

Trilogy 2.3

Installation and Administration Guide

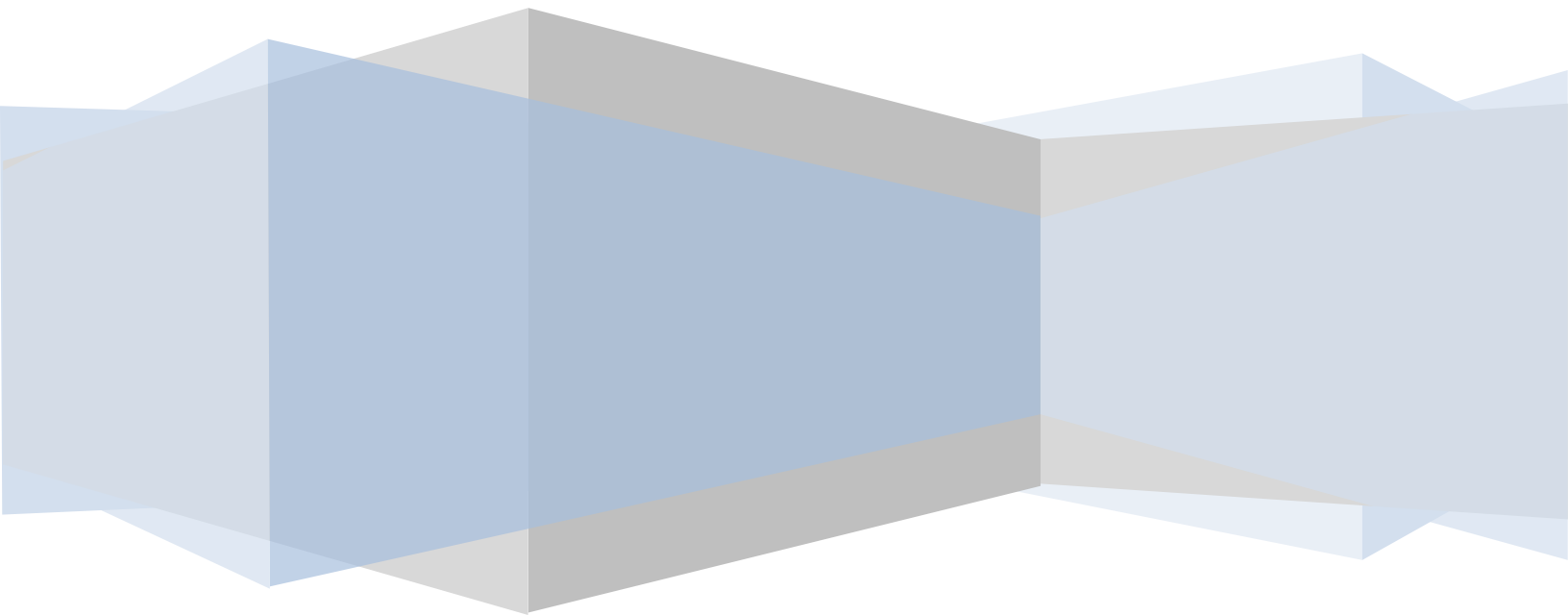


Table of Contents

1	Introduction	6
1.1	What is Trilogy?	6
1.2	About This Book	7
1.3	Typographical Conventions	7
1.4	Symbols Used in This Book	8
2	Trilogy – Overview.....	9
2.1	Trilogy Jobs	9
2.2	The Trilogy Configuration File	10
2.2.1	Client.....	10
2.2.2	Server	10
2.3	Running Server Side Scripts with <i>Trilogy</i>	13
2.4	Trilogy Client Service	14
3	Installation Guide	16
3.1	Installing a Trilogy Server.....	16
3.1.1	Windows	16
3.1.2	Unix	19
3.2	Installing a Trilogy Client.....	21
3.2.1	Windows	21
3.2.2	Unix	23
3.3	Licensing the Trilogy Server.....	25
3.4	Starting the Trilogy Server	27
3.4.1	Windows	27
3.4.2	Unix	28
3.5	Testing the installation	28
3.5.1	UNIX client.....	28
3.5.2	Windows client.....	29
3.6	Stopping the Trilogy Server	30
3.6.1	Windows	30
3.6.2	Unix	30
4	Trilogy Dialogs	31
4.1	Creating Dialogs.....	31
4.2	Differences between Unix and Windows <i>Trilogy</i> Dialogs	35
4.3	Creating Password Fields	36
4.4	Adding a Title to the Dialog.....	37
4.5	Creating Drop Down Lists	38
4.6	Pre-Populating Dialogs	41
4.7	Pre-Populating Dialogs Containing Drop Down Lists.....	43
4.8	Checkboxes and Radio Buttons.....	45
4.8.1	Creating Checkboxes and Radio Buttons.....	45
4.8.2	Pre-populating Dialogs containing Radio Buttons and Checkboxes	47

4.9	Disabling Fields	48
4.10	Renaming the Buttons	48
4.11	Validating Dialog Data	49
4.12	Validating Command Line Parameters Before Displaying Dialog	51
4.13	Using the Same Script to perform Multiple Functions.....	52
4.14	Creating Named Frames in Dialogs	54
4.15	Grouping Radio Buttons with Named Frames.....	56
4.16	Creating Tabbed Dialogs.....	58
4.17	Adding Banners	60
4.18	Creating Dialogs with Scripts	63
4.19	Creating an Icon in the System Tray	65

5 The List Box 67

5.1	Introduction.....	67
5.2	List Box Directives	68
5.2.1	Controlling the List Box Appearance	68
5.2.2	Creating List Box Content.....	68
5.2.3	Identifying List Box Column Names	69
5.2.4	Identifying List Box Column Widths	70
5.2.5	Auto Sizing Columns.....	71
5.2.6	Creating Hidden Columns	72
5.2.7	Controlling List Box Width	72
5.2.8	Controlling List Box Height	73
5.2.9	Adding an "Apply" Button to the Dialog	73
5.2.10	Sorting the List Box.....	73
5.2.11	Automatic Sorting	73
5.2.12	Controlling List Box Selections	74
5.2.13	Selecting List Box Rows	75
5.2.14	Automatically Selecting List Box Rows	76
5.2.15	Adding a Right-Click Menu to the List Box	79
5.2.16	Automatically Refreshing the List Box	81
5.2.17	Adding Double Click to the List Box	81
5.2.18	Adding Icons to Each Row	82
5.2.19	Row Context.....	82
5.2.20	Controlling List Box Script Execution.....	85
5.3	Example – Building a List Box Application	86

6 Linking Fields..... 91

6.1	Introduction to Linked Fields	91
6.2	Creating Linked Fields.	91
6.3	Linking Drop Down Lists	91
6.4	Linking radio buttons and checkboxes	95
6.5	Linking a Field to the List Box	97
6.6	Linking List Box to Fields	97
6.7	Linking Data Entry Fields	98
6.8	Linking a Field to Itself	99
6.9	Caveats.....	100

7 Linking Jobs..... 102

7.1	Controlling Access	102
7.2	Overview of Job Linking.....	102
7.3	Preventing Jobs from Running.....	103

8 Groups and the Group Processor..... 106

8.1	Overview	106
8.2	Group Processor Plug-In.....	106
8.3	Controlling Job Access.....	107
8.4	Group Processor Functions.....	108

9 Trilogy Client Service for Windows 114

9.1	Overview	114
9.2	Installing Trilogy Client as a Service.....	114
9.3	System Tray Icon	115
9.4	Running Jobs from the System Tray.....	116
9.5	Sending "Balloon" Notifications.....	118
9.5.1	Notifying Groups	119
9.5.2	Notifying Users	120
9.5.3	Notifying Client Machines	120
9.5.4	Automatic Notification Routing	120
9.5.5	Balloon Icon Types	121
9.5.6	Balloon Display Order	122
9.5.7	Standard Output As Balloon Message.....	122
9.6	Notify Users of Job Running	122
9.7	Client Port Number	123
9.8	Advanced Configuration – Communicating Across Subnets.....	124

10 The Scheduler..... 127

10.1	Introduction.....	127
10.2	Specifying a Scheduled Job.....	127
10.3	Related Directives.....	128
10.3.1	NotifyRunGroup	128
10.3.2	Environment	128
10.3.3	Param.....	128
10.4	Specifying Run Times.....	128
10.5	Specifying Run Days	129
10.6	Specifying Run Dates	130
10.7	Specifying Run Months	131
10.8	Setting Standard Input.....	132
10.9	Dependent Jobs.....	132
10.10	Environment Variables	134

11 Trilogy - Command Line Options 135

11.1	Trilogy Client	135
11.2	Trilogy Server	137

12	Server Side Job Control	138
12.1	Introduction.....	138
12.2	Environment	138
12.3	Trilogy Server Environment	140
12.4	How Jobs are started	140
12.4.1	Unix/Linux Servers	140
12.4.2	Windows Servers	141
12.5	Buffering	141
12.5.1	Unix/Linux.....	142
12.5.2	Windows	142
12.5.3	UseTTY Directive	142
12.6	Standard Input.....	142
12.7	Receiving Standard Input	144
12.8	Listing Running Jobs	145
12.9	Stopping Server-Side Jobs.....	146
12.9.1	Unix/Linux Servers	146
12.9.2	Windows Servers	146
12.10	Server Side Scripts – Environment Variables set by Trilogy	147
12.11	Running Jobs in Background	149
13	trilogy.conf – Reference Guide	150
14	Trilogy Scripting Engine	217
14.1	Introduction.....	217
14.2	Using the Trilogy Scripting Engine Client Side – Overview	219
14.3	Using the Trilogy Scripting Engine Server Side – Overview	219
14.4	Scripting Engine Methods	220
	Index	272

1 Introduction

1.1 What is Trilogy?

Trilogy is a client-server utility that allows predefined tasks to be run on a *Trilogy Server* either on request from a *Trilogy Client* or automatically via a built-in scheduler. Native GUI Dialogs can be presented at the client so that a client user can interact with these server-side scripts. These dialogs are defined at the server using simple text-based files and can be populated with values from scripts run on the server. Thus – once a machine has a *Trilogy Client* installed - entire GUI-based client-server applications can be easily created and made available to end-users without any further software distribution being required.

Trilogy makes it simple to create client/server applications – server code can be written in any language with which the programmer is familiar. There are no new APIs to learn, no libraries to link. Instead, *Trilogy* allows the server-side script to access the client GUI dialog via environment variables and *Trilogy* takes the server-side script's standard output and error streams and uses them to populate the client dialog. Because of this it is easy to re-use existing scripts and turn them into client/server applications with minimal effort.

Trilogy has a number of advantages over other technologies (such as web servers and browsers):

- Small footprint. Both server and client are small applications that can be launched quickly. A *Trilogy* client dialog will typically open in less than a second (depending on complexity).
- Native Look-and-Feel. *Trilogy* renders the dialog as a native GUI tailored to the platform on which it is running. Users can interact with the application in the same way as they would interact with any other native application.
- Extend Existing tools. Most IDEs, SCM tools and other applications allow local client programs to be invoked from inside their environment. *Trilogy* dialogs can then be presented as though they were a natural part of the tool, extending its capability and allowing for sophisticated integrations that feel “natural” (where launching a separate product or using a web page would not).
- Agile Development. Dialogs can be created in seconds that would take hours of development using other tools. A change can be made on the server which is then visible instantly at the client with no software distribution required.
- Multi-platform. Dialogs can be created once and will display natively on any platform on which the client can run.
- Simple Scripting. Server-side scripts can be written in whatever language is desired including (but not limited to) Shell Scripts, VBScript JScript, Python, Perl, C++ etc. These scripts interact with the client-side dialog using environment variables and by printing lines to their standard output. *Trilogy* takes care of all the complex client-server communication.
- Leverage Existing Investment. It is easy (and quick) to re-use existing scripts and turn them into client-server applications.
- Easy launch. Windows users can run (and interact with) server-side scripts by clicking on the *Trilogy* Icon in the Windows Notification Area (System Tray).

- Notifications. Server scripts (running on any platform) can send out balloon-style notifications to Windows Users. Users can quickly see when jobs are running without having to open another application window.
- Easy File Transfer. Files can be uploaded and downloaded using a secure transfer between client and server.

Trilogy clients are:

GUI Client Tool (Windows, Unix or Linux - launched by command line)
Trilogy Client Service for Windows (Docked in System Tray)
Trilogy Scripting Engine for Windows (COM Object)

1.2 About This Book

Some basic scripting knowledge is assumed. Examples throughout the book are provided in a number of languages such as Unix Shell Script and Windows VBScript. There is no “preferred” language for *Trilogy* – any server-side script (or binary executable) that can be invoked from a command line interface and that can read environment variables can interact with a client-side *Trilogy* Dialog.

1.3 Typographical Conventions

Throughout this manual, the following typographical conventions apply:

<code>Courier</code>	Text shown in this font indicates computer output or program listing.
<code>Courier Bold</code>	Text shown in this font indicates text that should be entered by the user.

1.4 Symbols Used in This Book

The following symbols are used to highlight areas of text that are of particular interest:



Text Highlighted with this symbol refers to a point raised elsewhere in the manual.



Text Highlighted with this symbol refers to important points



Text Highlighted with this symbol refers to Microsoft Windows only (either client side or server side).

2 Trilogy – Overview

2.1 Trilogy Jobs

A *Trilogy Job* runs on a *Trilogy Server*. A *Trilogy* job can be invoked either from a *Trilogy Client* or from *Trilogy's* built-in scheduler. If a *Trilogy* client is invoking a job on the *Trilogy* server it does so by specifying its *Trilogy Job Name*. This name is used by the *Trilogy* server to ascertain the actual path of the program to invoke along with any other directives or attributes that controls the server-side program execution. There are a number of advantages to this approach:

- If you wish to move the location of the script (or indeed, the script name itself) you can do so simply by telling *Trilogy* server the new location and file name – there are no amendments needed at the client.
- The "real" path is not visible at the client. This has a security benefit.
- Server jobs can run as any user without the need to provide a username/password to the client.
- Only specified scripts can be run on the remote server – if *Trilogy* doesn't know about it, a client can't run it.

The mapping of *Trilogy Jobs* to actual path and file names is done in the *Trilogy* configuration file `trilogy.conf` on the *Trilogy* server node. The format and content of this configuration file is detailed in the next section.

If the *Trilogy* configuration file specifies a dialog for the *Trilogy Job*, this dialog is presented at the client automatically. The user then interacts with this dialog and the server-side job is run only when the user clicks "OK". Any scripts invoked during this process can ascertain the contents of the *Trilogy* dialog by reading environment variables that are set by *Trilogy* before the scripts are run.



Scripts running on Windows Servers can also determine the content of the client-side dialog by using methods contained within the *Trilogy Scripting Engine*.

All communication between *Trilogy* clients and servers is encrypted. You can therefore include passwords in the interactions between client and server without it being sent across the network in free-text.

2.2 The Trilogy Configuration File

`trilogy.conf` exists on both *Trilogy* client and server nodes. It has a similar syntax on both nodes:

2.2.1 Client

```
Server=<name>
Port=<port num>

<Server Name>:
    HostName=<name>
    Port=<port num>

<Server Name>:
    HostName=<name>
    Port=<port num>
```

2.2.2 Server

```
Server=localhost
Port=<port num>

Trilogy Name:
    Directive=Value
    Directive=Value

Trilogy Name:
    Directive=Value
    Directive=Value
```

On the client, the `Server=` directive specifies the default location of the *Trilogy* server. If no other host is specified when the client is invoked, the request is made to this default server.



Trilogy server installations automatically install a client. Because of this the `trilogy.conf` file contains both `server` and `port` directives. The `server` directive is purely for the client's use and points it at the local server (`localhost`) by default.

The `Port=` directive exists on both server and client. It indicates the port number on which the *Trilogy* server is listening for incoming requests: When the *Trilogy* server starts, it reads this port number from its `trilogy.conf` file and begins to listen on that port. When a *Trilogy* client makes a request it does so to the `Server` and `Port` specified in its `trilogy.conf` file.



On the command line client, the port number and server name can be overridden with the `-b` and `-p` switches. Similarly, the *Trilogy Scripting Engine* can override the settings in the local `trilogy.conf` file by the use of the `SetPort` and `SetServerName` methods. See *Command Line Options* and *Trilogy Scripting Engine* later in this document for more information.

The `trilogy.conf` file on the *Trilogy* client allows for multiple *Trilogy Server* Instances to be defined. For example, suppose the client file is defined like this:

```
Server=Homer
Port=2301

Bart:
  Server=Burns
  Port=2302

Marge:
  Server=Maggie
  Port=2303

Lisa:
  Server=Lisa
  Port=2301
```

In this example, *Trilogy* client requests will normally be routed to the *Trilogy Server* listening on port 2301 on the Server *Homer*. However, if the user specifies a different server (for example by using the `-b` flag on the *Trilogy* command-line client) then the request is routed to the server and port associated with the specified identifier. Thus:

```
trilogy -b Marge MYJOB
```

This will send a request to the *Trilogy Server* listening on port 2303 on the server Maggie.

If the specified identifier is not found in the local `trilogy.conf`, the client will perform normal OS-based hostname resolution to determine the server location. Unless the port number has been specified otherwise, the port used is the default set at the top of the `trilogy.conf` file.

The `trilogy.conf` file on the *Trilogy* server also includes the mapping of the *Trilogy Jobs* that are specified in the calls from the clients. Each *Trilogy Job* is defined on its own line, left justified and ending with a colon (:). Beneath the *Trilogy Job* entry is a list of *Trilogy Directives* that define how *Trilogy* should respond to requests from a client to invoke the particular *Trilogy Job*. A blank line ends the entry-set for this *Trilogy Job*. This format is known as a *Stanza File*.

A stanza entry consists of one or more directives. A Directive is constructed in the format:

DirectiveName=Value

For example, consider the following entry in the server's `trilogy.conf`:

```
validate_user_name:  
    Program=/home/trilogy/programs/verify_user_name
```

This tells *Trilogy* that any request from a client to run "validate_user_name" will actually result in the execution of the server side script:

```
/home/trilogy/programs/verify_user_name
```



Any parameters passed to the *Trilogy* client are automatically passed to the server-side script. You can also use a *Trilogy* directive to set parameters which are passed to the job automatically.

The "Value" part of the directive can include environment variables. Thus, if you want to place scripts relative to the TRILOGYHOME directory, you could enter:

```
validate_user_name:  
    Program=$TRILOGYHOME/programs/verify_user_name
```

Note, though, that environment variables in `Program` directives are only read when the *Trilogy* server is started. Creating new environment variables after this point will not result in them being available to *Trilogy* until it is restarted.

2.3 Running Server Side Scripts with *Trilogy*

At its most basic, *Trilogy* can run jobs on its server based either on requests from *Trilogy* clients or when its built-in scheduler determines that a job needs to be run. For client interactions, any data appearing on the Standard Input of the client will be routed to the standard input of the server-side script. For the Command Line client, the default output behaviour is for the standard output of the server job (`Stdout`) to appear as the standard output of the client. The standard error output of the server job (`Stderr`) appears as a pop-up box on the client. These options can be overridden on the *server* with the `trilogy.conf` stanza directives `stdout=` and `stderr=` respectively, thus:

```
stdout=display|discard|popup|report|filechooser:<filename>|file:<filename>
stderr=display|discard|popup|report|filechooser:<filename>|file:<filename>
```

The options are:

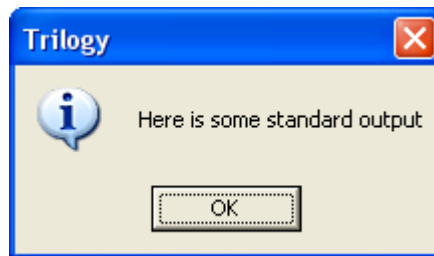
<code>display</code>	send the relevant stream from the server-side job to the relevant stream of the invoking client. This is the default for the standard output stream.
<code>discard</code>	ignore all output from the relevant stream.
<code>popup</code>	sends all output from the relevant stream to a pop-up dialog box on the client. This is the default for the standard error stream.
<code>report</code>	sends the relevant stream to a scrollable textbox on the client. This textbox is contained within a dialog that gives total run time and (when the server-side job completes) the job's exit code. A user can therefore see the output of the server-side job as it runs in "real time". Only one report box can exist for a particular job. If both <code>stdout</code> and <code>stderr</code> are set to "report", only one report dialog is opened at the client and both streams appear in it.
<code>filechooser:<filename></code>	a dialog is opened, allowing the user to select a file on the client to which the stream should be written. The default location and filename is given by the <code><filename></code> argument.
<code>file:<filename></code>	specifies a client-side file to which the stream contents should be written.

Note that the appearance of the pop-up dialog box varies slightly depending if the output is from standard error or standard output:

stderr=popup



stdout=popup



This ensures that the dialogs appear in the Windows convention, with the appropriate icon displayed either for an error (standard error) or for information (standard output)

2.4 Trilogy Client Service



This section only applies to Windows Clients.

Windows clients can install *Trilogy Client* as a Service. In this case a *Trilogy* Icon is visible in the Desktop *Notification Area*.

Here is an example of the *Trilogy Client* running as a Service on a client PC and present in the Notification Area:



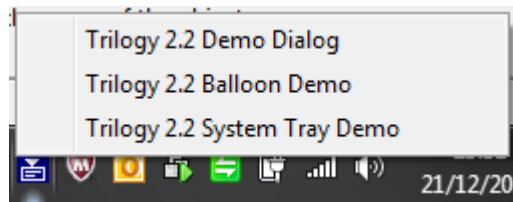
When *Trilogy Client* is running as a Service, the user can interact with the Notification Area Icon. The user can:

- Right-Click to invoke jobs on one or more *Trilogy* Server(s)

- Receive “Balloon” style notifications from jobs running on *Trilogy* Servers.
- Receive a visual indication when specific *Trilogy* jobs are running on the *Trilogy* Server.

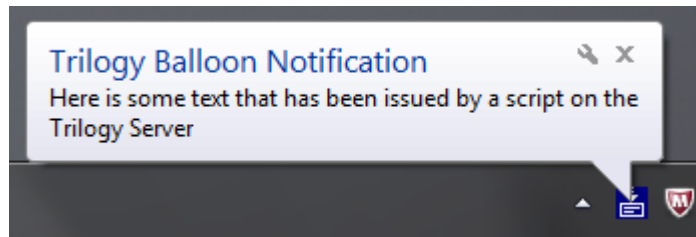
The jobs a user can select from the Notification Area Icon can be determined by the user groups of which the user is a member.

Here is an example of a user right-clicking on the *Trilogy* Icon within the Notification Area:



Jobs that are to be shown within the Notification Area are identified by directives within the server-side `trilogy.conf` configuration file.

Similarly, “Balloon” style notifications can be sent to individual users, all users or users within certain user groups. This can be done from either Unix or Windows Servers:



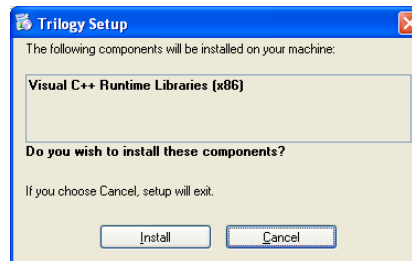
User groups for access control and for balloon style notifications are determined via a “plug-in” mechanism at the *Trilogy* Server. *Trilogy* Server ships with plug-ins for file-based user groups. Other plug-ins are available for download from the Trinem website (for example, a plug-in is available for CA Technologies Software Change Manager).

3 Installation Guide

3.1 Installing a Trilogy Server

3.1.1 Windows

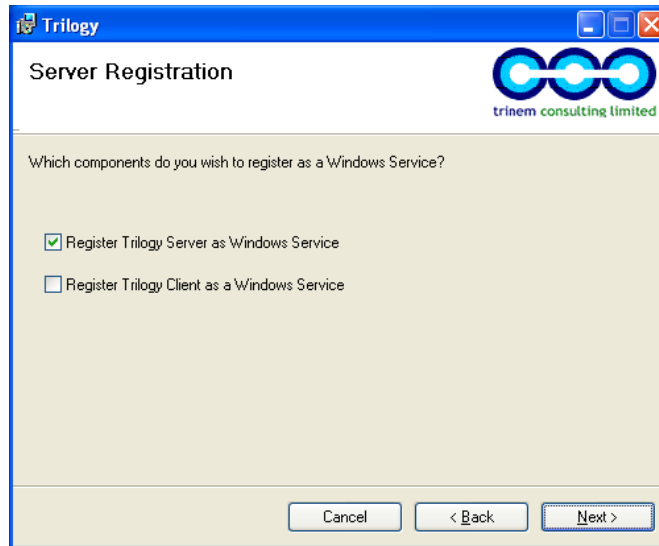
- Insert the installation CD into the CD Drive.
- The installation wizard may start automatically. If it does not...
 - Open Windows Explorer
 - Navigate to the CD Drive containing the *Trilogy* installation media
 - Double-click the **setup.exe** icon.
- If your machine does not have the Visual C++ run-time libraries installed, you will be presented with the following dialog:



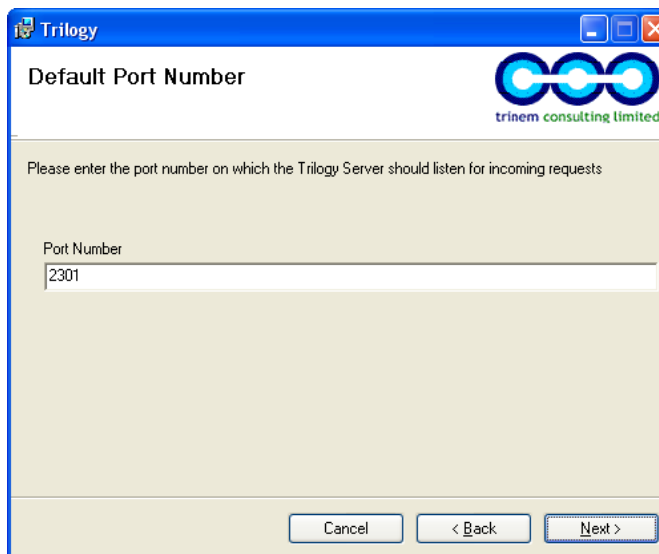
- Should this dialog be presented then you will need to click "Install" in order to install the pre-requisite C++ libraries.
- Once the pre-requisite C++ runtime libraries have been installed (or if they are already installed) you will see a welcome page as follows:



- Click "Next"
- Read and confirm acceptance of the license conditions by clicking "I Agree" and clicking "Next". The following dialog will be displayed:

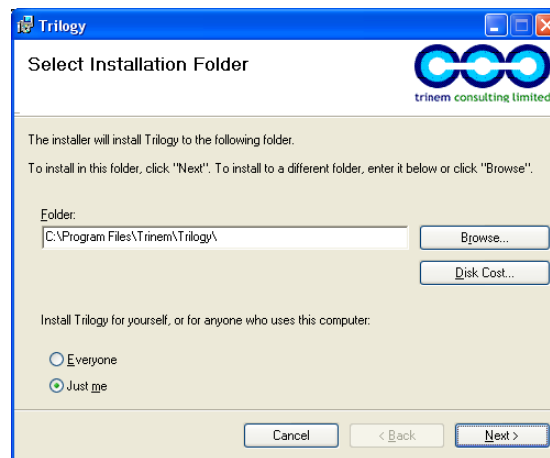


- Installing a *Trilogy* Server automatically installs a *Trilogy* Client. On this screen, you can select whether to register the *Trilogy* Client and Server as windows services. This has the advantage of starting the *Trilogy* Client and Server automatically whenever the server is rebooted. The *Trilogy Client Service* is used to place a *Trilogy* Icon in the System Tray (Notification Area) on the desktop for selecting and running remote jobs and for receiving "Balloon" style notifications from the *Trilogy* Server. The option to register the *Trilogy* Server is selected by default; that for the client is not. If you do not wish the server to be registered as a service then deselect this option before continuing. If you wish the *Trilogy* Client to be started as a service, then select the "Register Trilogy Client as a Windows Service" before continuing.
- Click "Next". The following dialog is displayed:

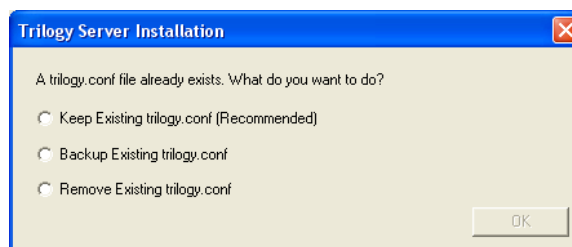


- This is where you set the port number on which the *Trilogy* Server will listen for incoming requests. It defaults to 2301. If you want to change this default, do this here.

- Click "Next". The following dialog is displayed:



- Confirm the install folder (defaults to `c:\Program Files\Trinem\Trilogy` on 32-bit Windows machines or `C:\Program Files (x86)\Trinem\Trilogy` on 64-bit Windows machines). Change this if required.
- Click "Next". You are presented with a dialog that indicates that the software installation is about to begin. This is the last point in which you can click "back" to change the installation folder. If you wish to proceed, click "Next" and the software installation will begin.
- If you are upgrading from a previous version of *Trilogy*, then you will be notified that a `trilogy.conf` file already exists. A dialog box will be presented with three options:



- Select one of the options and click OK to continue.
 - Keep Existing `trilogy.conf` (Recommended) retains the original `trilogy.conf`. All server-side job definitions are retained. This is the recommended approach. However, if you are upgrading from a Trilogy 1.x or 2.1 installation you will not have the new DEMO jobs created which demonstrates the new features.
 - Backup Existing `trilogy.conf`. The existing configuration file is renamed to `trilogy.conf.orig` and a new `trilogy.conf` is created, containing the port number specified in the installation dialog and the new DEMO jobs.
 - Remove Existing `trilogy.conf`. The `trilogy.conf` is simply replaced with the new version.
- Installation of the Windows server is now complete. A temporary license key (`trilogy.lic`) is created automatically in the installation directory (the same directory in which `trilogy.conf` is located). This temporary license key is for

30 days and supports up to 2 remote clients. This will be sufficient to evaluate the *Trilogy* software. If you wish to run *Trilogy* permanently you will need to copy a new license file to the installation directory. See "Licensing the Trilogy Server" below for more information.

- Note, installing a server automatically installs a client. You cannot install a server without a client.

3.1.2 Unix

- Login to the Unix server as root. If you do not have root access then see below.
- Insert the installation CD into the CD Drive.
- Mount the CD if necessary using the appropriate "mount" command. Some systems may auto-mount the CD.
- Create a directory where you want *Trilogy* to be installed. The owner of the directory is not important but the user who starts the *Trilogy* daemon process must have at least read access to the directory.
- Copy the appropriate "tar" file to this directory. The tar file is located on the CD media under:

```
UNIX/<platform>/TrilogyServer_<platform>.tar.gz
```

- Navigate to the *Trilogy* directory. Uncompress the tar file by entering:

```
gunzip TrilogyServer_<platform>.tar.gz
```

- Extract the contents of the tar file with:

```
tar xvf TrilogyServer_<platform>.tar
```

- An "install" directory will have been created. Navigate to this directory:

```
cd install
```

- Run the install script located in this directory:

```
./install.sh
```

- The license is displayed. Use the space bar to page through the license agreement. At the end of the agreement, you will be asked if you accept the terms of the license. If you do enter Y and press enter.
- You will be asked to enter a port number on which the Trilogy Server will listen for incoming requests. Hit ENTER to accept the default (2301) or enter a new number here.
- If this is an upgrade from a previous release of *Trilogy*, then you will be prompted that a `trilogy.conf` file already exists. You then have options to either keep this file (recommended), backup the existing file or to simply overwrite the file with a new version.
 - Keep Existing `trilogy.conf` (Recommended) retains the original `trilogy.conf`. All server-side job definitions are retained. This is the

recommended approach. However, if you are upgrading from a Trilogy 1.x installation you will not have the new DEMO job created which demonstrates the new linked and disabled fields feature.

- Backup Existing `trilogy.conf`. The existing configuration file is renamed to `trilogy.conf.orig` and a new `trilogy.conf` is created, containing the port number specified in the installation dialog and the new DEMO job.
 - Remove Existing `trilogy.conf`. The `trilogy.conf` is simply replaced with the new version.
- Trilogy Server installation is now complete.
 - You should then copy your license file to the installation directory. This is the same directory in which the `trilogy.conf` file is located. You can download a demo license for evaluation from <http://www.trinem.com>. See "Licensing the Trilogy Server" below for more information.
 - Set the PATH to include `$TRILOGYHOME/bin`.

Server Installation for non-root users:

If you do not have root access to the server machine, then you will not be able to mount the CD media. You will have to extract the `TrilogyServer_<plat>.tar` file for your particular platform and copy it to an appropriate area on the Server. Then unzip and extract the contents of the tar file and run `install.sh` as above.

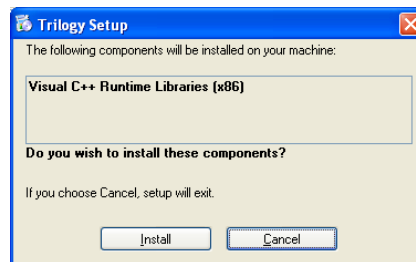
3.2 Installing a Trilogy Client

3.2.1 Windows



Note, if you have previously installed a *Trilogy* Server on this node, you will have a client installed automatically. These instructions should be followed to install a client on remote nodes only.

- Insert the installation CD into the CD Drive.
- The installation wizard may start automatically. If it does not...
 - Open Windows Explorer
 - Navigate to the CD Drive containing the *Trilogy* installation media
 - Double-click the **setup.exe** icon.
- If your machine does not have the Visual C++ run-time libraries installed, you will be presented with the following dialog:



- If this dialog is presented then click "Install" in order to install the pre-requisite C++ libraries.
- Once the pre-requisite C++ runtime libraries have been installed (or if they are already installed) you will see a welcome page as follows:



- Click "Next"
- Read and confirm acceptance of the license conditions by clicking "I Agree" and clicking "Next". The following dialog will be displayed:

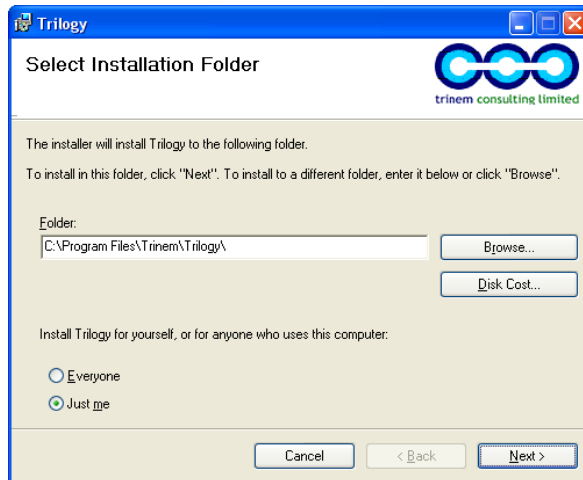
- Enter the name of the node where the *Trilogy* Server is running and change the default port number on which the server is listening (if required).

The "Server Name" entered during the install sets the default server name and port number in the client side *trilogy.conf* file. On the command line client, the port number and server name can be overridden with the `-b` and `-p` switches.



Similarly, the *Trilogy Scripting Engine* can override the settings in the local `trilogy.conf` file by the use of the `SetPort` and `SetServerName` methods. See *Command Line Options* and *Trilogy Scripting Engine* later in this document for more information.

- Click "Next". The following dialog is displayed:



- Confirm the installation folder (defaults to `c:\Program Files\Trinem\Trilogy`). Change this if required.
- Click "Next". You are presented with a dialog that indicates that the software installation is about to begin. This is the last point in which you can click "back" to change the installation folder. If you wish to proceed, click "Next" and the software installation will begin.
- Installation of the Windows client is now complete.

3.2.2 Unix



Note, if you have previously installed a *Trilogy* Server on this node, you will have a client installed automatically. These instructions should be followed to install a client on remote nodes only.

- Login to the Unix server as root. If you do not have root access then see below.
- Insert the installation CD into the CD Drive.
- Mount the CD using the appropriate "mount" command.
- Create a directory where you want the *Trilogy* client to be installed.
- Copy the appropriate "tar" file to this directory. The tar file is located on the CD media under:

```
UNIX/<platform>/TrilogyClient_<platform>.tar.gz
```

- Navigate to the *Trilogy* directory. Uncompress the tar file by entering:

```
gunzip TrilogyClient_<platform>.tar.gz
```

- Extract the contents of the tar file with:

```
tar xvf TrilogyClient_<platform>.tar
```


- An “install” directory will have been created. Navigate to this directory:

```
cd install
```

- Run the install script located in this directory:

```
./install.sh
```

- The license is displayed. Use the space bar to page through the license agreement. At the end of the agreement, you will be asked if you accept the terms of the license. If you do enter Y and press enter.
- You will then be prompted to enter the default location of the *Trilogy Server*. This is the machine name on which the *Trilogy Server* is running. The client machine must be able to resolve this name to an IP address.
- You will then be prompted to enter the port number on which the Trilogy Server is listening for incoming connections. This defaults to 2301. If you want to accept this default, then just hit ENTER. Otherwise, enter your desired port number and hit enter.



The “Server Name” and “port number” entered during the Unix Client Install sets the default server name and port number in the client side *trilogy.conf* file. On the command line client, the port number and server name can be overridden with the `-b` and `-p` switches.

- Installation of the client is now complete. Note that the environment variable TRILOGYHOME needs to be defined for any user wishing to invoke the Unix *Trilogy* client. You can do this either in the central profile (`/etc/profile` or `/etc/.login`) or in any “wrapper” scripts you are using to invoke *Trilogy*. TRILOGYHOME needs to point to the directory containing the `trilogy.conf` file.

Client Installation for non-root users:

If you do not have root access to the client machine, then you will not be able to mount the CD media. You will have to extract the `TrilogyClient_<platform>.tar` file for your particular platform and copy it to an appropriate area on the Server. Then unzip and extract the contents of the tar file and run `install.sh` as above.

3.3 Licensing the Trilogy Server



When Trilogy Server is first installed it automatically creates a 30-day, two user license. Follow the steps below to replace this temporary key with a permanent key specific to your organisation.

Licensing covers the expiry date of the server, the number of client nodes that are allowed to connect to it and (optionally) the hostname of the server node. This is done by entries in a license file. This file is called `trilogy.lic` and is located in the `$TRILOGYHOME` directory on the server node.

Trilogy clients are *not* licensed. They check for their continued validity against the *Trilogy* Server specified in the `trilogy.conf` file. In practice, this means that a new license can be applied system-wide by editing the appropriate `trilogy.lic` on the server node only.

Here is an example `trilogy.lic`:

```
LICENSE KEY:  MOGQ-BNSO-VWIA-YYPG-BWVI
CUSTOMER NAME: demo
EXPIRY DATE:  31/08/08
NODES:        10
```



The expiration date is specified in European Date Format (DD/MM/YYYY)

The customer name, expiry date and permitted number of clients (NODES) are encrypted in the license key. Therefore, changes to any of these fields require a new license key to be generated. Trinem will supply an appropriate `trilogy.lic`.

Do not attempt to edit the file. If the license key is not recoded to match the other fields, the license file will be considered invalid and the *Trilogy* Server will not start.

The NODES field specifies how many unique client nodes are allowed to connect to the server. When a *Trilogy* Client connects to a *Trilogy* Server, its hostname is recorded by the server. This hostname is regarded as a unique client identifier. Once the number of unique hostnames has been recorded, subsequent connection requests from *new* clients (whose hostnames have not previously connected) are refused.

This technique means that multiple connections are permitted from a single client with the loss of only one end-user license.



If the license is node-locked (i.e.: it will only run on a particular server) then the license file contains an additional `HOSTNAME:` line and the license key itself is longer.

3.4 Starting the Trilogy Server

3.4.1 Windows

- Ensure that the server has been licensed with the appropriate entries in the %TRILOGYHOME%/trilogy.lic file.
- The windows server can be started either from the command line (DOS Prompt) or by registering it as a Windows Service. The advantage of registering the server as a service is that it can be automatically started when the box is booted and it does not occupy a DOS box.
- To register the Windows *Trilogy* Server as a service:
- Open a DOS Prompt
- Navigate to the folder where you installed *Trilogy* (%TRILOGYHOME%).
- Enter **trilogyserver -install**
- Open Control Panel
- Depending on platform, either double-click on "Services" or double-click on "Administrative Tools" and then double-click on "Services".
- Locate the "Trilogy Server" entry. Click on the start button.
- The *Trilogy* Server process is now running.



On XP platforms and later you can choose to have the *Trilogy* Server run as a different user. Right Click on the "Trilogy Server" entry, select "Properties" and then click on the "Log On" tab. Remember that all jobs run by *Trilogy* inherit the permissions of the user who started the *Trilogy* Server. It may therefore be advantageous to select a non-system user to start the *Trilogy* Server.

- To remove (deregister) the Windows *Trilogy* Server as a service:
- Open a DOS Prompt
- Navigate to the folder where you installed *Trilogy* (%TRILOGYHOME%).
- Enter **trilogyserver -uninstall**
- Open Control Panel
- Depending on platform, either double-click on "Services" or double-click on "Administrative Tools" and then double-click on "Services".
- Verify that the "Trilogy Server" entry has been removed.
- To start the Windows *Trilogy* Server from a DOS Prompt
- Open a DOS Prompt
- Navigate to the folder where you installed *Trilogy* (%TRILOGYHOME%).
- Enter **trilogyserver**
- You will receive a message indicating that the process has started.

3.4.2 Unix

- Ensure that the server has been licensed with the appropriate entries in the `$TRILOGYHOME/trilogy.lic` file.
- Login as the user who you want *Trilogy* jobs to run as.
- Navigate to the directory where you installed the *Trilogy* server (`$TRILOGYHOME`).
- Enter:

`bin/trilogyd`
- You will receive a message that the server has started and is listening for incoming connections.



Note, the server automatically puts itself in background (daemon process).

3.5 Testing the installation

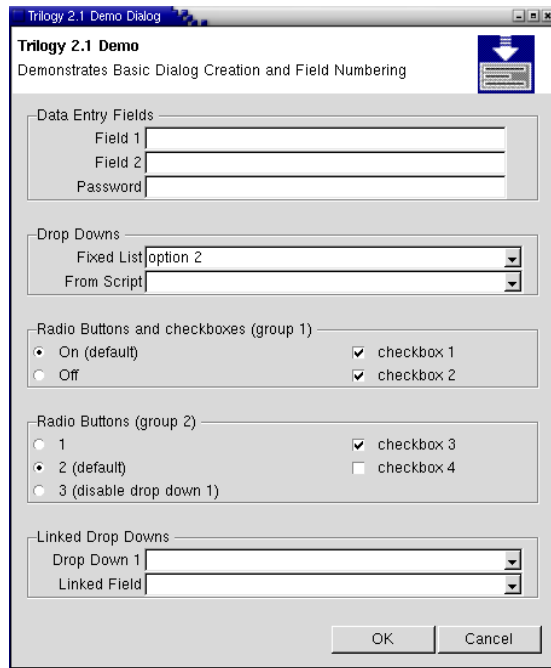


By default, the *Trilogy* Server listens for incoming connections on port 2301. This can be changed during the installation process. If you have a firewall between the client and the server then it must be configured to allow TCP traffic to port 2301 (or whichever port you configured during the install).

Ensure the *Trilogy* server is running

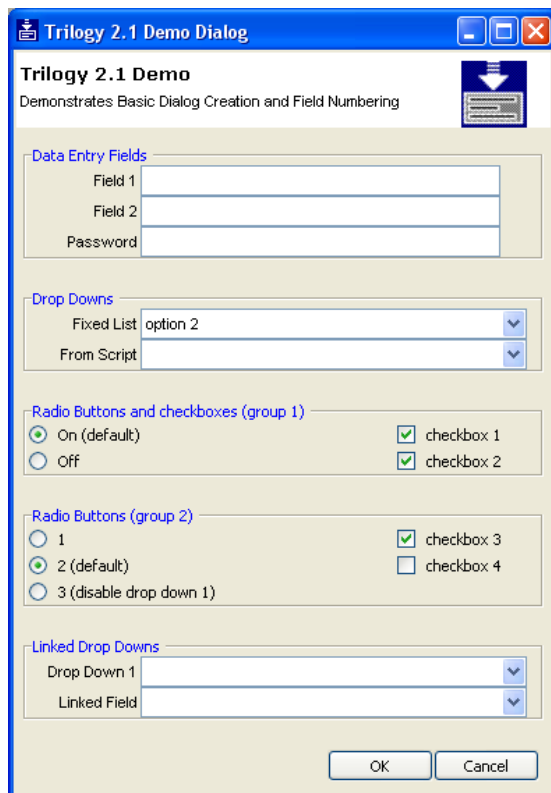
3.5.1 UNIX client

- Ensure that the `DISPLAY` environment variable is set to the IP address of your X-Server or `0:0` if running locally.
- Navigate to `$TRILOGYHOME`
- Enter `bin/trilogy demo`
- A dialog will be displayed:



3.5.2 Windows client

- Open a DOS command box
- Navigate to %TRILOGYHOME%
- Enter **trilogy demo**
- A dialog will be displayed
- The dialog will be similar to this:



This demo dialog illustrates the basic principles of *Trilogy*. All dialog definitions and scripts are held on the server whilst the client displays the dialog at run time and interacts with the server.

The scripts and screen definitions for this demo dialog are located in `$TRILOGYHOME/demo` on Unix servers and `%TRILOGYHOME%\demo` on Windows servers.



More examples can be found in Appendix A at the end of this document.

3.6 Stopping the Trilogy Server

3.6.1 Windows

- If the *Trilogy* Server was installed as a Windows Service:
 - Log in as "Administrator" or as a user with full admin privileges.
 - Open Control Panel
 - Depending on platform, either double-click on "Services" or double-click on "Administrative Tools" and then double-click on "Services".
 - Select the Trilogy Server process and select "stop".
- If the *Trilogy* Server is running from a DOS prompt:
 - Simply hit CTRL-C to stop the process.

3.6.2 Unix

- Login as the user who started the `trilogyd` daemon process.
- Navigate to the directory where you installed *Trilogy* (`$TRILOGYHOME`)
- Enter `bin/trilogyd -shutdown`
- You will receive a message that the server has been shutdown.

4 Trilogy Dialogs

4.1 Creating Dialogs

The *Trilogy* command line client can display data entry dialogs if these have been defined on the server. The advantage of defining such dialogs at the server cannot be overstressed – dialogs can be created once and are then available instantly at any *Trilogy* client (both on Windows and Unix platforms). Should a dialog require modification then it can be done once and all *Trilogy* clients will inherit the modified dialog.

