SYSPRO Power Tailoring

A developer's guide

This is the second of two books written on tailoring SYSPRO. It will help you understand the various ways in which you can customize your ERP solution if you have development skills.

Paul Hollick

SYSPRO™
Simplifying your Success

SYSPRO Power Tailoring Part 2

Customizing and Personalizing SYSPRO using Development Skills

By Paul Hollick

Disclaimer
Every effort has been made to ensure that the contents of this book are factual and correct. However, this does not mean that the book is free from error. In addition, this book is intended as an introduction to the concepts and tenets of the subject matter and should be used as such. This publication is designed to provide general information regarding the subject matter covered. Each business is unique; laws and practices often vary from country to country and are subject to change. Because each factual situation is different, specific advice tailored to the particular circumstances should be obtained from a qualified expert. SYSPRO Ltd in no way accepts responsibility for any damage or loss incurred due to decisions and actions that are taken based on this book. The authors and publisher specifically disclaim any liability resulting from the use or application of the information contained in this book, and the information is not intended to serve as legal advice related to individual situations.

Foreword

SYSPRO Power Tailoring is a vast topic, covering everything from dragging a column to a new position in a listview, to writing a .NET User Control that uses business objects and is embedded in a customized pane. Some of the new Power Tailoring functionality that was released with SYSPRO 7 (and a few items released with SYSPRO 7 Update 1) opens up amazing possibilities.

This book is the second in the Power Tailoring series. The first covered Power Tailoring that you could perform without possing any development skills. This second book builds on that, and takes you through what can be achieved if you have basic development skills and what you can achieve if you have previously done some development.

SYSPRO's philosophy has always been to empower the users of the product to be self-sufficient. To this effect we supply online help, tutorials, training material, feature demos, reference material, and books. This material is made available via the SYSPRO InfoZone, and assistance can be found on the SYSPRO Forums (from both internal staff, and members of the SYSPRO community).

If you use SYSPRO's Power Tailoring to perform your customization or personalization, you can be assured that upgrading to a later version of the software will be a simple process. After the upgrade, any VBScripting, calls to business objects, customized panes, and all of your personalization will continue to work as it did before.

I hope that you find this book informative and useful.

Phil Duff, Founder and CEO, SYSPRO

Table of Contents

Introduction

No computer software perfectly matches a customer's requirements out-of-the-box, unless it was specifically written for them.

Power Tailoring is the name SYSPRO gives to your ability to tailor both the SYSPRO functionality and user interface to fit your requirements. This includes both customization and personalization. Customization can be understood as the modification of a product according to individual, organizational or regional preferences while personalization is the process of tailoring the product and screens to an individual's needs or desires.

Most people's idea of customization is something that takes months or years to perform. It also requires many meetings to define the scope, construct a specification, sign-off the specification, progress meetings, user acceptance testing, and final acceptance sign-off. Only then do you really know what you have, and whether it will work for you.

Depending on how the customization was done, you may be tied to a specific version of the software, or have to redo the customization again on a later version when you upgrade (with all the testing and costs associated with this).

The idea behind Power Tailoring is that small and medium-sized customization should be able to be performed by someone within the organization. With a little training it should also be possible for them to perform the big customizations. This should significantly reduce the time and costs of any changes.

We also don't want you to be stuck on a particular version of the software, so the Power Tailoring that you perform (including the VBScripting, and use of the business objects) should continue to work when you upgrade to a later version. Again, significantly reducing the costs involved, and giving you peace of mind.

It was at the end of 2010 when I first discussed writing a book on Power Tailoring with SYSPRO's CEO, Phil Duff. At that time I was performing all the first-line testing of the new Power Tailoring features in SYSPRO 6.1, as they were being developed. I was also writing tutorials on how to use these new features. We wanted one place for people to go to find out everything about SYSPRO's Power Tailoring capabilities, so the idea of a book was born.

We originally thought that it would be a small book of around 200 pages, and take around three months to write. However, once I started going through all the existing Power Tailoring features in the

product, and then looking at those that were planned, it became obvious that this was going to be a much bigger job.

It also became clear that there were two distinct audiences for the book; one that wanted to use the standard options to perform their Power Tailoring, and another that wanted to use everything that was available. Once the book was started, I realised that to cater for these audiences would require the book to be written in two parts. As it got bigger it became obvious that it was two completely separate books, and the decision was made to separate them.

The first book (covering all the power tailoring that could be performed without development skills) ended up being 250 pages long. It was completed at the end of 2011 and was released in February 2012.

The second book (the “full fat” version) was started as soon as the first book was released. This was at the beginning of SYSPRO 7 development, so it was decided that the book should also contain all the SYSPRO 7 Power Tailoring items. SYSPRO 7 contains much more Power Tailoring functionality than the previous versions (such as Flow Graphs, Tiles, the Application Builder, etc.). One of the drawbacks of writing a book alongside the development of the product is that when changes are made to the product you have to retrofit them into the book.

Three years later the book is now available.

No book is completed without the assistance of others. The first thank you must go to Phil Duff who gave me the opportunity to write these books. Before embarking on this process I had never written anything longer than a tutorial. I hope that his confidence that I could do this was well-founded.

The second person that I must thank is Benita Ravyse, who spent many long hours testing that everything I wrote was true. I think that she derived personal satisfaction from every error she found. I also think that she received great joy at seeing my face drop when she gave me back her changes for each chapter. Each page that required a change had a little sticker on it. Originally, I think each color meant something specific, but she soon ran out of some colors and then it was “any color will do”. For her sins, Benita now performs all the first-line Power Tailoring testing.

The final word of thanks must go to Even Nesset, who had the unenviable task of taking what I thought was the finished book and proofreading it (for grammar, style, tense, consistency, etc.). Now there is four weeks of his life that he will never get back (and he has just corrected this sentence…).

Volume 1 (Chapters 1 – 9)

Chapter 1 - An e.net Solutions Overview

Before jumping straight in and explaining what customization can be done, it is worth taking the time to get a basic understanding of e.net Solutions, and its components. Once you get beyond very basic SYSPRO customization you are likely to interact with e.net Solutions business objects to post or retrieve data.

What is e.net Solutions?

Put simply, e.net Solutions is a framework that makes it easier to get information into (and out of) SYSPRO without using the normal SYSPRO user interface. It uses the existing SYSPRO security, configuration options, and operator settings. This framework consists of a COM object, a WCF service, a DCOM client, business objects, web services, Web-Based Applications, and the Document Flow Manager. Each of these items will be covered in more detail below.

The basic design tenets for e.net Solutions were that it must be simple to use, must be extensible, must be version independent, and must use the existing SYSPRO security and settings. It must also shield the complexity of the SYSPRO database from the developer, to discourage developers from accessing the database directly (with all the problems that can arise if done incorrectly).

Basic SYSPRO architecture

A typical SYSPRO installation consists of an application server that contains most of the SYSPRO programs, and one or more clients that contain the user interface (and some more programs). The data can be stored in SYSPRO's own data format on the application server, or within Microsoft SQL Server. Microsoft SQL Server can reside on the SYSPRO application server, or it can be on its own separate server. Services are running on the SYSPRO application server that look out for incoming requests from SYSPRO clients.

When an operator runs SYSPRO, some programs and the user interface are loaded into the client workstation's memory, and some programs are loaded into the application server's memory. After entering their credentials the operator clicks on the *Login* button. A call is made to the application server, and the required programs are loaded into its memory. Data is moved back and forth between the user interface on the client workstation, and the program on the application server, in what is known as a *datablock*. **Figure 1-1** shows the basic SYSPRO architecture

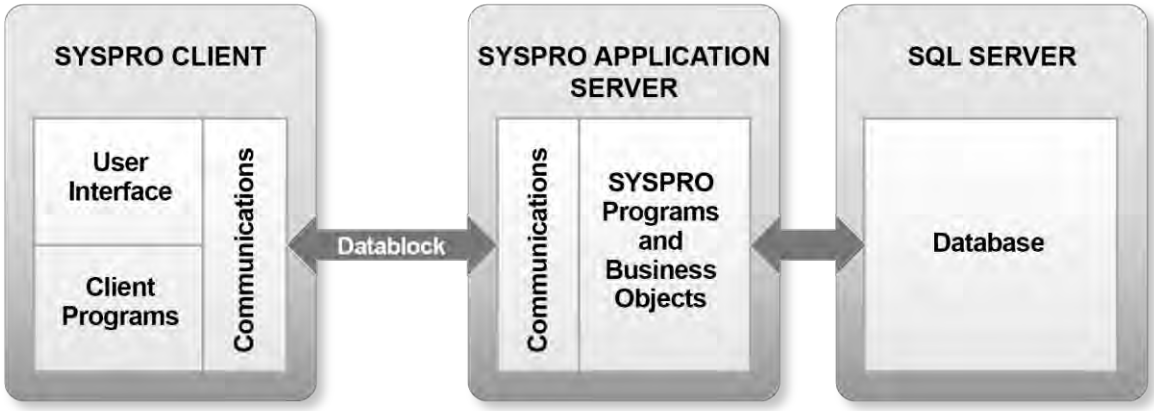


Figure 1-1: The basic SYSPRO architecture

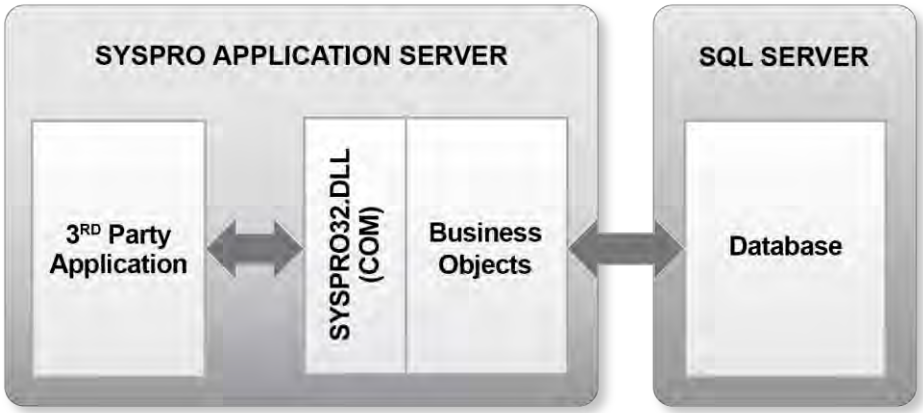
Basic e.net Solutions Architecture

The e.net Solutions architecture does away with the need for the SYSPRO user interface. Business objects are pieces of SYSPRO business logic that can be called programmatically from outside of SYSPRO. They are passed instructions in XML, and after processing the request (updating the database if required) they return the results in XML. In many places within SYSPRO the SYSPRO programs call business objects to retrieve information, and all of the standard reports in *SYSPRO Reporting Services* use business objects to retrieve their data.

Most of the interaction with the business objects is done through the e.net Solutions COM object which is called `SYSPRO32.dll` (or `SYSPRO64.dll` in a 64-bit environment). The COM object for versions prior to SYSPRO 7.0 was called `Encore.dll`.

As one of the design tenets of e.net Solutions is that any development should continue to work when a site upgrades to a new version, `Encore.dll` is still included for this backwards-compatibility. This e.net Solutions COM object exposes the business objects using a COM interface. This COM object must reside in the SYSPRO `Base` folder on the application server, and the business objects must reside in the SYSPRO `Program` folder on the application server.

There are several ways to interact with the business objects. The simplest is if the business objects are to be called from a third party application or script that resides on the SYSPRO application server. The COM object receives the request from the application and calls the specified business object, passing it the XML that was supplied. The business object processes the XML and returns the resulting XML to the COM object, which routes it back to the application (see **Figure 1-2**).



