LISTEN.
THINK.
SOLVE.®

PharmaSuite®

PHASES OF THE IPC PACKAGE

RELEASE 8.4

TECHNICAL MANUAL

PUBLICATION PSIP-PM003C-EN-E-DECEMBER-2017
Supersedes publication PSIP-PM003B-EN-E

Allen-Bradley • Rockwell Software

Rockwell Automation

Contact Rockwell See contact information provided in your maintenance contract.

Trademark Notices FactoryTalk, PharmaSuite, Rockwell Automation, Rockwell Software, and the Rockwell Software logo are registered trademarks of Rockwell Automation, Inc.

The following logos and products are trademarks of Rockwell Automation, Inc.:

FactoryTalk Shop Operations Server, FactoryTalk ProductionCentre, FactoryTalk Administration Console, FactoryTalk Automation Platform, and FactoryTalk Security.
Operational Data Store, ODS, Plant Operations, Process Designer, Shop Operations, Rockwell Software CPGSuite, and Rockwell Software AutoSuite.

Other Trademarks ActiveX, Microsoft, Microsoft Access, SQL Server, Visual Basic, Visual C++, Visual SourceSafe, Windows, Windows 7 Professional, Windows Server 2008, Windows Server 2012, and Windows Server 2016 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe, Acrobat, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ControlNet is a registered trademark of ControlNet International.

DeviceNet is a trademark of the Open DeviceNet Vendor Association, Inc. (ODVA).

Ethernet is a registered trademark of Digital Equipment Corporation, Intel, and Xerox Corporation.

OLE for Process Control (OPC) is a registered trademark of the OPC Foundation.

Oracle, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation.

All other trademarks are the property of their respective holders and are hereby acknowledged.

Warranty This product is warranted in accordance with the product license. The product's performance may be affected by system configuration, the application being performed, operator control, maintenance, and other related factors. Rockwell Automation is not responsible for these intervening factors. The instructions in this document do not cover all the details or variations in the equipment, procedure, or process described, nor do they provide directions for meeting every possible contingency during installation, operation, or maintenance. This product's implementation may vary among users.

This document is current as of the time of release of the product; however, the accompanying software may have changed since the release. Rockwell Automation, Inc. reserves the right to change any information contained in this document or the software at any time without prior notice. It is your responsibility to obtain the most current information available from Rockwell when installing or using this product.

Contents

Figures

Chapter 1

Introduction

This documentation contains important information how to configure and adapt phase building blocks of the IPC Phase Package including information about actions that need to be performed in FactoryTalk® ProductionCentre and PharmaSuite.

The manual is an addition to the "Technical Manual Configuration and Extension" [A2]-[A6] (page 27).

The information is structured in the following sections:

Adapting Data Collection and Data Representation Phases

- **Get values**, **Show values** phases
  - Creating a data collection phase (page 5)
  - Creating a data representation phase (page 10)
  - Technical details (page 11)

Adapting Trigger Phases

- **Time-based trigger**, **Counter-based trigger** phases
  - Technical overview (page 13)
  - Plugin class (page 16)
  - Model class (page 20)
  - Trigger phase class (page 21)
  - Creating a trigger phase (page 22)
  - For details on how to debug the OE server and the phases running on it, see section "Debugging PharmaSuite Event Sheets" in chapter "Monitoring PharmaSuite and Related Components" in "Technical Manual Administration" [A11] (page 27).

Finally, the Reference Documents (page 27) section provides a list of all the documentation that is referenced in this manual.

Intended Audience

This manual is intended for system engineers, phase developers, and system administrators who maintain phase building blocks of the IPC Package.

They need to have a thorough working knowledge of FactoryTalk ProductionCentre, PharmaSuite, and PharmaSuite building blocks. Thus we highly recommend to participate in a FactoryTalk ProductionCentre and PharmaSuite training before starting on the tasks described in this manual.
For more information on S88, EBR, and phase building blocks, see also "Technical Manual Developing System Building Blocks" [A7] (page 27).

Naming Recommendations

When naming your artifacts (e.g. objects in Process Designer, classes, interfaces, methods, functions, building blocks) consider the following recommendation:

- Define and make use of a vendor code consisting of up to three uppercase letters as prefix (e.g. MYC for My Company).
- Prefix your artifacts with your vendor code, separated by _.
  X_ and **RS_** are reserved for PharmaSuite.
- Observe the naming conventions defined in "Technical Manual Developing System Building Blocks" [A7] (page 27).

Typographical Conventions

This documentation uses typographical conventions to enhance the readability of the information it presents. The following kinds of formatting indicate specific information:

Bold typeface Designates user interface texts, such as

- window and dialog titles
- menu functions
- panel, tab, and button names
- box labels
- object properties and their values (e.g. status).

Italic typeface Designates technical background information, such as

- path, folder, and file names
- methods
- classes.

CAPITALS	Designate keyboard-related information, such as ■ key names ■ keyboard shortcuts.
`Monospaced typeface`	Designates code examples.

Chapter 2

Adapting Data Collection and Data Representation Phases

This chapter applies to the **Get values** and **Show values** phases. You will learn how to adapt a data collection and a data representation phase.

A data collection phase gathers data records that can later be shown by a data representation phase. The data representation phase can show the data of multiple runs of the referenced data collection phase. This set-up can be realized either in the context of an event-triggered operation (ETO) or with loops in the underlying master recipe.

For more information on data collection and data representation phases, see also "Functional Requirement Specification IPC Phases" [A8] (page 27) and "User Manual IPC Phases" [A9] (page 27).

TIP

Java classes of phases end with a version number, e.g. *MESParamDatProvDef0100* for version 1.00. In the following description, *MESParamDatProvDef<version>* is used.

Creating a Data Collection Phase

This section describes how to create a data collection phase.