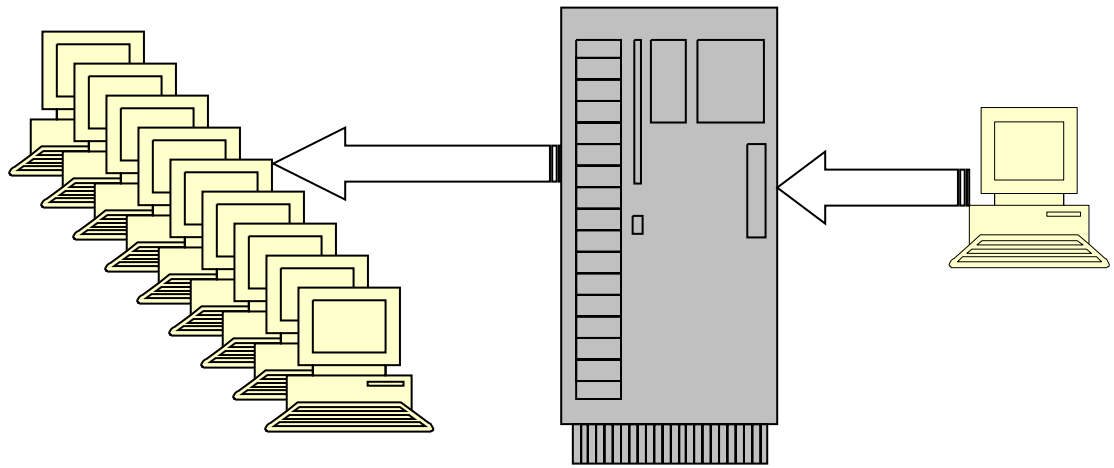


Figure 4.1: By creating Dialog Definitions on the Server, they are available instantly to your entire installed user base with no software distribution required.

In one sense, *Trilogy* Dialogs are similar to a web interface in that the client renders a display that is defined entirely on the server. Unlike web interfaces, however, *Trilogy* Dialogs look like a native Windows (or Unix) application interface. They display quickly, have a familiar “application style” look and feel and can be invoked from inside other client tools and IDEs whilst maintaining the feel that the dialog is a seamless extension of that tool. Unlike HTML, creating a *Trilogy* Dialog can take seconds rather than hours.

Data entry fields are represented by square brackets like this:

Drop-Down lists are represented by braces like this:
Radio buttons are represented by a letter "oh" on its

own, like this:
 Checkboxes are represented by a letter x on its own,
 like this:

Named Frames are represented by a single dash followed by the name of the frame, like this:

Tabbed Pages are represented by a > char, followed by the name of the tab like this:

$$[\quad]$$
$$\{ \quad \quad \quad \}$$

○

X

- named frame

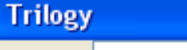
```
> tab name
```

Here is an example dialog file, entered in
\$TRILOGYHOME/SCREENS/credentials.scn:

```
Username [
Password [
```

```
GET_CREDENTIALS:
  Dialog=$TRILOGYHOME/SCREENS/credentials.scn
  Program=$TRILOGYHOME/Scripts/ProcessCredentials
```

```
trilogy get credentials "Trinem Software"
```



Trilogy 2.3 Administrator Guide
www.trinem.com

script. In other words, `$TRILOGYHOME/Scripts/ProcessCredentials` is invoked with parameter 1 set to "Trinem Software".



In order that the invoked script can read the values from the dialog, *Trilogy* places the values from the client GUI into environment variables `TRIFIELD n` , where n represents the number of the field on the screen. Fields are numbered from left to right and from top to bottom. Therefore, in this case, `TRIFIELD1` represents the *Username* field and `TRIFIELD2` represents the *Password* field. When the script runs on the *Trilogy* server, it can access the data entered on the dialog by accessing the environment variables `TRIFIELD1` and `TRIFIELD2`.



Scripts running on Windows Servers can also determine the content of the client-side dialog by using the `GetField()` method contained within the *Trilogy Scripting Engine*. In addition, scripts written in either JScript or VBScript (determined by the file extension `.js` or `.vbs`) will be automatically invoked with the appropriate cscript interpreter.

Here is a more complex example, which shows the field ordering and how they map to the corresponding environment variables:

The image shows a screenshot of a Windows-style dialog box titled "Trilogy". The dialog is divided into three sections: "First Section", "Second Section", and "Third Section".

- First Section:** Contains three text input fields labeled "Server", "User Name", and "Password".
 - \$TRIFIELD1 points to the "Server" field.
 - \$TRIFIELD2 points to the "User Name" field.
 - \$TRIFIELD3 points to the "Password" field.
- Second Section:** Contains four input fields: "Area From", "Sub Area", "Area To", and "Copy Options".
 - \$TRIFIELD4 points to the "Area From" field.
 - \$TRIFIELD6 points to the "Sub Area" field.
 - \$TRIFIELD5 points to the "Area To" field.
 - \$TRIFIELD7 points to the "Copy Options" field.
- Third Section:** Contains two text input fields labeled "Source Project" and "Dest Project".
 - \$TRIFIELD8 points to the "Source Project" field.
 - \$TRIFIELD9 points to the "Dest Project" field.

At the bottom of the dialog are "OK" and "Cancel" buttons.



This example contains frames identified as "First Section", "Second Section" and "Third Section". Creating these *Named Frames* is covered later in this document.

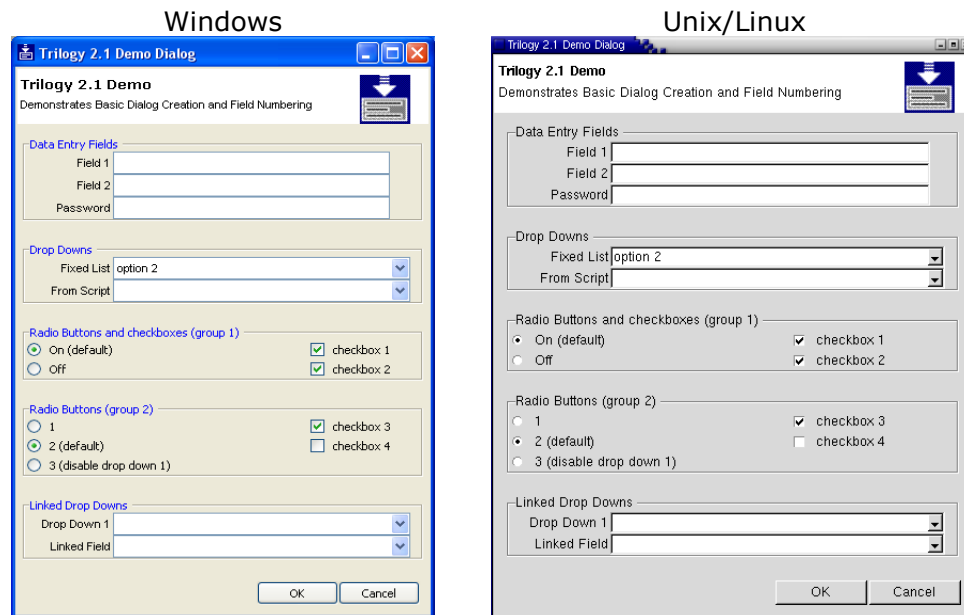
By creating a standalone client application that invokes the *Trilogy* command line client in this way, a systems administrator can build sophisticated server-side scripts that can interact with the client user.

Presenting dialogs in this way allows multiple parameters to be passed to the server-side process without the need to concern the end-user with either parameter order or quoting syntax. In the next few sections, we will cover how such *Trilogy* dialogs can be enhanced and validated.

4.2 Differences between Unix and Windows *Trilogy* Dialogs

As already discussed, *Trilogy* dialogs can be defined once on a *Trilogy* server and are then available instantly at any *Trilogy* client, regardless of its operating platform (Windows, Unix or Linux). It is important to note that there is no difference in the way the dialog is defined – *Trilogy* takes care of organising the display on the relevant platform.

However, there are minor differences in the way the various GUI components are rendered. For example, here is the same demo dialog shown rendered on both Windows and Unix/Linux clients:



Note, these dialogs show named frames, checkboxes, radio buttons and drop-down lists. The creation of these GUI components will all be covered later in this section..

As can be seen, each GUI is rendered in the appropriate way for the client platform.

The remainder of this document will mostly show only Windows look-and-feel. Be aware that if you are working on a Unix platform the rendering of some components will be different.



The remaining sections of this chapter show how *Trilogy* dialogs can be validated and enhanced. Bear in mind that all the scripts described herein to validate or populate dialogs run server-side. This means that a Systems Administrator can set up and define complex server-side scripts without any software distribution to the clients.

4.3 Creating Password Fields

If you do not wish the contents of a field to be visible when data is entered, place one or more asterisk (*) characters anywhere between the opening and closing square brackets that define the field.

Here is an example:

```
User Name [ ]
Password  [* ]
```

Will be displayed by *Trilogy* like this:

When text is entered into the fields, the User Name field will be visible but the Password field will be "starred out":



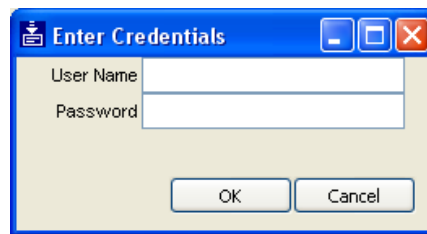
Although the display will not show the entered text, the server-side script can still read what was actually entered into the field by accessing the appropriate TRIFIELD environment variable (or, on Windows Servers, using the `GetField` method) in the usual way.

4.4 Adding a Title to the Dialog

In the example above, the title of the dialog was "Trilogy". This is the default *Trilogy* dialog name. To override this title with your own, include a `Title=` directive in the server-side `trilogy.conf`:

```
GET_CREDENTIALS:  
  Title=Enter Credentials  
  Dialog=$TRILOGYHOME/SCREENS/credentials.scn  
  Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
```

Now, when the dialog is displayed at the client, the title bar shows the Title specified in the `Title=` directive:



4.5 Creating Drop Down Lists

A *drop down list* is a field where the user can select a single item from a list of possible options. This is useful for circumstances where you wish to restrict a field's input or to assist the user in some way.

Trilogy supports the creation of drop down lists by the use of the `PopulateFieldnWith=` directive in the server's `trilogy.conf`. The *n* represents the field number as described earlier (fields are numbered from left to right and top to bottom, starting at 1). The *value* for this directive can either be a fixed list of hard-coded values or it can be the name of a script. If you specify a script then the output from the script is used to populate the drop-down list.



It is important to realise that the definition of the screen layout (defined with the `Dialog=` directive) remains the same regardless of whether the data entry screen is a simple data-entry field or a drop down list. A data entry field will automatically be turned into a drop-down list if more than one value is placed in it by the `PopulateFieldnWith=` directive. If you wish to force *Trilogy* to display a drop-down list rather than a data-entry field, use braces rather than square brackets for the field definition.

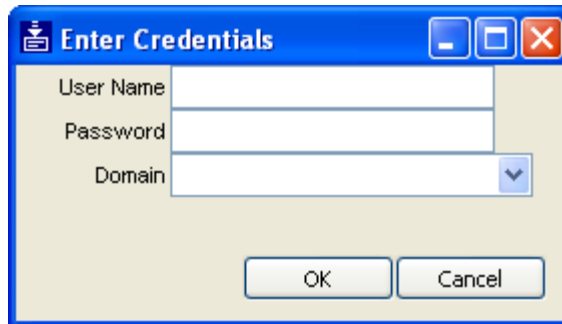
For example, if we wished to add a drop-down (such as a domain name) to our "get credentials" screen above, we could modify the server-side screen definition to look like this:


```
User Name  [ ]
Password  [*]
Domain    { }
```

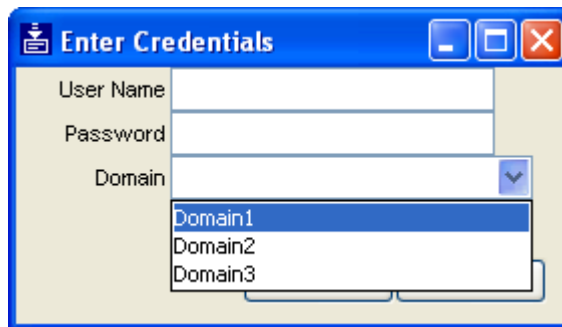
...and amend the server-side `trilogy.conf` job entry to populate this 3rd field with a fixed list of values:

```
GET_CREDENTIALS:
  Title=Enter Credentials
  Dialog=$TRILOGYHOME/SCREENS/credentials.scn
  Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
  PopulateField3With={Domain1,Domain2,Domain3}
```

Now, when the *Trilogy* client is invoked, the third field becomes a drop down:



Drop down lists are identified by the presence of a selection button  next to the field. When this button is pressed, the drop down list is presented:



The user can then only select from one of the three, hard-coded options. When the dialog is submitted (by pressing "OK") the selected option will be passed in the environment variable `TRIFIELD3` just as a normal data-entry field would.

Although powerful, such hard-coded lists are a little restrictive. You may wish to generate the list with an external program - perhaps to read the values from an external file or database or to vary the list content dependent on some server-side logic

To support this, *Trilogy* allows you to specify the name of a script in the `PopulateFieldnWith=` directive. When the dialog is invoked, *Trilogy* will run this script, *passing it any command line values that were included on the command line on the client*. The standard output from this script is then used to populate the drop down list – the first line of output becomes the first selectable item, the second line becomes the second selectable item and so on.



Note, this method of using scripts to populate fields is an important concept in *Trilogy*. Remember, the first *line* of the invoked script's standard output is the first selectable item, the second *line*, the second item and so on.

Since the parameters that were passed to the *Trilogy* client are passed to this server-side script, the script can vary its output based on the values passed. For example, by invoking the client (on a windows machine) thus:


```
trilogy get_credentials %PROCESSOR_ARCHITECTURE%
```

The architecture of the client's machine is passed as parameter 1 to the `PopulateFieldnWith=` script. The script can then generate its output based on this architecture and the drop-down list can be varied.



If the server-side script exits with a non-zero exit code, then the field on the client dialog is disabled. You can use this to prevent users from entering data into fields. This technique is discussed in more detail later in this document.

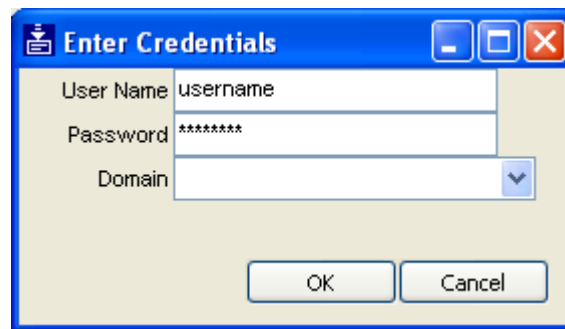
4.6 Pre-Populating Dialogs

You may find it useful to pre-populate fields when the dialog is first displayed. You can use this to fill-in certain values based on the invoking context.

In order to support this, *Trilogy* employs the `PopulateWith=` directive. This works almost identically to the `PopulateFieldnWith=` directive. You can specify either a fixed list of discrete values (with the syntax `{field1,field2,field3,...fieldn}`) or you can specify a script. In the first case, the first specified fixed value is placed in the first field, the second in the second and so on. Here is an example:

```
GET_CREDENTIALS:
  Title=Enter Credentials
  Dialog=$TRILOGYHOME/SCREENS/credentials.scn
  Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
  PopulateField3With={Domain1,Domain2,Domain3}
  PopulateWith={username,password}
```

Displays:



In this case, "username" has been placed in field 1 and "password" has been passed in field 2. Since field 2 is set as a password field (by including an asterisk character between the opening and closing braces in `credentials.scn`) the word "password" is hidden.

If you specify a script, *Trilogy* will invoke the script (passing any command line parameters as previously discussed) and will then read the standard output from the script – the first *line* becoming the first field, the second *line* becoming the second field and so on.

For example, suppose we have a script that simply outputs the parameters passed to it, like this:

```
#!/bin/ksh
echo $1
echo $2
```

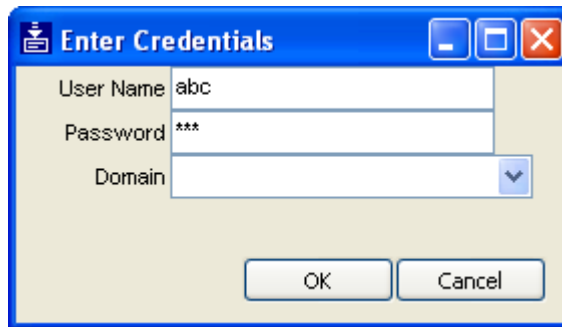
If we place this script in `$TRILOGYHOME/SCRIPTS/prepop`, and add it to the server-side `trilogy.conf` like this:

```
GET_CREDENTIALS:  
Title=Enter Credentials  
Dialog=$TRILOGYHOME/SCREENS/credentials.scn  
Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials  
PopulateField3With={Domain1,Domain2,Domain3}  
PopulateWith=$TRILOGYHOME/SCRIPTS/prepop
```

Then invoke the client-side UDP like this:

```
trilogy get_credentials abc def
```

Then the dialog displayed will look like this:



This is because the first field is populated with the first line of standard output from the script `$TRILOGYHOME/UDPS/prepop (echo $1)`. The second field is populated with the second line of standard output from the script (`echo $2`). The script has been passed the same command-line parameters as were passed in the call to the *Trilogy* client. Note, there is no third line output by the script so the Domain drop-down (`TRIFIELD3`) is not populated. Pre-populating drop-down lists is covered in the next section.

4.7 Pre-Populating Dialogs Containing Drop Down Lists

If you want to pre-populate a dialog containing drop down lists then you can simply use a combination of `PopulateWith=` and `PopulateFieldnWith=` directives. In this case, *Trilogy* will invoke all the `PopulateFieldnWith=` directives to create the drop down lists and will then invoke the `PopulateWith=` directive to set the initial values for the entry fields. When the drop down list is pre-populated in this way, *Trilogy* will set the initial value of the list to the appropriate value. Note that this will only work provided that the `PopulateWith=` directive results in the field being set to a value included in the output of the `PopulateFieldnWith=` directive. If not, the list is left with no initial value.

As an example, let's extend the "prepop" script described in the previous section to output a third parameter. This will be used to populate `TRIFIELD3`. On our dialog, this is a drop-down list containing the domain names.

The job definition in the server-side `trilogy.conf` file stays the same:

```
GET_CREDENTIALS:
    Title=Enter Credentials
    Dialog=$TRILOGYHOME/SCREENS/credentials.scn
    Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
    PopulateField3With={Domain1,Domain2,Domain3}
    PopulateWith=$TRILOGYHOME/SCRIPTS/prepop
```

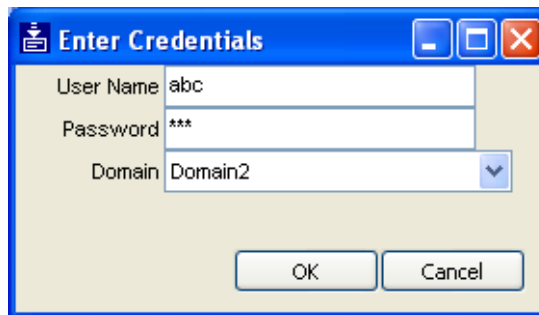
But the "prepop" script is modified as shown:

```
#!/bin/ksh
echo $1
echo $2
echo $3
```

If *Trilogy* client is invoked like this:

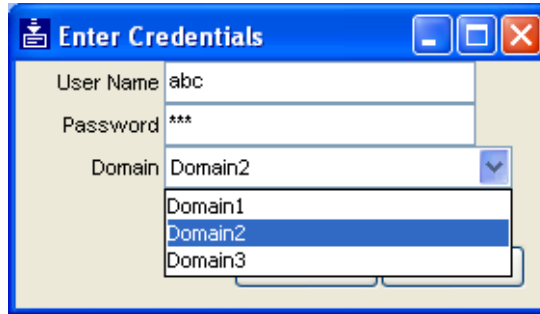
```
trilogy get_credentials abc def Domain2
```

Then the dialog is shown like this:



As can be seen, field 3 is still a drop down list but it has been preset to the value placed in field 3 by the `PopulateWith=` script. If the drop-down list button is

selected, the list can be seen just like before but with the pre-populated value already selected:

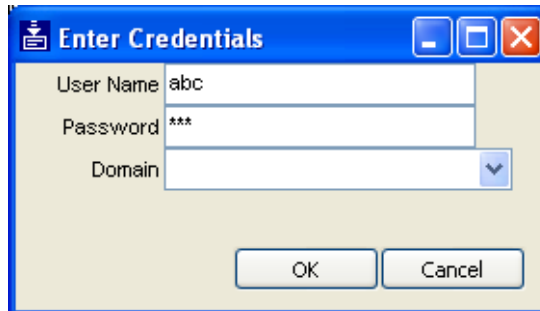


If, on the other hand, the `PopulateWith=` directive results in a value being set for the field which does not occur in the list, then no value is preselected and the field is left blank.

To illustrate this, assume the client is invoked as follows:

```
trilogy add_user_to_project abc def NOSUCHDOMAIN
```

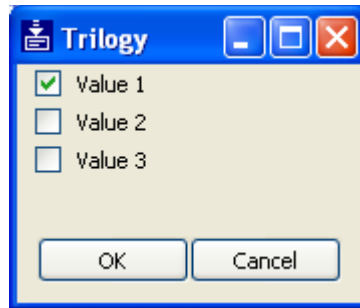
In this case, *Trilogy* will invoke the script `$TRILOGYHOME/UDPS/prepop` which will simply echo the passed parameters on separate lines: the first parameter on line 1 (which *Trilogy* will place in the first field), the 2nd parameter on line 2 (which *Trilogy* will place in the second field) and "NOSUCHDOMAIN" on line 3 (which *Trilogy* will place in the 3rd field). Because the third field is a drop down list made up of the three values "Domain1", "Domain2" and "Domain3" and since "NOSUCHDOMAIN" is not one of these values, then *Trilogy* will leave the list with no initial selection:



4.8 Checkboxes and Radio Buttons

Trilogy also supports the creation of checkboxes and radio buttons.

A checkbox is an on-off toggle. It is typically displayed like this:



Clicking inside the checkbox changes its state from selected to unselected and vice-versa.

Radio Buttons, on the other hand, are usually arranged in groups. Only one of a group of radio buttons can be selected at any one time. Selecting a new radio button automatically clears the state of all the other radio buttons in the group.



Here, selecting the "Value 2" radio button automatically clears the "Value 1" and "Value 3" buttons.

4.8.1 Creating Checkboxes and Radio Buttons

You identify checkboxes inside the *Trilogy* dialog definition file by including a single "x" character where you want the checkbox to be placed. You can use either a lower-case "x" or an upper-case "X". In the latter case, the checkbox is displayed SELECTED by default (unless the dialog has been pre-populated with a `PopulateWith=` script –see below).

Conversely, you identify radio buttons inside the *Trilogy* dialog definition file by including a single "o" (letter oh) character where you want the radio button to be placed. Just like checkboxes, you can use an upper or lower case "o". If you use an upper-case "O" then the radio button will be selected by default unless this behaviour is altered by the use of a `PopulateWith=` script.

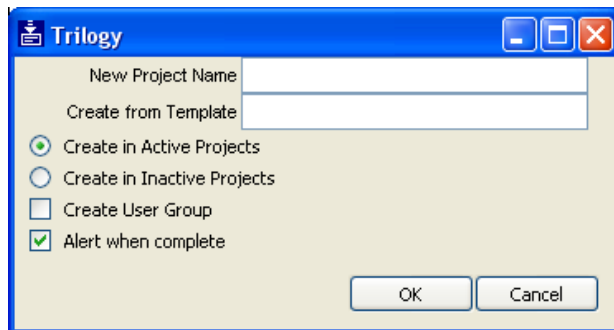
All radio buttons on the dialog are grouped together – selecting one clears all the others. If you want your dialog to have multiple sets of radio-buttons then you will need to group them using *Named Frames*. See *Creating Named Frames in Dialogs* below for more information.

Each checkbox or radio button corresponds to a TRIFIELD environment variable available to the server-side script, just as a data-entry field would. The values returned into these variables are "1" if the checkbox or radio button is selected and "0" otherwise.

Here is an example of a dialog containing checkboxes and radio buttons:

```
New Project Name      [
Create from Template  [
O Create in Active Projects
o Create in Inactive Projects
x Create User Group
X Alert when complete
```

This will be displayed by the *Trilogy* client like this:



Note that the capital O has resulted in the first radio button being preset. Similarly, the capital X has resulted in the second checkbox being preset.

Any server-side script run as a result of this dialog can access the radio buttons and checkboxes via TRIFIELD environment variables in the normal way. Thus, in this case:

TRIFIELD1	is the new project name
TRIFIELD2	is the template name
TRIFIELD3	is "1" if the radio button "Create in Active Projects" is set, "0" otherwise
TRIFIELD4	is "1" if the radio button "Create in Inactive Projects" is set, "0" otherwise
TRIFIELD5	is "1" if the checkbox "Create User Group" is selected, "0" otherwise
TRIFIELD6	is "1" if the checkbox "Alert when complete" is selected, "0" otherwise.

4.8.2 Pre-populating Dialogs containing Radio Buttons and Checkboxes

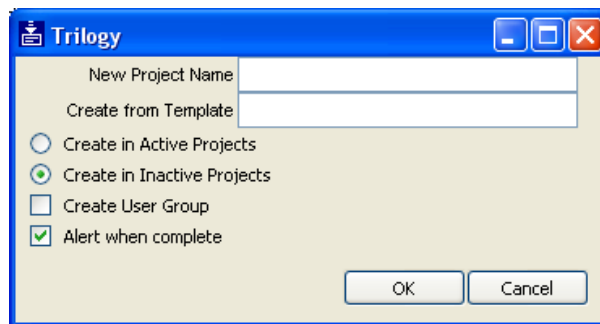
Radio buttons and checkboxes are data entry components that map to `TRIFIELD` environment variables just like regular data entry fields and drop-down lists. This means that they can be preset using `PopulateWith=` and `PopulateFieldnWith=` directives just like any other field.

In this case, a value of "1" sets the checkbox or radio button, any other value clears it. If fields are set with these directives then the default settings (indicated with the capital O or X marks) are ignored by *Trilogy*.

Thus, we could add a simple directive to `trilogy.conf` to select the other radio button:

```
demo2:
    Dialog=$TRILOGYHOME/demo/screens/demo2.scn
    PopulateField4With={1}
```

Now, when *Trilogy* displays this dialog, it selects field4 (the second radio button):



Of course, in itself this is not much use. What is of more value is setting the default values from a script. The script can then derive the appropriate values.

Remember that the output from a `PopulateWith=` script is mapped to the dialog with the first line of output going to field1, the second line to field2 and so on. Therefore, in this case, the third line of output should be "1" to set the first radio button, the fourth line should be "1" to set the second radio button and so on.

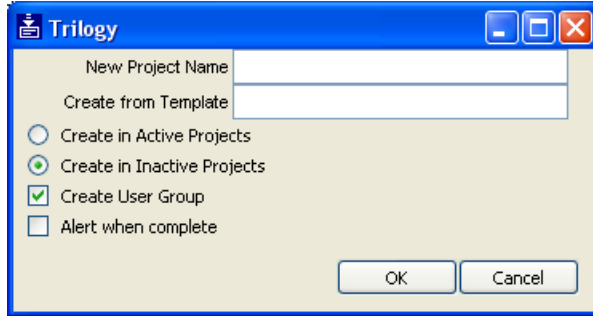
Therefore, if we write a simple script like this:

```
#!/bin/sh
echo "" # New Project Name
echo "" # Create From Template
echo "0" # Create in Active Projects (off)
echo "1" # Create in Inactive Projects (on)
echo "1" # Create User Group (on)
echo "0" # Alert when Complete (off)
```

and we add the `PopulateWith=` directive to `trilogy.conf`:


```
demo2:
Dialog=$TRILOGYHOME/demo/screens/demo2.scn
PopulateWith=$TRILOGYHOME/demo/scripts/demo2pop.sh
```

Then, when *Trilogy* displays the dialog, it sets the radio buttons and checkboxes according to the output from this script:



4.9 Disabling Fields

A field is automatically disabled should its associated `PopulateFieldnWith` script exit with a non-zero exit code. You can use this technique to disable fields under circumstances you determine.

4.10 Renaming the Buttons

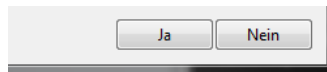
The two buttons "OK" and "Cancel" are present at the bottom of every dialog. The Cancel button simply closes the window with no further action being performed, the "OK" button submits the dialog and causes the *Trilogy* Server to run any server-side script associated with the `ValidateWith=` and `Program=` directives.

You may find it necessary to rename these buttons, either because of local language concerns or to better reflect their function. This is done with the `OKButtonText=` and `CancelButtonText=` directives. These directives allow the text on the buttons to be changed.

For example, including these directives in the server-side `trilogy.conf` for a job:

```
OKButtonText=Ja
CancelButtonText=Nein
```

Results in the dialog being displayed at the client with the default button names changed:



Note, the functionality of the button is not changed – only the text displayed on it.

4.11 Validating Dialog Data

One way of validating dialog data is to have your server-side script validate the submitted data at the point where the user submits the dialog. Obviously, this is desirable anyway but to have this as the *only* point of data validation it is not ideal from an end-user's perspective. Once the job has been submitted (by pressing OK), the GUI dialog disappears. It is very frustrating to fill out a (potentially) large dialog and to submit it, only to be told that there was some form of error in the input and the user is expected to re-enter all the data again.

Fortunately, *Trilogy* allows you to validate the contents of the dialog before it allows the "main" server-side (`Program=`) task to run. This is implemented with the `"ValidateWith="` directive in `trilogy.conf`.

The `ValidateWith=` directive specifies a server-side script that *Trilogy* will invoke in order to validate the form. As with the `PopulateWith=` and `PopulateFieldnWith=` directives, the specified script is passed all the command-line parameters that were passed to the *Trilogy* client. The `ValidateWith=` script is also passed the contents of the fields – in other words the environment variables `TRIFIELD1`, `TRIFIELD2` etc are all set to the values filled out on the *Trilogy* Client GUI. Given this information, the script will be able to validate the entered data.

Trilogy handles the output of this script in the following way:

If the script completes with an exit code of 0 then *Trilogy* assumes that validation was successful. In this case, the GUI disappears and the main program (specified with the `Program=` directive) is run.

If the script completes with a non-zero exit code then *Trilogy* assumes that validation has failed. In this case, the standard error output of the script is displayed as a pop-up box on the *Trilogy* client, and the GUI is not cleared. This allows the user to correct the error and submit the form again.

Here is an example of a simple validation script. This script ensures that the user has filled in all required values in our `get_credentials` job:

```
#!/bin/ksh
function ExitWithError
{
    echo "$*">&2
    # failure
    exit 1
}
[[ -z $TRIFIELD1 ]] && ExitWithError "You must specify a User Name"
[[ -z $TRIFIELD2 ]] && ExitWithError "You must specify a Password"
[[ -z $TRIFIELD3 ]] && ExitWithError "You must specify a Domain"
# success
exit 0
```

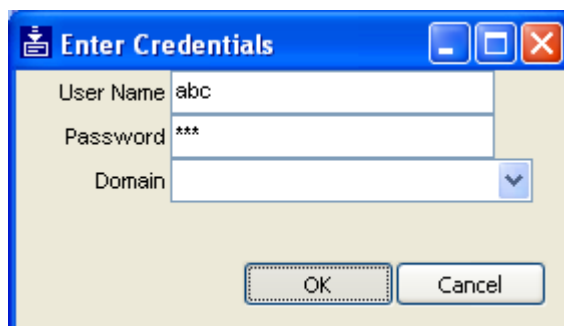
Assuming that this script is in `$TRILOGYHOME/SCRIPTS/validate` we can add the following directive to the server's `trilogy.conf`:

```
ValidateWith=$TRILOGYHOME/SCRIPTS/validate
```

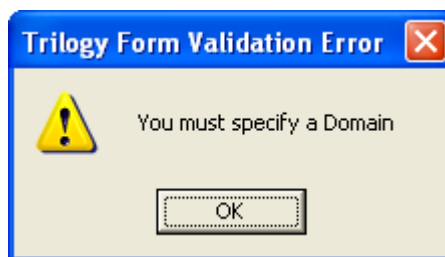
Making the whole stanza look like this:

```
GET_CREDENTIALS:  
  Title=Enter Credentials  
  Dialog=$TRILOGYHOME/SCREENS/credentials.scn  
  Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials  
  PopulateField3With={Domain1,Domain2,Domain3}  
  PopulateWith=$TRILOGYHOME/SCRIPTS/prepop  
  ValidateWith=$TRILOGYHOME/SCRIPTS/validate
```

If the user attempts to submit the dialog without having filled in both fields, they are now prevented from doing so:



Clicking OK on the client dialog, causes *Trilogy* to run the server side script `$TRILOGYHOME/SCRIPTS/validate`. Since `TRIFIELD3` is NULL, the script writes "You must specify a Domain" to its standard error and exits with a failure condition. At the client, a pop-up dialog appears, containing the validate script's standard error output:



Clicking OK on this pop-up dialog simply clears it, leaving the original data-entry dialog on screen, ready for another attempt.

4.12 Validating Command Line Parameters Before Displaying Dialog

You may find it necessary to validate the passed command-line parameters before bringing up the GUI for data entry.

In order to allow this, *Trilogy* supports the directive `PreValidateWith=` in the server-side `trilogy.conf` file. This directive refers to a script which, if specified, is invoked prior to the GUI being displayed. This script is passed all the command line parameters which were included in the call from the *Trilogy* client.

If this *Pre-Validate* script returns 0 then the GUI is displayed in the normal way. If it returns any other value then the standard error output from the script is displayed as a pop-up dialog box at the client and the GUI dialog is not displayed.



Since the nodename and username of the client is made available in the environment variables `TRICLIENTNODENAME` and `TRICLIENTUSERNAME`, you can use this technique to deny access to functions based on either the location from where the request is being made or the user who is performing the operation. See *Server Side Scripts – Environment Variables set by Trilogy* later in this document.

4.13 Using the Same Script to perform Multiple Functions

Discussions so far have shown multiple scripts each performing a single function. We have shown how to create scripts to:

- populate the dialog
- populate a field
- validate form data
- pre-validate the command line parameters
- execute once the dialog's "okay" button has been pressed.

Now supposing we want to populate multiple fields – if we had a separate script for each field then we could potentially have a lot of different scripts to maintain.

To work around this problem, *Trilogy* allows you to write a single script and refer to it multiple times in `trilogy.conf`. When the script is invoked, *Trilogy* sets environment variables that allow the script to determine why it has been called and to perform the appropriate action.

The main environment variable that controls this is `TRIReason`. When the script is invoked, it can read `TRIReason` which will be set to one of the following values:

<code>PREVALIDATION</code>	Script has been run as a result of a <code>PreValidateWith=</code> directive.
<code>VALIDATION</code>	Script has been run as a result of a <code>ValidateWith=</code> directive.
<code>POPULATE</code>	Script has been run either as a result of a <code>PopulateWith=</code> or a <code>PopulateFieldnWith=</code> directive.
<code>SCRIPT</code>	Script has been run as a result of a <code>Program=</code> directive.
<code>LISTBOX</code>	Script has been run as a result of a <code>ListBoxScript=</code> directive (see later sections for a description of the listbox).
<code>DIALOG</code>	Script has been run as a result of a <code>DialogScript=</code> directive (see later sections for a description of creating dialogs from scripts).
<code>TIMED</code>	Script has been run as a result of the built in scheduler (See Chapter 10 later in this document for a discussion of the scheduler).

In addition, the environment variables `TRICurrentField` and `TRICHangedField` are set whenever `TRIReason` is set to "POPULATE". `TRICurrentField` reflects the field number that is being populated. If 0, this means that the script has been invoked via a `PopulateWith=` directive and the whole dialog is being pre-populated. Any other number gives a reference to the field number to be

populated (remember that fields start at 1 and are numbered from left-to-right and top-to-bottom).



The demo that ships with the *Trilogy* Server uses a single script to populate and validate the client dialog and to provide the run-time functionality (when the user presses Ok). Examine this script to see how these environment variables are used. The demo script is in `$TRILOGYHOME/demo/scripts/demo.sh` on Unix Servers and in `%TRILOGYHOME%\demo\scripts\demo.bat` on Windows Servers.

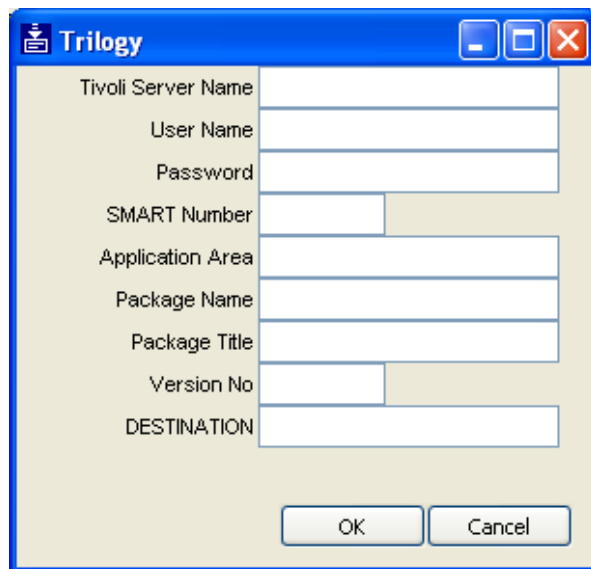
4.14 Creating Named Frames in Dialogs

A named frame is a means by which logical groups of fields can be represented on a dialog. Named frames are often used within GUI based applications to clarify and group together various data entry components.

Trilogy supports the creation of named frames by the use of special lines within the screen definition. For example, take this as an example dialog definition file:

```
Tivoli Server Name [
    User Name [
        Password [*
            SMART Number [
Application Area [
    Package Name [
    Package Title [
        Version No [
    DESTINATION [
```

This will be displayed by *Trilogy* as follows



Whereas this will work it is not particularly attractive to look at and it looks less like a native Windows/Unix GUI application.

Named frames can be created in the dialog definition file by starting a line with a single dash followed by a space and then the name of the frame. A single dash on its own ends the frame. This is only required if you wish to have fields outside of a frame – if you create a new frame the old one is closed automatically.

Here is an example:

```

- Tivoli Server
Tivoli Server Name [
    User Name [
        Password [*
]

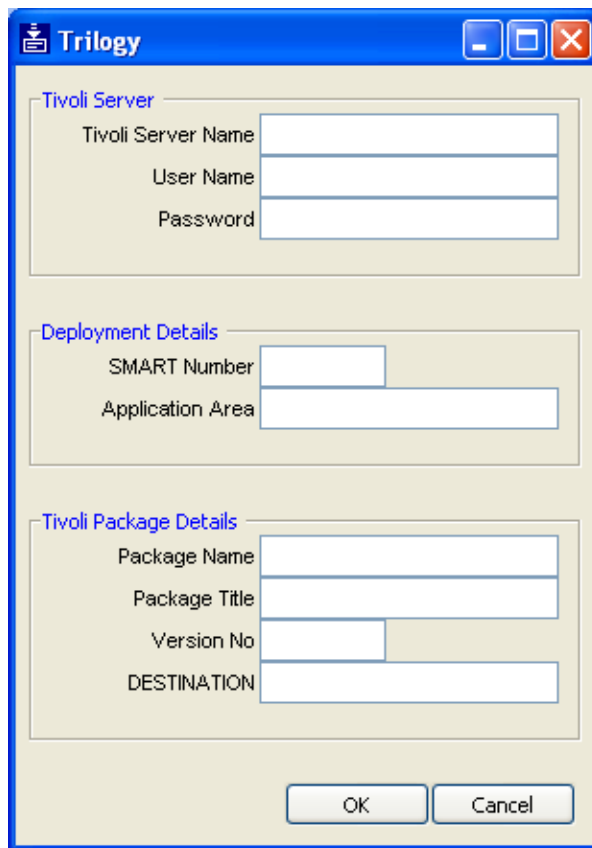
- Deployment Details
    SMART Number [
    Application Area [
]

- Tivoli Package Details
    Package Name [
    Package Title [
    Version No [
    DESTINATION [
]
-

```

Here, the dialog has been broken up into three distinct "frames". The first three fields are included in the frame "Tivoli Server", the next two (*SMART Number* and *Application Area*) are in the frame "Deployment Details" and the remaining fields are grouped together in the frame "Tivoli Package Details".

Trilogy will display this dialog as follows:



The screenshot shows a Windows-style dialog box titled "Trilogy". It contains three distinct frames, each with a title bar and a group of input fields:

- Tivoli Server**: Contains three input fields labeled "Tivoli Server Name", "User Name", and "Password".
- Deployment Details**: Contains two input fields labeled "SMART Number" and "Application Area".
- Tivoli Package Details**: Contains four input fields labeled "Package Name", "Package Title", "Version No", and "DESTINATION".

At the bottom of the dialog are "OK" and "Cancel" buttons.

This obviously looks a lot clearer and more like a "proper" dialog.

4.15 Grouping Radio Buttons with Named Frames

Another advantage of named frames is that *Trilogy* uses them to group radio buttons together. In other words, radio buttons in one frame operate independently of radio buttons in another frame.

You may need to do this if your dialog requires different groups of radio buttons. As an example, let's create a front-end to some simple server-side library search tool.

The dialog is defined like this:

```
- Search Criteria
ISBN      [
Title     [
Author    [
- Edition Filter
O Latest Edition Only
o All Editions
- Location Filter
O This Library Only
o All Libraries within 10 mile radius
o Country-wide
- Language Filter
O Any          o English      o French
O German       o Dutch        o Polish
```

There are now three distinct "groups" of radio buttons each of which acts independently from the other:

The first group is declared in the frame "Edition Filter" and includes the two options for "Latest Edition Only" and "All Editions"

The second group is declared in the frame "Location Filter" and includes three options for "This Library Only", "All Libraries within 10 mile radius" and "Country-wide".

The third group is declared in the frame "Language Filter" and consists of 6 options, Any, English, French, German, Dutch and Polish.

Trilogy interprets and displays this dialog as follows:

The screenshot shows a window titled "TrilogY" with a blue title bar. Inside, there are four sections, each with a blue header and a light beige background:

- Search Criteria:** Contains three text input fields labeled "ISBN", "Title", and "Author".
- Edition Filter:** Contains two radio buttons: "Latest Edition Only" (which is selected) and "All Editions".
- Location Filter:** Contains three radio buttons: "This Library Only" (selected), "All Libraries within 10 mile radius", and "Country-wide".
- Language Filter:** Contains six radio buttons arranged in two rows: "Any" (selected), "English", "French", "German", "Dutch", and "Polish".

At the bottom right of the window are "OK" and "Cancel" buttons.

Each group of radio buttons in each named frame now operates independently of the other. In other words, selecting the "All Editions" radio button automatically clears the "Latest Editions Only" radio button since both of these buttons are in the same group. However, no other radio button is affected.

Radio Buttons within tabs are also grouped together and operate independently. See *Creating Tabbed Dialogs* below for more information.

4.16 Creating Tabbed Dialogs

If your dialog is particularly complex (there are lots of data entry components grouped into different logical areas) then you may find it desirable to create a *tabbed dialog*. A tabbed dialog groups fields into a number of different pages (or *tabs*) which the user can then select. Displaying one tab automatically hides the fields on the other tabs.

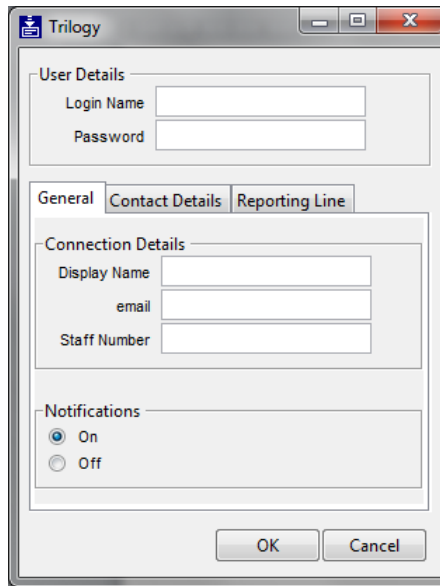
A tab can be created in the dialog definition file by starting a line with a single > character ("Greater Than" symbol) followed by a space and then the name of the tab. All the fields defined below this line will then be placed in this tab. When you want to create a new tab, enter another > line and all the fields below this will be placed in the new tab.

Here is an example:

```
- User Details
Login Name [ ]
Password [ ]
> General
- Connection Details
Display Name [ ]
email [ ]
Staff Number [ ]
- Notifications
O On
o Off
> Contact Details
- Contact Details
Mobile No [ ]
Desk Number [ ]
- Branch Information
Branch ID { }
> Reporting Line
- Line Managers
Line Manager { }
Deputy { }
```

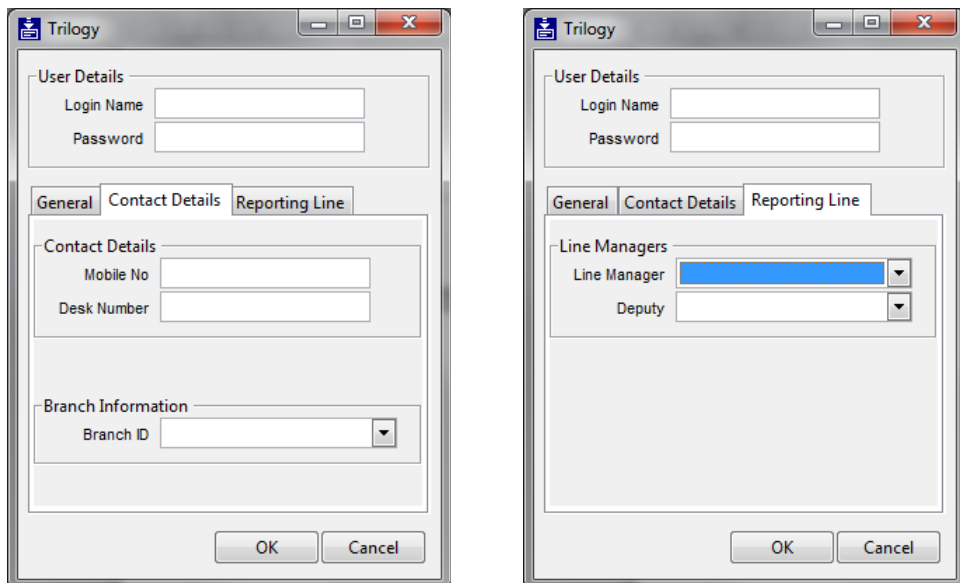
Here, 3 tabs have been specified with the > marker, namely "General" (line 4), "Contact Details" (line 12) and "Reporting Line" (line 18). The fields specified between each tab definition are then rendered in a separate tab. The tab identified as "General" will contain the fields from "Display Name" (TRIFIELD3) to the second radio button in the "Notifications" frame (TRIFIELD7). The tab identified as "Contact Details" will contain the fields from "Mobile No" (TRIFIELD8) to "Branch ID" (TRIFIELD10) and the tab identified as "Reporting Line" will contain the remaining fields "Line Manager" and "Deputy" (TRIFIELD11 and TRIFIELD12).