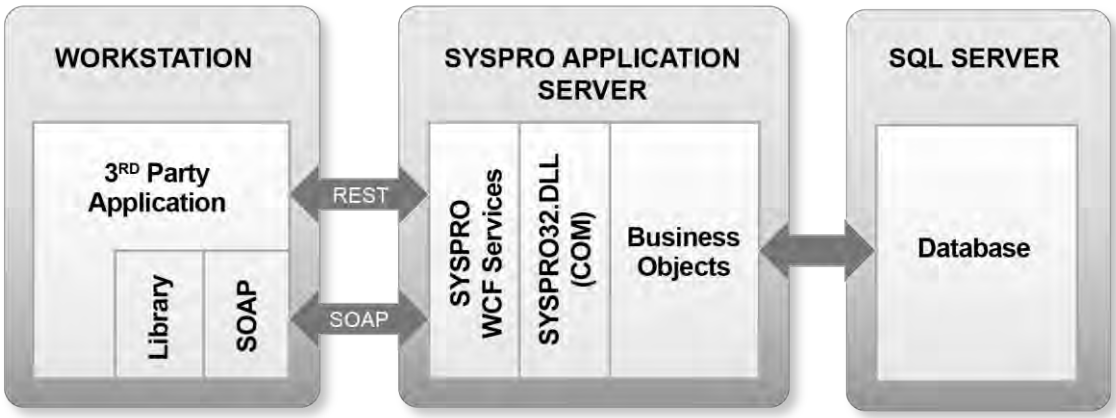
NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-2: Calling the business objects from the SYSPRO application server

If the business objects are to be called from a third party application that resides on a computer other than the SYSPRO application server, you need a means of routing the request to the business objects on the SYSPRO application server.

There are many ways to achieve this. The simplest (and newest) is when SYSPRO WCF Services is installed on the SYSPRO application server (see **Figure 1-3**). This exposes both SOAP (Simple Object Access Protocol) and REST (Representation State Transfer) endpoints.

A third party application running on a remote workstation can use either protocol to talk to the WCF service on the SYSPRO application server. The WCF service then communicates with the business objects via the COM object. If using the SOAP protocol, additional libraries are provided for the workstation to simplify the development process. Both SOAP and REST allow for communication over HTTP, enabling remote e.net Solutions communications to work over the internet.

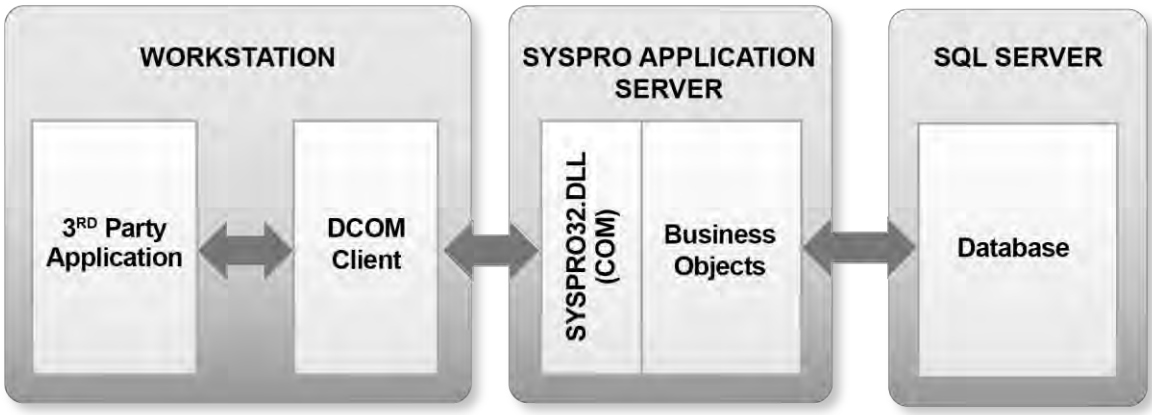


NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-3: Calling the business objects via *SYSPRO WCF Services*

A second way to do the remote calls is by installing the SYSPRO *Remote DCOM Client* on the workstation. When a call is made from the application to the COM object, DCOM intercepts the call on the workstation and routes it to the COM object on the SYSPRO application server. The COM object then passes the request to the business objects. Once the request has been processed the results are routed back via DCOM to the script that originally made the call on the remote computer. This process can be seen in **Figure 1-4**).

DCOM uses TCP port 135 to start its communication. As it is unlikely that port 135 will be open on most firewalls it is extremely unlikely that you could use e.net Solutions over the internet if you are using DCOM. Therefore, DCOM should only be used on an organization's internal network.

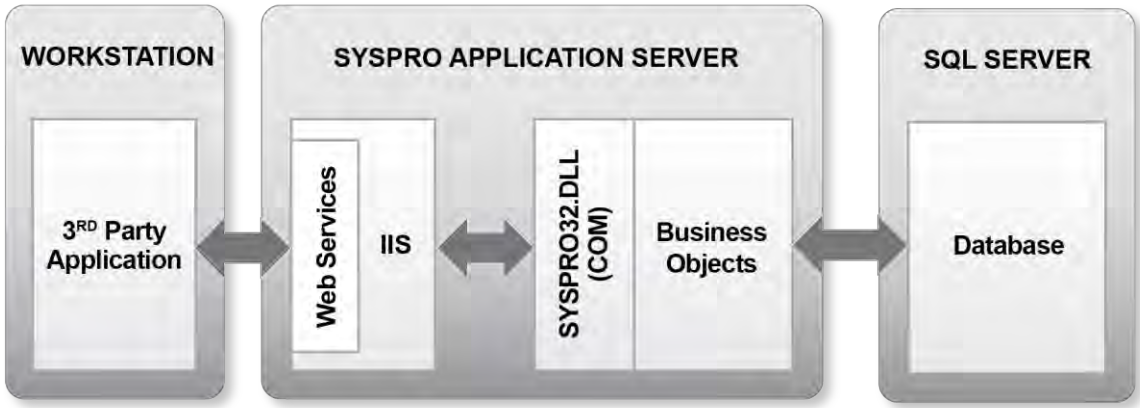


NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-4: Calling the business object from a remote computer using DCOM

Another way to talk to the business objects from a remote computer is using the SYSPRO *Web Services*. The web services expose the functionality of the business objects through a web interface. These are typically used where the third party application is being run on a machine on a different network/over the internet. There is a significant overhead when using e.net Solutions with Web Services, as all the messages must be wrapped in a SOAP envelope which consumes more bandwidth, and the envelope needs to be added/removed at each end.

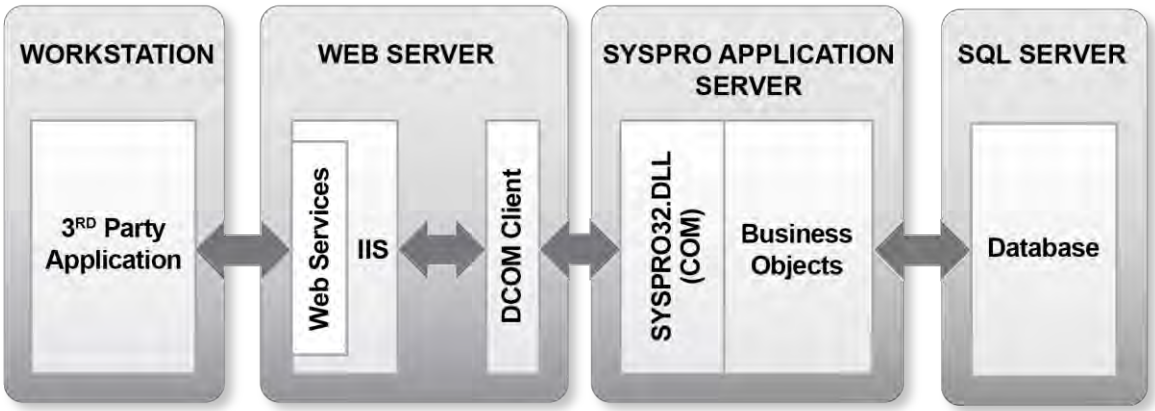
The web services also require a Microsoft IIS web server to be able to run. This IIS server could be installed on the SYSPRO application server, in which case the web services would talk directly to the COM object. **Figure 1-5** shows the scenario where IIS is installed on the SYSPRO application server.



NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-5: Calling the web services when IIS is on the SYSPRO application server

If the web services are installed on a web server that is not the SYSPRO application server, DCOM will still need to be configured between this web server and the SYSPRO application server. The application will call the web services on the IIS server. The web services will attempt to talk to the COM object, and this request will be intercepted by DCOM and routed to the COM object on the SYSPRO application server. The COM object will talk to the business objects and the response from the business objects will be routed back via the COM object to DCOM, and onto the web services. **Figure 1-6** shows this scenario.



NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-6: Calling the web services when IIS is not on the SYSPRO application server

The *Document Flow Manager* (DFM) is a means of automating the processing of incoming XML documents via the business objects. **Figure 1-7** shows how the DFM works. *Contracts* are used to configure what type of document to look for, and where it is located. Documents can be picked up from folders, or as attachments to emails, and passed to the *Microsoft Message Queue* (MSMQ) along with the contract information.

When they reach the top of this queue they are processed and the relevant business object is called. The results can be written out to files to specified folders, or attached to emails. The DFM and MSMQ must reside on the SYSPRO application server, and calls to the business objects are direct (they do not go through the COM object).

The contract can contain an XSL/T code to transform the XML file before it is passed to the business object, and it can also contain some to transform the output from the business object.

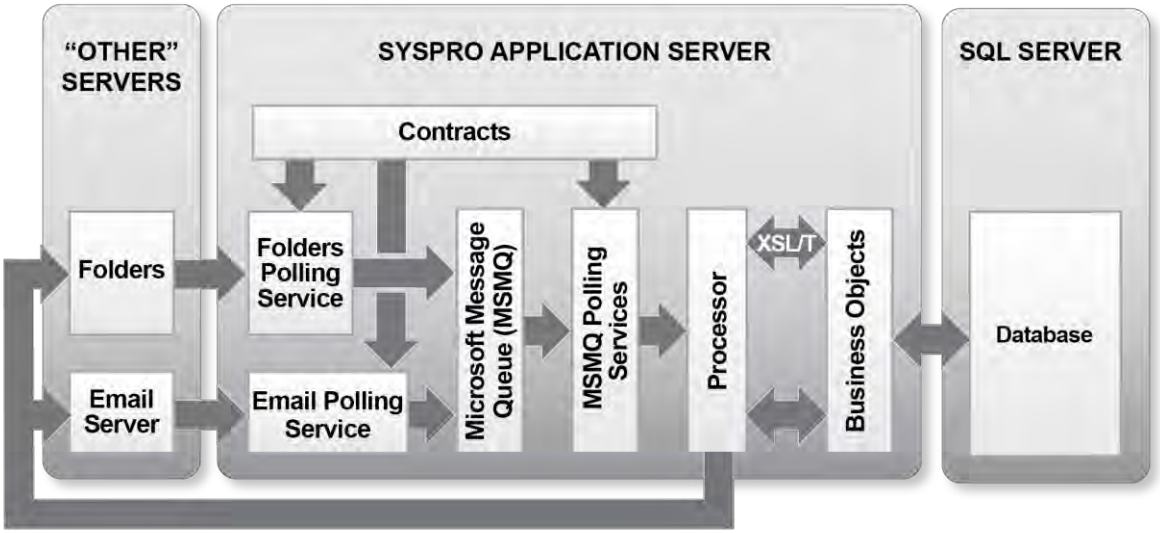
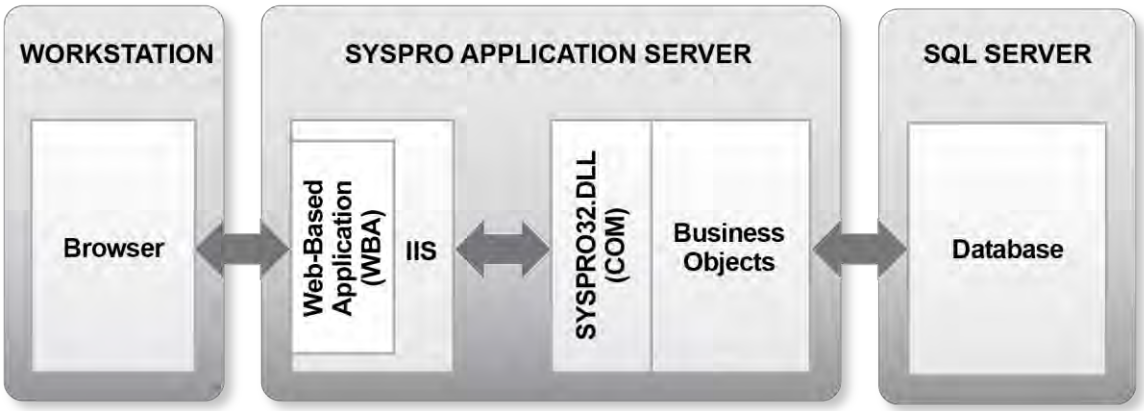


Figure 1-7: The *Document Flow Manager*

The SYSPRO *Web-Based Applications* (WBA) is a web version of some of the SYSPRO programs. The functionality of a WBA program is a subset of the SYSPRO program's functionality. The WBA are installed under Microsoft IIS, either on the SYSPRO application server, or on a remote IIS server.

