



# Workflow Studio Manual

October, 2010

Copyright (c) 2010 by tmssoftware.com bvba

Web: <http://www.tmssoftware.com>

E-mail: [info@tmssoftware.com](mailto:info@tmssoftware.com)

# Table of Contents

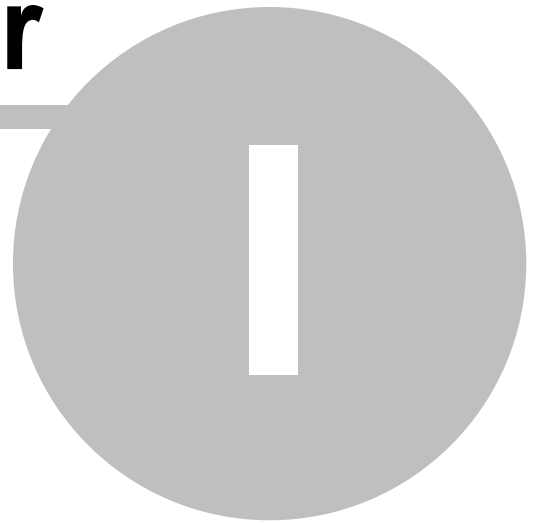
<b>Chapter I Introduction</b>	<b>5</b>
1 Overview .....	5
2 Licensing and Copyright Notice .....	5
3 What's New .....	6
4 Upgrading to version 1.5 .....	7
5 Upgrading to version 1.4 .....	7
6 Manual Installation .....	8
7 Getting Support .....	9
 <b>Chapter II Basic Concepts</b>	 <b>11</b>
1 Workflows and Tasks .....	11
2 Workflow Definition .....	11
3 Workflow Instance .....	11
4 Task Definition (concept) .....	11
5 Task Instance .....	12
6 Workflow Engine .....	12
7 Workflow Users and Groups .....	12
 <b>Chapter III Getting Started</b>	 <b>15</b>
1 Installation .....	15
2 Running Demos .....	16
3 Components Overview .....	16
TWorkflowStudio component .....	16
TWorkflowDB component .....	17
TWorkflowADODB component.....	18
TWorkflowDBXDB component.....	18
TWorkflowpFIBDB component.....	18
TWorkflowAnyDACDB component.....	19
Auxiliary components .....	19
4 "Hello world" tutorial .....	20
5 E-mail notifications .....	21
6 Monitoring expired tasks (task timeout) .....	21
7 Localization .....	21
 <b>Chapter IV Database Structure</b>	 <b>24</b>
1 Underlying Database Structure .....	24
2 Upgrading database from previous versions .....	26

<b>Chapter V Creating Workflow Definitions</b>	<b>28</b>
1 Workflow diagram objects .....	28
Start block .....	28
End block .....	28
Error block .....	28
Source connector .....	29
Target connector .....	29
Transition .....	29
Fork object .....	30
Join object .....	30
Decision block .....	30
Task block .....	31
Task definition properties .....	31
Approval block .....	37
Script block .....	37
Run workflow block .....	38
2 Workflow variables .....	39
3 Attachments .....	40
4 Expressions .....	41
5 Scripts .....	43
 <b>Chapter VI User interface windows</b>	 <b>46</b>
1 Workflow definitions dialog .....	46
2 Workflow definition editor .....	46
3 Task list dialog .....	48
 <b>Chapter VII Using Workflow Studio programatically</b>	 <b>54</b>
1 Running an instance based on a definition name .....	54
2 Workflow with workflow instance variables .....	54
3 Running an instance from code: full example .....	54
4 Retrieve the list of tasks for a specified user .....	55
5 Creating and editing an workflow definition .....	55
6 Running workflow instances for expired tasks .....	55
 <b>Chapter VIII Extending the scripting system</b>	 <b>57</b>
1 Accessing Delphi objects .....	57
Registering Delphi components .....	57
Access to published properties .....	57
Class registering structure .....	57
Calling methods .....	58
More method calling examples .....	59
Accessing non-published properties .....	59
Registering indexed properties .....	60
Retrieving name of called method or property .....	61

Registering methods with default parameters .....	61
<b>2 Accessing Delphi functions, variables and constants .....</b>	<b>62</b>
Overview .....	62
Registering global constants .....	62
Accessing global variables .....	62
Calling regular functions and procedures .....	63
<b>3 Using libraries .....</b>	<b>64</b>
Overview .....	64
Delphi-based libraries .....	64
The TatSystemLibrary library .....	66
Removing functions from the System library .....	68

# **Chapter**

---



# **Introduction**

# 1 Introduction

## 1.1 Overview

Workflow Studio is a Delphi VCL framework for Business Process Management (BPM). With Workflow Studio you can easily add workflow and BPM capabilities to your application, by allowing you or your end-user to create workflow definitions and running them.

Here are some examples of business process that can be automated by using Workflow Studio:

- Order management
- Sales management
- Hiring process
- Help desk tasks
- Sales and marketing tasks
- Project management
- Quality checking
- Warranty management
- Software deployment
- Product requirement and specification
- Expense tracking

Main tasks you can do with Workflow Studio are:

- Design workflow definitions visually in a diagram
- Run the workflow definitions
- Manage tasks generated by the workflows

## 1.2 Licensing and Copyright Notice

TMS Workflow Studio components trial version are free for use in non-commercial applications, that is any software that is not being sold in one or another way or that does not generate income in any way by the use of the application. The trial expires after some time, and might display some nag screens during usage.

For use in commercial applications, you must purchase a single license or a site license of Workflow Studio. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates for a full version cycle and priority email support. A single developer license allows ONE developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A single developer license is NOT transferable to another developer within the company or to a developer from another company. Both licenses allow royalty free use of the components when used in binary compiled applications.

**TMS Workflow Studio components, trial or registered versions, cannot be used to create a commercial general purpose workflow application. The main purpose of applications created using TMS Workflow Studio components must as such be different from the generic workflow capabilities offered by the components.**

The component cannot be distributed in any other way except through free accessible Internet Web pages or ftp servers. The component can only be distributed on CD-ROM or other media with written authorization of the author.

Online registration for Workflow Studio is available at <http://www.tmssoftware.com/go.asp?orders>. Source code & license is sent immediately upon receipt of check or registration by email. Payment grants users the right for a full version cycle source code updates (from version x.y to x+1.y)

Workflow Studio is Copyright © 2002-2010 TMS Software. ALL RIGHTS RESERVED.  
No part of this help may be reproduced, stored in any retrieval system, copied or modified, transmitted in any form or by any means electronic or mechanical, including photocopying and recording for purposes others than the purchaser's personal use.

## 1.3 What's New

### version 1.5.0.1 (Oct-2010)

- Fixed: Issues installing Workflow Studio on RAD Studio XE.

### version 1.5 (Oct-2010)

- New: Timeout feature for automatic expiration of workflow tasks.
- New: "Run workflow" block to run a separated workflow instance.
- New: RAD Studio XE support.
- New: Option to define hidden status in workflow tasks.
- New: Property DisplayTaskStatus in workflow diagram.
- New: Administrator privilege in user groups to allow updating of tasks assigned to different users.
- New: High level properties to handle with task status programmatically.
- New: Support for AnyDAC components.
- New: Event OnInitializeScriptEngine to initialize Script Engine object.
- New: Event BeforeSaveTaskInstance in WorkflowStudio.
- Improved: User interface for defining status in workflow tasks.
- Fixed: Issues with multiple outputs from script blocks.
- Fixed: Flickering when moving blocks on Workflow diagram.
- Fixed: Issues with wsClasses unit in C++Builder.

### version 1.4.1 (Jul-2010)

- New: Event "OnGetNow" to retrieve current date/time.
- Improved: small visual changes in several forms.
- Fixed: Error on Redo after undoing a transition line insertion.
- Fixed: Field labels are not painted when using XP Manifest (Delphi 7).
- Fixed: Error adding attachments to workflow definition.

### version 1.4 (Jan-2010)

- Improved: [WorkflowStudio global variable removed](#), allowing multiple instances of the component to be used.
- Improved: TCustomWorkflowDB.ComponentToString and ComponentFromString methods made protected.
- Improved: several methods in TCustomWorkflowDB made virtual.

### version 1.3 (Feb-2009)

- New: Redesigned workflow definition editor now based on internal diagram editor
- New: More modern toolbar in workflow definition editor (Delphi 2005 and up)
- New: Workflow definition editor now support grouping blocks for easier design
- New: "\_Workflow" variable available from scripts allowing access to the workflow diagram and its methods and properties
- New: TWorkflowStudio.OnRunFinished event - being fired when a workflow instance execution is finished
- Fixed: minor bug fixes

**version 1.2 (Oct-2008)**

- New: Delphi 2009/C++Builder 2009 support
- New: TWorkflowStudio.GroupAssignmentMode property allows to create a single task for multiple users in group
- Improved: prevent users from changing tasks assigned to other users
- Fixed: AV happening in threads in some situations

**version 1.1 (May-2008)**

- New: Support for C++Builder 6, 2006 and 2007
- New: Support for FIBPlus database components
- New: Spanish translation added
- New: OnTaskCreated and OnTaskFinished events in TWorkflowStudio component
- New: OnBeforeExecuteNode and OnAfterExecuteNode events in TWorkflowStudio component
- New: TTaskInstance.CreatedOn property
- New: TWorkflowInstance status: wsFinishedWithError
- New: TWorkflowStudio.OnWorkInsError event allows higher control of errors raised from an workflow execution
- New: It's possible now to use expressions in the "Assigned To" field in a task definition so that the task is assigned dinamically to an user/group
- New: Task List dialog now can be called in MDI mode, beside the existing modal mode. You can use, e.g., ShowUserTasksDlg('john', wfmMDI);
- Improved: Error handling while executing workflows. An error message is displayed when workflow is finished with error (can be avoided with OnWorkInsError)
- Improved: thread performance to execute workflow instances
- Fixed: minor bugs in TWorkflowADODB and TWorkflowDBXDB caused AV when deleting a connection component at design time

**version 1.0 (Oct-2007)**

- First release

## 1.4 Upgrading to version 1.5

Workflow Studio version 1.5 include some new features that required small changes in underlying database structure.

Before upgrading Workflow Studio from previous versions to version 1.5, the database structure must be updated.

For details about the needed changes, see the section [Upgrading database from previous versions](#).

## 1.5 Upgrading to version 1.4

When upgrading the Workflow Studio from version 1.3 (or earlier) to version 1.4, the following items must be observed:

- The global variable WorkflowStudio was removed. Replace any reference to WorkflowStudio by reference to the component TWorkflowStudio you are using.
- The class TWorkflowDiagram was moved to the unit wsDiagram.
- Components derived from TListView (like TTaskListView) now have a WorkflowStudio property that must be set to reference the TWorkflowStudio component being used.
- The component TWorkflowDiagram also has a new property WorkflowStudio that must refer to a TWorkflowStudio component.



## 1.6 Manual Installation

If you selected manual installation during setup, follow this instructions to manually install Workflow Studio in Delphi or C++Builder:

### 1) From Delphi, C++Builder

Add the following directories to the Delphi library path (Tool | Environment Options):

```
{WS}\source\workflow\source  
{WS}\source\diagram\source  
{WS}\source\scripter\source
```

Where {WS} is the installation directory of Workflow Studio.

### 2) From Delphi, C++Builder

Choose : File, Open and browse for the correct workflowstudioX.dpk package file for Delphi or workflowstudioX.bpk for C++Builder:

- workflowstudio5.dpk: Delphi 5
- workflowstudio6.dpk: Delphi 6
- workflowstudio7.dpk: Delphi 7
- workflowstudio2005.dpk: Delphi 2005
- workflowstudio2006.dpk: Delphi 2006/C++Builder 2006
- workflowstudio2007.dpk: Delphi 2007/C++Builder 2007
- workflowstudio2009.dpk: Delphi 2009/C++Builder 2009
- workflowstudio2010.dpk: Delphi 2010/C++Builder 2010
- workflowstudio2011.dpk: Delphi XE/C++Builder XE
- workflowstudioc6.bpk: C++Builder 6

### Notes for users of the Scripter Studio and/or Diagram Studio

If you're trying to install Workflow Studio with one of the packages above, use the following instructions (applies only to registered version. Trial version cannot must be installed separately):

a) Make sure you update your existing Scripter and Diagram Studio files with the ones provided in Workflow Studio package. In other words, overwrite your Diagram Studio files with the ones in {WS}\source\diagram\source directory, and overwrite your Scripter Studio files with the ones in {WS}\source\scripter\source.

b) Instead of browsing for the packages in {WS}\source\workflow\packages directory, use the packages in {WS}\source\workflow\packages\_min directory. Browse, open and install the correct one according to your Delphi/C++Builder version:

- workflowstudiom5.dpk: Delphi 5
- workflowstudiom6.dpk: Delphi 6
- workflowstudiom7.dpk: Delphi 7
- workflowstudiom2005.dpk: Delphi 2005
- workflowstudiom2006.dpk: Delphi 2006/C++Builder 2006
- workflowstudiom2007.dpk: Delphi 2007/C++Builder 2007
- workflowstudiom2009.dpk: Delphi 2009/C++Builder 2009
- workflowstudiom2010.dpk: Delphi 2010/C++Builder 2010

- workflowstudiom2011.dpk: Delphi XE/C++Builder XE
- workflowstudiomc6.bpk: C++Builder 6

## 1.7 Getting Support

### General notes

Before contacting support:

- Make sure to read the tips, faq and readme.txt or install.txt files in component distributions.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component you have a problem.
  - Specify which Delphi or C++Builder version you're using and preferably also on which OS.
  - In case of IntraWeb or ASP.NET components, specify with which browser the issue occurs.
  - For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.
- 
- Send email from an email account that
    - 1) allows to receive replies sent from our server
    - 2) allows to receive ZIP file attachments
    - 3) has a properly specified & working reply address

### Getting support

For general information: [info@tmssoftware.com](mailto:info@tmssoftware.com)

Fax: +32-56-359696

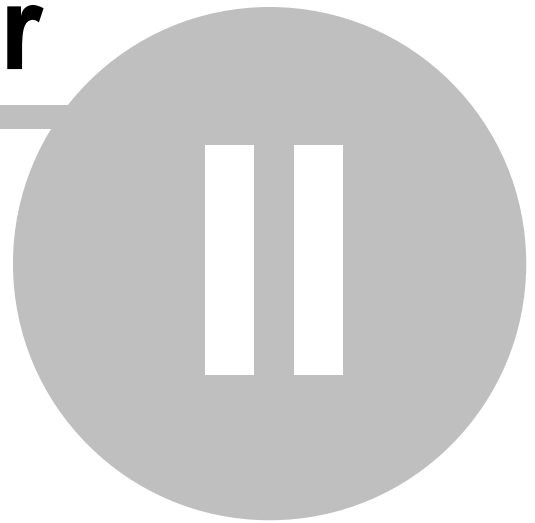
For all questions, comments, problems and feature request for VCL components:

[help@tmssoftware.com](mailto:help@tmssoftware.com).

To improve efficiency and speed of help, refer to the version of Delphi, C++Builder, Visual Studio .NET you are using as well as the version of the component. In case of problems, always try to use the latest version available first.

