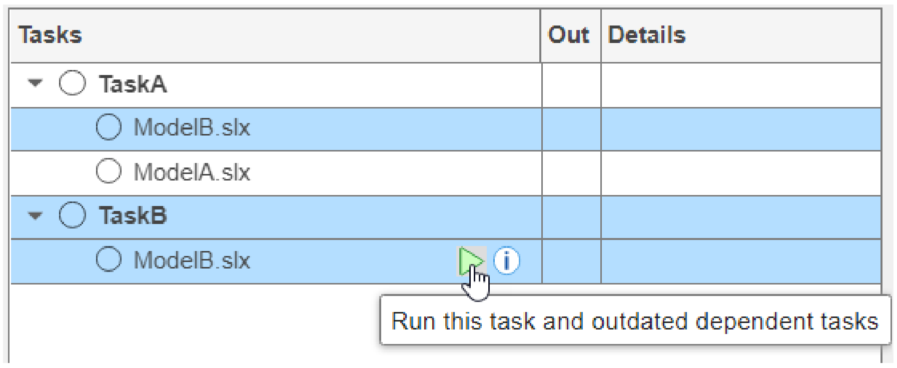
- `true` — When the build system runs the dependencies of a task, the build system runs only the task iterations that the tasks have in common.
- `false` — When the build system runs the dependencies of a task, the build system runs all task iterations. This behavior is useful when you have a task that creates new project artifacts and a task that runs on each artifact in the project. The second task depends on all project artifacts generated by the first task.

For example, suppose you have two tasks: `TaskA` and `TaskB`:

- `TaskA` runs on `ModelA` and `ModelB`.
- `TaskB` runs only on `ModelB` and depends on `TaskA`.

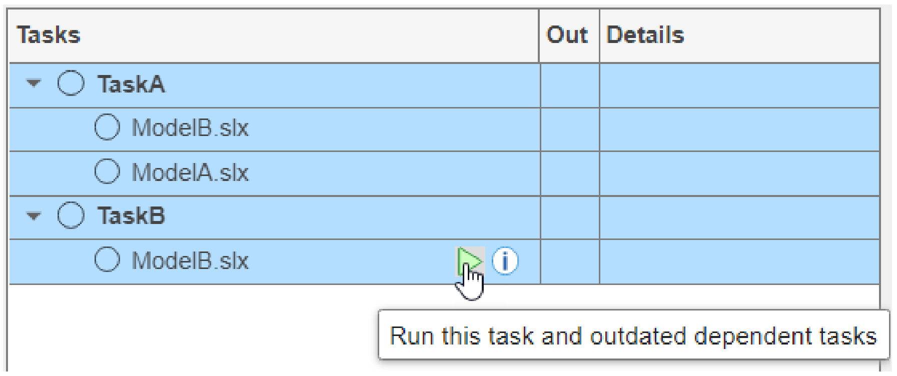
If you run `TaskB` and:

- `IterationArtifactMatching` is `true`, `TaskA` runs only on `ModelB`.



Tasks	Out	Details
TaskA		
ModelB.slx		
ModelA.slx		
TaskB		
ModelB.slx		

- `IterationArtifactMatching` is `false`, `TaskA` runs on both `ModelA` and `ModelB`.



Tasks	Out	Details
TaskA		
ModelB.slx		
ModelA.slx		
TaskB		
ModelB.slx		

Example: `dependsOn(TaskB,TaskA,IterationArtifactMatching=false)`

Data Types: `logical`

`WhenStatus` — Setting that controls when dependencies run
`"Pass"` (default) | `["Pass","Fail"]` | `["Pass","Fail","Error"]`

Setting that controls when dependencies run, specified as either:

- `"Pass"` — Only run the task if the dependencies pass. For example, if `TaskB` depends on `TaskA`, `TaskA` needs to pass before `TaskB` runs. If `TaskA` fails or errors, `TaskB` does not run.
- `["Pass","Fail"]` — Only run the task if the dependencies either pass or fail. For example, if `TaskB` depends on `TaskA`, `TaskA` needs to either pass or fail before `TaskB` runs. If `TaskA` errors, `TaskB` does not run.
- `["Pass","Fail","Error"]` — The task runs, even if the dependencies fail or error. For example, if `TaskB` depends on `TaskA`, `TaskA` can pass, fail, or error and `TaskB` still runs.

Example: `dependsOn(TaskB,TaskA,WhenStatus=["Pass","Fail"])`

Data Types: `string`

Version History

Introduced in R2022a

padv.Task.run

Package: padv

Run task

Syntax

```
taskResult = run(taskObj)
taskResult = run(taskObj,inputArtifacts)
```

Description

`taskResult = run(taskObj)` runs the task represented by `taskObj` and returns the result from the task.

How a task runs depends on how the you define the task. You can define tasks using a function or a class:

- Function-based tasks — Runs the function specified by the `Action` property of the task.
- Class-based task — Runs the `run` function implemented inside the class definition.