For this purpose, perform the following steps:

1. Create an output for the phase data reference (page 6).
2. Create the getter for the phase data reference output (page 6).
3. Implement the data provider to be used by a data representation phase (page 7).

This description assumes that you already have a phase with static (persistent) outputs, i.e. an ATDefinition stores the outputs and generated output access classes are available.

Creating an Output for the Phase Data Reference

The output for the phase data reference is only necessary if a data representation phase shall display the data collected by your data collection phase.

> **TIP**
>
> The output must be added manually, since there is no corresponding column in the ATDefinition of your phase's output.
> For this purpose, use the generated **phase output data bean**, i.e. the *MESRtPhaseOutput<base name of your phase>* class.

To add an additional output for the phase data reference to the existing outputs in the **phase output data bean** of your phase, add a fixed *IBuildingBlockOutputDescriptor* for the phase data reference.

```
static {
    // add phase data reference to (static) outputs
    IBuildingBlockOutputDescriptor descriptor =
              createOutputDescriptorForPhaseDataReference("phaseDataReference");
    OUTPUT_DESCRIPTORS.add(descriptor);
  }
```

The phase data reference can be referenced by a data representation phase with the information flow mechanism.

Creating the Getter for the Phase Data Reference Output

To return your phase data reference output to a referencing phase in the **phase output data bean** of your phase, implement the getter for the phase data reference.

> **TIP**
>
> The getter name must be *getPhaseDataReference()*, since the output is called *phaseDataReference* (see "Creating an Output for the Phase Data Reference" (page 6)).

```
public PhaseDataReference getPhaseDataReference() {
    return new PhaseDataReference(getParent(), MyDataProvider.class);
  }
```

Pass the class of your data provider implementation as second parameter to the constructor.

Implementing the Data Provider to Be Used by a Data Representation Phase

TIP

For performance reasons, PharmaSuite uses SQL statements to retrieve the data. The default implementation of the *public List<PhaseDataRow> getData(IMESChoiceElementceScope, IMESRtPhase rtPhaseOfCaller)* method automatically creates the statements for the supported data scopes. If you create your own statements, e.g. to support a new scope, please be aware of the fact that future schema changes may require an update of your statements when migrating to another PharmaSuite or FactoryTalk ProductionCentre release.

Step 1

Create your data provider by extending *AbstractPhaseDataProvider*:

- Create a new class that extends *AbstractPhaseDataProvider* (*AbstractPhaseDataProvider* implements the *IPhaseDataProvider* interface)

```
public class MyDataProvider extends AbstractPhaseDataProvider {
  public MyDataProvider(long rtPhaseKey) {
    super(rtPhaseKey);
  }
}
```

- Make sure to implement a constructor that expects the key of the underlying runtime phase of your data collection phase as parameter.

Step 2

The *getAttributeDescriptors()* method obtains meta information of the actual values the *getData()* method returns to the data representation phase.

Basic sample implementation of getAttributeDescriptors()

```
@Override
 public List<IPhaseDataAttributeDescriptor> getAttributeDescriptors() {
    List<IPhaseDataAttributeDescriptor> result = new ArrayList<>();

    // descriptor for instance count
    String instanceCountLabel = "Run";
    IPhaseDataAttributeDescriptor descriptor = new
              StringPhaseDataAttributeDescriptor(instanceCountLabel, "instanceCount");
    result.add(descriptor);

    // actual value
    String shortDescription = "String value";
    String attributeIdentifier = "value";
    descriptor = new StringPhaseDataAttributeDescriptor(shortDescription,
                                                        attributeIdentifier);
    result.add(descriptor);

    // descriptor for phase completion time & user
    String phaseDataTimeLabel = "Signature / Time";
    descriptor = new StringPhaseDataAttributeDescriptor(phaseDataTimeLabel,
                     "phaseDataTimeAndSignature");
```

```
    result.add(descriptor);

    return result;
  }
```

- Make sure to use the data type-specific subclasses of *AbstractPhaseDataAttributeDescriptor*:

Subclass	Data type
MeasuredValuePhaseDataAttributeDescriptor	MeasuredValue
StringValuePhaseDataAttributeDescriptor	String
BooleanPhaseDataAttributeDescriptor	Boolean
OptionValuePhaseDataAttributeDescriptor	OptionValue

- In our example, we also pass the run and signature information in addition to the actual value.
- We recommend to create the attribute descriptors only once during object construction and remember them, e.g. in a field.

Step 3

Implement *getPhaseDataTableName()* to return the name of the database table containing the phase data:

```
  private static final String PHASE_DATA_TABLE_NAME = //
      MESRtPhDatPAVStr.SQL_TABLE_NAME_PREFIX + MESRtPhDatPAVStr.ATDEFINITION_NAME;
@Override
  public String getPhaseDataTableName() {
    return PHASE_DATA_TABLE_NAME;
  }
```

Step 4

Implement the *getColumnDescriptors()* method:
The method describes the database columns to be fetched from the database, especially from the phase data table:

```
@Override
  public List<ColumnDescriptor> getColumnDescriptors() {
    List<ColumnDescriptor> result = new ArrayList<>();
    getColumnDescriptorsForInstanceCount(result);
    // actual value
    ColumnDescriptor columnDescriptor = new ColumnDescriptor(
          COLUMNDESCRIPTOR_PREFIX_PHASEDATA + "X_ActualValue_S",
          IDataTypes.TYPE_STRING);
    result.add(columnDescriptor);
    getColumnDescriptorsForPhaseCompletionInfo(result);
    return result;
  }
```

- Create column descriptors to provide the actual data values or rows to the data representation phase (see *createObjectRecordFromStringRecord ()* method).
- The SQL statements use the column descriptors to fetch the collected data.