When displayed at the client, *Trilogy* will render the dialog like this:



The image shows a Windows-style dialog box titled "Trilogy". It has three tabs: "General", "Contact Details", and "Reporting Line". The "General" tab is selected. Inside the "General" tab, there are three sections: "User Details" with "Login Name" and "Password" text boxes; "Connection Details" with "Display Name", "email", and "Staff Number" text boxes; and "Notifications" with "On" (selected) and "Off" radio buttons. At the bottom are "OK" and "Cancel" buttons.

Only the fields contained within the first tab are displayed. Clicking on each tab brings up the set of fields appropriate to that tab and hides the others:



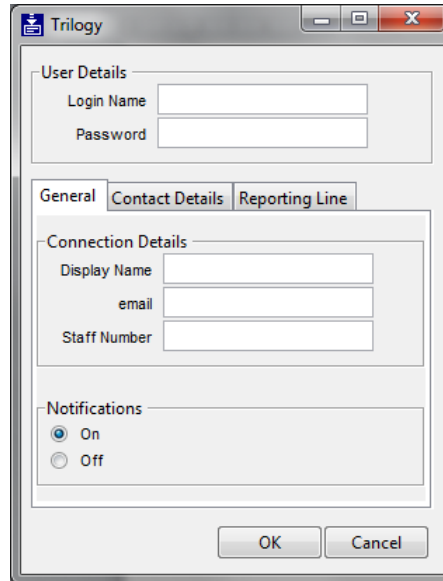
The image shows two side-by-side instances of the "Trilogy" dialog box. The left instance has the "Contact Details" tab selected, showing "Mobile No" and "Desk Number" text boxes, and a "Branch Information" section with a "Branch ID" dropdown. The right instance has the "Reporting Line" tab selected, showing a "Line Managers" section with a "Line Manager" dropdown (highlighted in blue) and a "Deputy" dropdown. Both instances have "OK" and "Cancel" buttons at the bottom.



Field numbering is not affected by grouping fields into different tabs. In the case TRIFIELD8 is the first field on the "Contact Details" tab, TRIFIELD11 is the first field on the "Reporting Line" tab.

4.17 Adding Banners

You can add a banner to the top of any *Trilogy* client-side dialog by placing appropriate entries into the server-side `trilogy.conf` file. Adding a banner to our tabbed dialog changes its appearance from this:

A screenshot of the Trilogy application window. The title bar says "Trilogy". Inside, there's a "User Details" section with "Login Name" and "Password" text boxes. Below that are three tabs: "General", "Contact Details", and "Reporting Line". The "General" tab is selected, showing a "Connection Details" section with "Display Name", "email", and "Staff Number" text boxes. At the bottom of the "General" tab is a "Notifications" section with "On" (selected) and "Off" radio buttons. At the very bottom are "OK" and "Cancel" buttons.

to this:

A screenshot of the Trilogy application window, now with a banner. The title bar says "Trilogy". The banner area at the top contains the text "Enter User Info" and "Please enter the information for the new user" next to a download icon. Below the banner is the same "User Details" section with "Login Name" and "Password" text boxes. Below that are the same three tabs: "General", "Contact Details", and "Reporting Line". The "General" tab is selected, showing the same "Connection Details" section with "Display Name", "email", and "Staff Number" text boxes. At the bottom of the "General" tab is the same "Notifications" section with "On" (selected) and "Off" radio buttons. At the very bottom are "OK" and "Cancel" buttons.

The screen definition file remains the same. The banner content is determined purely by entries in the server-side `trilogy.conf` file.

Here is the new job definition on the *Trilogy* Server:

```

GET_CREDENTIALS:
    Title=Enter Credentials
    Banner=on
    BannerHeading=Enter User Info
    BannerText=Please enter the information for the new user
    Dialog=$TRILOGYHOME/SCREENS/credentials.scn
    Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
    PopulateField3With={Domain1,Domain2,Domain3}
    PopulateWith=$TRILOGYHOME/SCRIPTS/prepop
    ValidateWith=$TRILOGYHOME/SCRIPTS/validate

```

The new directives are:

```
Banner=on
```

Switches the banner on. This directive can be placed either outside of a job-stanza (i.e.: at the global level alongside `Port=` and `Server=` directives) in which case it switches the banner on for any job which doesn't explicitly specify `Banner=off`. Or it can be placed in the job stanza entry (as shown above) in which case it switches the banner on for that job only.

The next directive is:

```
BannerHeading=Enter User Info
```

Specifies the text to be displayed on the first line of the banner. If this directive is not specified and `Banner=on`, then the first line of the banner will be the same as the job title (specified with the `Title=` directive).

The next directive is:

```
BannerText=Please enter the information for the new user
```

This specifies the text to be displayed on the *second* line of the banner (in a smaller font). If this directive is not specified and `Banner=on` then the second line of the banner is left blank.

The right-hand side of the banner contains an icon. This defaults to the *Trilogy* icon as shown. However, this can be easily changed by including a directive:

```
BannerGraphic=<path to GIF file>
```

in the server-side `trilogy.conf`. Note, that the GIF file specified must reside on the server. In common with all *Trilogy* dialogs, all definitions are server-side.

As an example, let us take a GIF file containing an image and reference it in the `trilogy.conf` job definition. The complete stanza entry now reads like this:

```

GET_CREDENTIALS:
    Title=Enter Credentials
    Banner=on
    BannerHeading=Enter Credentials
    BannerText=Please enter your login details
    BannerGraphic=$TRILOGYHOME/SCREENS/woman.gif
    Dialog=$TRILOGYHOME/SCREENS/credentials.scn
    Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
    PopulateField3With={Domain1,Domain2,Domain3}
    PopulateWith=$TRILOGYHOME/SCRIPTS/prepop
    ValidateWith=$TRILOGYHOME/SCRIPTS/validate

```

Note the `BannerGraphic` entry in the job stanza. This points at the server-side GIF file "woman.gif". If the GIF image looks like this:



Then, when the client dialog is displayed, it will look like this:

Note, there are size and format limitations of the GIF file. The maximum dimensions are 200 pixels wide by 52 pixels high. If the GIF image exceeds these dimensions then it is ignored and the default *Trilogy* Icon image is used. For more information see *BannerGraphic* in *Chapter 13 – trilogy.conf reference guide*.



The *Crystal Icon Set* is included in the `icons` directory on the server (`$TRILOGYHOME/icons`). You can use these icons in your dialogs.

4.18 Creating Dialogs with Scripts

As well as static screen definition files as shown, *Trilogy* also supports a `DialogScript` directive in the server-side `trilogy.conf` file. When used in a job stanza, this tells the server to run the specified script and use its standard output as a screen definition file. From that point onwards, *Trilogy* operates as if the screen definition had been generated from a static script.

As usual, the server-side dialog script is passed all the command line parameters that were passed to the client when it was invoked. This means that the script can generate different output depending on how it was invoked. Therefore the dialog displayed at the client can change dependent on the parameters passed at run-time.

As an example, we will modify the `get_credentials` job shown earlier in this section. We will change it so that the "Domain" drop-down list is only shown if "showdomain" is passed as a parameter:

First, we create a script to generate the dialog:

```
#!/bin/ksh
echo "User Name  ["
echo "Password  ["*
#
# Now iterate through the parameters, looking for "showdomain"
#
for p in "$@"
do
    if [[ "$p" = "showdomain" ]]
    then
        #
        # Echo the domain drop-down list and exit
        #
        echo "Domain      {"
        exit 0
    fi
done
exit 0
```

Then, we change the job definition in the server-side `trilogy.conf` file to use this script rather than the static file "credentials.scn":


```

GET_CREDENTIALS:
    Title=Enter Credentials
    Banner=on
    BannerHeading=Enter Credentials
    BannerText=Please enter your login details
    BannerGraphic=$TRILOGYHOME/SCREENS/woman.gif
    #Dialog=$TRILOGYHOME/SCREENS/credentials.scn
    DialogScript=$TRILOGYHOME/SCRIPTS/gencredscn.sh
    Program=$TRILOGYHOME/SCRIPTS/ProcessCredentials
    PopulateField3With={Domain1,Domain2,Domain3}
    PopulateWith=$TRILOGYHOME/SCRIPTS/prepop
    ValidateWith=$TRILOGYHOME/SCRIPTS/validate

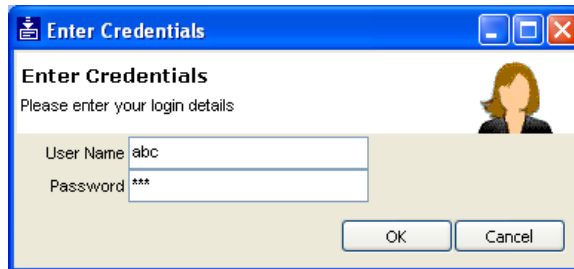
```

Note, if the dialog only has two fields in it then the `PopulateField3With` directive will not be invoked.

Invoking the *Trilogy* client like this:

```
trilogy get_credentials abc def
```

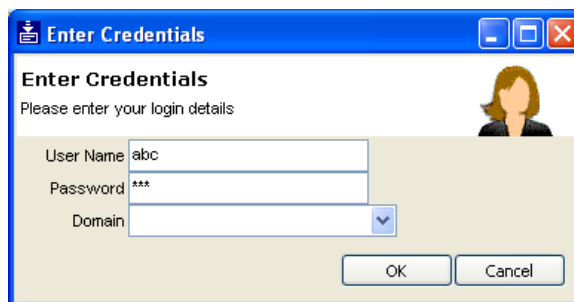
...causes the Trilogy server to run `gencredscn.sh`. This outputs two lines of screen definition. The script then iterates through the parameters ("abc" and "def") and – finding no "showdomain" flag – it simply exits with success. The resulting dialog is then shown at the client:



Invoking the client like this:

```
trilogy get_credentials abc def showdomain
```

...causes the server-side script "gencredscn.sh" to output a third line containing the drop-down field for the domain. The `PopulateField3With` directive is then used to populate the field in the normal way. The resulting dialog is then shown at the client:



4.19 Creating an Icon in the System Tray



This section only applies to Windows Clients. Jobs can still be invoked from Linux/Unix clients but since they do not have a System Tray (Notification Area), they will not exhibit the behaviour described in this section.

Windows Clients can be set so that an icon is created in the System Tray (Notification Area) whilst the job is running. This icon animates whilst the job is running to provide a visual indication that a server-side job is in progress.

Having a job iconize to the system tray also allows it to receive pop-up “Balloon” style notifications from the server-side job. This allows the server-side job to send out notifications as it runs, notifying the user of progress or error conditions etc.



If the Windows client is running the *Trilogy Client Service* then the icon will be permanently present in the System Tray. In these circumstances, the existing icon is re-used. Only one *Trilogy* Icon will appear in the System Tray regardless of how many jobs the user invokes. For more information see Chapter 8 later in this document.

In order that the client “docks” to the System Tray, the following directive needs to be included in the relevant job stanza in the server-side `trilogy.conf` file:

SystemTray=Yes|No|Hold

No	is the default and means no icon is created in the client System Tray.
Yes	means that a <i>Trilogy</i> Icon is created in the client’s System Tray whenever the job is running. Note, the icon appears when the program or script identified by the Program= directive is invoked (after any dialog has been presented and committed by clicking OK). When the server-side job completes, the icon disappears (after any balloon notifications have been dismissed or have timed-out).
Hold	This option is the same as Yes except that when the server-side job completes, the icon does not automatically disappear. It stops animating and displays a “success” or “fail” icon and is only removed once the user right-clicks on the icon and selects “Quit”.



If the Windows client is running the *Trilogy Client Service* then the icon will not disappear following the completion of the server-side job.

Docking a job to the System Tray may be useful in the following circumstances:

- The server-side job is likely to run for a considerable period of time. In these circumstances it is useful to give the user feedback that the job is still running.
- You wish to send "Balloon" style notifications to the client from the server-side job. You need to have an Icon present in the System Tray in order for the Balloon to be issued. If the client is not running the *Trilogy Client Service* then you will need to create an icon in order to issue such notifications.

There are some other differences to be aware of when the job is docked to the System Tray.

1. If `Stdout` or `Stderr` is set to "Report" then the report window is not initially displayed. The user can open the Report Window by double-clicking on the *Trilogy* Icon in the System Tray. Similarly, iconizing the Report Window causes it to disappear from the Task Bar – it effectively iconizes to the System Tray.
2. If `Stdout` or `Stderr` is set to "Popup" then the standard out/standard error streams are presented as Balloon Notifications. If the standard out/standard error streams contain more data than will fit into a single balloon, then the output is split across as many balloons as necessary – the next balloon displaying automatically when the first is dismissed or times out.

5 The List Box

5.1 Introduction

Trilogy client dialogs can also contain a *list box*. If present, the list box is shown at the bottom of the client dialog. Just like drop-down lists, a list box is populated from a server-side script. However, unlike drop-down lists, the output from the script is not only parsed line-by-line but also by column. It is therefore possible to assemble a list box containing various columns of information.

When the dialog is submitted, rows selected in the client-side list box are made available to the invoked server-side script by the use of environment variables in the same way as drop-down lists, entry fields and radio buttons/checkboxes.



Windows Servers can also use the *Trilogy Scripting Engine* to retrieve the selected contents of the list box.

Here is an example dialog, containing a list-box:

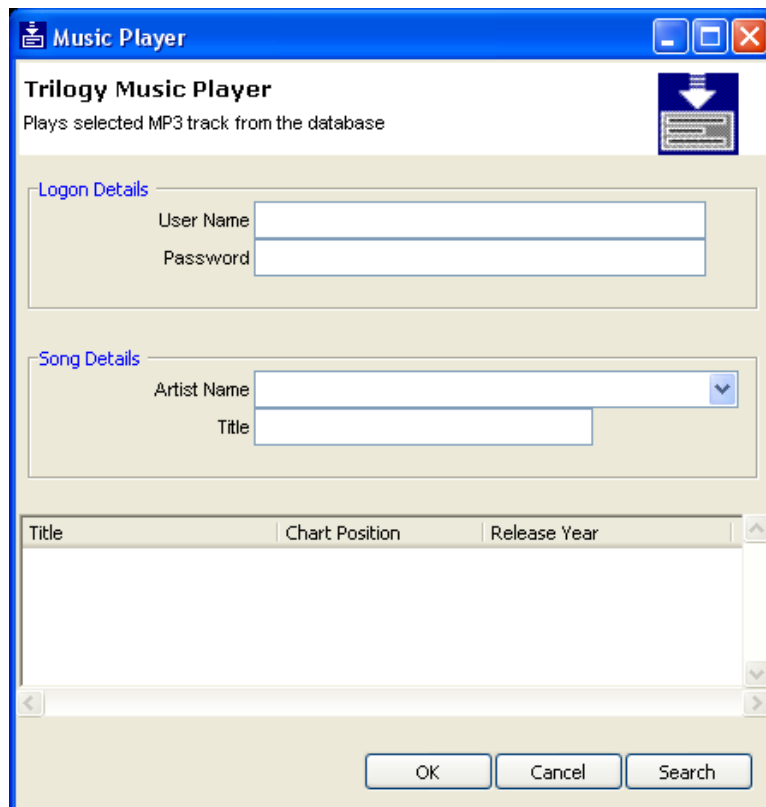


Figure 5.1: A list box included in the dialog.

The user can select an "Artist Name" from the drop-down list. When the Search button is clicked, the list box is populated:

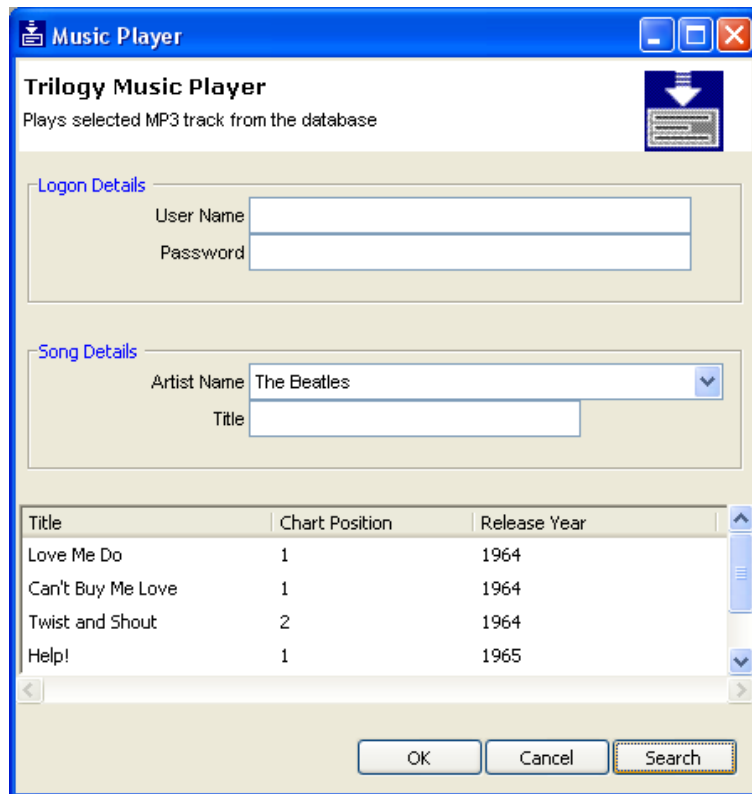


Figure 5.2: A populated list box.

5.2 List Box Directives

There are a number of directives in the server-side `trilogy.conf` file that control the presence and behaviour of the list-box when it is displayed at the client:

5.2.1 Controlling the List Box Appearance

```
ListBox=on|off|auto
```

Included in a job stanza, this directive controls the presence of the list box on the associated client-side dialog. If `ListBox=on` (or `yes`) then the list box is displayed. If `ListBox=off` (or `no`) then the client-side dialog contains no list box. This is the default. If `ListBox=auto` then the list box is only added to the client-side dialog if the server-side script that generates its content actually outputs some data to its standard output stream. If the server script's standard output contains no data, no list box is displayed.

5.2.2 Creating List Box Content

The server side script that generates the list-box contents is identified by the `ListBoxScript` directive:

```
ListBoxScript=<pathname>
```

or

```
PopulateListBoxWith=<pathname>
```

Both of these directives are functionally identical and you can use either. The directive identifies the location of the server-side script that generates the list box content.

When the list box requires to be populated, the server-side script identified by the `ListBoxScript` (or `PopulateListBoxWith`) directive is executed. It is passed any parameters specified in the job stanza (via any `Param` or `Params` directives), followed by any parameters passed to the *Trilogy* command line client or added via the *Trilogy Scripting Engine*. The standard output from this script is then parsed to produce the list box content for display on the client dialog. Each line of script output is taken as a new record and each will occupy its own row within the list box. Similarly, within each line of output, *Trilogy* will split the record into individual fields and each field is then presented in its own column within the list box.

To generate the output shown in the list box in the example above, the server-side list box script (identified by the `ListBoxScript` or `PopulateListBoxWith` directive) will have to generate the following to its standard output:

```
Love Me Do,1,1964  
Can't Buy Me Love,1,1964  
Twist and Shout,2,1964  
Help!,1,1965  
Yellow Submarine,2,1966  
Lady Madonna,4,1968  
Hey Jude,1,1968
```

Trilogy will then read each line of this output, split it at each field separator character (default is a comma but this can be changed) and inserts the resulting set of columns into the appropriate fields within the displayed list box at the client.

The List Box Script has access to all the `TRIFIELD` environment variables which indicate the content of the dialog (`$TRIFIELD1` being the first field, `$TRIFIELD2` being the second field and so on). In this way, the server-side script can control its output based on the content of the client-side dialog. In the example above, the List Box Script will read the value of `$TRIFIELD3` (The "Artist Name" drop-down list) to identify the artist being requested, `$TRIFIELD4` (The "Title" data entry field) and will use the two values to assemble a list of matching songs which are then displayed in the client list box.

5.2.3 Identifying List Box Column Names

Each column of data in a list box has a header containing text which identifies the content of the column. In the example above the headers were "Title", "Chart Position" and "Release Year".

You can set the text which appears in these column headers in one of two ways.

The first way is to use the `ColumnNames` directive. This takes a list of column names in the usual *Trilogy* list syntax. For example, to include the columns "Title", "Chart Position" and "Release Year" as shown in the example above, the following directive would have to be included in the relevant job stanza entry in the server-side `trilogy.conf`:

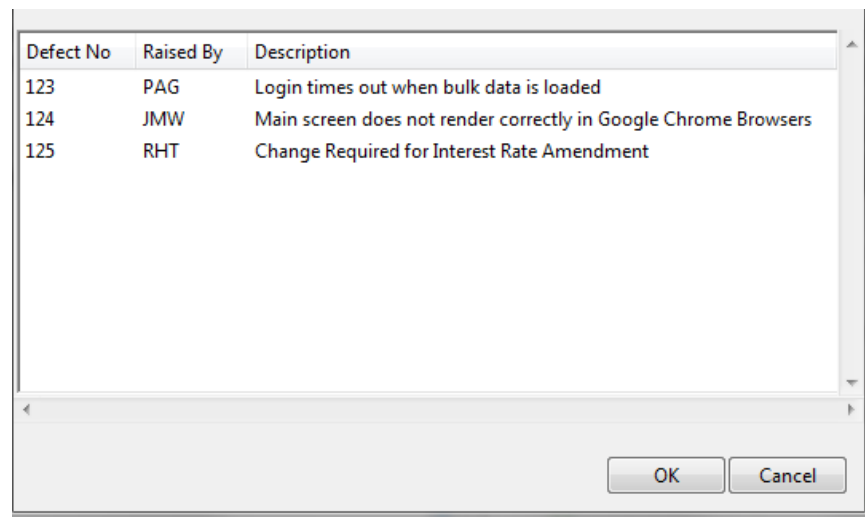
```
ColumnNames={Title,Chart Position,Release Year}
```

The second way is to allow the output from the List Box Script to define the header values. This happens when no `ColumnNames` directive is given in the job stanza entry. Under these circumstances, *Trilogy* will take the *first* row of data output by the List Box Script and use it to define the text which appears in each column header.

For example, assume the list box script outputs the following text to its standard output:

```
Defect No,Raised By,Description
123,PAG,Login times out when bulk data is loaded
124,JMW,Main screen does not render correctly in Google Chrome Browsers
125,RHT,Change Required for Interest Rate Amendment
```

Trilogy will render the List Box as follows:



Defect No	Raised By	Description
123	PAG	Login times out when bulk data is loaded
124	JMW	Main screen does not render correctly in Google Chrome Browsers
125	RHT	Change Required for Interest Rate Amendment

5.2.4 Identifying List Box Column Widths

The initial widths of the List Box columns are controlled by the `ColumnWidths` directive. Included in the same job stanza as the `ColumnNames` directive (within the server-side `trilogy.conf`) this directive specifies the width of each column in pixels. Note that this width is the *initial* width. The user of a *Trilogy* dialog can expand and contract the width of each displayed column within the list box by dragging the border between the columns with the left-hand mouse key depressed.

```
ColumnWidths={150,180,100}
```

In the case above, the "Title" column would have an initial width of 150 pixels, the "Chart Position" column would have an initial width of 180 pixels and the "Release Year" column would have an initial width of 100 pixels.



If you specify fewer entries in `ColumnWidths` than there are columns generated by the List Box Script then any "additional" columns are sized automatically. In other words, if you specify `ColumnWidths={100,100,120}` and the script generates 5 columns of data, then columns 4 and 5 will be sized automatically. See *Auto Sizing Columns* below for more information.

5.2.5 Auto Sizing Columns

You can specify a column width of "-" (a single dash character without quotes) to indicate to *Trilogy* that the column width should be calculated automatically. In this case, *Trilogy* will determine the width of the longest piece of data in that column and use that to set the column width. Therefore the column will be wide enough to display the longest data item in that column automatically.

Be careful using this option as long data may adversely affect the aesthetics of the dialog – especially if `AutoStretch` is set (see *Controlling List Box Width* below for more information).

Columns are sized automatically if there is no `ColumnWidth` directive (in which case all columns are sized automatically) or the column number being analysed is greater than the number of columns specified in the `ColumnWidth` directive.

For example, this:

```
ColumnWidths={100,120,-,140}
```

Means that the first column will have a width of 100 pixels; the second column will have a width of 120 pixels, the third column will be sized automatically to accommodate the data contained within it and the fourth column will have a width of 140 pixels.

When a column is sized, the text included in the header is taken into account. *Trilogy* will not size the column such that the text in the header is truncated. For example, if the Column Name is "Reported?" and the column contains the values "Y" and "N", *Trilogy* will create the column width so that it is wide enough to contain the word "Reported?" in the header (rather than making it wide enough to only contain "Y" or "N").

5.2.6 Creating Hidden Columns

Setting a column width of 0 (using the `ColumnWidths` directive above) causes the column to be treated by *Trilogy* as a special case. A column with a width of 0 is not displayed within the list box at the *Trilogy* Client, nor can it be expanded in order to be made visible. It does, however, still have content and the output from the list box script is still used to populate its content. This can be used to create “hidden” fields whose contents can be passed to other server-side scripts on actions from the *Trilogy* Client – this will be detailed later in this chapter.

5.2.7 Controlling List Box Width

The client-side dialog will normally stretch to accommodate the width of all the columns in the list box. The result of this is that the list-box is normally displayed without a horizontal scroll bar.

This is the default behaviour. However, there may be times when you wish the “main” *Trilogy* dialog to be narrower than the list box. This may be required when the list-box includes a lot of columns or the columns are particularly wide. In this case, the “main” *Trilogy* dialog may stretch beyond the point where it looks aesthetically pleasing.

The directive `AutoStretch` controls the automatic scaling of the List Box. If it is absent from the server side job-stanza then the value defaults to `Yes`. This means that the “main” dialog will stretch to meet the initial width of the List Box (which means the width of all the columns included in it).

By including the directive `AutoStretch=no` (or `off`), then this behaviour is disabled. In this case the width of the dialog is controlled either by:

- a) The length of the longest line within the dialog specified by the `Dialog` or `DialogScript` directive in the job stanza or
- b) The width given by the `ListBoxWidth` directive. If a `ListBoxWidth` is specified then the dialog will grow horizontally (if required) to the desired width in pixels. If the “parent” dialog is wider than the specified `ListBoxWidth` then the list box will stretch to the width of the parent dialog.

In either case the List Box content will be constrained by the width of the List Box. A horizontal scroll bar will be presented if required to allow the user to scroll the list box sideways to view all the columns.



Specifying a `ListBoxWidth` will automatically turn off the `AutoStretch` option.

5.2.8 Controlling List Box Height

The height of the List Box is controlled by the `ListBoxHeight` directive. This specifies the height of the list box in rows. If this value is not specified it defaults to 4.

5.2.9 Adding an “Apply” Button to the Dialog

The directive `ApplyButton` within a job stanza in the server-side `trilogy.conf` controls the appearance of the apply button on the client dialog when it is displayed. If it is set to `On` (or `yes`), the apply button appears. If it is set to `No` (or `off` – the default) the Apply button is absent. Setting `PopulateListBox=OnApply` will add the Apply Button automatically regardless of the setting of the `ApplyButton` directive – see above.

The text on the button can be changed from the default (Apply) by use of the `ApplyButtonText` directive. This can be used to specify any text to appear on the button instead of “Apply”. For example:

```
ApplyButtonText=Search
```

Causes the dialog to display a “Search” button instead of “Apply”. If `PopulateListBox=OnApply` then clicking the “Search” button will cause the server-side List Box Script (identified with `ListBoxScript` or `PopulateListBoxWith` in the same job stanza) to be executed and the results to be sent to the client-side list box within the Trilogy Dialog.

5.2.10 Sorting the List Box

Once the List Box is displayed, the user can click on any of the column headers to sort the List Box by that column. When the column heading is first clicked, the sort takes place in ascending mode (A-Z, 0-9). When the column is clicked again the sort takes place in descending mode (Z-A, 9-0).

When a sort has been applied to a List Box column then a “Decorator” will be applied to the column heading, indicating that a sort has been applied. This decorator will take the place of an upward arrow (for ascending sorts) or a downward arrow (for descending sorts).

The sort takes place in “Dictionary Order”. *Trilogy* will ignore case (except when two character strings would otherwise be identical). In addition, if two strings contain embedded numbers then the numbers compare as integers, not characters. For example, “bigBoy” would sort between “bigbang” and “bigboy”, and “x10y” would sort between “x9y” and “x11y”.

5.2.11 Automatic Sorting

Trilogy allows a sort to be applied automatically after the list box is populated with the `ListBoxScript` (either when the dialog is first displayed or when the

apply button is clicked, depending on the setting of `PopulateListBox`). This is functionally identical to the user clicking on the appropriate column heading after the List Box has been populated except for the fact that it occurs automatically.

To specify an automatic sort, use the `AutoSort` directive in the job stanza in the server-side `trilogy.conf` file. `AutoSort` has the following syntax:

```
AutoSort=[-]ColumnNumber
```

The `-` character is optional. If present, it indicates the column should be sorted in descending mode. If absent, the column is sorted in ascending mode.

The `ColumnNumber` parameter indicates to *Trilogy* the number of the column in the List Box which should be automatically sorted following the run of the `ListBoxScript`. Columns start at 1 and increase from left to right.



Hidden columns need to be counted. For example, if your column widths are `{0,100,50,100}` and you want to sort by the second *displayed* column, then you would need to set `AutoSort` to 3 (the third column) since column 1 is not displayed (it is a hidden column).

For example:

```
AutoSort=2
```

Will cause *Trilogy* to sort the List Box in ascending order of the second column after the server-side `ListBoxScript` has been invoked to populate the List Box.

```
AutoSort=-3
```

Will cause *Trilogy* to sort the List Box in descending order of the third column after the server-side `ListBoxScript` has been invoked to populate the List Box.

After `AutoSort` has been applied, the appropriate “Decorator” is placed on the column to indicate it has been sorted.



`AutoSort` can sort by a hidden column. This is obviously not possible manually. If sorting has taken place on a hidden column then no decorator is visible (since the hidden column has no header).

5.2.12 Controlling List Box Selections

The list box can be configured to allow multiple, single or no rows to be selected by the user. The directive `Selections` controls this behaviour.

`Selections=Multiple`. This allows the user to select one or more rows from the list box. This is the default option. If the `Selections` directive is not specified in the job stanza then the user will be allowed to select one or more rows in the list box.

`Selection=Single`. This only allows the user to select a single row from the List Box.

`Selection=None`. This disables the selection of any rows in the List Box. However, *Trilogy* itself can still select rows using the `AutoSelect` mechanism. See *Automatically Selecting List Box Rows* below for more information.

5.2.13 Selecting List Box Rows

A user can select one or more rows in the list box (dependent on the setting of the `Selections` directive in the job stanza). When the dialog is submitted, these selected rows are made available to the server-side script (identified by the `Program=` directive in the job stanza) via the use of environment variables. This is the same technique used to pass Dialog Content and other information to the script as detailed in the previous sections.

Unlike Dialog Content, these “List Box” environment variables do not have fixed names such as `TRIFIELD1`. Rather, their names are based on the associated Column Names (identified by the `ColumnNames` directive in the same job stanza). The environment variables are created from these column names as follows:

```
TRI_{COLUMN_NAME}_{SELECTION_NUMBER}
```

The `{COLUMN_NAME}` is based on the associated Column Name from the `ColumnNames` directive, converted to upper case and with spaces converted to underscore characters. The `{SELECTION_NUMBER}` is based on the number of the selection – not the row number within the list box. In other words, the first selected row will be selection number 1, the second row will be selection number 2 and so on.

Suppose, for example, that the user had selected two rows from the list box:

Title	Chart Position	Release Year
Love Me Do	1	1964
Can't Buy Me Love	1	1964
Twist and Shout	2	1964
Help!	1	1965

When the OK button is clicked, the server-side job is run and the following environment variables will be available to it:

\$TRIFIELD1	<i>User Name</i>
\$TRIFIELD2	<i>Password</i>
\$TRIFIELD3	<i>"The Beatles"</i>
\$TRIFIELD4	<i>Song Title (for searching)</i>
\$TRI_TITLE_1	<i>"Love Me Do"</i>
\$TRI_CHART_POSITION_1	<i>"1"</i>
\$TRI_RELEASE_YEAR_1	<i>"1964"</i>
\$TRI_TITLE_2	<i>"Twist and Shout"</i>
\$TRI_CHART_POSITION_2	<i>"2"</i>
\$TRI_RELEASE_YEAR_2	<i>"1964"</i>

5.2.14 Automatically Selecting List Box Rows

You may wish to automatically "pre-select" certain list box rows when the list box is populated (either when the dialog is first displayed, when the Apply button is clicked or when a field is changed). This may be useful if you are displaying a list of entries in a database and wish to highlight which ones are currently "selected").

To do this, *Trilogy* uses the `AutoSelectColumn` and `AutoSelectValue` directives in the relevant job stanza within the server-side `trilogy.conf` file. Like `AutoSort` the `AutoSelectColumn` directive specifies a column number. If this

directive is present, *Trilogy* will look at each row in the list box (whenever the list box is populated) and see if the value for the specified column in that row matches the value given by the `AutoSelectValue` directive. If it does, then the entire row is selected. If it does not, the row is left unselected.

`AutoSelectValue` defaults to "Y". Therefore, if this directive is not specified but `AutoSelectColumn` is set, then *Trilogy* will look for a "Y" in the specified column and automatically select any row in which this occurs.

`AutoSelectValue` can be specified as a distinct value, like this:

```
AutoSelectValue=CRITICAL
```

A number of discreet values, like this:

```
AutoSelectValue={IMPORTANT, URGENT, CRITICAL}
```

or, if the values are numeric, as a range of values:

```
AutoSelectValue=10-100
```

In this case, *Trilogy* will select any row whose column (identified by the `AutoSelectColumn` directive) contains a value that falls in the specified range (inclusive).

As an example, consider the following *Trilogy* Job Definition:

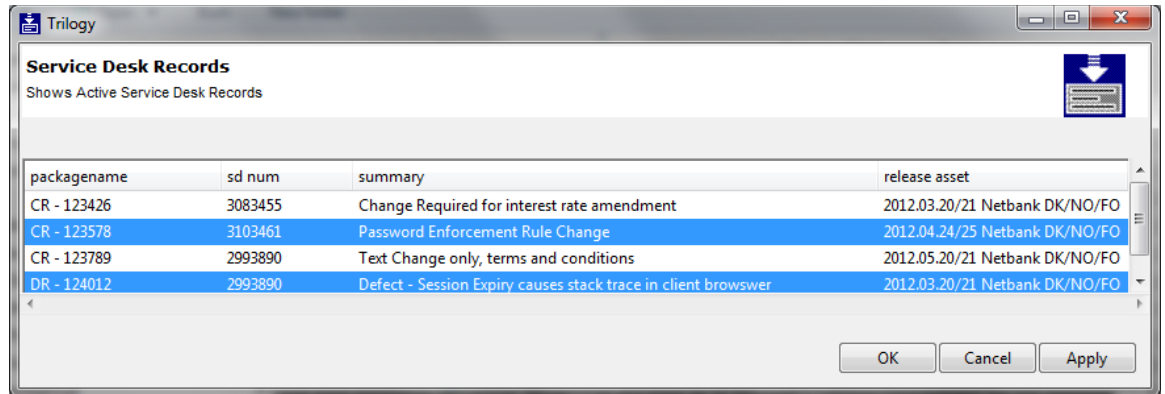
```
SD_INTERFACE:
  Banner=On
  BannerHeading=Service Desk Records
  BannerText=Shows Active Service Desk Records
  Dialog=$TRILOGYHOME/demo/screens/sdc.scn
  ListBox=Auto
  ColumnNames={select,packid,packagename,sd num,summary,release asset}
  ColumnWidths={0,0,150,100,400,190}
  PopulateListBoxWith=$TRILOGYHOME/demo/scripts/plb.bat
  PopulateListBox=OnDisplay
  ListBoxSep=|
  ApplyButton=on
  AutoSelectColumn=1
```

This dialog will display a list of records from a Service Desk system. We wish to select any rows representing records that are assigned to us. To do this, the List Box Script `plb.bat` can look at the environment variable `$TRICLIENTUSERNAME` – this will be set to the login ID of the client user. It can then output a Y as the first field if the record is assigned to that user.

So if the script output this:

```
N|81570|CR - 123426|3083455|Change Required for ...
Y|81618|CR - 123578|3103461|Password Enforcement Rule Change ...
N|81406|CR - 123789|2993890|Text Change only, terms and conditions ...
Y|81432|DR - 124012|2993890|Defect - Session Expiry causes stack trace ...
```

Then the client-side dialog would look like this:

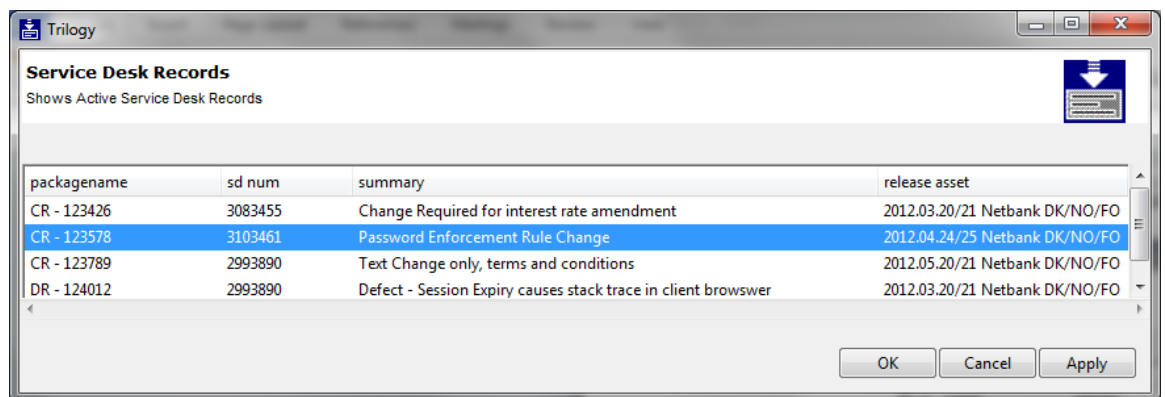


This is because the 2nd and 4th records have a "Y" in their first (hidden) column position. Since the `AutoSelectColumn` is set to "1" and there is no `AutoSelectValue` directive, *Trilogy* will pre-select any row with a "Y" in the first column position.

By changing the value of `AutoSelectColumn` and adding `AutoSelectValue`, we can automatically select other rows. For example, this:

```
AutoSelectColumn=4
AutoSelectValue=3103461
```

Will result in this display:



Since *Trilogy* is now looking in column 4 (sd num) for a value of 3103461.

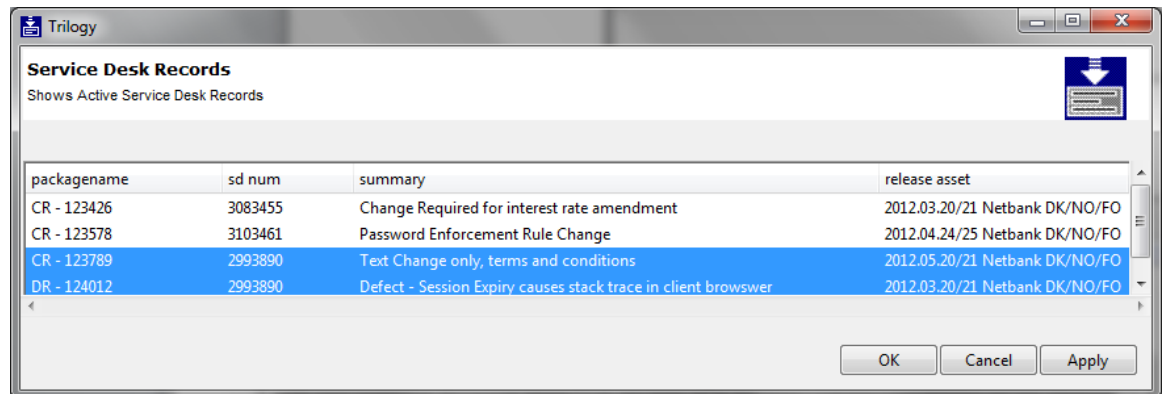


Remember that *Trilogy* counts columns from 1, including any hidden columns. The first two columns from the List Box script are hidden (their column width is 0). So the "sd num" column is 4 even though it is only the 2nd column on the dialog.

Similarly, we can use a range of values to select the row. For example, this:

```
AutoSelectColumn=4
AutoSelectValue=2000000-3000000
```

Will result in this display:



packagename	sd num	summary	release asset
CR - 123426	3083455	Change Required for interest rate amendment	2012.03.20/21 Netbank DK/NO/FO
CR - 123578	3103461	Password Enforcement Rule Change	2012.04.24/25 Netbank DK/NO/FO
CR - 123789	2993890	Text Change only, terms and conditions	2012.05.20/21 Netbank DK/NO/FO
DR - 124012	2993890	Defect - Session Expiry causes stack trace in client browser	2012.03.20/21 Netbank DK/NO/FO

Since *TrilogY* will now select any row whose column 4 (sd num) is between 2000000 and 3000000.

A couple of points worthy of note:



- You can use `AutoSelection` along with `AutoSort` in order to simplify your server scripts. For example, your server script could use one query to list all the open records assigned to the invoking user, followed by another, separate query to list all the open records NOT assigned to the invoking user. By using a hidden field to identify which ones are assigned to the invoking user and setting `AutoSort` to the "sd num" you can list the records in order of their ID whilst selecting the rows appropriate to the invoking user.
- `AutoSelection` works even if the `Selection` option is set to `None`. This means that rows can be pre-selected and the user cannot change the selection – effectively once the list box is displayed, the selection is read-only.

5.2.15 Adding a Right-Click Menu to the List Box

TrilogY also gives you the ability to create a pop-up menu that appears whenever you right-click on a row within the list box. This menu can then invoke other *TrilogY* jobs, allowing you to run additional server-side tasks on list box selections before (or perhaps instead of) simply clicking on the OK button to launch the server side script identified by the `Program` directive.

The directive that controls this is called `OnRightClick`. If right click menu functionality is desired in the list box, it should be included in the relevant job stanza within the server-side `trilogY.conf`. It has the following syntax:

```
OnRightClick={job1,job2...}
```


`job1`, `job2` etc refer to other *Trilogy* Jobs referenced within the same server-side `trilogy.conf` file. When the user right-clicks on a selection within the client-side dialog list box, a menu appears listing the *Titles* of the jobs listed in the `OnRightClick` job list. When the user selects one of these menu options, the selected *Trilogy* job is executed.



The jobs listed are those available to the invoking user's *group*. See *Groups and the Group Processor* later in this document for more information. This means that you can provide different menu options for different individuals dependent on their roles within your organisation.

The "secondary" job is passed all the same command line arguments from the *Trilogy* Client as the "primary" job (the job that contains the list box). In addition, it has access to the content of the "primary" job dialog. Since the "secondary" job could have a dialog of its own, it is obviously not possible to use `TRIFIELD` variables to allow the secondary job to read the primary dialog. Therefore, the Primary Dialog's content is made available to the secondary job via the environment variables `TRIPASSEDFIELDn` where *n* is the number of the field in the Primary Dialog. Selected List Box values are passed in `TRIPASSED_columnname_n`.



- `TRIPASSED` variables are only set if the secondary job has a dialog associated with it. If it does not, then the variables from the dialog are accessible in `TRIFIELDn` variables in the normal way.
 - Any `PreValidate=` script associated with the secondary job is executed first before any dialog is displayed or the program is executed. You can use this to validate the selection(s) made in the list box.
-

The secondary job can run to completion without the primary dialog being cleared from the client screen. Therefore, once the secondary job has completed, a user can right click again and run another secondary job.

The List Box content is refreshed automatically after the secondary job has run to completion. This means you can have a secondary job which modifies the content of the list box in some way (for example, removing or adding an entry to what is displayed). After the secondary job completes, the List Box script is re-run and the new content displayed.



You can make *Trilogy* display a different right-click menu depending on the data in the row you have clicked on. This is done by specifying a *context* and having different right click menus for each context. See *Row Context* below for more information.

5.2.16 Automatically Refreshing the List Box

You may wish the List Box to periodically refresh itself. This may be useful if the list box contains dynamic information such as a list of running processes, tasks awaiting a user's attention or other "dynamic" type content. In this case, you may wish the list box to automatically refresh in order to capture the latest output.

This is achieved by use of the `AutoRefresh` option. When specified in the job stanza, this tells *Trilogy* to periodically repopulate the list box. It does this by rerunning the server-side script specified by the `ListBoxScript` (or `PopulateListBoxWith`) directive, reparsing the output and re-populating the list box. This happens automatically and without any client user intervention

`AutoRefresh` is only available if `PopulateListBox` is set to `OnDisplay`. If `PopulateListBox` is set to `OnApply`, then `AutoRefresh` has no effect – the user still has to click on the Apply button to refresh the list box.

`AutoRefresh` has the following syntax:

```
AutoRefresh=on|off|n
```

If set to `off` (the default) then no automatic refresh takes place.

If set to `on` then the List Box will automatically refresh every 5 seconds.

If set to `n` (where `n` is a number) then the List Box will automatically refresh every `n` seconds.

The sort column order and any selected rows are retained after a refresh.



Although *Trilogy* will retain any selected rows in the list box, it is worth bearing in mind that it does this by row number. This means that if the server-side script creates a new row and this appears before a selected row, then the selection may change.

Be cautious about the `AutoRefresh` setting. Every time a refresh is performed, *Trilogy* Client will request that *Trilogy* Server rerun the List Box Script. This can create a significant load on the server if the job is allowed to be run from multiple clients.

5.2.17 Adding Double Click to the List Box

You can also have the list box regenerate whenever a row in the list box is double-clicked. This allows you to "drill-down" into entries, for example to open a folder to show its content.

This function is controlled by the `DoubleClick` directive. This defaults to `off`. To enable Double-Click in the list box, use the directive `DoubleClick=on` in the job stanza.

When `DoubleClick` is set to `On`, a user can double-click on a row in the list box and the server side list box script is executed and the output used to regenerate the list box. The list box script can use the TRI environment variables to establish which row has been selected and use it to regenerate the list box content.