If a remote IIS server is used, the WBA will use WCF, Web Services, or DCOM to talk to the business objects on the SYSPRO application server. The WBA ships with its source code, which enables developers to extend its functionality.

Figure 1-8 shows the WBA installed on the SYSPRO application server. When calls are made they go directly to the COM object. **Figure 1-9** (later in this chapter) shows the WBA installed on a remote IIS server, and it is using the *Remote DCOM Client* to intercept to call to COM and route it to the SYSPRO application server.



NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-8: The *Web-Based Applications* installed on the SYSPRO application server

The business objects can also be called from within SYSPRO, such as from a customized pane. There are several ways that this can be done, and this will be covered in more detail later in this book.

Overview of the Different Communication Methods
The diagrams above show the different ways that e.net Solutions can be configured. Most of these cover the scenarios where the third party application is on a machine other than the SYSPRO application server. To finish off this section, **Table 1-1** contains a table that explains the benefits and drawbacks of each of these communication methods.

Until recently, the only options available for remotely calling the e.net Solutions business objects were DCOM and the web services. Each of these had environments where they were better suited than the other. For example, if you wanted to run over the internet, or through a firewall, web services were the only way as DCOM would not work in this environment, but you needed IIS to be installed. DCOM was significantly faster than web services, but could be difficult to configure in some circumstances, and if SQL Server was using NT Authentication in a 3-tier environment it would not work as the credentials are not passed through to the SQL Server (this is known as the double-hop environment).

The *SYSPRO WCF Services* communication environment has none of these restrictions and is the recommended way of communicating with the business objects if the application is not running on the SYSPRO application server. **Table 1-1** shows the different communications methods, along with the benefits and drawbacks of each.

	Benefits	Drawbacks
DCOM	Faster than web services	Can be difficult to configure Will not work over internet/firewalls Does not support “double-hop”
Web Services	Works over internet/firewalls	Uses more bandwidth Additional overhead for envelope Requires IIS to be installed
WCF	Works over internet/firewalls Can be dedicated to specific firewall port Can add extra endpoints Faster than web services	

Table 1-1: A table showing the benefits and drawbacks of each method of remotely communicating with the business objects

The COM Object

The *Component Object Model* (COM) provides a language-neutral interface that allows developers to interact with the component without knowing the intricacies of what is happening behind the scenes. The business objects are exposed through a COM Object called `SYSPRO32.dll` that resides in SYSPRO's `Base` folder on the SYSPRO application server. `SYSPRO32.dll` contains four classes:

- *Utilities* - used for logging on/off, retrieving operator preferences and calling a third party application written using Net Express (that does not have a user interface).
- *Query* – used for calling query business objects. The queries do not update or change the underlying data in the database; they simply retrieve data, applying business rules to the returned information.
- *Setup* - used to create and modify semi-static items such as stock codes and customers.
- *Transaction* - used to post transactional data like inventory movements, sales orders and invoices.

Each class contains several methods.

The *Utilities* class consists of the following methods:

- *Logon* - to logon to e.net Solutions
- *Logoff* - to logoff from e.net Solutions
- *GetLogonProfile* - to retrieve information/user preferences for the currently logged on session
- *Run* - to run a Net Express program

The *Query* class consists of the following methods:

- *Browse* - to generically return specified columns from all rows of the named table
- *Fetch* - to generically return all columns from a specified row in the named table
- *Query* - to access query-type business objects
- *NextKey* - to generically retrieve a key alphabetically higher than the supplied key
- *PreviousKey* - to generically retrieve a key alphabetically lower than the supplied key

The *Setup* class consists of the following methods:

- *Add* - to add single or multiple items with a single call
- *Update* - to change single or multiple items with a single call
- *Delete* - to delete single or multiple items with a single call

The *Transaction* class consists of the following methods:

- *Build* - to retrieve information helpful during the post
- *Post* - to post transactions

Business Objects Overview

A business object is a SYSPRO program that does not have a user interface. It resides in the SYSPRO `Program` folder and is designed to accept text in the form of XML, and return XML.

Many of the earliest business objects were queries, and had the same functionality as the matching SYSPRO program. However, the later business objects tend to be subsets of the SYSPRO program, and do very specific jobs, so two or more business objects provide the same functionality as one SYSPRO program. For example, in the case of the *Inventory Movements* program, there are nine business objects to provide the same functionality (one for *Receipts*, one for *Cost Changes*, one for *Transfers Out* etc.). Business objects can be thought of as the building blocks that enable you to construct your own solution.

Business objects use the same security model as the core SYSPRO product, so you do not need to build this into your solution. The security configuration is handled from within SYSPRO, and configured against the operator code, operator group, or role. There are additional *e.net* options against the operator *Activities*, *Electronic Signatures*, *Events*, and *Triggers*, to allow for fine-tuning your requirements. If an operator is prevented from seeing specific information within SYSPRO (such as costs, a particular warehouse, bank account details, etc.) the XML that the business object returns to them will not contain this information.

Business object names follow a naming convention similar to SYSPRO programs in that they are six characters long, and the first three characters refer to the module to which they belong. For example, *APS* for *Accounts Payable*, *INV* for *Inventory Control*, *COM* for business objects that are included with the *System Manager*, and *IMP* for business objects that are used across multiple modules. The fourth character of the business object name specifies the type of business object (*Q* for *Query*, *S* for *Setup*, *T* for *Transaction* and *R* for *Build*). For example, the inventory query business object is ***INVQRY***, and the business object to post a sales order is ***SORTOI***.

To use a business object outside of SYSPRO it must be licensed (see the *Licensing* section below), and you must logon to e.net Solutions first. The logon is performed using the *Logon* method that is part of the *Utilities* class, and requires you to supply certain pieces of information such as the operator code and operator password. If the logon is successful a 34-character *UserID* will be returned. This *UserID* must be supplied each time that the business object is called, and if the *UserID* is valid, the business object is accessed using the permissions associated with the operator that performed the logon.

Licensing Overview

A SYSPRO Licensing Overview

SYSPRO is licensed concurrently, which means that when an operator logs in they consume one license, no matter which part of SYSPRO they are using. When they log out the license is available for anyone else to use. If you have a 16 user license for SYSPRO, any 16 users can be logged in at any one time. If you need to have a 17th user logged in, you must license the next user band for all modules, even if the operator will only ever use one module.

Also, if you want to license an additional module, you must license this for the user band that you have currently have, even if only one operator will ever use this new module. So if you have a 16 user system and you purchase a new module, this must be licensed for 16 users too.

An e.net Solutions Licensing Overview

To call a business object from outside of SYSPRO requires it to be licensed for the current operator code. The idea of the e.net Solutions licensing is “pay only for what you use”, and the e.net Solutions business objects are licensed in a different way to the core SYSPRO product. Instead of using concurrent licensing based on the modules that you have, the business objects are licensed by *Functional Area* and *Operator*.

A functional area is a subset of a module. For instance, *Accounts Payable* has two functional areas, *Accounts Payable Primary Query* that contains the queries, and *Accounts Payable Primary Posting* that contains all the posting business objects.

Inventory Control has seven functional areas, *Inventory Primary Query*, *Inventory Primary Posting*, *Inventory Product Configurator, Inventory Stock Take System*, *Inventory Goods in Transit System*, *Inventory Inspection System*, and *Inventory Serial Tracking*.

So if you have several employees working in the warehouse that processes inventory movements, only this functional area needs to be licensed for them, not all the modules, or even the *Inventory Control* module. Functional area licenses are available in multiples of 10.

Another way that the e.net Solutions licensing differs from the SYSPRO licensing is with the concept of a *Named User*. Instead of each user consuming a license from the pool when they login, and releasing it back to the pool when they logout, a license for a functional area is allocated against a named operator code, and only that operator code can consume that individual license.

If an operator is accessing a business object from within SYSPRO (such as with a *Customized Pane* or *Associated Pane*) and they are using the built in calls (such as *CallBO*, *CallTrn*, etc.) they do not require an e.net Solutions license, as they are already consuming a SYSPRO license.

When using the *Document Flow Manager*, its license includes it accessing any of the business objects. They do not need to be licensed separately for the DFM to use them. However, this only covers the DFM using them. If you want to access them using another application as well, then they will need to be licensed for this application to use them.

The licensing of the *Web-Based Applications* is also done using functional areas, but the functional areas for the WBA are different to the normal business object functional areas.

A detailed section on licensing appears in *Chapter 2*.

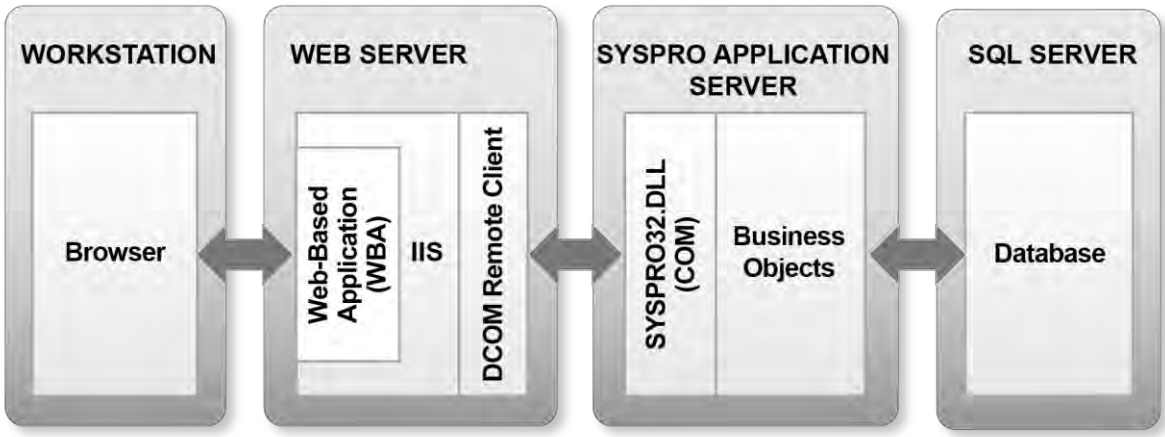
Web-Based Applications

SYSPRO *Web-Based Applications* (WBA) provides a mini web version of SYSPRO. All interaction between the WBA and the SYSPRO data is done through e.net Solutions business objects, and no direct access of the database is performed. The WBA only contains the web user interface for some of the functions provided within SYSPRO, such as sales order entry, stock movements, requisitions, queries, etc., and some of these user interfaces only cater for a subset of the functionality of the equivalent SYSPRO program.

The WBA user interface is written in ASP.Net, and the source code for the user interface is provided with the WBA so that a developer can extend the functionality of the existing applications, or supply new ones and seamlessly add them to the menu system.

If the WBA is being used by internal staff members, it could be installed on Microsoft IIS on the SYSPRO application server. However, if external people will be accessing the WBA this is not

recommended as it can become a security risk. If the IIS server and SYSPRO application server are on different servers, one of the other methods described in the *Basic e.net Solutions Architecture* section above must be used to enable the WBA to talk to `SYSPRO32.dll` on the SYSPRO application server. In **Figure 1-9** the browser is accessing the WBA on the standalone web server. When the WBA call a business object via COM, DCOM intercepts this call and routes it to COM on the SYSPRO application server, which talks to the business object. Responses from the business object are routed back to the WBA via COM/DCOM.



NOTE: SYSPRO32.DLL is for 32-bit systems from SYSPRO 7 onwards
SYSPRO64.DLL is for 64-bit systems from SYSPRO 7 onwards
ENCORE.DLL is for systems prior to SYSPRO 7

Figure 1-9: The *Web-Based Applications* installed on a standalone web server

Document Flow Manager

The *Document Flow Manager* (DFM) is a means of automating the processing of documents from either inside, or outside of the organization. It consists of three separate services that are run on the SYSPRO application server, and configurable contracts. The contracts configure where to look for the document, what type of document to look for, what to do with a document when it is found, which business object to call, and what to do with the response from the business object. This can include transforming the document using XSL/T before and/or after processing by the business object.

Of the three services, the first service polls folders at a set interval looking for documents that have arrived. If it finds a file matching the criteria set out in the contract it picks it up and places it in the Microsoft Message Queue (MSMQ) along with the contract details. The second service polls email

accounts on a Microsoft Exchange Server looking for attachments that match the contract criteria. If attachments are found it places the attachment into the MSMQ along with the contract details.