Step 5

Implement *createObjectRecordFromStringRecord ()* to convert a single string-based data row as it is returned by the executed SQL statement into the actual (typed) data row that is returned by the *getData()* method:

createObjectRecordFromStringRecord ()

```
@Override
  public Object[] createObjectRecordFromStringRecord(String[] stringRecord) {
    Object[] result = new Object[getAttributeDescriptors().size()];

    result[0] = getInstanceCountFromStringRecord(stringRecord);
    final int stringRecordOffset = 1; // instance count and template count

    // actual value
    final int actualValueIndex = 1;
    Class targetDataType = getAttributeDescriptors().get(actualValueIndex)
                           .getDataType();
    Object obj = getObjectFromString(stringRecord[actualValueIndex +
                                     stringRecordOffset], targetDataType);
    result[actualValueIndex] = obj;

    final int signatureInfoStartIndex = 2;
    final int numOfColumnsSelectedForPhaseCompletionInfo = 9;
    String[] stringRecordPhaseCompletionInfo = Arrays.copyOfRange(stringRecord,
                  signatureInfoStartIndex + stringRecordOffset,
                  signatureInfoStartIndex + stringRecordOffset +
                  numOfColumnsSelectedForPhaseCompletionInfo);
    result[signatureInfoStartIndex] = getPhaseCompletionInformationFromStringRecord(
                                      stringRecordPhaseCompletionInfo);

    return result;
  }
```

- Convert each of the string result attributes to an object that corresponds to the type defined in *getAttributeDescriptors()*.
- The size of the result array is identical to the size of the attribute descriptors.
- Please be aware that the number of columns in the SQL statement (i.e. the number of column descriptors) may exceed the number of attribute descriptors (i.e. the number of actual values per data record). Example: an actual data value is determined by multiple columns of the SQL statement result (e.g. when providing phase completion signature information as a single **String** value).

Creating a Data Representation Phase

This section describes how to create a data representation phase.

For this purpose, perform the following step:

- Get data from a data collection phase (page 10).

This description assumes that you already have a data collection phase that stores phase data and provides a phase data reference output.

Getting Data from a Data Collection Phase

Your data representation phase requires the **Data Provider Definition (RS) [1.0]** phase parameter.

Step 1

Get the processes parameter:

```
MESParamDatProvDef<version> dataProviderDefinition =
                  getRtPhase().getProcessParameterData(MESParamDatProvDef<version>
                 .class, "parameterIdentifier");
```

Step 2

Get the data provider of the data collection phase from the process parameter:

```
PhaseDataReference dataReference = dataProviderDefinition.getDataReference();
IPhaseDataProvider dataProvider = dataReference.createPhaseDataProvider();
```

- If the data representation phase has been activated before the data collection phase has been completed, the data reference can be null. In this case, use the *determinePhaseDataReference()* method of *IS88ExecutionService* to determine the data reference

```
PhaseDataReference dataReference = ServiceFactory.getService(IS88ExecutionService.class)
                                   .determinePhaseDataReference(getRtPhase(),
                                   "parameterIdentifier", "dataReference");
```

Then, you can use this phase data reference directly and obtain the phase data provider by calling *createPhaseDataProvider()* on the reference. Alternatively, you can update your *MESParamDatProvDef<version>* parameter with this newly obtained phase data reference and retrieve it again as described above.

TIP
When you determine a phase data reference with *IS88ExecutionService.determinePhaseDataReference()*, you always get a valid reference provided there is a completed instance of the referenced data collection phase in the requested scope. In order to avoid the above mentioned distinction, you can always call *IS88ExecutionService.determinePhaseDataReference()* without making a detour on reading the corresponding parameter value of *MESParamDatProvDef<version>* first.

Step 3

Get the collected phase data and the associated attribute descriptors:

```
List<IPhaseDataAttributeDescriptor> referencedPhaseAttributeDescriptors =
                                    dataProvider.getAttributeDescriptors();
IMESChoiceElement scope = MESChoiceListHelper.getChoiceElement(IPhaseDataProvider.
                  SCOPE_CHOICE_LIST_NAME, dataProviderDefinition.getDataScope());
List<PhaseDataRow> referencedPhaseData = dataProvider.getData(scope, getRtPhase());
```

- During execution, the phase data is represented by a list of phase data rows. A phase data row contains all values from one data collection phase instance (run). Example: Display of 3 runs with the *Measured Value*, *Boolean Value*, *Measured Value*, and *Option Value* data types.

10 g	True	20 g	Option 1
20 g	False	35 g	Option 1
14 g	False	21 g	Option 2

  The phase data attributes are associated with their corresponding attribute descriptors. The attribute descriptors provide meta information of the corresponding attributes, e.g. the data type of the attribute values and a method to convert an attribute value to its localized or recipe-specific representation, respectively.

Technical Details

When working with data collection and data representation phases, you should be familiar with the following technical details:

- **Show values** (data representation) phase of PharmaSuite:
  Besides the data columns, the **Show values** phase expects two additional columns. The first column contains the operation run information, the last (second) column contains the timestamp and signature information of the corresponding **Get values** phase.
  If you use your data collection phase in conjunction with the PharmaSuite **Show values** phase, be aware of this fact and design your data collection phase accordingly. This also applies to the data provider of your data collection phase. The *getAttributeDescriptors()*, *getColumnDescriptors()*, and *createObjectRecordFromStringRecord()* methods are affected (see "Implementing the Data Provider to Be Used by a Data Representation Phase" (page 7)). The *AbstractPhaseDataProvider* class offers support for the two additional columns, see *getColumnDescriptorsForInstanceCount()*, *getColumnDescriptorsForPhaseCompletionInfo()*, *getPhaseCompletionInformationFromStringRecord()*, and *getInstanceCountFromStringRecord()* methods.
  Please note that the *AbstractPhaseDataProvider* class provides methods for the

two additional columns. However, the class does not use them itself, i.e. this base class can be used also for implementing phase data providers where the collection phase does not provide the two additional columns.