5.2.18 Adding Icons to Each Row

You may wish to add a decorator icon to each row in the list box. This can be used to indicate what the data in each row represents.

To do this, you first need to tell *Trilogy* to allow space for the icon. To do this, use the `ListBoxIcons` directive. This defaults to "No" (or "off"). To turn on icons for each row in the list box, set this directive to "yes" (or "on") within the job stanza. Like this:

```
ListBoxIcons=Yes
```

This will then cause *Trilogy* to allow space for the icon to the left of each row.

To identify which icon to display use the `icon` directive. This should be set to the full path of a GIF image file on the server. This GIF image should be a 16 pixel x 16 pixel icon.

For example:

```
Icon=$TRILOGYHOME/icons/16x16/folder.gif
```

This will then cause the specified GIF image to appear next to each row in the List Box at the client.



You can make *Trilogy* display different icons which change depending on the data in each row. This is done by specifying a *context* and having different icons for each context. See *Row Context* below for more information.

5.2.19 Row Context

Each row displayed in a List Box can represent different types of data. For example, a directory listing could contain both folders/directories and files. You may wish to identify each row with a different icon (so that the user has a visual indicator of the file type) and the right-click menu may need to differ depending on whether the user has selected a row containing a directory or a file.

Trilogy supports this by use of a *Context*. Each row in the list box has its own context and that context is used to identify which icon should be displayed (if the row has an icon) and/or which right-click menu is appropriate for that row.

Contexts are identified by specifying a *Column Number* which contains the character string that will be used to identify the different contexts. This Column Number is identified using the `ContextColumn` directive in the server-side job stanza.

For example, this:

```
ContextColumn=2
```

Tells *Trilogy* that the second column in the list box contains the character string which identifies the context.



The Context Column can, of course, be a hidden field (just as with `AutoSelect`). The column should only ever contain alpha-numeric string data which is then used to name the context.

When `ContextColumn` is set, *Trilogy* will examine the specified column for each row of the List Box. The *context* for the row is then set to the string contained within the column for that row.

Both the `Icon=` and `OnRightClick=` directives can be modified to be context-specific by specifying the name of the context within the directive, like this:

```
IconContext=<path to gif file>  
OnRightClickContext={JOB1,JOB2...}
```

Here, **Context** is the name of the context. You can specify any number of `Icon` and `OnRightClick` contexts within a job stanza.

For example, suppose your server-side script generates a directory listing. Your script generates the following output:

```
DIR,My Documents,  
DIR,My Music,  
FILE,myfile.txt,1234
```

To display a different icon for each row (and change the right-click menu according to what is clicked on), we specify the following in our job stanza:

```
FILELIST:  
.  
.  
ColumnNames={Type,Name,Size}  
ColumnWidths={0,200,100}  
ContextColumn=1  
OnRightClickFILE={DOWNLOAD,DISPLAY}  
IconDIR=$TRILOGYHOME/icons/16x16/folder.gif  
IconFILE=$TRILOGYHOME/icons/16x16/file.gif  
.  
.
```

Note that the `OnRightClick` directive has been modified to include the context in which it operates (`OnRightClickFILE`). Only rows where the word `FILE` appears in the first column position (as identified by the `ContextColumn` directive) will see this menu. This means that a user selecting a folder and right-clicking will not see the `DOWNLOAD` and `DISPLAY` menu options. This menu is only available when selecting a row with `FILE` in the first (hidden) column.

Similarly, there are now two `Icon` directives – each of which contains the context to which it applies (`IconDIR` and `IconFILE`). A row with `DIR` in the first column position (as identified by the `ContextColumn` directive) will display the “folder.gif” icon, a row with `FILE` in the first column position will display the “file.gif” icon.



Context Names are not case-sensitive. `IconFILE=` and `IconFile=` would both be used if the Context Column contained `File`, `FILE` or `File`.

Trilogy will default to using the non-context specific `Icon` or `OnRightClick` menu should a context-specific version not exist for that row. For example, supposing a row contained the text `UNKNOWN` in the first column. In this instance, *Trilogy* would not know which context to apply since there are no `IconUNKNOWN` or `OnRightClickUNKNOWN` directives present in the job stanza. In such cases, *Trilogy* will use the `Icon=` or `OnRightClick=` directives if they are present. These non-context specific directives are therefore the default.

If these default directives are not present (as in the example above) then no default is used – no icon is displayed for rows where the context cannot be matched to an icon and no right-click menu is presented.

For example, our job definition could look like this:

```
FILELIST:
.
.
ColumnNames={Type,Name,Size}
ColumnWidths={0,200,100}
ContextColumn=1
OnRightClick={DOWNLOAD,DISPLAY}
OnRightClickDIR=
Icon=$TRILOGYHOME/icons/16x16/file.gif
IconDIR=$TRILOGYHOME/icons/16x16/folder.gif
.
.
```

Here, the “default” icon is `file.gif`. This will be displayed for any row which does not have `DIR` in the first column. Only rows which have `DIR` in their first column will display the `folder.gif` icon. Similarly, the default right-click menu option contains `DOWNLOAD` and `DISPLAY` jobs. However, right-clicking on a row with the `DIR` context shows no menu (`OnRightClickDIR=` is empty).

5.2.20 Controlling List Box Script Execution

The server-side List Box Script can be executed in one of six scenarios:

- When the *Trilogy* Client Dialog is first displayed (default)
- When the “Apply” button is clicked
- When a field changes in the dialog (see *Chapter 6 – Linking Fields*).
- After a right-click job has completed execution
- When a row is double-clicked in the list box.
- Automatically on a timed basis.

You may find it necessary for the List Box Script to behave in different ways depending on the reason for the refresh. Whenever the list box script is run, the environment variable `TRIREASON` is set to “LISTBOX”. In order to determine why the List Box is being refreshed, the list box script should examine the environment variable `TRILISTBOXREASON`. This will allow the list box script to determine under what circumstances it is being run:

TRILISTBOXREASON	Set when...
APPLY	Apply button has been clicked.
FIELDCHANGED	A field has been changed that is linked to the listbox. Used as a result of an <code>OnFieldChangeUpdate={LB}</code> clause (see <i>Chapter 6 - Linking Fields</i> below for more information).
REFRESH	Listbox is being refreshed automatically (AutoRefresh)
POSTRC	Listbox is being run following the execution of a right-click job.
DBLCLICK	User has double-clicked on an entry in the list-box.

The directive `PopulateListBox` controls the “Apply” button action. It is included in the job stanza in the server-side `trilogy.conf`. If it is set to `OnDisplay` (the default) then the list box script is executed on the server whenever the client side dialog is first opened and the resulting output is used to populate the client-side list box. If it set to `OnApply` then the list box is displayed with the columns and widths populated but with no data. Only when the Apply button is clicked at the client is the server side List Box Script executed and its standard output used to populate the client dialog as described above.

Setting `PopulateListBox=OnApply` will add an “Apply” button to the client-side dialog automatically. It can be added manually (and the text on the button can be changed) as described below.

You can also link the list box to a field so that when the field changes, the list box script is rerun and the list box regenerated.



Linking Fields is covered in *Chapter 6: Linking Fields* later in this document. Populating the List Box when a field changes works independently of the setting of `PopulateListBox`.

5.3 Example – Building a List Box Application

In this section we will bring all of these components together to build a simple remote process list application – this will list all the running processes on a Unix Server and allow us to terminate or suspend any job we select from the list box.

First, we need to define our client-side dialog. We need to be able to filter jobs based on their name (perhaps we're using this tool to allow us to look for jobs that regularly cause us trouble or perhaps to monitor their usage). So we'll need a data entry field for that. That's all we need, so the dialog definition file is pretty simple:

```
- Process Details
Process Name [                ]
```

We create this file with a text editor and save it in `$TRILOGYHOME/screens/ProcessDetails.scn`

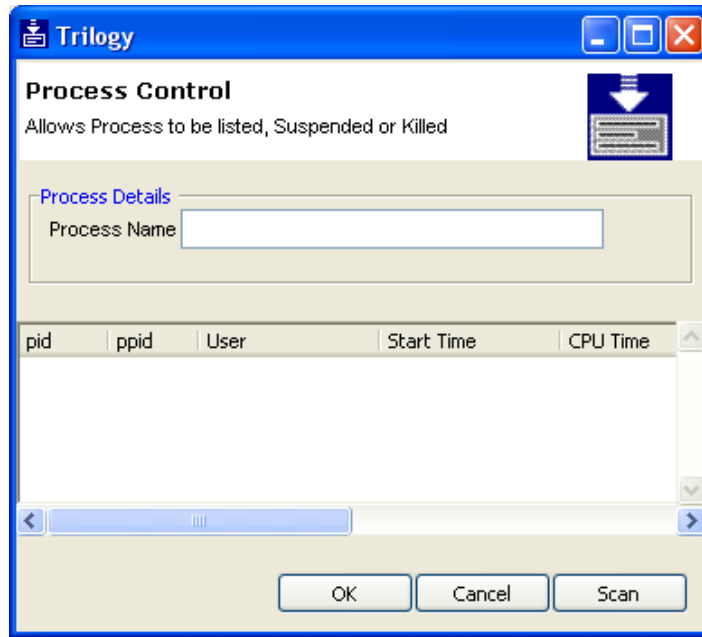
Now we create our server-side `trilogy.conf` entries to set up the job and create the dialog. Amending `trilogy.conf` on our Unix Server, we add the following entries:

```
PROCESS_CONTROL:
  Dialog=$TRILOGYHOME/screens/ProcessDetails.scn
  Banner=on
  BannerHeading=Process Control
  BannerText=Allows Process to be listed, Suspended or Killed
  ListBox=Yes
  PopulateListBox=OnApply
  ApplyButton=Yes
  ApplyButtonText=Scan
  ColumnNames={pid,ppid,User,Start Time,CPU Time,Process Name}
  ColumnWidths={50,50,100,100,100,400}
  AutoStretch=No
```

Save the server-side `trilogy.conf` file with these additional entries. Moving to our client machine we invoke *trilogy*:

```
trilogy process_control
```

The following dialog is displayed:



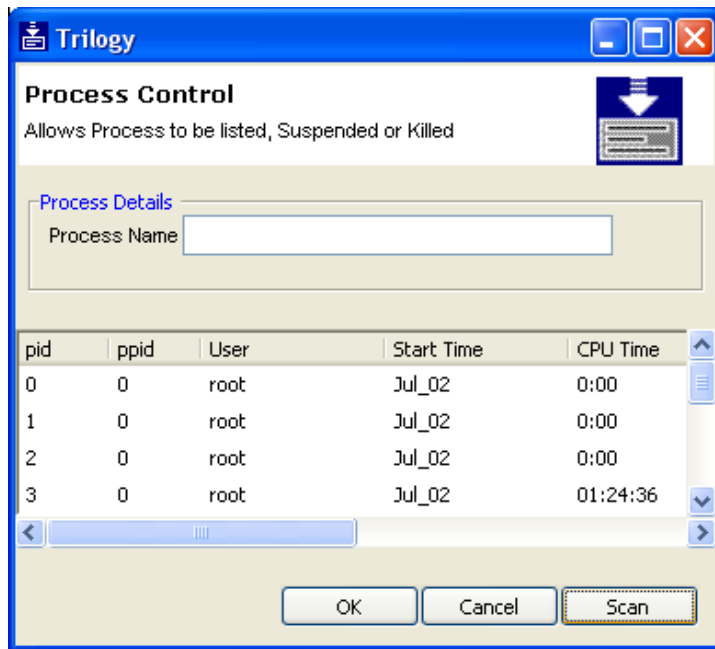
So far, this looks fine. Let's add the List Box Script. This script needs to perform a "ps" listing, filtering the output based on jobs containing the text specified in our "Process Name" field (\$TRIFIELD1). Here's the script:

```
#!/bin/ksh
[[ -z "$TRIFIELD1" ]] && TRIFIELD1="^.*$"
ps -eo "pid ppid user stime time comm" | nawk -v p="$TRIFIELD1" '
{
    if (NR>1 && $NF ~ p)
    {
        print $1,"$2","$3","$4","$5","$6
    }
}'
```

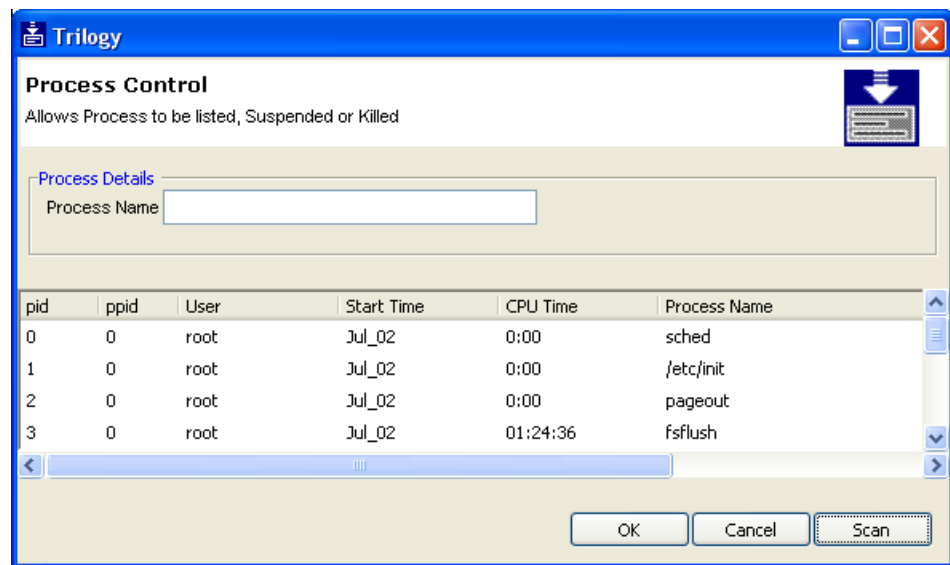
This script is for Sun Solaris – it will differ under other Unix variants. Save this file to \$TRILOGYHOME/scripts/plist on the server. Now add the ListBoxScript directive to our server-side trilogy.conf:

```
ListBoxScript=$TRILOGYHOME/scripts/plist
```

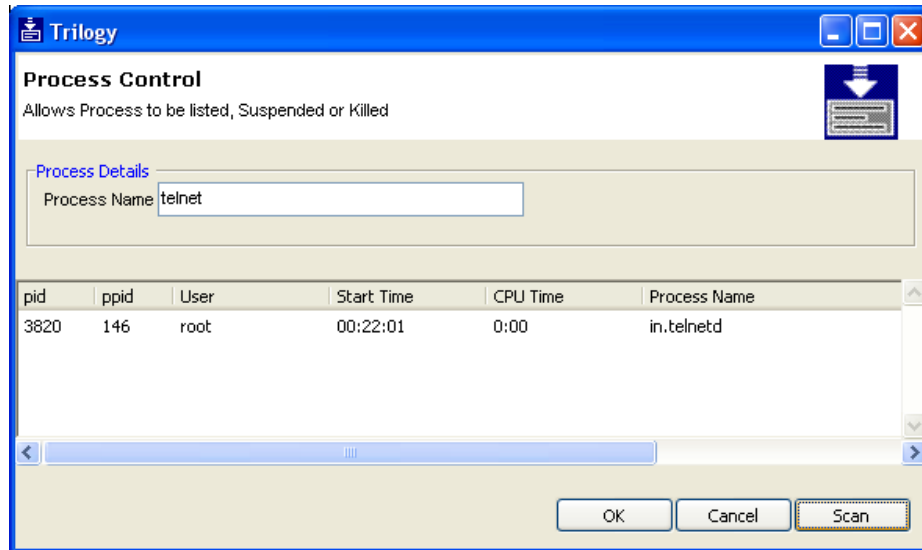
Now on the client, we can invoke *Trilogy* again to bring up the dialog. This time when we click "Scan" (which is the Apply button) the server-side "plist" script will run. If nothing is entered in the Process Name field, then \$TRIFIELD1 will be blank and the script will list all the processes on the system.



Since AutoStretch was set to "No", we need to drag the dialog to expand the list box:



Entering some text into "Process Name" and clicking Scan again results in a filtered list:



This looks like what we need. However, there is still no mechanism to allow us to suspend or terminate jobs. This is where the `OnRightClick` directive comes in.

We amend the server-side `trilogY.conf` so that it now looks like this:

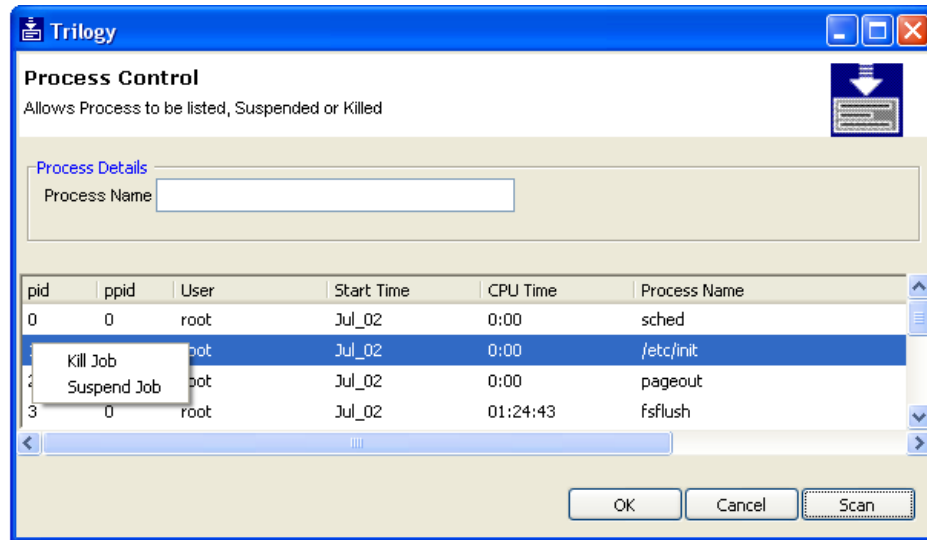
```
PROCESS_CONTROL:
    Dialog=$TRILOGYHOME/screens/ProcessDetails.scn
    Banner=on
    BannerHeading=Process Control
    BannerText=Allows Process to be listed, Suspended or Killed
    ListBox=Yes
    PopulateListBox=OnApply
    ApplyButton=Yes
    ApplyButtonText=Scan
    ColumnNames={pid,ppid,User,Start Time,CPU Time,Process Name}
    ColumnWidths={50,50,100,100,100,400}
    ListBoxScript=$TRILOGYHOME/scripts/plist
    OnRightClick={KILL_JOB,SUSPEND_JOB}

KILL_JOB:
    Title=Kill Job
    Program=/usr/bin/kill
    Param=-9
    Param=$TRI_PID_1

SUSPEND_JOB:
    Title=Suspend Job
    Program=/usr/bin/kill
    Param=-STOP
    Param=$TRI_PID_1
```

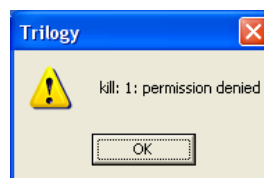
Note the only addition to our original `PROCESS_CONTROL` job is the `OnRightClick` directive. This points us at two additional *TrilogY* jobs – `KILL_JOB` and `SUSPEND_JOB`. The titles of these jobs (Kill Job and Suspend Job) will now be listed in a pop-up menu at the client when we right-click on a selection in the list box.

Opening the dialog on our client, we now see that right-clicking on a selection in the list-box brings up the menu:



Selecting "Kill Job" on `/etc/init` will run the Trilogy Job "KILL_JOB" on the server. This will run the server-side program `/usr/bin/kill`, passing it two parameters (defined by the `Param=` directives). The first parameter (`-9`) is the signal number to pass to the process – 9 equating to SIGKILL which cannot be caught by the process and will kill it). The second parameter to `/usr/bin/kill` is the process id. This is passed from the primary dialog because it is in the environment variable `$TRI_PID_1`. This is because the first column is called "pid" so *Trilogy* constructed an environment variable called `$TRI_PID_1` and set it to the first selected PID column value.

Trilogy will therefore issue the `/usr/bin/kill -9` command to the process id specified in the PID column. By default, the standard error stream of an invoked process is routed to a pop-up dialog box. So if we try to do that on an operating system process like `/etc/init`, this is what we get:



If we try this on a process to which we do have control, the process will be killed.

6 Linking Fields

6.1 Introduction to Linked Fields

Sometimes there is a requirement to have a field change dynamically whenever another field on the dialog is changed. For example, fields can be disabled or enabled dependent on the setting of a checkbox or radio button; a drop-down list can be populated with different content based on the selection in another drop-down list; a read-only data entry field can be populated dynamically as a user types into another data entry field and so on.

6.2 Creating Linked Fields.

Linked Fields are defined in *Trilogy* with the use of a directive entry in the server-side `trilogy.conf` file. This entry is placed in the desired job stanza and specifies which fields should be updated whenever a specified field is changed on the client dialog.

The directive is called `OnFieldChangeUpdate` and its syntax is as follows:

```
OnFieldChangeUpdate={fieldno,fieldno...}
```

Where *n* is the field which needs to change to trigger the update and `fieldno` is *either* the number of a field to be updated *or* is the word LB or LISTBOX to update any attached list box present on the dialog.

A field is updated by *Trilogy* automatically running the appropriate `PopulateFieldnWith` script in the same job stanza whenever the "source" field changes. Similarly, the list box is updated by *Trilogy* automatically running the `ListBoxScript` (or `PopulateListBoxWith`) whenever the "source" field changes.

6.3 Linking Drop Down Lists

As an example, consider the following server-side `trilogy.conf` entries for the *Trilogy* Job "SHOW_CARS":

```
SHOW_CARS:
Dialog=$TRILOGYHOME/screens/cars.scn
PopulateField1With=$TRILOGYHOME/scripts/show_manufacturers.sh
PopulateField2With=$TRILOGYHOME/scripts/show_cars.sh
PopulateField3With=$TRILOGYHOME/scripts/show_engines.sh
OnFieldChange1Update={2,3}
OnFieldChange2Update={3}
```

Here is the dialog "cars.scn" defined on the server in `$TRILOGYHOME/screens/cars.scn`:

```
- Vehicle Choice
Manufacturer {
Model Range {
Engine Size {
```

And here is how it is displayed at the client:

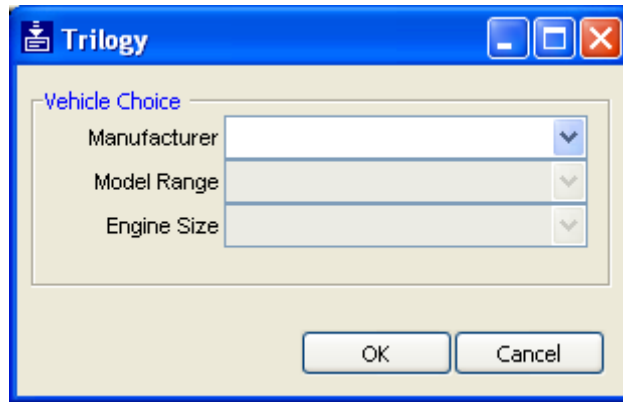


Figure 5.1: Initially Populated Dialog

When the dialog is first displayed at the client, the `show_manufacturers.sh` script is run on the server, producing a list of vehicle manufacturers to its standard output. This is then used to populate the client-side drop down in field 1.

The `show_cars.sh` and `show_engines.sh` scripts will also run. However, they should produce no output to leave the corresponding drop-down lists on the client dialog blank. Indeed, it may be best if they exited with a fail condition (exit code non-zero) so that the *Trilogy* Client will disable the field as shown above. To do this, they can either determine the value of `$TRIFIELD1` (which will be NULL when the dialog is first displayed) or they can examine the environment variable `$TRICHANGEDFIELD` which will be null on initial dialog creation.

The client now sees a dialog with a drop-down list of vehicle manufacturers and nothing in the Model Range (`TRIFIELD2`) and Engine Size (`TRIFIELD3`) drop down lists. However, when the user selects a manufacturer from the drop down list on the client dialog, *Trilogy* Server will automatically run `show_cars.sh` and `show_engines.sh` again. This is because `OnFieldChange1Update` is set to 2, 3. The field update script for field 2 is `show_cars.sh` and the field update script for field 3 is `show_engines.sh`. This time, however, `$TRIFIELD1` will be set to whichever manufacturer was selected by the client user, and `$TRICHANGEDFIELD` will be set to "1". The `show_cars.sh` script can then produce a list of vehicles built by the specified manufacturer. Sending this list to its standard output will result in the client dialog being updated and field 2 (the "Model Range" drop-down list) being populated with the appropriate list of vehicles.

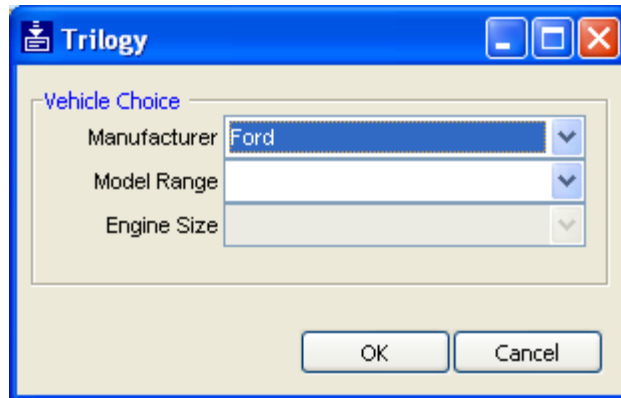


Figure 5.2: Selecting a manufacturer populates fields 2 & 3.

The `show_engines.sh` script is designed to show the engines available for a particular car. Since the user has not selected a car from field 2 yet (the list is populated now but no selection has been made), then `$TRIFIELD2` is still NULL. Given that `$TRICHANGEDFIELD` is "1" (the manufacturer selection has changed) then `show_engines.sh` can simply return nothing and exit with a non-zero exit code which will result in the "Engine" Size drop down field on the client dialog remaining blank and disabled (as shown).

Now the user selects a car from the field 2 drop-down list. As soon as the selection is made in the client dialog, *Trilogy* Server will automatically run `show_engines.sh` again. This is because field 2 has changed and `OnFieldChange2Update` is set to 3. The field update script for field 3 is `show_engines.sh`. This time, `$TRIFIELD1` is the name of the selected manufacturer and `$TRIFIELD2` is the name of the selected vehicle. `$TRICHANGEDFIELD` is 2. This will allow the server-side script `show_engines.sh` to send a list of engines available for that particular vehicle to its standard output which – in turn – will populate the "Engine Size" drop down in the client dialog.

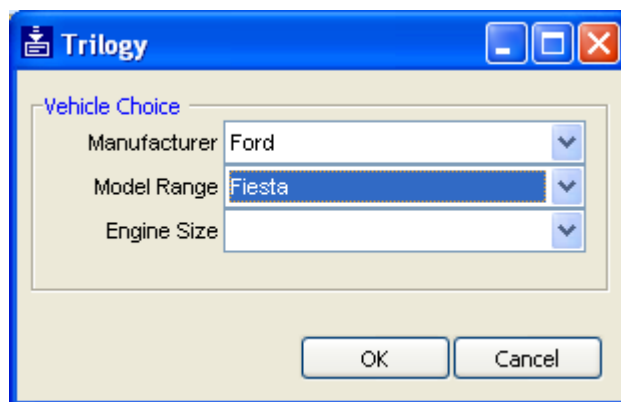


Figure 5.3a – Selecting a model updates the Engine Size field....

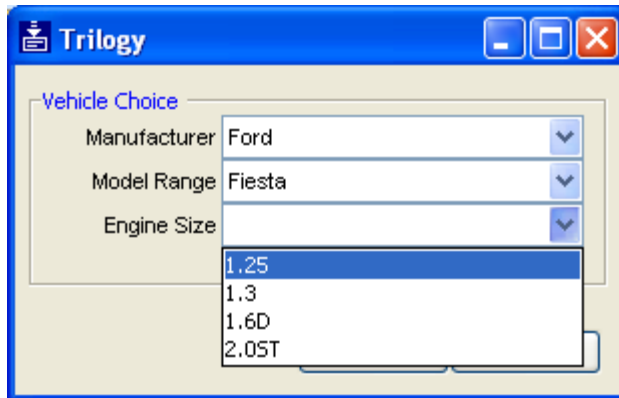


Figure 5.3b - ...with the engines available for the chosen model range.

If the manufacturer selection changes, the drop-down list in field 2 will change to show a new list of vehicles and the drop-down list in field 3 will be cleared. In this way, the drop-down lists in the dialog dynamically amend their content based on the selections made by the user.

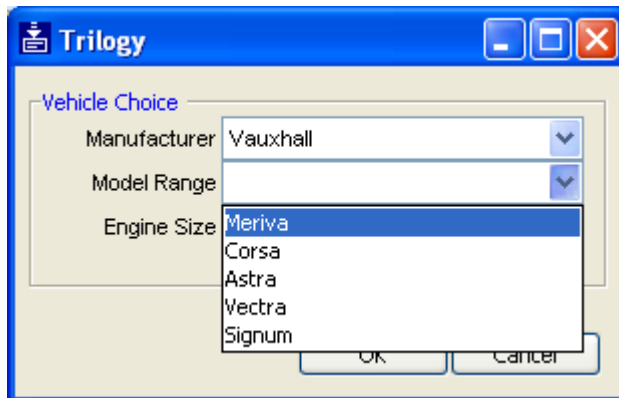


Figure 5.4 – changing the Manufacturer re-populates the model range drop-down list.

6.4 Linking radio buttons and checkboxes

The same technique can also be used whenever a checkbox or radio button is clicked. You can use this to disable certain fields on the dialog if a checkbox or radio button is not selected. Remember that the `TRIFIELD` environment variable corresponding to the radio button or checkbox is "1" if the item is selected and "0" otherwise.

Consider the following example. We have created a dialog in which only one of two data entry fields can be considered valid. To switch between the two we create a pair of radio buttons. Selecting one radio button enables one data-entry field and disables the other. The dialog will look like this:

```
Enter Some Data
O Enter This      [
o Enter That      [
```

`TRIFIELD1` is the "Enter This" radio button, `TRIFIELD2` is the data entry field on the same line, `TRIFIELD3` is the "Enter That" radio button and `TRIFIELD4` is *its* data entry field.

If you wished to disable the data entry field for the non-selected entry field you would set up something like this in the server-side `trilogy.conf`:

```
ENTER_DATA:
  Dialog=$TRILOGYHOME/screens/EnterSomeData.txt
  OnFieldChange1Update={2,4}
  OnFieldChange3Update={2,4}
  PopulateField2With=$TRILOGYHOME/scripts/EnterThis.sh
  PopulateField4With=$TRILOGYHOME/scripts/EnterThat.sh
```

Here is what `EnterThis.sh` would look on a Unix Server:

```
#!/bin/ksh
case "$TRICHANGEDFIELD" in
  1)      exit 0;;
  3)      exit 1;;
  *)      exit 0;;
esac
```

And here is `EnterThat.sh`:

```
#!/bin/ksh
case "$TRICHANGEDFIELD" in
  1)      exit 1;;
  3)      exit 0;;
  *)      exit 1;;
esac
```

Here's what the dialog will look like when it is first displayed:

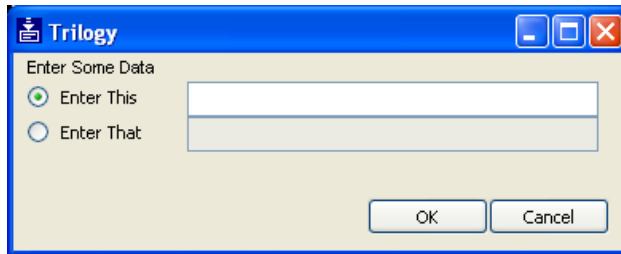


Figure 5.5 – Initial display. *EnterThat.sh* exits with failure and disables the field.

When the dialog is first displayed, *Trilogy* will run the server side scripts *EnterThis.sh* and *EnterThat.sh* – these scripts can determine if they are being invoked during an initial population by checking the value of the environment variable `$TRICHANGEDFIELD` – if it is NULL then they are being invoked during initial population and *EnterThat.sh* can exit with a failure and *EnterThis.sh* can exit with success. This results in the initial dialog display shown in figure 5.5 above.

Now, whenever one of the radio buttons is clicked, the *EnterThis.sh* and *EnterThat.sh* scripts are run again. They can then use `$TRICHANGEDFIELD` to determine which radio button has been clicked and can exit with a success or failure code accordingly. For example, if `$TRICHANGEDFIELD` is 1 then "Enter This" has been clicked and *EnterThis.sh* can return 0 (success) and *EnterThat.sh* can return 1 (failure). This will result in the "Enter This" data entry field being available at the client, whilst the "Enter That" data entry field is disabled. Similarly, if `$TRICHANGEDFIELD` is 3 then that means "Enter That" has been clicked and *EnterThis.sh* can return 1 (failure) and *EnterThat.sh* can return 0 (success). This results in "Enter This" data entry field being disabled and the "Enter That" data entry field being enabled.

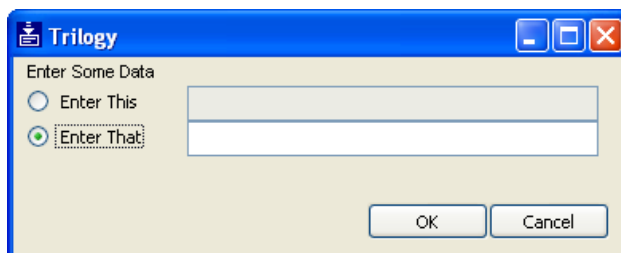


Figure 5.6 – Clicking on the second radio button disables the first data entry field and enables the second.

6.5 Linking a Field to the List Box

You can link the list box (if displayed) to a field such that when the field changes, the list box is automatically regenerated.

To do this, specify either LB or LISTBOX as the “field” to change in a `OnFieldChangeUpdate=` directive.

For example, to update the list box whenever a drop-down field changes its value, use:

```
.  
.PopulateField1With={Value1,Value2,Value3}  
OnFieldChangeUpdate={2,LB}  
PopulateListBoxWith=$TRILOGYHOME/scripts/listbox.vbs  
PopulateField2With=$TRILOGYHOME/scripts/field2.bat  
ListBox=Yes  
. .
```

These directives ensure that whenever the value of the first field (a drop-down containing “Value1”, “Value2” and “Value3”) changes, *Trilogy* will run `$TRILOGYHOME/scripts/listbox.vbs` to repopulate the list box and `$TRILOGYHOME/scripts/field2.bat` to repopulate the drop-down for field 2.

6.6 Linking List Box to Fields

You can also link the list box (if displayed) to other fields such that whenever the list box content or selection changes, the specified fields are automatically refreshed.

To do this, use the `OnListBoxChangeUpdate=` directive. This is similar to the `OnFieldChangeUpdate=` directive in that the value is a set of fields that should be updated whenever the list box content changes (or a selection is made or changed within the list box).

For example, this directive:

```
OnListBoxChangeUpdate={2,3}
```

Specifies that fields 2 and 3 should be re-populated whenever the List Box content changes.

The list box content changes when:

- It is manually refreshed by clicking the apply button
- It is refreshed automatically (`AutoRefresh` is set)
- The selection inside the list box is changed by the user selecting rows.

6.7 Linking Data Entry Fields

You can also set up a link between different data entry fields. For example, you might wish to populate a read-only field with a string whose value changes as the user types into another data entry field.

This is very simple to set up in *Trilogy*. Let's say that we had a "Create New Project" dialog. This allows us to type in a free-text project name which will be converted to a fixed-format project name as we type. Let's say we need to prefix the name of the project with "PREFIX_" and convert any lower-case letters to upper-case and any spaces to underscores. Thus, "My Project" would become "PREFIX_MY_PROJECT".

First, let's create an entry dialog. This will have two fields – the field we're going to type into and a second (disabled) field that will display the project name to be created. The dialog looks like this:

```
- Project Details
Project Name      [
Created Project   [
```

Now, we create a server-side script to populate the second field. The script is quite simple – it simply takes the current value of "Project Name" (`$TRIFIELD1`) and converts it into our desired format:

```
#!/bin/ksh
newname=$(echo "$TRIFIELD1" | tr ' [a-z] ' '[A-Z]_')
echo "PREFIX_${newname}"
exit 1
```

Note the "exit 1" at the end of the script. This failure indication is used by *Trilogy* to disable the field at the client, thereby preventing the user from amending its content directly.

We set up the server-side `trilogy.conf` to create our job reference. Here are the entries:

```
CREATE_PROJECT:
    Dialog=$TRILOGYHOME/demo/screens/testlink.scn
    PopulateField2With=$TRILOGYHOME/demo/scripts/echoname.sh
    OnFieldChange1Update={2}
```

If the user invokes the dialog at our client, this is what they see:

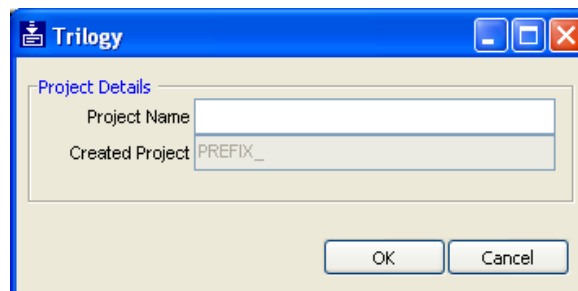


Figure 5.7 – Initial Display with dynamic field update.

Now, every time they type a character into the "Project Name" field, *Trilogy* will trigger the server-side job `echoname.sh`. This will echo the current value of "Project Name" (`$TRIFIELD1`), convert it according to our rules (lower case to upper case, spaces to underscores) and echo it out to the standard output. *Trilogy* will then use that output to update `TRIFIELD2` (Created Project). Since this happens for every typed character, the second field on the dialog updates dynamically as the user types:

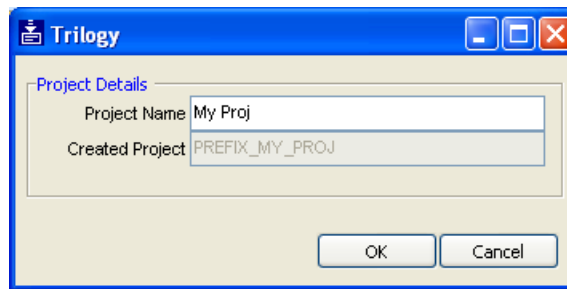


Figure 5.8 – Second entry field is updated as the user types into the first field.

6.8 Linking a Field to Itself

Linking a field to itself can be useful if you wish to restrict the characters that can be entered into a data-entry field. This is a very similar technique to that outlined above for linking two data entry fields. In this case, however, we will echo the converted string back to the updating field.

For example, let's assume we have a field that is only supposed to contain numeric characters. We could test for this when the dialog is submitted (with the `ValidateWith` clause) but it would be much better if we could prevent non-numeric characters being entered in the first place.

To do this, we create a simple server-side script to filter out all non-numeric characters:

```
#!/bin/ksh
echo "$TRIFIELD1" | sed 's/[^0-9]//g'
exit 0
```

Then we set up the server-side `trilogy.conf` to trigger this script whenever the content of the data entry field changes:

```
TESTLINK:
    Dialog=$TRILOGYHOME/demo/screens/testlink.scn
    PopulateField1With=$TRILOGYHOME/demo/scripts/echoname2.sh
    OnFieldChange1Update={1}
```

The `OnFieldChange1Update={1}` tells *Trilogy* that whenever field 1 changes, it should be refreshed with the output from `echoname2.sh`. That takes the old value of `TRIFIELD1` and strips all non-numeric characters from it. It is therefore impossible for the user to enter any non-numeric characters into the field:

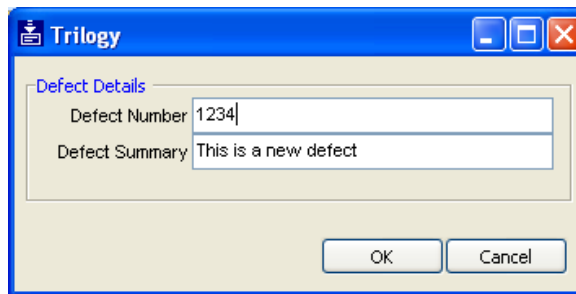


Figure 5.9 – It is impossible to enter non-numeric characters in the "Defect Number" field.

6.9 Caveats

There are a couple of issues to bear in mind when designing *Trilogy* Dialogs with linked fields.

- On initial population, no `TRIFIELD` environment variables are set – even if a `PopulateWith` script has run to pre-populate the dialog content. A script called to populate a field cannot use `TRIFIELD` variables to read the dialog content unless it has been triggered due a field change (`$TRICHANGEDFIELD` not null).
- Linking fields to radio buttons only works when the chosen radio button is selected. For example consider the following dialog definition we used earlier:

```
Enter Some Data
O Enter This      [
o Enter That      [
```

`TRIFIELD1` is the "Enter This" radio button, `TRIFIELD2` is the data entry field on the same line, `TRIFIELD3` is the "Enter That" radio button and `TRIFIELD4` is *its* data entry field.

If you wished to disable the data entry field for the non-selected entry field you might set up something like this in the server-side `trilogy.conf`:

```
ENTER_DATA:
    Dialog=$TRILOGYHOME/screens/EnterSomeData.txt
    OnFieldChange1Update={2,4}
    UpdateField2With=$TRILOGYHOME/scripts/EnterThis.sh
    UpdateField4With=$TRILOGYHOME/scripts/EnterThat.sh
```

This looks intuitively correct, when the radio button for "Enter That" is clicked, Field 1 (Radio button for "Enter This") will clear. That's a change and therefore fields 2 and 4 should be updated. Unfortunately, *Trilogy* does not work that way. Only the field that is *clicked on* will trigger the running of the field update scripts. You would therefore need to have two

`OnFieldChange``nUpdate` entries, one for each radio button as shown in the example earlier.

7 Linking Jobs

7.1 Controlling Access

A *Trilogy* Job can have access controls placed on it so that it can only be invoked:

- By users in specified *user groups*
- If other specified *Trilogy* Jobs are currently running
- If other specified *Trilogy* jobs are *not* currently running



User Groups and the ability to control access to *Trilogy* Jobs via these groups is discussed in the next chapter.

Creating conditions on jobs such as they cannot be run unless other named jobs are running (or cannot be run if other named jobs are running) is known as *Linking Jobs*.

7.2 Overview of Job Linking

Linking Jobs allows you to either:

- Place controls on a *Trilogy* job such that it *cannot* be invoked if another named job is currently running.
- Place controls on a *Trilogy* job such that it can *only* be invoked if another named job is currently running.

Suppose you wish to prevent jobs from being run if another *Trilogy* Job is currently running. As an example, suppose you have two build processes. You would set up your server-side `trilogy.conf` file like this:

```
BUILD_APP_1:
  Dialog=$TRILOGYHOME/screens/build1.scn
  Program=$TRILOGYHOME/scripts/build1.bat

BUILD_APP_2:
  Dialog=$TRILOGYHOME/screens/build2.scn
  Program=$TRILOGYHOME/scripts/build2.bat
```

Any user from any client can now invoke `BUILD_APP_1` or `BUILD_APP_2`.

However, suppose that each build job uses the same resources. In this case, each job is mutually exclusive – when `build1.bat` is running, one cannot run `build2.bat` and vice-versa.

To enforce this restriction, you will need to link the jobs.

7.3 Preventing Jobs from Running

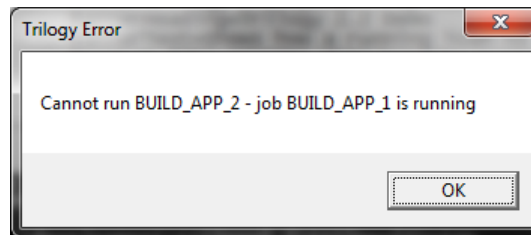
To prevent *Trilogy* jobs from being run when they may clash with other jobs, you need to link the jobs such that each job is mutually exclusive. To do this, *Trilogy* supports the `DenyIfJobRunning=` directive. This allows you to name the jobs (or job) that, if running, prevents the associated job from being invoked.

Taking the example above, by placing this directive into the appropriate job stanza, you can prevent `BUILD_APP_2` from being invoked if `BUILD_APP_1` is currently running (and vice versa):

```
BUILD_APP_1:
  Dialog=$TRILOGYHOME/screens/build1.scn
  Program=$TRILOGYHOME/scripts/build1.bat
  DenyIfJobRunning=BUILD_APP_2

BUILD_APP_2:
  Dialog=$TRILOGYHOME/screens/build2.scn
  Program=$TRILOGYHOME/scripts/build2.bat
  DenyIfJobRunning=BUILD_APP_1
```

Now, any attempt to invoke `BUILD_APP_1` whilst `BUILD_APP_2` is running will fail. Similarly any attempt to invoke `BUILD_APP_2` whilst `BUILD_APP_1` is running will be denied. The user will receive an appropriate error message:

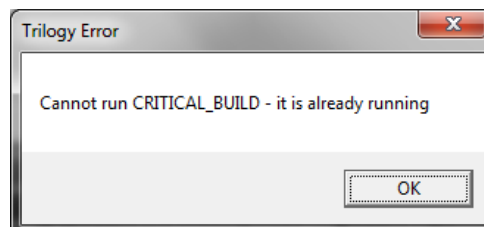


You can also use this technique to prevent a job from being invoked more than once at any one time. By setting the `DenyIfJobRunning` attribute to point to its own job name, you can effectively prevent the job from being invoked more than once.

This job, for example, can only ever be run one at a time:

```
CRITICAL_BUILD:
  Dialog=$TRILOGYHOME/screens/build1.scn
  Program=$TRILOGYHOME/scripts/build1.bat
  DenyIfJobRunning=CRITICAL_BUILD
```

Note, that in these circumstances, the error message displayed is different:



Note, you can list more than one job in the `DenyIfJobRunning` directive. Simply list each job name and separate them with commas:

```
BUILD_APP_1:
  Dialog=$TRILOGYHOME/screens/build1.scn
  Program=$TRILOGYHOME/scripts/build1.bat
  DenyIfJobRunning=BUILD_APP_1,BUILD_APP_2

BUILD_APP_2:
  Dialog=$TRILOGYHOME/screens/build2.scn
  Program=$TRILOGYHOME/scripts/build2.bat
  DenyIfJobRunning=BUILD_APP_1,BUILD_APP_2
```

Here each build is mutually exclusive and only one instance of itself can be run at any one time.



Please be aware, that the server-side job (i.e.: that identified by the `Program` directive in the server-side `trilogy.conf` file) has to be have been started by *Trilogy*. Scripts can still be invoked to populate dialogs and list-boxes. If any dialogs are associated with the job then they must have been committed (by clicking OK).

It is also possible to define a *Trilogy* job that can only be invoked if another job is currently running. For example, you may wish to create a job that monitors (in real time) the output of another job.