# Chapter

---



# Basic Concepts

## 2 Basic Concepts

Some basic concepts are presented here, for your clear understanding of how Workflow Studio works and its main components.

### 2.1 Workflows and Tasks

Workflow Studio works with two major concepts: workflows and tasks.

A workflow is a representation of a business process. In Workflow Studio, the workflow concept is split in two more specific concepts: [workflow definition](#), which is the specification of a business process, and [workflow instance](#), which is a running business process.

A task is a pending work for a user. In Workflow Studio, the task concept is split in two more specific concepts: [task definition](#), which is the specification of a task, and [task instance](#), which is actually a existing pending task for a user.

### 2.2 Workflow Definition

A workflow definition is the representation of a business process. For easy understanding, we can compare the workflow definition to a flowchart which specifies how the business process work.

In a workflow definition you specify which actions are to be performed (update database, send e-mail, run a script and, more important, create a task), and in which order. If you are creating a workflow definition for order processing, for example, then you might want to check if the order amount is higher than 10 000. If not, then create an approval task for the local manager. If yes, create an approval task for the director. In any case of approval, send an e-mail for the financial department.

You can use the workflow designer to visually build the flowchart for the workflow definition. All the workflow definitions are kept in the database. Each workflow definition receives a name that uniquely identifies it (for example, "order processing", "software deployment", "help desk support", etc.).

### 2.3 Workflow Instance

A workflow instance is a running instance of a [workflow definition](#). A single workflow definition will generate an unlimited number of workflow instances.

For example, you can have a single workflow definition for order processing, and for each order, you will have a workflow instance. In a workflow definition you might have a variable named "Order number". Each workflow instance will have its own order number, and the variable "Order number" will have a different value. Each workflow instance will have its own state and internal variable values.

A workflow instance can be started, running or finished. All workflow instance records are kept in the database, even the finished ones.

### 2.4 Task Definition (concept)

A task definition specifies a task to be created for a user. It's not the task itself, but a specification for the task.

In the task definition you specify the subject, task name, description, the user, a list of valid status, and

other properties. A task definition is always "inside" a workflow definition. One of the actions you can define in a workflow definition is generating tasks, and the task definition is part of the action specification.

For example, in a workflow definition for order processing, you might want to create a task for the manager to approve the order. In this case, the task definition would be something like this:

**Subject:** Order approval

**Description:** Please approve the order [OrderNo]

**User:** Manager

**Valid Status:** Waiting approval, approved, rejected

## 2.5 Task Instance

A task instance is a task created for an user based on a [task definition](#). A single task definition can generate several task instances.

A task instance is created when a [workflow instance](#) is run and reaches to a point where a task must be created, based on a task definition. At that time, the task instance is created for a specified user.

Each user has a list of his/her pending task instances. Once the task is finished it is removed from the list of pending tasks. There is still an option for listing the closed tasks, in the task list window.

Each task instance has its own record in the database. Even if the task is closed, the record is not deleted.

## 2.6 Workflow Engine

The workflow engine is the place where the [workflow instances](#) are run. It creates the instance, runs it, and terminates it when the workflow instance is finished. The workflow engine can run various instances at the same time.

In current version, the workflow engine is just an internal thread-based class that creates a thread for each running workflow instance, and manages those threads.

## 2.7 Workflow Users and Groups

Workflow Studio is strongly based on tasks, which in turn are always assigned to an user or a group of users. So, Workflow Studio does also need to use information about users and groups.

Workflow Studio does not provide a full user control system, it does not have login dialogs, add/remove user, group definition interfaces, etc.. You must build the user management available in your application, or use a 3rd party tool to do that, like TMS Security System (<http://www.tmssoftware.com/go?tss>).

However, Workflow Studio needs to know about users and groups. So, you must fill in a list of valid users and groups, that will be used by Workflow Studio. This can be done at the beginning of the program. The code below is an example that shows how to add users and groups to Workflow Studio.

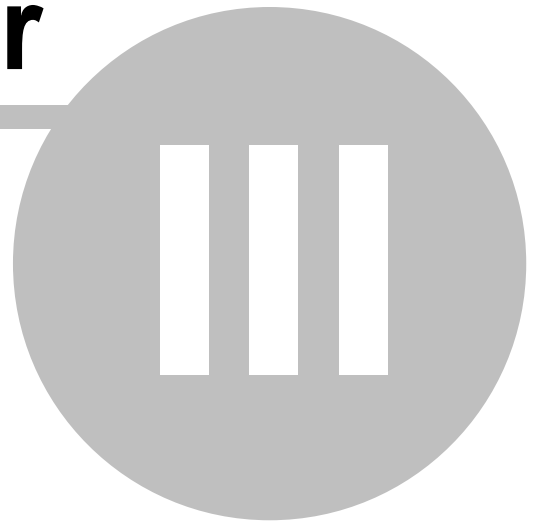
```
{Add users and groups}
With WorkflowStudio.UserManager do
begin
  {Add all users}
  Users.Clear;
  Users.Add('1', 'John', 'john@domain');
  Users.Add('2', 'Sarah', 'sarah@domain');
```

```
Users.Add('3', 'Scott', 'scott@domain');
Users.Add('4', 'Mario', 'mario@domain');
Users.Add('5', 'Tina', 'tina@domain');
{Add groups and specify which users belong to each group}
Groups.Clear;
With Groups.Add('managers') do
begin
    UserIds.Add('1'); //John
    UserIds.Add('2'); //Sarah
end;
With Groups.Add('programmers') do
begin
    UserIds.Add('3'); //Scott
    UserIds.Add('4'); //Mario
    UserIds.Add('5'); //Tina
end;
end;
```

Note that two groups were created, and each group contains a list of user id's that belong to that group. Workflow Studio uses the user information to assign tasks, send e-mails, and other user-based tasks.

# Chapter

---



# Getting Started

## 3 Getting Started

### 3.1 Installation

For trial versions of Workflow Studio, just run the installer and everything will be installed accordingly. If you're using registered version, take the following steps to install:

**1. Create a directory for your workflow studio files {\$WS} and unzip the zip file distribution (wsreg.zip) in {\$WS} directory**

**2. Your folder structure should look like this:**

```
{WS}\dbscripts
{WS}\demos (including subdirectories)
{WS}\languages (including subdirectories)
{WS}\doc
{WS}\source\diagram\source
{WS}\source\scripter\source
{WS}\source\workflow\source
{WS}\source\workflow\packages
{WS}\source\workflow\packages_min
{WS}\tutorials
```

**3. Adding directories to library path**

Add the following directories to your Delphi/C++ Builder library path (menu Tools, Environment options, Directories - or Library). Remember to replace {\$WS} by your real Workflow Studio directory location:

```
{WS}\source\workflow\source
{WS}\source\diagram\source
{WS}\source\scripter\source
```

**4. Installing the package**

From Delphi / C++ Builder, choose menu option File, Open and browse for the correct workflowstudioX.dpk package file for Delphi or workflowstudioX.bpk for C++Builder. The packages are in the {\$WS}\source\workflow\packages directory:

workflowstudio5.dpk	Delphi 5
workflowstudio6.dpk	Delphi 6
workflowstudio7.dpk	Delphi 7
workflowstudio2005.dpk	Delphi 2005
workflowstudio2006.dpk	Delphi 2006
workflowstudio2007.dpk	Delphi 2007
workflowstudio2009.dpk	Delphi 2009
workflowstudio2010.dpk	Delphi 2010
workflowstudio2011.dpk	Delphi XE

Click "Install" to install the package. All components should be installed now.



## 5. Note for register users of Diagram Studio and/or Scripter Studio - installing alternative packages

Workflow Studio uses some files from Diagram and Scripter Studio. If you have installed Diagram and/or Scripter Studio in your machine, you can install different package files (which include only minimum files for Workflow Studio).

Overwrite your existing scripter and/or diagram files with the ones provided in the Workflow Studio package.

Follow the same installation steps from 1 to 2. In step 3, add the directories to library path only if you don't have the package previously installed. For example, only add {\$WS}\source\scripter\source to library path if you don't have Scripter Studio already installed in your Delphi environment. Then, install the correct package for your Delphi version (the packages are in the {\$WS}\source\workflow\packages\_min):

workflowstudiom5.dpk	Delphi 5
workflowstudiom6.dpk	Delphi 6
workflowstudiom7.dpk	Delphi 7
workflowstudiom2005.dpk	Delphi 2005
workflowstudiom2006.dpk	Delphi 2006
workflowstudiom2007.dpk	Delphi 2007
workflowstudiom2009.dpk	Delphi 2009
workflowstudiom2010.dpk	Delphi 2010
workflowstudiom2011.dpk	Delphi XE

After compiling, Delphi might show a warning telling you that the package requires other packages. Let it make the corrections to the package (adding the required packages) and then install again. Ignore warnings about file being implicated included in the package.

## 3.2 Running Demos

If you have followed [installation](#) procedure accordingly, you will be able to run the demos. Just browse for the demo project, compile it and run.

## 3.3 Components Overview

Here is a brief summary of the installed components.

### 3.3.1 TWorkflowStudio component



This is the main component of the package. A single TWorkflowStudio instance should be added to the whole application, and from this component you have access to various methods and properties needed to work with Workflow Studio programmatically.

The global variable WorkflowStudio contains a reference to the main TWorkflowStudio component of the application.

TWorkflowStudio component provides the following main object properties:

```
property WorkflowManager: TWorkflowManager;
```

The TWorkflowManager object provides methods to manipulate [workflow definitions](#) and [workflow instances](#) (creating, deleting, signaling, etc.).

```
property TaskManager: TTaskManager;
```

The TTaskManager object provides methods to manipulate tasks, specially task instances.

```
property WorkflowEngine: TWorkflowEngine;
```

The TWorkflowEngine object provides methods to the [workflow engine](#), which runs the workflow instances.

```
property UserManager: TWorkflowUserManager;
```

The TWorkflowUserManager object is used to manipulate [workflow users and groups](#).

```
property ScriptEngine: TWorkflowScriptEngine;
```

The TWorkflowScriptEngine object is used to parse and evaluate [expressions](#) and [scripts](#).

```
property UserInterface: TCustomWorkflowUserInterface;
```

The TCustomWorkflowUserInterface object provides methods for displaying the predefined windows and dialogs of Workflow Studio, like the [task list dialog](#), [workflow definition editor](#) and [workflow definition dialog](#).

```
property WorkflowDB: TWorkflowDB;
```

The [TWorkflowDB](#) object is the component which makes the layer to save/load data to/from the database.

You must specify which TWorkflowDB component you want to use to access database.

### 3.3.2 TWorkflowDB component



The TWorkflowDB component provides a layer between application-level workflow data and the database server. This component builds all SQL commands used to insert, delete and update data in the database, but when it comes to execute the SQL statements, it does nothing.

You must provide event handlers for the events OnCreateQuery, OnExecuteQuery and OnAssignSQLParams.

```
TCreateQueryEvent = procedure(Sender: TObject; SQL: string;  
    var Dataset: TDataset; var Done: boolean) of object;
```

Here you must provide your own code to create a TDataset descendant which provides connection to the database. You must also set the SQL statement (SELECT) provided. You must return a TDataset object in the parameter Dataset, and set Done parameter to true.

```
TExecuteQueryEvent = procedure(Sender: TObject; Dataset: TDataset;  
    var Done: boolean) of object;
```

Here you receive your created TDataset descendant and call its specific method to execute an sql

statement. The SQL statement should have already set in the component in the OnCreateQuery event.

```
TAssignSQLParamsEvent = procedure(Sender: TObject; Dataset: TDataset;  
  AParams: TParams; var Done: boolean) of object;
```

Here you receive a list of parameters (AParams) and you must set the parameters in your TDataset descendant, provided in Dataset parameter. Each component has an specific issue with parameters, and here you might deal with those specific issues, specially with blobs and memos.

This way you have total flexibility to use the component package you want to access your preferable database.

Workflow Studio provides some TWorkflowDB descendants form some widely used component sets (like ADO and dbExpress). In that case, you just drop the component you want to use (like TWorkflowADODB), assign it to the [TWorkflowStudio](#) component, and that's it, you don't need to set anything else.

### 3.3.2.1 TWorkflowADODB component



Provides ready-to-use ADO layer to the database. Use it if you want to use ADO components to access your database.

All you have to do is to set the Connection property to a valid TADOConnection component.

### 3.3.2.2 TWorkflowDBXDB component



Provides ready-to-use dbExpress layer to the database. Use it if you want to use dbExpress components to access your database.

All you have to do is to set the Connection property to a valid TADOConnection component. Optionally, you can also set DBType property to the database you are going to use, this will provide specific issues treatment for each database.

### 3.3.2.3 TWorkflowFIBDB component



Provides ready-to-use FIBPlus (<http://www.devrace.com/en/fibplus>) layer to the database. Use it if you want to use FIBPlus components to access your database.

All you have to do is to set the Database property to a valid TpFIBDatabase component.

#### Installing the component

Since not every Delphi environments have the FIBPlus components installed, TWorkflowFIBDB component is not installed by default. To install it, do the following steps:

- 1) Add {\$WS}\source\workflow\source\fibplus directory to the Delphi library path, where {\$WS} is the root directory of Workflow Studio files
- 2) Open package *wspFIBpck.dpk* located in the directory above and install it.
- 3) If Delphi suggests changes to the package like adding required packages (it will at least include

FIBPlus package to the list of required packages), accept it and install again until all warnings are gone.

### 3.3.2.4 TWorkflowAnyDACDB component



Provides ready-to-use AnyDAC from DA-SOFT (<http://www.da-soft.com/anydac/>) layer to the database. Use it if you want to use AnyDAC components to access your database. All you have to do is to set the Connection property to a valid TADConnection component.

#### Installing the component

Since not every Delphi environments have the AnyDAC components installed, TWorkflowAnyDACDB component is not installed by default. To install it, do the following steps:

- 1) Add {\$WS}\source\workflow\source\anydac directory to the Delphi library path, where {\$WS} is the root directory of Workflow Studio files.
- 2) Open package *wsAnyDACpck.dpk* located in the directory above and install it.
- 3) If Delphi suggests changes to the package like adding required packages (it will at least include AnyDAC packages to the list of required packages), accept it and install again until all warnings are gone.

### 3.3.3 Auxiliary components

The following components are provided with Workflow Studio, although you might not need to use them. They are components and controls used internally by the Workflow Studio framework, and you can use them if you want to customize Workflow Studio, like building your own dialog windows.



#### TWorkflowDiagram

Contains the workflow definition diagram. It's used in the [workflow definition editor](#). Although you will probably not add a new TWorkflowDiagram component to a form, you might often use some of its methods and properties when using Workflow Studio programatically.



#### TWorkDefListView

It's a TListView descendant which shows a list of the workflow definitions in the database. Used in the [workflow definition dialog](#).



#### TTaskListView

It's a TListView descendant which shows a list of task instances based on some filters (assigned to a user, or belonging to a workflow instance). Used in the [task list dialog](#).



#### TAttachmentListView

It's a TListView descendant which shows the attachment files in a specified [attachment](#). It's used in the [task definition properties](#) windows and also in the [task list dialog](#).