The third service polls the MSMQ looking for items placed in the queue by either of these other two services, then picks them up along with the contract details. If the contract specifies that the XML document should be transformed using XSL/T before processing it, performs the transformation. It then calls the business object and passes the document to it. The response from the business object can be written out to a folder or emailed (and if specified by the contract, the document can be transformed first).

If an error is returned by the business object, three error message XML files are written to the error folder. The first contains the original XML document before any XSL/T transformation; the second contains the document after any transformation, and the third contains the error message.

Chapter 2 - Using Business Objects

What is a Business Object ?

A business object is a piece of SYSPRO business logic that does not have a normal SYSPRO user interface. It is designed to accept its input in the form of XML, and return its response in XML. Because it does not have a user interface it cannot be "chatty" like some SYSPRO programs. Some of these programs get to a certain point, display a prompt, and then process differently depending on the operator's response. A business object needs to have all the information required before it can start processing, as it can't go back to the operator for clarification. Because of this, most business objects that post information perform a subset of the functionality of the equivalent SYSPRO program. For example, the eleven primary functions performed by the *Movements* program within the *Inventory* module are covered by nine separate business objects.

Security

As a business object uses SYSPRO business logic, it conforms to the same security model as the core SYSPRO product. Therefore you do not need to build this into your solution. There are additional operator *Activities*, *Electronic Signatures*, *Events*, *Triggers*, etc., that are specific to e.net Solutions to help you fine-tune your security. These appear in **Figures 2-1**, **2-2**, **2-3** and **2-4**.

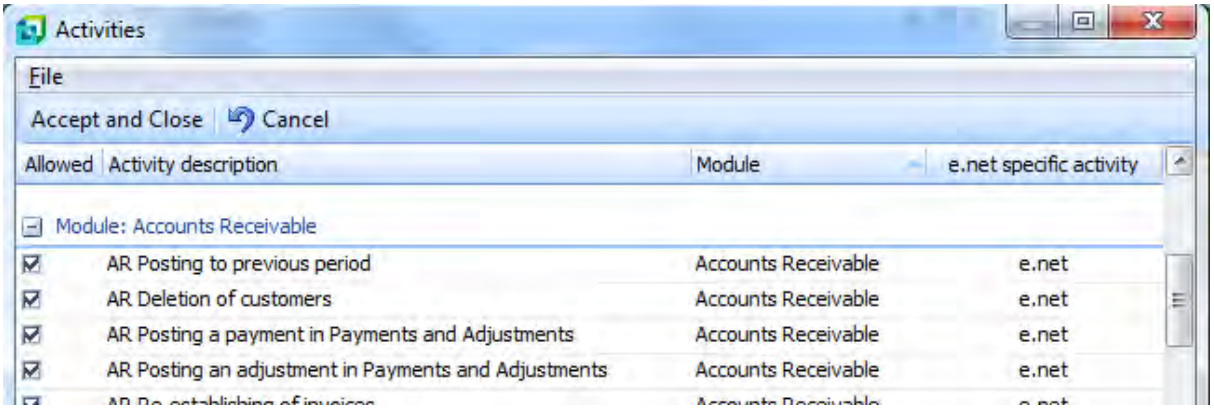


Figure 2-1: Some of the e.net Solutions operator *Actions*

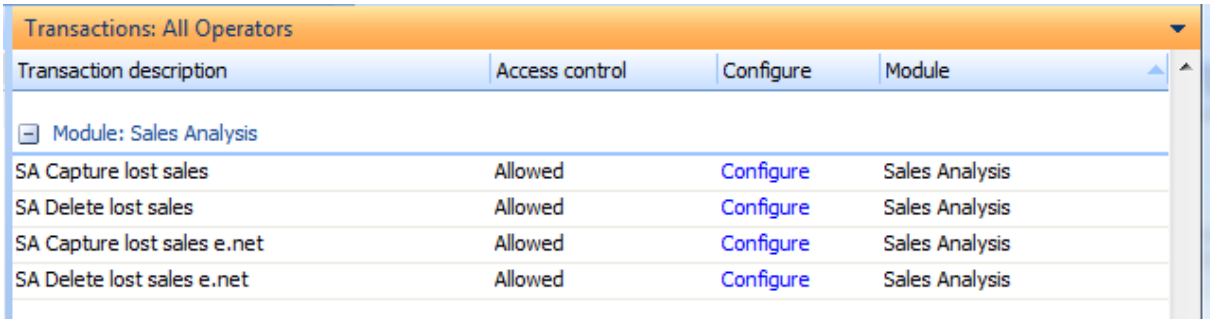


Figure 2-2: Two of the four displayed *Electronic Signatures* are specific to e.net Solutions

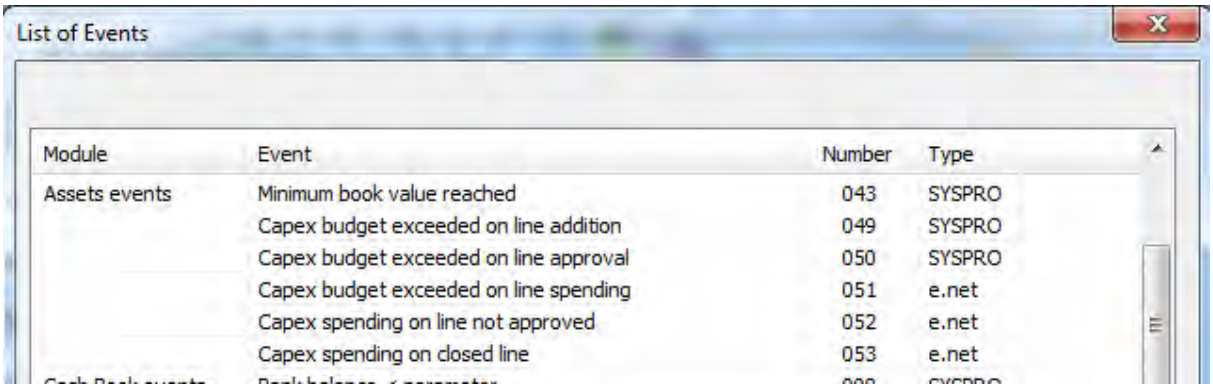


Figure 2-3: Some of the e.net Solutions *Events*

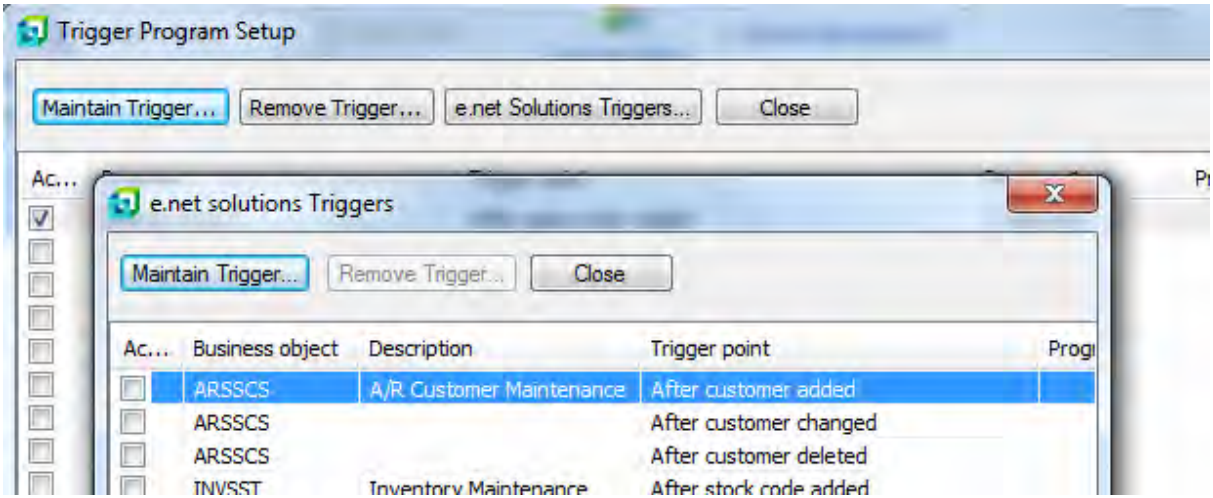


Figure 2-4: The separate e.net Solutions *Triggers* screen

If an operator is prevented from seeing specific information within the core SYSPRO product (such as costs, a particular warehouse, bank account details etc.) the XML that the business object returns to them will also not contain this information.

Access to the business objects can also be restricted at operator group level. Operator groups that have the *System administrator* option checked are not restricted from accessing any business objects. Operator groups without the *System administrator* option checked can be used to restrict which business objects can be accessed (along with which SYSPRO programs can be accessed). Within the *Security Access* listview against the operator group are four "Modules", which are e.net Solutions – Query, e.net Solutions – Setup, e.net Solutions – Transactions, and e.net Solutions – Utilities (see **Figure 2-5**). These relate to the e.net Solutions classes. Within each of these modules are the business object entries, and against each of these is an *Access allowed* checkbox that allows/restricts access.

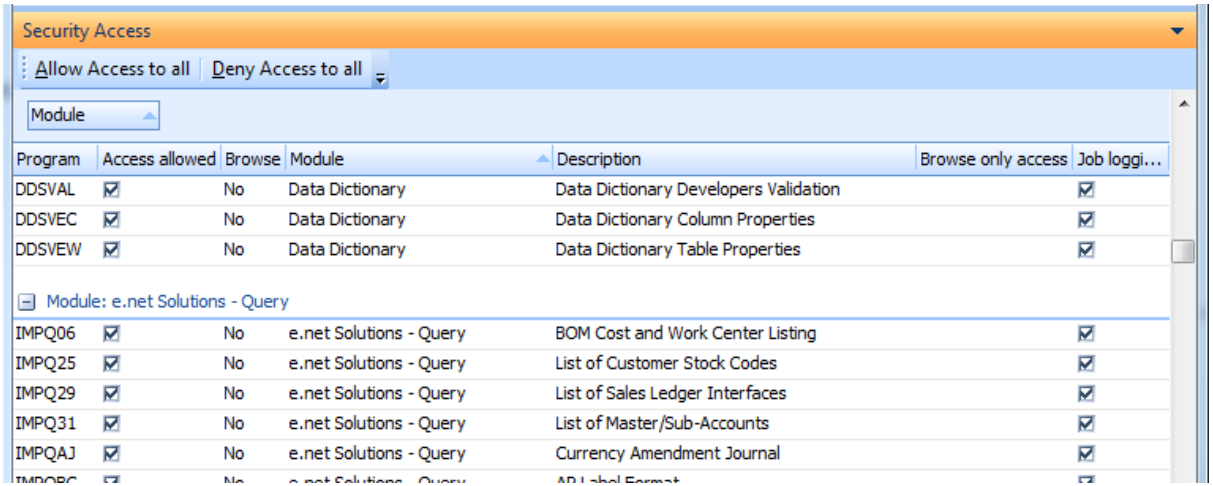


Figure 2-5: An operator group with the *System administrator* option checked

An operator can be restricted to only accessing certain customers using the *Customers* option against the *Access control* section on the *E.net* tab of *Operator Maintenance* (see **Figure 2-6**). The default is that the operator can access all customers, and the restriction only comes into effect if the *Selection* radio button is selected. The restriction can be to only access customers linked to the specified salesperson, or to the supplied list of customer numbers. There is also an option to include only these customers in the results from the browse and fetch business objects.

Also on the *E.net* tab of the operator maintenance program is the *Customer order* screen. This contains additional restrictions and preferences that come into effect for the business objects when sales orders are being processed (**see Figure 2-7**). The defaults can be overridden by supplying the relevant information to the business objects.

