ROBOTICS

Application manual

Production Framework

Trace back information:
Workspace 23B version a10
Checked in 2023-06-19
Skribenta version 5.5.019

Application manual
Production Framework
Version 1.1

Document ID: 3HAC085600-001
Revision: A

The information in this manual is subject to change without notice and should not be construed as a commitment by ABB. ABB assumes no responsibility for any errors that may appear in this manual.

Except as may be expressly stated anywhere in this manual, nothing herein shall be construed as any kind of guarantee or warranty by ABB for losses, damage to persons or property, fitness for a specific purpose or the like.

In no event shall ABB be liable for incidental or consequential damages arising from use of this manual and products described herein.

This manual and parts thereof must not be reproduced or copied without ABB's written permission.

Keep for future reference.

Additional copies of this manual may be obtained from ABB.

Original instructions.

Table of contents

Overview of this manual ... 7

1 Introduction to Production Framework 9

1.1 System overview ... 9
1.2 Orders ... 10
1.3 The production loop ... 11
1.4 Events ... 13
1.5 Component architecture ... 14
1.5.1 Component: Order Library ... 15
1.5.2 Component: Event Library ... 16
1.5.3 Component: Order Controller ... 17
1.5.4 Component: Logger ... 18
1.5.5 Component: Progress Tracer ... 19

2 Installation 21

2.1 Prerequisites ... 21
2.2 Installation options ... 22

3 Default components 23

3.1 Default Order Library ... 24
3.2 Default Event Library ... 28
3.3 Default Order Controller ... 30
3.3.1 Input signals ... 33
3.3.2 Output signals ... 35
3.3.3 Example PLC interaction sequences ... 38
3.4 Default Logger ... 40
3.5 Default Progress Tracer ... 42

4 Reference: Base framework RAPID instructions 43

4.1 PFAbortCycle – Aborts the current cycle ... 43
4.2 PFClearQueue – Clears the order queue ... 44
4.3 PFCurrentOrder – Retrieves information about the current order ... 45
4.4 PFCurrentState – Retrieves information about the current production loop state ... 47
4.5 PFEngine – Starts the production loop ... 48
4.6 PFEngineError – Generates an engine error ... 50
4.7 PFEngineErrorCode – Retrieves the last generated engine error code ... 52
4.8 PFExitEngine – Exits the production loop ... 53
4.9 PFIsAborted – Checks whether the current order is aborted ... 54
4.10 PFLastOrderCompletion – Retrieves information about the latest order ... 55
4.11 PFLog – Logs a message ... 57
4.12 PFPlaceOrder – Places an order ... 58
4.13 PFProgressReport – Generates a Progress Tracer report ... 61

5 Base framework RAPID data types and constants 63

5.1 PFBaseState – Production loop state ... 63
5.2 PFEventType – Event type for production loop / system events ... 64
5.3 PFLogLevel – Log level for filtering logs ... 66
5.4 PFOrderCompletionState – Completion state of last order placed ... 67
5.5 PFProgressReportType – Identifier for progress reports ... 68
5.6 PFProgressSource – Source of progress reports ... 72
5.7 PFResult – General response data type ... 73
5.8 Engine error codes ... 74

6 Miscellaneous 77

6.1 Running custom code when PFEngine is called ... 77

6.2 Running custom code before/after any event type 78
6.3 Setting Idle state cyclic event interval 79
6.4 Disabling the Progress Tracer component 80

7 PFView - FlexPendant interface 81

8 Advanced: Custom components 83

8.1 Templates 83
8.2 Replacing a component 84

Overview of this manual

About this manual

This manual describes the option Production Framework for RobotWare 7 with OmniCore.

It contains information regarding both basic out-of-the-box usage and more detailed information on how to customize certain functionality by replacing framework components with custom implementations.

Who should read this manual?

This manual is intended for:

- Robot programmers
- System integrators
- Function package developers

Prerequisites

The reader should have a basic knowledge of:

- RAPID programming
- System parameter configuration

For more advanced usage, such as implementing custom framework components, a more advanced knowledge of RAPID programming might be useful.

References

Reference	Document ID
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC065038-001*

Revisions

Revision	Description
A	Released with RobotWare 7.10.

This page is intentionally left blank

1 Introduction to Production Framework

1.1 System overview

What is Production Framework?

In short, Production Framework is a customizable modular platform for order based external control of an ABB robot.

It shares some features with the older product Production Manager but is generally more focused on providing a flexible and customizable platform rather than a provided-as-is fixed solution.

The main purpose of the framework is to handle orders from an external source, typically a PLC in charge of managing the various equipment in the cell. These orders are then executed by the framework by running user-specified RAPID routines.

Base framework features

The framework has built-in support for:

- Safely transferring orders from the TRAP execution level to normal execution level
- A state-based production loop
- Events, which can be used by the programmer to run code at various times in the production loop, or when certain system events happen
- Multi-tasking and MultiMove abstraction layer for easier (compared to using the basic RAPID API) synchronization of orders and events that are running on several RAPID tasks
- Aborting orders
- Enqueueing orders
- Customizable order constraints
- General logging
- Traceability
- Running independently on any RAPID task, including background tasks

Customization

To facilitate the usage of the framework in any system, some behavior – such as the PLC protocol used to control the orders, or the way how orders are defined – can be changed by replacing framework *components*. These custom components are typically implemented as RAPID modules and are using a well-defined and documented interface towards the base framework.

A set of *default components* are provided as part of Production Framework and can optionally be installed. For many use cases, these basic variants will be sufficient.

For more information regarding the component architecture, see *Component architecture on page 14*.

1.2 Orders

Orders

Orders are what drive Production Framework.

An order is the intent to have the robot do a specific pre-defined job without loading a separate RAPID program. The origin of the orders is decided by the implementation of the *Order Controller* component, but typically the orders come from an external source. The default Order Controller component takes in orders from the I/O system using a signal protocol. A PLC (*Programmable Logic Controller*) is often used to give these orders.

Using a custom implementation of the Order Controller component, orders could come virtually from anywhere. For example from a network socket, from user interaction with the teach pendant, or from a RAPID background task reading an order schedule from a file.

Each order from the Order Controller comes in the form of an *order code*. This code is used to fetch more detailed information about the order (from the Order Library component), e.g. the RAPID procedure that should be executed. More on this in *Component architecture on page 14*.

Service orders

There are two types of orders, normal orders and *service* orders. Normal orders are occasionally referred to as parts.

Service orders are typically used to run equipment maintenance routines, or orders of a more manual nature, such as putting the robot in a certain pose for inspection.

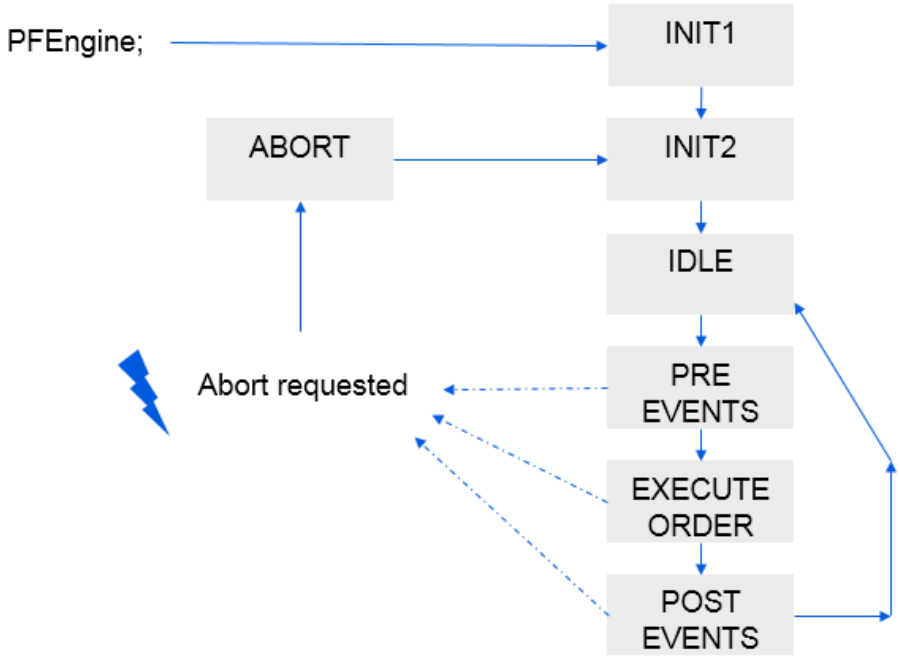
The order code for the two types can overlap, so it is permitted to have a normal order with the same order code as a service order. However, when placing a service order, the service status of the order is explicitly indicated by the Order Controller.

When orders are queued, service orders take precedence over normal orders.

1.3 The production loop

Traversing the states

Production Framework is a *state-based* framework. One of its most central concepts is a state machine which in this manual is referred to as the *production loop*.



xx1800002238

The framework is always in a single production loop state on each RAPID task. The tasks are independent from a framework point of view, which means that each task has its own instance of the framework, including a current production loop state.

Most states have one or more associated event types. User events of a certain type will be executed during the corresponding state. For more information about events, see *Events on page 13*.

Initially, after resetting the Program Pointer, the framework is in a *START* state. This state is active until the task calls the routine `PFEngine` to start the production loop engine.

When the engine is started, the state is changed to *INIT1*. By utilizing events, this state can be used to initialize equipment or run other code that needs to be executed before the framework is ready to receive its first order.

INIT1 is followed by *INIT2*, which is similar. However, the *INIT2* state will also be entered after an abort.

After the two initialization states, the production loop will enter the IDLE state and wait for orders. In all other states, the framework is considered busy. The framework will remain in the *IDLE* state until a new order is successfully received.

Continues on next page

When an order has been placed, the production loop will enter the *PRE-EXECUTION EVENTS* state. Here events will be fired, executing code that should be run before each order.

When all pre-execution events have been fired, the framework will continue to the state *EXECUTE ORDER*. As the name suggests, while in this state the actual order will be executed. The order corresponds to a user-defined RAPID procedure.

Following the order execution is a *POST-EXECUTION EVENTS* state, similar to the *PRE-EXECUTION EVENTS*. When all declared events in this state have been fired, the production loop goes back to the *IDLE* state, accepting new orders.

An *abort* can be requested, either directly from RAPID (see the routine `PFABortCycle` in *PFAbortCycle – Aborts the current cycle on page 43*, and `PFIsAborted` in *PFIsAborted – Checks whether the current order is aborted on page 54*), or by using the Order Controller (the default Order Controller has a digital input signal for this).

When an order has been aborted, the state will change to ABORT. When all events for this state have been fired, the state is changed back to *INIT2*.

Normally the `PFEngine` routine never finishes, hence the name production loop. However, it can be requested to exit itself by calling the `PFExitEngine` routine (see *PFExitEngine – Exits the production loop on page 53*). After exiting, the state will be *EXITED* until the engine is started again or the Program Pointer is moved to main.

The current state of the production loop can be retrieved by using the RAPID routine `PFCurrentState` (see *PFCurrentState – Retrieves information about the current production loop state on page 47*).

1.4 Events

User events

The user can declare *events* which will be fired either when some production loop states are entered, or when some situations arise in the system, e.g. the controller is stopped or the user changes controller operation mode. When fired, a user-specified RAPID procedure for each event is executed.