### **TTaskStatusCombo**

It's a TComboBox descendant which shows the current status of a [task instance](#), and the drop down list shows the available status. If user changes the combo value, it automatically changes the value of the status in the task instance object. It's used in the [task list dialog](#).



### **TTaskLogListView**

It's a TListView descendant which shows the audit log for the changes in a task instance. It's used in the [task list dialog](#).

## **3.4 "Hello world" tutorial**

Workflow Studio provides basic online tutorial in Flash which display the basic steps to get an application running. Watching that tutorial will help you to understand the basics to start. The link to the online tutorial is included in *tutorials* folder.

Here we will provide a very simple list of tasks you should do to get Workflow Studio to run:

1. [Install](#) the components.

2. **Create the tables and fields for Workflow Studio in your database.**

Workflow Studio provides several SQL scripts for creating needed tables and fields for some database vendors (e.g., Oracle, Microsoft SQL Server, etc.), but you can create yourself in the database server you want.

3. **Create a new VCL Application in Delphi.**

4. Drop a [TWorkflowStudio](#) component.

5. Drop one of available [TWorkflowDB](#) components

(TWorkflowADODB for ADO, TWorkflowDBXDB for dbExpress, etc.).

6. **Drop a component for database connection and configure it to connect to your database**

(TADOConnection if you're using ADO, TSQLConnection if you're using dbExpress, etc.).

7. **Associate your TWorkflowDB component to your database connection component using Connection property (or analog property)**

8. **Associate your TWorkflowStudio component to your TWorkflowDB component using WorkflowDB property**

9. [Add valid users](#) to your TWorkflowStudio component before application starts.

It can be done in FormCreate method of application's main form, for example

10. **That's it, you have it configured. Now you can use some methods to call the standard dialogs in Workflow Studio, like [workflow definitions dialog](#) and [task list dialog](#).**

### 3.5 E-mail notifications

There are several points in the workflow definition where an e-mail can be sent. An example is when a [task instance](#) is created for an user. If the [task definition properties](#) of this task instance is marked as "Send e-mail notification", an e-mail will be sent to the user notifying him that the task instance was created and assigned to him.

However, there is no built-in code to send e-mails in Workflow Studio. When an e-mail is to be sent, the event OnSendMail of TWorkflowStudio component is fired. So, if you want your workflow so support e-mail sending, create an event handler for TWorkflowStudio.OnSendMail event, and send the e-mail yourself from there, using your own method.

The signature for the OnSendMail event is below:

```
type
  TEmailInformation = record
    ToAddr: string;
    From: string;
    Bcc: string;
    Cc: string;
    Subject: string;
    Text: string;
  end;

procedure(Sender: TObject; TaskIns: TTaskInstance; AUser: TWorkflowUser;
  AEmailInfo: TEmailInformation; var Sent: boolean) of object;
```

So, use AEmailInfo parameter to build your e-mail message, using ToAddr, From, Bcc, CC, Subject and Text properties.

Set Sent parameter to true when the e-mail is sent. For extra information (you will often use only AEmailInfo), you can use TaskIns and AUser parameters to know which task instance generated the e-mail, and for each [workflow user](#) the e-mail is about to be sent.

### 3.6 Monitoring expired tasks (task timeout)

Workflow Studio supports task expiration (timeout). It means you can define a lifetime for a task. After a [task instance](#) is created, the workflow waits for it to be finished. If the task doesn't finish until the expiration date, the task will expire automatically, and the workflow will follow the path you have defined for expired tasks.

In the [task definition properties](#) you can define the [expiration](#) settings.

For the tasks to be effectively expired, you must have some kind of monitor that checks for all pending tasks in a regular interval, and then perform the correct operations on the expired tasks. TWorkflowStudio component provides a single method to perform this operation, but nevertheless, you must build this monitor yourself. Read more in section "[Running workflow instances for expired tasks](#)"

### 3.7 Localization

Workflow Studio provides an easy way to localize the strings. All strings used in user interface (messages, button captions, dialog texts, menu captions, etc.) are in a single file names wsLanguage.pas.

In the *languages* folder, included in Workflow Studio distribution, there are several wsLanguage.pas files available for different languages. Just pick the one you want and copy it to the official directory of your workflow studio source code.

If the language you want does not exist, you can translate it yourself. Just open wsLanguage.pas file and translate the strings to the language you want.

As a final alternative, you can translate the wsLanguage.txt file, also included in ws\_languages.zip file, and send the new file to us. The advantage of this approach is that this file is easier to translate (you don't have to deal with pascal language) and can be included in the official Workflow Studio distribution. This way we keep track of changes in translatable strings and all new strings are marked in the upcoming releases. This way, you will always know what is missing to translate, and do not need to do some kind of file comparison in every release of Workflow Studio.

So, in summary, to localize Workflow Studio strings:

**Option 1**

- Pick the correct wsLanguage.pas file from the ws\_languages.zip file, according to the language you want.
- Replace the official wsLanguage.pas (in source code directory) by the one you picked.

**Option 2**

- Translate the official wsLanguage.pas directly

**Option 3**

- Translate the wsLanguage.txt file and send it to us ([support@tmssoftware.com](mailto:support@tmssoftware.com)).
- We will send you back a translated wsLanguage.pas file and this translation will be included in official release.

# **Chapter**

---



**IV**

# **Database Structure**



## 4 Database Structure

Workflow Studio saves and loads data into a database. It is database-based. You must create the needed tables and fields in your database in order to use Workflow Studio.

Workflow Studio distribution includes some SQL scripts for easy creation of tables in the database. The scripts are in the *dbscripts* folder. The scripts provide so far are:

**wsSQLServer.sql**: for Microsoft SQL Server databases

**wsOracle.sql**: for Oracle databases

**wsFirebird.sql**: for Firebird/Interbase databases

As an alternative, you can also manually create the tables and fields in your database, just use the same [underlying database structure](#).

### 4.1 Underlying Database Structure

Below is the structure for the tables used by Workflow Studio.

The field types are described for Microsoft SQL Server, but you can just translate it for the database server you want.

For example, Image field is a blob field, while Text is a blob or memo field.

All Datetime fields must contain both date and time parts (not only date).

#### **Table wsattachment**

<b>Field</b>	<b>Data type</b>	<b>Options</b>
id	Int	Primary key. Required.
workkey	Int	
createdon	Datetime	
filecontent	Image	
objecttype	Int	

**Table wstaskinstance**

Field	Data type	Options
id	Int	Primary key. Required.
task	Text	
createdon	Datetime	
userid	VarChar(50)	
comments	Text	
name	VarChar(50)	
subject	VarChar(50)	
description	Text	
workflowinstancekey	Int	
workflowdefinitionkey	Int	
completed	VarChar(1)	
modifiedon	Datetime	
modifieduserid	VarChar(50)	

**Table wsworkflowdefinition**

Field	Data type	Options
id	Int	Primary key. Required.
workflow	Text	
name	VarChar(255)	

**Table wsworkflowinstance**

Field	Data type	Options
id	Int	Primary key. Required.
workflow	Text	
workflowdefinitionkey	Int	
createdon	Datetime	
modifiedon	Datetime	
finishedon	Datetime	
nextruntime	Datetime	

**Table wstasklog**

Field	Data type	Options
taskinstancekey	Int	Primary key. Required.
eventdate	Datetime	Primary key. Required.
operation	VarChar(1)	Primary key. Required.
userid	VarChar(50)	
info	VarChar(100)	
info2	VarChar(100)	

## 4.2 Upgrading database from previous versions

If you have the database structure created for Workflow Studio in versions prior to 1.5, some changes in structure are required to keep the component working properly.

Workflow Studio distribution includes some SQL scripts for easy updating the workflow tables in database, located in *dbscripts* folder. The scripts provided so far are:

**wsSQLServerUpdate.sql**: for Microsoft SQL Server databases;

**wsOracleUpdate.sql**: for Oracle databases;

**wsFirebirdUpdate.sql**: for Firebird/Interbase databases.

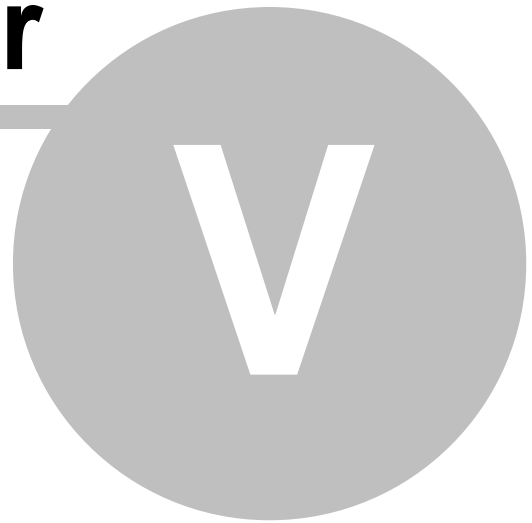
Following are listed the database changes required to upgrade Workflow Studio version. The field types are described for Microsoft SQL Server, but you can just translate it for the database server you want.

- **From previous versions to version 1.5:**

1. New field in table **wsworkflowinstance**:  
**nextruntime** Datetime

# **Chapter**

---



# **Creating Workflow Definitions**

## 5 Creating Workflow Definitions

One of the main tasks using Workflow Studio is to create [workflow definitions](#) that will represent the business process. You create a workflow definition in the [workflow definitions dialog](#).

A workflow definition is basically a workflow diagram where you add diagram objects. You can also define workflow variables and workflow attachments to be used in the workflow diagram. While building the workflow definition, you can take benefit from expressions and scripts to make it more flexible and adaptable to different situations.

### 5.1 Workflow diagram objects

A diagram object is something that you put inside a workflow diagram. It can be a block which perform a specified operation, a transition, or some special objects that control the execution flow, like connectors and forks.

#### 5.1.1 Start block



**Overview:** indicates where the process starts

**Allowed inputs:** 0

**Allowed outputs:** 1

**Description:**

Only one Start block can exist in a workflow diagram.

#### 5.1.2 End block



**Overview:** indicates where the process ends

**Allowed inputs:** many

**Allowed outputs:** 0

**Description:**

Only one End block can exist in a workflow diagram.

#### 5.1.3 Error block



**Overview:** Error block is executed when an error occurs while executing the workflow diagram.

**Allowed inputs:** 0

**Allowed outputs:** 1

**Description:**

Only one Error block can exist in a workflow diagram.

Whenever an error occurs during the execution of the workflow diagram, the execution flow goes to the error block and then follow the path specified by it (the next block after the error block). You can use error block to perform some clean up and then finish the execution of workflow diagram.

### 5.1.4 Source connector



**Overview:** The source connector is a passthrough block to increase readability of diagram. It makes execution flow to a [target connector](#).

**Allowed inputs:** many

**Allowed outputs:** 0

**Description:**

If the execution flow reaches a source connector, then the diagram jumps to a target connector that is related to the source connector.

The "relation" between the source and target connector is done by the text inside them. If the text is the same, the connection is established. For example, when the execution reaches a source connector labeled "A", then it jumps to the target connector labeled "A". This way you can have several source-target connections in a single diagram.

### 5.1.5 Target connector



**Overview:** The target connector is a passthrough block to increase readability of diagram. It receives execution flow from a related [source connector](#).

**Allowed inputs:** 0

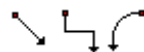
**Allowed outputs:** 1

**Description:**

If the execution flow reaches a source connector, then the diagram jumps to a target connector that is related to the source connector.

The "relation" between the source and target connector is done by the text inside them. If the text is the same, the connection is established. For example, when the execution reaches a source connector labeled "A", then it jumps to the target connector labeled "A". This way you can have several source-target connections in a single diagram.

### 5.1.6 Transition



**Overview:** Connects one block to another, indicating the execution flow of diagram.

**Description:**

You use a transition (line, or wire) to connect a block to another. The transition indicates the execution flow, meaning that it flows from the source block to the target block.

You create a transition by pressing down the mouse button at one link point (blue "x" in blocks) in the source block, dragging the mouse, and releasing the button at one link point in the target block. An arrow indicates the execution flow.

You can create a normal transition, a side transition or an arc transition. All behave the same, the difference is only visual.

You might not be able to attach a transition to a block, depending on the allowed inputs and outputs to the block. For example, if you try to create a transition that targets a Start block, you won't be able to do it, because a Start block does not allow inputs.

Some blocks allow multiple transitions as outputs. In this case, each transition must be labeled so that

the diagram knows which transition must be used according to a specified condition. Only one transition is used for leaving a block. So, for example, two transitions might be connected to the output of a [Decision block](#), but one should be labeled "yes" and the other should be labeled "no", so the diagram will know which transition to take according to the result of the decision condition.

### 5.1.7 Fork object



**Overview:** Creates parallel execution paths.

**Allowed inputs:** many

**Allowed outputs:** many

**Description:**

The fork object is used to create parallel execution paths. When a diagram of a workflow instance is started, there is a single execution path (started by the [Start block](#)). If the execution reaches a fork block, the flow is split in several parallel paths (depending on the number of outputs in the fork) the execute simultaneously.

At the end, all execution paths must finish at the same [join object](#), otherwise the diagram is incorrect. Once all parallel execution paths finish, the main execution path starts again, from the join object.

### 5.1.8 Join object



**Overview:** Ends parallel execution paths created by a [fork object](#).

**Allowed inputs:** many

**Allowed outputs:** 1

**Description:**

The join object is used to end and join parallel execution paths. When a diagram of a workflow instance is started, there is a single execution path (started by the [Start block](#)). If the execution reaches a fork block, the flow is split in several parallel paths (depending on the number of outputs in the fork) the execute simultaneously.

At the end, all execution paths must finish at the same join object, otherwise the diagram is incorrect. Once all parallel execution paths finish, the main execution path starts again, from the join object.

The main execution path will restart in the join object, going through the next block connected to its output. The main execution path will not restart until all execution paths created by the fork object are finished.

### 5.1.9 Decision block



**Overview:** Changes the execution flow according to a boolean condition

**Allowed inputs:** many

**Allowed outputs:** 2 ("yes" and "no")

**Description:**

Use a decision block to change the execution flow according to a boolean condition. When execution flow reaches a decision block, the condition of the block is evaluated. If it is true, then the execution path goes through the [transition](#) labeled "yes". If it is false, it goes through the transition labeled "no".

### 5.1.10 Task block



**Overview:** Creates task instances for users based on its task definitions.

**Allowed inputs:** many

**Allowed outputs:** many (restricted to status list)

**Description:**

The task block is one of the more important block types in a diagram. With task block you can specify task definitions to be created for users. When the execution flow reaches a task block, it creates task instances for each task definition.

The task block itself is nothing but a set of task definitions. So, to use the task block, just create one or more task definitions and set the [task definition properties](#).

The workflow execution will stop at the task block until all task instances are finished. A task instance is finished when its status change to a completion status.

After all task instances are finished, the execution flow continues, according to these rules:

Task block has only one output

The execution flow goes through that path.

Task block has two or more outputs

Each output transition must be labeled with the name of a completion status. The execution path goes through the transition which is labeled with the same text as the task block output, which is the completion status of the task instance, i.e., if the task instance was completed as "approved" (a valid completion status, for example), then the execution path goes through the transition labeled "approved".