To do this, add the directive `AllowIfJobRunning` directive to the appropriate server-side `trilogy.conf` job stanza. The parameter to this directive is a list of any *Trilogy* Jobs that need to be running before this job is allowed to be started. If any of the listed jobs are running then the job is allowed to be started. If none of the listed jobs are running, then the job cannot be invoked.

Here is an example:

```
BUILD_APP_1:
    Dialog=$TRILOGYHOME/screens/build1.scn
    Program=$TRILOGYHOME/scripts/build1.bat
    DenyIfJobRunning=BUILD_APP_1,BUILD_APP_2

BUILD_APP_2:
    Dialog=$TRILOGYHOME/screens/build2.scn
    Program=$TRILOGYHOME/scripts/build2.bat
    DenyIfJobRunning=BUILD_APP_1,BUILD_APP_2

MONITOR_BUILD_OUTPUT:
    Program=$TRILOGYHOME/scripts/showbuildresults.bat
    Stdout=Report
    AllowIfJobRunning=BUILD_APP_1,BUILD_APP_2
```

Here, the *Trilogy* Job `MONITOR_BUILD_OUTPUT` is only allowed to be invoked if either `BUILD_APP_1` or `BUILD_APP_2` is running. If this is true, then a report window is opened and the standard output of the script `showbuildresults.bat` is written into it.



If access to a *Trilogy* Job is denied due to other jobs running (or not running) then the job will not appear as a right-click option from the system tray even if the job is shown as being available from the System Tray Menu. See *Trilogy Client Service for Windows* later in this document for more information on creating Menu Options in the System Tray.

8 Groups and the Group Processor

8.1 Overview

Trilogy Server can control access to *Trilogy Jobs* by use of *Groups*. A group consists of one or more users. When a *Trilogy Client* connects to the server, it identifies the client via the user's login id. *Trilogy* can then identify which group the user is in and allow (or deny) permission to run *Trilogy Jobs*.

In order to make this process as flexible as possible, *Trilogy Server* uses a *Group Processor*. A Group Processor is a library (DLL on Windows platforms, .a or .so on Linux/Unix) which *Trilogy Server* calls in order to determine the group membership for a user. This mechanism allows for *Trilogy* to interact with a number of different sources for its group information:

- A file on the *Trilogy Server*
- LDAP server
- CA Technologies *Software Change Manager*

It is also possible to create your own plug-in provided it meets the interface specification. This specification is detailed below.

Use of the Group Processor is optional. If you restrict access to a job via a `Group=` directive (see *Controlling Job Access* below) then you must specify a Group Processor, otherwise the job will not be accessible.



If your clients are using the *Trilogy Client Service for Windows* (described in the next section) you can also use Groups to notify group members that jobs are running or send them "Balloon" style pop-up notifications from a server-side job (either Linux/Unix or Windows)

8.2 Group Processor Plug-In

The Group Processor is identified by use of the `GroupProcessor=` directive in the server-side `trilogy.conf` file. This is a global directive (i.e.: not associated with a job stanza) and the value should be a full path to the desired Group Processor library file.

For example, to use the "File" mechanism to map users to groups, use the following directive:

Windows:

```
GroupProcessor=$TRILOGYHOME/filegroups.dll
```

Unix/Linux:

```
GroupProcessor=$TRILOGYHOME/filegroups.so
```

You can use other Group Processor plug-in DLLs if desired.

The *Trilogy* Server will need to be restarted to install the Group Processor. When *Trilogy* detects the presence of a Group Processor plug-in, it will load the library and ensure that all the required functions are accessible. Should the library pass this test, then the `trigp_initialize()` function within the library is called. If this returns successfully, then the library is considered valid and Group Processing is activated.

More information concerning the functions contained within the Group Processor are detailed later in this document.

8.3 Controlling Job Access

A *Trilogy Job* can have group access restrictions placed on it by using a `group=` directive in the appropriate *Trilogy Job* stanza entry in the server-side `trilogy.conf` file.

For example, the following `trilogy.conf`:

```
Server=localhost
Port=2301
GroupProcessor=$TRILOGYHOME/filegroups.dll

BuildApp1:
  Dialog=$TRILOGYHOME/screens/builapp1.scn
  Program=$TRILOGYHOME/scripts/buildapp1.vbs
  Group={developer}

BuildApp2:
  Dialog=$TRILOGYHOME/screens/builapp2.scn
  Program=$TRILOGYHOME/scripts/buildapp2.vbs
  Groups={developer,devmgr}

ChangePassword:
  Dialog=$TRILOGYHOME/screens/chngpsswd.scn
  Program=$TRILOGYHOME/scripts/chngpsswd.bat
  Group={devmgr}

Timesheet:
  Dialog=$TRILOGYHOME/screens/timesheet.scn
  Program=$TRILOGYHOME/scripts/timesheet.bat
```

In this case, a user in the "developer" user group can run the *Trilogy* jobs "BuildApp1" and "BuildApp2". A user in the "devmgr" user group can run the *Trilogy* jobs "BuildApp2" and "ChangePassword".

Any client can invoke the "Timesheet" job as there is no `Group=` directive in the job stanza entry and therefore no group restrictions are enforced.

A user in the “developer” user group cannot run the “ChangePassword” job. If they attempt to do so, the *Trilogy* Client will return the error “Trilogy Job not found”.



The error message displayed by the client is always “Trilogy Job Not Found” regardless if the job does not exist or the user is not in the correct group. This is a security feature. The server-side log (if logging is enabled) will record whether the job was denied due to group permission or did not exist.

When receiving a request from a client to run a job with group access restrictions, the *Trilogy* server will interact with the specified Group Processor plug-in (in this example, *filegroups.dll*). If no Group Processor is specified or the Group Processor failed to initialise properly, then access to the job is automatically denied. This prevents access to a job being accidentally granted if the Group Processor should fail.

8.4 Group Processor Functions

Trilogy can interact with any group processor library provided it exports the following functions and it meets the “C” interface standard.



By using “C” (rather than C++) you can use any C Compiler to create your own Group Processor.

When a Group Processor is specified, *Trilogy Server* will load the library and ensure that the following functions are present.

Function Name	Description
<code>int triggp_initialise()</code>	Initialise the Group Processor
<code>unsigned long triggp_get_last_update_timestamp()</code>	Returns the last update timestamp
<code>char **triggp_get_groups_for_user(char *UserName)</code>	Get the groups for a user
<code>char **triggp_get_users_in_group(char *GroupName)</code>	Get the users for a group
<code>void triggp_delete_list(char **List)</code>	Remove a returned list

These functions are described in more details in the following pages.

`int trigp_initialise()`

This function is called when *Trilogy Server* starts. This process can be used to set up connections to whichever system is providing the group information (for example, a remote LDAP server or a relational database). The function should return 0 on success. *Trilogy* will only use the Group Processor should its initial call to `trigp_initialise()` return 0.

unsigned long triggp_get_last_update_timestamp()

This function is called by *Trilogy* before any call to the group membership functions. The Group Processor should return a number that represents the last time the group membership changed. The number should increase whenever the group membership changes in any way.

For the "file" Group Processor, this represents the timestamp when the group membership file was last amended. Any change to the group membership file will result in a later timestamp.

Trilogy uses the value returned from this function to determine whether to call the function in the Group Processor. If the value has not changed since the last time it was called, then it uses its own "cached" version of the data and thus avoids calling the Group Processor. This has significant performance benefits should the Group Processor need to access a remote system or a relational database.

```
char **trigp_get_groups_for_user(char *UserName)
```

This function is called by *Trilogy* when it needs to ascertain which group(s) a user is a member of. The return value should be an array of pointers to character strings with the last entry a NULL pointer.

For example, assume the user "Dave" is a member of the groups "Developer" and "DevMgr". When "Dave" attempts to access a function that is group restricted, *Trilogy* will call this function with a pointer to the null-terminated character string "Dave" as a parameter:

```
char **res = trigp_get_groups_for_user("Dave")
```

This function should then return a pointer to an array of character pointers, with the last pointer set to NULL. Thus:

```
res[0]="Developer"  
res[1]="DevMgr"  
res[2]=(char *)0
```

This will allow *Trilogy* to determine the user's group membership and allow or disallow access accordingly.



Trilogy will only call this function if a call to `trigp_get_last_update_timestamp()` returns a higher value than the previous call OR it has not previously asked for the group membership list for this user.

The list returned should be allocated using a heap-based memory manager (`malloc()` for example). When *Trilogy* is finished processing this list it will call `trigp_delete_list` in the Group Processor to release the memory.


```
char **trigp_get_users_in_group(char *GroupName)
```

This function is called by *Trilogy* when it needs to ascertain which user(s) are members of the specified group. The return value should be a array of pointers to character strings with the last entry a NULL pointer.

For example, assume the group "Developer" has the users "Dave", "Julie", "Mandy" and "Simon" as members. When *Trilogy* is trying to establish which users should receive group notifications (see the next section), this function will be called with a pointer to the null-terminated character string "Developer" as a parameter:

```
char **res = trigp_get_users_in_group("Developer")
```

This function should then return a pointer to an array of character pointers, with the last pointer set to NULL. Thus:

```
res[0]="Dave"  
res[1]="Julie"  
res[2]="Mandy"  
res[3]="Simon"  
res[4]=(char *)0
```

This will allow *Trilogy* to determine which users are in the group and determine which users should receive the notification.



Trilogy will only call this function if a call to `trigp_get_last_update_timestamp()` returns a higher value than the previous call OR it has not previously asked for the users who are members of this group.

The list returned should be allocated using a heap-based memory manager (`malloc()` for example). When *Trilogy* is finished processing this list it will call `trigp_delete_list` in the Group Processor to release the memory.

```
void trigg_delete_list(char **List)
```

This function is used to delete the list previously returned by the functions `trigg_get_groups_for_user()` and `trigg_get_users_in_group()`. *Trilogy* will call this function automatically when it has finished processing the values returned from these functions. This function should free the member returned by these functions.

9 Trilogy Client Service for Windows



This section applies only to Windows Clients

9.1 Overview

Windows clients can choose to install *Trilogy Client* as a Windows Service. When the client is installed as a Windows Service, it places an icon in the System Tray (Also known as the Notification Area). The user can then right-click on this icon to select and run specified *Trilogy Jobs* (those identified as being available from the System Tray).

Having the *Trilogy Client* running as a Windows Service also allows Balloon-Style “notifications” to be sent from *Trilogy Servers* to users in particular groups.



Group Membership is established through a Group Processor (see previous section).

These Balloon Notifications can be sent to specific users, specific groups, specific client machines or a combination of all three. You can also specify that certain user groups should be notified when certain *Trilogy Jobs* are running. When this is configured, the *Trilogy* icon on the desktops of users in the specified group(s) will blink whenever certain jobs are running. This can be used – for example – to notify developers that a build is in progress.

9.2 Installing Trilogy Client as a Service

Trilogy Client can be installed as a service during the normal installation procedure (see section 3.2). If this has not been done, then it can be added as a service after installation by running:

```
trilogy -install
```

from a command window. This will then add “Trilogy Client” to the Services list.



Users of Windows Vista, Windows 7 or Windows Server will need to ensure that the command window has been run with Administrative Privileges (run as Administrator).

The service is set to “Automatic” so that it starts automatically when the Windows machine is started.

9.3 System Tray Icon

When the "Trilogy Client" Service is running an icon appears in the Desktop System Tray (Known as the Notification Area in Windows Vista and Windows 7)



Users of Windows Vista or Windows 7 will need to ensure that "Trilogy Client" is displayed in the Notification Area. The icon will not be displayed by default. To configure, click on the small up-arrow on the left-hand side of the Notification Area and click "Customize" or use Control Panel -> Appearance and Personalization -> "Taskbar and Start Menu Properties" and then click on "Customize" within the "Notification Area" frame.

In either case, in the resulting dialog, identify the "Trilogy Client" and then change the corresponding drop-down to "Show Icon and Notifications".

The icon that is visible in the System Tray will vary depending on whether the Trilogy Client has successfully connected to the Trilogy Server.



Figure 8.1 Trilogy Icon connected to the Trilogy Server successfully.



Figure 8.2 Trilogy Icon when connection to the Trilogy Server cannot be established.

The *Trilogy Client Service* will try periodically (every minute) to connect to the *Trilogy Server*. The server details (hostname and port number) of the *Trilogy Server* is held in the client-side `trilogy.conf` file. When a connection is established, the icon will change to the "Available" state automatically. Similarly, should the connection to the *Trilogy Server* be lost, then the icon will automatically change to the "Unavailable" state. There is therefore no need to restart the *Trilogy Client* service as it will reattempt connection to the server by itself.



Only one *Trilogy* Icon can appear in the System Tray at any one time. This means that if you have a *Trilogy Job* with a `Systray=Y` directive in its job stanza (which places an icon in the system tray when it runs), then the icon used by the *Trilogy Client Service* is used by the job instead.

9.4 Running Jobs from the System Tray

If connection to the *Trilogy Server* has been established then the user can right-click on the icon and select from one or more jobs to which they have access.

Jobs are only selected for inclusion in the right-click menu if the directive `TrayMenu=Y` is included in the job stanza in the server-side `trilogy.conf` file.

The menu presented uses the Job's "title" field to construct the menu item. If the job does not have a "title" attribute then the job name is presented instead.

Access to the job is based on any `Group=` directive in the job stanza entry. If a group attribute is present then the Group Processor is invoked to check whether the invoking client user (their login id) is a member of the specified group. If they are not, then the job is not presented on the right-click menu.

For example, consider the following server-side `trilogy.conf` file.

```
BUILD_APP1:
  Program=$TRILOGYHOME/scripts/buildapp1.bat
  Title=Build Trilogy
  Group=developer
  TrayMenu=Yes

BUILD_APP2:
  Program=$TRILOGYHOME/scripts/buildapp2.bat
  Title=Build RTI
  Group=developer
  TrayMenu=Yes

DAILY_TIMESHEET:
  Program=$TRILOGYHOME/scripts/timesheet.vbs
  Dialog=$TRILOGYHOME/screens/timesheet.scn
  Title=Enter Daily Timesheet
  TrayMenu=Yes

DEPLOY_APP1:
  Program=$TRILOGYHOME/scripts/deployapp1.bat
  Title=Deploy Trilogy
  Group=Test Manager
  TrayMenu=Yes

DEPLOY_APP2:
  Program=$TRILOGYHOME/scripts/deployapp2.bat
  Title=Deploy RTI
  Group=Test Manager
  TrayMenu=Yes

CHECK_CODE:
  Program=$TRILOGYHOME/scripts/checkcode.bat
```

If a user in the "developer" user group right-clicks on their *Trilogy Client Service* icon in the System Tray (Notification Area) then they are presented with a menu based on the jobs marked as being available from the System Tray and to which the user has access:

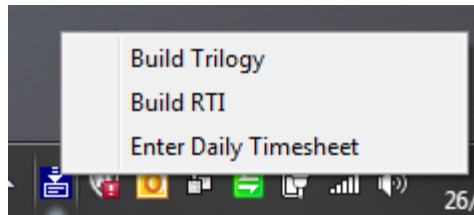


Figure 8.3 Right clicking on the Trilogy Icon on the client presents a menu of server-side jobs to invoke including those jobs marked with TrayMenu=Yes and to which the user has been allowed access.

In this example, the user sees “Build Trilogy” (the job title for the Trilogy Job BUILD_APP1), “Build RTI” (the job title for the Trilogy Job BUILD_APP2) and “Enter Daily Timesheet” (the job title for the Trilogy Job DAILY_TIMESHEET).

BUILD_APP1 and BUILD_APP2 are presented because these are restricted to users within the “developer” user group. DAILY_TIMESHEET is presented as there is no Group= directive in the job but TrayMenu=Yes. CHECK_CODE is not presented since it does not have a TrayMenu=Yes directive. DEPLOY_APP1 and DEPLOY_APP2 are not presented since the user is not in the “Test Manager” user group.

If a user is in the “Test Manager” user group and they right-click on their *Trilogy Client Service* icon in the System Tray (Notification Area), then they are presented with a menu tailored to their role:

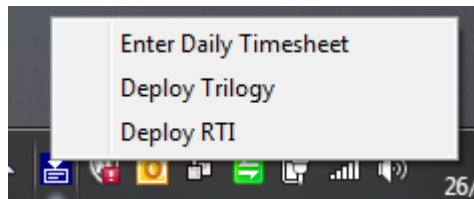


Figure 8.4 The menu presented is based on any user roles.

Selecting any of these jobs allows the server-side job to be run. Any dialog specified in the job will be presented first in the usual way.

The job(s) that are displayed take any `DenyIfJobRunning` and `AllowIfJobRunning` directives into account. If the job is not allowed to be invoked because another job is running (or is not running) then the corresponding menu option is not presented.

9.5 Sending “Balloon” Notifications

A job running on the *Trilogy Server* can send “Balloon” style notifications to *Trilogy Clients* which are docked to the System Tray (Notification Area). These Balloon Notifications can be sent to a specific user, a specific client machine, a user group (or groups) or a combination of all three.



A balloon message can be sent to the user who started the job without needing the *Trilogy Client Service* to be running. For this to work, the `SysTray=Yes` directive must be included in the job stanza which will cause *Trilogy Clients* running on Windows to create a *Trilogy Icon* in the System Tray and animate it to show the job is running.

In order to send a balloon notification, the server-side script can either run the command line tool “trinityfy” (for Unix or Windows servers) or – if the server side script is written in JScript or VBScript on Windows –it can use *the Trilogy Scripting Engine*.

The `trinityfy` command-line tool issues a balloon notification to each matching user who has a *Trilogy Icon* docked to the System Tray (Notification Area).

The `trinityfy` command-line tool has the following options:

<code>-b <servername></code>	The name of the <i>Trilogy Server</i> . Defaults to the server name contained within the server-side <code>trilogy.conf</code> file.								
<code>-p <port number></code>	The port number on which the <i>Trilogy Server</i> is listening. Defaults to the port number contained within the server-side <code>trilogy.conf</code> file.								
<code>-group <group names></code>	A comma-separated list of user groups to receive the notification.								
<code>-client <client machine></code>	The hostname of the client machine to receive the notification.								
<code>-user <user name></code>	The name of the user to receive the notification								
<code>-title <title></code>	The title to be placed on the balloon								
<code>-text <text></code>	The text body of the balloon								
<code>-icon <icontype></code>	The icon to be placed on the balloon. One of: <table><tr><td><code>none</code></td><td>No Icon (default)</td></tr><tr><td><code>warning</code></td><td>A Warning Icon</td></tr><tr><td><code>error</code></td><td>An Error Icon</td></tr><tr><td><code>info</code></td><td>An Information Icon</td></tr></table>	<code>none</code>	No Icon (default)	<code>warning</code>	A Warning Icon	<code>error</code>	An Error Icon	<code>info</code>	An Information Icon
<code>none</code>	No Icon (default)								
<code>warning</code>	A Warning Icon								
<code>error</code>	An Error Icon								
<code>info</code>	An Information Icon								
<code>-now</code>	Displays the balloon immediately. Balloon Notification messages are normally queued with the “next” balloon being displayed when the first balloon is removed due to timeout or is closed by the user.								

For example, running the following command will cause a balloon notification to be displayed on the appropriate client:

```
trinity -title "The Balloon Title" -text "The Balloon Text" -icon  
warning -user Dave
```

In this case, Trilogy will determine at which client machine(s) the user with the login ID of "Dave" is logged in, and will display the balloon accordingly at each appropriate desktop.

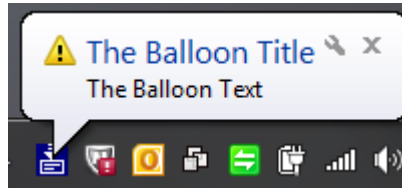


Figure 8.5 A Balloon Notification sent from a server-side script.

The balloon will be displayed for an amount of time before it fades from view. If the user does not move the mouse or operate the keyboard, the balloon will stay permanently (since the user may well be away from their machine) but it will fade when mouse or keyboard activity resume.



Remember that "trinity" is a server-side command line tool and, as such, is available on Unix and Linux servers as well as Windows. It is therefore possible to have a Unix Shell Script notify Windows Clients of its activity via balloon messages.

The following criteria can be used to determine the user desktops to be targeted with a balloon-style notification.

9.5.1 Notifying Groups

To notify a group use either:

```
-group group-name
```

or

```
-group group-name,group-name ...
```

Either the name of a single group or a comma-separated list of groups. This causes *Trilogy* to find each user associated with the specified group(s) and locate the desktop on which they are logged-in. Each identified user will then receive the balloon notification.

9.5.2 Notifying Users

To notify a user:

```
-user user-name
```

This will find each client machine and desktop where the user is logged in and send the notification to that desktop.

Note, the notification will be sent to each client desktop which has a *Trilogy Client* icon docked to the System Tray and where the specified user is logged in. This targeting can be further restricted by adding a `-client` directive as outlined below.

9.5.3 Notifying Client Machines

To send a notification to a client machine, use:

```
-client client-hostname
```

This will send the notification balloon to each desktop which exists on the specified client machine. If three people are logged into this machine, each user will receive the balloon notification.

If you wish to restrict the notification to a particular desktop on the client machine then use `-client` along with `-user`. This will then target the notification to a particular user on that particular machine.

9.5.4 Automatic Notification Routing

If `-client` is not specified on the command line then `trinotify` will take the value from the environment variable `TRICLIENTNODENAME`.

If `-user` is not specified on the command line then `trinotify` will take the value from the environment variable `TRICLIENTUSERNAME`.

Both `TRICLIENTNODENAME` and `TRICLIENTUSERNAME` are set automatically by *Trilogy* whenever it invokes a server-side job on request from a client. Therefore, a server-side job invoking `trinotify` without specifying any other routing parameters will automatically send the notification to the invoking user's desktop on the client machine from where they instigated the job. Thus, running `trinotify` like this:

```
trinotify -title "Build Information" -text "Build is Complete"
```

Will automatically send the balloon notification to the client desktop where the job was started.

9.5.5 Balloon Icon Types

When using Balloon Notifications, it is possible to specify different icons for the balloon. Using trinotify you can specify an icon type using the `-icon <icontype>` directive. The result of each icon type is shown here:

Icon = none

```
trinotify -title "The Balloon Title" -text "The Balloon Text" -icon none -user Dave
```

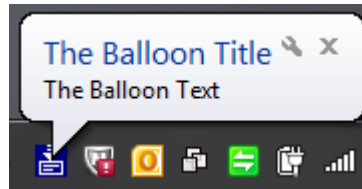


Figure 8.6. A Balloon Notification with no icon

Note, "none" is the default for the icon.

Icon = Warning

```
trinotify -title "The Balloon Title" -text "The Balloon Text" -icon warning -user Dave
```

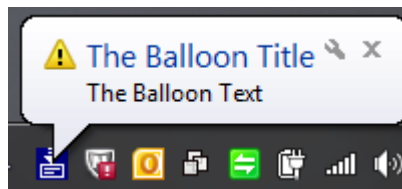


Figure 8.6 A Balloon Notification with a Warning Icon

Icon = Info

```
trinotify -title "The Balloon Title" -text "The Balloon Text" -icon info -user Dave
```

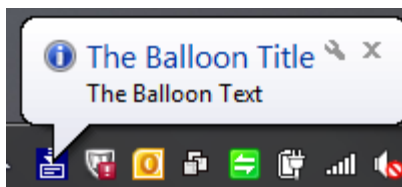


Figure 8.7 A Balloon Notification with an Information Icon

Icon = Error

```
trinotify -title "The Balloon Title" -text "The Balloon Text" -icon  
error -user Dave
```

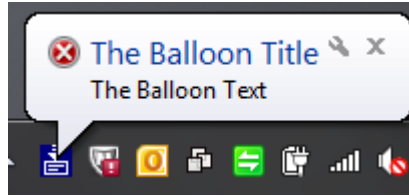


Figure 8.8 A Balloon Notification with an Error Icon

9.5.6 Balloon Display Order

Normally, balloons are displayed by the *Trilogy Client* in the order in which they were called by the server-side script. This means that if the server-side script sends a "Job Started" balloon notification followed by a "Job Ended" balloon notification, then the client will display "Job Started" and will only display "Job Ended" when the first "Job Started" balloon has either timed out or has been closed by the client user.

There may be circumstances where you want the balloon to be displayed immediately. For example, the "Job Ended" message should really replace the "Job Started" message immediately it is issued. In that way, the user will know that the job is actually complete without having to wait for the balloon to change.

In addition, it may be desirable to show error conditions immediately so the users are aware of any issue as soon as it has arisen.

To this end, `trinotify` supports a `-now` flag. If `trinotify` is called with this flag then any *Trilogy* balloon notification that is currently being displayed by the docked *Trilogy* Icon will be removed and the new balloon content will immediately replace it.

Note that any "pending" balloon notifications are still held in the queue at the client and are displayed after the "now" balloon has been cleared (either by the user closing the balloon or via a timeout).

9.5.7 Standard Output As Balloon Message

If the job's Standard Output stream is set to "Popup" (`stdout=popup`) and the job is set to run in the System Tray (`SystemTray=Yes`) then the standard output stream from the server-side job is displayed at the client as a balloon message rather than a dialog box. If the standard output exceeds the amount of characters that will fit into a single balloon then the output is split and multiple balloons are displayed containing each part of the standard output.

9.6 Notify Users of Job Running

You may find it desirable to notify groups of users when a job is running. This can be done with balloon notifications (by sending a balloon message to the

appropriate user group notifying them that a server-side job has started and again when it is complete).

However, if the job is particularly long-running then your users may need to know that it is still running.

To support this, *Trilogy Server* supports a `NotifyRunGroup=` directive. This directive should be placed in the server-side `trilogy.conf` in the stanza entry for the required job. By using this directive, you can specify a group (or groups) which are to be notified when the *Trilogy Job* is running.

For example:

```
DO_BUILD:
    Program=$TRILOGYHOME/scripts/build.sh
    NotifyRunGroup=Developer
```

When the "DO_BUILD" job is running, every user in the "Developer" user group (as determined by the Group Processor) who has a docked *Trilogy Client* icon, will see the icon flash as though a job were running.

When the job completes the icon on the Developer users' desktops will stop flashing.

You can notify more than one user group by including a comma-separated list of groups like this:

```
DO_BUILD:
    Program=$TRILOGYHOME/scripts/build.sh
    NotifyRunGroup={Developer,DevMgr}
```

Here, users in the Developer and the DevMgr user groups are notified whenever the DO_BUILD job is running.



Remember that users in the targeted group need to have the *Trilogy Client* docked in the system tray (Notification Area) in order to see the "Job Running" indication. Informing users that a job is running only sets the Icon blinking. There is no indication as to which job is executing. It is recommended that the server-side job issues a Balloon Notification to the same group(s) in order that the target user(s) can identify which job has started or stopped.

9.7 Client Port Number

Both "Balloon" style notifications and the icon animation that indicates that a job is running require a message to be sent from the *Trilogy Server* to the appropriate *Trilogy Client*(s). To do this, each *Trilogy Client* running on a client machine opens a UDP port to listen for these messages from the *Trilogy Server*.

Normally, the *Trilogy* Client opens an “ephemeral” UDP port (that is, a random port number that is not being used by any other application) and sends the port number it is using to the *Trilogy* Server so that server knows how to communicate with that client.

If you have a firewall running, you may need to “fix” the UDP port that the client uses to allow a connection to take place from the *Trilogy* Server to the *Trilogy* Client.

To fix the UDP port, use the directive `ClientPort` in the client-side `trilogy.conf` file. For example, the directive:

```
ClientPort=60000
```

instructs the client to open the UDP port 60000 to listen for incoming connections. You should then open UDP port 60000 through the firewall to allow the server to connect to the client.



You can see the port number that the client is using by running the *Trilogy* Server using the `-showdocked` option. This will list all the Windows Client Services that have registered with the server as being docked, along with the port number that they are using.

9.8 Advanced Configuration – Communicating Across Subnets

When a Windows Client notifies the server that it is “docked” (and can receive balloon style notifications), the server records the client’s IP address and the UDP port on which it is listening for notifications. When the server wishes to send the client a notification, it does so by communicating with the client using this IP address and the UDP port.

The reason for using an IP address is that host name resolution may not necessarily work from a server to a client. It is normal for a client to be able to resolve the IP address of “Server12” – less common that “Server12” can resolve the IP address of “Client9812”.

However, there may be issues if the server is on a different subnet to the clients. In this case the server will attempt to communicate with the client using its IP address – if the IP address is in a different network then the message may not be able to be routed to the client.

For example, suppose the server is in the 192.x.x.x network and the clients are in the 10.x.x.x network. The default gateway from the 10.x.x.x network will allow the client to connect to the server. However, when the server attempts to send a message to a client in the 10.x.x.x network it may be unable to route.

The problem is compounded if the subnets share a common network ID. This can happen if the *Trilogy* Server is installed in a hosted (cloud) environment with NAT translation between the networks. Here, the *Trilogy* Server may exist in a 10.x.x.x network, the clients are in *another* 10.x.x.x network with connectivity between the two performed by a single gateway with NAT. In this case, the server will see a client IP address of 10.x.x.x and will assume that this is in its own network. This means that messages from the server to the clients will be lost.

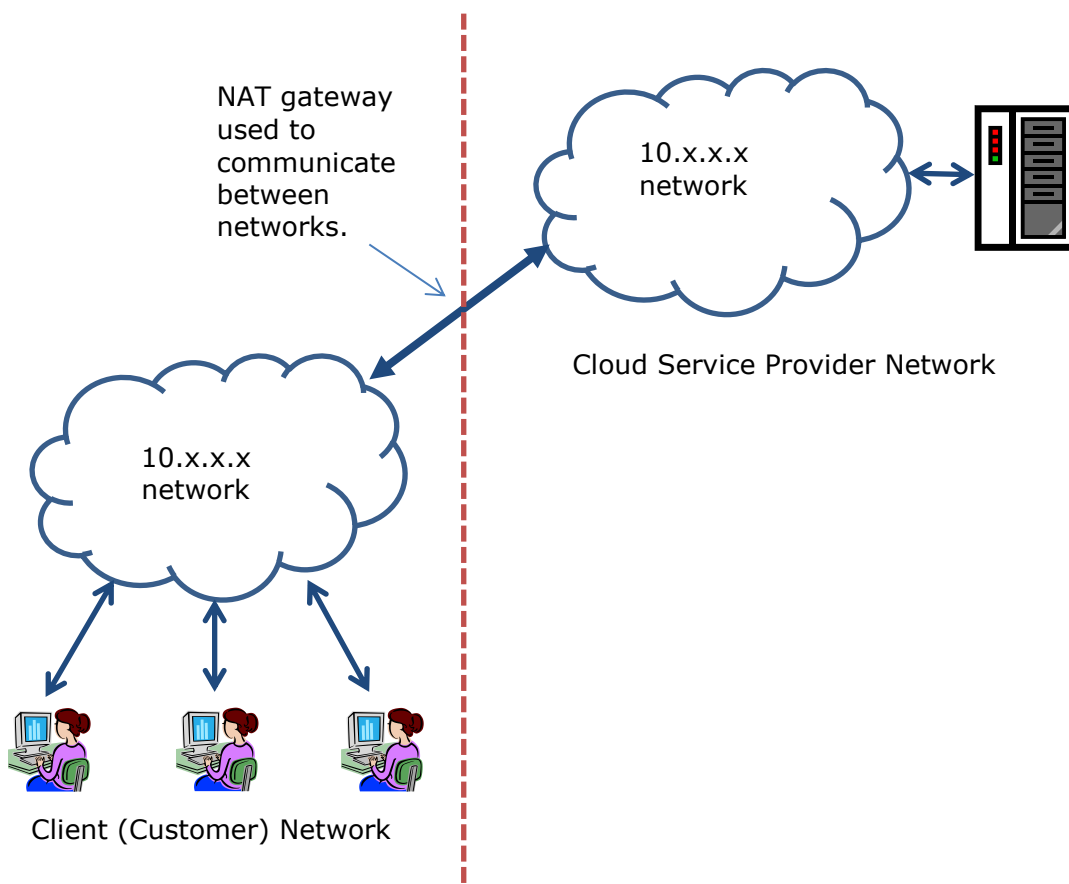


Figure 9.1 In a hosted environment, sending notifications to clients is complicated by the lack of a common network.

The solution to this problem is to set up a *Trilogy* Server as a “relay”. To do this, install another *Trilogy* Server on a machine in the same subnet as the clients. The only function of this server is to receive messages from the “main” *Trilogy* Server and route them on to the appropriate client(s) within its own network.

The “main” *Trilogy* Server (the one hosted outside of the subnet where the clients are connected) can then be pointed to this “relay” server with the use of a `RelayServer=` and a `RelayPort=` directive in its `trilogy.conf` file. This entry is outside of any job stanza (alongside the `Port=` directive).

The server identified by the `RelayServer=` directive should be addressable from the external network so will probably require a NAT entry to enable IP traffic to be routed from the external network to the client network.

If a `RelayServer=` directive is present then no attempt is made by the *Trilogy* Server to send messages direct to the client. Instead, the message is sent to the *Trilogy* Server identified by the `RelayServer=` directive (to the port identified by the `RelayPort=` directive). This *Trilogy* Server will then route the message on to the required client(s).

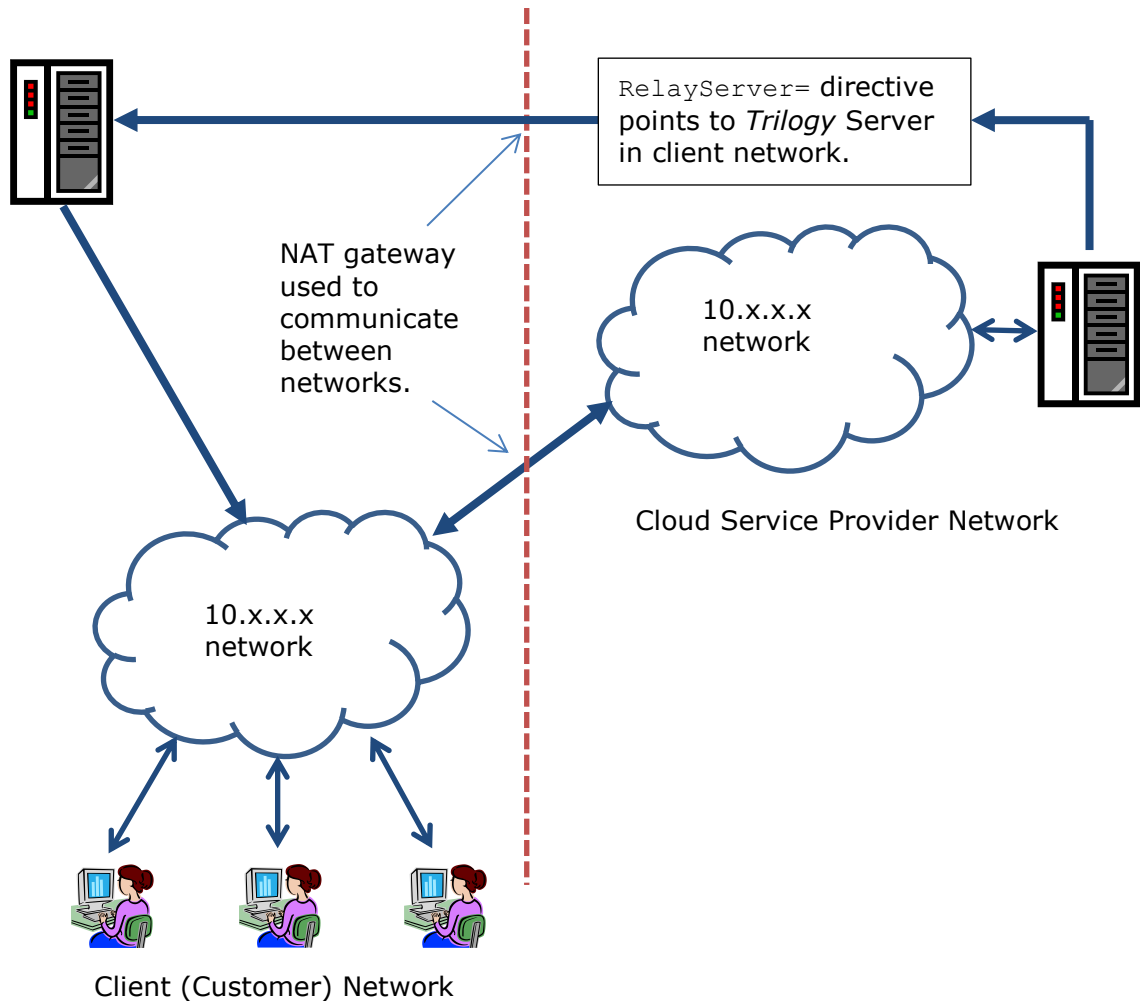


Figure 9.2 Use Relay to route client notifications across networks.

10 The Scheduler

10.1 Introduction

Trilogy Server has a built-in scheduler. You can use scheduler directives to allow *Trilogy* Jobs to be invoked automatically on specified dates and times.

The scheduler also allows dependent jobs to be configured. This means that other jobs can be launched automatically dependent on the exit status of a scheduled job. This means you can run a task on an automatic timed basis (for example, a build) and – if that job was successful – run a second job automatically (for example, a deployment). If the timed job were to fail, then *Trilogy* can automatically run another job (for example, to send out an email warning of the failure).

The directive `NotifyRunGroup` can be used to indicate to particular groups of users that the job is running (just as if it had been invoked by a user) and the job itself can use issue balloon-style notifications when it runs.



A scheduled job cannot interact with the user beyond the notifications described above. Therefore, any User-Interface directives included in the job stanza (for example `Dialog` or `DialogScript`) are ignored.

10.2 Specifying a Scheduled Job

For a *Trilogy* job to be scheduled to run under control of the scheduler, the directive `AutoRun=Yes` must be specified in the job stanza. If this is not done, then the job will not be under scheduler control.

When `AutoRun=yes` is specified, other directives control the dates and times on which the job will be invoked. The only mandatory directive is `AutoRunTimes`. This lists the times of day on which the job will be run. All other directives are optional and are used to limit the dates and days on which the job will be run.

When a job runs under control of the *Trilogy* scheduler, the script or program specified by the `Program=` directive is invoked on the *Trilogy* Server whenever the pre-determined date and time is reached. The Standard Output and Standard Error of the job is captured and stored in a log file.



A dependent job has access to this log file. It can read the output from its “parent” job (the job on which it depends) and can use it as it wants (for example, by sending it as an email attachment).

10.3 Related Directives

A job running under the control of the *Trilogy* Scheduler cannot display or interact with any client-side dialog since the job was not instigated from a client machine. Therefore, any directives related to client-side dialog definitions are ignored. For example, `Dialog`, `DialogScript`, `PopulateFieldnWith`, `PopulateListBox`, etc. are all ignored when a job is running under control of the scheduler.

However, you may find the following directives useful when specifying a scheduled job:

10.3.1 NotifyRunGroup

You can use `NotifyRunGroup` to indicate to the end-users that a timed job is in progress. The server side script can also run “trnotinify” to send out balloon-style notifications to the appropriate user group.

10.3.2 Environment

You can set up an environment for the server-side job so that it runs with the appropriate environment variables.

10.3.3 Param

Use the `Param` (or `Params`) directive to specify parameters that are passed to the script specified by the `Program` directory.

10.4 Specifying Run Times

The directive `AutoRunTimes` is used to specify the times of day on which the job will be run. Times are specified in 24 hour format.



The timer resolution is one minute. *Trilogy* Server will scan for jobs to run every minute (whenever the minute of the system clock changes). Times are specified in hours and minutes – it is not possible to specify seconds.

One or more times can be specified. For example, this job entry:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunTimes=15:00
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Means that the job “TIMED_JOB” will run at 3pm (the script `timedjob.vbs` will be invoked automatically on the *Trilogy* Server at 3pm).

This:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunTimes={0:00,3:00,15:15,23:09}
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Means that the job "TIMED_JOB" will run at Midnight, 3am, 3:15pm and 11:09pm (the script `timedjob.vbs` will be invoked automatically on the *Trilogy* Server at these times).

You can also specify a single range of times, along with an "interval" like this:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunTimes=20:00-21:15
  AutoRunInterval=15
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

This means that the job will run automatically at 8pm and then every 15 minutes until 9:15pm. That is, the script `timedjob.vbs` will be run automatically at:

8pm	(20:00)
8:15pm	(20:15)
8:30pm	(20:30)
8:45pm	(20:45)
9pm	(21:00)
9:15pm	(21:15)

Without any other directives, the job will run every day at the times specified by the `AutoRunTimes` directive. If you want to limit the job to run on particular dates or days, then you can use other directives listed below.

10.5 Specifying Run Days

If you want to limit the days on which the job will run, use the `AutoRunDays` directive.

The value of `AutoRunDays` is a single day, a group of days (separated by commas), a range of days or a combination of any of these.

Days can be specified either using common English abbreviations (Mon, Tue, Wed, Thu, Fri, Sat, Sun) or as numbers (Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6 or Sunday = 7)

For example, this job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDays=Mon
  AutoRunTimes=20:00-21:15
  AutoRunInterval=15
```

```
Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Is run automatically every 15 minutes between 8pm and 9:15pm but only on a Monday.

This job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDays=Mon,Fri
  AutoRunTimes=20:00-21:15
  AutoRunInterval=15
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Is run automatically every 15 minutes between 8pm and 9:15pm on a Monday and Friday.

This job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDays=Mon-Fri
  AutoRunTimes=20:00-21:15
  AutoRunInterval=15
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Is run automatically every 15 minutes between 8pm and 9:15pm on Monday, Tuesday, Wednesday, Thursday and Friday. This is equivalent to this:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDays=1-5
  AutoRunTimes=20:00-21:15
  AutoRunInterval=15
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

10.6 Specifying Run Dates

To limit the job to run on particular dates, use the `AutoRunDates` directive.

`AutoRunDates` specifies a single date, a group of dates (separated by commas), a range of dates or a combination of these.

Dates are a number between 1-31 and refer to the day of the month.

For example, this job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDates=1
  AutoRunTimes=20:00
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Is run automatically at 8pm on the first day of every month.

You can specify a number of dates:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDates=1-5
  AutoRunTimes=20:00
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

This job runs at 8pm on the 1st, 2nd, 3rd, 4th and 5th of each month.

You can combine `AutoRunDates` with `AutoRunDays`, in which case the two criteria are combined to limit when the job can run. For example this job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunDates=1-7
  AutoRunDays=Mon
  AutoRunTimes=20:00
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Runs on the first Monday of every month (the first Monday that coincides with a date in the range 1st-7th of the month).

10.7 Specifying Run Months

You can further restrict a scheduled job to run only during particular months. To do this use the `AutoRunMonths` directive.

Months can be specified either using common English abbreviations (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec) or as numbers (January = 1, December = 12).

`AutoRunMonths` can be given as a single month, a group of months (separated by commas), a range of months or a combination of these.

For example, this job:

```
TIMED_JOB:
  AutoRun=yes
  AutoRunMonths=Jan-Nov
  AutoRunDates=1-7
  AutoRunDays=Mon
  AutoRunTimes=20:00
  Program=$TRILOGYHOME/scripts/timedjob.vbs
```

Runs at 8pm on the first Monday of every month except in December.

10.8 Setting Standard Input

A job's standard input is normally closed. However, you can specify that a job should read its standard input from a server-side file.

To do this, use the `AutoRunStandardInput` directive. For example:

```
AutoRunStandardInput=$TRILOGYHOME/standardinput.txt
```

When the job runs the script specified by the `Program` directive will have its standard input taken from the file `standardinput.txt` in the `TRILOGYHOME` directory.



Any environment variables in the specified file name are not expanded until the job is being executed. This means you can have dependent jobs whose standard input is taken as the output from the timed job. Simply specify the standard input of the dependent job like this:

```
AutoRunStandardInput=$TRIPARENTLOGFILE
```

See *Dependent Jobs* and *Environment Variables* below for more information.

10.9 Dependent Jobs

A job running under control of the *Trilogy* Scheduler can have *dependent* jobs. A dependent job is a job that is run automatically by *Trilogy* whenever the timed job completes.

A dependent job can be set to run whenever the “parent” job (that is, the job on which it is dependant) exits either with a successful exit code (0) or a non-successful exit code ($\neq 0$). In this way, you can invoke jobs which run automatically following the completion of a timed job to either:

- a) Continue processing (for example, an automatic deploy following a timed build) or
- b) Handle an error condition (for example, by emailing a failure warning).

A job invoked by the scheduler can have zero, one or multiple dependent jobs. Dependent jobs are specified by the `AutoRunOnSuccess` or `AutoRunOnFailure` directives.

For example, consider these jobs:

```
BUILD:
  AutoRun=yes
  AutoRunMonths=Jan-Nov
  AutoRunDays=Wed,Fri
  AutoRunTimes=8:00
  Program=$TRILOGYHOME/scripts/build.sh
  Param=/build/build_directory
  AutoRunOnSuccess=DEPLOY
  AutoRunOnFailure=LOGFAIL

DEPLOY:
  Environment=$TRILOGYHOME/scripts/env.txt
  Program=$TRILOGYHOME/scripts/deploy.sh
  Param=/build/build_directory

LOGFAIL:
  Program=$TRILOGYHOME/scripts/logfail.sh
```

Here, the "BUILD" timed job is set to execute on Wednesday and Friday mornings at 8am (except in December because the site has a change freeze in force during December). The script `build.sh` is executed by *Trilogy* automatically at these times. The `Param` directive specifies that the script is passed a single parameter `/build/build_directory`. If the `build.sh` script exits with a success code (exit code 0) then *Trilogy* will automatically run the job(s) specified by the `AutoRunOnSuccess` directive. In this example, this means that *Trilogy* will automatically run the `DEPLOY` job. This will then run the `deploy.sh` script in order to deploy the newly created build. The `deploy.sh` script is passed a single parameter of `/build/build_directory`.

If the `BUILD` job were to fail (`build.sh` returns a non-zero exit code), then *Trilogy* will run the job(s) specified by the `AutoRunOnFailure` directive. This means that *Trilogy* will automatically run the job `LOGFAIL`. This will run the script `logfail.sh` which can then notify users that the job has failed. This can be done with a balloon-style notification to Windows Clients (for example) or with an email.

A dependent job can read the output from its parent job (the job on which it is dependant) by using the environment variable `TRIPARENTLOG`. This variable gives the full path of the output log file containing the standard output and error streams from the parent job. Thus, "logfail.sh" could, for example, send this file as an attachment in an email.