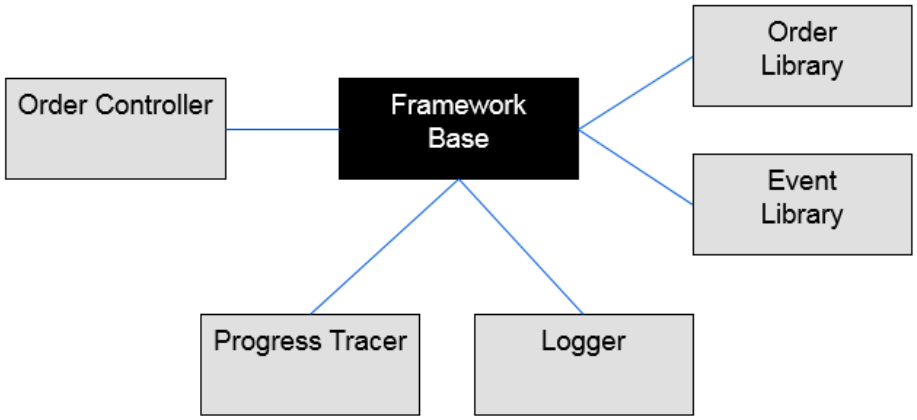
How events are declared depends on the Event Library component. The default Event Library allows the user to declare events as data variables in RAPID which will be automatically detected during runtime. The default Event Library is further described in *Default Event Library on page 28*.

A full list of available event types is available in *PFEventType – Event type for production loop / system events on page 64*.

1.5 Component architecture

Topology

As mentioned earlier, the framework consists of a base component and a handful of replaceable components. Each replaceable component has a default implementation that handles its area of responsibility in a predefined way. If this behavior cannot fulfill all requirements, it can be removed and replaced by a user implemented variant.



xx1800002239

Each default component has no direct dependency on any other component except on the base component. This design allows for easy component replacement. If custom components are implemented, nothing prevents them from having dependencies on each other, although it is recommended to avoid it if possible, since this practice might make component recycling more difficult.

The RAPID API used to communicate with the base component is well defined and component module templates with empty API routines can be can be installed to get started.

For more information on how to implement and use custom components, see *Advanced: Custom components on page 83*.

A short description of the replaceable component class responsibilities is listed below. The behavior of the default components is described in *Default components on page 23*.

Continues on next page

1.5.1 Component: Order Library

Responsibilities

The Order Library component is responsible for fetching order information.

Given an order code and order type (service or not) by the base component, the Order Library first confirms that the corresponding order is known, and then provide detailed information about that order.

This information includes a description, a name of the procedure that should be executed, and details regarding synchronization between several framework-managed RAPID tasks.

It is also the Order Library's responsibility to *validate* orders. This means deciding whether the framework is permitted to run an order *right now*. If the Order Library finds any reason to deny the framework permission to run the order, it can refuse and provide a reason as a string.

Each order will be validated twice, when it is placed, and then once again when the production loop enters the *EXECUTE ORDER* state.

For information about the default Order Library, see *Default Order Library on page 24*.

1.5.2 Component: Event Library

Responsibilities

The Event Library component is similar to the Order Library, except that it handles information regarding user events instead of orders.

It is the responsibility of the Event Library component to keep track of declared events, and in which sequencing they should be executed during firing of each event type.

For information about the default Event Library, see *Default Event Library on page 28*.

1.5.3 Component: Order Controller

Responsibilities

The Order Controller component is the front end to the framework during runtime.

It is the responsibility of the Order Controller to receive orders, most likely from outside the controller, and place them using the API on the base framework.

It also falls upon the Order Controller to handle feedback from the base framework and, if required, forward that information to the order giver. This feedback includes status updates regarding an order, such as whether it was successful or not. Maybe the order was refused or perhaps it could not be placed since the order code was unknown. Such information needs to be propagated to the PLC, user or other controlling mechanism.

For information about the default Order Controller, see *Default Order Controller on page 30*.

1.5.4 Component: Logger

Responsibilities

The Logger component is a generic logging facility which can be used by the user to do any kind of logging during runtime (via the base framework). See the routine `PFLog` in *PFLog – Logs a message on page 57*.

The Logger supports a set of *log levels*, e.g. *Information, Warning, Error, Debug* and *Progress*. Each log entry must use one of these defined levels. This information can then be used by to Logger component to filter or separate the different kinds of information.

The Progress log level is used by the default Progress Tracer component to log each progress report, see *Default Progress Tracer on page 42*.

For information about the default Logger, see *Default Logger on page 40*.

1.5.5 Component: Progress Tracer

Responsibilities

The Progress Tracer component has no intrinsic responsibility by itself. However, the base framework will periodically send *Progress Reports* to it, which a custom Progress Tracer can use for any purpose desired.

There are many reports generated by the base framework during the traversal of the production loop, and it is also possible for user code to generate reports via the base framework. See the routine `PFProgressReport` in *PFProgressReport – Generates a Progress Tracer report on page 61*.

The reports are similar to log entries, but each report type has a unique identifier, and all variable data are sent as variables. This enables a custom Progress Tracer to react and run code when specific things happen.

The Progress Reports could be considered as more fine-grained event types.

The default Progress Tracer does nothing, except logging the reports on the *Progress* log level. See *Default Progress Tracer on page 42*.

This page is intentionally left blank

2 Installation

2.1 Prerequisites

RobotWare options

Production Framework requires a license for:

- *3404-1 Production Framework*

The Production Framework add-in

Production Framework is not a part of the RobotWare distribution, instead it is distributed as a RobotWare add-in. This allows for independent release cycles not dependent on the RobotWare releases.

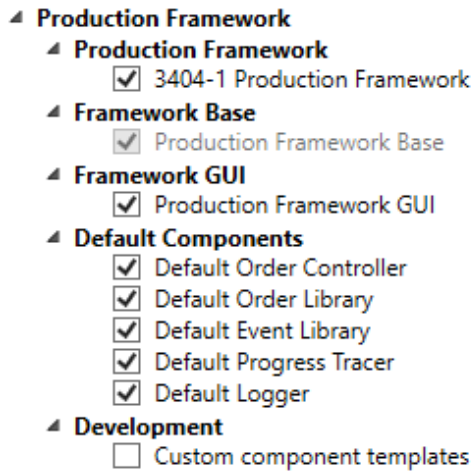
Any version of the product can be downloaded from the RobotStudio RobotApps Gallery.

During installation of the system with Installation Manager, add the downloaded add-in as a separate product next to RobotWare.

2.2 Installation options

Add-in installation options

Given that the Production Framework add-in has been downloaded and selected as an additional product in Installation Manager, the following options will be available under the **Applications** options tab.



xx2200001373

Option	Description
Production Framework	Required for all Production Framework setups.
Framework GUI	The optional FlexPendant application (see *PFView - FlexPendant interface on page 81*).
Default Order Controller	Default implementation of the Order Controller component (see *Default Order Controller on page 30*). Unless a custom Order Controller will be used, this option is required.
Default Order Library	Default implementation of the Order Library component (see *Default Order Library on page 24*). Unless a custom Order Library will be used, this option is required.
Default Event Library	Default implementation of the Event Library component (see *Default Event Library on page 28*). Unless a custom Event Library will be used, this option is required.
Default Progress Tracer	Default implementation of the Progress Tracer component (see *Default Progress Tracer on page 42*). Unless a custom Progress Tracer will be used, this option is required.
Default Logger	Default implementation of the Logger component (see *Default Logger on page 40*). Unless a custom Logger will be used, this option is required.
Custom component templates	If this option is selected, RAPID templates for custom components (see *Templates on page 83*) will be copied to HOME:/PFTemplates/ during installation. Keep in mind that existing files will be overwritten.

3 Default components

Overview

This section describes the *default* implementation of the required components for Production Framework. This set of components are distributed together with the base framework and can optionally be installed.

The default components are designed for a generic use case and can be used for many projects, but not all.

The default components have no direct dependencies on each other, so they can be used in any constellation combined with custom components.

For a short description of the responsibilities of each component, see *Component architecture on page 14*.

Continues on next page

3.1 Default Order Library

Short description

The default Order Library implementation handles order declarations by letting the user declare variables of the predefined `RECORD` datatypes `partdata` and `servicedata`.

The user only needs to declare these variables as global variables (`VAR` or `PERS`), and initialize them with relevant data. The default Order Library will find them automatically during runtime.

Order codes are restricted to numerical values.

The partdata datatype

The `partdata` datatype is used for declaring available normal (non-service) orders. The name is inherited from the older product *Production Manager*.

```
RECORD partdata
  dnum plcCode;
  string description;
  string procName;
  string tasklist;
  string validAt;
ENDRECORD
```

First, a `plcCode` is required. This is almost the same thing as an *order code* – a unique code for identifying this specific order. However, although the base framework supports using a `string` as order code, a `dnum` is used here instead. The default Order Library (and the default Order Controller) handles order codes as numeric values, not full strings. However, although the base framework supports using a string as order code, a dnum is used here instead. The default Order Library (and the default Order Controller) handles order codes as numeric values, not full strings. This is to enable easier communication over an I/O signal protocol.

The `plcCode` element must be unique. If several order declarations share the same `plcCode`, an engine error (`PF_ERR_DUPLICATE_ORDER_NUM`) will be generated during runtime when the order is placed.

The `description` element should contain a useful description or name of the order.

The `procName` element shall contain the name of the RAPID procedure that should be called when executing the order.

If synchronization (at start and end of the order execution) of this order with another order on another RAPID task is required, the `tasklist` element shall contain a list of the participating RAPID tasks. The task names must be separated by either a ‘,‘ (comma) character, a ‘:‘ (colon) character or a ‘;‘ (semicolon) character. E.g. "`T_ROB1,T_ROB2`" or "`T_ROB1;MyTask1`". This list must be identical in all participating orders/tasks, including the sequencing of the task names.

Continues on next page

Note that for the synchronization to work, there must be orders declared in each participating task, which all shall run the framework. All tasks must start executing the orders by themselves. Only the synchronization is automatic.

If synchronization is not needed, `tasklist` should contain an empty string.

The `validAt` string is used for custom order validation, more about this below. If custom order validation is not used, then this argument has no effect, i.e. it can be set to an empty string.

An example declaration of two orders, here without synchronization:

```
MODULE MyModule
  VAR partdata myOrderA
    := [100,"My first order","MyProc1","",""];
  VAR partdata myOrderB
    := [200,"My second order","MyProc2","",""];

  PROC MyProc1()
    ! Code to run when executing the first order
  ENDPROC

  PROC MyProc2()
    ! Code to run when executing the second order
  ENDPROC
ENDMODULE
```

Note that the `partdata` declarations do not need to be in the same RAPID module as the procedure definitions.

The servicedata datatype

The `servicedata` datatype is very similar to `partdata`, although it is used for declaring service orders.

```
RECORD servicedata
  dnum plcCode;
  string description;
  string procName;
  string tasklist;
  string validAt;
ENDRECORD
```

All elements in the `servicedata` datatype have the same function as the corresponding elements in `partdata`.

Order validation

The default Order Library does not by itself support active validation as described in *Component: Order Library on page 15*. All orders will be accepted automatically.

However, it provides a simple possibility for customization without the need for a complete custom Order Library component.