If more than one task instance was created by the task block, either from the same task definition or from different task definitions, then the task block output will be the most common completion status among the task instances. For example, if two task instances finish as "rejected" and one task instance finishes as "approved", then the task block output will be "rejected" and the execution path will follow the transition labeled the same.

### 5.1.11 Task definition properties

The task definitions are created in the task definition editor (Tasks dialog) for the [Task block](#). When you double-click a Task block, the dialog is shown.

You can use *Add* and *Delete* buttons to add or delete task definitions. The list view at the left of the dialog displays the existing task definitions for the block. The name of task definition is displayed.

**Name**

Valid identifier that uniquely identifies the task definition. In the example below, "SetDateTask".



### General

#### **Subject**

Contains the subject of the task. Used as a summary for e-mail messages or for the task list. Accepts [expressions](#).

#### **Description**

Multi-line description of the task. Here all the instructions about the task should be inserted. Accepts [expressions](#).

#### **Assignment**

The name of the user (or group) that the task instance will be assigned to. If it is a user, a single task instance will be created and will be assigned to that user. If it is a group, then the behaviour depends on the value of *TWorkflowStudio.GroupAssignmentMode* property:

*gamMultipleTasks*: This is default value. A task will be created for each user in the group. So, if a group has users "john" and "maria", one task will be created for John, and another to Maria, and the tasks will be independent (both will have to be concluded)

*gamSingleTask*: A single task will be created that will be visible for all users in the group. If you later include/remove users to/from the group, the existing tasks will become not visible for users removed from the group, and will become visible to users added to group. Any user from the group can update the task, including finishing it.

#### **Send mail notification**

If true (checked), then an e-mail notification will be sent to the user when the task instance is created.

The screenshot shows a 'Tasks' dialog box with a list of tasks on the left and a form on the right. The list contains one task, 'SetDateTask'. The form has tabs for 'General', 'Status', 'Attachments', 'Fields', and 'Expiration'. The 'General' tab is selected, showing the following fields:

- Subject:** Set deployment date for [CompanyName]
- Description:** Please specify a start date for the project. The company is [CompanyName]. Set the start date in the "Fields" tab. You must then close this task by setting its status to "closed". You can also attach some files to the "Attachments" tab, if needed. Thank you.
- Assignment:** John (selected from a dropdown menu)
- Send mail notification:** ☐ (unchecked)

At the bottom of the dialog are 'Ok' and 'Cancel' buttons.

## **Status**

### **Status list**

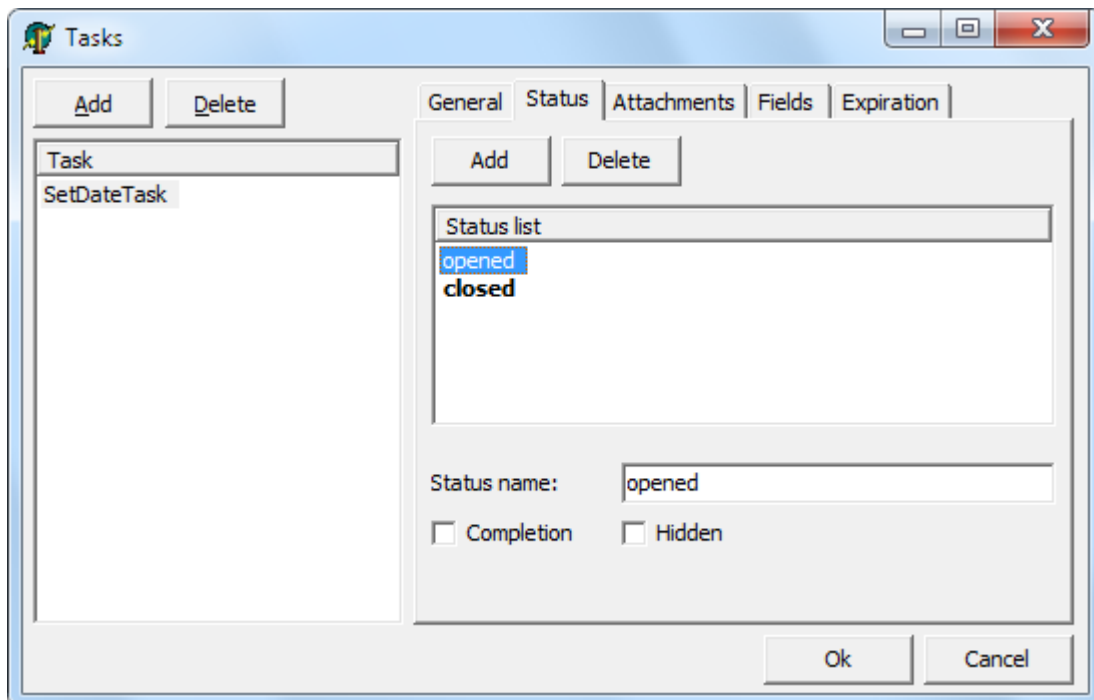
Contains a list of the valid status for the task.

The initial status of the task will be the first status in the list.

Some status can be marked as *completion status*, by checking the "Completion" option. When the status of a task instance changes to a completion status, the task is considered finished. You can have more than one status marked as a completion status.

Status with the "Hidden" option checked is a possible status to the task, but will not be displayed in user interface for changing the task status (like in the status combo box).

When a task instance is created, the valid status are displayed in a combo box in the [task list dialog](#), and then user can change the status of a task.



### **Attachment permissions**

Defines the permissions for the users when handling [attachments](#).

#### **Show attachments**

Shows the attachments tab in the task list. If false (unchecked), the user will not be able to do anything related to attachments (add, view, etc.). If true (checked) the user will be able to, at least, open and view attachments.

#### **Allow remove attachments**

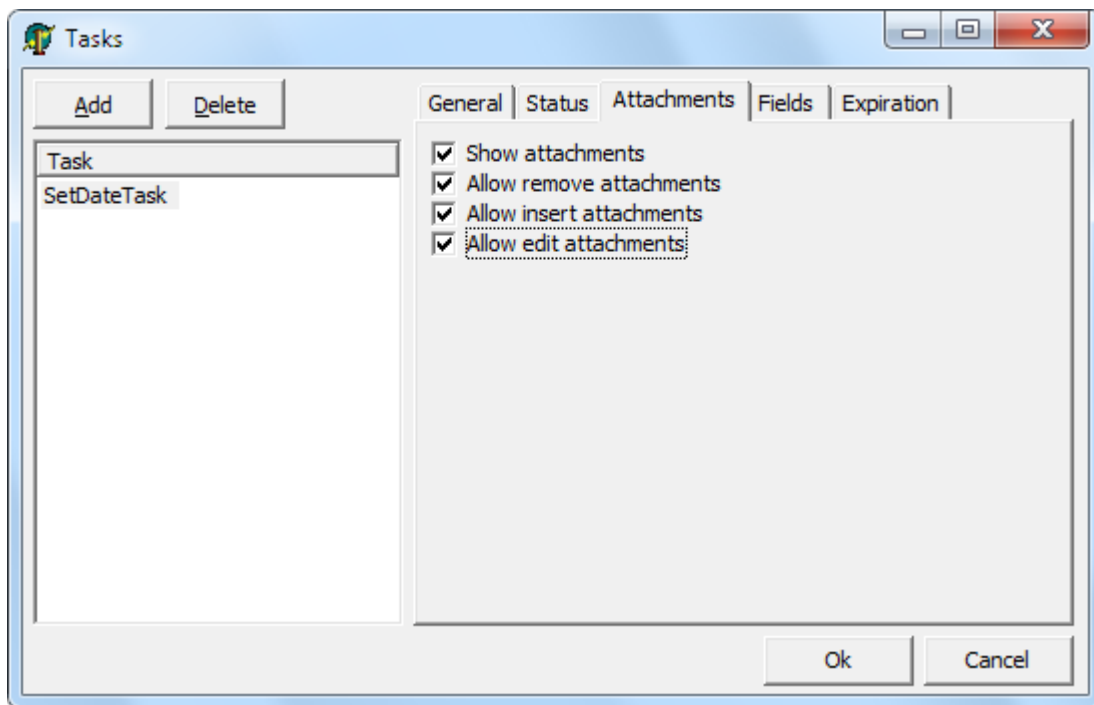
If true (checked), allows the user to remove attachments from the task instance

#### **Allow insert attachments**

If true (checked), allows the user to insert new attachments to the task instance

#### **Allow edit attachments**

If true (checked), allows the user to open an existing attachments for editing, and updating its content.



### **Fields**

Defines fields for the task instance. One or more fields can be defined for a task. A field is a placeholder for a value. All task fields are shown when a task instance is being displayed for a user in the [task list dialog](#). Fields can be useful for users to see extra information related to the task, or for users to input valuable information. They add an extra level of flexibility.

A field is always related to a [workflow variable](#), which is the real container for the field value. A field does not have a "value" it just reads/writes values from/to a workflow variable.

#### **Text caption**

It is the name of the field which will be displayed for the user

#### **Workflow variable**

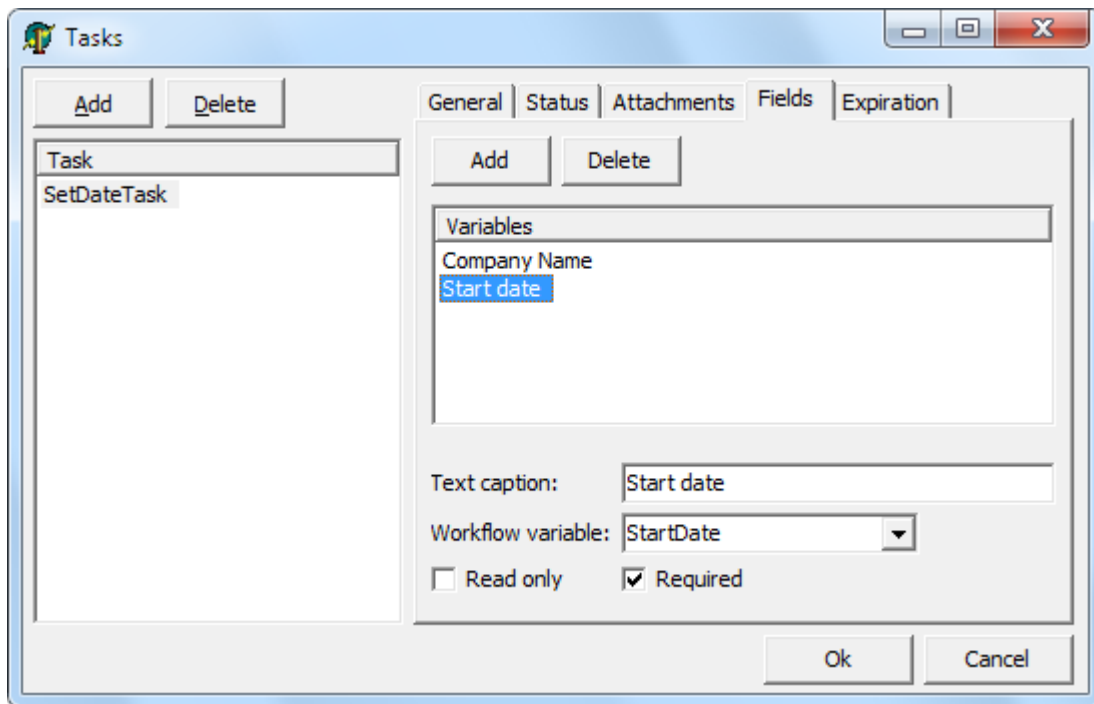
The name of the workflow variable related to the field. Fields read the value from the workflow variable, and write back the value to the workflow variable, if the user changes the value.

#### **Read only**

Mark the field as read-only, so the information is visible but not editable.

#### **Required**

When a field is required, the task instance can not be saved (altered) until a value is filled in the field. So, users cannot add, remove, edit attachments, or change task status, or change any other information in the task instance, if the field specified as required is empty.



## **Expiration**

Defines the date/time for expiration of a workflow task. If a task has a defined expiration date/time, when it exceeds this date/time without being closed by an user (changed to a completion status), its status is automatically changed to an expiration status. To use this feature check the section [Running workflow instances for tasks expiration](#).

### **Task does not expire**

This is the default option. If checked, the task will never expire, and will keep assigned to the current user/group until finished (changed to a completion status).

### **Expiration term**

Check this option to enter an expiration term for the task, and enter the amount of days, weeks or months (integer or floating point value). The expiration date will be calculated from the creation date of the task.

### **Expiration date/time**

Check this option to enter a fixed due date/time for the task.

### **Custom date/time expression**

This option allows to enter a custom expression to evaluate the task expiration date/time. The expression must return a DateTime value and may use variables from the current workflow, task properties (is common to use `_Task.CreatedOn` property to retrieve the task creation date/time and calculate the expiration date), as well as all the features allowed in [expressions](#).

### **Expiration status**

Determines the status to which the task will be automatically changed after expiring. It must be a completion status.

The screenshot shows a Windows-style dialog box titled "Tasks". It has a list on the left with "Task" and "SetDateTask". Above the list are "Add" and "Delete" buttons. The right side of the dialog has several tabs: "General", "Status", "Attachments", "Fields", and "Expiration". The "Expiration" tab is selected. It contains three radio buttons: "Task does not expire", "Expiration term:", and "Expiration date/time:". The "Expiration term:" option is selected. Below it is a text box with the value "2" and a dropdown menu showing "weeks". Below that is the "Expiration date/time:" option, which is unselected, with a date picker showing "22/10/2010" and a time picker showing "00:00:00". Below that is the "Custom date/time expression:" option, which is unselected, with a text box containing the expression `_Task.CreatedOn + 2 * 7`. At the bottom of the tab is the "Expiration status:" label and a dropdown menu showing "expired". At the very bottom of the dialog are "Ok" and "Cancel" buttons.

### 5.1.12 Approval block



**Overview:** It's a special [task block](#) which has a single approval task definition.

**Allowed inputs:** many

**Allowed outputs:** many (restricted to status list)

**Description:**

The approval block is just a task block which has a single [task definition](#). This task definition is an approval task definition and it's a regular task definition with the difference that some properties are already initialized, with the subject, description, and specially the status list.

The approval task comes with three valid status in status list: "opened", "approved" and "rejected". The approved and rejected status are completion status for the task.

You can change the approval task definition properties as you want, just like in a task block. The only difference to the task block is that you cannot create more than one task in the approval block.

### 5.1.13 Script block



**Overview:** Executes a [script](#) code.

**Allowed inputs:** many

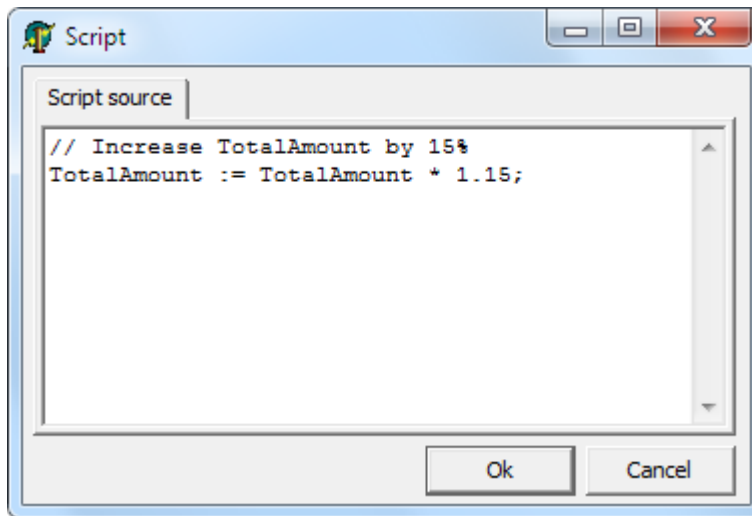
**Allowed outputs:** one (if the script does not return any value) or many (depending on possible script results)

**Description:**

The script block just executes a script code. In most cases, the script block will have only one output. But you can also use multiple outputs from script block, it will depend on script result. You can define the script result by using *result* variable:

```
result := 'result1';
```

If you have more than one output, each leaving [transition](#) should have a label, and the execution flow will take the transition which label is the same as the script result. In the example above, you must have a transition labeled "result1" so the execution will follow that path.