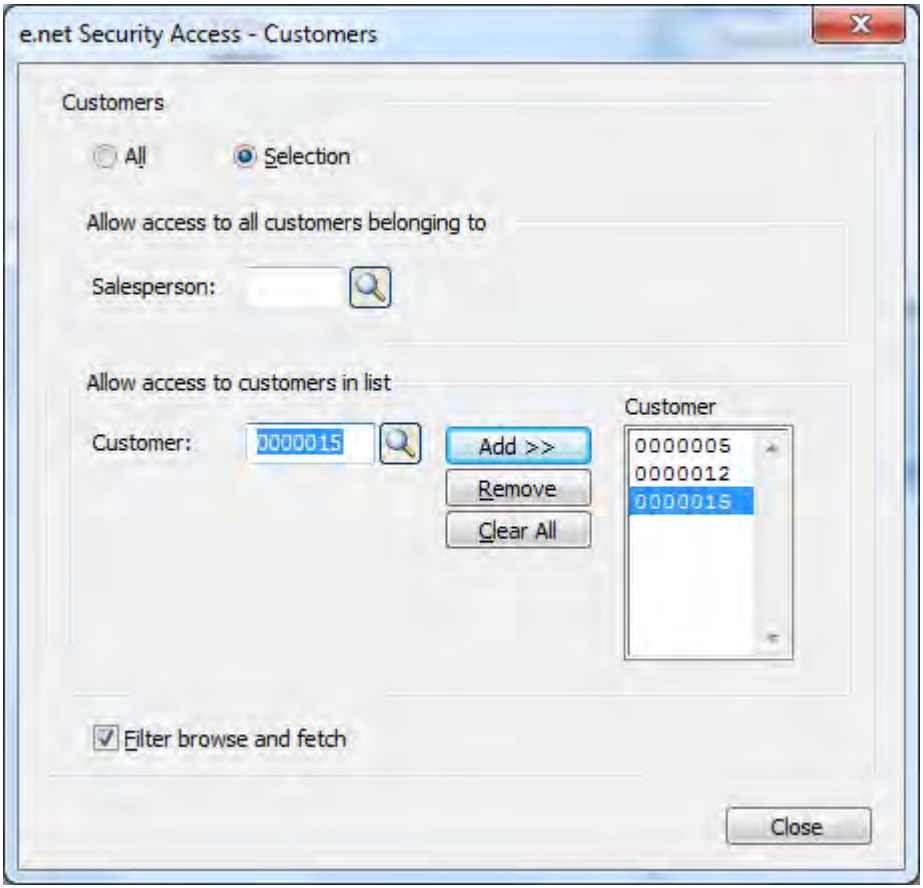


Figure 2-6: The screen used to restrict which customers an operator can see.

SYSPRO32.dll / SYSPRO64.dll / Encore.dll

The Component Object Model (COM) provides a language-neutral interface that allows developers to interact with the component through a standardized interface, without knowing the intricacies of what is happening behind the scenes. This interaction can be from the same machine as the COM object, or across different machines using DCOM. SYSPRO's COM object is called `SYSPRO32.dll` (for 32-bit systems, `SYSPRO64.dll` for 64-bit systems, and `Encore.dll` for systems before SYSPRO 7) and it resides in the SYSPRO `Base` folder on the SYSPRO application server. The functionality of the business objects is exposed through `SYSPRO32.dll`/`SYSPRO64.dll`/`Encore.dll` using Classes and Methods.

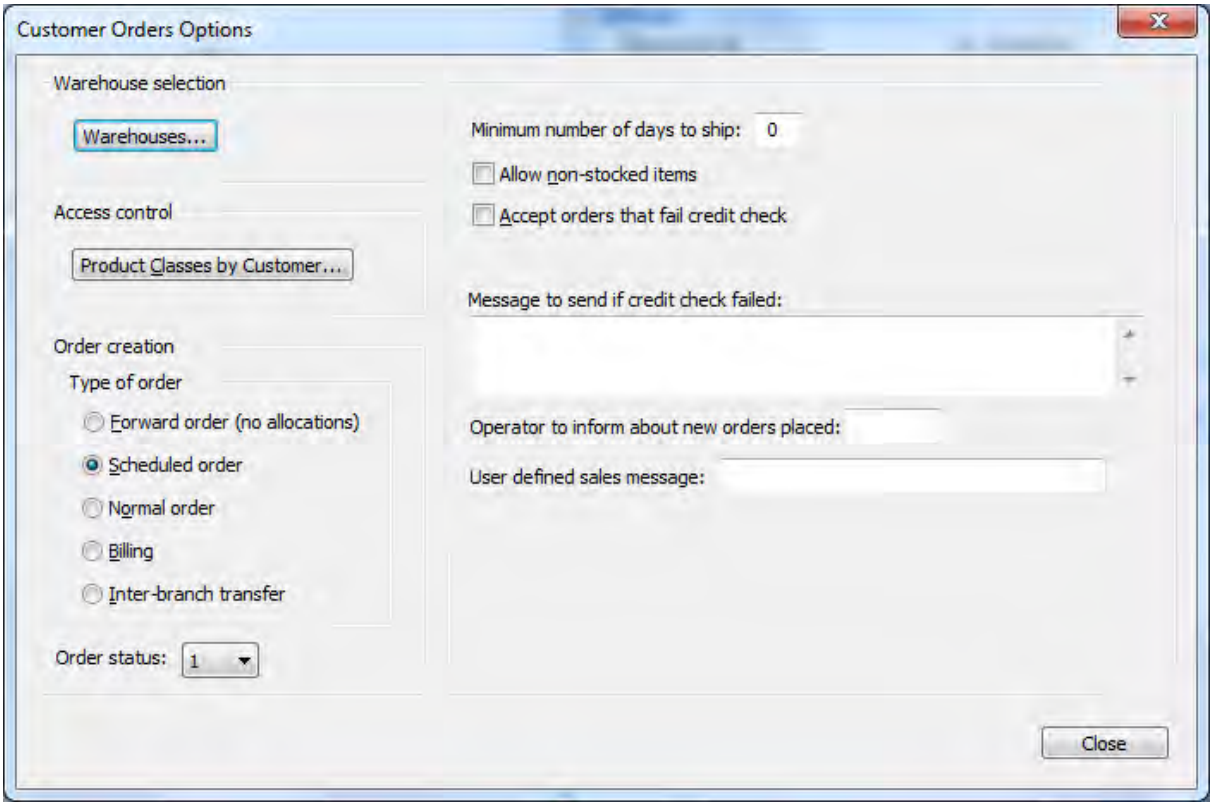


Figure 2-7: Sales order restrictions and defaults.

The e.net Solutions Classes

This SYSPRO COM object exposes four Classes:

- *Utilities Class* – used for logging on/off, retrieving operator preferences and calling third party applications written using Net Express that do not have a user interface.
- *Query Class* – used for calling query business objects. The queries do not update or change the underlying data in the database; they simply retrieve data and apply business rules to the returned information.
- *Setup Class* – used to create and modify semi-static items such as stock codes and customers.
- *Transaction Class* – used to post transactional data like inventory movements, sales orders and invoices.

The e.net Solutions Methods

A method is an instruction to call a specific routine within a business object, and each e.net Solutions class contains several methods.

The *Utilities* class consists of the following methods:

- *Logon* – to logon to e.net Solutions
- *Logoff* – to logoff from e.net Solutions
- *GetLogonProfile* – to retrieve information/user preferences for the currently logged on user session
- *Run* – to run a Net Express program

The *Query* class consists of the following methods:

- *Browse* – to return specified columns from all rows of a table
- *Fetch* – to return all columns from a specified row in a table
- *Query* – to call query-type business objects
- *NextKey* – to retrieve a key from a specified table that is alphabetically higher than the supplied key
- *PreviousKey* – to retrieve a key from a specified table that is alphabetically lower than the supplied key

The *Setup* class consists of the following methods:

- *Add* – to add single or multiple items with a single call
- *Update* – to change single or multiple items with a single call
- *Delete* – to delete single or multiple items with a single call

The *Transaction* class consists of the following methods:

- *Build* – to retrieve information helpful during the post
- *Post* – to post transactions

Some business objects contain multiple methods (such as those belonging to the *Setup* class that each contain the *Add*, *Update*, and *Delete* methods), whereas others only contain one (such as those belonging to the *Transaction* class where there is one business object for the *Build* method, and a separate one for the *Post* method).

The e.net Solutions UserID and the Logon Process

To use a business object outside of SYSPRO you must be logged-on to e.net Solutions. When you successfully logon you will receive a 34-character *UserID*. This is made up of a 32-character GUID (Globally Unique IDentifier, which is a number so large that the probability of the same number being generated randomly twice is negligible). This is followed by a single digit e.net Solutions instance number, which specifies which instance of e.net Solutions to use if more than one copy resides on the machine. The final digit can contain a 0, 1, or 2. If this is 0 it means that this is a normal e.net Solutions user. If it is a 1 it means that this is the first of two possible SYSPRO Espresso users using this login, and if it is a 2 it means that it is the second. Below is an example of an e.net Solutions *UserID*. The first 32 characters are the GUID, the 33rd character is the SYSPRO instance number of 5, and the 34th states that this is a normal e.net Solutions user, not a user of SYSPRO Espresso.

```
2A2E961E250D0147899BFE396ACB263C50
```

When you run SYSPRO you do so from a shortcut (on the menu or desktop) that you double-click. This specifies the name of the SYSPRO executable to run, and SYSPRO's `Base` folder. SYSPRO uses this to find the configuration files (or uses the entries against the shortcut) and all of its components/folders.

With e.net Solutions there is no shortcut/trigger program to run (you just logon and start processing) so your computer needs to know where to find e.net Solutions. During installation an entry is created in your computers Windows Registry that specifies the location of your SYSPRO `Base` folder. The entry that is created is called *BaseDir*, and it exists within a key called *e.net solutions*.

It is possible to install multiple copies of SYSPRO on the SYSPRO application server. This tends to be done in a test or support environment, where each install is a different version (see **Figure 2-8**). To cater for each of these copies having their own instance of e.net Solutions, the *e.net solutions* Registry key allows for *multiple BaseDir* entries, named *BaseDir*, *BaseDir1*, *BaseDir2*, *BaseDir3* up to *BaseDir9*. The instance number for *BaseDir* is 0, *BaseDir1* is 1, *BaseDir2* is 2 etc.

When you log into e.net Solutions and receive the *UserID*, entries are written to a file called `ADMSTATE.DAT` (with its matching index file). This resides in SYSPRO's `Work` folder on the application server. Prior to SYSPRO 7 the state file was called `COMSTATE.DAT` (with its matching index file) and this resided in SYSPRO's `Base\Comstate` folder on the application server.

This file keeps the "state" information for each user such as the company settings at the time they logged on, their operator permissions, restrictions etc. The key to this file is the *UserID*. When you logoff of e.net Solutions the entries related to this *UserID* are removed from the `ADMSTATE` file.

When you call a business object from outside of SYSPRO you must supply a valid *UserID*. This is how SYSPRO keeps track of what each user can do and see. If you do not supply a valid *UserID* when you call a business object it will not run.

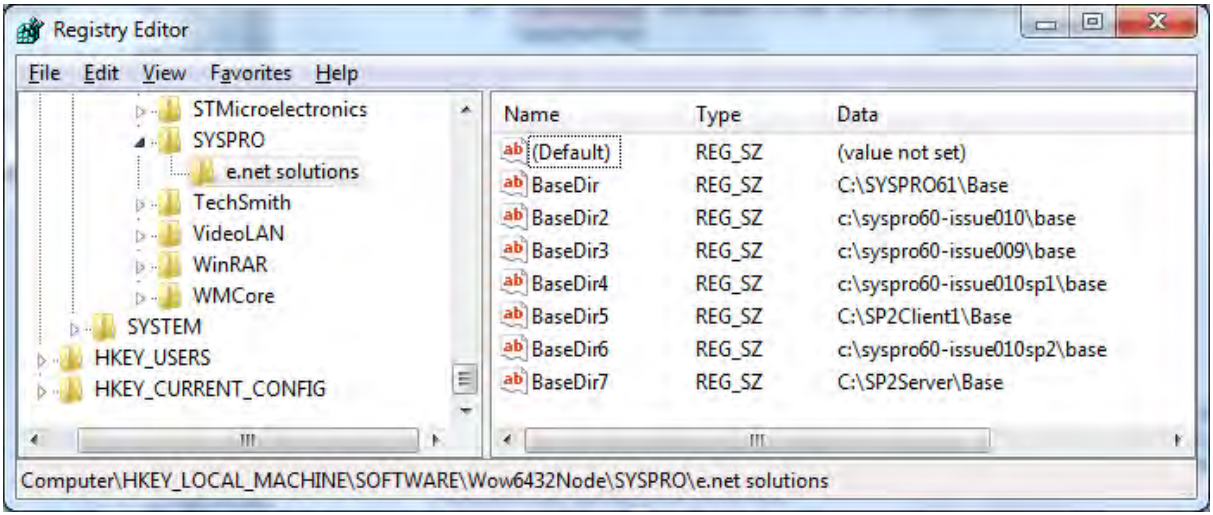


Figure 2-8: The *e.net solutions* Registry Key

When you perform the logon there are eight pieces of information that you can provide. Some are mandatory (such as the SYSPRO operator code) but most are optional, or can be made to pick up a default setting.