- The default implementation of the data provider (*AbstractPhaseDataProvider*) supports the following scopes:
  - *EnumPhaseDataProviderScope.WITHIN_SAME_OPERATION*:
    Collects data only from the current operation, i.e. the data collection phase runs within the same (runtime) operation as the referenced data representation phase.
    Hence, the result is always empty when a data representation phase references a data collection phase from a parallel operation.
  - *EnumPhaseDataProviderScope.ACROSS_ALL_OPERATIONS*:
    Collects data from all (runtime) operation instances in which the referenced data collection phase has run. This applies to operations of multiple ETO runs, operations within loops, or even operations available across reactivated unit procedures, etc.
- By default, PharmaSuite supports the following data types for use in the data provider by offering corresponding subclasses of *AbstractPhaseDataAttributeDescriptor*:
  - MeasuredValue
  - String
  - Boolean
  - OptionValue
- Limited support of dynamic expressions:
  In general, the configuration of process parameters as part of recipe authoring is supported by the information flow mechanism, which includes the ability to define dynamic expressions.
  However, some limitations apply if a data representation phase is used to present data that previously has been collected by a data collection phase.
  In this case, the system only supports dynamic expressions as long as the result of each expression during execution (at runtime) is constant within a given process order/control recipe. This applies to all phases, process parameters, and attributes of a data collection phase that are also (indirectly) passed to a data representation phase by means of its data provider and which are relevant to the corresponding phase data attribute descriptors (i.e. instances of *IPhaseDataAttributeDescriptor)*.
  Example: In *BooleanPhaseDataAttributeDescriptor*, the display texts for **true** and **false** are defined in the process parameter attributes that contain dynamic expressions. If the data provider of your data collection phase uses *BooleanPhaseDataAttributeDescriptor*, then the expressions should be constant within a given order/control recipe.

Chapter 3

Adapting Trigger Phases

This chapter applies to the **Time-base trigger** and **Counter-based trigger** phases. You will learn how to adapt a trigger phase.

A trigger phase, as its name implies, triggers the creation of runs of an event-triggered operation (ETO instances). The underlying master recipe links one or more ETO template operations to a trigger phase.

For more information on trigger phases, see also "Functional Requirement Specification IPC Phases" [A8] (page 27) and "User Manual IPC Phases" [A9] (page 27).

TIP

Java classes of phases end with a version number, e.g. *AbstractTriggerPhasePlugIn0100* for version 1.00. In the following description, *AbstractTriggerPhasePlugIn<version>* is used.

Technical Overview

PharmaSuite provides a framework that allows system integrators to implement their own server-run trigger phases.
The framework is based on a generic abstract trigger phase executor that implements the generic life cycle of a server-run trigger phase.

The generic trigger phase observes the ETO templates connected with the trigger phase. As soon as the first ETO template is available for processing, trigger events can be created. After an ETO template has become available for processing, the trigger phase completes itself if at any later point in time there is no ETO template available for processing anymore. (This is even the case if some of the ETO templates that were triggered by the phase have not been available for processing yet.)
The generic trigger phase maintains a recipe-defined timeout. The timeout defines the maximum tolerated duration between phase start and the activation of the first ETO template. If the timeout expires, a system-triggered exception is recorded and the trigger phase is completed automatically. If no timeout is configured, a default of 30 minutes is used. This helps to prevent locked trigger operations in case there is no ETO template.

The API of the generic trigger phase provides a method to trigger the creation of the ETO instances. In addition, the generic trigger phase hides most of the technical and infrastructural details of a server–run phase.
As a trigger phase developer, you can focus on the algorithm that determines when the

creation of ETO instances shall be triggered. To realize the algorithm, you have to implement the trigger phase plugin (page 16).

The diagrams show an overview of the classes involved in the trigger phase framework. A phase developer must implement the *MyPlugin*, *<ModelType>*, and *MyTriggerPhaseExecutor* classes.