By default, when you create a `padv.Task` object, the task is a function-based task, even if you do not specify an `Action` property for the task.

`taskResult = run(taskObj,inputArtifacts)` uses the artifacts specified by `inputArtifacts` as inputs to the task. The `InputQuery` property of the task specifies the query that provides the `inputArtifacts` for the task.

Examples

Run Task

Create a new `padv.Task` object and run the task.

Create a new `padv.Task` object `myTaskObj` that represents a task named `myTask`.

```
myTaskObj = padv.Task("myTask");
```

Use the `run` object function to run the task. Save the results to the variable `taskResults`.

```
taskResults = run(myTaskObj)
```

```
taskResults = 

  TaskResult with properties:

             Status: Pass
    OutputArtifacts: [0×0 padv.Artifact]
            Details: [1×1 struct]
       ResultValues: [1×1 struct]
```

In this example, there is no `Action` associated with the task and the task returns a `padv.TaskResult` with a `Status` of `Pass`.

Input Arguments

`taskObj` — Task object that represents task
`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

`inputArtifacts` — Artifacts that are inputs to task
cell array of `padv.Artifact` objects

Artifacts that are inputs to the task, specified as a cell array of `padv.Artifact` objects.

If you specified the `InputQuery` property for a task, the `InputQuery` automatically passes a cell array of `padv.Artifact` objects to `inputArtifacts` when you run the task.

Output Arguments

`taskResult` — Result from task
`TaskResult` object

Result from the task, returned as a `TaskResult` object.

Version History

Introduced in R2022a

padv.Task.runsAfter

Package: padv

Specify preferred execution order for tasks

Syntax

```
runsAfter(taskObj,predecessors)
runsAfter( ___ ,Name=Value)
```

Description

`runsAfter(taskObj,predecessors)` specifies a preferred execution order for tasks. If possible, the build system runs the predecessor tasks, specified by `predecessors`, before the task represented by `taskObj`.

`runsAfter( ___ ,Name=Value)` specifies how the build system handles the preferred execution order using one or more `Name=Value` arguments.

Examples

Specify Preferred Execution Order for Two Tasks

Use the `runsAfter` function to specify that one task should run after another task.

Open the Process Advisor example project.

```
processAdvisorExampleStart
```

Open the `processmodel.m` file. The file is at the root of the project.

Replace the contents of the `processmodel.m` file with the following code:

```
function processmodel(pm)
    arguments
        pm padv.ProcessModel
    end

    FirstTask = padv.Task("FirstTask");
    SecondTask = padv.Task("SecondTask");

    runsAfter(SecondTask,FirstTask);

    addTask(pm,FirstTask);
    addTask(pm,SecondTask);

end
```

This code uses `padv.Task` to create two task objects: `FirstTask` and `SecondTask`.

The object function `runsAfter` specifies that `SecondTask` should run after `FirstTask`.

The function `addTask` adds the task objects to the `padv.ProcessModel` object.

Open the **Process Advisor** app. In the MATLAB Command Window, enter:

```
processAdvisorWindow
```

In the toolstrip, click the **Run All** button. You can see that **SecondTask** runs after **FirstTask**.

Input Arguments

`taskObj` — Task object that represents task

`padv.Task` object

Task object that represents a task, specified as a `padv.Task` object.

Example: `myTaskObj = padv.Task("myTask");`

`predecessors` — Tasks that should run before `taskObj` runs

string | character vector | `padv.Task` object

Tasks that should run before `taskObj` runs, specified as either:

- The name of a task, specified as a string or character vector.
- A `padv.Task` object.

Example: `runsAfter(SecondTask,"FirstTask")`

Example: `runsAfter(SecondTask,FirstTask)`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `runsAfter(SecondTask,FirstTask,StrictOrdering=true)`

`IterationArtifactMatching` — Setting that controls which predecessor task iterations run

`true` or `1` (default) | `false` or `0`

Setting that controls which predecessor task iterations run, specified as a numeric or logical `1` (`true`) or `0` (`false`):

- `true` — When the build system runs the predecessors of a task, the build system runs only the task iterations that the tasks have in common.
- `false` — When the build system runs the predecessor of a task, the build system runs all task iterations. This behavior is useful when you have a task that creates new project artifacts and a task that runs on each artifact in the project. The second task should run after all project artifacts generated by the first task.

For example, suppose you have two tasks: `FirstTask` and `SecondTask`:

- `FirstTask` runs on `ModelA` and `ModelB`.
- `SecondTask` runs only on `ModelB` and should run after on `FirstTask`.

If you run `SecondTask` and:

- `IterationArtifactMatching` is `true`, `FirstTask` runs only on `ModelB`.
- `IterationArtifactMatching` is `false`, `FirstTask` runs on both `ModelA` and `ModelB`.