Operator – Supply a valid SYSPRO operator code. Depending on a setup option, this may also be a network user login (*Ribbon bar* | *Setup* tab | *General Setup* dropdown | *System Setup* | *General* tab).

Operator Password – Supply the password that matches the operator code.

Company ID – The company to which you want to logon. If you have a default company configured against this operator code, and you do not supply a company ID, the default will be used (*Ribbon bar* | *Setup* tab | *Operators* | select operator | *Edit* button | *Defaults* tab).

Company Password – The password for this company ID.

Language Code – This is the language code to be used for error and information messages. If you do not supply a language code it will use the language code that appears against the operator within SYSPRO (*Ribbon bar* | *Setup* tab | *Operators* | select operator | *Edit* button | *E.net* tab). If the language files do not exist for this language code it will default to English. The valid codes are *EN* for English, *FR* for French, *ES* for Spanish, and *ZH* for Simplified Chinese. Prior to SYSPRO 6.1 the language codes were numeric (*05*, *07*, *12*, and *15* respectively). If you logon using the numeric language codes it will continue to work as this functionality has been retained for backwards-compatibility. If you are logging on to the e.net Solutions Web-Based Applications, it will use this language code for the menus and screens.

Log Level – This can be either 0 or 1. If the value of 1 is supplied, e.net Solutions will report all XML elements that are invalid, whereas with 0 these elements are just ignored.

SYSPRO Instance – This tells e.net Solutions which *BaseDir* entry to use in the Windows Registry, and consequently which copy of SYSPRO and data to use. A value of 0 will use *BaseDir*.

XMLIn – Some additional parameters can be supplied within an XML string at logon time. These affect how e.net Solutions interacts with this user for the whole time that they remain logged on. If used, the XMLIn string has a root element of *<Logon>*. The other elements are :

- *<Siteid>* - Occasionally SYSPRO will provide functionality that is only activated when a specific site code is present (such as an Early Adopter who is testing this functionality before it goes on general release). The <Siteid> element is the means of supplying the site ID.

- *<FailWhenAlreadyLoggedIn>* - The default behavior when logging on, and there is already an operator logged on with the same operator code, is to log out the original operator and logon the new one. This removes the original user's *UserID* from the `ADMSTATE` file. When the <FailWhenAlreadyLoggedIn> element is supplied with the value of *Y*, if there is already an entry in the `ADMSTATE` file for this operator code, this logon attempt will fail and the *UserID* that was already in the `ADMSTATE` file will remain. The message that is returned is "The system has detected that a prior instance of the operator code is still logged in".

- *<BuildToPost>* - The <BuildToPost> element is used to inform a *Build* method business object that it should return the XML output with its element names more easily used by the matching *Post* business object.

- *<SourceApp>* - The <SourceApp> element takes a 30 character string that describes the application in use, and is intended for future use. For example, SYSPRO's electronic signature system will provide a mechanism for using a *$SourceApp* system variable in conditional statements, detail logging, and email notifications.

- *<SystemInformationReqd>* - This element causes addition information to be returned by a business object within a *<SystemInformation>* node such as the company name, operator group, currency etc. In addition, those items that would normally be returned as attributes of the root element (such as language, decimals, role, business object version, etc.) are returned as elements within the *<SystemInformation>* node.

- *<DefaultRole>* - A SYSPRO operator can belong to more than one role. When they login to SYSPRO they will assume their *Primary Role*, and they can change roles using the *Switch Roles* option on the *Home* tab of the ribbon bar. The displayed roles names are descriptions, and the roles are stored as three digit numbers starting at 001.

When an operator who belongs to multiple roles logs on to e.net Solutions, they will assume their primary role. However they can override this by logging on with the *<DefaultRole>* element containing the role number. If the supplied number is not valid for this operator, their default role will be used.

The role number can be found using the *Roles* program (*Ribbon bar* | *Setup* tab | *Roles*). By default the *Role* column does not appear in this listview, but it can be selected from the *Field Chooser* (the Field Chooser can be found on the context-sensitive menu that appears when you right-click on the column header). If required, you can sort on either the *Role* or *Description* by clicking on the relevant section of the column header.

- *<NameSpaceReqd>* - When a *Query* or *Build* business object is populating its return XML with data that comes directly from a table (rather than being a calculated amount), unless there is a reason to do otherwise it will use an element name that matches the table's column name. There are many business objects that return data from multiple tables, and some of the column names in the different tables are the same, such as *Description* and *Name*. It is incorrect to have multiple elements with the same name when they contain data from different tables, as you will not know which one contains information from which table.

  To cater for this the business objects use *Namespaces* for the elements that have the same name from different tables. The namespaces are declared as attributes against the root element, and are a six character *File Code* for this table name (such as *SALSLS* for the *Sales Analysis Salesperson* table, and *TBLCUR* for the *Currency* table). The following is an example containing the namespaces that appear against the root element of a business object.

  ```
  xmlns:SALSLS="SALSLS" xmlns:TBLART="TBLART" xmlns:TBLCUR="TBLCUR"
  ```

  When a duplicate element name needs to be written out, the element name is prefixed with the namespace. Columns that are duplicated but come from the primary table used by this business object will appear without a namespace. Duplicated columns from all other tables will have namespaces. For example, as the ***ARSQRY*** business object needs to return *Name* elements with values from both the *Customer Master* table and the *Salesperson* table it writes out the one from the *Customer Master* table as *<Name>* and the one from the *Salesperson* table as *<SALSLS:Name>*.

  Some third party applications and development environments do not cater for namespaces. To solve this problem the *<NameSpaceReqd>* element can be set to *N*. This removes the namespace attributes from the root element and replaces the colon between the "namespace" and the element name with an underscore, effectively making this just an element name. So the *<SALSLS:Name>* element would become *<SALSLS_Name>*.

Some examples of *XMLIn* XML strings are :

```
<Logon><FailWhenAlreadyLoggedIn>Y</FailWhenAlreadyLoggedIn></Logon>

<Logon><NameSpaceReqd>N</NameSpaceReqd><Siteid>A0555</Siteid></Logon>

<logon><buildtopost>Y</buildtopost></logon>
```

Naming Conventions

Like SYSPRO programs, business objects have a six character program name. The first three characters consist of a module code (for example, *INV* is the module code for the *Inventory Control* module). If the business object is not related to a specific module, the first three characters will be *COM*. The fourth character specifies to which class/method the business object belongs. The remaining two characters can contain anything.

Below is a list that explains to which class/method a business object belongs, depending on the fourth character of its name:

- Q – A *Query* class business object
- S – A *Setup* class business object
- T – A *Post* method business object (*Transaction* class)
- R – A *Build* method business object (*Transaction* class)

Some example of business objects are :
INVQRY – a *Query* class, *Query* method business object to retrieve information about a stock code.

SORTOI – a *Transaction* class, *Post* method business object to add, change or delete sales orders.

INVSST – a *Setup* class business object that can be used with the *Add*, *Update* and *Delete* methods to add/update/delete a stock code.

SORRSH – a *Transaction* class, *Build* method business object that is used to retrieve customer information to be supplied to the ***SORTOI*** business object.

Error Messages and Warnings

A business object can return errors in two different ways. The first is called an *exception*, which is where the error is such that the business object cannot continue. There are six types of errors that will cause an exception message:

- When attempting to logon and there is an invalid value. For example, the operator is invalid.
- When the supplied *UserID* is not valid/no longer valid.

- When calling a *query* or *build* business object and the supplied key value is not valid. For example, when using either the ***ARSQRY*** or ***SORRSH*** business objects and the supplied customer account number is not valid.
- When a file error is reported, such as a corrupt file index.
- When there is a parsing error when SYSPRO is processing the supplied XML. For example, when the supplied closing element name does not match the opening element name.
- When the debug option is set during the logon and one of the XML element names is not valid for this business object.

For example, if the customer maintenance business object (*ARSSCS*) is called using the Update method, and the supplied customer does not exist, it throws an exception. It does this because there is no point in the business object continuing through and validating all the other entries, as the primary key is invalid.

The second way of a business object returning an error is where the message appears in the XML returned from a business object. The business object returns the messages this way so that it can return more than one message, if more than one issue is detected. The alternative would have been to return one exception and then once the developer had fixed this and resubmitted the XML, it would return the next message. That is not a very efficient way to work.

The messages returned in the XML can be broken down into two types, *warnings* and *error messages*. A warning is a message that lets you know that something isn't right, but it isn't so serious that you can't continue, and depending on the parameter XML the business object will let you override it. An error message is so serious that it will not let you continue.

The warning and error messages are returned in the following structure, in this case for a *Work in Progress* job.

```
<job>
  <Value>00000153</Value>
  <ErrorNumber>300004</ErrorNumber>
  <ErrorDescription>Job number '00000153' not found</ErrorDescription>
</job>
```

As the problem relates to the value against the *Job* element, the error information is returned within a *<Job>* node. Within this are the elements *<Value>*, *<ErrorNumber>* and *<ErrorDescription>*. The *<Value>* element contains the value that was against *<Job>* element in the supplied XML. The *<ErrorNumber>* element contains the error number that is returned by the business object, and *<ErrorDescription>* contains the description for this error number in the language selected when you logged on to e.net solutions.

Most *Post* method, and *Setup* class business objects have the *<IgnoreWarnings>* element within the parameter XML string that allows you to manipulate the behavior of these messages. If the

<IgnoreWarnings> element contains the value *N*, all warnings will appear as error messages (see below).

```
<qtyissued>
  <Value/>
  <ErrorNumber>300092</ErrorNumber>
  <ErrorDescription>The quantity to issue is greater than the outstanding quantity
against the allocation</ErrorDescription>
</qtyissued>
```

If the *<IgnoreWarnings>* element contains the value *Y*, messages relating to warnings will not be included in the returned XML. If this means that no messages are returned in the XML the business object will continue as if no messages were generated. If the message above was the only one that was previously returned by the business object the business object could continue.

If the *<IgnoreWarnings>* element contains the value *W*, all warning messages will have a *<WarningNumber>* element instead of an *<ErrorNumber>* element, and a *<WarningDescription>* element instead of a *<ErrorDescription>* element. The values against these elements will still be the same.

```
<qtyissued>
  <Value/>
  <WarningNumber>300092</WarningNumber>
  <WarningDescription>The quantity to issue is greater than the outstanding
quantity against the allocation</WarningDescription>
</qtyissued>
```

The value of *W* allows a developer to search through the returned XML to see if there are any errors that will prevent the posting from happening (those with an *<ErrorNumber>* element). They can also search through the returned XML for any warnings (those with a *<WarningNumber>* element), and depending on which warnings are present, decide whether to resubmit the XML with the *<IgnoreWarnings>* element containing a *Y*.

The English error messages are kept in four text files in the `Programs` folder of the SYSPRO application server. These are called `MSGCOMEN.IMP`, `MSGDISEN.IMP`, `MSGFINEN.IMP` and `MSGMANEN.IMP`. The structure of an error message filename is that it begins with MSG, the next three letters relate to the SYSPRO development area (COMmon, FINancials, DIStribution, and MANufacturing), followed by the two letter language code used during the e.net Solutions logon (where EN is for English).

The error number consists of two parts, the module number followed by the four digit error code. In one of the examples above, the error number is 300004. At the top of each error file is a list of the modules that this file covers. Below is the header section for the `MSGMANEN.IMP` file where you can see that module 30 is *Work in Progress*.

```
; SYSPRO message file - Manufacturing Messages
;
; Copyright (c) 1994-2010 SYSPRO Ltd. All rights reserved.
;
; @(#) Version : 6.1.028     Last updated : 2011/11/21 07:50
;
; Includes the following modules
; 28 - Bill of Material Messages
; 29 - Quotation Messages
; 30 - Work in Progress Messages
; 31 - Requirements Planning Messages
; 37 - Engineering Change Control Messages
; 70 - Projects and Contracts
; 71 - Product Configurator
; 72 - Inventory forecasting
; 73 - Families and Groupings
; 74 - Inventory Optimization
; 7? - Reserved for Future Manufacturing Messages
```

Further down the file is a section for each module that contains the error messages that match these numbers. The section for the *Work in Progress* module is the third one from the top. Below are the first few entries from this section.

```
; 30 - Work in Progress Messages
300001 Work in Progress module not installed
300002 Work in Progress control record not found
300003 Job number '%1' not numeric
300004 Job number '%1' not found
300005 Job number '%1' on hold
```

The text against 300004 contains `300004 Job number '%1' not found`. The *%1* variable in the text is replaced at run time with a value supplied by the business object, in this case it would be the job number. Some error messages have two or even three variables that are replaced at run time.