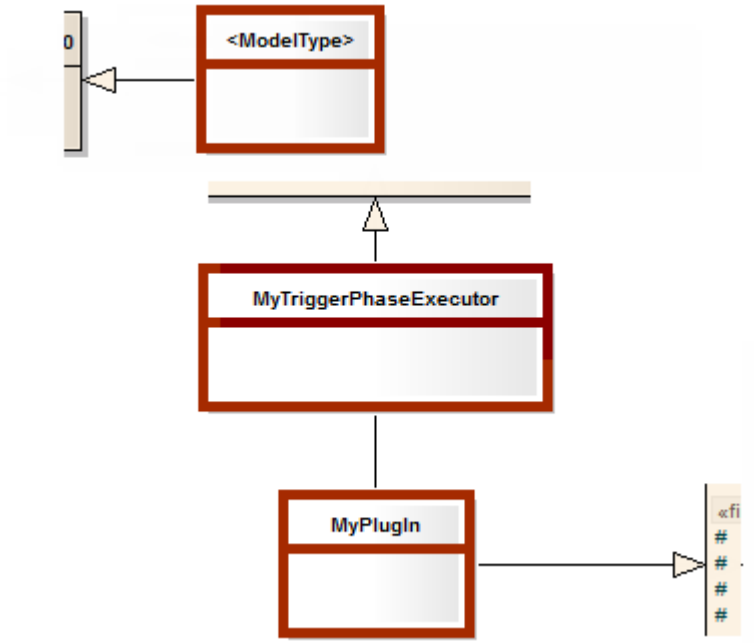

Figure 1: High-level overview

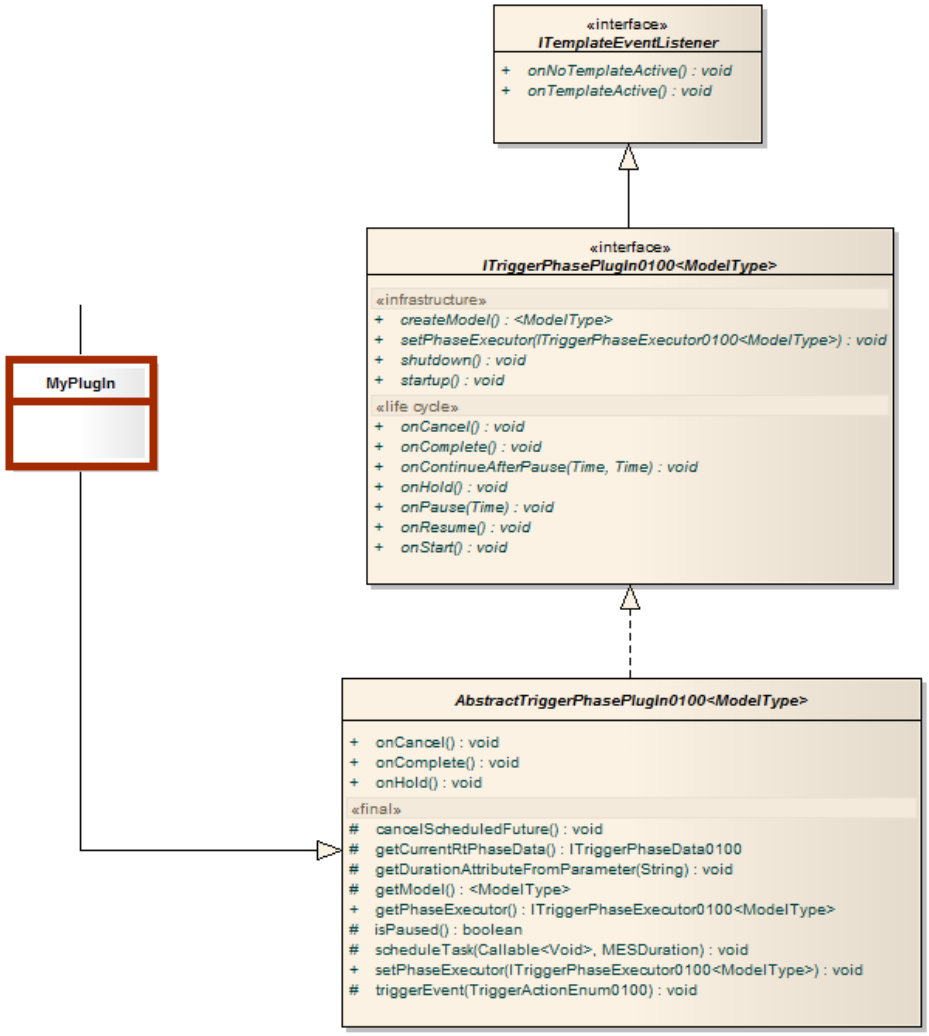

Figure 2: MyPlugin class

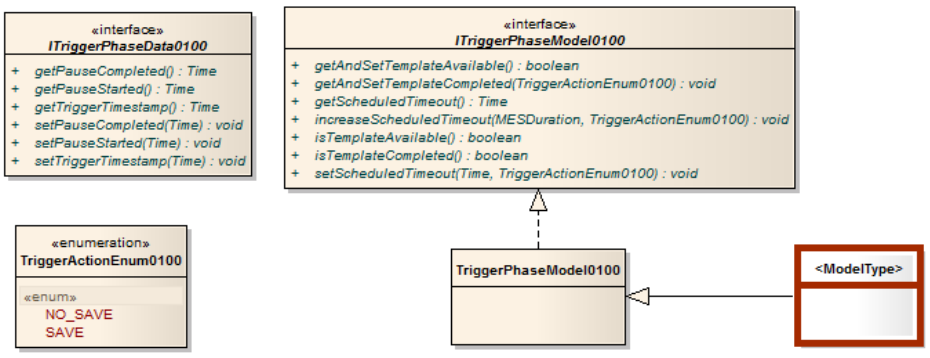

Figure 3: <ModelType> class

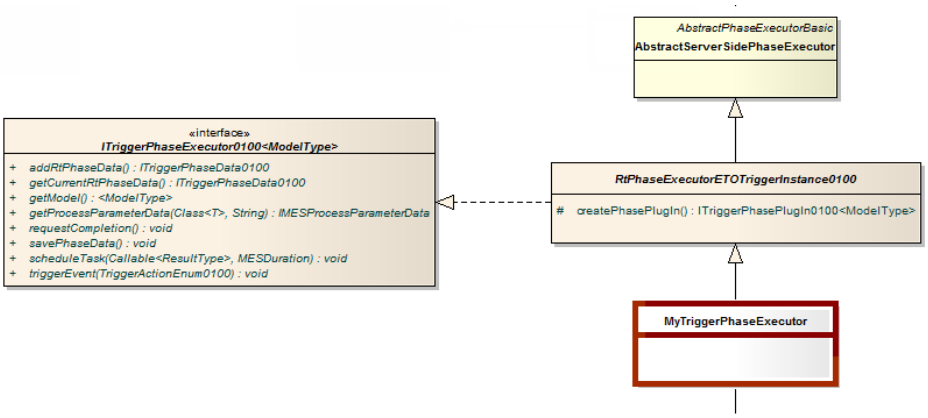

Figure 4: MyTriggerPhaseExecutor class

The Plugin Class

Responsibility and Concept

The trigger phase plugin class contains the algorithm to determine **when** to trigger the creation of ETO instances. In most use cases, the plugin has to **wait** for certain external events.

The implementation of the **wait** is the core of the plugin. It must be implemented without blocking. All server-side phases run within a fixed-size thread pool. That means blocking plugins may prevent processing of other phases. Therefore it is important to observe the following design rules:

- The plugin must not perform long running processing tasks.
- The plugin must not block.