Example: `runsAfter(SecondTask,FirstTask,IterationArtifactMatching=false)`

Data Types: `logical`

`StrictOrdering` — Setting that controls whether build system ignores circular relationships between tasks
`false` or `0` (default) | `true` or `1`

Setting that controls whether the build system ignores circular relationships between tasks, specified as a numeric or logical `0` (`false`) or `1` (`true`). By default, if you specify a circular relationship between tasks, the build system ignores the relationship. For example, if you specify both `runsAfter(SecondTask,FirstTask)` and `runsAfter(FirstTask,SecondTask)`, the build system ignores the `runsAfter` relationship.

If you specify `StrictOrdering` as `true`, the build system generates an error when you try to run tasks that have a circular relationship.

Example: `runsAfter(SecondTask,FirstTask,StrictOrdering=true)`

Data Types: `string`

Version History

Introduced in R2022a

padv.TaskResult

Create and access results from task

Description

A `padv.TaskResult` object represents the results from a task. The `run` function of a `padv.Task` creates a `padv.TaskResult` object that you can use to access the results from the task. When you create a custom task, you can specify the results from your custom task. You can also use the function `getProcessTaskResults` to view a list of the last task results for a project. The **Process Advisor** app uses task results to determine the task statuses, output artifacts, and detailed task results that appear in the **Tasks**, **Out**, and **Details** columns of the app.

Creation

Syntax

`resultObj = padv.TaskResult()`

Description

`resultObj = padv.TaskResult()` creates a result object `resultObj` that represents the results from a task.

Properties

`Status` — Task result status
`Pass` (default) | `Fail` | `Error`

Task result status, returned as:

- `Pass` — A passing task status. The task completed successfully without any issues.
- `Fail` — A failing task status. The task completed, but the results were not successful.
- `Error` — An error task status. The task generated an error and did not complete.

The `Status` property determines the task status shown in the **Tasks** column in the **Process Advisor** app.

For custom tasks, you can specify the task result status as either:

- `padv.TaskStatus.Pass` — Sets the `Status` property to `Pass`.
- `padv.TaskStatus.Fail` — Sets the `Status` property to `Fail`.
- `padv.TaskStatus.Error` — Sets the `Status` property to `Error`.

For example, `taskResult.Status = padv.TaskStatus.Fail` sets the `Status` property of the task result object to `Fail` to represent a failing task status.

Example: `Fail`

`OutputArtifacts` — Artifacts created by task
empty array `padv.Artifact.empty()` (default) | `padv.Artifact` object | array of `padv.Artifact` objects

Artifacts created by the task, returned as a `padv.Artifact` object or array of `padv.Artifact` objects.

The `OutputArtifacts` property determines the output artifacts shown in the **Out** column in the **Process Advisor** app.

The build system only manages output artifacts specified by the task result. For custom tasks, use the `OutputPaths` argument to define the output artifacts for the task result.

`Details` — Temporary storage for task-specific data
`struct` with no fields (default) | `struct`

Temporary storage for task-specific data, returned as a `struct`. The build system uses `Details` to store task-specific data that other build steps can use.

Note that `Details` are temporary. The build system does not save `Details` with the task results after the build finishes.

Data Types: `struct`

`ResultValues` — Number of passing, warning, and failing conditions
`struct` with `Pass: 0`, `Warn: 0`, `Fail: 0` (default) | `struct` with fields `Pass`, `Warn`, `Fail`

Number of passing, warning, and failing conditions, returned as a `struct` with fields:

- `Pass` — Number of passing conditions returned by task
- `Warn` — Number of warning conditions returned by task
- `Fail` — Number of failing conditions returned by task

The `ResultValues` property determines the detailed results shown in the **Details** column in the **Process Advisor** app.

For example, the task `padv.builtin.task.RunModelStandards` runs several Model Advisor checks and returns the number of passing, warning, and failing checks. If you run the task and one check passes, two checks generate a warning, and three checks fail, `ResultValue` returns:

```
ans = 

  struct with fields:

    Pass: 1
    Warn: 2
    Fail: 3
```

Data Types: `struct`

`OutputPaths` — Define `OutputArtifacts` for task result
character vector | string

This property is write-only.

`OutputArtifacts` for task result, specified as a list of paths.

The build system adds each path specified by `OutputArtifacts` to the `OutputArtifacts` argument as a `padv.Artifact` object with type `padv_output_file`.

Example: `taskResultObj.OutputPaths = string(fullfile(pwd,filename))`

Data Types: `char` | `string`

Object Functions

- `applyStatus`

Examples

Create Task Result for Custom Task

Create a `padv.TaskResult` object for a custom task that has a failing task status, outputs a single `.json` file, and 1 passing condition, 2 warning conditions, and 3 failing conditions.