Continues on next page

To take advantage of this functionality, declare a global RAPID routine with an argument list like the example below.

```
PROC MyCustomValidation(
  bool preCheck,
  string orderCode,
  bool service,
  string validAt,
  INOUT PFResult result,
  INOUT string refusedReason
  )

  VAR bool isValid;

  ! Validation code here
  ! ...

  IF isValid THEN
    ! Ok to run order
    result := PF_RESULT_OK;
  ELSE
    ! Order is refused, reason code 100
    refusedReason := "100"
    result := PF_RESULT_REFUSED;
  ENDIF
ENDPROC
```

Then, to activate this hook routine, set the global variable `pfExtValidationHook` to the name of the routine:

```
pfExtValidationHook := "MyCustomValidation";
```

This can be done from anywhere; the main routine, anytime during runtime, or preferably from the `PFInitEngineHook` routine described in *Running custom code when PFEngine is called on page 77*.

The arguments are:

- `preCheck`, a `bool` with the value `TRUE` if this is the validation request performed when the order is placed, before any events are executed. `FALSE` if this is the validation request performed after pre-execution events, right before the order is about to be executed.
- `orderCode`, the order code associated with the order.
- `service`, set to `TRUE` if the order is a service order, otherwise `FALSE`.
- `validAt`, from the corresponding field in the `partdata` or `servicedata` instance. This is used for the hook routine to make a validation decision. The value could be anything that fit into a string; a positioner station number, a tool state or anything other kind of restriction or combination of restrictions as defined by this customization.
- `result`, an `INOUT` variable where the result of the validation should be stored. This should be set to `PF_RESULT_OK` if the order is valid,

Continues on next page

PF_RESULT_REFUSED if not valid, or possibly PF_RESULT_ERROR if there is a problem with the validation process itself.

- refusedReason can optionally be used in combination with setting the result to PF_RESULT_REFUSED. The value should be a reason for refusing the order. Any string can be used. It is forwarded to the Order Controller component when the latter is notified that the order was refused. If the string can be parsed as an integer, the default Order Controller will set a group output signal to that number. See *Default Order Controller on page 30* for more information.

3.2 Default Event Library

Short description

The default Event Library implementation handles user event declarations by letting the user declare variables of the predefined `RECORD` datatype `eventdata`.

The user only needs to declare these variables as global variables (`VAR` or `PERS`), and initialize them with relevant data. The default Order Library will find them automatically during runtime.

The eventdata datatype

The evType element signifies the event type. This is what decides the condition for when the event should be fired. Available event types are listed in *PFEventType – Event type for production loop / system events on page 64*.

The `eventdata` datatype is used for declaring events of a specified event type.

```
RECORD eventdata
  PFEventType evType;
  dnum sequence;
  string description;
  string procName;
  string tasklist;
ENDRECORD
```

The `evType` element signifies the event type. This is what decides the condition for when the event should be fired. Available event types are listed in *PFEventType – Event type for production loop / system events on page 64*.

The sequence element is used to help the default Event Library with the sequence ordering in case several event declarations share the same event type. Declared events of the same event type will be fired according to sequence number, beginning on the lower end, progressing to higher values. Any positive number that can be held by a `dnum` can be used. Using identical numbers will result in undefined behavior, all of those events will be fired, but the sequencing is not guaranteed to be the same each time.

The `description` element should contain a useful description or name of the event.

The `procName` element shall contain the name of the RAPID procedure that should be called when firing the event.

If synchronization (at start and end of the order execution) of this event with another event on another RAPID task is required, the `tasklist` element shall contain a list of the participating RAPID tasks. The task names must be separated by either a ',' (comma) character, a ':' (colon) character or a ';' (semicolon) character. E.g. "`T_ROB1,T_ROB2`" or "`T_ROB1;MyTask1`". This list must be identical in all participating events/tasks, including the sequencing of the task names.

Note that for the synchronization to work, there must be events declared in each participating task, which all shall run the framework. Each task must meet the event type firing condition by itself. Only the synchronization is automatic.

Continues on next page

If synchronization is not needed, `tasklist` should contain an empty string.

An example declaration of two pre-execution events:

```
MODULE MyModule
  VAR eventdata myEventA
    := [PF_EV_PREPART,10,"EvA","FireEv1",""];
  VAR eventdata myEventB
    := [PF_EV_PREPART,5,"EvB","FireEv2",""];

  PROC FireEv1()
    TPWrite "A";
  ENDPROC

  PROC FireEv2()
    TPWrite "B";
  ENDPROC
ENDMODULE
```

Before a non-service order *(also called a part)* is executed, events of the type `PF_EV_PREPART` are fired. Both events declared above will be executed according to the `sequence` numbers, printing `B` and then `A`.

3.3 Default Order Controller

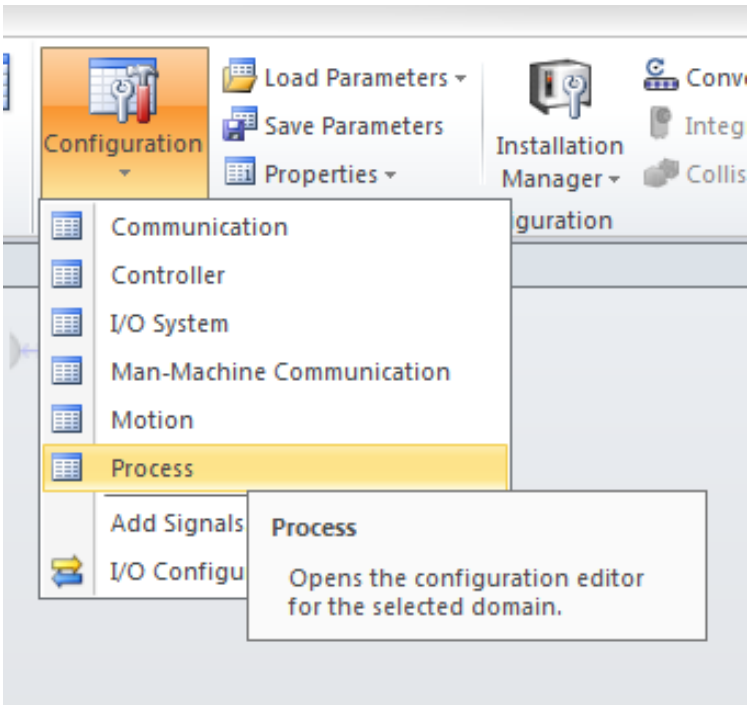
Short description

The default Order Controller implements a basic I/O signal protocol for taking orders from an external source. The signals used can be configured in the system parameters, topic *Process*.

Order codes are restricted to numerical values.

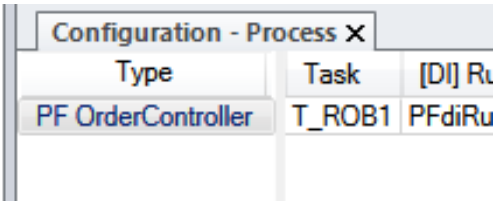
Configuration

To access the PROC configuration from RobotStudio, locate the **Configuration** button under the **Controller** ribbon tab. Click on **Process** as shown in the image below.



xx1800002240

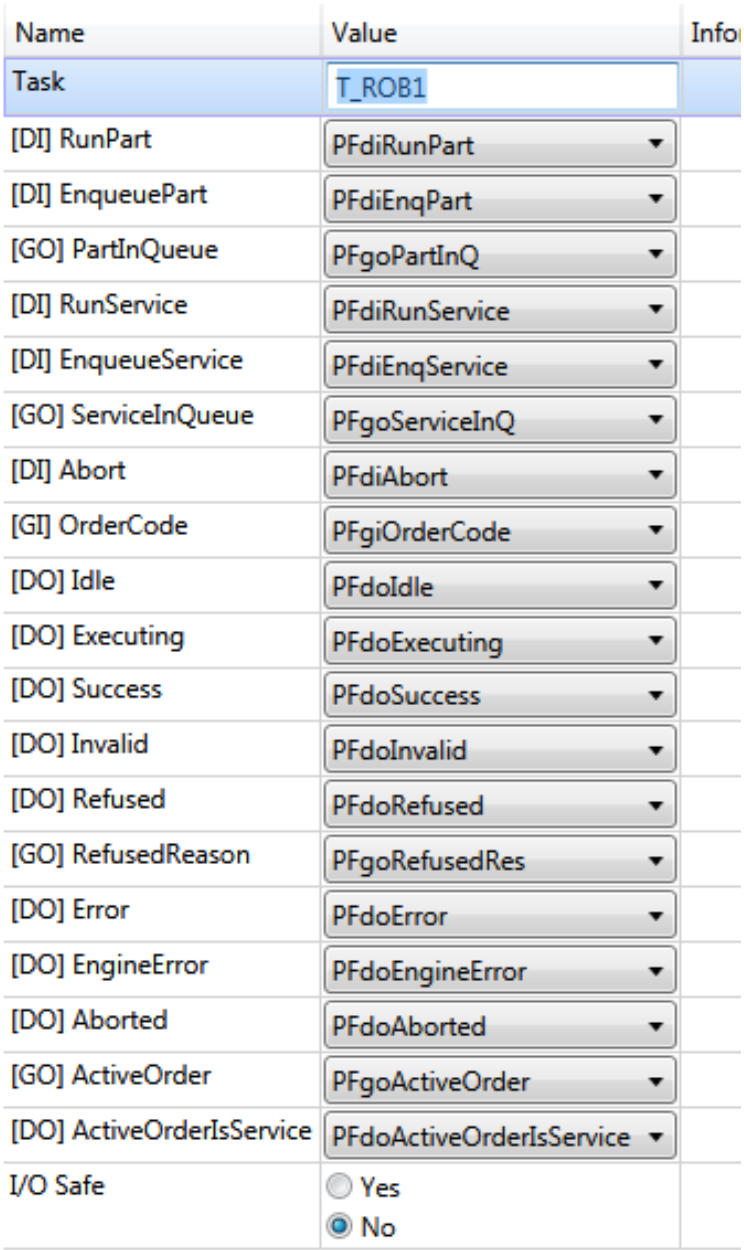
Select the type **PF OrderController**.



xx1800002241

Continues on next page

The default Order Controller needs a separate configuration for each task that should be managed by the framework. After installation, one configuration is automatically set up for the T_ROB1 task, using a set of default signals.



xx1800002242

These default signals can be mapped to a desired I/O device in the I/O System configuration.

Continues on next page

Alternatively, the default Order Controller can here be configured to use other signals.

Note

If using the framework on more than one task, each task needs its own configuration, and its own set of signals.

Note

The default set of output signals are configured to use the *PFOCSafeLevel* I/O safe level. This safe level is installed with the default Order Controller and will cause the output signals to keep their values while/after restarting the controller. *PFOCSafeLevel* can optionally be used on manually configured signals.

Other than signal selection, the configuration also has an option for *I/O Safe*. If set to *Yes*, input events from the input signals will be enqueued during program stop and executed when the program is resumed. A maximum of one event from each input signal will be enqueued.

If set to *No*, inputs are ignored when the program is stopped.

Continues on next page

3.3.1 Input signals

RunPart

Default signal: *PFdiRunPart*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request a normal order (part) to be placed. This signal should only be set when the Idle digital output is set.