#### 5.1.14 Run workflow block



**Overview:** runs a separated workflow instance.

**Allowed inputs:** many

**Allowed outputs:** 1

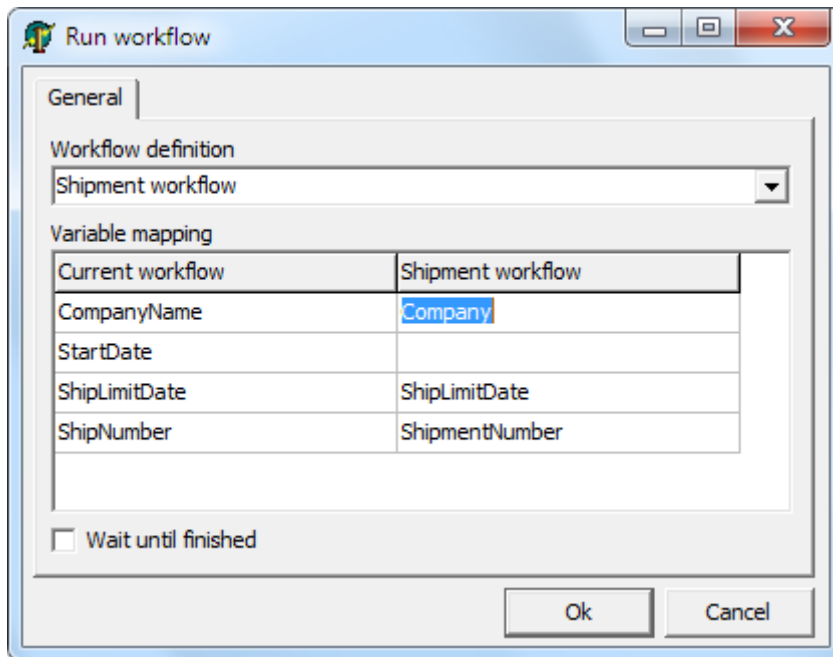
**Description:**

The run workflow block allows running a new workflow (subworkflow), separated from the current instance, in a synchronous or asynchronous way. To exchange information between the workflow instances, a variable mapping should be specified in the block definition.

If "Wait until finished" flag is checked, the current workflow will wait the subworkflow to finish, in order to continue. The variable mapping is bidirectional in this case.

If "Wait until finished" flag is unchecked, the current workflow will continue execution normally regardless of subworkflow status. Variable mapping is unidirectional only.

The variable mapping lists which variables in the subworkflow will be updated. In the screenshot example below, variable "Company" in Shipment workflow (subworkflow) will receive the value of variable "CompanyName" in the current workflow. In the case of bidirectional mapping, when Shipment workflow finishes, the value of "CompanyName" will be updated again, with the value of variable "Company".



## 5.2 Workflow variables

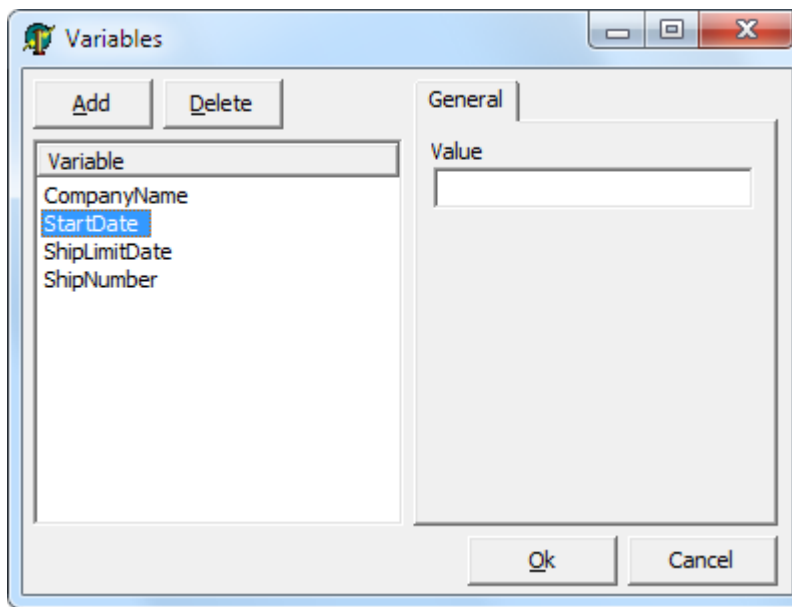
A workflow variable is a named "slot" in a workflow where you can save information, just like a variable in a program.

You create the variable in the [workflow definition](#), and each [workflow instance](#) created from the workflow definition will have a copy of that workflow variable. So, for example, a workflow definition for order processing might have the workflow variable "OrderNo". Once a workflow instance is created, the OrderNo variable will be there, and you can read/write values from/to that variable, and it's valid only for that workflow instance.

To define a workflow variable:

1. Open the [workflow definition editor](#).
2. Open the menu option *Workflow | Variables...*
3. The variable editor will be displayed.





The variable editor shows a list of defined workflow variables. You can add and delete variables using "Add" and "Delete" buttons.

All you need to do is define a name for the variable (in this example, two variables named "CompanyName" and "StartDate"). Optionally you can define a start value for the variable, in the "Value" edit box.

Once a workflow variable is defined you can use them in [expressions](#) and [scripts](#).

## 5.3 Attachments

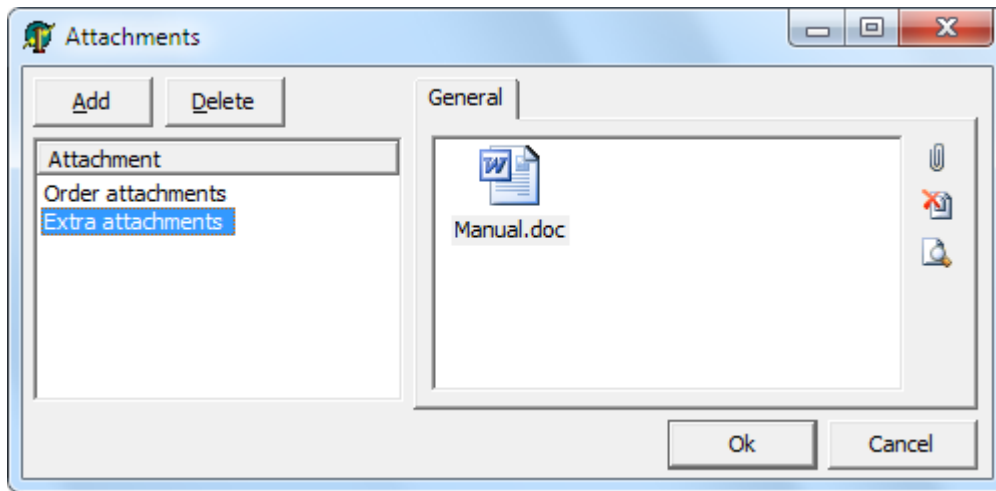
Attachments are a powerful feature that allows each [workflow instance](#) to have one or more files associated with it. Users can add, edit or remove attachments (depending on their [attachment permissions](#)) while dealing with the tasks.

As an example, a user can attach a file (or a set of files) to a task, and then another task can be created for another user, and this new user can see the file attached by the previous one. Or, users can edit and update attachments along the workflow execution. This makes stronger collaboration between users while a workflow instance is being executed.

An *attachment* is actually a container for a group of files (*attachment items*). These "containers" (attachments) can be created in the [workflow definition](#), and several attachments can be created. By default a single attachment is always created in the workflow definition, named "Attachments".

To define attachments in a workflow definition:

1. Open the [workflow definition editor](#).
2. Open the menu option *Workflow | Attachments...*
3. The attachment editor will be displayed.



Here you can create several attachments, and name each one. In the example above, two attachments were created: "Order attachments" and "Extra attachments".

Note that you can add attachment items (files) in the attachment in this window. But remember that this is a workflow definition, several workflow instances will be created from this one. If you add a file here, that file will be present in each workflow instance created. So, in general, you only create empty attachments here, and each workflow instance will have its own files in there, if any.

While an attachment is at a workflow instance level and you can manipulate attachments any time, they are strongly related to [task instances](#). When a task instance is created, it is listed in the [task list dialog](#). The attachments are also displayed in that dialog (if the task definition allowed it in the [attachment permissions](#)). And then users can add, edit or remove attachments. When the task instance is saved by the user, the files (attachment items) are updated in the attachment and are saved together with the workflow instance.

## 5.4 Expressions

Expressions are a powerful way to customize the workflow definition. You can use expressions in several properties of some blocks, specially in [task definition properties](#) of a [task block](#). When the property is about to be used, all expressions in the text are converted to their values. Expressions are identified by brackets "[" and "]". Below are examples of expression usage:

**Subject:** This is a subject about order number [OrderNo].

**Description:** Please mr. [UserName], solve this task until date [DateToStr(StartDate + 30)].

In these examples we have three expressions: OrderNo, UserName and DateToStr(StartDate + 30). Note that expressions use [workflow variables](#). The value of the workflow variable is evaluated and used in the expression. As an example, the result text, after evaluating the expressions, would be:

**Subject:** This is a subject about order number 1042.

**Description:** Please mr. Smith, solve this task until date 12-05-2007.

Besides workflow variables, expressions also accept:

[arithmetic operators](#)

+, -, /, \*, div, mod

[boolean/logical operators](#)

and, or, not, xor

relational operators

<>, =, <, >, <=, >=

bitwise operators

shl, shr

numeric constants

153 (integer), 152.43 (decimal), \$AA (hexa)

string constants

'This is a text'

char constants

#13 (return character)

Delphi-like functions and procedures

Abs

AnsiCompareStr

AnsiCompareText

AnsiLowerCase

AnsiUpperCase

Append

ArcTan

Assigned

AssignFile

Beep

Chdir

Chr

CloseFile

CompareStr

CompareText

Copy

Cos

CreateOleObject

Date

DateTimeToStr

DateToStr

DayOfWeek

Dec

DecodeDate

DecodeTime

Delete

EncodeDate

EncodeTime

EOF

Exp

FilePos

FileSize

FloatToStr

Format

FormatDateTime

FormatFloat

Frac

GetActiveOleObject

High

Inc

IncMonth  
InputQuery  
Insert  
Int  
IntToHex  
IntToStr  
IsLeapYear  
IsValidIdent  
Length  
Ln  
Low  
LowerCase  
Now  
Odd  
Ord  
Pos  
Raise  
Random  
ReadLn  
Reset  
Rewrite  
Round  
ShowMessage  
Sin  
Sqr  
Sqrt  
StrToDate  
StrToDateTime  
StrToFloat  
StrToInt  
StrToIntDef  
StrToTime  
Time  
TimeToStr  
Trim  
TrimLeft  
TrimRight  
Trunc  
UpperCase  
VarArrayCreate  
VarArrayHighBound  
VarArrayLowBound  
VarIsNull  
VarToStr  
Write  
WriteLn

## 5.5 Scripts

You can use scripts in Workflow Studio, like in the Script Block, for example.

Scripts are a powerful way to customize your workflow definition. You can do various tasks with a script block that would not be possible to do with other blocks. The default syntax language for script is Pascal. So, you can do almost everything you can do in regular Pascal.

Scripts are also a way to manipulate [workflow variables](#). In scripts they are just regular variables. For example, if you have defined a workflow variable named "TotalAmount", you can read the variable

value using this code:

```
//Calculate comission for the sale  
Comission := TotalAmount * 0.2;
```

and also change the variable value using this code:

```
//Increase TotalAmount by 15%  
TotalAmount := TotalAmount * 1.15;
```

Avoid to use message boxes and other dialogs or windows that requires user interaction for the script to continue, this might cause problems. Keep in mind that the workflow is running in background in a thread, and all actions performed by the workflow should be silent and smooth. If you need user interaction you often need to use a [task block](#) to create a task for the user.

# **Chapter**

---



**VI**

## **User interface windows**

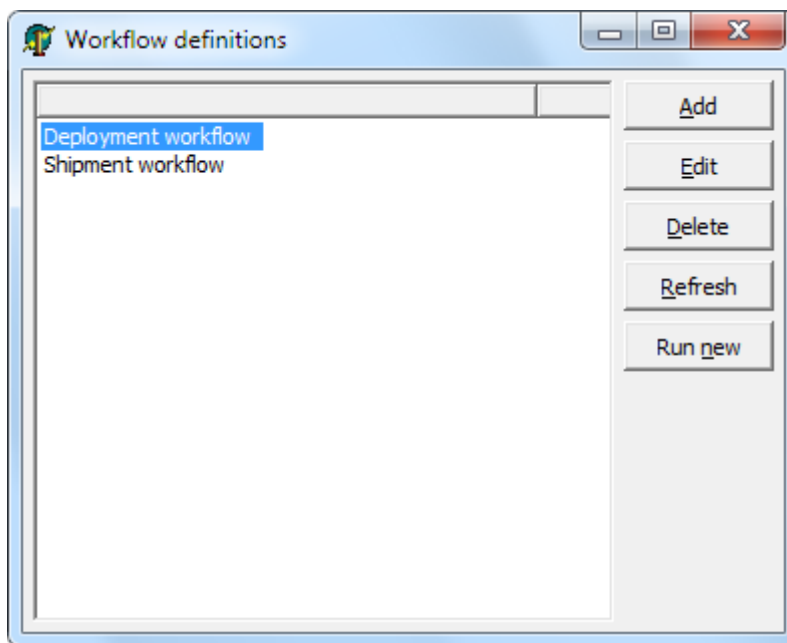
## 6 User interface windows

### 6.1 Workflow definitions dialog

The workflow definitions dialog lists all the [workflow definitions](#) saved in the database. You can open it by using TWorkflowUserInterface component:

```
WorkflowUserInterface1.ShowWorkflowDefinitionsDlg;
```

A windows like this will open:



In this window you can add, edit and delete workflow definitions. Buttons available:

**Add:** Adds a new workflow definition to database

**Edit:** Opens the [workflow definition editor](#) to edit the selected item.

**Delete:** Removes the workflow definition from the database

**Refresh:** Updates the list of workflow definitions from the database

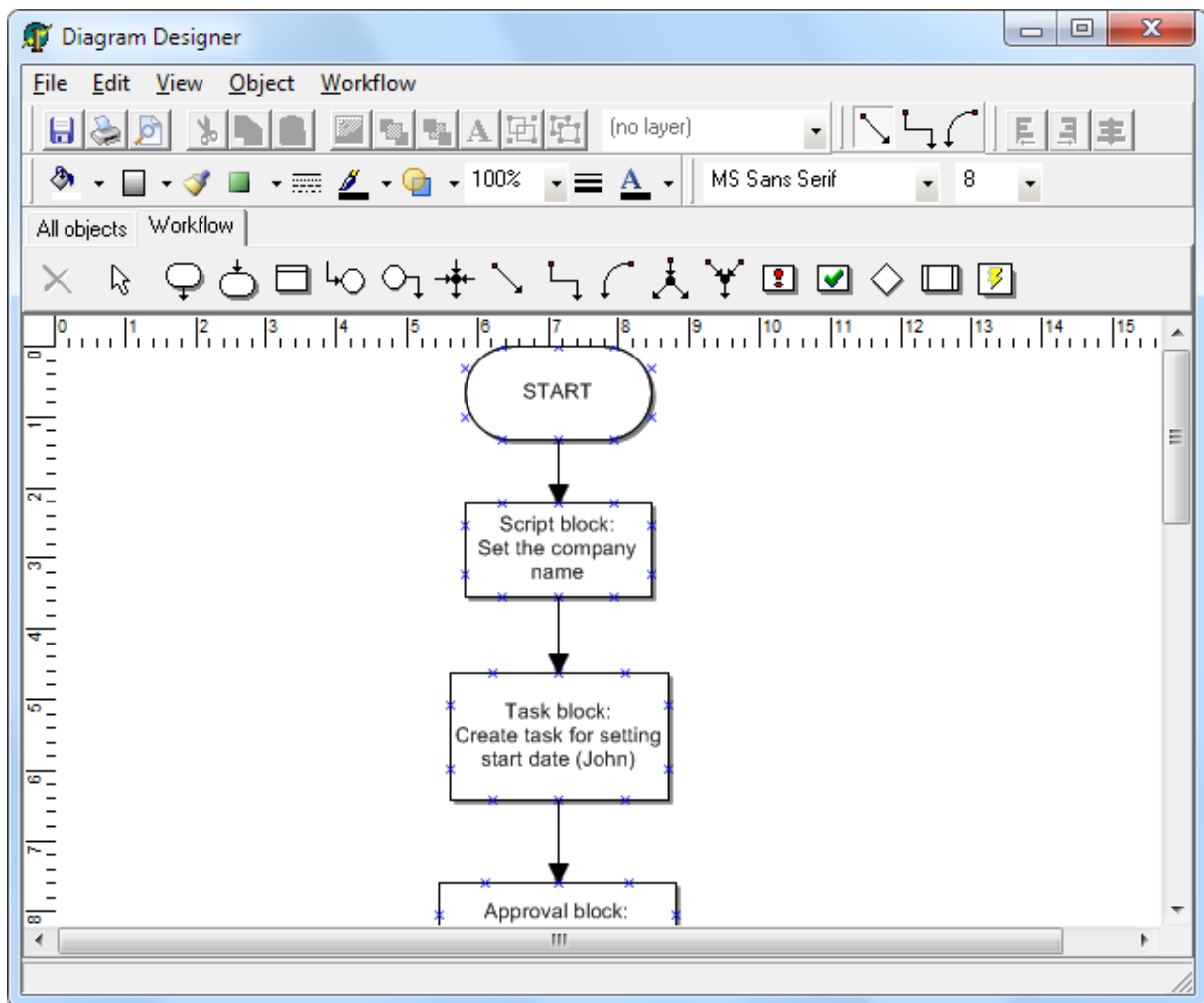
**Run new:** Creates and runs a new [workflow instance](#) based on the selected workflow definition.

### 6.2 Workflow definition editor

The workflow definition editor is where you create and edit workflow definitions. You can open it to edit a workflow definition by using TWorkflowUserInterface component:

```
WorkflowUserInterface1.EditWorkflowDefinition(MyWorkflowDefinitionObject);
```

A windows like this will open:



In the workflow definition editor you can do several tasks. Below are the most common ones:

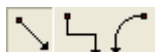
#### Adding objects to diagram

To add objects to diagram, click one of the objects in the object toolbar, and then click in the diagram at the position you want the object to be inserted (or press the mouse button, move the mouse and release the button, to insert the object with a specified size).



#### Creating transitions between objects

You can create a transition between block by choosing one of the transitions types in the transition toolbar:



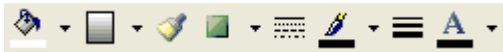
If a transition type is already selected (button pressed), you don't need to reselect the transition type every time you want to create a transition.



Once a transition type is selected, press the mouse button in any link point of the source block (a link point is the blue "x" displayed in the diagram object), drag the mouse, and then release the mouse button in any link point of the target block. A transition will be created connecting the source block to the target block.

### Changing visual appearance of diagram objects

You can change visual appearance of diagram objects by using some options in the menu and/or the toolbars. You can change background color, border color, border style, brush style, pen width, font, and more. The appearance of diagram objects do not affect the way the workflow definition will behave, it's just for visual purposes.



### Saving

Once the diagram is built, save it by choosing the menu option *File / Save*, or pressing the save button in the toolbar.

### Workflow definition options

Using the menu *Workflow*, you can click one of the submenu options to define some specific options for the workflow definition, like [workflow variables](#) and [workflow attachments](#).

### Other operations

The workflow definition editor also provides:

- preview/printing capabilities;
- clipboard operations;
- top and left rulers;
- grid;

and other features.

## 6.3 Task list dialog

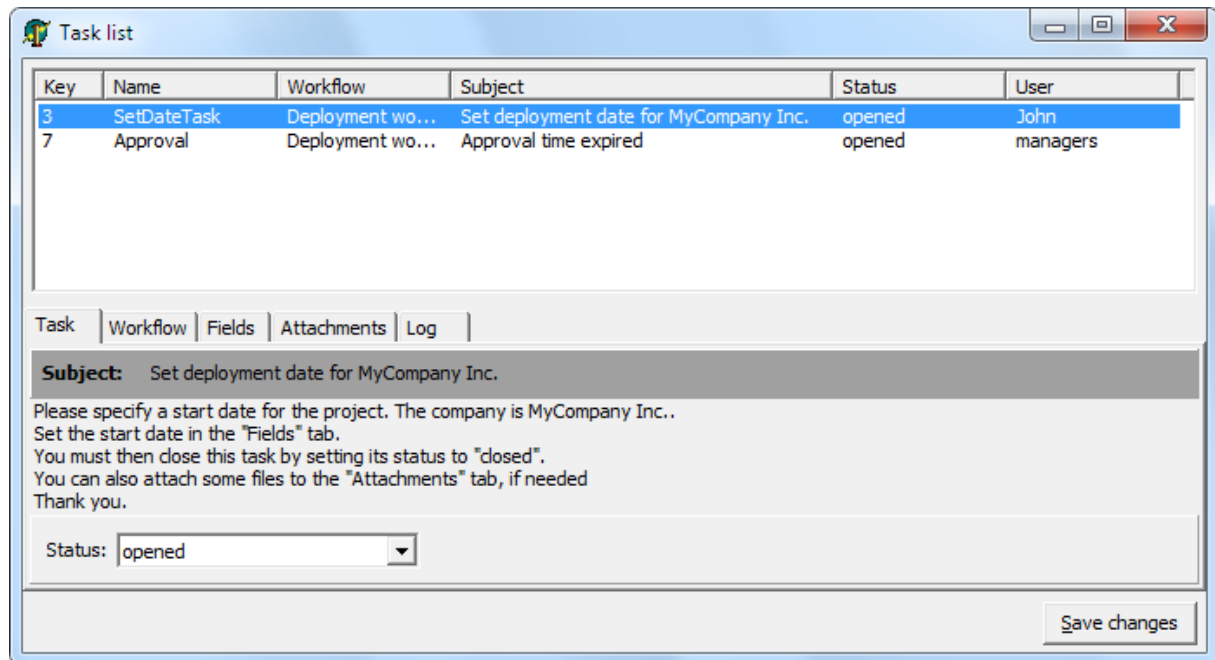
The task list dialog is the main window for user interface. It displays all the [task instances](#) assigned to an [user](#) or to a specific [workflow instance](#). In a production environment, this dialog will be used daily by users to check the pending tasks for all current business processes.

To open the task list dialog, use the TWorkflowUserInterface component:

```
//Shows all tasks assigned to the user 'scott'
WorkflowUserInterface1.ShowUserTasksDlg('scott');
//Shows all tasks assigned to users scott, tina and john
WorkflowUserInterface1.ShowUsersTasksDlg('scott,tina,john');
//Shows all tasks created from a specific workflow instance (key 29)
WorkflowUserInterface1.ShowWorkInsTasksDlg('29');
```

Note that you must pass the user id to the function, not the user name. To make the example above more clear, we considered that the user name is also the user id.

The following window will open:



### Task list view

The task list view shows all the pending tasks for the user (it can also show closed tasks by using popup menu *View / Show all tasks*). The information displayed is:

**Key:** The unique id for the task

**Name:** The name of the task definition which generated the task

**Workflow:** The name of the [workflow definition](#) which generated the workflow instance which the task belongs to.

**Subject:** The subject of the task, as specified in the task definition

**Status:** Current status of the task

**User:** The user which the task is assigned to

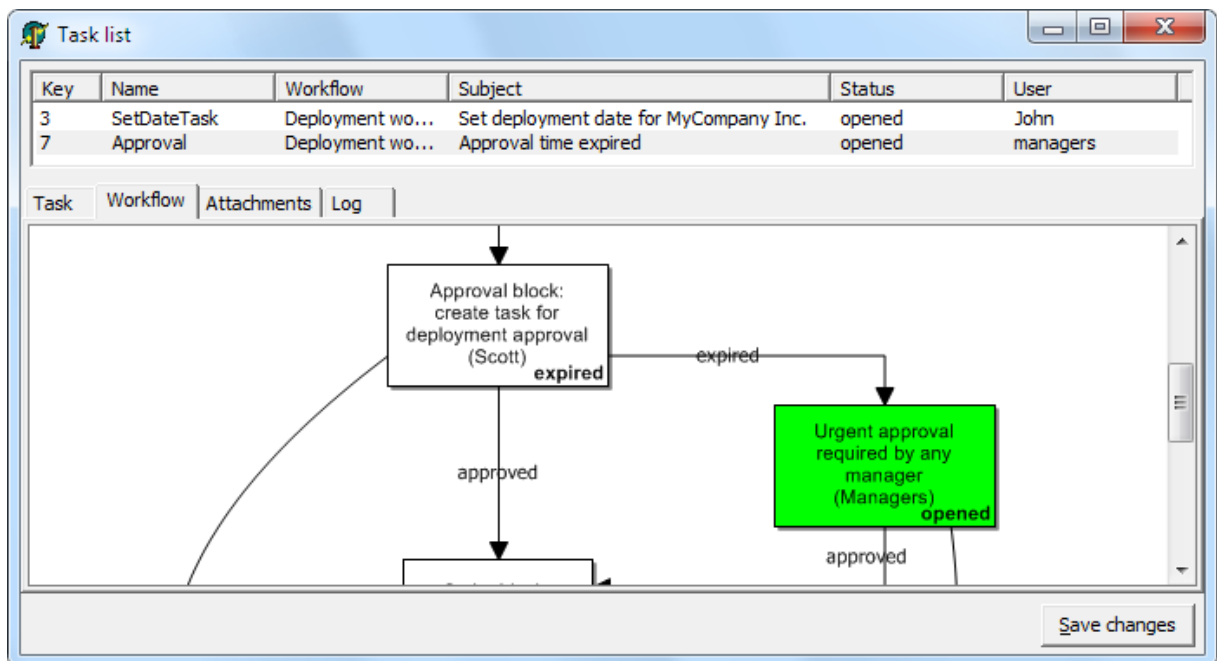
### Task tab

The task tab shows the details of the task selected in the task list view. It shows the subject, the description of the task and the current status of the task.

Users can change the status by changing the combo value.

### Workflow tab

The workflow tab shows the workflow diagram and in which situation the diagram is, related to the selected task in the task list view. It's the status of the workflow instance, not the task status. In general, if a task is opened, the workflow diagram is in a task block, because the task block waits for the task to be completed.



### Fields tab

The fields tab shows the available fields for the current selected task. The fields are defined in the task definition properties.

Here users can see the value and also change the value of fields (if the field is not read-only).

Are mentioned in the [task definition properties](#) section, field values are read and written from/to workflow variables.

Fields marked with an asterisk (\*) are required, and users cannot save changes to the task until such fields are filled.

Key	Name	Workflow	Subject	Status	User
3	SetDateTask	Deployment wo...	Set deployment date for MyCompany Inc.	opened	John
7	Approval	Deployment wo...	Approval time expired	opened	managers

Task Workflow Fields Attachments Log

Company Name: MyCompany Inc.

Start date (\*): 20/10/2010

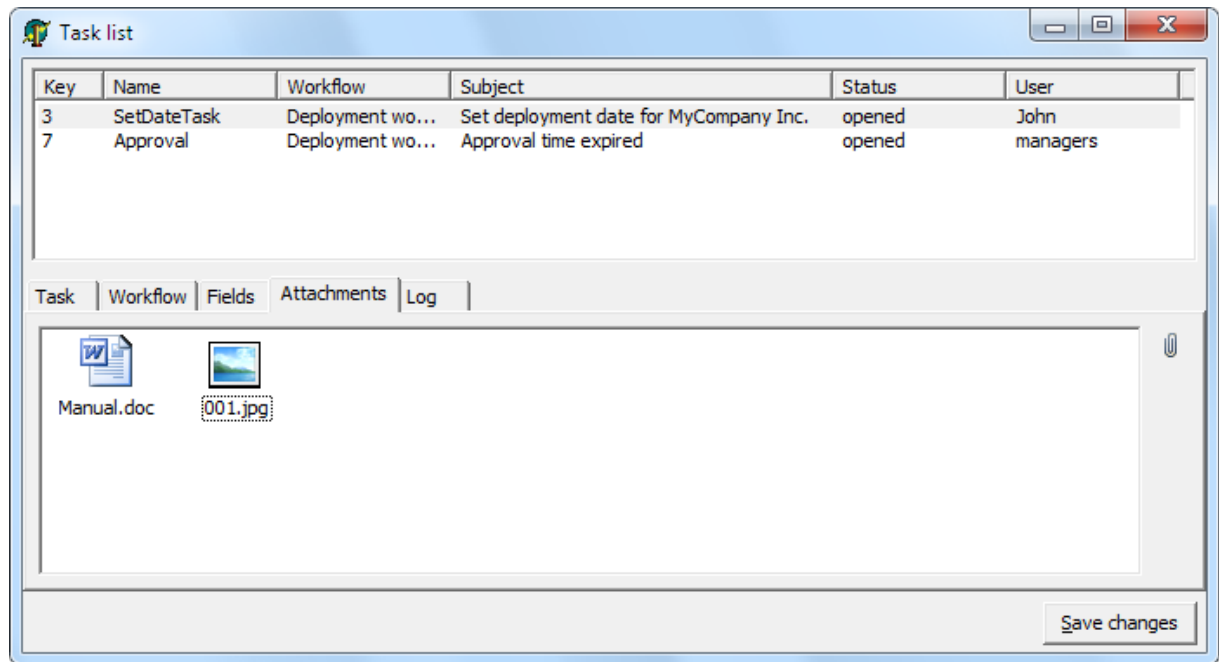
Save changes

### Attachments tab

The task list dialog shows a tab for each [attachment](#) in the workflow instance. In the screenshot below, a single attachment named "Attachments" was created, that's why the tab is named "Attachments". But you can have as many tabs as the number of attachments defined.