Open the **Process Advisor** example project.

```
processAdvisorExampleStart
```

The model `AHRS_Voter` opens with the **Process Advisor** pane to the left of the Simulink canvas.

In the **Process Advisor** pane, click the **Edit process model** 📝 button to open the `processmodel.m` file for the project.

Replace the contents of the `processmodel.m` file with this example process model code:

```
function processmodel(pm)
    % Defines the project's processmodel

    arguments
        pm padv.ProcessModel
    end

    addTask(pm,"ExampleTask",Action=@ExampleAction);

end

function taskResult = ExampleAction(~)

    % Create a task result object that stores the results
    taskResult = padv.TaskResult();

    % Specify the task status shown in the Tasks column
    taskResult.Status = padv.TaskStatus.Fail;

    % Specify the output files shown in the Out column
    outputFile = "tools\sampleChecks.json";
    taskResult.OutputPaths = string(fullfile(pwd,outputFile));

    % Specify the values shown in the Details column
    taskResult.ResultValues.Pass = 1;
    taskResult.ResultValues.Warn = 2;
    taskResult.ResultValues.Fail = 3;
end
```

Save the `processmodel.m` file.

Go back to the **Process Advisor** app and click **Refresh Tasks** to update the list of tasks shown in the app.

In the top left corner of the **Process Advisor** pane, switch the filter from **Model** to **Project**.

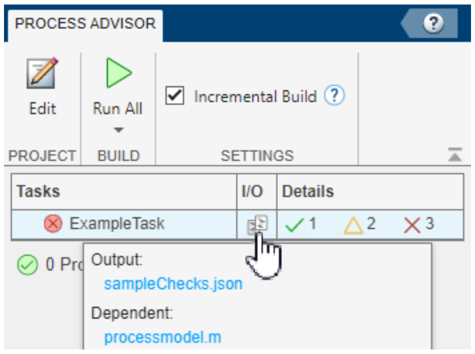
In the top right corner of the **Process Advisor** pane, click **Run All**.

- The **Tasks** column shows a failing task status to the left of **ExampleTask**. This code from the example process model specifies the task status shown in the **Tasks** column:

  ```
  taskResult.Status = padv.TaskStatus.Fail;
  ```
- The **Out** column shows an output artifact associated with the task. This code from the example process model specifies the output artifact shown in the **Out** column:

  ```
  taskResult.OutputPaths = string(fullfile(pwd,outputFile));
  ```
- The **Details** column shows 1 passing condition, 2 warning conditions, and 3 failing conditions. This code from the example process model specifies the detailed task results shown in the **Details** column:

  ```
  taskResult.ResultValues.Pass = 1;
  taskResult.ResultValues.Warn = 2;
  taskResult.ResultValues.Fail = 3;
  ```



Tasks	I/O	Details
ExampleTask		✓ 1 △ 2 ✕ 3

0 Pr

Output:
sampleChecks.json
Dependent:
processmodel.m

Version History

Introduced in R2022a

padv.TaskResult.applyStatus

Package: padv

Apply new task status if priority is higher

Syntax

```
applyStatus(resultObj,taskStatus)
```

Description

`applyStatus(resultObj,taskStatus)` applies a new task status `taskStatus` to the task result object `resultObj` if the priority level of `taskStatus` is higher than the current `Status` property of the task result object.

The priority levels from lowest to highest are:

- `padv.TaskStatus.Pass`
- `padv.TaskStatus.Fail`
- `padv.TaskStatus.Error`

Note The function `applyStatus` can only change the `Status` to a higher priority status. For example, if you apply a failing status and then apply a passing status, the status remains a failing status because the priority of `padv.TaskStatus.Fail` is higher than the priority of `padv.TaskStatus.Pass`.

```
taskResult = padv.TaskResult(); % By default, Status is Pass.
applyStatus(taskResult, padv.TaskStatus.Fail); % Status changes to Fail.
applyStatus(taskResult, padv.TaskStatus.Pass); % Status remains Fail.
taskResult
```

```
taskResult = 

  TaskResult with properties:

             Status: Fail
    OutputArtifacts: [0×0 padv.Artifact]
            Details: [1×1 struct]
       ResultValues: [1×1 struct]
```

To set the `Status` property of a task result object to a specific value, manually set the property to either `padv.TaskStatus.Pass`, `padv.TaskStatus.Fail`, or `padv.TaskStatus.Error`. For example, to set the `Status` of a task result object `taskResult` to `Pass`, use `taskResult.Status = padv.TaskStatus.Pass`.

Examples

Apply Status to Task Result

Use `applyStatus` to update the `Status` property of a task result object. If the status is a higher priority status, `applyStatus` updates the `Status` property of the task result object.

Create a task result object. By default, the `Status` property of the task result object is specified as `Pass`.

```
taskResult = padv.TaskResult();
```

Suppose the task needs to generate an error. Use `applyStatus` to apply an error task status, specified by `padv.TaskStatus.Error`.

```
applyStatus(taskResult,padv.TaskStatus.Error);
```

`padv.TaskStatus.Error` has a higher priority than a passing task status, so `applyStatus` updates the `Status` property of the task result object.