If you write your own product, and it just needs to return the error message, you can display the contents of the *<ErrorDescription>* element. There may be occasions where your product needs to react in a certain way depending on the error that is returned. If your product will only ever be used in English, it is possible that you could pattern-match on the contents of the returned *<ErrorDescription>*. However, if your product can be used in multiple languages, or the error message contains one of the variables that get populated at run time, you should rather use the contents of the *<ErrorNumber>* element as this will be consistent across all languages and does not contain the expanded contents of the variable.

Licensing

Unlike SYSPRO which uses concurrent licensing, e.net Solutions licenses its use by SYSPRO operator code. The basics of e.net Solutions licensing, and how it differs from SYSPRO licensing, was covered in the *Licensing Overview* section of chapter 1.

You only need to license e.net Solutions if you are going to use it outside of SYSPRO. If you are going to call a business object from within SYSPRO (such as from a customized pane, or SYSPRO form) you should not need to license e.net Solutions, as this is covered by your SYSPRO license. The e.net Solutions *System Manager* must be licensed for you to call a business object from outside of SYSPRO. Without this you cannot call any business object.

Functional Areas

The e.net Solutions business objects are grouped together in *Functional Areas*. The e.net Solutions System Manager has three functional areas that are licensed when you purchase the System Manager. These are the *e.net System Query Functional Area*, the *e.net System Posting Functional Area*, and the *e.net System Setup Functional Area*. The first contains all the queries that are not module-specific such as the generic browse, generic fetch, message inbox query, and the custom form query. The second contains posting business objects that are not module-specific such as writing to the message inbox, changing the operator password, and writing to the NT Event Log (the content of which can be seen in the Windows Event Viewer, under Control Panel, Administrative Tools). The third contains all the *Setup* class business objects that create and modify semi-static data for all modules (such as adding/updating/deleting a customer, adding/updating/deleting a stock code).

Other than those above that are linked to the System Manager, functional areas are subsets of SYSPRO modules. Some modules have two functional areas, and others have many more, such as the inventory control module that has seven functional areas. All the modules and their associated functional areas can be seen in the Business Object Reference Library (BORL) which can be found on the SYSPRO Support Zone. This lists the modules and their functional areas, with the business objects listed within them.

To be able to license a functional area you must have the e.net Solutions System Manager licensed, and you must also have the corresponding SYSPRO module licensed. For example, if you want to license the *Inventory Primary Query Functional Area* you must have the e.net Solutions System Manager and the *Inventory Control* module licensed.

The functional areas are licensed in user bands of tens. So if you need to have eight people accessing the *Inventory Primary Posting Functional Area* you would need to buy a 10 user license and if you needed to have 12 people you would need to buy a 20 user license. As you are only licensing the functional area it will be significantly cheaper than licensing a module.

License.XML

The `License.XML` file contains your SYSPRO license details, including which e.net Solutions functional areas you have licensed, and how many licenses you have. If you change your number of

SYSPRO users, modules, functional areas, or renew your annual license, you will be sent a new `License.XML` file. Each time one of these files is imported into SYSPRO to update your SYSPRO license, it should also be imported into the e.net Solutions section.

The e.net Solutions section of the `License.XML` file is imported through SYSPRO's *System Setup* program (*Ribbon bar* | *Setup* tab | *General Setup* | *System Setup* | *Setup* tab). At the bottom of the *Setup* tab is the *Configure e.net License* button (see **Figure 2-9**).

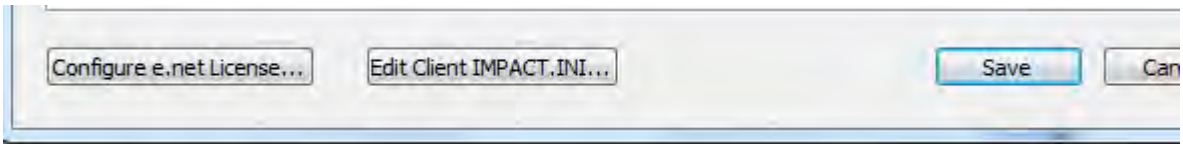


Figure 2-9: The *Configure e.net License* button on the *System Setup* screen

This button starts the wizard that enables you to import the functional area licenses from `License.XML`, and apportion them to specific operator codes. The first option in the wizard is to either import the licenses from the `License.XML` file, or to apportion the licenses that have already been imported (see **Figure 2-10**). If the *Import LICENSE.XML file* option is selected and you click on the *Next* button you will be prompted to locate the import file. This will default to `License.XML` in you SYSPRO `Work` folder. Once you have selected the file, click on the *Next* button to import its contents. Click on the *Finish* button to complete the import. You will be taken to a screen where you can either apportion the licenses (to the business object or Web-Based Applications) or maintain the e.net classes (see **Figure 2-11**). Alternatively, you can click on the *Cancel* button to exit this program, but the imported license file will remain imported.

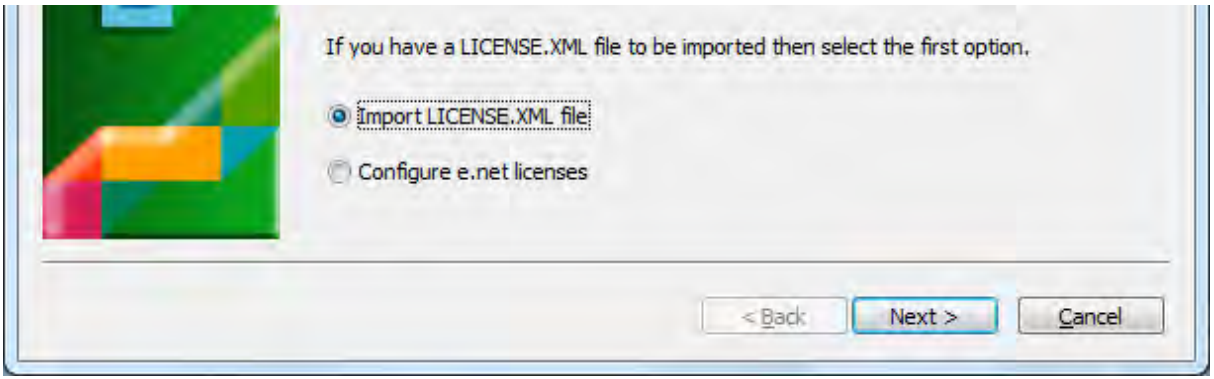


Figure 2-10: The start of the *Configure e.net License* wizard

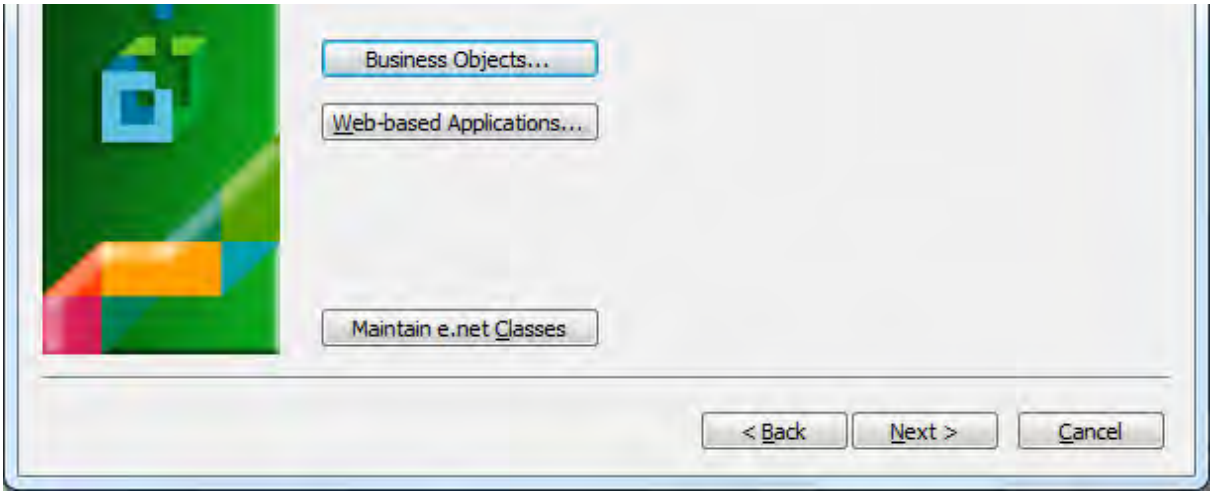


Figure 2-11: Apportioning licenses or maintaining web-based application classes

There are two types of e.net Solutions functional area licenses. The first is for use with the business objects and the second is for use with the Web-Based Applications. Unless you have specifically licensed the web-based application functional areas, your license will contain the business object functional area licenses. Click on the *Business Objects* button and the *Apportion e.net Licenses* screen will be displayed, which is divided into three sections (see **Figure 2-12**).

On the left of the screen is a list of all of the functional areas, and immediately after each one is the number of licenses that have been purchased for this functional area. In **Figure 2-12** the number against each of the functional areas is 5 as this is from a test system. If it were a live system the numbers could be either zero, or a multiple of 10.

On the right of the screen is a list of all the operator codes on the system which can have licenses apportioned to them. Special operator codes such as those beginning with two underscore characters will not appear here, as they cannot be used to logon to e.net Solutions. The operator list reflects the license status for the functional area that is highlighted in the left pane. In **Figure 2-12** the pane on the right reflects the apportioned licenses for the highlighted *AP Primary Query* functional area. The columns in this pane are *Licensed*, *Named user*, *Name*, *Type*, and *Concurrent users*. The *Licensed* column reflects whether this operator code is allowed to use this functional area. The *Concurrent users* column specifies how many licenses will be consumed if this operator is checked. All except one contain the number 1, whereas operator *NRUser* has the number 3. This means that this operator code is configured as a *Guest/anonymous* operator and can have three people logged on to e.net Solutions using this operator code. The rest are *Named users*. See the sections below on *Named users* and *Guest/anonymous* users for more details.