In addition, status signals will be cleaned up, i.e. the signals **Success**, **Invalid**, **Error**, **Aborted**, **Refused** and **RefusedReson** will be set to 0.

The order number will be read from the *OrderCode* group input. See the output signals for the various results of making an order.

EnqueuePart

Default signal: *PFdiEnqPart*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request a normal order (part) to be placed. If the production loop is busy with another order, this order will be enqueued. Only one normal order can be in the queue at any given time. If another normal order already is in the queue, it will be replaced.

In addition, status signals will be cleaned up, i.e. the signals **Success**, **Invalid**, **Error**, **Aborted**, **Refused** and **RefusedReson** will be set to 0.

The order number will be read from the *OrderCode* group input. See the output signals for the various results of making an order.

RunService

Default signal: *PFdiRunService*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request a service order to be placed. This signal should only be set when the Idle digital output is set.

In addition, status signals will be cleaned up, i.e. the signals **Success**, **Invalid**, **Error**, **Aborted**, **Refused** and **RefusedReson** will be set to 0.

The order number will be read from the *OrderCode* group input. See the output signals for the various results of making an order.

EnqueueService

Default signal: *PFdiEnqService*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request a service order to be placed. If the production loop is busy with another order, this order will be enqueued. Only one service order can be in the queue at any given time. If another service order already is in the queue, it will be replaced. Enqueued service orders will be executed before any enqueued normal order (part).

Continues on next page

3.3.1 Input signals
Continued

In addition, status signals will be cleaned up, i.e. the signals **Success**, **Invalid**, **Error**, **Aborted**, **Refused** and **RefusedReson** will be set to 0.

The order number will be read from the *OrderCode* group input. See the output signals for the various results of making an order.

ClearQueue

Default signal: *PFdiClearQ*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request any enqueued orders (both service and part) to be removed from the queue. This means that they will not be executed.

Abort

Default signal: *PFdiAbort*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will request the current order to be aborted. During an abort, any enqueued orders will be cancelled. For more information about aborting a cycle, see the `PFAbortCycle` routine in *PFAbortCycle – Aborts the current cycle on page 43*.

OrderCode

Default signal: *PFgiOrderCode*

Type: Group input

This group input should be set to the order code (in numeric form) of the order, normal or service, that should be used when setting any of the digital inputs *RunPart*, *EnqueuePart*, *RunService* or *EnqueueService*.

ResetStatus

Default signal: *PFdiResetStatus*

Type: Digital input

Setting this digital input to 1 from a previous state of 0 will clean up the status signals, i.e. the signals **Success**, **Invalid**, **Error**, **Aborted**, **Refused** and **RefusedReson** will be set to 0.

This has no effect on the function of the framework, but is rather a conveniece for cases where the PLC need to reset these signals prior to requesting a new order.

3.3.2 Output signals

PartInQueue

Default signal: *PFgoPartInQ*

Type: Group output

If a normal order (part) is currently in the queue, its numeric order code will be outputted here.

ServiceInQueue

Default signal: *PFgoServiceInQ*

Type: Group output

If a service order is currently in the queue, its numeric order code will be outputted here.

Idle

Default signal: *PFdoIdle*

Type: Digital output

This digital output is set to 1 when the production loop is idle and ready to accept new orders. If the output is set to 0, the production loop is currently busy.

Executing

Default signal: *PFdoExecuting*

Type: Digital output

This digital output is set to 1 when the production loop is currently executing an order (not including post/pre-execution events). When the output is set to 0, the production loop is either idle or is executing events.

Success

Default signal: *PFdoSuccess*

Type: Digital output

This digital output is set to 1 when the last executed cycle was successful (from a framework point of view). It is set to 0 again when a new order is requested or when the **ResetStatus** signal is set to 1. Note that it is the last executed, not last ordered cycle. This difference is important if the queue functionality is used.

Invalid

Default signal: *PFdoInvalid*

Type: Digital output

This digital output is set to 1 when an order is requested, but the order code is invalid (unknown). It is set to 0 again when a new order is requested or when the **ResetStatus** signal is set to 1.

Refused

Default signal: *PFdoRefused*

Continues on next page

3.3.2 Output signals
Continued

Type: Digital output

This digital output is set to 1 when an order is refused by the Order Library component. This means that the order code exists, but some condition for executing it is not met. It is set to 0 again when a new order is requested or when the **ResetStatus** signal is set to 1.

Note that the default Order Library will never refuse an order.

RefusedReason

Default signal: *PFgoRefusedReason*

Type: Group output

This group output is set in combination with the digital output Refused. If the reason for refusal - as given by the Order Library - is a numerical value, it will be outputted here.

Error

Default signal: *PFdoError*

Type: Digital output

This digital output is set to 1 when an order is requested, and one of the following conditions arise:

- The production loop was not idle, and the queue functionality was not used.
- The base framework responded to the request in an unexpected way.

It is set to 0 again when a new order is requested or when the **ResetStatus** signal is set to 1.

EngineError

Default signal: *PFdoEngineError*

Type: Digital output

This digital output is set to 1 when an Engine Error is encountered. This can have various causes, but typically means that there is something wrong with the configuration of the framework. See also the `PFEngineError` routine in *PFEngineError – Generates an engine error on page 50*.

It is set to 0 again when the framework is restarted, and the Order Controller is initialized.

Aborted

Default signal: *PFdoAborted*

Type: Digital output

This digital output is set to 1 when an abort has been granted.

It is set to 0 again when a new order is requested or when the **ResetStatus** signal is set to 1.

ActiveOrder

Default signal: *PFgoActiveOrder*

Type: Group output

Continues on next page

This group output is set to the numerical order code of the currently active order.

ActiveOrderIsService

Default signal: *PFdoActiveOrderIsService*

Type: Digital output

This digital output is set in combination with *ActiveOrder*. If set to 1, the active order is a service order, if 0 a normal order (part).

If no order is active, this output is set to 0.

3.3.3 Example PLC interaction sequences

Example 1: Executing a part, simplified

1 The PLC has the intention to execute a part with order code **106**.
2 The PLC waits until the *Idle* digital output signal is set to 1. This means that it is OK for the PLC to issue an order.
3 The PLC sets the *OrderCode* input signal group to the binary representation of **106**, i.e. 1101010.
4 The PLC sets the *RunPart* digital input signal to 1.
5 The *Idle* signal is reset to 0. The digital status output signal *Success* is also reset to 0.
6 The PLC can now reset the *RunPart* signal to 0.
7 The part (and any pre/post execution events) is executed.
8 When the execution is completed, the *Idle* signal is once again set to 1. The *Success* signal is also set to 1.
9 The PLC can now issue next order (3.)

Example 2: Executing a part, detailed

1 The PLC has the intention to execute a part with order code **106**.
2 The PLC waits until the *Idle* digital output signal is set to 1. This means that it is OK for the PLC to issue an order.
3 The PLC sets the *OrderCode* input signal group to the binary representation of **106**, i.e. 1101010.
4 The PLC sets the *RunPart* digital input signal to 1.
5 The *Idle* signal is reset to 0. The status output signals *Success*, *Invalid*, *Error*, *Aborted*, *Refused* (digital) and *RefusedReson* (group) are also reset to 0.
6 The *ActiveOrder output* signal group is set to **106**. The *ActiveOrderIsService* is kept at 0, since this is not a service order.
7 The PLC can now reset the *RunPart* signal to 0.
8 Pre-execution events are executed.
9 The *Executing* digital output signal is set to 1.
10 The part is executed.
11 The *Executing* signal is reset to 0.
12 Post-execution events are executed.
13 The *ActiveOrder* and *ActinveOrderIsService* signals are reset to their initial values (0).
14 When the execution is completed, the *Idle* signal is once again set to 1. The *Success* signal is also set to 1.
15 The PLC can now issue next order (3.)

Continues on next page

More

It is near impossible to to have a complete reference of all possible interaction scenarios. Therefore, it is recommended for new users of the framework to carefully read the signal descriptions above, and then install a simple virtual system with Production Framework. Manual interaction – the user pretending to be a PLC - using virtual signals in the I/O view in RobotStudio can then be performed for effective learning.

3.4 Default Logger

Short description

The default Logger component writes logs to files. It has support for filtering the different log levels available, and can rotate between several files to avoid filling up storage space.

Configuration

The default Logger can be configured to log or discard log entries from each log level. This is done through a set of global Boolean variables:

- `pflogErrOn`
- `pflogWarnOn`
- `pflogInfoOn`
- `pflogDebugOn`
- `pflogProgressOn`

All log levels are disabled by default, and will be disabled again after moving the program pointer to main.

They can be enabled/disabled at any time. One way is to enable them before calling `PFEngine`:

```
MODULE MainModule
  PROC main()
    pflogErrOn := TRUE;
    pflogProgressOn := TRUE;
    PFEngine;
  ENDPROC
ENDMODULE
```

An alternative way of setting these variables is to use the `UserInitEngine` callback routine described in *Running custom code when PFEngine is called on page 77*.

Log files

The logs are outputted to files under `HOME:/PFLogs/`.

Each task has its own log files, and the Logger automatically rotates between two files for each task to limit storage usage. When 10.000 lines have been appended to one file, the other one is emptied and reused.

The files are named `PF_LOG_{taskname}_{filenumber}.txt`, e.g. for T_ROB1, the first log is named `PF_LOG_T_ROB1_1.txt`.

See the default Progress Tracer in *Default Progress Tracer on page 42* for an example how a log might look like.

Continues on next page

Recommendations

To reduce wear on the writable storage media, keeping logs disabled in a production system is good practice. However, if constant logging is required, only log what is necessary. Especially the default Progress Tracer has quite verbose logging, which could be disabled by filtering the progress log level.

To enable the logging persistently, consider using the `UserInitEngine` callback routine described in *Running custom code when PFEngine is called on page 77*.

3.5 Default Progress Tracer

Short description

The default Progress Tracer component simply logs all Progress Reports using the Logger component on the progress log level.

Log format

Using the default Logger component, this is a few typical lines from a default Progress Tracer log:

```
[PROG] 2018-03-08_15:13:55 2: PR PFBase     >>> Entering state:
     EXEC <<< -s1- EXEC

[PROG] 2018-03-08_15:13:55 9: PR PFBase     >>> Validating the
     order before executing <<< -s1- 1 -s2- FALSE

[PROG] 2018-03-08_15:13:55 10: PR PFBase     >>> Executing order
     <<< -s1- 1 -s2- FALSE -s3- p1
```

Looking at the first log entry, the first field is the log level, in this case `[PROG]` for progress. This is followed by a date and timestamp. These fields are added by the default Logger component.

The default Progress Tracer adds a number for the corresponding `PFProgressReportType`, a ":" character, a `PR` tag followed by the source component, in this case `PFBase` (the base framework) and a separator for readability, >>>.

Then follows the default text message provided in the progress report, ended by another separator, <<<.

If the report contains variable information, this will be printed out at the end of the log entry. For a full list of predefined Progress Reports and their variable information, see the `PFProgressReportType` datatype in *PFProgressReportType – Identifier for progress reports on page 68*.

4 Reference: Base framework RAPID instructions

4.1 PFAbortCycle – Aborts the current cycle

Usage

Calling the PFAbortCycle procedure will cause any ongoing and queued orders to be aborted.