You can also specify multiple dependent jobs both for `AutoRunOnSuccess` and `AutoRunOnFailure`. To specify multiple dependent jobs, separate them with commas. For example, this:

```
BUILD:
  AutoRun=yes
  AutoRunMonths=Jan-Nov
  AutoRunDays=Wed,Fri
  AutoRunTimes=8:00
  Program=$TRIOLOGYHOME/scripts/build.sh
  Param=/build/build_directory
  AutoRunOnSuccess=DEPLOY,NOTIFY_SUCCESS
  AutoRunOnFailure=LOGFAIL
```

Means that if `build.sh` exits with a successful exit code (0), then *Trilogy* will invoke both the `DEPLOY` and `NOTIFY_SUCCESS` jobs in parallel.

Dependent jobs can be dependent on other dependent jobs – not just on timed jobs. You can use this to build a “chain” of dependent jobs that run automatically whenever a timed job completes.



Dependent jobs are only available to jobs invoked via the *Trilogy* Scheduler (Timed Jobs or Dependent Jobs). Dependent Jobs are not invoked when a job is started from a *Trilogy* Client.

10.10 Environment Variables

When a script is invoked by the scheduler, *Trilogy* sets the environment variable `TRIREASON` to “TIMED”. This is the same regardless of whether the job was invoked directly by the scheduler or as a dependent job. Dependent jobs have 3 other environment variables set:

<code>TRIPARENTJOB</code>	The name of the job that ran on which this job is dependent.
<code>TRIPARENTEXITCODE</code>	The exit status of the parent job
<code>TRIPARENTLOGFILE</code>	The name of the log file where the parent’s standard output and error streams are stored.

11 Trilogy - Command Line Options

11.1 Trilogy Client

A number of command line options are available to the Trilogy Command Line tool. These are details below.

-bg	Background working. Causes the <i>Trilogy</i> client to detach itself from its parent process and to continue running "in background". This is useful when invoking <i>Trilogy</i> from inside scripts such that the server-side job runs asynchronously. Notification dialogs can be brought to the screen by specifying a destination of "Popup" for either the standard out or standard error streams of the server process. See section 12.11 for more information.
-b <server name>	Specifies an alternate server name. If specified, this overrides the <code>Server=</code> directive in <code>trilogy.conf</code> .
-i <filename>	Specifies a client-side file from which the standard input of the server-side job will take its data. Technically, there is no difference between using the <code>-i <filename></code> option and redirecting the file to the Trilogy command-line client's standard input. However, using the <code>-i <filename></code> option sets the <code>TRISTDINFILENAME</code> environment variable for the server-side script. See <i>Server Side Job Control</i> below for more information.
-p <port number>	Specifies an alternate port number. If specified, this overrides the <code>Port=</code> directive in <code>trilogy.conf</code> . Use this directive if you have multiple <i>Trilogy</i> Servers each listening on different ports.
-nd	No Display. Prevents <i>Trilogy</i> from opening any dialog boxes. Useful when the <i>Trilogy</i> Client is operating on a remote machine when no user interaction is possible.
-jobs	Presents a dialog listing all the <i>Trilogy</i> Jobs accessible to the calling user.

-s

Unix Only. Using this option simulates the operation of the Windows *Trilogy Scripting Engine*. When this option is specified on a Unix Client, the associated script (Program=) is not run on the server. Instead, the values of the fields are printed to standard out in the form `TRIFIELDn="value"`, one entry per line. By using this technique it is easy to set the values into Unix shell scripts. Simply invoke *Trilogy* via an *expr* command to set the environment variables into the local script:

```
expr `trilogy -s my_dialog`
```

or

```
expr $(trilogy -s my_dialog)
```

-x

Unix Only. Used in conjunction with the `-s` option described above, this option adds `export TRIFIELDn="value"` command to each `TRIFIELDn="value"` output line.

11.2 Trilogy Server

Running the server process with no command line options simply starts the server. There are, however, a number of options that can be specified which return useful information:

-dumplic	Dumps out the current license usage. Lists the number of client nodes used, the maximum number of clients available (by virtue of the installed license key) and a list of each client hostname that has connected to this server.
-showdocked	Lists all the Windows clients that have the <i>Trilogy Client Service</i> running. The output shows the Windows user name (the user who is logged in at the client), the hostname of the client machine, its IP Address and the UDP port on which it is listening for incoming connections. This UDP port is used for both balloon style notifications and for animating the System Tray Icon during a <code>NotifyRunGroup</code> .
-showjobs	Lists the jobs that are currently running under the control of the <i>Trilogy Server</i> . This will list both scheduled jobs and jobs running as a result of a client request. The output shows the user who is running the job (<code>SCHEDULER</code> for timed jobs), the total run time since the job was started and the process id.
-killjob <process id>	Kills the job specified by the process id.
-shutdown	Shuts down the <i>Trilogy Server</i> .

12 Server Side Job Control

12.1 Introduction

All jobs invoked by *Trilogy* are executed by the *Trilogy* Server application and run on the same node as the *Trilogy* Server. In this section we will detail the characteristics of *Trilogy* Job Control – how the jobs are launched, how they communicate with the client, the environment they inherit and so on.

12.2 Environment

An *Environment* means the set of environment variables to which a process has access. On a Windows platform, environment variables are divided into "System" and "User" variables. Effectively, "System" variables are created and owned by the operating system, whilst "User" variables are created and owned by the individual user who has logged on to the machine.

On Unix/Linux, this differentiation between system and user variables is controlled by "system" level shell files such as /etc/profile and "local" shell files such as .profile within the user's home directory.

In either case, an application has access to these environment variables. Within an application, no distinction is made between "system" and "user" variables.

Environment variables can be used for many different purposes. Some, such as PATH are used by the operating system. Others (such as TRILOGYHOME) are used by individual applications.

Generally speaking, an application inherits a copy of the environment when it starts. An application can make changes to its copy of the environment but such changes are not visible to the "parent" environment – such changes are retained only in the application's copy of the environment.

Operating System Environment Variables	
Name	Value
A	10
B	20
C	30
D	40

Figure 8.1 – Environment Variables

Should an application launch (spawn) a new application, then this new application inherits a *copy* of the parent's environment.

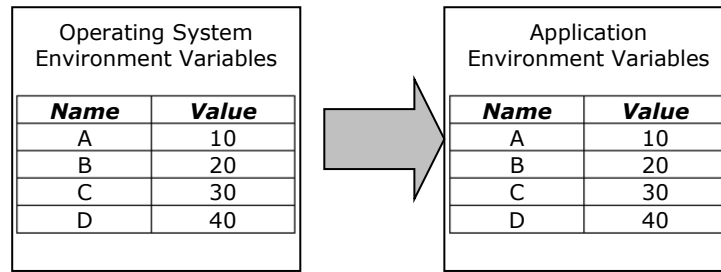


Figure 8.2: A launched application inherits a copy of the environment.

This copy of the environment is separate to the operating system's environment. Thus, any change made by the application to its environment is visible only to itself – the operating system sees no changes.

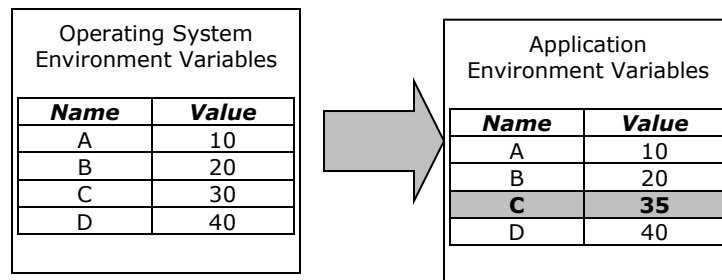


Figure 8.3: Here, the application has made a change to its Environment (C=35). The change is not visible to the caller.

If a parent process alters its environment and then starts a new process, this new process sees the changes made by its parent. However, any changes made by the child to *its* copy of the environment will not be visible to the parent.

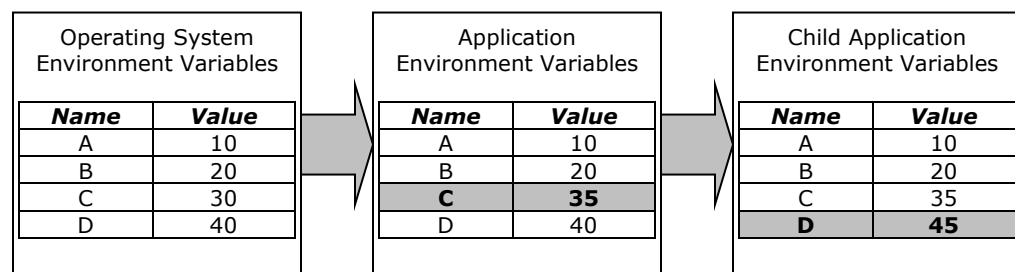


Figure 8.4: If an application launches a "Child" application, that child is given a copy of the environment of its parent. Thus, it sees any changes to the environment made by the parent (in this case C=35) but any changes it makes to its own environment are not visible to the parent (D=45).

12.3 Trilogy Server Environment

When *Trilogy* Server is started, a copy of the current environment is taken. Therefore – as discussed above – any jobs launched by *Trilogy* Server would normally inherit the same environment as that of *Trilogy* Server itself.

In practice, this means that variables such as `PATH` would be inherited by processes launched by *Trilogy*. This may lead to security considerations.

In order to work around such problems, *Trilogy* provides an `Environment` directive. This is included in the server-side `trilogy.conf` file – either globally (outside a job stanza) or associated with a particular job. In either case it points to a file containing environment definitions like this:

```
name=value
```

When programs are executed by *Trilogy* Server, the environment file is read and the environment variables referenced therein are added to the program's environment. If an environment variable referenced in this way already exists in the environment, then it is overwritten. In this way, you can control which environment variables are made available to launched programs and which values they take.

More information on the `Environment` directive is available in *Chapter 13 – trilogy.conf Reference Guide*.

Trilogy Server will set other environment variables before launching a child process and the name and content of these environment variables is detailed later in this section.

12.4 How Jobs are started

The manner in which jobs are started and controlled differ between Unix, Linux and Windows Servers.

12.4.1 Unix/Linux Servers

On Unix/Linux Servers, a connection from a *Trilogy* Client results in *Trilogy* Server forking a copy of itself to handle the connection. If a server-side job is required to be executed, then this copy does a `fork/exec` in order to execute the job. Anonymous pipes are used to direct the standard in, standard out and standard error streams from the launched application to the copy of the *Trilogy* Server and thence – if required – back to the client. Thus, when a process is running under the control of *Trilogy*, you will see three processes running on the Unix/Linux Server – `trilogyd` (the "master" *Trilogy* Server), another `trilogyd` (a child process whose parent process id is that of the "master" *Trilogy* Server) and finally the job itself (whose parent process id is that of the "child" *Trilogy* Server).

12.4.2 Windows Servers

On Windows Servers, a connection from a *Trilogy Client* starts a new thread. If a server-side job is required to be executed, then this thread takes a copy of the environment and passes this to a `CreateProcess` call. Threads are started to read and write the standard input, standard output and standard error of the launched jobs and anonymous pipes are used to direct these various streams to and from the launched application and thence - if required - back to the client.

If real-time output is required at the client (`Stdout=Report` or `Stdout=Output`) then a process called *trilogytty* is launched by the *Trilogy Server* and *trilogytty* then starts the desired application. This is used to change the buffering characteristics of the launched application. More information on buffering is provided below.

12.5 Buffering

On Unix, Linux and Windows systems, the Operating System controls the buffering of the standard output and error streams. Standard Error is not normally buffered. The buffering utilised for Standard Output will depend on where the stream is being routed. If the output is a terminal or console then "line" buffering is used. Whenever a line-end sequence is output by the application (Linefeed (0x0a) for Unix and Linux, Carriage-Return, Linefeed (0x0d 0x0a) for Windows) then the output buffer is flushed. This means that whenever an application writes a line to its standard output, the results are visible immediately.

If the output is not a terminal (it has been redirected to a file for example, or the output is being piped to another application) then "block" buffering is used. The buffer is only flushed when it becomes full. Such buffers are typically around 4k in size.

Any job launched by *Trilogy Server* has its output redirected via pipes so that it can be read by the *Trilogy Server* and pushed back over the network to the *Trilogy Client* if required. If the job being launched is a script (shell script or batch file) this normally makes no difference. However, if the job launched is a binary executable (compiled code) then the run-time libraries will determine that the output is being sent to a pipe and the standard output will be set to block-buffered.

What this means in practice is that the output from the server-side job is collated together until the output buffer is full (4,000 characters or so) or the job completes.

If a job has been invoked via *Trilogy* - and `Stdout` is `Report` or `Output` - then real-time output is required (line-buffering). The way that *Trilogy Server* achieves this is different on Unix, Linux and Windows Servers.

12.5.1 Unix/Linux

On Unix/Linux servers, a free terminal device is opened (usually `/dev/tty nn` where nn is a two-digit number). Output from the server-side job is written to this terminal device and *Trilogy Server* reads this output from this terminal device. This has the effect of making the output of the job look like it is going to a terminal (in other words, `isatty` will return `true`). The result is line-buffered output and the client will display the results of the server-side job in real-time.

A side-effect of this process is that jobs whose output changes dependent on whether they are routing their output to a terminal device or a pipe (such as `ls`), will act as though they are talking to a terminal.

12.5.2 Windows

On Windows servers, a job called `trilogytty` is launched and this job then starts the required server-side job. `trilogytty` creates an invisible console for the job. As the job writes to the console, `trilogytty` reads its output and sends it back to the *Trilogy Server* which then routes it to the *Trilogy Client*. This results in real-time output.



A side effect of `trilogytty` is that it will corrupt binary output. If you are planning on sending binary output to the standard output of your Windows server-side script with `StdOut=Output` (and redirecting this output at your client) then you should set `UseTTY=no`. See `UseTTY` below for more information.

12.5.3 UseTTY Directive

The `UseTTY` directive in the server-side `trilogy.conf` file overrides the default buffering behaviour. When the `Stdout` directive of the job is set to `Report` or `Output` then `UseTTY` is set to "Y" automatically. For any other value, `UseTTY` is set to "N". When `UseTTY` is set to "Y", the output of the server-side job is forced to be line-buffered using the techniques described above. Therefore, this flag is only necessary in a job stanza if you wish to override *Trilogy's* default behaviour.

More information on the `UseTTY` flag can be found in the *trilogy.conf Reference Guide* in the next section.

12.6 Standard Input

A server side job run by *Trilogy Server* as a result of a `Program=` directive, takes its standard input from *Trilogy Server* itself. The source of this standard input varies depending on settings within the server-side `trilogy.conf` file and/or what is being fed to the standard input of the *Trilogy Client*.

Unless otherwise directed, any standard input passed to the *Trilogy Client* will be passed verbatim over the network and be fed to the standard input of the server-side job. The `stdin` directive in the server-side `trilogy.conf` file can override this, forcing input for a particular job to be taken from a file (either fixed or via a file-chooser dialog). Note, that this only applies to jobs launched as a result of the `Program` directive (user has clicked OK on the client-side dialog or there was no dialog to display). Any server-side scripts invoked to create drop-down lists, populate dialogs or listboxes or validate/prevalidate dialogs are passed no standard input.

There are a number of ways of setting the standard input stream:

Pipes

On Both Windows and Unix clients, data can be “piped” into the standard input of the *Trilogy* command-line client.

For example:

Windows:

```
type myfile.txt | trilogy copyfile
```

Unix/Linux:

```
cat myfile.txt | trilogy copyfile
```

In both these cases, the contents of the file “myfile.txt” is passed to the standard input of the *Trilogy* client and thence onto the script defined by the `Program` directive in the server-side `trilogy.conf` file.

With the `-i` command line switch.

On both Windows and Unix clients, the `-i <filename>` switch opens the specified file and passes its content to the standard input of the server-side job.

When invoked in this way, the server-side environment variable `TRISTDINFILENAME` is set to equal the filename of the client-side file specified with the `-i` switch. The server-side job can then read this environment variable to determine the filename of the client file.



Windows Server Scripts running VBScript or JScript can also use the `GetInputFilename` method of the *Trilogy Scripting Engine* to get this value.

Using a filechooser

Both Windows and Unix clients can be set to prompt the user to select a file using a file chooser dialog. To configure this behaviour, include the following directive in the server-side `trilogy.conf` file for the desired job:

```
Stdin=Filechooser
```

When the job is started (after any dialog has been presented and submitted by clicking OK) a file chooser dialog is opened. The user can then select a file to be opened and passed to the standard-input of the server side job.

When invoked in this way, the server-side environment variable `TRISTDINFILENAME` is set to equal the filename of the client-side file chosen by the file chooser. The server-side job can then read this environment variable to determine the filename of the client file.



Windows Server Scripts running VBScript or JScript can also use the `GetInputFilename` method of the *Trilogy Scripting Engine* to get this value.

Trilogy Scripting Engine

The methods `OpenFile`, `ChooseFile`, `SetStandardInput` and `SetStream` can be used by Windows Clients running VBScript or JScript to open a file and pass it to the standard input of the server-side script.

When invoked in this way, the server-side environment variable `TRISTDINFILENAME` is set to equal the filename of the client-side file specified by the client-side script. The server-side job can then read this environment variable to determine the filename of the client file.



Windows Server Scripts running VBScript or JScript can also use the `GetInputFilename` method of the *Trilogy Scripting Engine* to get this value.

12.7 Receiving Standard Input

Server side scripts are passed the standard input from the client.

Unix servers can access this stream by means of a simple redirect:

```
> /tmp/${basename "$STDINFILENAME"}
```

This will save the standard input to the same filename as that chosen on the client.

When writing server-side scripts for Windows, you can use the “tee” command that ships with *Trilogy Server for Windows* to provide similar functionality.

The “tee” command



tee is only available on Trilogy Servers running on the Windows platform. Unix Servers have “tee” available as part of the Operating System.

tee has the following options:

```
tee [ -a ] [<filename>]
```

With no parameters, tee simply copies its standard input (supplied by *Trilogy Server*) to its standard output.

With an optional filename, tee copies its standard input to the named file.

The -a flag specifies ASCII transfer mode. In this mode, text files sent from Unix or Linux clients have their line endings translated automatically by tee such that LF line endings are translated into CR-LF for Windows platforms.



Do not specify the -a flag if the standard input data is binary.

12.8 Listing Running Jobs

Trilogy Server maintains a list of all the jobs that are running under its control. If you have access to the *Trilogy Server*, you can ask it to list all the jobs that are currently running.

To do this, enter:

```
trilogyserver -showjobs
```

This will produce a list showing the *Trilogy* Job Name, the user who started it, the total amount of time it has been running and the process id.

If the TRIEASON environment variable is set to LISTBOX then the output is a Comma-Separated-Value list. This means it is easy to create a *Trilogy* Job to show this output on a client machine inside a list box.

You can use the process ID to stop a server-side job.

12.9 Stopping Server-Side Jobs

If the `Stdout` or `Stderr` is set to `Report` and `AllowStop` is not "N", then the report window opened at the client includes a "Stop" button. If the user clicks this button, then the server-side job – and any children it has launched – is terminated.

A user with access to the Trilogy Server can also terminate any running job by running `trilogyserver -killjob <process id>`. The process id can be found by running `trilogyserver -showjobs` as outlined above.

The way the running job is terminated differs between Windows, Unix and Linux Servers.

12.9.1 Unix/Linux Servers

When a server-side job is launched it is set to be head of a Process Group. Should the client terminate the job, then `SIGINTR` (-2) is sent to all members of the Process Group followed 2 seconds later by `SIGKILL` (-9). This will result in the launched process – and any descendants – being terminated.

12.9.2 Windows Servers

A snapshot is taken of the process list and the list is traversed recursively in order to capture the invoked process itself and all its descendants. Any process with no child process is terminated first (Using `TerminateProcess`) followed by the process that launched that process and so on until the invoked process itself is terminated. In this way, all processes invoked as a result of the Server Side job are terminated.

12.10 Server Side Scripts – Environment Variables set by Trilogy

In addition to `TRIFIELD1` – `TRIFIELDn` which represent the contents of the *Trilogy* dialog, *Trilogy* also makes other environment variables available to the invoked server-side script. These are listed below:

TRIJOBNAME or TRIDIALOGNAME	Indicates the <i>Trilogy Name</i> used to invoke the <i>Trilogy</i> client. For example, if the client was invoked as: <pre>trilogy mydialog "p1"</pre> then <code>TRIJOBNAME</code> would be set to <code>"mydialog"</code> . This can be used by validation and target scripts to determine how they were invoked. For example, you can use different dialogs and yet invoke the same server-side script. The script can then use this variable to determine its course of action. Note, <code>TRIDIALOGNAME</code> is also set to maintain backward compatibility with earlier versions of <i>Trilogy</i> . Its use is deprecated.												
TRICLIENTNODENAME	The name of the client node (hostname) from which the request was issued.												
TRICLIENTUSERNAME	The login name of the user who is issuing the request.												
TRIReason	Gives the reason why <i>Trilogy</i> has invoked the script. Is set to one of: <table><tr><td>PREVALIDATE</td><td>Script has been run as a result of a <code>PreValidateWith=</code> directive.</td></tr><tr><td>VALIDATION</td><td>Script has been run as a result of a <code>ValidateWith=</code> directive.</td></tr><tr><td>POPULATE</td><td>Script has been run either as a result of a <code>PopulateWith=</code> or a <code>PopulateFieldnWith=</code> directive.</td></tr><tr><td>SCRIPT</td><td>Script has been run as a result of a <code>Program</code> directive.</td></tr><tr><td>LISTBOX</td><td>Script has been run as a result of the <code>ListBoxScript=</code> (or <code>PopulateListBoxWith=</code>) directive.</td></tr><tr><td>DIALOG</td><td>Script has been run in order to create a dialog definition.</td></tr></table>	PREVALIDATE	Script has been run as a result of a <code>PreValidateWith=</code> directive.	VALIDATION	Script has been run as a result of a <code>ValidateWith=</code> directive.	POPULATE	Script has been run either as a result of a <code>PopulateWith=</code> or a <code>PopulateFieldnWith=</code> directive.	SCRIPT	Script has been run as a result of a <code>Program</code> directive.	LISTBOX	Script has been run as a result of the <code>ListBoxScript=</code> (or <code>PopulateListBoxWith=</code>) directive.	DIALOG	Script has been run in order to create a dialog definition.
PREVALIDATE	Script has been run as a result of a <code>PreValidateWith=</code> directive.												
VALIDATION	Script has been run as a result of a <code>ValidateWith=</code> directive.												
POPULATE	Script has been run either as a result of a <code>PopulateWith=</code> or a <code>PopulateFieldnWith=</code> directive.												
SCRIPT	Script has been run as a result of a <code>Program</code> directive.												
LISTBOX	Script has been run as a result of the <code>ListBoxScript=</code> (or <code>PopulateListBoxWith=</code>) directive.												
DIALOG	Script has been run in order to create a dialog definition.												

	<p>TIMED</p> <p>Script has been run by the <i>Trilogy</i> Scheduler (AutoRun=yes)</p>										
	<p>The invoked script can use this variable to determine the actions it should perform. Using this field and TRICURRENTFIELD (below) you can use a single server script to perform all the validation, population and actions required by your <i>Trilogy</i> client-side dialog.</p>										
TRILISTBOXREASON	<p>Gives the reason why <i>Trilogy</i> is running the listbox script. When TRIREASON is set to LISTBOX, TRILISTBOXREASON is set to one of:</p> <table> <tr> <td>APPLY</td><td>Apply button has been clicked.</td></tr> <tr> <td>FIELDCHANGED</td><td>A field has been changed that is linked to the listbox. Used as a result of an OnFieldChangeUpdate={LB} clause.</td></tr> <tr> <td>REFRESH</td><td>Listbox is being refreshed automatically (AutoRefresh)</td></tr> <tr> <td>POSTRC</td><td>Listbox is being run following the execution of a right-click job.</td></tr> <tr> <td>DBLCLICK</td><td>User has double-clicked on an entry in the list-box.</td></tr> </table> <p>The invoked script can use this variable to determine how to populate the listbox.</p>	APPLY	Apply button has been clicked.	FIELDCHANGED	A field has been changed that is linked to the listbox. Used as a result of an OnFieldChangeUpdate={LB} clause.	REFRESH	Listbox is being refreshed automatically (AutoRefresh)	POSTRC	Listbox is being run following the execution of a right-click job.	DBLCLICK	User has double-clicked on an entry in the list-box.
APPLY	Apply button has been clicked.										
FIELDCHANGED	A field has been changed that is linked to the listbox. Used as a result of an OnFieldChangeUpdate={LB} clause.										
REFRESH	Listbox is being refreshed automatically (AutoRefresh)										
POSTRC	Listbox is being run following the execution of a right-click job.										
DBLCLICK	User has double-clicked on an entry in the list-box.										
TRILISTBOXSELECTIONS	The number of rows selected in any list box.										
TRICURRENTFIELD	<p>Set to the "current field" when <i>Trilogy</i> is running the script as a result of a PopulateWith= or PopulateFieldnWith= directive. In the former case, TRICURRENTFIELD is set to 0 indicating that the entire dialog is being pre-populated. In the latter case, TRICURRENTFIELD is set to the field number that <i>Trilogy</i> is wishing the script to provide values for.</p>										
TRICHANGEDFIELD	<p>The field number that has changed thereby triggering a rerun of the field populate script. Used as a result of an OnFieldChangeUpdate={} clause.</p>										
TRISTDINFILENAME	<p>The name of the file on the client from which standard input is being taken (provided it has been specified with the -i option)</p>										

TRIPARENTJOB	Script has been run as a dependent job by the <i>Trilogy Scheduler</i> – this variable contains the <i>Trilogy</i> Job Name of the “parent” job that was run.
TRIPARENTEXITCODE	Script has been run as a dependent job by the <i>Trilogy Scheduler</i> – this variable contains the Exit Code of the “parent” job that was run.
TRIPARENTLOG	Script has been run as a dependent job by the <i>Trilogy Scheduler</i> – this variable contains the file name of the log file containing the “parent” job standard output and error streams.

12.11 Running Jobs in Background

You may wish to have a server-side job take place "in background" without holding up your calling session. In this way, the job can continue to run on the server whilst the client is freed for other purposes. Upon completion, the client can be notified by a pop-up dialog.

The *Trilogy* command-line client gives you a simple option to support asynchronous execution of server-side jobs. By including the `-bg` (background) flag on the command line, the *Trilogy* Client will exit immediately with a 0 exit code (success) and will then continue to operate "in background". When the server-side task completes, its standard output and standard error will be routed back to the client where it can be displayed as a pop-up dialog box.

13 trilogy.conf – Reference Guide

This section details each entry in the server-side `trilogy.conf` file that controls the appearance of the client-side dialog and which controls how server-side jobs are executed on response from the client.

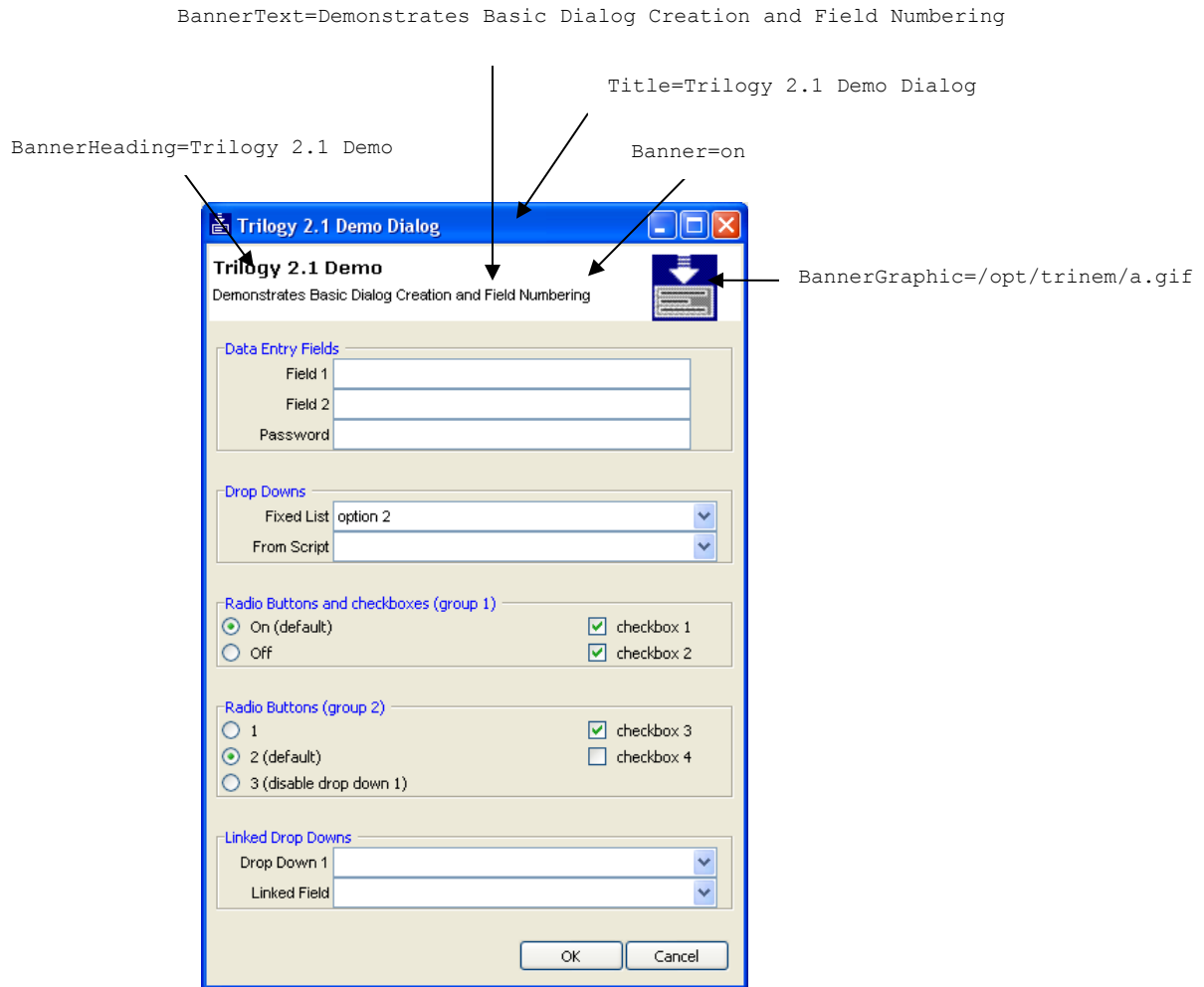


Figure 9.1: Anatomy of a Dialog

Logfile

Syntax:

```
Logfile=<path to logfile>
```

Description:

This entry is set at the global level in the server-side `trilogy.conf` (i.e.: outside of a job stanza entry). Its purpose is to enable the logging of server-side activity information and – if enabled – to specify the location and filename of the log file.

Special environment variables are set which are expanded automatically whenever the logfile is written. By constructing the logfile name with these variables, it is possible to automatically create a new log file every day or every month, or have a fixed log file capturing all output.

For example:

```
Logfile=$TRILOGYHOME/log/$YYYY$MM$DD.log
```

Will create a new log file for every day of activity.

```
Logfile=$TRILOGYHOME/log/Trilogy_$DD.log
```

Will create a logfile called `Trilogy_01.log` on the 1st of the month, `Trilogy_31.log` on the 31st of the month.

Log file output contains server-side script runs, requests from clients as well as debug and failure indications. It can be very useful, when trying to track a problem with a *Trilogy* job, to look at the log file. However, switching logging on will inevitably blunt performance. It is probably best to leave logging off and only switch it on when trying to trace a problem.

Environment Variables:

\$YYYY	4-digit current year
\$YY	2-digit current year
\$MM	2-digit month (01-12)
\$DD	2-digit day of month (01-31)

Port

Syntax:

```
Port=<Port Number>
```

Description:

This entry is set at the global level in the server-side `trilogy.conf` (i.e.: outside of a job stanza entry). It specifies the port number on which the *Trilogy* Server should listen for connection requests.

For example:

```
Port=2301
```

Means that the Trilogy Server will listen for incoming connections on port 2301.

RelayServer

Syntax:

```
RelayServer=<ServerName>
```

Description:

This entry is set at the global level in the server-side `trilogy.conf` (i.e.: outside of a job stanza entry). It specifies the hostname of a *Trilogy* Server which is being used to route notification messages to clients.

For example:

```
RelayServer=remotehost.client.net  
RelayPort=2301
```

Means that any notification messages that the *Trilogy* Server would normally send directly to client machine(s) (for example, balloon-style notifications) are, instead routed to the *Trilogy* Server hosted on `remotehost.client.net` which is listening on port 2301.

This directive is useful when the *Trilogy* Server and the *Trilogy* Clients are located on different subnets where the clients are not directly addressable from the server. In this case, routing messages to clients has to take place via an intermediate server.

See also:

`RelayPort`, *Chapter 9: "Windows Client Service"*

RelayPort

Syntax:

```
RelayPort=<Port Number>
```

Description:

This entry is set at the global level in the server-side `trilogy.conf` (i.e.: outside of a job stanza entry). It specifies the port number of a *Trilogy* Server which is being used to route notification messages to clients.

For example:

```
RelayServer=remotehost.client.net  
RelayPort=2301
```

Means that any notification messages that the *Trilogy* Server would normally send directly to client machine(s) (for example, balloon-style notifications) are, instead routed to the *Trilogy* Server hosted on `remotehost.client.net` which is listening on port 2301.

This directive is useful when the *Trilogy* Server and the *Trilogy* Clients are located on different subnets where the clients are not directly addressable from the server. In this case, routing messages to clients has to take place via an intermediate server.

See also:

`RelayServer`, *Chapter 9: "Windows Client Service"*

AllowIfJobRunning

Syntax:

```
AllowIfJobRunning=jobname[,jobname...]
```

Description:

Set in a job stanza, this directive specifies that the job can only be executed provided that one of the specified *Trilogy* Jobs is currently running. If none of the listed jobs are currently executing then access to the job is denied.

Note, if access to the job is denied then the job will not be presented if a user right-clicks on the *Trilogy Client Service* icon, even if the `Traymenu` directive is set to `Yes` and the user is in the appropriate user group.

See also:

`DenyIfJobRunning`, `Traymenu`

AutoRefresh

Syntax:

`AutoRefresh=on|off|n`

Description:

Specifies than an automatic refresh applies to the List Box. Whenever an automatic refresh occurs, the server-side script that populates the list box is rerun and the client-side list box is regenerated with no client user interaction.

By default, no automatic refresh takes place. The List Box script is run only when the dialog is first displayed, when the Apply button is clicked or when a linked field is changed (depending on the value of `PopulateListBox` and any `OnFieldChange``nUpdate` directives).

Included in a Job Stanza, the `AutoRefresh` directive tells *Trilogy* to rerun the server-side `ListBoxScript` periodically.

If it is set to off (the default) no automatic refresh takes place.

If it is set to on then an automatic refresh takes place every 5 seconds.

If it is set to *n* (where *n* is an integer number) then the refresh takes place every *n* seconds.

Any selections or column sorts are retained following an `AutoRefresh`.

See also:

`ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`,
`ColumnNames`, `ColumnWidths`, `OnRightClick`, `AutoSort`,
Chapter 5 "The List Box"

AutoRun

Syntax:

AutoRun=yes|no

Description:

Specifies that the job should be run under control of the *Trilogy Scheduler*.

The `AutoRun` directive defaults to `no`. If this directive is not specified, then the job can only be invoked by a *Trilogy Client*. Other `AutoRun` directives are ignored if this directive is absent or is set to `no`.

If this directive is set to `yes`, then the job can be started by the *Trilogy Scheduler*. In this case, the directive `AutoRunTimes` also needs to be specified in the same job stanza to give the times at which the job should be run.

The other `AutoRun` directives are optional and – if specified – serve to further restrict when the job will be run.

See also:

`AutoRunTimes`, `AutoRunDates`, `AutoRunInterval`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunTimes

Syntax:

`AutoRunTimes=hh:mm[,hh:mm ...] | [hh:mm-hh:mm]`

Description:

Specifies the times of day that a job should run when `AutoRun` is set to `yes`.

`AutoRunTimes` can specify a single time, a group of times (separated by commas) or a range of times. If a range is given, then the directive `AutoRunInterval` also needs to be set.

Times are specified in 24 hour format, with the hours and minutes separated by a colon. For example 03:27 is 3:27 am, 15:30 is 3:30 pm.

See also:

`AutoRun`, `AutoRunInterval`, `AutoRunDates`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunInterval

Syntax:

`AutoRunInterval=n`

Description:

Specifies the number of minutes that should elapse between each automatic run of the job. The interval is applied between the start and end times given by the `AutoRunTimes` directive.

If `AutoRunTimes` specifies a range of values (`AutoRunTimes=hh:mm-hh:mm`) then `AutoRunInterval` is a required directive.

If `AutoRunTimes` specifies a single time or a group of individual times, then `AutoRunInterval` is ignored.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunDates`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunDates

Syntax:

`AutoRunDates=dom [,dom ...]`

Description:

Specifies the day of the month on which the job should be run. Dates are specified as a number from 1 – 31.

This directive is ignored unless `AutoRun=yes`.

This directive needs to be specified along with `AutoRunTimes`. When specified, the job will run at the times specified with `AutoRunTimes` but only when the day of the month matches that given by `AutoRunDates`.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunDays

Syntax:

```
AutoRunDays=day[-day] [,day[-day] ...]
```

Description:

Specifies the day of the week on which the job should be run. Days are specified with the common English abbreviation (Mon, Tue, Wed, Thu, Fri, Sat, Sun) or as a number (Monday = 1 ... Sunday = 7)

This directive is ignored unless `AutoRun=yes`.

This directive needs to be specified along with `AutoRunTimes`. When specified, the job will run at the times specified with `AutoRunTimes` but only when the day of the week matches one of the days that given by `AutoRunDays`.

This directive can be specified along with `AutoRunDates` to restrict the days on which the job will run to specified dates (for example, `AutoRunDays=Mon` and `AutoRunDates=1-7` will cause the job to run on the first Monday of the month).

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDates`, `AutoRunMonths`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunMonths

Syntax:

```
AutoRunMonths=month[-month] [,month[-month] ...]
```

Description:

Specifies the month of the year on which the job should be run. Months are specified with the common English abbreviation (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec) or as a number (January = 1 ... December = 12)

This directive is ignored unless `AutoRun=yes`.

This directive needs to be specified along with `AutoRunTimes`. When specified, the job will run at the times specified with `AutoRunTimes` but only when the month of the year matches one of the days that given by `AutoRunMonths`.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDates`, `AutoRunDays`, `AutoRunOnSuccess`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunStandardInput

Syntax:

```
AutoRunStandardInput=<filename>
```

Description:

Specifies a server-side file that is to be used for the standard input of the job when it runs under the control of the scheduler.

If this directive is not specified then the job has no standard input.

Any environment variables specified in the filename are not expanded until the job is being executed. Therefore, you can include *Trilogy*-generated environment variables such as `$TRIPARENTLOGFILE` so that a dependent job can read its parent's standard output.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDates`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunOnSuccess

Syntax:

```
AutoRunOnSuccess=job [,job ...]
```

Description:

Specifies job(s) to be started should the script specified by the `program` directive for this job exit with a successful exit code (exit code = 0).

This directive has no effect if the *Trilogy* job is launched by a *Trilogy* Client. It only affects jobs launched by the *Trilogy* Scheduler.

If more than one job is specified, then each job is launched in parallel.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDates`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnFailure`, *Chapter 10 "The Scheduler"*

AutoRunOnFailure

Syntax:

```
AutoRunOnFailure=job [,job ...]
```

Description:

Specifies job(s) to be started should the script specified by the `program` directive for this job exit with a non-successful exit code (exit code `!= 0`).

This directive has no effect if the *Trilogy* job is launched by a *Trilogy* Client. It only affects jobs launched by the *Trilogy* Scheduler.

If more than one job is specified, then each job is launched in parallel.

See also:

`AutoRun`, `AutoRunTimes`, `AutoRunInterval`, `AutoRunDates`, `AutoRunDays`, `AutoRunMonths`, `AutoRunOnSuccess`, *Chapter 10 "The Scheduler"*

AutoSort

Syntax:

`AutoSort=[-]ColumnNumber`

Description:

Specifies that an automatic sort should be applied to a List Box whenever it is populated.

By default, no sort takes place. When a List Box is populated (by the `ListBoxScript`) then the rows are displayed in the order they were output by the server-side script.

Included in a Job Stanza, the `AutoSort` directive forces *Trilogy* into applying a sort on the List Box after it is populated. This is functionally identical to the user clicking on the appropriate column heading after the List Box has been populated, except that it is done automatically.

Columns are numbered from 1. Hidden columns are counted. It is possible to automatically sort by a hidden column which is not possible manually.

The sort happens in ascending mode, unless a negative `ColumnNumber` is given to the `AutoSort` directive (the `ColumnNumber` starts with a - character). In this case the sort happens in descending mode.

See also:

`ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`,
`ColumnNames`, `ColumnWidths`, `OnRightClick`, *Chapter 5 "The List Box"*

AutoSelectColumn

Syntax:

`AutoSelectColumn=ColumnNumber`

Description:

Controls if and how rows are automatically selected when the list box is populated.

If this directive is not specified then no rows are selected when the list box is populated. This is the default.

If this directive is specified then it gives the number of a column in which to look for specific values. These values are specified by the `AutoSelectValue` directive. If `AutoSelectValue` is not set, then this value defaults to "Y".

Any row which has a value in the specified column that matches that specified by `AutoSelectValue` is automatically selected.

Note, that this selection happens independently of the `Selections` option. Thus, `AutoSelectColumn` can select multiple rows, even if `Selections` is set to `Single` or `None`. In the latter case, this prevents the user from changing the selections made by `AutoSelectColumn`.

See also:

`AutoSelectValue`, `AutoSort`, `ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`, `ColumnNames`, `ColumnWidths`, `OnRightClick`,
Chapter 5 "The List Box"

AutoSelectValue

Syntax:

```
AutoSelectValue=val  
AutoSelectValues={val1,val2...}  
AutoSelectValue=fromn-ton
```

Description:

Works with the `AutoSelectColumn` directive in order to control if and how rows are automatically selected when the list box is populated.

This directive is ignored unless `AutoSelectColumn` is specified in the same job definition.

If this directive is not specified then it defaults to "Y".

The `AutoSelectValue` directive specifies which value(s) should be present in the column specified by the `AutoSelectColumn` directive in order for the row to be automatically selected when the list box is populated. These values can be specified in one of three ways:

- As a discreet value: `AutoSelectValue=val1`
- As a set of discreet values: `AutoSelectValue={val1,val2}`
- As a range of numbers (from low number to high number). This only works with numeric values in the specified column:
`AutoSelectValue=12-15`

See also:

`AutoSelectColumn`, `AutoSort`, `ListBoxScript`, `PopulateListBox`,
`ApplyButton`, `ListBoxSep`, `ColumnNames`, `ColumnWidths`, `OnRightClick`,
Chapter 5 "The List Box"

AutoStretch

Syntax:

`AutoStretch=on|off`

Description:

Controls how the dialog is sized with relation to the list box (if the list box is included in the dialog).

If `Autostretch` is set to `on` then the dialog will grow automatically so that its width is the same as the width of the list box (i.e.: the sum of all the widths of the columns in the list box). This is the default. If no `AutoStretch` directive is included in the Job Stanza then this is the behaviour that the dialog will exhibit when it is displayed at the client.

If `Autostretch` is set to `off` then the list box's width will be bounded to the size of the parent dialog. A horizontal scroll bar will be presented if necessary. In this case the dialog's width is controlled by the text in the `Dialog` file (or the lines generated by the `DialogScript`).

See also:

`ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`,
`ColumnNames`, `ColumnWidths`, `OnRightClick`, *Chapter 5 "The List Box"*

Banner

Syntax:

`Banner=on|off`

Description:

Controls the appearance of the graphic banner at the top of the client dialog. This can be set at a global level (i.e.: outside of a job stanza entry) in which case the setting applies to all jobs which do not stipulate a value for Banner. In other words, setting `Banner=on` at a global level will mean all jobs that do not stipulate `Banner=off` will have the banner presented on the dialog. Setting `Banner=off` at the global level will mean all jobs that do not stipulate `Banner=on` will have no banner presented on the dialog.

The banner consists of a Banner Heading, Banner Text and a Banner Graphic (gif image). The GIF image will default to the Trilogy default icon, the Banner Heading will default to be the same as the job `Title` and the Banner Text will be clear. For the best effects, these settings should be specified for any job which has a Banner.

See *Figure 9.1: Anatomy of a Dialog* above to see the demo dialog and how the banner is constructed from the various components.

See also:

`BannerHeading`, `BannerText`, `BannerGraphic`.

BannerHeading

Syntax:

```
BannerHeading=<heading text>
```

Description:

Set in a job stanza, this indicates the text to be displayed on line 1 of the Banner.

Note: the banner must be `on` for the Banner Heading to be displayed.

See also:

`Banner`, `BannerText`, `BannerGraphic`

BannerText

Syntax:

```
BannerText=<text>
```

Description:

Set in a job stanza, this indicates the text to be displayed on line 2 of the Banner.

Note: the banner must be `on` for the Banner Text to be displayed.

See also:

`Banner`, `BannerHeading`, `BannerGraphic`

BannerGraphic

Syntax:

```
BannerGraphic=<path to GIF file>
```

Description:

Set in a job stanza, this specifies a path to a GIF file (on the server) which will be displayed in the right hand side of the banner whenever the dialog is displayed at the client.

In order for the image to be displayed correctly in the client banner, the following conditions must be met:

- The file exists on the server and is readable by the *Trilogy* Server.
- The file is in GIF format (GIF87a or GIF89a)
- The image is less than or equal to 200 pixels (width) x 52 pixels (height)

If all these conditions are met then the image is included in the client-side dialog. If any of the conditions are not met, then a warning message is written to the *Trilogy* log file (if logging is enabled), and the image in the banner reverts to the *Trilogy* default icon.

Note: the banner must be `on` for the graphic to be displayed.

See also:

`Banner`, `BannerText`, `BannerHeading`, `LogFile`

CancelButtonText

Syntax:

```
CancelButtonText=<text>
```

Description:

Defaults to "Cancel". Specifies what text should appear on the Cancel Button. When this button is clicked the dialog exits with no further action.

See Also:

`ApplyButtonText`, `OKButtonText`

ContextColumn

Syntax:

`ContextColumn=Column Number`

Description:

Specifies the number of the column in the List Box which is to be used to identify the context for the row.

Contexts can be named for both `Icon` and `OnRightClick` directives. Identifying a Context Column allows a list box row to display different Icon Decorators and/or different right-click menus depending on the row content.

`ContextColumn` defaults to 0 (no context). If set, it identifies the column from which to read the row context.