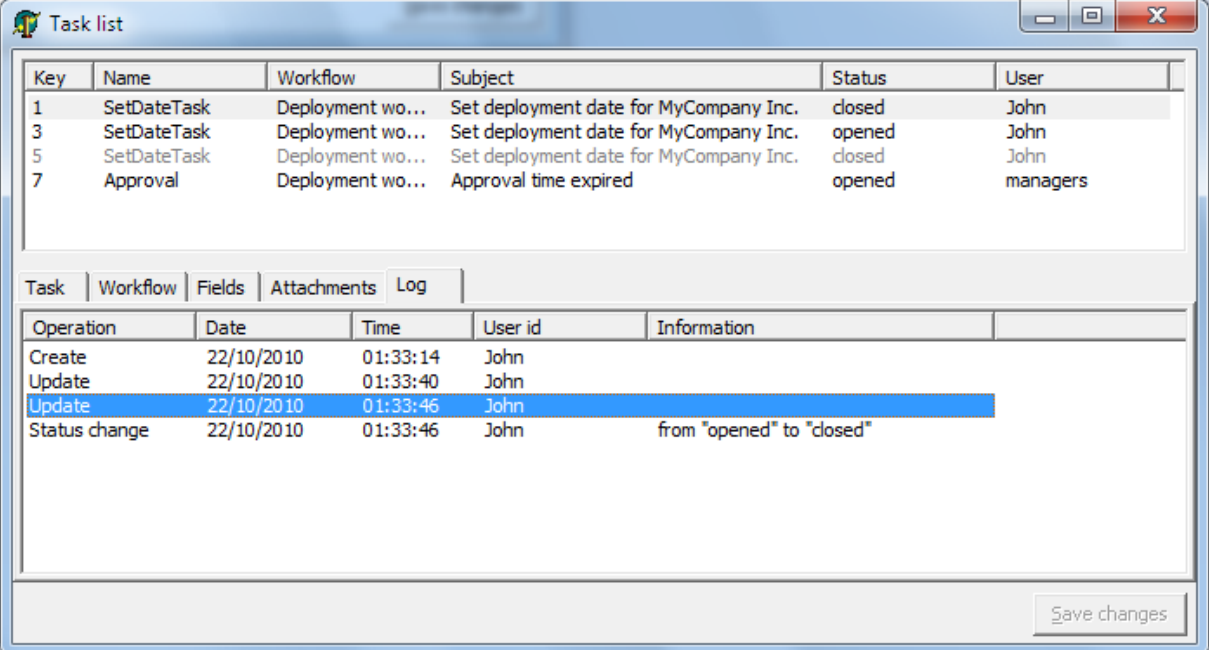
In each attachment tab you can add, remove, open and also edit attachments (depending on the attachment permissions defined in the [task definition properties](#)). You use the buttons at the right to perform these operations. You can also drag and drop files to the attachment area to add new attachment files.

The changes made to the attachment files are only saved when the user clicks the "Save changes" button.



### Log tab

The log tab shows all operations performed in the task by users. It's a log view where you can see when the task was created, updated, finished, and which user did that. The log also shows the status changes of the task.



The screenshot shows a window titled "Task list" with a standard Windows-style title bar. Inside, there are two main sections. The top section is a table with columns: Key, Name, Workflow, Subject, Status, and User. It contains four rows of task data. The bottom section is a log table with columns: Operation, Date, Time, User id, and Information. It shows a sequence of operations performed on the tasks, with the "Update" operation highlighted in blue. A "Save changes" button is located at the bottom right of the window.

Key	Name	Workflow	Subject	Status	User
1	SetDateTask	Deployment wo...	Set deployment date for MyCompany Inc.	closed	John
3	SetDateTask	Deployment wo...	Set deployment date for MyCompany Inc.	opened	John
5	SetDateTask	Deployment wo...	Set deployment date for MyCompany Inc.	closed	John
7	Approval	Deployment wo...	Approval time expired	opened	managers

Operation	Date	Time	User id	Information
Create	22/10/2010	01:33:14	John	
Update	22/10/2010	01:33:40	John	
Update	22/10/2010	01:33:46	John	
Status change	22/10/2010	01:33:46	John	from "opened" to "closed"

Save changes

# **Chapter**

---



**VII**

# **Using Workflow Studio programmatically**

## 7 Using Workflow Studio programmatically

This chapter describes how to use Workflow Studio programmatically. It lists some common tasks and how to do them from Delphi code.

### 7.1 Running an instance based on a definition name

It might be a common task to run a workflow instance based on a workflow definition name. The code below shows how to do that. See also the [full example](#) of running a workflow instance from Delphi code.

```
function TForm1.RunSomeWorkflow(ADefinitionName: string);
var
  wdf : TWorkflowDefinition;
  wfi : TWorkflowInstance;
begin
  wdf := WorkflowStudio.WorkflowManager.FindWorkflowDefinitionByName(ADefinitionName);
  wfi := WorkflowStudio.WorkflowManager.CreateWorkflowInstance(wdf);
  WorkflowStudio1.WorkflowEngine.RunWorkflow(wfi);
end;
```

### 7.2 Workflow with workflow instance variables

Workflow variables are a very used feature. You will often need to set or read workflow variable values to/from Delphi variables in order to make a strong integration of workflow with your application. The code below illustrates how to set a variable value. See also the [full example](#) of running a workflow instance.

```
var
  wfi : TWorkflowInstance;
  wvar: TWorkflowVariable;
begin
  wvar := wfi.Diagram.Variables.FindByName('OrderNo');
  if Assigned(wvar) then
    wvar.Value := AOrderNo;
end;
```

### 7.3 Running an instance from code: full example

The following procedure below is a small example which shows how to run a workflow instance from Delphi code.

It runs the instance based on a workflow definition name, passes an order number so that it is included as a variable in the workflow instance (the variable "OrderNo" must already exist in the workflow definition), runs the instance and retrieve the record id for the newly created instance.

```
// Run a workflow definition from a specified definition name, and return the record key
// in the function result. Set the workflow variable "OrderNo" from the AOrderNo parameter
function TForm1.RunSomeWorkflow(ADefinitionName: string; AOrderNo: integer): string;
var
  wdf : TWorkflowDefinition;
  wfi : TWorkflowInstance;
  wvar: TWorkflowVariable;
  i : Integer;
begin
  wdf := WorkflowStudio.WorkflowManager.FindWorkflowDefinitionByName(ADefinitionName);
  wfi := WorkflowStudio.WorkflowManager.CreateWorkflowInstance(wdf);
  result := wfi.Key;
  wvar := wfi.Diagram.Variables.FindByName('OrderNo');
```

```

    if Assigned(wvar) then
        wvar.Value := AOrderNo;
        WorkflowStudio1.WorkflowEngine.RunWorkflow(wfi);
    end;

```

## 7.4 Retrieve the list of tasks for a specified user

The code below retrieves the list of tasks for a specified user. You can also check the Count property of the list to see if the user has no open tasks assigned to him/her (Count = 0 means no tasks).

```

function TForm1.CreateTaskListForUser(UserID: string) :TTaskInstanceList;
var
begin
    result := TTaskInstanceList.Create(TTaskInstanceItem);
    WorkflowStudio.TaskManager.LoadTaskInstanceList(result, tfUser, UserID, true);
end;

```

## 7.5 Creating and editing an workflow definition

The code below creates a new workflow definition in database named "order processing" and opens the workflow definition editor for editing the newly created definition.

```

procedure TForm1.CreateAndEditDefinition(AName: string);
var
    wdf : TWorkflowDefinition;
begin
    //First check if the workflow definition already exists
    wdf := WorkflowStudio.WorkflowManager.FindWorkflowDefinitionByName(AName);
    if not Assigned(wdf) then
    begin
        wdf := TWorkflowDefinition.Create(nil);
        wdf.Name := AName;
        WorkflowStudio.WorkflowManager.SaveWorkflowDefinition(wdf);
    end;
    WorkflowStudio.UserInterface.EditWorkflowDefinition(wdf);
    wdf.Free;
end;
...
begin
    CreateAndEditDefinition('order processing');
end;

```

## 7.6 Running workflow instances for expired tasks

Workflow tasks can have defined a date/time for [expiration](#). If a task has a defined expiration date/time, when it exceeds this date/time without being closed by a user (changed to a completion status), its status is automatically changed to an expiration status.

However, this expiration of tasks is not done automatically by Workflow Studio. Since it requires a monitor being executed periodically to run the pending workflows and check for the tasks to be expired, you have to implement this monitor in your application, according to your needs (for example, a program scheduled in system task scheduler, a service, or even a timer inside the application).

All that needs to be done by this monitor is call a method from workflow engine:

```

WorkflowStudio.WorkflowEngine.RunPendingWorkflowInstances;

```



# Chapter

---



## Extending the scripting system

## 8 Extending the scripting system

Workflow Studio provides a scripting system which can be used in several places of Workflow Studio framework. Expressions use the scripting system, and the script block as well, which runs scripts. You can increase integration between Workflow Studio and your Delphi application by extended the scripting system.

You can register your own classes, methods, properties, variables, functions and procedures so they can be accessible from script. The following topics describe some common tasks to extend the scripting system.

The examples shown in the following topics use a Scripter object. This Scripter must be "prepared" with definition of custom methods, properties etc. To access this Scripter object, implement a handler to the `TWorkflowStudio.OnInitializeScriptEngine` event:

```
procedure TfmMain.WorkflowStudio1InitializeScriptEngine(AEngine: TWorkflowScriptEngine);
begin
    // prepare the workflow scripter
    TScripterEngine(AEngine).Scripter.DefineMethod(...);
end;
```

### 8.1 Accessing Delphi objects

The following topics show how to register Delphi objects in the scripting system.

#### 8.1.1 Registering Delphi components

One powerful feature of scripter is to access Delphi objects. This way you can make reference to objects in script, change its properties, call its methods, and so on. However, every object must be registered in scripter so you can access it. For example, suppose you want to change caption of form (named Form1). If you try to execute this script:

```
SCRIPT:
Form1.Caption:='New caption';
```

you will get "Unknown identifier or variable not declared: Form1". To make scripter work, use `AddComponent` method:

```
CODE:
Scripter.AddComponent(Form1);
```

Now scripter will work and form's caption will be changed.

#### 8.1.2 Access to published properties

After a component is added, you have access to its published properties. That's why the caption property of the form could be changed. Otherwise you would need to register property as well. Actually, published properties are registered, but scripter does it for you.

#### 8.1.3 Class registering structure

Scripter can call methods and properties of objects. But this methods and properties must be registered in scripter. The key property for this is `TatCustomScripter.Classes` property. This property holds a collection of registered classes (`TatClass` object), which in turn holds its collection of registered properties and methods (`TatClass.Methods` and `TatClass.Properties`). Each registered method and

property holds a name and the wrapper method (the Delphi written code that will handle method and property).

When you registered Form1 component in the previous example, scripter automatically registered TForm class in Classes property, and registered all published properties inside it. To access methods and public properties, you must register them, as showed in the following topics.

### 8.1.4 Calling methods

To call an object method, you need to register it. For instance, if you want to call ShowModal method of a newly created form named form2. So we must add the form it to scripter using AddComponent method, and then register ShowModal method:

#### CODE:

```
procedure TForm1.ShowModalProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(TCustomForm(CurrentObject).ShowModal);
end;

procedure TForm1.PrepareScript;
begin
    Scripter.AddComponent(Form2);
    With Scripter.DefineClass(TCustomForm) do
        begin
            DefineMethod('ShowModal', 0, tkInteger, nil, ShowModalProc);
        end;
end;
```

#### SCRIPT:

```
ShowResult:=Form2.ShowModal;
```

This example has a lot of new concepts. First, component is added with AddComponent method. Then, DefineClass method was called to register TCustomForm class. DefineClass method automatically check if TCustomForm class is already registered or not, so you don't need to do test it.

After that, ShowModal is registered, using DefineMethod method. Declaration of DefineMethod is:

```
function DefineMethod(AName:string; AArgCount:integer; AResultDataType: TatTypeKind;
AResultClass:TClass; AProc:TMachineProc; AIsClassMethod:boolean=false): TatMethod;
```

**AName** receives 'ShowModal' – it's the name of method to be used in script.

**AArgCount** receives 0 – number of input arguments for the method (none, in the case of ShowModal)

**AResultDataType** receives tkInteger – it's the data type of method result. ShowModal returns an integer. If method is not a function but a procedure, AResultDataType should receive tkNone.

**AResultClass** receives nil – if method returns an object (not this case), then AResultClass must contain the object class. For example, TField.

**AProc** receives ShowModalProc – the method written by the user that works as ShowModal wrapper.

And, finally, there is ShowModalProc method. It is a method that works as the wrapper: it implements a call to ShowModal. In this case, it uses some useful methods and properties of TatVirtualMachine class:

property CurrentObject – contains the instance of object where the method belongs to. So, it contains the instance of a specified TCustomForm.

method ReturnOutputArg – it returns a function result to scripter. In this case, returns the value returned by TCustomForm.ShowModal method.

### 8.1.5 More method calling examples

In addition to previous example, this one illustrates how to register and call methods that receive parameters and return classes. In this example, FieldByName:

**SCRIPT:**

```
AField:=Table1.FieldByName('CustNo');
ShowMessage(AField.DisplayLabel);
```

**CODE:**

```
procedure TForm1.FieldByNameProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(integer(TDataset(CurrentObject).FieldByName(GetInputArgAsString(0))));
end;

procedure TForm1.PrepareScript;
begin
    Scripter.AddComponent(Table1);
    With Scripter.DefineClass(TDataset) do
        begin
            DefineMethod('FieldByName',1,tkClass,TField,FieldByNameProc);
        end;
end;
```

Very similar to [Calling methods](#) example. Some comments:

- FieldByName method is registered in TDataset class. This allows use of FieldByName method by any TDataset descendant inside script. If FieldByName was registered in a TTable class, script would not recognize the method if component was a TQuery
- DefineMethod call defined that FieldByName receives one parameters, its result type is tkClass, and class result is TField.
- Inside FieldByNameProc, GetInputArgAsString method is called in order to get input parameters. The 0 index indicates that we want the first parameter. For methods that receive 2 or more parameters, use GetInputArg(1), GetInputArg(2), and so on.
- To use ReturnOutputArg in this case, we need to cast resulting TField as integer. This must be done to return any object. This is because ReturnOutputArg receives a Variant type, and objects must then be cast to integer

### 8.1.6 Accessing non-published properties

Just like methods, properties that are not published must be registered. The mechanism is very similar to method registering, with the difference we must indicate one wrapper to get property value and another one to set property value. In the following example, the "AsFloat" property of TField class is registered:

**SCRIPT:**

```
AField:=Table1.FieldByName('Company');
ShowMessage(AField.Value);
```

**CODE:**

```
procedure TForm1.GetFieldValueProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(TField(CurrentObject).Value);
end;

procedure TForm1.SetFieldValueProc(AMachine: TatVirtualMachine);
```

```

begin
  With AMachine do
    TField(CurrentObject).Value:=GetInputArg(0);
  end;

  procedure TForm1.PrepareScript;
  begin
    With Scripter.DefineClass(TField) do
      begin
        DefineProp('Value',tkVariant,GetFieldValueProc,SetFieldValueProc);
      end;
    end;
  end;
end;

```

DefineProp is called passing a tkVariant indicating that Value property is Variant type, and then passing two methods GetFieldValueProc and SetFieldValueProc, which, in turn, read and write value property of a field object. Note that in SetFieldValueProc method was used GetInputArg (instead of GetInputArgAsString). This is because GetInputArg returns a variant.