Basic examples

The following example illustrates the instruction PFAbortCycle:

Example 1

```
PROC myOrderProc
  IF someCondition THEN
    PFAbortCycle;
  ENDIF
  ! Run the order as planned.
  ! This will not be executed if aborted.
ENDPROC
```

Arguments

[\allTasks]

Data type: switch

An abort will automatically be requested on all tasks managed by the framework.

Other tasks will be aborted from the TRAP execution level, which means that they will be aborted as soon as possible. See *Program execution on page 43*.

Program execution

If the framework is not currently running an order, or any associated pre/post-execution events, PFAbortCycle will have no effect.

If PFAbortCycle is called from normal execution level, the cycle will be aborted immediately. This means that in that case there is no need to put a RETURN statement after the call to PFAbortCycle since the program pointer will be moved to framework control.

If PFAbortCycle is called from any other execution level, such as a TRAP, the cycle will be aborted as soon as possible, which generally is as soon as the user code returns the program pointer to framework control. User code can check at any time whether the cycle has been aborted by using the instruction PFIsAborted. The reason here for not aborting immediately is to allow for a more controlled behavior since a TRAP can happen at any time.

Syntax

```
PFAbortCycle ';'
```

4.2 PFClearQueue – Clears the order queue

Usage

Calling the `PFClearQueue` procedure will clear the order queue from any previously enqueued orders, both part and service. Orders that have been started are not affected.

Basic examples

The following example illustrates the instruction `PFClearQueue`:

Example 1

```
TRAP TrClear
  PFClearQueue;
ENDTRAP
```

Arguments

None

Syntax

```
PFClearQueue ';'
```

Related information

For information on enqueuing orders, see *PFPlaceOrder – Places an order on page 58*.

4.3 PFCurrentOrder – Retrieves information about the current order

Usage

Calling the `PFCurrentOrder` function will retrieve information about the order currently being processed by the production loop. Optionally, it will also retrieve information about queued orders.

Basic examples

The following example illustrates the instruction `PFCurrentOrder`:

Example 1

```
VAR bool processing;
VAR string orderCode;
VAR bool service;

processing := PFCurrentOrder(orderCode, service);

IF processing THEN
  ! Current order code in "orderCode".
  ! The order is a service if "service" is TRUE
ELSE
  ! No order is being processed.
ENDIF
```

Return value

Data type: `bool`

`TRUE` if an order is currently being processed, `FALSE` if not.

Arguments

`orderCode`

Data type: `string`

Variable or persistent to store the order code of the current order.

`service`

Data type: `bool`

Variable or persistent to store `TRUE` if the current order is a service order, `FALSE` if not.

`[\queuedOrderCode]`

Data type: `string`

Variable or persistent to store the queued non-service order code.

`[\queuedServiceOrderCode]`

Data type: `string`

Variable or persistent to store the queued service order code.

Continues on next page

4.3 PFCurrentOrder – Retrieves information about the current order
Continued

Syntax

```
PFCurrentOrder '('
  [ orderCode ':=' ] <variable or persistent (INOUT) of string>
       ','
  [ service ':=' ] <variable or persistent (INOUT) of bool> ','
  [ '\' queuedOrderCode ':=' <variable or persistent (INOUT) of
       string> ',' ]
  [ '\' queuedServiceOrderCode ':=' <variable or persistent (INOUT)
       of string> ',' ] ')'
```

4.4 PFCurrentState – Retrieves information about the current production loop state

Usage

Calling the `PFCurrentState` function will retrieve the current production loop state.

Basic examples

The following examples illustrate the instruction `PFCurrentState`:

Example 1

```
VAR PFBaseState currentState;
currentState := PFCurrentState();
```

Example 2

```
IF PFCurrentState() = PF_STATE_IDLE THEN
  ! Currently in the idle state
ENDIF
```

Return value

Data type: `PFBaseState`

The current production loop state.

Arguments

None

Syntax

```
PFCurrentState '()'
```

Related information

For information about the possible production loop states, see the reference for the `PFBaseState` data type in *PFBaseState – Production loop state on page 63*.

4.5 PFEngine – Starts the production loop

Usage

Calling the `PFEngine` procedure will start the production loop engine. This is typically done from the main routine of a task that should be managed by Production Framework.

Basic examples

The following example illustrates the instruction `PFEngine`:

Example 1

```
PROC main
  PFEngine;
ENDPROC
```

Arguments

None

Program execution

The program pointer will never return from this routine until the production loop is exited – by user request or because of an engine error, see *Error handling on page 48*.

See `PFExitEngine` in *PFExitEngine – Exits the production loop on page 53* for information on how to request an engine exit.

The engine only runs and manages the task it was started in. It can be started on any number of tasks. Each task has its own production loop and set of states. Unless synchronized using the task synchronization functionality provided by the framework or by basic Multitasking / MultiMove functionality, the managed tasks are acting completely independent from each other.

Limitations

- This routine is designed to be called from normal execution level. It should never be called from a `TRAP` or a system event routine.
- `PFEngine` is not reentrant. It should never be called from a framework event or from an order procedure.

Error handling

The following recoverable error(s) can be generated. The error(s) can be handled in an ERROR handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`PF_ENGINE_ERROR`	An engine error was raised within the framework. See the routine `PFEngineErrorCode` in *PFEngineErrorCode – Retrieves the last generated engine error code on page 52*.
`PF_NO_RW_OPT_ERROR`	The RobotWare option for Production Framework is not installed. This is required. See installation instructions in *Prerequisites on page 21*.

Continues on next page

Syntax

```
PFEngine ';'
```

4.6 PFEngineError – Generates an engine error

Usage

Calling the `PFEngineError` procedure will generate an engine error, which will cause the framework to take proper actions and then exit the production loop.

An engine error indicates that there is an internal error in a framework component, or that there is a framework configuration error in the system. `PFEngineError` should not be used for runtime errors or process errors. This means that `PFEngineError` should normally be called from custom components only.

An engine error should ideally never appear during production, but rather during programming and integration of the system. If used for errors that can occur during production, its usage should be restricted to irrecoverable errors.

Basic examples

The following example illustrates the instruction `PFEngineError`:

Example 1

```
IF someCondition THEN
  PFEngineError \exitNow, MY_ERR_CODE, "Something bad happened!";
ENDIF
```

Arguments

`[ \exitNow ]`

Data type: `switch`

The execution will be stopped as soon as possible. The program pointer will be moved from the calling code to base framework control before exiting. This means that no statements in the calling code after this call will be executed.

If `PFEngineError` is called from an execution level other than normal, this argument will be ignored.

`errorCode`

Data type: `num`

A unique numeric error code that identifies this error. The following intervals should be used:

Description	Interval
Reserved by the base framework	10000-19999
Order Controller (OrderController) component	20000-29999
Order Library (OrdersDef) component	30000-39999
Event Library (EventsDef) component	40000-49999
Progress Tracer (Progress) component	50000-59999
Logger (Logger) component	60000-69999

For other sources, any numbers other than the ones above can be used.

`msg`

Data type: `string`

Continues on next page

A message which will be logged in addition to the error code.

Program execution

Unless the `exitNow` argument is used, production loop execution will stop at some time in the future, normally when the base framework gains control over the program pointer.

Syntax

```
PFEngineError
  [ '\' exitNow ',' ]
  [ errorCode ':=' ] < expression (IN) of num > ','
  [ msg ':=' ] < expression (IN) of string > ';'
```

Related information

For information about pre-defined engine errors, see *Engine error codes on page 74*.

4.7 PFEngineErrorCode – Retrieves the last generated engine error code

Usage

The `PFEngineErrorCode` function is used to retrieve the code of the last generated engine error.

`PFEngineErrorCode` is typically called from the ERROR handler in the routine where `PFEngine` has raised the `PF_ENGINE_ERROR RAPID` error, to find out what kind of engine error occurred.

Basic examples

The following examples illustrate the instruction `PFEngineErrorCode`:

Example 1

```
PROC main
  PFEngine;
ERROR
  IF ERRNO = PF_ENGINE_ERROR THEN
    IF PFEngineErrorCode() = PF_ERR_EXEC_ORDER THEN
      ! Could not execute order
      TRYNEXT;
    ENDIF
  ENDIF
ENDPROC
```

Return value

Data type: `num`

The code of the last engine error. Value 0 if no engine error has occurred yet.

Arguments

None

Syntax

```
PFEngineErrorCode '(' ')'
```

Related information

For information about pre-defined engine errors, see *Engine error codes on page 74*.

4.8 PFExitEngine – Exits the production loop

Usage

Calling the `PFExitEngine` will exit the production loop started with the `PFEngine` procedure, and will eventually cause the program pointer to return from `PFEngine`.

Basic examples

The following example illustrates the instruction `PFExitEngine`:

Example 1

```
IF someCondition THEN
  PFExitEngine \now;
ENDIF
```

Arguments

`[ \now ]`

Data type: `switch`

The execution will be stopped as soon as possible. The program pointer will be moved from the calling code to base framework control before exiting. This means that no statements in the calling code after this call will be executed.

If `PFExitEngine` is called from an execution level other than normal, this argument will be ignored.

`[ \allTasks ]`

Data type: `switch`

PFEngine will automatically be exited on all tasks managed by the framework.

PFEngine in other tasks will have the exit requested from the TRAP execution level, which means that the exit will occur as soon as possible. See *Program execution on page 53*.

Program execution

Unless the `now` argument is used, production loop execution will stop at some time in the future, normally when the base framework gains control over the program pointer.

Syntax

```
PFExitEngine
  [ '\' now ] ';'
```

4.9 PFIsAborted – Checks whether the current order is aborted

Usage

The `PFIsAborted` function is used to check whether a cycle in the production loop has been aborted (using `PFAbortCycle`) but still has not arrived at the idle state.

This function is useful to detect from normal execution level that an abort has been requested from a `TRAP`.

Basic examples

The following example illustrates the instruction `PFIsAborted`:

Example 1

```
IF PFIsAborted() THEN
  ! Move robot back to a safe position and/or
  ! other cleanup measures.
  RETURN;
ENDIF
```

Return value

Data type: `bool`

`TRUE` if the cycle has been aborted. `FALSE` if not, or if the idle state has already been reached.

Arguments

None

Syntax

```
PFIsAborted '()'
```

4.10 PFLastOrderCompletion – Retrieves information about the latest order

Usage

The `PFLastOrderCompletion` function is used to check the completion status of the last order that was placed.

`PFLastOrderCompletion` can be called at any time, including after moving the program pointer to main. A typical scenario for using this function is to decide whether the program pointer was moved to main while the production loop was executing an order.

Optionally, the order code and service status of the order can be retrieved as well.

Basic examples

The following examples illustrate the instruction `PFLastOrderCompletion`:

Example 1

```
PROC main
  IF PFLastOrderCompletion() = PF_LC_STATE_PENDING THEN
    ! An order was being executed when
    ! the program pointer was moved to
    ! main.
  ENDIF
  PFEngine;
ENDPROC
```

Example 2

```
! PF_EV_IDLE_CYCLIC or PF_EV_IDLE event routine
PROC myIdleEv
  IF PFLastOrderCompletion() = PF_LC_STATE_ABORTED THEN
    ! The last order was aborted.
  ENDIF
ENDPROC
```

Return value

Data type: `PFOrderCompletionState`

The completion state of the last order placed.