See Also:

`Icon`, `OnRightClick`, *Chapter 5: "The List Box"*.

DenyIfJobRunning

Syntax:

```
DenyIfJobRunning=jobname[,jobname...]
```

Description:

Set in a job stanza, this directive specifies that the job can be run only if none of the *Trilogy* jobs named in the parameter are currently running. If any of the named jobs is running, then access to the job is denied.

By specifying its own job name, this directive can be used to prevent a job from being executed more than once at any one time.

Note, if access to the job is denied then the job will not be presented if a user right-clicks on the *Trilogy Client Service* icon, even if the `TrayMenu` directive is set to `Yes` and the user is in the appropriate user group.

See also:

`AllowIfJobRunning`. `TrayMenu`

Dialog

Syntax:

```
Dialog=<path to Dialog File>
```

Description:

Specifies a path to a file on the server containing the dialog definition for the job. If this entry is not present in a job stanza, then a client invoking this job will result in the immediate execution of the server-side program/script identified by the `Program` directive. If a `Dialog` directive *is* present, then a client invoking the job will be presented with a dialog constructed at run-time from the contents of this server-side dialog file.

Section 4.2 gives more information on how the dialog file is constructed.

See also:

`Banner`, `DialogScript`, `PopulateDialog`, `PopulateFieldnWith`,
`OnFieldChangeUpdate`

DialogScript

Syntax:

```
DialogScript=<path to executable>
```

Description:

Specifies the path to a server-side script whose standard output will be used to construct the client-side dialog. The output from this script will be taken as a dialog definition file and *Trilogy* will proceed with presenting the dialog at the client, just as if a fixed `Dialog` definition file had been stipulated for the job.

The advantage of using `DialogScript` is that the script invoked is passed all the parameters passed to the *Trilogy* client (as are all *Trilogy* Server Scripts) as well as any additional command line parameters specified for the job. It is therefore possible to create dialogs whose content and layout change dependent on the passed parameters.

See also:

`Dialog`

DoubleClick

Syntax:

DoubleClick=off|on|yes|no

Description:

Specifies whether a user is allowed to double-click entries within the List Box (if displayed). This value defaults to off/no (no Double Click functionality is provided). If set to yes (or on), then a user can double-click on rows within the displayed list box. When this is done, the server-side list box script is run and the list box is repopulated at the client.

The list box script can use the TRI_ variables to establish which row(s) have been selected and double-clicked. It can then generate a new list accordingly.

See also:

ListBoxScript, PopulateListBox, ApplyButton, ListBoxSep, ColumnNames, ColumnWidths, OnRightClick, *Chapter 5 "The List Box"*

Environment

Syntax:

```
Environment=<path to environment file>
```

Description:

Specifies the path to a server-side file which contains lines in the format

```
var=name
```

Each line is read and, if it contains a variable declaration as described above, it is added to the environment of any job executed by *Trilogy* as a result of a `PreValidateWith`, `ValidateWith`, `PopulateWith`, `PopulateFieldnWith`, `ListBoxScript` or `Program` directive.

The directive can be associated with a *Trilogy* Job – in which case the environment variables contained within the file are only added to those server-side programs invoked for that Job – or it can be global (outside of a job stanza) in which case it is added to the environment of every program invoked by *Trilogy* regardless of the job.

If the environment variables referenced in the file are already in the environment then they are overwritten. Therefore, you can use this directive to enforce security by setting default PATH values etc.

See also:

`PreValidateWith`, `ValidateWith`, `PopulateWith`, `PopulateFieldnWith`, `ListBoxScript`, `Program`, *Chapter 11 "Server Side Job Control"*

Group

Syntax:

```
Group=GroupName  
or  
Groups=GroupName [,GroupName ...]
```

Description:

Specifies a user group (or list of user groups) that have access to the specified job.

If the client user (identified by their login id) is not part of the specified user group, then access to the job is denied.

Trilogy will invoke the specified GroupProcessor in order to calculate the group membership. Therefore, a Group Processor plug-in must be enabled before this directive can be used. If the "Group" directive is used without a Group Processor being specified (or the Group Processor has not been initialized properly) then access to the job will be denied for all users.

If the job also has the `TrayMenu` directive set to `Yes`, then Group Membership is taken into account when deciding which jobs are presented to the client user when they right-click on the System Tray Icon (*Trilogy Client Service*).

See also:

`GroupProcessor`, `TrayMenu`

GroupProcessor

Syntax:

GroupProcessor=<Path to Group Processor Library>

Description:

This entry is global and, if specified, points to a library (Windows DLL or Unix/Linux Shared Object or Archive) that contains a Group Processor.

Functions in the Group Processor are called when *Trilogy* needs to determine:

- If the client user is part of a user group (when trying to access jobs which have a Group restriction)
- Which users are part of a specified user group (when sending group notifications)

See also:

Group, TrayMenu, *Chapter 8: "Groups and the Group Processor"*

Icon

Syntax:

`Icon[context]=<path to GIF file>`

Description:

Specifies the full path of a GIF file which represents an icon to display in the list box. This option is ignored unless the `ListBoxIcon` directive is set to `Yes` (or `On`).

The GIF file specified should be a maximum of 16x16 pixels in order to fit into the list box row.

[`context`] is optional. If specified, it indicates that the icon should only be used for rows which belong in the specified context. The directive `ContextColumn` is used to indicate which column in the list box output contains the Context.

Any number of `Icon` directives can be included in a job stanza, provided they are in different contexts.

See Also:

`ListBoxIcon`, `ContextColumn`, *Chapter 5: "The List Box"*.

NotifyRunGroup

Syntax:

NotifyRunGroup=GroupName[,GroupName ...]

Description:

If included in a Job Stanza, this directive tells *Trilogy* to notify every client user who is

- in one or more of the specified user group(s) and
- is running the *Trilogy Client Service*

that the job is running. *Trilogy* notifies those users by animating the *Trilogy Client Icon* in the clients' system tray (Notification Area) to indicate the job is running.

When the job completes, the *Trilogy Client Icon* in the clients' system tray (Notification Area) stops animating.

This can be useful to notify users that a job (for example a build or a deployment) is in progress.

Note, *Trilogy* will invoke the Group Processor in order to determine which users to notify when the job is running. Therefore, a Group Processor plug-in must be enabled before this directive can be used. If the "NotifyRunGroup" directive is used without a Group Processor being specified (or the Group Processor has not been initialized properly) then no notification will be provided.

See also:

GroupProcessor, *Chapter 8: Groups and the Group Processor*

OkButtonText

Syntax:

```
OkButtonText=<text>
```

Description:

Defaults to "OK". Specifies what text should appear on the OK Button. When this button is clicked the server side `ValidateWith=` script is invoked first (if specified) and, if this exits with a success exit code (0) the server side `Program=` script is invoked (if specified).

See Also:

`ApplyButtonText`, `OkButtonText`

OnFieldChange n Update

Syntax:

```
OnFieldChange $n$ Update={field1, field2, field3 ...}
```

Description:

Links related fields in a dialog. The “ n ” is a number indicating the field number to be linked (e.g.: OnFieldChange4Update=...)

This directive is used to update the dialog dynamically whenever a field on the dialog changes. When the field indicated by the number n in the OnFieldChange n Update directive is changed, *Trilogy* will run the Field Populate Script(s) for each field number specified in the associated list (and any List Box Script if the field name is given as “LB”). If the fields are drop-down lists, their contents are cleared and the field populate scripts are once more run on the server to repopulate the lists.

This can be used to create dynamic dialogs. For example, you may have a drop-down list containing car manufacturers, another containing models of car and a third containing engine sizes and fuel types. The “models” and “engine size” fields are initially blank. When the user selects a manufacturer, *Trilogy* updates the second drop-down with a list of models made by that manufacturer. When a model is selected, the third drop-down contains a list of engine sizes.

The scripts are invoked in an identical way to initial population (when the client-side dialog is first created) but when the invocation is due to a field being changed, the environment variable \$TRICHANGEDFIELD is set and the contents of the dialog are made available through the environment variables \$TRIFIELD1 to \$TRIFIELD n where n is the number of fields in the dialog.

Environment Variables:

\$TIREASON	"POPULATE" / "LISTBOX"
\$TRIJOBNAME	Trilogy Job ID
\$TRICLIENTNODENAME	Host Name of the invoking client
\$TRICLIENTUSERNAME	User Name of the invoking user
\$TRICURRENTFIELD	Field number being updated
\$TRICHANGEDFIELD	n (where n is the number of the field being changed).
\$TRIFIELD1-	Dialog Content
\$TRIFIELDDn	
\$TRI_columnname_1-	List Box Selections
\$TRI_columnname_n	

See also:

Dialog, DialogScript, PopulateField n With, OnListBoxChangeUpdate,
Chapter 6 – Linking Fields

OnListBoxChangeUpdate

Syntax:

```
OnListBoxChangeUpdate={field1, field2, field3 ...}
```

Description:

Links fields on a dialog to the List Box such that when the list box content or the current list box selection changes, the corresponding fields are repopulated.

Trilogy will run the Field Populate Script(s) for each field number specified in the associated list. If the fields are drop-down lists, their contents are cleared and the field populate scripts are once more run on the server to repopulate the lists.

Environment Variables:

\$TRI_REASON	"POPULATE"
\$TRI_JOBNAME	Trilogy Job ID
\$TRI_CLIENTNODENAME	Host Name of the invoking client
\$TRI_CLIENTUSERNAME	User Name of the invoking user
\$TRI_CURRENTFIELD	Field number being updated
\$TRI_CHANGEDFIELD	"LISTBOX"
\$TRI_FIELD1-	Dialog Content
\$TRI_FIELDn	
\$TRI_columnname_1-	List Box Selections
\$TRI_columnname_n	

See also:

Dialog, DialogScript, PopulateFieldnWith, OnFieldChangeUpdate,
Chapter 6 – Linking Fields

PopulateWith

Syntax:

```
PopulateWith=<path to script> | {field1, field2, field3 ...}  
or  
PopulateDialogWith=<path to script> | {field1, field2, field3 ...}
```

Description:

Specifies the initial field values for a dialog. There are two ways in which this directive can be used. The first is to specify the path to a server-side script. In this case, the script is run and the standard output from the server-side script is used to pre-populate the client dialog. The first line of the server-side script's output is used to populate the first field of the client-side dialog, the second line of the script's output is used to populate the second field of the dialog and so on.

Any server-side script invoked to populate a dialog is passed all the parameters specified in any `Param` or `Params` directive in the job stanza and also any passed from the command line client or via the `AddParameter` method in the *Trilogy Scripting Engine*.

The second method involves specifying a fixed list of values in the format {field1, field2, field3 ...}. In this case, the value of "field1" is used to set the initial value of the first field on the client dialog, the value of "field2" the second field on the client dialog and so on.

When the field being populated is a drop-down list then the value for the field must exist as one of the drop-down options. If this is the case then the option is automatically selected from the drop-down. If the field content specified does not exist as an option in the drop-down then no selection is made and the drop-down is left as the default (non-selected).

When the field being populated is a radio button or checkbox then a value of "1" selects the field and a value of "0" clears it. If multiple radio buttons within the same frame are selected with "1", then only the last radio button retains its selected status.

Environment Variables:

\$TRIReason	"POPULATE"
\$TRIDIALOGNAME	Trilogy Job ID
\$TRICLIENTNODENAME	Host Name of the invoking client
\$TRICURRENTFIELD	"0"

See also:

`Dialog`, `DialogScript`, `PopulateFieldnWith`

PopulateFieldnWith

Syntax:

```
PopulateFieldnWith=<path to script> | {option1, option2, option3 ...}
```

Description:

Specifies the values for a drop-down field in a dialog. The "n" is a number indicating the field number to be populated (e.g.: `PopulateField4With=...`)

There are two ways in which this directive can be used. The first is to specify the path to a server-side script. In this case, the script is run and the standard output from the server-side script is used to populate the client field. The first line of the server-side script's output becomes the first option in the drop-down list, the second line of the script's output becomes the second option and so on. If the script exits with a non-zero exit code (fail), then the corresponding client field is disabled.

Any server-side script invoked to populate a field is passed all the parameters specified in any `Param` or `Params` directive in the job stanza and also any passed from the command line client or via the `AddParameter` method in the *Trilogy Scripting Engine*.

The second method involves specifying a fixed list of values in the format `{option1, option2, option3 ...}`. In this case, the value of "option1" becomes the first option in the drop-down list, the value of "option2" becomes the second option and so on.

When the field being populated is a radio button or checkbox then a value of "1" selects the field and a value of "0" clears it. If multiple radio buttons within the same frame are selected with "1", then only the last radio button retains its selected status.

Environment Variables:

<code>\$TRIReason</code>	"POPULATE"
<code>\$TRIDialogName</code>	Trilogy Job ID
<code>\$TRIClientNodeName</code>	Host Name of the invoking client
<code>\$TRICurrentField</code>	<i>n</i> (where <i>n</i> is the number of the field being populated).
<code>\$TRICHangedField</code>	NULL for Initial Dialog population, otherwise the number of the field that changed to trigger this update – see <code>OnFieldnUpdate</code> .
<code>\$TRIField1-</code> <code>\$TRIFieldn</code>	All NULL on Initial Dialog population, otherwise Dialog Content - see <code>OnFieldnUpdate</code> .

See also:

`Dialog`, `DialogScript`, `PopulateWith`, `OnFieldnUpdate`

Selections

Syntax:

`Selections=Multiple|Single|None`

Description:

Controls how many rows the user is allowed to select from the list box (if the client dialog includes a list box)

Multiple is the default. If `Selections` is not specified, then the list box defaults to this behaviour. Multiple allows one or more rows to be selected from the list box.

Single specifies that only one row at a time can be selected in the list box.

None specifies that no rows can be selected in the list box.

See also:

`ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`,
`ColumnNames`, `ColumnWidths`, `OnRightClick`, *Chapter 5 "The List Box"*

SystemTray

Syntax:

`SystemTray=Yes|No|On|Off|Hold`

Description:

Specifies that the associated job should cause Windows Clients to create an icon in the System Tray (Notification Area) whilst the job is running.

A Windows Client requires an icon in the System Tray in order to receive *Balloon Style* notification messages. If your server-side job needs to send such Balloon notifications to the invoking client (and the client is not running the *Trilogy Client Service*) then you will need this directive in order to create an icon when the job is running.

The icon will animate whilst the server-side job is running. What happens when the server side job completes depends on the value of `SystemTray`:

If the `SystemTray` parameter is set to `Yes` (or `On`) then the icon will disappear automatically when the server-side job completes (after the last balloon notification has been displayed).

If the `SystemTray` parameter is set to `Hold` then the icon will stop animating and will receive a decorator appropriate to the exit status of the server-side job – either a green tick (job exited with code 0) or a red exclamation mark (job exited with a non-zero exit code). The icon will only disappear when the user right-clicks on the icon and selects “Quit”.

Note, if the client is running the *Trilogy Client Service*, then the icon will already be present in the System Tray. When the job completes, the icon will remain in the System Tray.

The presence of this directive also changes the behaviour of any Report Window. If `Stdout` or `Stderr` is set to “Report”, then the Report Window is not displayed immediately the job starts. Instead, it starts iconized to the System Tray. The client user has to double-click on the *Trilogy* Icon in the System Tray in order to open the Report Window. Similarly, iconizing the Report Window removes it from the display and the Task Bar – effectively it iconizes to the System Tray.

Note, Unix and Linux Clients silently ignore this directive.

See also:

`Stdout`, `Stderr`, *Chapter 9 “Trilogy Client Service for Windows”*

Title

Syntax:

```
Title=<title>
```

Description:

Specifies the title to be displayed in the window's title bar at the top of the dialog. Defaults to "trilogy". Also used as a menu title should the job be available via a right-click in a List Box or when invoked from the *Trilogy Client Service* in the System Tray or Notification Area.

See also:

Banner, Dialog, DialogScript, OnRightClick, TrayMenu, *Chapter 5 "The List Box"*, *Chapter 9 "Trilogy Client Service for Windows"*

TrayMenu

Syntax:

TrayMenu=Yes|No|On|off

Description:

Specifies that the associated job should be available to users of the Windows *Trilogy Client Service*.

Windows clients running the *Trilogy Client Service* will have a *Trilogy* Icon displayed in their System Tray (Notification Area). Right Clicking on this icon will present a list of *Trilogy* Jobs that they can invoke.

The list of jobs available to the user is built as follows:

- Only Jobs with TrayMenu=Yes (or TrayMenu=On) are included
- If the Job has a Group= directive then the Group Processor is invoked in order to determine user membership of the group. If the client user (determined by their login id) is not in the specified group, then access to the job is denied and it is not listed.
- If the job has an AllowIfJobRunning directive then the list of associated jobs is checked. If none of the specified jobs is currently running, then access to the job is denied and it is not listed.
- If the job has a DenyIfJobRunning directive then the list of associated jobs is checked. If any of the specified jobs is currently running, then access to the job is denied and it is not listed.

See also:

AllowIfJobRunning, DenyIfJobRunning, Title, Group, GroupProcessor, Chapter 8 "Groups and the Group Processor", Chapter 9 "Trilogy Client Service for Windows"

ListBox

Syntax:

`ListBox=on|off|yes|no|auto`

Description:

Specifies whether a ListBox is to be displayed at the bottom of the client dialog. If set to on/yes then the listbox is displayed. If set to off/no (the default) then no list box is displayed.

The "auto" setting (`ListBox=auto`) displays the listbox only if the listbox script's standard output contains some data. If output is produced, the listbox is displayed and populated with the standard output from the listbox script. If the listbox script produces no output then the listbox is not displayed.

See also:

`ListBoxScript`, `PopulateListBox`, `ApplyButton`, `ListBoxSep`, `ColumnNames`, `ColumnWidths`, `OnRightClick`, *Chapter 5 "The List Box"*

OnRightClick

Syntax:

```
OnRightClick[context]={jobname,jobname...}
```

Description:

Specifies a list of *Trilogy* jobnames to be listed whenever a selection in a listbox row is right-clicked.

If a list box is displayed and no `OnRightClick` directive is present in the job stanza, then right-click functionality is disabled at the client. If an `OnRightClick` is present in the job stanza, then - when a user right-clicks on a selection in the list box - a pop-up menu is presented. The menu options are taken from the `Title` directives of the specified *trilogy* jobs.

Only jobs which are allowed to be run by the invoking user are displayed in the right-click menu (see *Group Processor*).

Should a user select one of the jobs from this menu, then the selected job is executed in the normal *Trilogy* manner. The original job (along with its dialog and list box) becomes the "primary" job; the invoked job becomes the secondary job. If a `Dialog` directive is present, then this secondary dialog is displayed, any secondary `PrevalidateWith` and `ValidateWith` scripts are run and - finally - the secondary program specified by the `Program` directive is invoked. The standard output and standard error streams from this invoked secondary program are handled according to the `Stdout` and `Stderr` entries for the secondary job and not for the primary (invoking) job.

In order for the secondary job to establish which entries have been selected in the primary listbox, environment variables are set. The names of these environment variables are taken from the column titles in the list box (with any spaces replaced with underscores and any lower case letters translated to upper case) along with a trailing number which indicates its selection number.

For example, if the primary job contained a list box with these columns:

```
ColumnNames={ProcessID,Parent Process,Full Path}
```

...and these right-click jobs

```
OnRightClick={KillProcess,MoreDetails}
```

...and the `KillProcess` job was defined like this:

```
KillProcess:
  Title=Kill This Process
  Program=/usr/bin/kill
  Param=-9
  Param=$TRI_PROCESSID_1
```

Then when the user right-clicks in the list box (containing the columns "ProcessID", "Parent Process" and "Full Path") they will be presented a pop-up menu containing two options, one of which will be "Kill This Process" (the menu option being taken from the Title of the secondary job `KillProcess`).

If the user selects "Kill This Process", then the secondary job `KillProcess` will run. The program `/usr/bin/kill` will be invoked with two parameters, the first being `-9` (as specified in the `Param` directive of the secondary job), the second being the process ID of the selected row in the list box. This is set in the environment variable `$TRI_PROCESSID_1` – the `TRI_` being the standard prefix, `PROCESSID` being an upper-cased representation of the column title `ProcessID` and `_1` meaning the first selected row. The other environment variables available to this secondary job are `TRI_PARENT_PROCESS_1` (`PARENT_PROCESS` being derived from the column title `Parent Process`) and `TRI_FULL_PATH_1` (`FULL_PATH` being derived from the column title `Full Path`).

When specifying this directive, the `[context]` is optional. If specified it indicates that the right-click menu only applies to rows which are in the specified context. A row is identified as being in a particular context by use of the `ContextColumn` directive which indicates which row of the List Box output contains the context name.

See Also:

`ColumnNames`, `ColumnWidths`, `ListBox`, `ListBoxScript`, `ContextColumn`, *Chapter 5 "The List Box"*.

HelpText

Syntax:

```
HelpText=<help text>
```

Description:

Specifies text to be displayed at the client should a user run `trilogy -jobs` to list the available *Trilogy* jobs on this server.

See also:

UseTTY

Syntax:

`UseTTY=yes|no|on|off`

Description:

- Unix Servers: Specifies whether the server-side program invoked when the OK button is clicked on the client-side dialog runs with an attached terminal (tty) or not.
- Windows Servers: Runs the server-side program with a console attached, reading output from the console.

Depending on the server-side operating system – and the language in which the server-side program is written – the setting of `UseTTY` can have a bearing on the output of the program and on the buffering of the standard output. For example, binary applications tend to be line-buffered if they have a controlling terminal and block-buffered if their standard output is directed to a pipe. In a similar manner, Unix programs like “ls” can detect whether they are talking to a terminal or a pipe or file and adjust their output accordingly.

The default setting of this flag depends on the value of the `stdout=` directive in the stanza. If `stdout=report` or `stdout=display` then `UseTTY` defaults to `yes`. This ensures that the application invoked has its standard output set to line-buffered and the report display (or client standard output) is updated in real-time. If `stdout` is set to any of the other valid values then `UseTTY` defaults to `no`.

You can therefore use the `UseTTY` directive to change the default behaviour. For example, using `stdout=report` and `UseTTY=no` in the same job stanza means that the server-side job runs with no attached tty (or console for Windows Servers) and the output will be block-buffered. This means that the client-side report display will not appear to be running in real time and will tend to receive updates only sporadically (typically when the job ends). Similarly, if you use `stdout=popup` and `UseTTY=yes` in the same job stanza, the server-side job will run with an attached TTY and the output will be line-buffered. Although this will have no effect on the client display (since in this mode, *Trilogy* will collate all the output before displaying the results in a pop-up dialog) it may well have a bearing on the format of the output if the invoked program uses C library calls such as `isatty()` to determine if the output is being routed to a terminal or not.

The default setting should be sufficient for most cases but you can use the `UseTTY` flag to change the default if desired.

See Also:

`Program`, `Stdout`

PreValidateWith

Syntax:

```
PreValidateWith=<script name>
```

Description:

Runs the specified script on the server before any dialog is displayed at the client. The script invoked is passed all the parameters that were specified in any `Param=` or `Params=` directives, followed by the command line parameters that were passed to the client (command line tool) or via the `AddParameter` method of the *Trilogy Scripting Engine*. A Pre-Validate script is typically used to validate the command-line parameters passed. If the script exits with a non-zero exit code (error) the script's standard error output is displayed as a pop-up dialog box at the client and the dialog (if any) is not displayed. Main script (specified with the `Program=` directive in the job stanza) is not executed.

Environment Variables:

<code>\$TRIReason</code>	"PREVALIDATE"
<code>\$TRIDIALOGNAME</code>	Trilogy Job ID
<code>\$TRICLIENTNODENAME</code>	Host Name of the invoking client
<code>\$TRICURRENTFIELD</code>	""
<code>\$TRICHANGEDFIELD</code>	""

See Also:

`Program`, `ValidateWith`, `Param`

ValidateWith

Syntax:

```
ValidateWith=<script name>
```

Description:

Runs the specified script on the server after the OK button has been pressed on the client-side dialog but before the Main Script (`Program=`) is executed. The script invoked is passed all the parameters that were specified in any `Param=` or `Params=` directives, followed by the command line parameters that were passed to the client (command line tool) or via the `AddParameter` method of the *Trilogy Scripting Engine*. All populated fields on the dialog are present in `$TRIFIELDn` environment variables. A Validate script is typically used to validate the fields entered in the client-side dialog. If the script exits with a non-zero exit code (error) the script's standard error output is displayed as a pop-up dialog box at the client and the dialog is not cleared. The main script (`Program=`) is not executed. When the user clears the client pop-up containing the standard-error output from the server-side script, the dialog is still present allowing them to correct any error before clicking "OK" again to resubmit the dialog. At this point the script specified in this `ValidateWith=` directive is run again. Only when the invoked script returns a zero exit code is the client-side dialog destroyed and the main server-side script (`Program=`) run.

Note, this script is only invoked if a dialog is present in the job description. If no dialog is defined (Either with `Dialog=` or `DialogScript=`) then this directive is ignored.

Environment Variables:

<code>\$TRIReason</code>	"VALIDATION"
<code>\$TRIDIALOGNAME</code>	Trilogy Job ID
<code>\$TRICLIENtNODENAME</code>	Host Name of the invoking client
<code>\$TRICURRENTFIELD</code>	""
<code>\$TRICHANGEDFIELD</code>	""
<code>\$TRIFIELD1 -</code> <code>\$TRIFIELDn</code>	Contents of Client Side Dialog

See Also:

`Program`, `PreValidateWith`, `Param`

ApplyButton

Syntax:

`ApplyButton=on|off|yes|no`

Description:

Specifies whether the "apply" button should be present on the dialog. Defaults to off/no. If set to Yes or On, the Apply Button is displayed next to the OK and Cancel buttons on the bottom of the dialog.

The button has no functionality unless the `ListBoxScript=` directive is also specified in the job. Clicking on the Apply button on the client side dialog runs the script on the server identified by the `ListBoxScript` directive.

See Also:

`ApplyButtonText`, `PopulateListBox`, `ListBoxScript`, `ListBox`

ApplyButtonText

Syntax:

```
ApplyButtonText=<text>
```

Description:

Defaults to "Apply". Specifies what text should appear on the Apply Button (if set with `ApplyButton=on`).

See Also:

`ApplyButton`, `PopulateListBox`, `ListBoxScript`, `ListBox`

PopulateListBox

Syntax:

`PopulateListBox=OnDisplay|OnApply`

Description:

Specifies when the server-side `ListBoxScript` should be run to populate the client-side list box (if a list box has been specified with `ListBox=on` or `ListBox=auto`). If set to `OnDisplay` (the default) the `ListBoxScript` is executed on the server at the point when the dialog is first created at the client and its output is used to populate the listbox. If set to `OnApply`, the apply button is added to the client dialog automatically (regardless of the setting of the `ApplyButton=` directive) and the `ListBoxScript` is only run on the server when the Apply button is clicked on the client-side dialog.

If `PopulateListBox=OnDisplay` and the apply button is present on the client-side dialog (by use of the `ApplyButton=on` directive) then the server-side `ListBoxScript` is executed both on initial dialog display at the client and whenever the apply button is clicked on the client dialog.

See Also:

`ListBoxScript`

ListBoxHeight

Syntax:

`ListBoxHeight=Num Rows`

Description:

Specifies the height of any list box which is included in the client-side dialog (due to `ListBox=On` or `ListBox=Auto`). The height is specified as a number of rows. If this parameter is not given then any list box will default to 4 rows in height.

See Also:

`ListBox`, `AutoStretch`

ListBoxWidth

Syntax:

`ListBoxWidth=Num Pixels`

Description:

Specifies the initial width in pixels of any list box which is included in the client-side dialog (due to `ListBox=On` or `ListBox=Auto`). If this parameter is not given then any list box will either stretch horizontally in order to accommodate all the columns in the list box (if `AutoStretch=on`) or will be constrained by the size of the dialog in which it is contained (if `AutoStretch=off`).

Note that setting this option automatically disables `AutoStretch`. Any `AutoStretch` directive in the same job definition as `ListBoxWidth` is ignored.

See Also:

`ApplyButton`, `AutoStretch`, `PopulateListBox`, `ListBoxWidth`, `ListBoxHeight`, `ListBoxScript`, `ListBox`, *Chapter 5 "The List Box"*

ListBoxScript

Syntax:

```
ListBoxScript=<script name>
```

or

```
PopulateListBoxWith=<script name>
```

Description:

Runs the specified script on the server whenever the `listbox` in the client dialog (if specified with `Listbox=on` or `Listbox=auto`) requires to be populated. This can be when the dialog is first displayed (`PopulateListBox=OnDisplay`) or whenever the Apply button is clicked on the client dialog (`PopulateListBox=OnApply`).

If the server-side script exits with code 0 (success), then its standard output is parsed and broken into columns based on the `ListboxSep` character (normally a comma). The client list box is then populated with the output from this server-side script, organized into the appropriate columns.

If `AutoSort` is specified for the Job, then the List Box is sorted automatically by the specified column after the `ListboxScript` has been executed.

If the server-side script exits with a non-zero exit code (failure), then the client-side list box is not populated. Any standard error from the server-side script is displayed as a pop-up dialog at the client.

Environment Variables:

<code>\$TRIReason</code>	"LISTBOX"
<code>\$TRIDIALOGNAME</code>	Trilogy Job ID
<code>\$TRICLIENNTODENAME</code>	Host Name of the invoking client
<code>\$TRICURRENTFIELD</code>	"0"
<code>\$TRICHANGEDFIELD</code>	""
<code>\$TRIFIELD1 -</code> <code>\$TRIFIELDN</code>	Contents of Client Side Dialog

See Also:

`Listbox`, `ListboxSep`, `PopulateListBox`, `AutoSort`

ListBoxSep

Syntax:

```
ListBoxSep=<char>
```

Description:

Specifies the character that is used to delimit columns in the standard output from the server-side script specified in the `ListboxScript=` directive. Defaults to a comma.

See Also:

`ListboxScript`, `PopulateListBox`, `ApplyButton`, `AutoSort`

ColumnNames

Syntax:

```
ColumnNames={col1,col2,col3...}
```

Description:

Specifies a list of names for the columns in the client-side List Box (if one has been specified with `ListBox=yes` or `ListBox=auto`).

The column names are not only used for the column headings but are also used when setting environment variables for any server-side scripts that need to determine the selected rows (if any) in the client-side list box.

If this directive is not present and a list box is included in the dialog, then the column names will be taken from the first row of the output from the `ListBoxScript`.

See Also:

`ColumnWidths`, *Chapter 5 "The List Box"*.

ColumnWidths

Syntax:

```
ColumnWidths={width1,width2,width3...}
```

Description:

Specifies the widths (in pixels) of the columns specified in the `ColumnNames=` directive.

A column width of 0 is a special case. Columns with a width of 0 are suppressed from the client side list box dialog and cannot be displayed. However, the output from the server side script used to populate the client-side list box (`ListBoxScript=`) is still used to populate these "hidden" columns. When the row is selected, these columns still have their environment variables set just as if they had been displayed. This technique can be used to create hidden fields (containing object ids for example) that will be useful to the server-side scripts you wish to invoke but which you do not wish to display to the end-user at the client.

`AutoSort` can be applied to a hidden column.

A column width of "-" (a single dash without quotes) will cause *Trilogy* to automatically set the column width to that of the longest field within the column.

See Also:

`AutoSort`, `ColumnNames`, `AutoStretch`, *Chapter 5 "The List Box"*.

AllowStop

Syntax:

`AllowStop=Yes|No`

Description:

When the `stdout` or `stderr` streams are set to `Report`, a dialog is displayed at the client showing the output from the server-side script (defined by the `Program=` directive) as it happens in real-time. This client-side dialog normally includes a "Stop" button which – if pressed – terminates the server-side job and any of its child processes. This is the default behaviour. To suppress the appearance of the "Stop" button, include the directive `AllowStop=No` in the job stanza.

Note, that this option does not prevent the job from being killed using the `-killjob` option from the *Trilogy Server*.

See also:

`Stderr`, `Stdout`, `UseTTY`

Param

Syntax:

```
Param=<Param>
```

Description:

Creates a fixed parameter, which is pre-pended to any parameter list passed from the client. Any server-side job invoked by *Trilogy* (for field population, pre-validation, dialog validation, list box population or as the main program) is passed any parameters set by a `Param=` (or `Params=`) call, followed by any parameters passed to the command line client or via the `AddParameter` call of the *Trilogy Scripting Engine*.

If more than one `Param=` directive is included in the job stanza, then the first directive is taken as supplying the first parameter, the second directive supplies the second and so on.

For example, if a server-side job is defined like this:

```
vscript:
  Program=c:\windows\system32\myscript.vbs
  Param=x1
```

a client invoking *Trilogy* like this:

```
trilogy vscript p1 p2
```

will result in the `myscript.vbs` script being invoked. This script can then read its parameters which will be "x1" "p1" and "p2".

See Also:

```
Program, PopulateWith, PopulateFieldnWith, PreValidateWith,
ValidateWith, ListBoxScript
```

Params

Syntax:

```
Params={param1,param2...paramN}
```

Description:

Creates a list of parameters which are pre-pended to any parameter list passed from the client. Any server-side job invoked by Trilogy (for field population, pre-validation, dialog validation, list box population or as the main program) is passed any parameters set by a `Param=` (or `Params=`) call, followed by any parameters passed to the command line client or via the `AddParameter` call of the *Trilogy Scripting Engine*.

`Param=` and `Params=` directives can be intermixed freely in the job stanza. In this instances, the parameters are assembled in the order in which they appear in the stanza.

For example, if a server-side job is defined like this:

```
deploy:
  Program=c:\program files\Trinem\trilogy\dm\dm.exe
  Param=-b
  Param=$BROKER
  Params=(-usr,abc,-pw,def)
  Param=-en
  Param=project1
```

a client invoking *Trilogy* like this:

```
trilogy deploy pack1 pack2
```

will result in the server-side script "dm.exe" being invoked as follows:

```
dm.exe -b $BROKER -usr abc -pw def -en project1 pack1 pack2
```

See Also:

`Program`, `PopulateWith`, `PopulateFieldnWith`, `PreValidateWith`,
`ValidateWith`, `ListBoxScript`

Stdout

Syntax:

```
Stdout=discard|display|popup|report|filechooser|file
```

Description:

Determines how *trilogy* handles the standard output stream from the server-side script identified by the `Program` directive in the job stanza.

discard

The standard output from the job is discarded. No standard output from the server side job is displayed at the client.

display

The standard output from the server-side job is routed back to the *trilogy* client where it is sent to the *trilogy* client's standard output. This is the default.

popup

The standard output from the server-side job is routed back to the *trilogy* client where it is collated and displayed in a pop-up dialog when the server-side job has completed. If the client is running on Windows and `SystemTray=Yes` then the popup is displayed as a balloon-style notification.

report

Opens a scrolling window on the client display. The standard output from the server-side job is routed back and displayed in this window. The window contains a real-time clock that monitors total run time, along with a STOP button (unless `AllowStop=No` has been included in the job stanza) that allows for the server-side job to be terminated on request from the client.

filechooser:<filename>

Opens a file chooser dialog on the client, allowing a user to specify a target file. If `<filename>` is specified, then the file chooser dialog selects the relevant directory/file by default and allows the user to change the selection. Once the file is selected and the client-side file chooser dialog is closed, the server-side job is run and its standard output is routed back to the *trilogy* client and saved in the client-side file chosen.

file:<filename>

The standard output from the server-side job is routed back to the client and saved in the specified client-side file.

See Also:

`Program`, `Stderr`

Stderr

Syntax:

```
Stderr=discard|display|popup|report|filechooser|file
```

Description:

Determines how *trilogy* handles the standard error stream from the server-side script identified by the `Program` directive in the job stanza.

discard

The standard error from the job is discarded. No standard error from the server side job is displayed at the client.

display

The standard error from the server-side job is routed back to the *trilogy* client where it is sent to the *trilogy* client's standard error.

popup

The standard error from the server-side job is routed back to the *trilogy* client where it is collated and displayed in a pop-up dialog when the server-side job has completed. This is the default.

report

Opens a scrolling window on the client display. The standard error from the server-side job is routed back and displayed in this window. See `Stdout` for more information. Note, if both `stdout` and `stderr` are set to `report` then only one report window is created and both streams are routed into this window. Under these circumstances the standard error stream is displayed in red (to differentiate it from the standard output stream).

filechooser:<filename>

Opens a file chooser dialog on the client, allowing a user to specify a target file. If `<filename>` is specified, then the file chooser dialog selects the relevant directory/file by default and allows the user to change the selection. Once the file is selected and the client-side file chooser dialog is closed, the server-side job is run and its standard error is routed back to the *trilogy* client and saved in the client-side file chosen.

file:<filename>

The standard error from the server-side job is routed back to the client and saved in the specified client-side file.

See Also:

`Program`, `Stdout`

Stdin

Syntax:

```
Stdin=filechooser:<filename>|file:<filename>
```

Description:

Normally, the standard input of the *trilogy* client is routed to the standard input of the *trilogy* server-side job identified by the `Program` directive. Including this directive in the job stanza allows the server-side job to take its standard input from a client-side file (either a fixed file or one chosen by a file chooser dialog).

filechooser:<filename>

Opens a file chooser dialog on the client, allowing a user to specify a source file. If <filename> is specified, then the file chooser dialog selects the relevant directory/file by default and allows the user to change the selection. Once the file is selected and the client-side file chooser dialog is closed, the server-side job is run and its standard input is taken from the client-side file chosen.

file:<filename>

The standard input for the server-side job is taken from specified client-side file.

Environment Variables:

<code>\$TRISTDINFILE</code>	Client-Side filename of the standard input.
-----------------------------	---

See Also:

`Program`, *Chapter 10: Trilogy Client Command Line Options*

Program

Syntax:

Program=<Path to Program>

Description:

Specifies the server-side program to be invoked when the OK button is clicked on any client-side dialog (or when any `PreValidate` script has exited successfully if no dialog is present or immediately on request from the *Trilogy* Client or *Trilogy Scripting Engine* if neither a `PreValidate` script nor a `Dialog` is present).

The standard input for the program is taken from the standard input from the *Trilogy* command line client or from a client-side file (if a `Stdin` directive is present in the job stanza or the *Trilogy* client has been invoked with the `-i <file>` option). The standard output and standard error streams are handled according to the options specified by the `Stdout` and `Stderr` directives in the job stanza.

Environment Variables:

\$TRIReason	"SCRIPT"
\$TRIDIALOGNAME	Trilogy Job ID
\$TRICLIENTNODENAME	Host Name of the invoking client
\$TRICLIENTUSERNAME	The Login id of the invoking client user
\$TRICURRENTFIELD	""
\$TRICHANGEDFIELD	""
\$TRISTDINFILE	Client Side file from which standard input is being taken.
\$TRIFIELD1 -	Contents of Client Side Dialog
\$TRIFIELDn	
\$TRI_columnname_1-	Client Side List Box Selections
\$TRI_columnname_n	

See Also:

`Stdin`, `Stdout`, `Stderr`, `OnRightClick`

14 Trilogy Scripting Engine



This Chapter is specific to Windows Platforms. The *Trilogy Scripting Engine* is available both Client Side and Server Side. Server Side refers to Windows Servers only.

14.1 Introduction

The *Trilogy Scripting Engine* is a Windows ActiveX™ (COM) component that can be accessed from scripts written in VB Script, JScript, Perl or any other Windows-based scripting or programming language that can instantiate a COM object. The control identifies itself as being safe so it can be safely embedded into windows-based HTML forms without warnings being generated of unsafe ActiveX components.

The Scripting Engine can be used on Windows Clients (where it can be used to invoke remote *Trilogy* Jobs on *Trilogy* Servers, passing those jobs parameters and capturing their output). It can also be used in *Trilogy* Jobs written in VBScript or JScript running under Windows Servers. Using the *Trilogy Scripting Engine* from such scripts makes it easier to read the content of the client-side dialog than would normally be the case using the normal method of reading Environment Variables. In addition, the Scripting Engine exposes a number of additional methods that make writing server-side scripts in VBScript or JScript much easier.

The *Trilogy Scripting Engine* can be inserted into the script by the use of the `CreateObject` command in VBScript/Perl or by the use of the `ActiveXObject` class in JScript. Note, that this will only work if the relevant dll (`atlcom3.dll`) has been registered. This should happen automatically when *Trilogy* is installed through its supplied installer. Should it be necessary to register this dll manually this can be done by opening a DOS prompt, switching to the *Trilogy* home directory and typing:

```
regsvr32 atlcom3.dll
```



Users of Windows Vista, Windows 7 or Windows Server will need to ensure that the command window has been run with Administrative Privileges (run as Administrator). Otherwise, `regsvr32` will fail.

Here is an example of how to embed the *Trilogy Scripting Engine* in both VB Script and JavaScript:

VB Script

```
Dim Trilogy  
Set Trilogy = CreateObject("Trilogy.Scripting");
```

JavaScript

```
var Trilogy = new ActiveXObject("Trilogy.Scripting");
```

Perl

ActiveState:

```
use OLE;  
$Trilogy = CreateObject OLE "Trilogy.Scripting"  
or die "Cannot create Trilogy Scripting Object"
```

Standard Distro:

```
use Win32::OLE;  
Win32::OLE::CreateObject("Trilogy.Scripting",$Trilogy)  
    or die "Cannot create Trilogy Scripting Object"
```

All of these methods create an object called "Trilogy" which then encapsulates various methods.

14.2 Using the Trilogy Scripting Engine Client Side – Overview

The general technique when using the *Trilogy Scripting Engine* from a client is:

- Use `SetProgramID` to set the Program ID (*Trilogy* Job Name).
- Use the `AddParameter` method to add any command-line parameters you wish to pass to the server-side script
- Either:
 - Use the `ShowDialog` method to bring up the dialog associated with the Trilogy Name specified in `SetProgramID` or...
- Use the `AddFieldValue` method to add any field values you wish the server-side script to use. These will be set in `TRIFIELD1` to `TRIFIELDn` as though the normal client GUI has been invoked.
- Use the `SetStandardInput` method to set the standard input (if any) for the server-side script.
- Call the `Execute` method to run the server-side script
- Use the `getline` method to read the output from the server-side script.

14.3 Using the Trilogy Scripting Engine Server Side – Overview

The general technique when using the *Trilogy Scripting Engine* from a server-side *Trilogy* Job is:

- Use the `GetExecutionReason()` method to determine the reason the script has been called
- Use `GetField()` to read the client-side dialog content

14.4 Scripting Engine Methods

The methods available to the *Trilogy* Scripting Engine are documented below. Note that *object* refers to the object name used in the script when you instantiated the ActiveX Scripting component – i.e.:


```
Set object = CreateObject("Trilogy.Scripting");
```

Each method can be run either client-side (when a client is running jobs and interacting with a remote Trilogy Server) or server-side (when a Windows Server is running a script which is using methods in the Trilogy Scripting Engine). A few methods can be run either on the client or the server.

Each method is identified as being able to be run client-side or server-side.

If an attempt is made to run a client-side method on the server (or vice-versa) then an error is returned by the method.

SetServerName

	Client
	Server

`object.SetServerName servername`

Sets the name of the host where the *Trilogy* Server is located. Used to override the `Server=` setting in the client's local `trilogy.conf` file. Use this if you need to run a job on a different *Trilogy* Server than the default.

Example:

VB Script

```
Trilogy.SetServerName "ServerName"
```


JavaScript

```
Trilogy.SetServerName("ServerName");
```

Perl

```
$Trilogy->SetServerName("ServerName");
```

SetServerPort

	Client
	Server

object.SetServerPort Port Number

Sets the number of the port on which the *Trilogy* Server is listening. Used to override the `Port=` setting in the client's local `trilogy.conf` file. Use this if you need to run a job on a different *Trilogy* Server than the default.

Example:

VB Script

```
Trilogy.SetServerPort 2032
```


JavaScript

```
Trilogy.SetServerPort(2032) ;
```

Perl

```
$Trilogy->SetServerPort(2032) ;
```

Set Program ID

	Client
	Server

`object.SetProgramID programid`

Sets the *Trilogy Job Name*. This is used by *Trilogy* to locate the attributes on the *Trilogy* Server which defines the job to be run. Note, invoking this method automatically clears the associated parameter and field list. See *Adding Command Line Parameters* and *Setting Field Values* below.

Example:

VB Script

```
Trilogy.SetProgramID "add_user_to_project"
```


JavaScript

```
Trilogy.SetProgramID("add_user_to_project");
```

Perl

```
$Trilogy->SetProgramID("add_user_to_project");
```


Adding Command Line Parameters

	Client
	Server

object.AddParameter ParameterValue

Adds a new parameter *ParameterValue* to the end of the current parameter list. Since the specified parameter is always added to the end of the current list, use the first call to add the first parameter, the second to add the second and so on.



Remember, this list is cleared whenever you invoke the *SetProgramID* method. Therefore, if you want to invoke two *Trilogy* jobs, you must specify the parameter list again after the second call to *SetProgramID*.

Example:

VB Script

```
Trilogy.AddParameter "Parameter 1"  
Trilogy.AddParameter "Parameter 2"  
Trilogy.AddParameter "Parameter 3"
```

JavaScript

```
Trilogy.AddParameter("Parameter 1");  
Trilogy.AddParameter("Parameter 2");  
Trilogy.AddParameter("Parameter 3");
```

Perl

```
$Trilogy->AddParameter("Parameter 1");  
$Trilogy->AddParameter("Parameter 2");  
$Trilogy->AddParameter("Parameter 3");
```

Adding Field Values

✓	Client
	Server

```
object.SetField FieldNumber ParameterValue
```

Sets the *Trilogy* field identified by *FieldNumber* to the passed *ParameterValue*. When the server-side script is invoked, it will have access to these values via the `TRIFIELDn` environment variables. In other words, running this command in your script:

```
Trilogy.SetField 2 "test project"
```

... causes `TRIFIELD2` to be set to "test project" when the server-side script is invoked. This means that server-side scripts using the *Trilogy Scripting Engine* can also access the field content using the `GetField` method.



Remember, all the fields are reset following a call to the *SetProgramID* method. Also, all the fields are reset should you invoke the *ShowDialog* method.

You can use this technique to allow field values from your own dialogs (such as HTML based forms) to be passed to *Trilogy* clients.

Example:

VB Script

```
Trilogy.SetField 1 "Field 1 value"  
Trilogy.SetField 2 "Field 2 value"  
Trilogy.SetField 3 "Field 3 value"
```


JavaScript

```
Trilogy.SetField(1,"Field 1 value");  
Trilogy.SetField(2,"Field 2 value");  
Trilogy.SetField(3,"Field 3 value");
```

Perl

```
$Trilogy->SetField(1,"Field 1 value");  
$Trilogy->SetField(2,"Field 2 value");  
$Trilogy->SetField(3,"Field 3 value");
```

Displaying a Trilogy Dialog

	Client
	Server

```
res = object.ShowDialog
```

Opens up the appropriate *Trilogy* dialog, identified via the *Trilogy Name* set in the call to `SetProgramID`. `ShowDialog` does not return until the user exits the dialog (either by cancelling it or pressing "okay" to submit the values).