### 8.1.7 Registering indexed properties

A property can be indexed, specially when it is a TCollection descendant. This applies to dataset fields, grid columns, string items, and so on. So, the code below illustrates how to register indexed properties. In this example, Strings property of TStrings object is added in other to change memo content:

#### SCRIPT:

```

ShowMessage(Memo1.Lines.Strings[3]);
Memo1.Lines.Strings[3]:=Memo1.Lines.Strings[3]+' with more text added';

//This is a comment

```

#### CODE:

```

procedure TForm1.GetStringsProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ReturnOutputArg(TStrings(CurrentObject).Strings[GetArrayIndex(0)]);
  end;

  procedure TForm1.SetStringsProc(AMachine: TatVirtualMachine);
  begin
    With AMachine do
      TStrings(CurrentObject).Strings[GetArrayIndex(0)]:=GetInputArgAsString(0);
    end;

    procedure TForm1.PrepareScript;
    begin
      Scripter.AddComponent(Memo1);
      With Scripter.DefineClass(TStrings) do
        begin
          DefineProp('Strings',tkString,GetStringsProc,SetStringsProc,nil,false,1);
        end;
      end;
    end;
  end;
end;

```

Some comments:

- DefineProp receives three more parameters than DefineMethod: **nil** (class type of property. It's nil because property is string type), **false** (indicating the property is not a class property) and **1** (indicating that property is indexed by 1 parameter. This is the key param. For example, to register Cells property of the grid, this parameter should be 2, since Cells depends on Row and Col).
- In GetStringsProc and SetStringsProc, GetArrayIndex method is used to get the index value passed by script. The 0 param indicates that it is the first index (in the case of Strings property, the only one).

### 8.1.8 Retrieving name of called method or property

You can register the same wrapper for more than one method or property. In this case, you might need to know which property or method was called. In this case, you can use `CurrentPropertyName` or `CurrentMethodName`. The following example illustrates this usage

```
procedure TForm1.GenericMessageProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    If CurrentMethodName = 'MessageHello' then
      ShowMessage('Hello')
    Else if CurrentMethodName = 'MessageWorld' then
      ShowMessage('World');
end;

procedure TForm1.PrepareScript;
begin
  With Scripter do
    begin
      DefineMethod('MessageHello', 1, tkNone, nil, GenericMessageProc);
      DefineMethod('MessageWorld', 1, tkNone, nil, GenericMessageProc);
    end;
end;
```

### 8.1.9 Registering methods with default parameters

You can also register methods which have default parameters in scripter. To do that, you must pass the number of default parameters in the `DefineMethod` property. Then, when implementing the method wrapper, you need to check the number of parameters passed from the script, and then call the Delphi method with the correct number of parameters. For example, let's say you have the following procedure declared in Delphi:

```
function SumNumbers(A, B: double; C: double = 0; D: double = 0; E: double = 0):
double;
```

To register that procedure in scripter, you use `DefineMethod` below. Note that the number of parameters is 5 (five), and the number of default parameters is 3 (three):

```
Scripter.DefineMethod('SumNumbers', 5 {number of total parameters}, tkFloat, nil,
SumNumbersProc, false, 3 {number of default parameters});
```

Then, in the implementation of `SumNumbersProc`, just check the number of input parameters and call the function properly:

```
procedure TForm1.SumNumbersProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    begin
      Case InputArgCount of
        2: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1)));
        3: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
          GetInputArgAsFloat(2)));
        4: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
          GetInputArgAsFloat(2), GetInputArgAsFloat(3)));
        5: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
          GetInputArgAsFloat(2), GetInputArgAsFloat(3), GetInputArgAsFloat(4)));
      end;
    end;
end;
```

## 8.2 Accessing Delphi functions, variables and constants

The following topics describe how to register regular procedures, functions and global variables in scripting system.

### 8.2.1 Overview

In addition to access Delphi objects, scripter allows integration with regular procedures and functions, global variables and global constants. The mechanism is very similar to accessing Delphi objects. In fact, scripter internally consider regular procedures and functions as methods, and global variables and constants are props.

### 8.2.2 Registering global constants

Registering a constant is a simple task in scripter: use `AddConstant` method to add the constant and the name it will be known in scripter:

**CODE:**

```
Scripter.AddConstant('MaxInt',MaxInt);
Scripter.AddConstant('Pi',pi);
Scripter.AddConstant('MyBirthday',EncodeDate(1992,5,30));
```

**SCRIPT:**

```
ShowMessage('Max integer is '+IntToStr(MaxInt));
ShowMessage('Value of pi is '+FloatToStr(pi));
ShowMessage('I was born on '+DateToStr(MyBirthday));
```

Access the constants in script just like you do in Delphi code.

### 8.2.3 Accessing global variables

To register a variable in scripter, you must use `AddVariable` method. Variables can be added in a similar way to constants: passing the variable name and the variable itself. In addition, you can also add variable in the way you do with properties: use a wrapper method to get variable value and set variable value:

**CODE:**

```
var
  MyVar: Variant;
  ZipCode: string[15];

procedure TForm1.GetZipCodeProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ReturnOutputArg(ZipCode);
end;

procedure TForm1.SetZipCodeProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ZipCode:=GetInputArgAsString(0);
end;

procedure TForm1.PrepareScript;
begin
  Scripter.AddVariable('ShortDateFormat',ShortDateFormat);
  Scripter.AddVariable('MyVar',MyVar);
  Scripter.DefineProp('ZipCode',tkString,GetZipCodeProc,SetZipCodeProc);
```

```

    Scripter.AddObject('Application',Application);
end;

procedure TForm1.Run1Click(Sender: TObject);
begin
    PrepareScript;
    MyVar:='Old value';
    ZipCode:='987654321';
    Application.Tag:=10;
    Scripter.SourceCode:=Memo1.Lines;
    Scripter.Execute;
    ShowMessage('Value of MyVar variable in Delphi is '+VarToStr(MyVar));
    ShowMessage('Value of ZipCode variable in Delphi is '+VarToStr(ZipCode));
end;

```

**SCRIPT:**

```

ShowMessage('Today is '+DateToStr(Date)+' in old short date format');
ShortDateFormat:='dd-mmmm-yyyy';
ShowMessage('Now today is '+DateToStr(Date)+' in new short date format');

ShowMessage('My var value was "'+MyVar+'"');
MyVar:='My new var value';

ShowMessage('Old Zip code is '+ZipCode);
ZipCode:='109020';

ShowMessage('Application tag is '+IntToStr(Application.Tag));

```

**8.2.4 Calling regular functions and procedures**

In scripter, regular functions and procedures are added like methods. The difference is that you don't add the procedure in any class, but in scripter itself, using DefineMethod method. The example below illustrates how to add QuotedStr and StringOfChar methods:

**SCRIPT:**

```

ShowMessage(QuotedStr(StringOfChar('+',3)));

```

**CODE:**

```

{ TSomeLibrary }
procedure TSomeLibrary.Init;
begin
    Scripter.DefineMethod('QuotedStr',1,tkString,nil,QuotedStrProc);
    Scripter.DefineMethod('StringOfChar',2,tkString,nil,StringOfCharProc);
end;

procedure TSomeLibrary.QuotedStrProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(QuotedStr(GetInputArgAsString(0)));
end;

procedure TSomeLibrary.StringOfCharProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(StringOfChar(GetInputArgAsString(0)[1],GetInputArgAsInteger(1)
));
end;

procedure TForm1.Run1Click(Sender: TObject);
begin

```



```
Scripter.AddLibrary(TSomeLibrary);
Scripter.SourceCode:=Memol.Lines;
Scripter.Execute;
end;
```

Since there is no big difference from defining methods, the example above introduces an extra concept: libraries. Note that the way methods are defined didn't change (a call to DefineMethod) and neither the way wrapper are implemented (QuotedStrProc and StringOfCharProc). The only difference is the way they are located: instead of TForm1 class, they belong to a different class named TSomeLibrary. The following topic covers the use of libraries

## 8.3 Using libraries

### 8.3.1 Overview

Libraries are just a concept of extending scripter by adding more components, methods, properties, classes to be available from script. You can do that by manually registering a single component, class or method. A library is just a way of doing that in a more organized way.

### 8.3.2 Delphi-based libraries

In script, you can use libraries for registered methods and properties. Look at the two codes below, the first one uses libraries and the second use the mechanism used in this doc until now:

#### CODE 1:

```
type
  TExampleLibrary = class(TatScripterLibrary)
  protected
    procedure CurrToStrProc(AMachine: TatVirtualMachine);
    procedure Init; override;
    class function LibraryName: string; override;
  end;

class function TExampleLibrary.LibraryName: string;
begin
  result := 'Example';
end;

procedure TExampleLibrary.Init;
begin
  Scripter.DefineMethod('CurrToStr', 1, tkInteger, nil, CurrToStrProc);
end;

procedure TExampleLibrary.CurrToStrProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ReturnOutputArg(CurrToStr(GetInputArgAsFloat(0)));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Scripter.AddLibrary(TExampleLibrary);
  Scripter.SourceCode:=Memol.Lines;
  Scripter.Execute;
end;
```

#### CODE 2:

```
procedure TForm1.PrepareScript;
begin
  Scripter.DefineMethod('CurrToStr', 1, tkInteger, nil, CurrToStrProc);
```

```

end;

procedure TForm1.CurrToStrProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(CurrToStr(GetInputArgAsFloat(0)));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    PrepareScript;
    Scripter.SourceCode:=Mem1.Lines;
    Scripter.Execute;
end;

```

Both codes do the same: add CurrToStr procedure to script. Note that scripiter initialization method (Init in Code 1 and PrepareScript in Code 2) is the same in both codes. And so is CurrToStrProc method – no difference. The two differences between the code are:

- The class where the methods belong to. In Code1, methods belong to a special class named TExampleLibrary, which descends from TatScripterLibrary. In Code 2, the belong to the current form (TForm1).
- In Code 1, scripiter preparation is done adding TExampleLibrary class to scripiter, using AddLibrary method. In Code 2, PrepareScript method is called directly.

So when to use one way or another? There is no rule – use the way you feel more comfortable. Here are pros and cons of each:

#### **Declaring wrapper and preparing methods in an existing class and object:**

- Pros: More convenient. Just create a method inside form, or datamodule, or any object.
- Cons: When running script, you must be sure that object is instantiated. It's more difficult to reuse code (wrapper and preparation methods)

#### **Using libraries, declaring wrapper and preparing methods in a TatScripterLibrary class descendant:**

- Pros: No need to check if class is instantiated – scripiter does it automatically. It is easy to port code – all methods are inside a class library, so you can add it in any scripiter you want, put it in a separate unit, etc..
- Cons: Just the extra work of declaring the new class

In addition to using AddLibrary method, you can use RegisterScripterLibrary procedure. For example:

```

RegisterScripterLibrary(TExampleLibrary);
RegisterScripterLibrary(TAnotherLibrary, True);

```

RegisterScripterLibrary is a global procedure that registers the library in a global list, so all scripiter components are aware of that library. The second parameter of RegisterScripterLibrary indicates if the library is load automatically or not. In the example above, TAnotherLibrary is called with Explicit Load (True), while TExampleLibrary is called with Explicit Load false (default is false).

When explicit load is false (case of TExampleLibrary), every scripiter that is instantiated in application will automatically load the library.

When explicit load is true (case of TAnotherLibrary), user can load the library dinamically by using uses directive:

#### **SCRIPT:**

```
Uses Another;
```

```
//Do something with objects and procedures register by TatAnotherLibrary
```

Note that "Another" name is informed by `TatAnotherLibrary.LibraryName` class method.

### 8.3.3 The `TatSystemLibrary` library

There is a library that is added by default to all scripiter components, it is the `TatSystemLibrary`. This library is declared in the `uSystemLibrary` unit. It adds commonly used routines and functions to scripiter, such like `ShowMessage` and `IntToStr`.

#### Functions added by `TatSystemLibrary`

The following functions are added by the `TatSystemLibrary` (refer to Delphi documentation for an explanation of each function):

- Abs
- AnsiCompareStr
- AnsiCompareText
- AnsiLowerCase
- AnsiUpperCase
- Append
- ArcTan
- Assigned
- AssignFile
- Beep
- Chdir
- Chr
- CloseFile
- CompareStr
- CompareText
- Copy
- Cos
- CreateOleObject
- Date
- DateTimeToStr
- DateToStr
- DayOfWeek
- Dec
- DecodeDate
- DecodeTime
- Delete
- EncodeDate
- EncodeTime
- EOF
- Exp
- FilePos
- FileSize
- FloatToStr
- Format
- FormatDateTime
- FormatFloat
- Frac
- GetActiveOleObject
- High
- Inc
- IncMonth
- InputQuery
- Insert
- Int
- Interpret (\*)

IntToHex  
IntToStr  
IsLeapYear  
IsValidIdent  
Length  
Ln  
Low  
LowerCase  
Machine (\*)  
Now  
Odd  
Ord  
Pos  
Raise  
Random  
ReadLn  
Reset  
Rewrite  
Round  
Scripter (\*)  
SetOf (\*)  
ShowMessage  
Sin  
Sqr  
Sqrt  
StrToDate  
StrToDateTime  
StrToFloat  
StrToInt  
StrToIntDef  
StrToTime  
Time  
TimeToStr  
Trim  
TrimLeft  
TrimRight  
Trunc  
UpperCase  
VarArrayCreate  
VarArrayHighBound  
VarArrayLowBound  
VarIsNull  
VarToStr  
Write  
WriteLn

All functions/procedures added are similar to the Delphi ones, with the exception of those marked with a "\*", explained below:

```
procedure Interpret(Ascript: string);
```

Executes the script source code specified by Ascript parameter

```
function Machine: TatVirtualMachine;
```

Returns the current virtual machine executing the script.

```
function Scripter: TatCustomScripter;
```

Returns the current scripter component.

```
function SetOf(array): integer;  
Returns a set from the array passed. For example:
```

```
MyFontStyle := SetOf([fsBold, fsItalic]);
```

### 8.3.4 Removing functions from the System library

To remove a function from the system library, avoiding the end-user to use the function from the script, you just need to destroy the associated method object in the SystemLibrary class:

```
MyScripter.SystemLibrary.MethodByName('ShowMessage').Free;
```