Arguments

`[ \orderCode ]`

Data type: `string`

Variable or persistent to store the order code of the last order placed.

`[ \service ]`

Data type: `bool`

Variable or persistent to store `TRUE` if the last order placed was a service order, `FALSE` if

not.

Continues on next page

4.10 PFLastOrderCompletion – Retrieves information about the latest order

Continued

Syntax

```
PFLastOrderCompletion '('
[ '\' orderCode ':=' <variable or persistent (INOUT) of string> ]
[ '\' service ':=' <variable or persistent (INOUT) of bool> ] ')'
```

Related information

For information about the possible completion states, see the reference for the `PFOrderCompletionState` data type in *PFOrderCompletionState – Completion state of last order placed on page 67*.

4.11 PFLog – Logs a message

Usage

The PFLog procedure is used to send a custom entry to be logged by the Logger component.

Basic examples

The following examples illustrate the instruction PFLog:

Example 1

```
IF someCondition THEN
  PFLog PFL_LVL_WARN, ["There might be an issue!"];
ENDIF
```

Example 2

```
VAR string userInput;
userInput := GetUserInput();
PFLog PFL_LVL_INFO, ["User input was ", userInput];
```

Arguments

`level`

Data type: PFLogLevel

Log level to be used.

`msg`

Data type: string{*}

The message to be logged. The array format is used to allow for messages longer than the 80 character limit of RAPID strings. The array elements will be concatenated during logging.

`NoNewLine`

Data type: switch

If this argument is used, no new line character will be appended at the end of the message.

This argument might be ignored depending on the Logger component.

Syntax

```
PFLog
[ level ':=' ] <expression (IN) of PFLogLevel> ] ','
[ msg ':=' ] <array variable {*} or persistent (IN) of string>
[ '\' NoNewLine ] ';'
```

Related information

For information about the available log levels, see the reference for the PFLogLevel data type in *PFLogLevel – Log level for filtering logs on page 66*.

4.12 PFPlaceOrder – Places an order

Usage

The `PFPlaceOrder` function is used to place a new order for execution by the production loop.

`PFPlaceOrder` should in most cases only be called by the Order Controller component, unless a custom Order Controller implementation is installed that is designed to allow other parts of the software to place orders.

If the default Order Controller is installed, `PFPlaceOrder` should not be used since this can cause undefined behavior.

Basic examples

The following examples illustrate the instruction `PFPlaceOrder`:

Example 1

```
TRAP IncomingOrder
  VAR PFResult result;
  VAR string orderCode;
  VAR string refusedReason;

  orderCode := GetOrderCode();

  result := PFPlaceOrder(
    orderCode, FALSE, refusedReason);

  TEST result
    CASE PF_RESULT_OK:
      ! Order placed successfully
    CASE PF_RESULT_UNDEFINED:
      ! No such order code
    CASE PF_RESULT_REFUSED:
      ! Order code exists, but
      ! order was refused for a reason
      ! stored in the refusedReason variable
    CASE PF_RESULT_BUSY:
      ! The production loop was busy processing
      ! another order and the allowQueue argument
      ! was not used
    DEFAULT:
      ! Something went wrong. Check the log
  ENDTEST
ENDTRAP
```

Return value

Data type: `PFResult`

The result of the attempt to place the order.

Continues on next page

Arguments

`[ \searchOnly ]`

Data type: `switch`

Cannot be used together with `\validateOnly` or `\allowQueue`.

Only perform a search for the provided order without validating or executing it. If the order number does not exist, `PFPlaceOrder` will return `PF_RESULT_UNDEFINED`.

`[ \validateOnly ]`

Data type: switch

Cannot be used together with `\searchOnly` or `\allowQueue`.

Only perform a search and validation of the provided order without executing it. If the order number does not exist, `PFPlaceOrder` will return `PF_RESULT_UNDEFINED`. If the order number exists, but validation fails, `PF_RESULT_REFUSED` will be returned, and `refusedReason` will be assigned the reason for refusal.

`[ \allowQueue ]`

Data type: switch

Cannot be used together with \searchOnly or \validateOnly.

Allow the order to be enqueued if the production loop currently is busy.

The queue has room for one non-service order and one service order. If the slot for the corresponding order type is already used, the old order will be discarded and replaced by this order.

An enqueued service order will always be executed before any enqueued non-service orders.

`orderCode`

Data type: `string`

The order code for the order that should be placed.

`service`

Data type: `bool`

`TRUE` if the placed order is a service order, `FALSE` otherwise. This argument is necessary since order codes for non-service orders and service orders can overlap.

`refusedReason`

Data type: `string`

Variable or persistent to store the reason for refusal if the order was refused during validation.

Syntax

```
PFPlaceOrder '('
[ '\' searchOnly ] |
[ '\' validateOnly ] |
[ '\' allowQueue ]
[ orderCode ':=' ] <expression (IN) of string ','
```

Continues on next page

4.12 PFPlaceOrder – Places an order
Continued

```
    [ service ':=' ] <expression (IN) of bool ','
    [ refusedReason ':=' ] <variable or persistent (INOUT) of string>
         ]
    ')'
```

Limitations

To protect against race conditions, `PFPlaceOrder` disables the interrupt queue while executing on normal execution level, and reenables it again when done. This means that if the interrupt queue was disabled (using `IDisable`) when `PFPlaceOrder` was called, it will still be enabled when `PFPlaceOrder` returns.

Related information

For information about the available result codes, see the reference for the `PFResult` data type in *PFResult – General response data type on page 73*.

For information about order validation, see *Component: Order Library on page 15*.

4.13 PFProgressReport – Generates a Progress Tracer report

Usage

The `PFProgressReport` procedure is used to generate a Progress Report that will be sent to the Progress Tracer component.

It is mainly provided for use within custom framework components, but could technically be used from any user code.

Basic examples

The following example illustrates the instruction `PFProgressReport`:

Example 1

```
RunPhase1;

currentMode := ValToStr(OpMode());
PFProgressReport
  PFP_SOURCE_CUSTOM, MY_CUSTOM_REPORT_1,
  "About to enter phase 2. OP mode is " + currentMode,
  \s1:=currentMode;

RunPhase2;
```

Arguments

`source`

Data type: PFProgressSource

The source component of the report. `PFP_SOURCE_BASE` is reserved for the base framework. The other constants of type `PFProgressSource` can be used, but only if they correspond to the actual component calling the `PFProgressReport` routine.

`report`

Data type: `PFProgressReportType`

The unique (for the provided source) identifier number that can be used by a custom Progress Tracer component to programmatically identify this report.

`defaultTxt`

Data type: `string`

A default text (in any desired language) that can be used for describing this report. All variable information in this string should also be provided separately in the s1-6 arguments below. This enables usage of the variable data within the Progress Tracer without parsing it from the `defaultTxt` string.

`[ \s1 ]`

Data type: `string`

`[ \s2 ]`

Data type: `string`

Continues on next page

4.13 PFProgressReport – Generates a Progress Tracer report
Continued

`[ \s3 ]`
Data type: `string`

`[ \s4 ]`
Data type: `string`

`[ \s5 ]`
Data type: `string`

`[ \s6 ]`
Data type: `string`

Syntax

```
PFProgressReport
[ source ':=' ] <expression (IN) of PFProgressSource> ','
[ report ':=' ] <expression (IN) of PFProgressReportType> ','
[ defaultTxt ':=' ] <expression (IN) of string>
[ '\' s1 ':=' <expression (IN) of string > ]
[ '\' s2 ':=' <expression (IN) of string > ]
[ '\' s3 ':=' <expression (IN) of string > ]
[ '\' s4 ':=' <expression (IN) of string > ]
[ '\' s5 ':=' <expression (IN) of string > ]
[ '\' s6 ':=' <expression (IN) of string > ]
```

Related information

For information about the available report sources, see the reference for the `PFProgressSource` data type in *PFProgressSource – Source of progress reports on page 72*.

For information about the available report types, see the reference for the `PFProgressReportType` data type in *PFProgressReportType – Identifier for progress reports on page 68*.

For information about the Progress Tracer component, see *Component: Progress Tracer on page 19*.

5 Base framework RAPID data types and constants

5.1 PFBaseState – Production loop state

Usage

`PFBaseState` is a data type used to represent a state in the production loop.

Predefined data

Value	Predefined constant	Description
0	`PF_STATE_START`	Initial state after moving program pointer to main.
1	`PF_STATE_INIT_1`	The "Init 1" state.
2	`PF_STATE_INIT_2`	The "Init 2" state.
3	`PF_STATE_IDLE`	The "Idle" state.
4	`PF_STATE_PRE_EV`	The "Pre-execution events" state.
5	`PF_STATE_EXEC`	The "Execution" state.
6	`PF_STATE_POST_EV`	The "Post-execution events" state.
7	`PF_STATE_ABORT`	The "Abort" state.
8	`PF_STATE_EXITED`	Used when the loop has exit, by user request or by engine error.

Characteristics

`PFBaseState` is an alias data type for `num` and consequently inherits its characteristics.

Related information

For information about the topology of the production loop states, see *The production loop on page 11*.

5.2 PFEventType – Event type for production loop / system events

Usage

PFEventType is a data type used to represent a type of event that can be triggered by the production loop entering states. There are also system events that can trigger asynchronously with the production loop. These are executed on the LEVEL_SERVICE execution level.

Note that the use of the word "service" in the context of execution levels is unrelated to the service status of framework orders.

Predefined data

Value	Predefined constant	Trigger condition	Execution level
1	PF_EV_INIT1	Entering the "Init 1" state.	LEVEL_NORMAL
2	PF_EV_INIT2	Entering the "Init 2" state.	LEVEL_NORMAL
3	PF_EV_IDLE	Entering the "Idle" state.	LEVEL_NORMAL
4	PF_EV_IDLE_CYCLIC	If configured, every n:th second while in "Idle state".	LEVEL_NORMAL
5	PF_EV_PREPART	Entering the "Pre-execution events" state and order is non-service.	LEVEL_NORMAL
6	PF_EV_PRESERVICE	Entering the "Pre-execution events" state and order is service.	LEVEL_NORMAL
7	PF_EV_POSTPART	Entering the "Post-execution events" state and order is non-service.	LEVEL_NORMAL
8	PF_EV_POSTSERVICE	Entering the "Post-execution events" state and order is service.	LEVEL_NORMAL
9	PF_EV_ABORT	Entering the "Abort" state.	LEVEL_NORMAL
10	PF_EV_EXIT	Entering the "Exit" state.	LEVEL_NORMAL
11	PF_EV_SYS_POWERON	RAPID system event routine POWER_ON	LEVEL_SERVICE
12	PF_EV_SYS_QSTOP	RAPID system event routine QSTOP	LEVEL_SERVICE
13	PF_EV_SYS_RESTART	RAPID system event routine RESTART	LEVEL_SERVICE
14	PF_EV_SYS_START	RAPID system event routine START	LEVEL_SERVICE
15	PF_EV_SYS_STOP	RAPID system event routine STOP	LEVEL_SERVICE
16	PF_EV_SYS_OPMODE	RAPID system event routine START and RESTART, if the controller operating mode has changed during the stop, or if the system has been rebooted using "Reset RAPID" or "Reset system".	LEVEL_SERVICE
17	PF_EV_SYS_LANG	RAPID system event routine START and RESTART, if the controller language setting has changed during the stop, or if the system has been rebooted using "Reset RAPID" or "Reset system".	LEVEL_SERVICE

Characteristics

PFEventType is an alias data type for num and consequently inherits its characteristics.