Apply a passing task status to the task result object. A passing task status is specified by `padv.TaskStatus.Pass`.

```
applyStatus(taskResult,padv.TaskStatus.Pass);
```

`padv.TaskStatus.Pass` does not have a higher priority than an error task status, so `applyStatus` does not change the `Status` of the task result object.

Inspect the properties of the task result object.

```
taskResult
```

Suppose you want to reset the status of the task result object to a passing task status. Manually specify the `Status` property as `padv.TaskStatus.Pass`.

```
taskResult.Status = padv.TaskStatus.Pass
```

```
taskResult = 

  TaskResult with properties:

             Status: Pass
    OutputArtifacts: [0×0 padv.Artifact]
            Details: [1×1 struct]
       ResultValues: [1×1 struct]
```

The task result object now has a passing task status.

Input Arguments

`resultObj` — Task result object
`padv.TaskResult` object

Task result object, specified as a `padv.TaskResult` object.

`taskStatus` — Task status
`padv.TaskStatus.Pass` | `padv.TaskStatus.Fail` | `padv.TaskStatus.Error`

Task status, specified as `padv.TaskStatus.Pass`, `padv.TaskStatus.Fail`, or `padv.TaskStatus.Error`.

Example: `padv.TaskStatus.Fail`

Version History

Introduced in R2022a

Built-In Tasks — Alphabetical List

The support package CI/CD Automation for Simulink Check contains several built-in tasks that you can use when you define your process. The built-in tasks have a default behavior, but you can reconfigure the built-in tasks to perform different actions. You can see a list of built-in tasks when you type `padv.builtin.task.` and use tab completion. You can also view the source code for the built-in tasks. The source code is located in the directory returned by this code:

```
fullfile(matlabshared.supportpkg.getSupportPackageRoot,...
"\toolbox\padv\build_service\ml\+padv\+builtin\+task")
```

The built-in tasks include:

Task Title	Task Name and Source Code File
Check Coding Standards (Ref)	• Name: `"padv.builtin.task.AnalyzeRefModelCode"` • File: `AnalyzeRefModelCode.m`
Check Coding Standards (Top)	• Name: `"padv.builtin.task.AnalyzeTopModelCode"` • File: `AnalyzeTopModelCode.m`
Check Modeling Standards	• Name: `"padv.builtin.task.RunModelStandards"` • File: `RunModelStandards.m`
Detect Design Errors	• Name: `"padv.builtin.task.DetectDesignErrors"` • File: `DetectDesignErrors.m`
Generate Code (Ref)	• Name: `"padv.builtin.task.GenerateCodeAsRefModel"` • File: `GenerateCodeAsRefModel.m`
Generate Code (Top)	• Name: `"padv.builtin.task.GenerateCodeAsTopModel"` • File: `GenerateCodeAsTopModel.m`
Generate SDD Report	• Name: `"padv.builtin.task.GenerateSDDReport"` • File: `GenerateSDDReport.m`
Generate Simulink Web View	• Name: `"padv.builtin.task.GenerateSimulinkWebView"` • File: `GenerateSimulinkWebView.m`
Inspect Code (Ref)	• Name: `"padv.builtin.task.RunCodeInspectionAsRefModel"` • File: `RunCodeInspection.m`
Inspect Code (Top)	• Name: `"padv.builtin.task.RunCodeInspectionAsTopModel"` • File: `RunCodeInspection.m`
Merge Test Results	• Name: `"padv.builtin.task.MergeTestResults"` • File: `MergeTestResults.m`
Run Tests	• Name: `"padv.builtin.task.RunTestsPerModel"` • File: `RunTestsPerModel.m`
Run Tests	• Name: `"padv.builtin.task.RunTestsPerTestCase"` • File: `RunTestsPerTestCase.m`

Note that if a task title includes **(Ref)**, the task runs on reference models in the project. If a task title include **(Top)**, the task runs on top models in the project.

Reference pages for the built-in task are listed alphabetically on the following pages.

Check Coding Standards (Ref)

This task uses Polyspace Bug Finder to analyze generated reference model code for run-time defects, coding standards, and code metrics. This task runs on the generated reference model code for each model in the project.

If a model does not have generated code, the task skips the analysis for the model and displays a warning message.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.AnalyzeRefModelCode`	**Check Coding Standards (Ref)**

Use the `addTask` function to add the task to the process model. To check if Polyspace Bug Finder is installed and setup before you add the **Check Coding Standards (Ref)** task, use this code:

```
if exist('polyspaceroot','file') % if Polyspace installed and set up
    psTaskObj = addTask(pm, padv.builtin.task.AnalyzeRefModelCode);
end
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.AnalyzeRefModelCode
```

Check Coding Standards (Top)

This task uses Polyspace Bug Finder to analyze generated code for run-time defects, coding standards, and code metrics. This task runs on the generated top model code in the project.