Instead, the plugin should implement a polling task using a scheduled executor provided by the framework. The polling task checks the trigger condition, triggers the creation of ETO instances, and finally reschedules the task for future execution.

Implementation

The trigger phase plugin class must implement the *ITriggerPhasePlugIn<version>* interface. You can use the *AbstractTriggerPhasePlugIn<version>* class as base class for your plugin implementation. This abstract class provides a default implementation for most interface methods. In addition, it contains a set of useful helper methods (page 19).

When implementing the life-cycle methods, keep in mind that they need to be multi-thread-aware, because of the event-driven nature of the phase.

The interface methods fall in one of the two categories:

- Infrastructural or technical methods (page 17)
- Life-cycle methods (phase start (page 17), business events (page 18), phase termination (page 18))

Infrastructural or Technical Methods

void setPhaseExecutor(ITriggerPhaseExecutor<version><ModelType> executor)
The method is the setter of the phase executor to be used by the plugin class. It is used by the generic trigger phase executor after instantiating the plugin. If you use the *AbstractTriggerPhasePlugIn<version>* as base class of your plugin, you do not need to implement the method because the *AbstractTriggerPhasePlugIn<version>* already provides a final implementation.

void startup()
The method is both a technical and a life-cycle method. It is invoked each time the phase is started or resumed and before any of the life-cycle methods. We recommend to use the method to read process parameter data and to initialize other resources the plugin may need.

void shutdown()
The method is both a technical and a life-cycle method. It is invoked once the current instance of the phase executor is shut down. No other life-cycle method is called after this method. It is the counterpart of *startup()*.

General Information on Life-cycle Methods

All life-cycle methods are invoked by the generic trigger phase executor. You should not invoke any of them from the plugin code.

All life-cycle methods are invoked after *startup()* and before *shutdown()*. We recommend to make all life-cycle methods multi-thread-safe by synchronizing them.

Life-cycle Methods Related to Phase Start

void onStart()
When the method is invoked you can assume that the phase is fully initialized, i.e. *startup()* has already been called. The phase starts doing business for the first time. The method is called only once during the life cycle of the phase.

void onResume()
The method is similar to *onStart()*. It is invoked instead of *onStart()* each time the phase is resumed. We recommend to use the method to determine the current internal state of the phase and to start processing the business logic based on that state. Keep in mind that the last known internal state may not be accurate anymore. For instance, the runtime unit procedure may have been paused, while the phase was held.

Life-cycle Methods Related to Business Events

void onTemplateActive()
The method is invoked only once during the life cycle of the phase. It indicates that the first ETO template is available for processing. After the method was invoked, the trigger phase is able to trigger the creation of ETO instances.

void onNoTemplateActive()
The method is invoked only once during the life cycle of the phase. If the method is invoked, you can assume that *onTemplateActive()* was invoked before, but in case of resume within a different instance of the phase. The method indicates that no ETO template is available anymore. It is no longer possible to trigger the creation of ETO instances.
The framework of the trigger phase automatically completes the phase after *onNoTemplateActive()* returns.

void onPause()
The method is invoked each time the enclosing runtime unit procedure is paused. We recommend not to trigger the creation of ETO instances while the enclosing runtime unit procedure is paused.
Keep in mind that you will receive *onPause()* events only if they occur after *onStart()* or *onResume()*, respectively. But you can use the *isPaused()* helper method to determine the current pause status of the runtime unit procedure.

void onContinueAfterPause()
The method is invoked each time the enclosing runtime unit procedure continues working after the pause is ended. We recommend to continue triggering the creation of ETO instances.
The method is the counter part of *onPause()*. You will receive *onContinueAfterPause()* events only if they occur after *onStart()* or *onResume()*, respectively.

Life-cycle Methods Related to Phase Termination

The life-cycle methods listed below indicate the termination of the current instance of the phase. You can assume that *shutdown()* is called afterwards. The *AbstractTriggerPhasePlugIn<version>* provides a common non-empty implementation for all of the listed termination methods. This implementation cancels the last scheduled task. This assumes that your plugin schedules only one polling task at any point in time. If you follow that approach and use the *AbstractTriggerPhasePlugIn<version>.scheduleTask()* to schedule your tasks, there is no need to override the implementation of these methods.

void onHold()
The method is invoked if the phase is put on hold.

void onComplete()
The method is invoked if the phase is completed.

void onCancel()
The method is invoked if the phase is canceled.

HELPER METHODS

The *AbstractTriggerPhasePlugIn<version>* provides several helper methods that can be used to implement a plugin.

void scheduleTask(final Callable<Void> callable, final MESDuration delay)
The method schedules a task for future execution. If you let your plugin implement the *Callable<Void>* interface, you can pass the plugin instance as first argument.

void cancelScheduledFuture()
The method cancels a task scheduled for future execution. It is supposed to be the last polling task you have scheduled with *scheduleTask()*. Although the termination methods (*onHold()*, *onComplete()*, and *onCancel()*) invoke the method for the sake of good housekeeping, you should also invoke the method whenever you think the execution of scheduled tasks is no longer necessary. Consider the *onPause()* and *onNoTemplateActive()* life-cycle-events as candidates to invoke *cancelScheduledFuture()*.

void triggerEvent(final TriggerActionEnum<version> doAfter)
The method triggers the creation of ETO instances. For each ETO template that is supposed to be triggered by the trigger phase and that is active at the time this method is invoked, an ETO instance creation is triggered. If the creation of ETO instances fails, for example in case the Triggered Operation Management (TOM) server is down, a system-triggered exception is recorded. In addition, for each invocation of the method, with at least one ETO instance creation being triggered, the trigger timestamp is populated into the current phase data record. Be aware that the phase data record is unsaved (see *ITriggerPhaseData<version> getCurrentRtPhaseData()* below).

boolean isPaused()
The method determines if the runtime unit procedure of the phase is currently paused. It may imply a middle tier call to retrieve an up-to-date pause status.