Continues on next page

Related information

For information about the topology of the production loop states, see *The production loop on page 11*.

For information about execution level, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

For information about using and configuring the event `PF_EV_IDLE_CYCLIC`, see *Setting Idle state cyclic event interval on page 79*.

5.3 PFLogLevel – Log level for filtering logs

Usage

`PFLogLevel` is a data type used to represent a type of log level. The Logger component uses log levels to signify importance and/or type of log.

Predefined data

Value	Predefined constant	Description
1	`PFL_LVL_ERR`	Error
2	`PFL_LVL_WARN`	Warning
3	`PFL_LVL_INFO`	Information
4	`PFL_LVL_DEBUG`	Debug
5	`PFL_LVL_PROGRESS`	Progress Tracer

Characteristics

`PFLogLevel` is an alias data type for `num` and consequently inherits its characteristics.

Related information

For information about using the log levels, see the `PFLog` instruction in *PFLog – Logs a message on page 57*.

5.4 PFOrderCompletionState – Completion state of last order placed

Usage

`PFOrderCompletionState` is a data type used to represent a state of the last order placed.

Predefined data

Value	Predefined constant	Last order, description
0	`PF_LC_STATE_UNKNOWN`	Unknown. Used after RAPID/system reset.
1	`PF_LC_STATE_PENDING`	Last known state was that the order was executing or about to start executing.
2	`PF_LC_STATE_FINISHED`	The order was finished normally.
3	`PF_LC_STATE_ABORTED`	The order was aborted.
4	`PF_LC_STATE_ERROR`	An error occurred during placing the order, or during the execution.

Characteristics

`PFOrderCompletionState` is an alias data type for `num` and consequently inherits its characteristics.

Related information

For information about using the order completion states, see the `PFLastOrderCompletion` instruction in *PFLastOrderCompletion – Retrieves information about the latest order on page 55*.

5.5 PFProgressReportType – Identifier for progress reports

Usage

`PFProgressReportType` is a data type used to represent a specific Progress Tracer report type.

This data type is typically used when implementing a custom Progress Tracer or when setting up custom Progress Tracer report points.

A custom Progress Tracer component can be implemented to add behavior at certain points during framework execution.

If custom progress report types are created, make sure to avoid overlap with the predefined constants below. Using values above 10000 is a good guideline. Also keep in mind that if custom components from different authors are installed, make sure that their values are not overlapping. Failing to do this can cause undesired behavior if a custom Progress Tracer is also used.

Each custom component should have any custom report types well documented.

Predefined data

The following report types are predefined by the base framework. The optional report data arguments S1-S6 (see related information) are described if present.

Value	Predefined constant	Report occurs when
1	`PFPR_BASE_ENGINE_START`	`PFEngine` is called.
2	`PFPR_BASE_ENTER_STATE`	A new production loop state is entered. S1: The name of the state ("INIT1", "INIT2", "IDLE", "PREEV", "EXEC", "POSTEV", "ABORT", "EXITED").
3	`PFPR_BASE_INIT_MODULE`	Base framework is about to initialize a component. S1: Name of the component ("PFLogger", "PFProgress", "PFOrdersDef", "PFEventsDef", "PFOrderController")
4	`PFPR_BASE_CB_IDLE`	Invoking the CallbackIdle routine on the Order Controller. The isIdle argument is TRUE. Also see PFPR_BASE_PLORD_CB_ST.
5	`PFPR_BASE_WAIT_ORDER`	Waiting for order in the "Idle" state.
6	`PFPR_BASE_SYNC_EXEC1`	Coordinating tasks before executing order. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Current order sync tag.
7	`PFPR_BASE_SYNC_EXEC2`	Coordinating tasks after executing order. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Current order sync tag.
8	`PFPR_BASE_VALIDATE1`	Not used. See `PFPR_BASE_PLORD_CHK`, used when validating an order when the order is placed.

Continues on next page

Value	Predefined constant	Report occurs when
9	PFPR_BASE_VALIDATE2	Validating order before executing order, after any Preexecution events have been fired. Invoking the ValidateOrder instruction on the Order Library component. S1: Current order code. S2: Current order is service ("TRUE","FALSE").
10	PFPR_BASE_EXEC	Executing order. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Current order procedure name.
11	PFPR_BASE_REFUSED	Order was refused before executing it. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Reason for refusal.
12	PFPR_BASE_CB_REFUSED	Invoking the CallbackRefused routine on the Order Controller component. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Reason for refusal.
13	PFPR_BASE_CB_SUCCESS	Invoking the CallbackSuccess routine on the Order Controller component. S1: Current order code. S2: Current order is service ("TRUE","FALSE"). S3: Numeric, cycle time for complete cycle including events. S4: Numeric, cycle time for executing the order, excluding events but including task synchronization time. S5: Numeric, cycle time for executing the order, excluding events and task synchronization time.
14	PFPR_BASE_REWIND_EV	Invoking the RewindSequences routine on the Event Library component.
15	PFPR_BASE_REWIND_EV_S	Invoking the RewindSysSequences routine on the Event Library component.
16	PFPR_BASE_NEXT_EV	Invoking the GetNextEvent routine on the Event Library component. S1: Numeric value of event type (see *PFEventType – Event type for production loop / system events on page 64*)
17	PFPR_BASE_NEXT_EV_S	Invoking the GetNextSysEvent routine on the Event Library component. S1: Numeric value of event type (see *PFEventType – Event type for production loop / system events on page 64*)
18	PFPR_BASE_N_EV_ERR	Result from Event Library routine GetNextEvent was not PF_RESULT_OK. S1: Numeric value of the returned result.
19	PFPR_BASE_N_EV_ERR_S	Result from Event Library routine GetNextSysEvent was not PF_RESULT_OK. S1: Numeric value of the returned result.

Continues on next page

5.5 PFProgressReportType – Identifier for progress reports
Continued

Value	Predefined constant	Report occurs when
20	PFPR_BASE_SYNC_EV1	Coordinating tasks before executing synchronized event. S1: Event procedure name. S2: Event sync tag
21	PFPR_BASE_SYNC_EV2	Coordinating tasks after executing synchronized event. S1: Event procedure name. S2: Event sync tag
22	PFPR_BASE_EXEC_EV	Executing event. S1: Event procedure name.
23	PFPR_BASE_EXEC_EV_S	Executing system event. S1: Event procedure name.
24	PFPR_BASE_PLORD	PFPlaceOrder was called. S1: Order code. S2: Order is service ("TRUE","FALSE"). S3: Function ("FIND","VALIDATE","ENQUEUE","PLACE")
25	PFPR_BASE_PLORD_BSY	PFPlaceOrder was called, the production loop was busy. S1: Order code. S2: Order is service ("TRUE","FALSE").
26	PFPR_BASE_PLORD_ORDINF	PFPlaceOrder was called, invoking Order Library routine GetOrderInfo. S1: Order code. S2: Order is service ("TRUE","FALSE").
27	PFPR_BASE_PLORD_UNDEF	PFPlaceOrder was called, invoked Order Library routine GetOrderInfo or ValidateOrder, order code was unknown. S1: Order code. S2: Order is service ("TRUE","FALSE").
28	PFPR_BASE_PLORD_NOTOK	PFPlaceOrder was called, invoked Order Library routine GetOrderInfo or ValidateOrder, did not get a valid return code. S1: Order code. S2: Order is service ("TRUE","FALSE").
29	PFPR_BASE_PLORD_CHK	PFPlaceOrder was called, invoking the ValidateOrder instruction on the Order Library component. S1: Current order code. S2: Current order is service ("TRUE","FALSE").
30	PFPR_BASE_PLORD_REFUSE	PFPlaceOrder was called, invoked Order Library routine ValidateOrder, order was refused. S1: Order code. S2: Order is service ("TRUE","FALSE"). S3: Reason for refusal.
31	PFPR_BASE_PLORD_DRYOK	PFPlaceOrder was called, order was OK, but the order was placed using one of the \searchOnly or \validateOnly optional arguments.

Continues on next page

Value	Predefined constant	Report occurs when
32	PFPR_BASE_PLORD_CB_ST	Invoking the CallbackIdle routine on the Order Controller. The isIdle argument is FALSE. Also see PFPR_BASE_CB_IDLE.
33	PFPR_BASE_DO_ABORT_REQ	Abort was requested.
34	PFPR_BASE_DO_ABORT_GRA1	Abort was granted. Using immediate abort. The abort was requested from normal execution level.
35	PFPR_BASE_DO_ABORT_GRA2	Abort was granted. Aborting when possible. The abort was requested from an execution level other than normal.
36	PFPR_BASE_DO_ABORT_DEN	Abort denied. The production loop was not busy.
37	PFPR_BASE_IDLE_CYCLIC	Cyclic interval timed out. Firing cyclic idle events. S1: Numeric, current idle cyclic interval.
38	PFPR_BASE_APP_EV_HOOK	Invoking function package application hook (see *Running custom code before/after any event type on page 78*). S1: Hook before or after event type ("PRE","POST"). S2: Numeric value of event type (see *PFEventType – Event type for production loop / system events on page 64*)
39	PFPR_BASE_EXIT_1	Engine exit was ordered. Exiting production loop when possible.
40	PFPR_BASE_EXIT_2	Engine exit was ordered. Exiting production loop immediately.
41	PFPR_BASE_EXIT_3	Engine exit is imminent.
42	PFPR_BASE_INIT_HOOK_1	Invoking user engine init hook, PFInitEngineHook.
43	PFPR_BASE_INIT_HOOK_2	User init hook has been invoked successfully.
44	PFPR_BASE_INIT_HOOK_3	User init hook could not be found.
45	PFPR_BASE_ENGINE_ERR	Raising RAPID error PF_ENGINE_ERROR from PFEngine.
46	PFPR_BASE_CLEAR_Q	Order queue cleared.

Characteristics

PFProgressReportType is an alias data type for num and consequently inherits its characteristics.

Related information

For information about Progress Tracer reports, see the PFProgressReport instruction in *PFProgressReport – Generates a Progress Tracer report on page 61* and the overview of the Progress Tracer component in *Component: Progress Tracer on page 19*.

5.6 PFProgressSource – Source of progress reports

Usage

PFProgressSource is a data type used to represent a source of a Progress Tracer report. All predefined reports are from the source PFP_SOURCE_BASE. However, custom implementations of components or other user code can use the existing source types.

Predefined data

Value	Predefined constant	Description
1	PFP_SOURCE_BASE	Framework base
2	PFP_SOURCE_ORDERSDEF	Order Library component
3	PFP_SOURCE_EVENTSDEF	Event Library component
4	PFP_SOURCE_ORDERCTRL	Order Controller component
5	PFP_SOURCE_CUSTOM	Anything else

Characteristics

`PFProgressSource` is an alias data type for `num` and consequently inherits its characteristics.

Related information

For information about Progress Tracer reports, see the `PFProgressReport` instruction in *PFProgressReport – Generates a Progress Tracer report on page 61* and the overview of the Progress Tracer component in *Component: Progress Tracer on page 19*.