If a model does not have generated code, the task skips the analysis for the model and displays a warning message.

Task Instance	Task Title in Process Advisor
padv.builtin.task.AnalyzeTopModelCode	**Check Coding Standards (Top)**

Use the `addTask` function to add the task to the process model. To check if Polyspace Bug Finder is installed and setup before you add the **Check Coding Standards (Top)** task, use this code:

```
if exist('polyspaceroot','file') % if Polyspace installed and set up
    psTaskObj = addTask(pm, padv.builtin.task.AnalyzeTopModelCode);
end
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.AnalyzeTopModelCode
```

Check Modeling Standards

This task uses the Model Advisor to check your models for modeling conditions and configuration settings that cause inaccurate or inefficient simulation of the system that the model represents. Running model standards checking can also help you verify compliance with industry standards and guidelines.

You can configure this task to specify which model standards the task runs. For example, you can specify a Model Advisor configuration file or list of check identifiers to include in the Model Advisor analysis. If you do not specify which model standards to run, the task runs a subset of high-integrity systems checks by default.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.RunModelStandards`	**Check Modeling Standards**

Use the `addTask` function to add the task to the process model:

```
maTaskObj = addTask(pm,padv.builtin.task.RunModelStandards);
```

You can reconfigure the task object to specify a different Model Advisor configuration file:

```
% Create a query that looks for your Model Advisor Configuration file
findMyConfigFile = padv.builtin.query.FindFileWithAddress(...
'ma_config_file', fullfile('tools','sampleChecks.json'));

% Add the configuration file as an input to the task
addInputQueries(maTaskObj,findMyConfigFile);
```

If you wanted to specify a list of check IDs instead of a configuration, you could modify the `RunOptions` of `maTaskObj`:

```
maTaskObj.RunOptions.CheckIDList = {'mathworks.jmaab.db_0032',...
'mathworks.jmaab.jc_0281'};
```

If you specify both a Model Advisor configuration file and a list of check IDs for a task, the task uses the Model Advisor configuration file.

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunModelStandards
```

Detect Design Errors

This task uses Simulink Design Verifier to statically detect run-time errors and dead logic and to derive design ranges on your model. Design error detection can identify dead logic, integer overflow, division by zero, and violations of design properties and assertions.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.DetectDesignErrors`	**Detect Design Errors**

Use the `addTask` function to add the task to the process model:

```
dedObj = addTask(pm,padv.builtin.task.DetectDesignErrors);
```

You can reconfigure the run options of the task object to change the analysis options:

```
dedObj.RunOptions.DetectDeadLogic = 'on';
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.DetectDesignErrors
```

By default, this task outputs a design error detection report and data file.

Generate Code (Ref)

This task uses Embedded Coder to generate code that other models can reference. By default, this task runs on the referenced models in the project. Referenced models are models that other models in the project reference. The task returns the generated code report as an output file.

Note This task generates code but does not build executable files.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.GenerateCodeAsRefModel`	**Generate Code (Ref)**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateCodeAsRefModel);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateCodeAsRefModel
```

Generate Code (Top)

This task uses Embedded Coder to generate code for standalone use. By default, this task runs on the top models in the project. Top models are models which are not referenced by any other models in the project. The task returns the generated code report as an output file.

Note This task generates code but does not build executable files.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.GenerateCodeAsTopModel`	**Generate Code (Top)**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateCodeAsTopModel);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateCodeAsTopModel
```

Generate SDD Report

This task uses Simulink Report Generator to generate a System Design Description (SDD) report from a predefined template. The System Design Description report provides a summary or detailed information about a system design represented by a model.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.GenerateSDDReport`	**Generate SDD Report**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateSDDReport);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateSDDReport
```

Generate Simulink Web View

This task uses the Simulink Report Generator to create a Web view for your models.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.GenerateSimulinkWebView`	**Generate Simulink Web View**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.GenerateSimulinkWebView);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.GenerateSimulinkWebView
```

Inspect Code (Ref)

This task uses the Simulink Code Inspector to detect unintended functionality in your reference models by establishing model-to-code and code-to-model traceability. The results of this task can help you to satisfy code-review objectives in DO-178 and other high-integrity standards.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.RunCodeInspection`	**Inspect Code (Ref)**

Use the `addTask` function to add the task to the process model and use the `IsTopModel` property to specify that the task should inspect reference model code:

```
addTask(pm,padv.builtin.task.RunCodeInspection("IsTopModel",false));
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunCodeInspection
```

Inspect Code (Top)

This task uses the Simulink Code Inspector to detect unintended functionality in your top models by establishing model-to-code and code-to-model traceability. The results of this task can help you to satisfy code-review objectives in DO-178 and other high-integrity standards.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.RunCodeInspection`	**Inspect Code (Top)**

Use the `addTask` function to add the task to the process model and use the `IsTopModel` property to specify that the task should inspect top model code:

```
addTask(pm,padv.builtin.task.RunCodeInspection("IsTopModel",true));
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunCodeInspection
```

Merge Test Results

This task uses Simulink Test and Simulink Coverage to generate a consolidated test results report and a merged coverage report for a model.

Task Instance	**Task Title in Process Advisor**
`padv.builtin.task.MergeTestResults`	**Merge Test Results**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.MergeTestResults);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.MergeTestResults
```

Run Tests (per model)

This task uses Simulink Test to run the test cases associated with your model. The task runs the test cases on a model-by-model basis. The **Process Advisor** shows the name of each model under the **Run Tests** task. Certain tests may generate code.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.RunTestsPerModel`	**Run Tests**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.RunTestsPerModel);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunTestsPerModel
```

Run Tests (per test case)

This task uses Simulink Test to run the test cases associated with your model. The task runs the test cases on a test-by-test basis. The **Process Advisor** shows the name of each test case under the **Run Tests** task. Certain tests may generate code.

Task Instance	Task Title in Process Advisor
`padv.builtin.task.RunTestsPerTestCase`	**Run Tests**

Use the `addTask` function to add the task to the process model:

```
addTask(pm,padv.builtin.task.RunTestsPerTestCase);
```

To view the source code for this built-in task, in the MATLAB Command Window, enter:

```
open padv.builtin.task.RunTestsPerTestCase
```

Built-In Queries — Alphabetical List

The support package CI/CD Automation for Simulink Check contains several built-in queries that you can use when you define your process. You can see a list of built-in queries when you type `padv.builtin.query.` and use tab completion. The tab completion shows a list of the available built-in queries.

The built-in queries include:

Query Instance	Description
`padv.builtin.query.FindArtifacts`	Returns each of the artifacts in project that meet the specified criteria For information, enter: `help padv.builtin.query.FindArtifacts`
`padv.builtin.query.FindFilesWithLabel`	Returns each of the files in the project that have the specified project label and project label category
`padv.builtin.query.FindFileWithAddress`	Returns the file at the specified address in the project
`padv.builtin.query.FindModels`	Returns each of the models in the project
`padv.builtin.query.FindModelsWithLabel`	Returns each of the models in the project that have the specified project label and project label category
`padv.builtin.query.FindModelsWithTestCases`	Returns each of the models that are associated with a test case
`padv.builtin.query.FindProjectFile`	Returns the project file
`padv.builtin.query.FindRefModels`	Returns each of the referenced models in the project
`padv.builtin.query.FindRequirements`	Returns each of the requirement sets within a project
`padv.builtin.query.FindRequirementsForModel`	Returns each of the requirement links associated with a model
`padv.builtin.query.FindTestCasesForModel`	Returns each of the test cases associated with the models in the project
`padv.builtin.query.FindTopModels`	Returns each of the top models in the project
`padv.builtin.query.GetDependentArtifacts`	Returns the dependent artifacts for the specified artifact
`padv.builtin.query.GetIterationArtifact`	Returns the artifact that the task is iterating over

Query Instance	Description
`padv.builtin.query.GetOutputsOfDependentTask`	Returns the outputs from the immediate dependent task

Note You can only specify certain queries as the input query of a task. For more information, see the documentation for "padv.Task.addInputQueries" in the chapter "Classes — Alphabetical List".

Use Built-In Query to Find Artifacts in Project

You can use the built-in query `padv.builtin.query.FindArtifacts` to find project artifacts that meet your specified criteria.

Use the arguments of `padv.builtin.query.FindArtifacts` to specify your search criteria. The input arguments include:

- `ArtifactType` — Type of artifact, specified as a character vector. For example `'sl_model_file'` for a Simulink model.
- `IncludeLabel` — Find artifacts with a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. For example, `{'Classification','Design'}`.
- `ExcludeLabel` — Exclude artifacts with a specific project label, specified as a cell array where the first entry is the project label category and the second entry is the project label name. For example, `{'Classification','Design'}`.
- `IncludePath` — Find artifacts where the path contains specific text, specified as a character vector. For example, `'HLR'` to find artifacts where the path contains the text `HLR`.
- `ExcludePath` — Exclude artifacts where the path contains specific text, specified as a character vector. For example, `'HLR'` to exclude artifacts where the path contains the text `HLR`.

For example, suppose you want to find the artifacts in the project that use the project label `Design` from the project label category `Classification`. You can create a `padv.builtin.query.FindArtifacts` query object and use the `IncludeLabel` argument to specify that the query should include artifacts with the specified the project label category and project label name.