MESDuration getDurationAttributeFromParameter(final String attributeName)
The method determines the *MESDuration* for process parameters of the **Duration** type. If you use process parameters of the **Duration** type to define polling events, think about the lowest value that makes sense. Even if you tend to allow very small durations to achieve high accuracy and precision, we recommend to consider the following aspects:

- A trigger phase triggers the creation of ETO instances and each ETO instance is supposed to be processed by a human operator. Hence, do not create ETO instances faster than they can be processed.
- Consider that it takes some time before a human operator can react after an ETO instance has become available at his Production Execution Client.

- High frequent polling tasks may create high load on all involved resources (e.g. CPU, I/O, and network traffic).
- Within a productive system there can be hundreds of trigger phases running at the same time.

final ITriggerPhaseExecutor<version><ModelType> getPhaseExecutor()
The method provides access to the phase executor. You may need the phase executor to create the model and to record system-triggered exceptions. Otherwise you should avoid to access the phase executor directly.

ModelType getModel()
The method provides access to the current model instance used by the trigger phase.

ITriggerPhaseData<version> getCurrentRtPhaseData()
The method provides access to the latest unsaved phase data record. If this is the first call or if the phase data has been saved before, a new and empty phase data record is created.

The Model Class

The trigger phase *<ModelType>* class implements the *ITriggerPhaseModel<version>* interface. The basic implementation as needed by the generic trigger phase is implemented by the *TriggerPhaseModel<version>* class.

The model of the trigger phase holds the internal status of the phase. It is a set of properties, which can be persisted. The *TriggerPhaseModel<version>* class uses the concept of the runtime phase context (*RtPhaseContext*). The *RtPhaseContext* works similar to the *RtOperationContext* used for Dispense phases (see "Technical Manual Phases of the Dispense Package" [A10] (page 27)). The runtime phase is the context holder of the *RtPhaseContext*, in contrast to the runtime operation in the case of the *RtOperationContext*. The *RtPhaseContext* may be deleted by the system after phase completion.

Using the *RtPhaseContext* to store the life-cycle status of a runtime phase has significant advantages:

- The PharmaSuite framework automatically loads and saves context data.
- Adding new properties is easy. You only need to implement a property's getter and setter methods within your model class. See code example in Step 7 in section "Creating a Trigger Phase" (page 22).
- When adding a new property, there is no need to create new ATDefinitions or run the phase generator.
- Phase-related runtime data is not stored in the phase data records. That means that the phase data records are not used for data containing only life-cycle status information, which does not belong to the executed batch record/report.

Methods

The *TriggerPhaseModel<version>* class basically contains the getter and setter methods for the properties. The setter methods contain a second argument. It indicates whether the model is saved immediately after the set action. The setter methods in particular are all invoked by the generic trigger phase. The following model methods are available:

Time getScheduledTimeout()
The getter method for the scheduled timeout. If the phase reaches the scheduled time without seeing an available ETO template, the phase is considered to have timed out.

void setScheduledTimeout(Time value, TriggerActionEnum<version> doAfterSet)
The setter method for the scheduled timeout.

void increaseScheduledTimeout(MESDuration duration, TriggerActionEnum<version> doAfterSet)
The method increases a previously set scheduled timeout by adding a duration.
The generic trigger phase increases the timeout if the unit procedure is paused. The method assumes that the scheduled timeout was set before.

boolean getAndSetTemplateAvailable(TriggerActionEnum<version> doAfterSet)
The method reads the current value of the **template available** property and sets the value to **true** if it has not been set before. Read and write operations are atomic.
The generic trigger phase invokes this method if an ETO template becomes available.

boolean isTemplateAvailable()
The method determines the value of the **template available** property.

boolean getAndSetTemplateCompleted(TriggerActionEnum<version>doAfterSet)
The method reads the current value of the **template completed** property and sets the value to **true** if it has not been set before. Read and write operations are atomic.
The generic trigger phase invokes this method if no ETO template is available anymore.

boolean isTemplateCompleted()
The method determines the value of the **template completed** property.

The Trigger Phase Class

The *MyTriggerPhaseExecutor* class implements the phase executor of a trigger phase. Your trigger phase extends the generic *RtPhaseExecutorETOTriggerInstance<version>* trigger phase executor. Within your extension, you only need to provide the factory method that creates your specific plugin. For details, see Step 5 in section "Creating a Trigger Phase" (page 22).

Creating a Trigger Phase

This section describes how to create a trigger phase.
The code examples are quite generic, but you have to replace the names indicated by red font.

Step 1

Define the specific process parameters in addition to the parameters used by the generic trigger phase:

- Timeout period
  - Type: Duration
  - Defines the maximum tolerated duration between phase start and the activation of the first ETO template.
- Timeout exception
  - Type: Exception
  - Defines the risk level and exception text used for the system-triggered exception.

Step 2

Define the phase data to be recorded in addition to the phase data used by the generic trigger phase:

- triggerTimeStamp
  - Type: DateTime
  - Contains the timestamp of the trigger. Each time an ETO instance creation is triggered by the trigger phase, a new phase data record is created.
- pauseStarted
  - Type: DateTime
  - Documents the start of a pause. It is written together with **pauseCompleted**.
- pauseCompleted
  - Type: DateTime
  - Documents the end of the pause.

Step 3

Generate the phase by running the phase generator.