5.7 PFResult – General response data type

Usage

`PFResult` is a general data type for returning results from framework component API calls.

Predefined data

Value	Predefined constant
-1	PF_RESULT_UNSET
0	PF_RESULT_OK
1	PF_RESULT_UNDEFINED
2	PF_RESULT_REFUSED
3	PF_RESULT_BUSY
4	PF_RESULT_ERROR

Characteristics

`PFResult` is an alias data type for `num` and consequently inherits its characteristics.

5.8 Engine error codes

Overview

Engine error codes are values of the datatype `num`.

In some cases, e.g. when implementing custom components, it might be useful to define custom engine errors. For information regarding declaration and usage of custom engine errors, see the `PFEngineError` routine in *PFEngineError – Generates an engine error on page 50*.

Most of the base framework error codes are caused by a failure when the framework tries to call a component using late binding. This will typically happen when the component is missing or replaced by a custom implementation that does not adhere to the API specification.

Predefined error codes, base framework

Value	Predefined constant	Description
10001	`PF_ERR_INIT_COMPONENT`	Failure when trying to call the `Init` routine on a component.
10002	`PF_ERR_UNH_RAISED_ERROR`	There was an unhandled RAPID error raised within `PFEngine`. Check the log for more information.
10003	`PF_ERR_CALLBACK_IDLE`	Failure when trying to call the `CallbackIdle` routine on the Order Controller component.
10004	`PF_ERR_GETSYNCNUM`	Internal error. The internal pool of sync numbers for task synchronization is empty.
10005	`PF_ERR_UNEXPECTED_VALID_RET`	The `ValidateOrder` routine on the Order Library component returned an unexpected value.
10006	`PF_ERR_CALL_VALIDATE_ORDER`	Failure when trying to call the `ValidateOrder` routine on the Order Library component.
10007	`PF_ERR_EXEC_ORDER`	Failure when trying to call the corresponding procedure for a given order.
10008	`PF_ERR_CALLBACK_REFUSED`	Failure when trying to call the `CallbackRefused` routine on the Order Controller component.
10009	`PF_ERR_CALLBACK_SUCCESS`	Failure when trying to call the `CallbackSuccess` routine on the Order Controller component.
10010	`PF_ERR_CALLBACK_ABORTED`	Failure when trying to call the `CallbackAborted` routine on the Order Controller component.
10011	`PF_ERR_CALL_REWIND`	Failure when trying to call the `RewindSequences` routine on the Event Library component.
10012	`PF_ERR_CALL_NEXT_EV`	Failure when trying to call the `GetNextEvent` routine on the Event Library component.
10013	`PF_ERR_FIRE_EV`	Failure when trying to call the corresponding procedure for a given event.

Continues on next page

Value	Predefined constant	Description
10014	PF_ERR_CALL_REWIND_SYS	Failure when trying to call the RewindSysSequences routine on the Event Library component.
10015	PF_ERR_CALL_NEXT_EV_SYS	Failure when trying to call the GetNextSysEvent routine on the Event Library component.
10016	PF_ERR_FIRE_EV_SYS	Failure when trying to call the corresponding procedure for a given sys-event.
10017	PF_ERR_CALL_GET_ORD_INFO	Failure when trying to call the GetOrderInfo routine on the Order Library component.
10018	PF_ERR_CALL_LOG	Failure when trying to call the Log routine on the Logger component.
10019	PF_ERR_CALL_PROGRESSREPORT	Failure when trying to call the ProgressReport routine on the Progress Tracer component.
10020	PF_ERR_UNKNOWN_STATE_TRANS	Internal error. Unknown state transition. Check the log for more information.
10021	PF_ERR_NONEXECUTABLE_STATE	Internal error. Unknown state. Check the log for more information.

Predefined error codes, default Order Library

Value	Predefined constant	Description
30001	PF_ERR_DUPLICATE_ORDER_NUM	Order code / plcCode is not unique among declared partdata or servicedata.

This page is intentionally left blank

6 Miscellaneous

6.1 Running custom code when PFEngine is called

PFInitEngineHook

When `PFEngine` is called, the framework will try to call a procedure named `PFInitEngineHook`. This procedure can be declared by the user or function package developer anywhere in global scope.

This can typically be used to set global configuration variables, such as enabling log levels in the default Logger component, or the settings described in this section.

Note that – as with any RAPID routine - only one `PFInitEngineHook` can be declared in global scope in each task.

An example, where the error log level is set to enabled from a module other than the main module.

```
MODULE SomeModule
  PROC PFInitEngineHook()
    pflogErrOn := TRUE;
  ENDPROC
ENDMODULE

MODULE MainModule
  PFEngine; ! Error logging will be enabled
ENDMODULE
```

6.2 Running custom code before/after any event type

PFApplictionPreEventHook and PFApplicationPostEventHook

In some situations, it is desirable to run code prior to or after events of a certain type are fired. This is typically needed by function package developers who can't control the user's choice of event sequencing, but still need to run code first or last.

Before and after running events of any type, the framework will try to call procedures named `PFApplicationPreEventHook` and `PFApplicationPostEventHook` respectively.

The use of the words "Pre" and "Post" in this case should not be confused with the *pre-execution* and *post-execution* event type.

The procedures must have an argument of the type `PFEventType`, which will contain the corresponding event type.

Example:

```
PROC PFApplicationPreEventHook(PFEventType evType)
  TEST evType
    CASE PF_EV_ABORT:
      ! Do something before any
      ! ABORT events are fired
    CASE PF_EV_INIT1:
      ! Do something before any
      ! INIT1 events are fired
  ENDTEST
ENDPROC
```

6.3 Setting Idle state cyclic event interval

pfIdleCyclicEvInterval

By default, the event type `PF_EV_IDLE_CYCLIC` is disabled. If configured, the event type will be fired recurrently at a defined frequency while the production loop is in the idle state.

The configuration is done by setting the global `num` variable `pfIdleCyclicEvInterval` to a positive number, representing the time in seconds for which the framework should wait between each trigger.

The `pfIdleCyclicEvInterval` variable will be reset to 0 (disabled) after moving the program pointer to main. It should preferably be initialized using the method described in *Running custom code when PFEngine is called on page 77*.

6.4 Disabling the Progress Tracer component

pfBaseProgressEnabled

For some rare cases, it is possible to speed up the framework execution overhead by disabling the use of the Progress Tracer component.

This is done by setting the global bool variable pfBaseProgressEnabled to FALSE.

The `pfBaseProgressEnabled` variable will be reset to `TRUE` (enabled) after moving the program pointer to main. It should preferably be initialized using the method described in *Running custom code when PFEngine is called on page 77*.

7 PFView - FlexPendant interface

Overview and prerequisites

PFView is a simple information interface for the FlexPendant. It executes within Production Screen, and therefore requires the *Production Screen* RobotWare option.

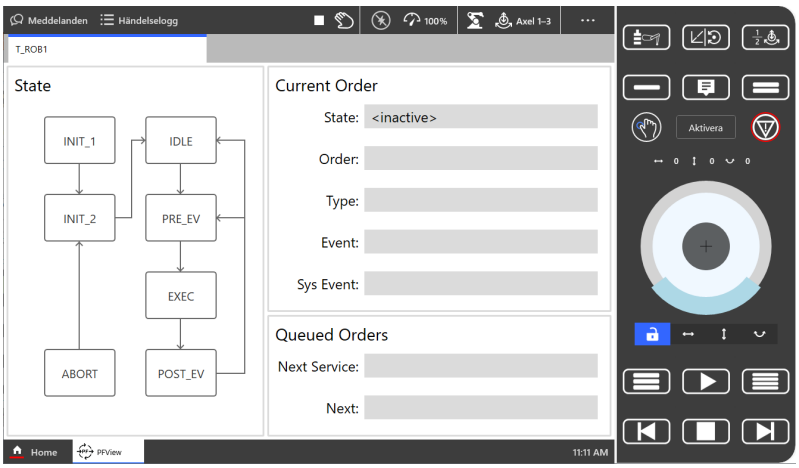
PFView is not strictly necessary for a Production Framework based system to function, but can rather be seen as a tool of convenience for the integrator and/or operator to get a quick overview of the current state of the base framework.

Since the default components are designed to be replaceable by function package developers, only information from the base framework is displayed. For the same reason, interaction with the framework through PFView is limited to avoid conflicts with a custom Order Controller component.

If more information and/or user interaction is required, a custom FlexPendant application more tightly coupled to the non-base components could be developed using any of the options available for creating FlexPendant applications.

Available information in PFView

PFView is started from the main window in Production Screen.



xx2200001372

The following information is available in **PFView**:

	Description
Task	Selection of the RAPID task to display information from.
State	A graphical view of the production loop with the current state highlighted with color.
Current Order – State	The current production loop state.
Current Order – Order	The description of the current order.
Current Order – Type	The type of the current order, Normal for non-service orders and Service for service orders.
Current Order – Event	The description of the current event being executed.
Current Order – Sys Event	Although not strictly related to the current order, the currently system event being executed.

Continues on next page

Continued

	Description
Queued Orders – Next Service	The description of any queued service order.
Queued Orders – Next	The description of any queued non-service order.

8 Advanced: Custom components

8.1 Templates

Locating the templates

Templates for custom components will be installed under `HOME:/PFTemplates/`, provided that the "*Custom component templates*" option was selected during installation.

These templates contain empty RAPID routines with plenty of comments. They serve both as skeleton files to get started and as documentation for the routines that each component needs to implement.

Refer to *Base framework RAPID data types and constants on page 63* for a reference of the RAPID instructions available from the base framework.

There is one template for each component:

Component	File
Order Library	`PFOrdersDef.sys`
Event Library	`PFEventsDef.sys`
Order Controller	`PFOrderController.sys`
Logger	`PFLogger.sys`
Progress Tracer	`PFProgress.sys`

Note

Since the template files are overwritten during system reset, always copy the files to a safe location before proceeding with the implementation.

8.2 Replacing a component

Removing the default component

A custom component cannot be installed at the same time as a corresponding default component. Make sure that the default component option for the component type being replaced is not selected in Installation Manager.

If already installed, it is necessary to deselect it in Installation Manager and run a system reset.

Installing a custom component

As mentioned in each template, it is important that the name of the RAPID module containing the component interface routines is the same as provided in the template. If this is not the case, the base framework will not be able to find the routines.

It is not important from a framework point of view how a custom component is loaded. However, it should be loaded on all tasks. The framework must be able to see the module, which must have all required routines declared.

A common way of obtaining this is to use the *Automatic Loading of Modules* (CAB_TASK_MODULES) type under the Controller (SYS) system parameters database.

ABB AB
Robotics & Discrete Automation
S-721 68 VÄSTERÅS, Sweden
Telephone +46 10-732 50 00

ABB AS
Robotics & Discrete Automation
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 22 87 2000

ABB Engineering (Shanghai) Ltd.
Robotics & Discrete Automation
No. 4528 Kangxin Highway
PuDong New District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

ABB Inc.
Robotics & Discrete Automation
1250 Brown Road
Auburn Hills, MI 48326
USA
Telephone: +1 248 391 9000

abb.com/robotics

3HAC085600-001, Rev A, en