```
findDesignArtifacts = padv.builtin.query.FindArtifacts(...
IncludeLabel = {'Classification','Design'});
```

When you run the query, the artifacts are returned in the first cell of the output cell array.

```
queryOutput = run(findDesignArtifacts);
designArtifacts = queryOutput{1}

artifacts = 

  1×24 Artifact array with properties:

    Type
    Parent
    Address
    UUID
    Label
    StorageAddress
```

Alternatively, you can use the `ExcludeLabel` argument to create a query that excludes artifacts that use that label:

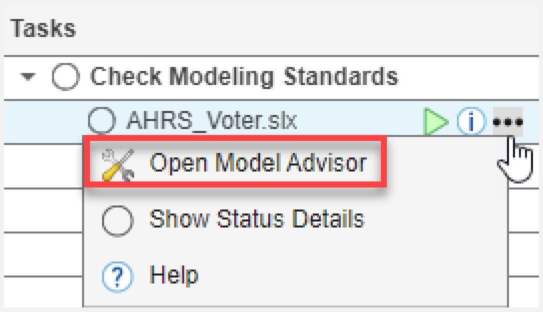
```
excludeDesignArtifacts = padv.builtin.query.FindArtifacts(...
ExcludeLabel = {'Classification','Design'});

queryOutput2 = run(excludeDesignArtifacts);
nonDesignArtifacts = queryOutput2{1};
```

For more information, enter:

```
help padv.builtin.query.FindArtifacts
```

Version History

Supported Releases and Updates	Support Package Update	Description

• R2022b Update 1 (and later updates)
• R2022a Update 4 (and later updates)
	December 2022	Features:
		• Automatically generate a pipeline configuration file for a GitLab pipeline by using the new function `padv.pipeline.generatePipeline`. For more information, see the section "Integrate into a GitLab CI System" or enter:
		`help padv.pipeline.generatePipeline`
		• Open the tool associated with a task by pointing to the task in the **Process Advisor** app and clicking **...** > **Open *`Tool Name`***.
		 Tasks
		Check Modeling Standards
		AHRS_Voter.slx
		Open Model Advisor
		Show Status Details
		Help
		• Automatically view detailed statuses, inputs, outputs, and dependencies for tasks and task results shown in the **Process Advisor** app.
		• The built-in task **Design Error Detection** now outputs the Simulink Design Verifier data file as an output in the **I/O** column.
		• Find artifacts in your project that meet specific search criteria by using the new built-in query `padv.builtin.query.FindArtifacts`.
		For information, enter:
		`help padv.builtin.query.FindArtifacts`
		• Find requirement sets in your project and requirement links to models by using the new built-in queries `padv.builtin.query.FindRequirements` and `padv.builtin.query.FindRequirementsForModel`, respectively.

Supported Releases and Updates	Support Package Update	Description
• R2022b Update 1 (and later updates) • R2022a Update 4 (and later updates)	November 2022	Features: • You can now open artifacts, in their associated tool, directly from the **Process Advisor** app. In the **Tasks** column, point to the name of an artifact and click the hyperlink. • If there is a new version of the support package available, the **Process Advisor** app shows an update icon in the bottom right corner. • The built-in task for generating a Simulink Web view now includes additional options like the ability to include user notes and export models in subfolders. To view the source code for the task, enter this code in the MATLAB Command Window: `open padv.builtin.task.GenerateSimulinkWebView` Fixes: • The **Process Advisor** app respects requests to cancel artifact analysis. • The tasks `padv.builtin.task.AnalyzeRefModelCode` and `padv.builtin.task.AnalyzeTopModelCode` return an error if Polyspace Bug Finder is either not installed or not linked to the current MATLAB installation.
	October 2022	Features: • The support package now supports R2022b for Update 1 and later updates. • Turn off incremental builds for a project by clearing the **Incremental Build** check box in the **Process Advisor** app. For more information, see the section "How to Disable Incremental Builds". • The build system and **Process Advisor** app take advantage of `runsAfter` relationships when determining the task execution order for tasks associated with the project.

Supported Releases and Updates	Support Package Update	Description

• R2022a Update 4 (and later updates)	September 2022	Features:
		
		• You can create a new example project instance that includes an example YAML file for configuring GitLab pipelines:
		
		`processAdvisorGitLabExampleStart`
		
		The example YAML file, `.gitlab-ci.yml`, is in the project root.
		• You can create a new example project instance that includes an example Jenkinsfile for configuring Jenkins pipelines:
		
		`processAdvisorJenkinsExampleStart`
		
		The example Jenkinsfile, `Jenkinsfile`, is in the project root.
		• Test harnesses are now tracked as dependencies for test cases.
		• Externally-saved input or output baselines (including `.mat` and Excel) are now tracked as dependencies for test cases.
		
		Fixes:
		
		• If you are using the project window and there is an error, the error dialog is able to open the artifact listed in the hyperlink.

	August 2022	Initial release.