Step 4

Adapt the phase data class to ensure that the data implements *ITriggerPhaseData<version>*.

```
public class MESRtPhaseDataMyTrigger<version> extends
             MESGeneratedRtPhaseDataMyTrigger<version> //
  implements ITriggerPhaseData<version> {
...
}
```

Step 5

Replace the generated phase executor using the following code as template.

```
public class MyTrigger<version> extends
             RtPhaseExecutorETOTriggerInstance<version><MyModel<version>> {
  public MyTrigger<version>(IPhaseCompleter inPhaseCompleter, IMESRtPhase rtPhase) {
    super(inPhaseCompleter, rtPhase);
  }
  @Override
  protected ITriggerPhasePlugIn<version><MyModel<version>> createPhasePlugIn() {
    return new MyPlugIn<version>();
  }
}
```

Step 6

Implement the *MyPlugIn<version>* plugin class.
For details see "The Plugin Class" (page 16).

Step 7

Implement the *MyModel<version>* model class. Example of a model class:

```
public class MyModel<version> extends TriggerPhaseModel<version> {
  public MyModel<version>(final ITriggerPhaseExecutor<version> executor) {
    super(executor);
  }
...
}
```

The model class typically contains bean properties. Within PharmaSuite phases, the properties are implemented as follows:

- Define a private enum(eration) that contains the properties. Implement a *getValue()* and *setValue()* method for each **enum** element. Both methods encapsulate the access to the *RtPhaseContext*. Make sure to use the properly typed access methods (*getTimeProperty()*, *setTimeProperty()*). The available methods can be found in the *AbstractRtEntityContext* class. If you need additional methods, please follow the description the operation context. For details, see "Technical Manual Phases of the Dispense Package" [A10] (page 27).

```
enum Property implements IContextProperty {
  /** The scheduled trigger timestamp. */
  SCHEDULED_TRIGGER_TIME {
    @Override
```

```
    public Time getValue(final TriggerPhaseModel<version> model) {
      return model.getContext().getTimeProperty(name());
    }

    @Override
    public <T> void setValue(final TriggerPhaseModel<version> model, final T value) {
      model.getContext().setTimeProperty(name(), (Time) value);
    }
  };
  /** Another property. */
  ANOTHER_PROPERTY {
    ...
  };

  RtPhaseContext getContext(final ITriggerPhaseExecutor<version> executor) {
    return executor.getRtPhaseContext();
  }
}
```

- Implement getter and setter methods for each property. The access to each property is synchronized.

```
public Time getScheduledTriggerTime() {
  synchronized (Property.SCHEDULED_TRIGGER_TIME) {
    return Property.SCHEDULED_TRIGGER_TIME.getValue(this);
  }
}

public void setScheduledTriggerTime(final Time value, final
                                    TriggerActionEnum<version> doAfterSet) {
  synchronized (Property.SCHEDULED_TRIGGER_TIME) {
    Property.SCHEDULED_TRIGGER_TIME.setValue(this, value);
  }
  saveAfterSet(doAfterSet);
}
```

Step 8

Implement a scriptlet.
If you plan to design a phase-specific report for your trigger phase, use the *TriggerPhaseScriptlet<version>* scriptlet. It contains some helpful methods:

long getNumberOfTriggerEvents(IBatchProductionRecordDocumentWrapper wrapper, String phaseID)
The method returns the number of trigger events. It counts the phase data records with a not-null trigger timestamp.

List<PauseEvent> getPhasePauseCollection(IBatchProductionRecordDocumentWrapper wrapper, String phaseID)
The method extracts a list of *PauseEvents* from the phase data records of the trigger phase. For each phase data record with a not-null **pauseStarted** or not-null **pauseCompleted** attribute, a *PauseEvent* is added to the result list. A *PauseEvent* is a comparable bean class that holds the **pauseStarted** and **pauseCompleted** properties. The list is sorted according to the ascending *{pauseStarted, pauseCompleted}* tuple.

TIP

The **Time-based trigger** and **Counter-based trigger** phases create only complete *PauseEvents*, i.e. both **pauseStarted** and **pauseCompleted** are not null.

Step 9

Design the phase-specific report.

Reference Documents

The following documents are available from the Rockwell Automation Download Site.

No.	Document Title	Part Number
A2	PharmaSuite Technical Manual Configuration & Extension - Volume 1	PSCEV1-GR008E-EN-E
A3	PharmaSuite Technical Manual Configuration & Extension - Volume 2	PSCEV2-GR008E-EN-E
A4	PharmaSuite Technical Manual Configuration & Extension - Volume 3	PSCEV3-GR008E-EN-E
A5	PharmaSuite Technical Manual Configuration & Extension - Volume 4	PSCEV4-GR008E-EN-E
A6	PharmaSuite Technical Manual Configuration & Extension - Volume 5	PSCEV5-GR005E-EN-E
A7	PharmaSuite Technical Manual Developing System Building Blocks	PSBB-PM007E-EN-E
A8	PharmaSuite Functional Requirement Specification IPC Phases	PSIP-RM003E-EN-E
A9	PharmaSuite User Manual IPC Phases	PSIP-UM003D-EN-E
A10	PharmaSuite Technical Manual Phases of the Dispense Package	PSDI-PM005E-EN-E
A11	PharmaSuite Technical Manual Administration	PSAD-RM008E-EN-E

TIP

To access the Rockwell Automation Download Site, you need to acquire a user account from Rockwell Automation Sales or Support.

Appendix A

Revision History

The following table describes the history of this document.

Changes related to the document:

Object	Description	Document
---	---	---

Changes related to "Introduction" (page 1):

Object	Description	Document
---	---	---

Changes related to "Adapting Data Collection and Data Representation Phases" (page 5):

Object	Description	Document
---	---	---

Changes related to "Adapting Trigger Phases" (page 13):

Object	Description	Document
The Plugin Class Implementation Helper Methods (page 19)	If the creation of ETO instances fails, for example in case the Triggered Operation Management (TOM) server is down, a system-triggered exception is recorded.	1.0

Index