Should the user press "okay" then any field values set by calls to the `SetField` method are lost and the field values are replaced by those from the dialog.

Note - unlike the *Trilogy* command-line tool, when a dialog is submitted (via its "okay" button), the Scripting Engine does *not* automatically run the associated server-side script. Instead, control is returned to the invoking script with the field contents available via calls to `GetField`. However, validation scripts (`PreValidateWith=` and `ValidateWith=`) and population scripts (`PopulateWith=` and `PopulateFieldnWith=`) are invoked by the *Trilogy* server in the normal way in order to construct and validate the dialog.

This means that you can use *Trilogy* merely to present more sophisticated dialogs to the end-user than would normally be available to VBScript or Perl. In this case, no `Program=` directive need be present in the `trilogy.conf` stanza entry. To read the field values into your script you can use the `GetField` method after `ShowDialog` has returned.

If you want to execute the server-side script you must invoke the `execute` method following the call to `ShowDialog`.

Return Values:

The `ShowDialog` method returns a code that your script can use to determine how to proceed:

Returned Value	Meaning
0	No Dialog. There is no <code>Dialog=</code> directive in the appropriate <code>trilogy.conf</code> stanza entry.
1	Dialog Not Found. There was a <code>Dialog=</code> directive but it referred to a file that was either not present or for which read permission was not granted.
2	Dialog Invalid. The dialog definition was syntactically incorrect and could not be converted into a Trilogy client-side GUI.
3	"OKAY". The user has exited the dialog by pressing the

	"okay" button.
4	"CANCEL". The user has cancelled the dialog by pressing the "cancel" button.
5	Pre-validation failed. A <code>PreValidateWith=</code> directive was present in the <code>trilogy.conf</code> stanza entry and the script returned a non-zero exit code. The standard error output from this script can be read with a call to <code>getline(2)</code> .
6	Cannot connect – number of permitted nodes exceeded.
7	Cannot connect – Server license has expired.

Example:

VB Script

```
res = Trilogy.ShowDialog

if res = 3 Then
    ' User has pressed "okay" - Proceed
Else
    ' something else has happened
End If
```



JavaScript

```
Trilogy.ShowDialog();
```

Perl

```
$Trilogy->ShowDialog();
```

Retrieving Field Values

	Client
	Server

```
res = object.GetField
```

Client-Side, this call returns the contents of the specified field number, following a call to either `SetField` or `ShowDialog`. If you want to read the values of fields following the display of a *Trilogy* dialog, then use this method.



Remember, that fields start at 1 and are numbered from left to right and from top to bottom of the dialog.

Also remember that Radio Buttons and Checkboxes return "1" if they are set and "0" otherwise.

Server-side (on Windows Servers) this call can be used to read the content of the client-side dialog when the server-side script is invoked.

Example:

VB Script

```
Field1Value = Trilogy.GetField(1)  
Field2Value = Trilogy.GetField(2)  
Field3Value = Trilogy.GetField(3)
```


JavaScript

```
Field1Value = Trilogy.GetField(1);  
Field2Value = Trilogy.GetField(2);  
Field3Value = Trilogy.GetField(3);
```

Perl

```
$Field1Value = $Trilogy->GetField(1);  
$Field2Value = $Trilogy->GetField(2);  
$Field3Value = $Trilogy->GetField(3);
```

Executing *Trilogy* Job

	Client
	Server

`object.execute [0|1]`

Runs the appropriate job on the *Trilogy* server, identified by the previously specified *Trilogy Job Name* (see "Set Program ID" above), passing it any parameters specified in previous calls to *AddParameter*. Values set via calls to *SetField* (or set as a result of a call to *ShowDialog*) are made available to the server-side script in the environment variable `TRIFIELD1` to `TRIFIELDn`.

Control normally returns to the calling script when the server side job completes. The only exception to this is if the job has `SystemTray=yes` in its configuration and the *Trilogy Client Service* is not running. In these circumstances, the job will create its own icon in the System Tray (also known as the Notification Area) and will animate this to indicate the job is running.

When the job has created an icon in this way, control will only return to the calling program when:

- The last "Balloon Style" notification has been dismissed (either by clicking to close it or by it timing out).
- If `SystemTray` is set to "Hold" then the icon has been dismissed (by right-clicking and selecting "Quit")

If control were to be returned before the last balloon was displayed then the invoking script may well exit which would remove the icon prematurely.

Therefore, *Trilogy* will not return the control to the calling script unless it is safe to do so.

To override this behaviour, `execute` takes an optional parameter. If you specify nothing or 0 (the default), `execute` will wait until the icon is idle (no outstanding balloon notifications and/or user has selected Quit on a held job). If you specify 1 (or any non-zero code) then `execute` will return immediately the server side job completes. Should the calling script exit, the icon will be removed from the System Tray and any outstanding balloon notifications will be lost.

Example:

VB Script

```
Trilogy.execute
```

JavaScript

```
Trilogy.execute();
```

Perl

```
$Trilogy->execute();
```

Finding Exit Status of Job

✓	Client
	Server

`object.exitcode`

Returns the exit code of the server-side script after the job has successfully executed.

Example:

VB Script

```
Dim retcode
retcode = Trilogy.exitcode
```


JavaScript

```
var retcode = Trilogy.exitcode();
```

Perl

```
$retcode = $Trilogy->exitcode();
```

Reading Results from Executed Jobs

	Client
	Server

```
object.getline (streamno)
```

Reads the next line from the specified stream number:

1 = standard output

2 = standard error

These streams represent the output from the invoked server-side script. In addition, stream 2 (standard error) can also represent the output from any pre-validation script that has returned a non-zero exit code prior to the display of a *Trilogy* dialog. See the `ShowDialog` method above.

Note, it is usual to use the `EndOfStream` method to determine whether there is data to read before invoking the `getline` method. See `EndOfStream` below.

Example:

VB Script

```
opline = Trilogy.getline(1)
errline = Trilogy.getline(2)
```

JavaScript

```
opline = Trilogy.getline(1);
errline = Trilogy.getline(2);
```

Perl

```
$opline = $Trilogy->getline(1);
$serrline = $Trilogy->getline(2);
```


Checking for end of stream

✓	Client
	Server

`object.EndOfStream(streamno)`

Returns 1 if the specified stream is exhausted (no more data available), or 0 if there is more data to be read. Streams are identified by:

1 = standard output

2 = standard error

Example:

```
while Trilogy.EndOfStream(1) = 0
  opline = Trilogy.getline(1)
wend
```

Displaying Standard Output as a Pop-up

✓	Client
	Server

`object.ShowStandardOutput()`

Displays all the standard output from the server-side job in a pop-up dialog.

Displaying Standard Error as a Pop-up

✓	Client
	Server

`object.ShowStandardError()`

Displays all the standard error from the server-side job in a pop-up dialog.

Setting Standard Input


✓	Client
	Server

```
object.SetStandardInput(filename)
```

Specifies that the standard input for the server-side job should be read from the client side file specified by *filename*.

The basename of the specified file (i.e. just the filename component minus any leading directory hierarchy) will be placed into the environment variable `TRISTDINFILENAME` which is then available in the invoked server-side script.

Choosing a File

	Client
	Server

```
filenum = object.ChooseFile(filename,mode)
```

Opens a file chooser dialog either to open a file for reading (mode="r") or writing (mode="w") and returns a corresponding handle to the opened file.

If the dialog is cancelled the return value is 0. Otherwise, the return value is a handle to the opened file.

If "filename" points to a directory then the directory is opened with no file pre-selected.

If "filename" points to a file then the directory in which the file is located is opened with the file pre-selected.

Opening a File

✓	Client
	Server

```
filenum = object.OpenFile(filename,mode)
```

Opens a file for reading (mode="r") or writing (mode="w") and returns a corresponding handle to the opened file.

If the file cannot be opened the return value is -1. Otherwise, the return value is a handle to the opened file.


Closing a File

✓	Client
	Server

`object.CloseFile(filenum)`

Closes a file previously opened with either ChooseFile or OpenFile. The parameter is the file handle previously returned by these calls.

Setting a stream to a file

	Client
	Server

```
res = object.SetStream(streamno,filenum)
```

Sets the specified stream (Standard Input, Standard Output or Standard Error) of the server-side job to come from or to the specified file handle (previously opened with `OpenFile` or `ChooseFile`).

`streamno` should be either 0 (Standard input), 1 (Standard Output) or 2 (Standard Error).

`filenum` should be a file handle as returned from `OpenFile` or `ChooseFile`.

Return value is one of:

- 0 Stream has been set successfully
- 1 `filenum` is invalid
- 2 `streamno` is invalid (not one of 0, 1 or 2)
- 3 `filenum` is opened in wrong mode. If you are setting the standard input stream (`streamno=0`) then `filenum` should have been opened in Read Mode. If you are setting standard output (`streamno=1`) or standard error (`streamno=2`) then `filenum` should have been opened in Write Mode.

Getting the Pathname of a File

✓	Client
	Server

```
path = object.GetPathName(filenum)
```

Returns the full path name (directory and filename) for the specified filenum.

filenum should be a file handle as returned from `OpenFile` or `ChooseFile`.

Getting the Directory Name of a File

✓	Client
	Server

```
path = object.GetDirName(filenum)
```

Returns the directory name for the specified filenum.

filenum should be a file handle as returned from `OpenFile` or `ChooseFile`.

Getting the File Name of a File

✓	Client
	Server

```
path = object.GetFileName(filenum)
```

Returns the file name for the specified filenum (i.e.: the base file name excluding the directory).

filenum should be a file handle as returned from `OpenFile` or `ChooseFile`.

Suspending Execution

✓	Client
✓	Server

`object.Wait(milliseconds)`

Suspends execution for the specified number of milliseconds.


Finding Job Name

	Client
✓	Server

```
jobname = object.GetJobName()
```

Returns the name of the *Trilogy* Job under which the server-side script is running.

Finding Number of List Box Selections

	Client
	Server

```
selections = object.GetListBoxSelectionCount()
```

Returns the number of selections in the client-side List Box.

Retrieving List Box Selections

	Client
✓	Server

```
value = object.GetListBoxField(rownum,columnname)
```

Returns the value of the specified column name for the specified selected row number.

`rownum` is the selected row number (from 1 to the value returned by `GetListBoxSelectionCount`)
`columnname` is the name of the column (as specified by the `ColumnNames` directive in the server-side `trilogy.conf` file).

Finding Standard Input Filename

	Client
✓	Server

```
filename = object.GetInputFileName()
```

Returns the name of the client-side file that is forming the standard input for the server-side job (assuming that the file has been specified through the `-i` flag of the *Trilogy Client* or has been specified via `Stdin=file:<filename>` or the filechooser).

If standard input is being “piped” into the *Trilogy Client* then this value is not set.

Finding Client Machine Name

	Client
✓	Server

```
clientnodename = object.GetClientNodeName()
```

Returns the hostname of the client machine from where the run request was issued.

Finding Client User Name

	Client
✓	Server

```
clientusername = object.GetClientUserName()
```

Returns the login ID of the client user who has issued the run request.

Finding Execution Reason

	Client
✓	Server

```
reason = object.GetExecutionReason()
```

Returns the reason why the server-side script has been invoked. The following string values are returned:

LISTBOX	Script has been run as a result of the <code>ListBoxScript=</code> (or <code>PopulateListBoxWith=</code>) directive.
POPULATE	Script has been run either as a result of a <code>PopulateWith=</code> or a <code>PopulateFieldnWith=</code> directive.
SCRIPT	Script has been run as a result of a <code>Program</code> directive.
VALIDATION	Script has been run as a result of a <code>ValidateWith=</code> directive.
PREVALIDATE	Script has been run as a result of a <code>PreValidateWith=</code> directive.
DIALOG	Script has been run in order to create a dialog definition.

Getting Current Field Number

	Client
✓	Server

```
fieldno = object.GetCurrentField ()
```

Returns the field that *Trilogy* wishes the script to populate (GetExecutionReason returns POPULATE).

Getting Changed Field Number

	Client
✓	Server

```
fieldno = object.GetChangedField ()
```

Returns the field that has changed on the client side dialog which has resulted in the linked field needing to be repopulated (`GetExecutionReason` returns `POPULATE`).

Writing to Standard Output

	Client
✓	Server

```
object.WriteToStandardOutput(text)
```

Writes the specified text to the Standard Output stream.

Writing to Standard Error

	Client
✓	Server

```
object.WriteToStandardError(text)
```

Writes the specified text to the Standard Error stream.

Sending a Balloon Message

	Client
✓	Server

```
object.Balloon(icon,title,text)
```

Creates a balloon message at the invoking client with the specified title, text and icon.

Icon should be one of:

- 0 No Icon
- 1 Information
- 2 Warning
- 3 Error

Note, the client must either be running *Trilogy Client for Windows Service* or the job need the `Systray=Yes` directive in order to create an icon in the notification area. The *Trilogy* icon needs to be in the System Tray (Notification Area) in order for the balloon to be received.

Sending a Balloon Message to a Group

	Client
✓	Server

`object. BalloonToGroup(groupname, icon, title, text)`

Creates a balloon message at each logged in user in the specified group with the specified title, text and icon.

Icon should be one of:

- 0 No Icon
- 1 Information
- 2 Warning
- 3 Error

Note, the targeted client(s) must be running *Trilogy Client for Windows Service* in order for the balloon to be received.

Trilogy Server will interact with the specified Group Processor in order to determine which users will receive the balloon message.

Appendix A – Examples

In this section we show a number of example scripts and dialogs.

A.1 Demo Dialog/Script

As discussed in the installation guide, the *Trilogy* server installation ships with a number of demo screens and scripts. You can invoke these demos on any client machine that can access the server and that has the *Trilogy* Client installed.



You can perform the client operations at the server machine since installing a *Trilogy* Server automatically installs a *Trilogy* Client.

To invoke the demonstration dialog, logon to a client machine and run the *Trilogy* command-line client with:

```
trilogy demo
```

This will bring a dialog to the screen. Pressing “okay” simply dumps the contents of the dialog in a pop-up box.



If you have installed the *Trilogy Client Service for Windows*, you can right click on the *Trilogy* Icon in the System Tray and select “Trilogy Demo Dialog”

The “source” screen definition is installed on the server in:

UNIX:
`$TRILOGYHOME/demo/screens/demo.scn`

WINDOWS:
`%TRILOGYHOME%\demo\screens\demo.scn`

The script used to populate the dialog, its pull-down lists and which takes the GUI parameters at the point of execution is installed on the server in:

UNIX:
`$TRILOGYHOME/demo/scripts/demo.sh`

WINDOWS:
`%TRILOGYHOME%\demo\scripts\demo.bat`

You can use these files (and the shipped `trilogy.conf` file) to gain an understanding of the field numbering convention and how dialogs can be created.

A.2 Change the Demo

Try changing the shipped demonstration dialog and see what happens.

A.2.1 Add a new tab

Using a text editor (such as vi or notepad) on the server, open up the demo dialog screen definition in:

Unix:

```
$TRILOGYHOME/demo/screens/demo.scn
```

Windows:

```
%TRILOGYHOME%\demo\screens\demo.scn
```

Add a new tab to the screen by adding the following lines to the bottom of the file:

```
> My New Tab
X A checkbox
x another checkbox
- Some Radio Buttons
O Yes
o No
Data Entry Field [
A Drop Down      { }
```

Save the file and then run the demo again from your client machine. See how a new tab is present on the dialog and how the additional fields have been rendered.

A.2.2 Populate the new Drop Down

You will notice that the drop-down field you added ("A Drop Down") has no content. This is because we have not told *Trilogy* how to populate the field.

On the Server, open up the `trilogy.conf` configuration file (this will be found in the `TRILOGYHOME` root directory of your installation). Find the entries which relate to the `DEMO` job. Add the following directive:

```
PopulateField22With={My Data 1,My Data 2,My Data 3}
```

Launch the demo dialog again, click on your new tab ("My New Tab") and see how the drop-down list "A Drop Down" is populated.

Create a script on the server which outputs some lines to standard output. Here is an example:

Windows:

```
@echo off
echo This Line 1 is from a Script
echo This Line 2 is from a Script
echo This Line 3 is from a Script
```

Unix/Linux:

```
#!/bin/sh
echo "This Line 1 is from a Script"
echo "This Line 2 is from a Script"
echo "This Line 3 is from a Script"
```



On Unix/Linux servers, make sure the script you've created is executable.

Change your new `PopulateField22With` line in `trilogy.conf` so that it points to this script rather than having a list of hard-coded values:

```
PopulateField22With=$TRILOGYHOME/demo/scripts/myscript.bat
```

(or equivalent depending on your platform and where you placed the script).

Rerun the demo at the client and see how the script's output populates your new drop-down list.

A.2.3 Change the Output Format

Alter `trilogy.conf` on the server and add a new directive to the `DEMO` job definition:

```
SystemTray=Yes
```

Rerun the demo on a windows client. Click OK on the presented dialog. Note that a *Trilogy* Icon appears in the System tray (notification area) and that the standard output from the server side script is now presented in a balloon from that icon. Note how the output is split across multiple balloon notifications. Note how the icon disappears after the last balloon is dismissed.

Alter `trilogy.conf` on the server and change the `SystemTray` directive to `Hold` so that it now looks like this:

```
SystemTray=Hold
```

Rerun the demo on a windows client. Click OK on the presented dialog. Note that a *Trilogy* Icon appears as before in the System Tray and that the standard output from the server side script is presented as a series of balloon style notifications. Note, however, that after the last balloon is dismissed the icon does not disappear and remains in the System Tray. In order to close the icon you need to right-click the Icon and select "Quit".

Alter `trilogy.conf` on the server and change the `Stdout` directive for the `DEMO` dialog from `Popup` to `Report` and switch off the `SystemTray` icon like this:

```
Stdout=Report
SystemTray=No
```

Rerun the demo at the client and see how the output is presented in a "Report" window.

Alter the `SystemTray` option to `Yes` and rerun the demo from the client. Note how the report window is not displayed. Double Click on the *Trilogy* Icon in the System Tray in order to open the report window. Iconize the Report Window and note how it disappears from the task bar – it is iconized back to the *Trilogy* Icon in the System Tray. Double clicking the icon will restore the report window.

You can close the job by right-clicking the *Trilogy* Icon in the System Tray and selecting "Quit" or opening the Report Window and clicking OK.

A.3 List Box Demo

Trilogy ships with a List Box Demonstration dialog. To run this demo either type:

```
trilogy listbox_demo
```

from a command line on a client machine or – if you are running on a Windows Client and you are running the *Trilogy Client Service* – right-click on the icon in the System Tray and select "Trilogy Listbox Demo"

Doing this will open a dialog containing a list box. You can select a decade from the first drop-down – this will populate the second drop-down list with a list of artists. Selecting an artist will populate the list box with some selected hits of the selected artist. Records that reached number 1 are illustrated with a star icon in the first column.

If you select "The Beatles" you will see 2 of their records automatically selected.

You can select one or more rows in the list box – doing this will populate a field showing the song selection made. You can also right-click in the list box and select the "Show Hidden Field Value" option – this will open a pop-up box showing the hidden fields relating to the selected row(s).

This demo illustrates the following principles:

- Linking fields
- Linking fields to the List Box
- Hidden List Box Fields
- Context Sensitive Icons
- Context Columns
- Right-Click Jobs
- Auto Sorting
- Auto Selection of Columns

Examine the `trilogy.conf` server-side file alongside the `listbox` script. The script can be found in either:

Unix/Linux:

```
$TRIOLOGYHOME/demo/scripts/listbox.sh
```

Windows:

```
%TRIOLOGYHOME%\demo\scripts\listbox.vbs
```

You should be able to see how the various *Trilogy* directives control how the script's output is presented at the client.

A.4 System Tray Demonstration

Trilogy ships with a dialog which illustrates how icons can be placed into the System Tray on Windows Clients and how that icon can be animated to show that the job is running.

To invoke this demo either type:

```
trilogy traydemo
```

from a command line on a Windows client machine or – if you are running the *Trilogy Client Service* – right-click on the icon in the System Tray and select "Trilogy System Tray Demo"

Doing this presents a dialog with two tabs – a "start" icon, title and text and an "end" icon, title and text. Fill in the details and click OK. A Trilogy Icon appears in the System Tray (if it was not there already) and a balloon is presented containing the information entered in the first tab. The script will then run for 15 seconds. During this time the icon animates in the System Tray to indicate to the user that the job is running. At the conclusion of this time the icon stops animating and a second balloon is presented containing the information entered in the second tab.

Examine the `trilogy.conf` server-side file alongside the `traydemo` script. The script can be found in either:

Unix/Linux:

```
$TRIOLOGYHOME/demo/scripts/traydemo.sh
```

Windows:

```
%TRIOLOGYHOME%\demo\scripts\traydemo.vbs
```

You should be able to see how the various *Trilogy* directives control how the script's output is presented at the client.

A.5 A UNIX Backup Program

In this example, we will show how to create a simple, server-side script which – when invoked by *Trilogy* – will allow a tar file to be created and saved on the machine on which the *Trilogy* Client is running. In this way, a simple “backup” can be made of a directory and the resulting tar file can be copied and saved on the instigating machine in a location specified by the user.

First, we need to create a simple server-side shell-script to create the tar file. Note that the file is written to the script’s standard output stream:

```
#!/bin/ksh
cd $HOME
tar cvf - .
```

This file should be saved on the server as “backup.ksh”. Note, that the tar command writes its output to the standard output stream (the – option specifies that output is to be sent to stdout). Also note the “v” option – this tells *tar* to write the name of each file to standard error as it’s being processed.

Now, we set up the server-side `trilogy.conf` file. We add the following entries:

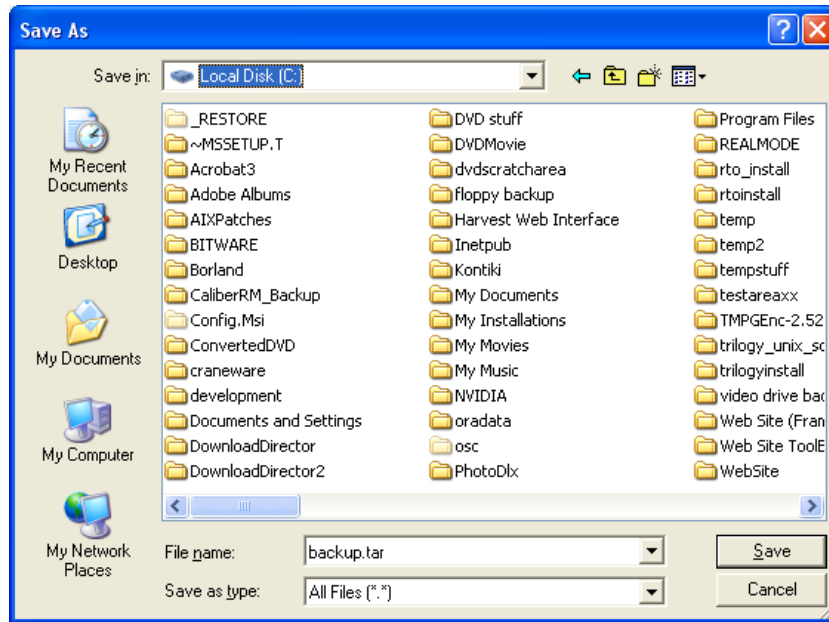
```
BACKUP:
    Program=$TRILOGYHOME/scripts/backup.ksh
    Stderr=Report
    Stdout=Filechooser:C:\backup.tar
```

This tells *Trilogy* to run the backup.ksh script on the server when a client requests us to run the “BACKUP” job. The standard error output (which is the list of files as they are processed) is to be displayed in a scrolling “report” window on the client. The standard output (which is the actual tarred up output file) is to be written to a file on the client, the precise name and location of which is to be specified by the user by means of a file chooser dialog. By default, the file chosen will be “backup.tar” in the user’s C: drive.

The job is now done. By invoking *Trilogy* as follows:

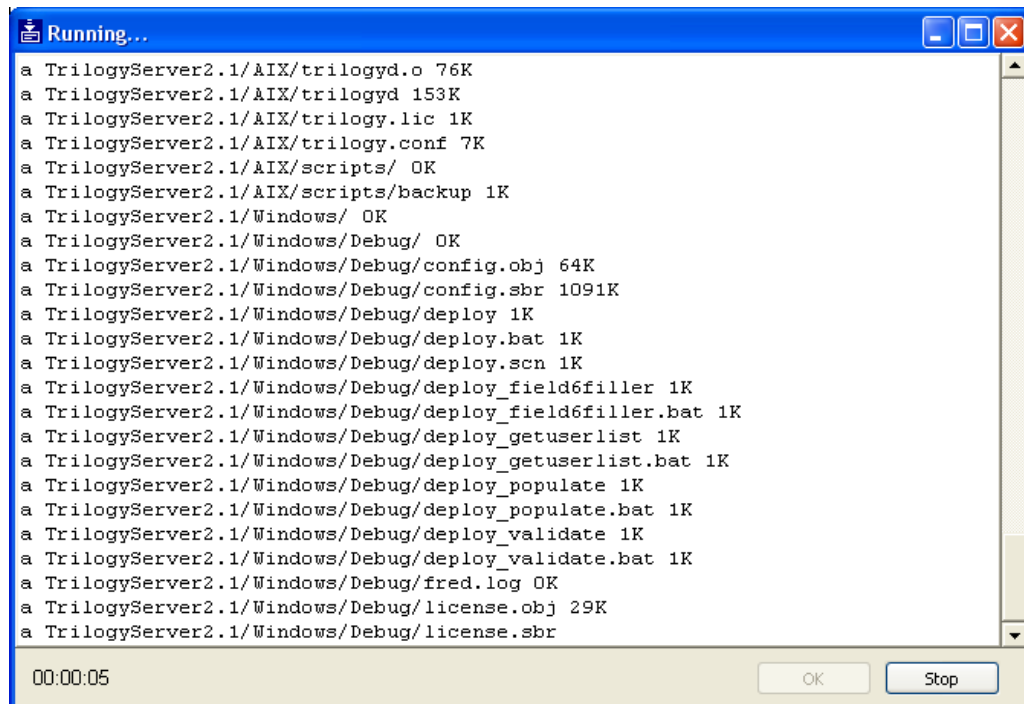
```
trilogy backup
```

the file chooser dialog is automatically opened:



By default, the file to be written is called “backup.tar”. The user now has the option of changing this name or the location in which it is to be stored.

Once the user has clicked “Save”, the main job (backup.ksh) is run on the server. The standard output of the job is passed back to the client in real time and written to the specified file. The standard error of the job is written to the “report” window, again in real time. The user can therefore see the progress of the backup as it occurs:



At the conclusion of processing, the backup will have been written to the chosen file on the *Trilogy* Client machine.

A.6 Trilogy Job Control

As mentioned in *Server Side Job Control*, you can invoke the *Trilogy Server* executable with the `-showjobs` option to list all the jobs that are running under *Trilogy* control. If `TRIReason` is set to `LISTBOX`, then the list generated is comma separated and is suitable for inclusion in a client-side List Box display.

By using the `AutoRefresh` option, an administrator can easily set up a dynamic monitoring tool which can be used to view (and terminate) *Trilogy* Jobs.

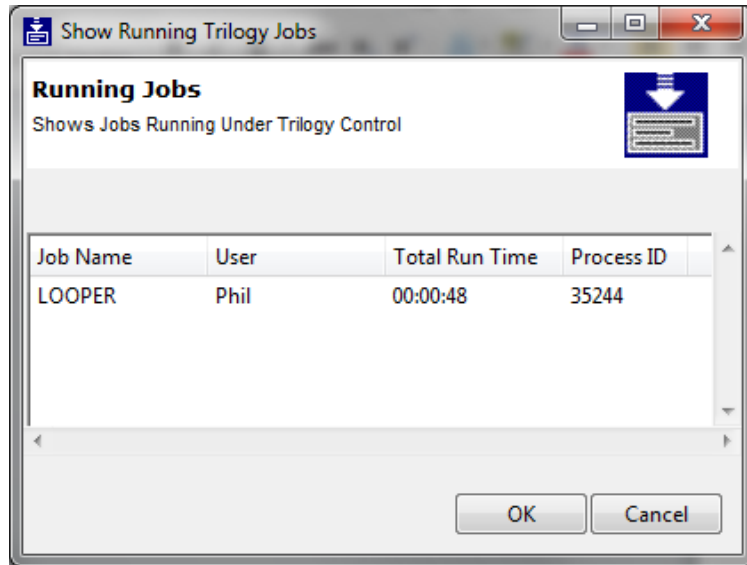
To do this, first create an empty dialog definition file (we do not need any entry fields for this dialog but you cannot display a list box without a dialog). An empty dialog definition file is a file containing no text (it's zero-length).

Then create two job definitions in the server-side `trilogy.conf` file as follows:

```
SHOW_RUNNING_JOBS:
  Dialog=$TRILOGYHOME/screens/running.scn
  Banner=On
  BannerHeading=Running Jobs
  Title=Show Running Trilogy Jobs
  BannerText=Shows Jobs Running Under Trilogy Control
  ListBox=On
  ColumnNames={Job Name,User,Total Run Time,Process ID}
  ColumnWidths={100,100,100,70}
  Selection=Single
  PopulateListBoxWith=$TRILOGYHOME/trilogyserver.exe
  Param=-showjobs
  OnRightClick={KILL_RUNNING_JOB}
  TrayMenu=Yes
  AutoRefresh=1
  Group=Administrators

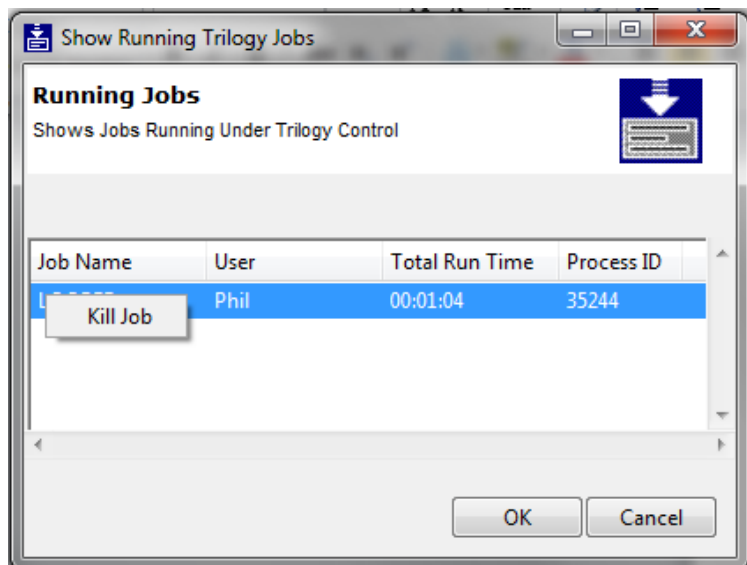
KILL_RUNNING_JOB:
  Title=Kill Job
  Program=$TRILOGYHOME/trilogyserver.exe
  Param=-killjob
  Param=$TRI_PROCESS_ID_1
  Stdout=Popup
```

The dialog definition file `running.scn` is an empty file. The list box is populated by executing `trilogyserver.exe` (`PopulateListBoxWith`) with a parameter of `-showjobs` (`Param=`). Since `TRIReason` will be set to `LISTBOX`, `trilogyserver.exe` will generate a comma-separated list of values which will be incorporated into the client-side list box. The List Box is set to automatically refresh every second (`AutoRefresh=1`) so the "Total Run Time" of each running job will increment in real time and any new jobs will appear automatically in the list.

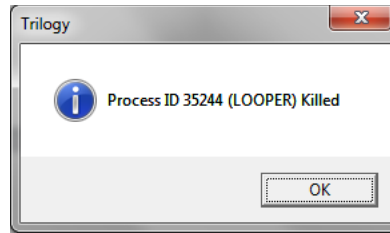


Dynamic Display (Total Run Time increments automatically).

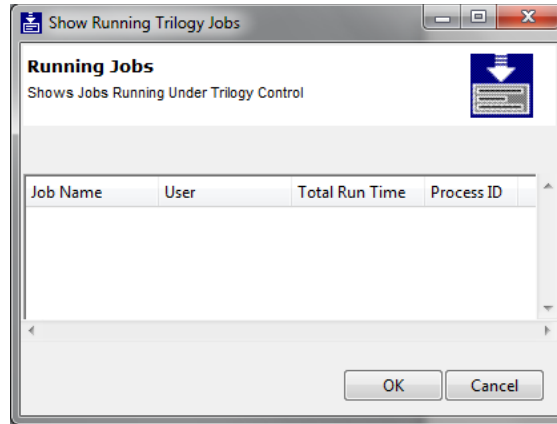
The user can select a single row in the list box (`Selection=Single`). By selecting and right-clicking on a job (`OnRightClick={KILL_RUNNING_JOB}`) the invoking client user can select a "Kill Job" menu option (`Title=Kill Job` in the `KILL_RUNNING_JOB` job). Selecting this option calls `trilogyserver.exe` with the `-killjob` parameter (`Param=-killjob`) and the selected process id as selected from the list box (`Param=$TRI_PROCESS_ID_1`). The output from the kill operation is then set to be displayed in a pop-up dialog (`Stdout=Popup`).



Right Clicking to bring up Kill Job option



Standard Output from Kill Job process present as pop-up



Job automatically removed from list.

Appendix B – License Terms and Conditions

B.1 Terms Used in this License

"We", "us" and "our" refers to Trinem Software. "You" and "your" refers to the individual or entity that has ordered the programs from us. "Programs" refers to the various software components that make up the entire *Trilogy* application suite and includes the documentation. "License" refers to your right to use the programs under the terms of this agreement. The laws of the United Kingdom govern this agreement. You and Trinem Software agree to submit to the exclusive jurisdiction of, and venue in, the courts of the United Kingdom in any dispute relating to this agreement.

B.2 License Overview

We are willing to license the programs to you only upon the condition that you accept all the terms contained in this agreement. Read the terms carefully as the installation of any portion of the *Trilogy* software on one or more of your organization's computers confirms your acceptance. If you are not willing to be bound by these terms please do not install any portion of the *Trilogy* software.

B.3 License Rights

We grant you a non-exclusive, non-transferable license to use the programs only for the purposes described in the documentation. *Trilogy* is licensed on a client basis - you should not install the *Trilogy* client software components on more workstations than are governed by your purchase order. Your purchase order will list the number of client workstations on which you are permitted to install *Trilogy*. If you wish to install the *Trilogy* client on more than this number of workstations then you must purchase additional licenses. Contact Trinem Software for more information.

B.4 Ownership and Restrictions

We retain all ownership and Intellectual Property Rights in the programs (excluding those open-source components supplied by external organisations – see below). The programs may be installed on your computers only and must not be installed on any other organisation's computers. You may make one copy of the original media for backup purposes.

You must not:

- 1) Remove or modify any program markings or any notice of our proprietary rights;
- 2) Make the programs available in any manner to any third party;
- 3) Use the programs to provide third-party training;
- 4) Assign this agreement or give or transfer the programs or an interest in them to another individual or entity;
- 5) Cause or permit reverse engineering or de-compilation of the programs except as expressly provided for in the license terms for any third-party software included in the distribution as described below.
- 6) Disclose results of any program operation, functionality or benchmark tests without our prior consent
- 7) Use any Trinem Software name, trademark or logo.

B.5 Export

You agree that United Kingdom export control laws and other application export and import laws govern your use of the programs, including technical data. You agree that neither the programs nor any direct product thereof will be exported, directly or indirectly, in violation of these laws, or will be used for any purpose prohibited by these laws including, without limitation, nuclear, chemical, or biological weapons proliferation.

B.6 Disclaimer of Warranty and Exclusive Remedies

THE PROGRAMS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. WE FURTHER DISCLAIM ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL WE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OR PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. OUR ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED THE PURCHASE PRICE OF THE PROGRAMS.

B.7 Technical Support

Our technical support team will provide web and email based technical support and updates to you for the programs licensed under this agreement for the duration of any technical support agreement detailed in the purchase order.

B.8 End of Agreement

You may terminate this agreement by destroying all copies of the programs. We have the right to terminate your right to use the programs if you fail to comply with any of the terms of this agreement, in which case you shall destroy all copies of the programs.

B.9 Relationship Between the Parties

The relationship between you and us is that of licensee/licensor. Neither party will represent that it has any authority to assume or create any obligation, express or implied, on behalf of the other party, nor to represent the other party as agent, employee, franchisee, or in any other capacity. Nothing in this agreement shall be construed to limit either party's right to independently develop or distribute software that is functionally similar to the other party's products, so long as proprietary information of the other party is not included in such software.

B.10 Entire Agreement

You agree that this agreement is the complete agreement for the programs and licenses, and the agreement supersedes all prior or contemporaneous agreements or representations. If any term of this agreement is found to be invalid or unenforceable, the remaining provisions will remain effective.

B.11 Acknowledgements and Third-Party Software License Agreements

Some components of *Trilogy* include software licensed from third-party vendors. The following are the license terms for these additional components:

TCL/TK LICENSE TERMS



This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

Crystal Icons Set

Crystal Icon Set, licensed under the LGPL.

TITLE: Crystal Project Icons
AUTHOR: Everaldo Coelho
SITE: <http://www.everaldo.com>
CONTACT: everaldo@everaldo.com

Copyright (c) 2006-2007 Everaldo Coelho.

Index

A

- AllowIfJobRunning, 104, 155
- AllowStop, 210
- ApplyButton, 201
- ApplyButtonText, 202
- AutoRun, 127, 157
- AutoRunDates, 130, 160
- AutoRunDays, 129, 161
- AutoRunInterval, 159
- AutoRunMonths, 131, 162
- AutoRunOnFailure, 132, 165
- AutoRunOnSuccess, 132, 164
- AutoRunStandardInput, 163
- AutoRunTimes, 128, 158
- AutoSelectColumn, 167
- AutoSelectValue, 168
- AutoSort, 73, 166
- AutoStretch, 72, 169

B

- Background Running, 135, 149
- Balloon Notifications, 118
 - Displaying Balloon Immediately, **122**
 - Icon Types, **121**
 - Notify Groups, 119
 - Notifying Client Machines, 120
 - Notifying Invoking Client User, **120**
 - Notifying Users, 120
 - Standard Output As, 122
- Banner, 170
- BannerGraphic, 173
- BannerHeading, 171
- BannerText, 172
- Binary Data
 - Redirecting, 142
 - UseTTY, 142
- Buffering, 141, 142

C

- Cancel Button
 - Renaming, 48, 174
- CancelButtonText, 48, 174
- Checkboxes, 45
 - Linking to Other Fields, 95
- Client Installation Guide - Unix, 23
- Client Installation Guide - Windows, 21
- ColumnNames, 69, 208
- ColumnWidths, 70, 209
- Command Line Options, 135
- ContextColumn, 175

D

- Demo
 - List Box, 260
 - Running, 28, 257
 - System Tray, 261
- DenyIfJobRunning, 103, 176
- Dialog, 177
- Dialogs
 - Allowed Characters in Fields, 99
 - Banner, 60, 170, 171, 172, 173
 - Checkboxes and Radio Buttons, 45
 - Drop-Down Lists, 38
 - Dynamic with Script, 63
 - Grouping Radio Buttons, 56
 - Linking Fields, 91, 186, 187
 - List Box, 67
 - ListBox, 194
 - Named Frames, 54
 - Overview, 31
 - Password Fields, 36
 - Populating Fields, 189
 - Pre-Populating, 188
 - Pre-populating fields, 41
 - Tabbed Dialogs, 58
 - Title, 37, 192
 - Validating, 49
- DialogScript, 178
- Disabling Fields, 40, 48
- DoubleClick, 179
- Drop Down Lists, 38

E

- Environment, 138, 140, 180
- Environment Variables, 12, 147

G

- Group, 181
- Group Processor, 106
 - Creating Custom, 108
 - Plug-In Mechanism, 106
 - trigp_delete_list(), 113
 - trigp_get_groups_for_user(), 111
 - trigp_get_last_update_timestamp(), 110
 - trigp_get_users_in_group(), 112
 - trigp_initialise(), 109
- GroupProcessor, 182
- Groups. *See* User Groups

H

- HelpText, 197

I

Icon, 183

Icons

- Adding to List Box, 82
- Context Sensitive, 82

J

Job Control

- Buffering, 141
- Environment, 138, 140
- Introduction to, 138
- Linking Jobs, 102
- Preventing Execution, 103
- Starting, 140
- Stopping, 146

L

Licensing, 25

Linking Fields, 91, 186, 187

- Disabling Fields, 93, *See* Disabling Fields
- Linking a Field to Itself, 99, *See* Dialogs: Allowed Characters in Fields
- Linking Data Entry Fields, 98
- Linking Drop Down Lists, 91
- Linking Radio Buttons and Checkboxes, 95
- Linking to List Box, 86, 91, 97
- OnFieldChange, 91

Linking Jobs, 102

- Preventing Jobs from Running, 102

List Box, 194

- Adding Apply Button, 73, 201
- Apply Button, 85
- AutoRefresh, 81, 156
- AutoSelectColumn, 76, 167
- AutoSelectValue, 76, 168
- AutoSort, 73, 166
- AutoStretch, 72, 169
- ColumnNames, 69, 208
- ColumnWidths, 70, 209
- Context, 80, 82
- Context Sensitive Icons, 82
- Context Sensitive Menus, 80, 82
- Demo, 260
- Double-Click, 81, 179
- Dynamic Content, 81, 156
- Example, 86
- Height, 73, 204
- Hidden Columns, 72, 209
- Icons, 82
- Limiting Selections, 74, 190
- OnRightClick, 79, 195
- Overview, 67
- Populating with Script, 85, 206
- Right Click Menu, 79, 195
- Row Icons, 82
- Selecting Rows, 75
- Selecting Rows Automatically, 76, 167, 168
- Sorting Automatically, 73, 166

Sorting Manually, 73

Width, 72, 169, 205

ListBoxHeight, 73, 204

ListBoxScript, 206

ListBoxSep, 207

ListBoxWidth, 205

Logfile, 151

M

Menus

- Adding to List Box, 79
- Context Sensitive, 82

N

Notification Area. *See* System Tray

- Configuring for Windows Vista and Windows 7, 115

Notifying Users of Scheduled Job, 128

NotifyRunGroup, 184

O

OK Button

- Renaming, 48, 185
- OkButtonText, 48, 185
- OnFieldChangeUpdate, 91, 186
- OnListBoxChangeUpdate, 97, 187
- OnRightClick, 79, 195

P

Param, 211

Params, 212

Password Fields, 36, *See* Dialogs

PopulateFieldnWith, 38, 39, 40, 41, 43, 47, 48, 49, 52, 91, 147, 148, 177, 180, 186, 187, 188, 189, 226, 250

PopulateListBox, 85, 203

PopulateListBoxWith, 206

PopulateWith, 52, 188

Port, 152

PreValidate, 216

PreValidateWith, 51, 199

Program, 216

R

Radio Buttons, 45

- Grouping with Named Frames, 56
- Linking to Other Fields, 95

RelayPort, 154

RelayServer, 153

Renaming Buttons, 48

S

Scheduler, 127

- AutoRun, 127, 157

- AutoRunDates, 130, 160
- AutoRunDays, 129, 161
- AutoRunInterval, 159
- AutoRunMonths, 131, 162
- AutoRunOnFailure, 165
- AutoRunOnSuccess, 164
- AutoRunStandardInput, 163
- AutoRunTimes, 128, 158
- Dependent Jobs, 132
- Environment Variables set by, 134
- Notifying Users, 128
- Passing Parameters to Scripts, 128
- Setting Environment, 128
- Scripting Engine
 - AddParameter, 224
 - Balloon, 255
 - BalloonToGroup, 256
 - ChooseFile, 236
 - CloseFile, 238
 - EndOfStream, 232
 - execute, 229
 - exitcode, 230
 - GetChangedField, 252
 - GetClientNodeName, 248
 - GetClientUserName, 249
 - GetCurrentField, 251
 - GetDirName, 241
 - GetExecutionReason, 250
 - GetField, 228
 - GetFileName, 242
 - GetInputFileName, 247
 - GetJobName, 244
 - getline, 231
 - GetListBoxField, 246
 - GetListBoxSelectionCount, 245
 - GetPathName, 240
 - OpenFile, 237
 - SetField, 225
 - SetPort, 222
 - SetProgramID, 223
 - SetServerName, 221
 - SetStandardInput, 235
 - SetStream, 239
 - ShowDialog, 226
 - ShowStandardError, 234
 - ShowStandardOutput, 233
 - Wait, 243
 - WriteToStandardError, 254
 - WriteToStandardOutput, 253
- Scripts, 13
- Selections, 190
- Server Installation Guide - Unix, 19
- Server Installation Guide - Windows, 16
- Server Side Job Control, 138
- Sorting the List Box
 - AutoSort, 73
 - Manually, 73
- Standard Error, 13

- Standard Input, 142
 - tee command, 145
 - Translating Line Endings, 145
- Standard Output, 13
- Starting Server - Unix, 28
- Starting Server - Windows, 27
- stderr, 13, 214
- stdin, 215
- stdout, 13, 213
- Stopping running jobs, 146
- Stopping Server - Unix, 30
- Stopping Server - Windows, 30
- System Tray, 114, 115
 - Balloon Notifications, 118
 - Demo, 261
 - Running Jobs from, 114
- SystemTray, 191

T

- Tabbed Dialogs, 58
- tee command, 145
- Title, 192
- TrayMenu, 193
- TRICHANGEDFIELD, 52, 92, 93, 96, 100, 148
- TRICLIENTNODENAME, 51, 120, 147
- TRICLIENTUSERNAME, 51, 120, 147
- TRICURRENTFIELD, 52, 148
- TRIDIALOGNAME, 147
- TRIFIELDn, 33, 46, 69
- Trilogy Client Service, 114
- trilogy.lic, 25
- TRIPASSEDFIELDn, 80
- TRIREASON, 52, 147, 148
- TRISTDINFILENAME, 135, 148

U

- Unix Backup - Example, 262
- User Groups
 - "Group" Directive, 181
 - "GroupProcessor" Directive, 182
 - Controlling Access to Jobs With, 107
 - Group Processor, 106
 - Sending Balloon Notification to, 119
- UseTTY, 142, 198

V

- Validate, 51, 199, 200
- ValidateWith, 49, 200
- Validating Dialogs, 49, 51, 199, 200, 216

W

- Windows Service, 27