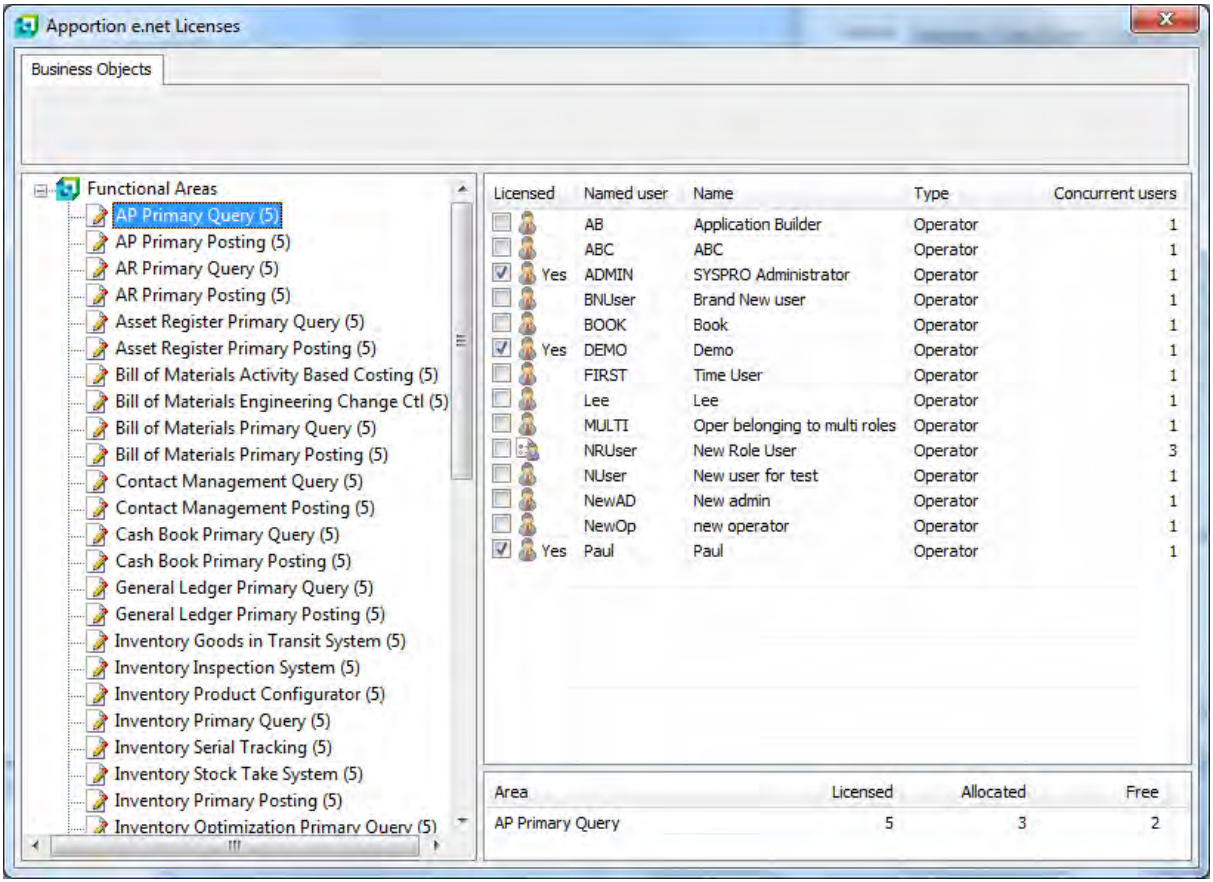


Figure 2-12: The *Apportion e.net Licenses* screen

The section at the bottom right of the screen shows how many licenses are available for this functional area, how many have already been allocated (by checking the operators), and how many are still free. If you attempt to apportion more licenses than you have available, the *Free* quantity will be displayed as a negative value (see **Figure 2-13**), a message to this affect is displayed in red at the top of the screen (see **Figure 2-14**), and the functional area name will have an asterisk after the number of licenses (see highlighted section of **Figure 2-14**). If you attempt to continue with the wizard and save the ove-apportionment, you will receive a message to this affect. If you then attempt to continue you will be prevented from saving these settings.

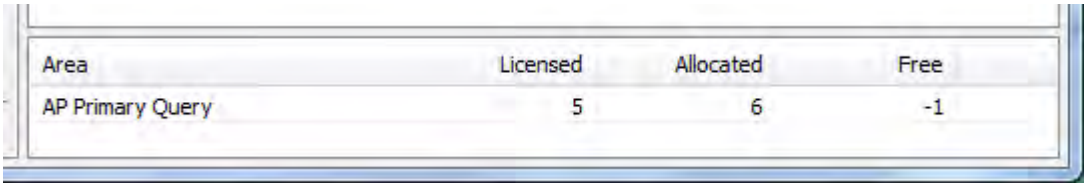


Figure 2-13: Over-apportioning the license displays a negative *Free* quantity

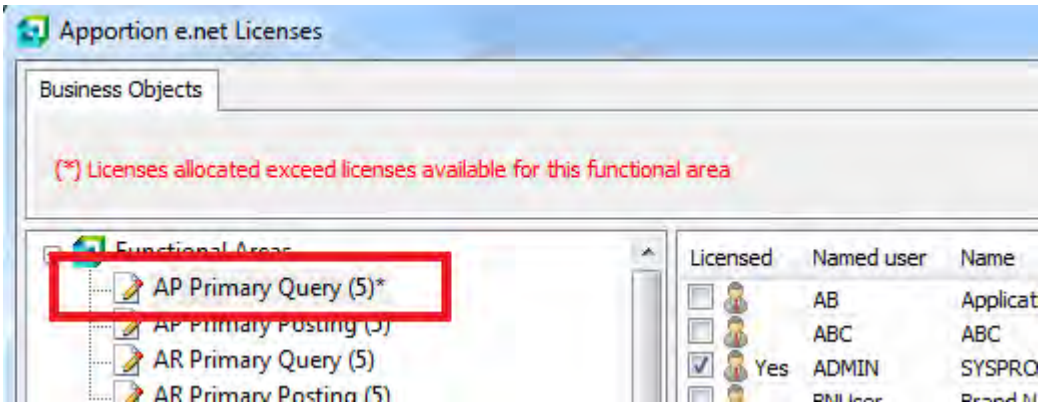


Figure 2-14: Over-apportioning license displays warning message

Once you have apportioned the required licenses to the operators (you do not need to apportion all of the licenses) click on the white X on the red background at the top right of the screen. You will be taken back to the *Configure e.net licenses* screen that appears in **Figure 2-11**, where you should click the *Next* button. Finally, click on the *Finish* button on the screen that is displayed.

If you have e.net Solutions Web-Based Applications licenses to apportion, this is done using the *Web-based Applications* button that appears in **Figure 2-11**. The screen that is displayed when this button is selected is almost identical to that for the *Business Objects* button, except for the *Web-Based Applications* tab at the top of the screen in place of the *Business Objects* one. On the right of the screen is the list of valid operator codes as with the business objects. If any have been configured, it will also contain the Web-Based Applications classes. A Web-Based Applications class is a means of having concurrent user licensing for the Web-Based Applications. You can define a class name, the number of members of a class, and the class timeout period (details about this appear in the *Class User* section below). When a class appears in the operator list it will also include the number of concurrent users that it allows, and consequently will consume if selected. **Figure 2-15** shows the class *SALES* that has three concurrent users.

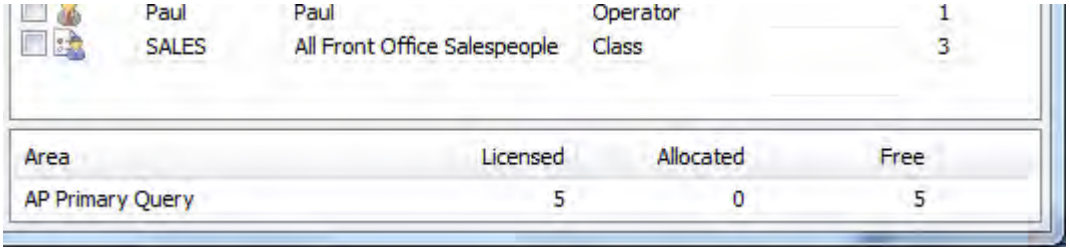


Figure 2-15: Apportioning licenses to a *Web-Based Applications Class*

Operator Licensing

Named User

When you add a new operator to your system the *Usage for e.net business objects* option defaults to *Named user*. This can be seen on the *E.net* tab of the *Operator Maintenance* program (see **Figure 2-16**). The *Named User* licensing differs from the SYSPRO concurrent licensing in that the license for this functional area is apportioned to the operator code, and once apportioned, only this operator can use it. *Named User* licensing is available for both business object functional areas, and web-based application functional areas.

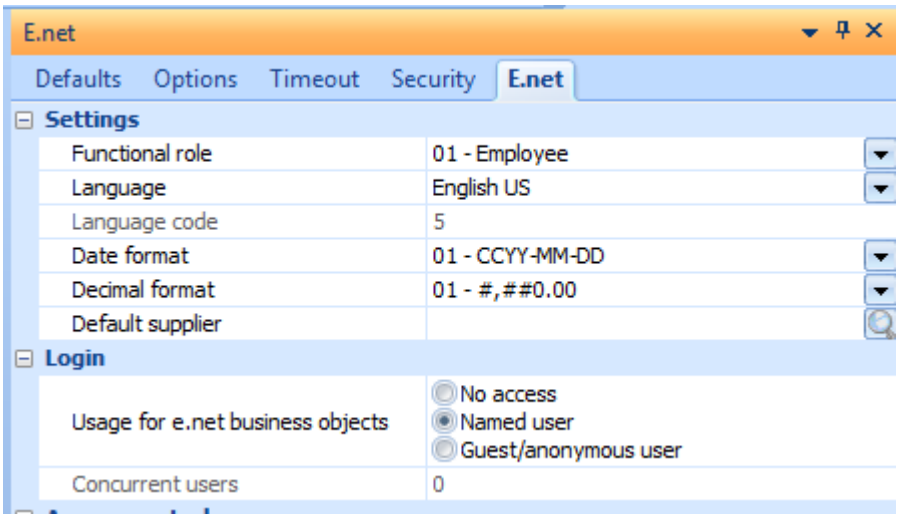


Figure 2-16: Setting the e.net Solutions licensing type to *Named user* access

The *Named User* licensing model is designed around each operator that is going to use e.net Solutions having their own operator code, and only one person to login to e.net Solutions using this operator code. If another person logs into e.net Solutions using this operator code, the original person will have their *UserID* revoked, and no longer be able to call business objects. When you have finished

using the business objects/e.net Solutions you should *logout* of e.net Solutions, which removes the *UserID* from the `ADMSTATE` file. To prevent this default behavior you can add the following XML to the login string (see the section *The e.net Solutions UserID and the Logon Process* above for more information).

```
<Logon><FailWhenAlreadyLoggedIn>Y</FailWhenAlreadyLoggedIn></Logon>
```

When an operator calls a business object from outside of SYSPRO, a check is made that they have a valid *UserID*, to which functional area the business object belongs, and if the operator group to which they belong is allowed to access the business object (*Ribbon bar* | *Setup* tab | *Groups* | select operator group | *Change* | locate the relevant e.net Solutions class within the *Security Access* section). If all of these allow access then the business object is called. **Figure 2-17** shows some of the business objects making up the *e.net Solutions – Query* class within the *Security Access* section of the *Operator Group* program.

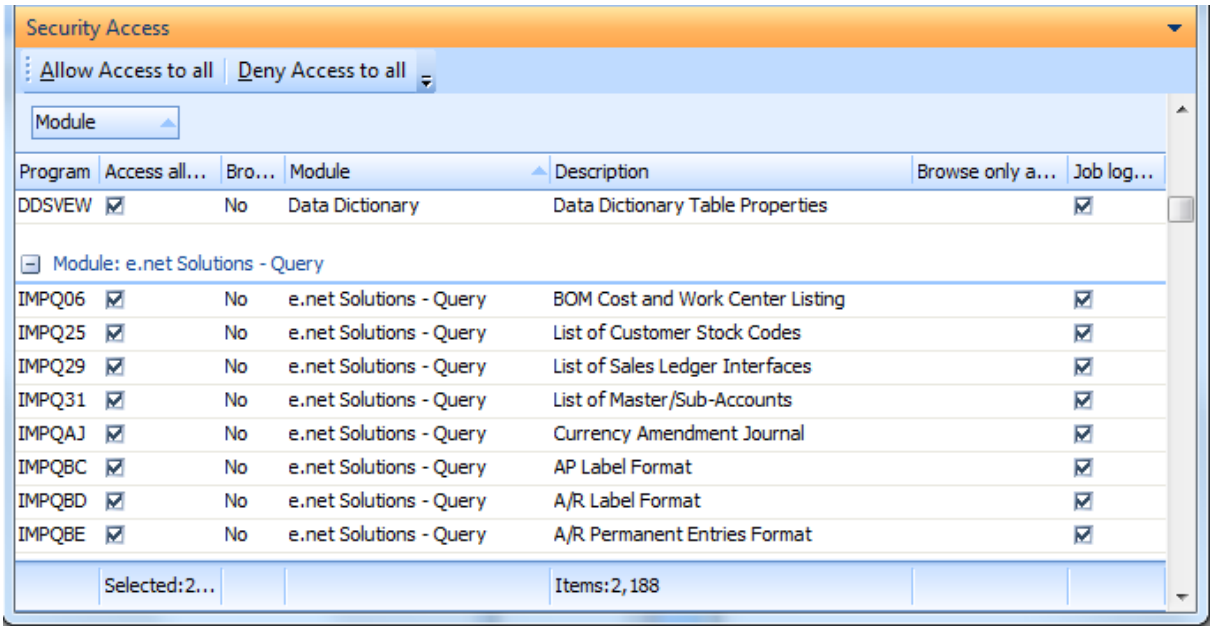


Figure 2-17: Setting group level access to e.net Solutions business objects

If an attempt is made to call a business object with an invalid *UserID* there is a five minute timeout before e.net Solutions responds.

No Access

If the *Usage for e.net business objects* option is set to *No access* on the operator's *E.net* tab, this prevents the operator from using any e.net Solutions business object outside of SYSPRO. If they attempt to login to e.net Solutions they will receive the message *Access denied to business objects for operator*, followed by their operator code.

Guest/anonymous User

A *guest/anonymous user* is a SYSPRO operator code that allows more than one e.net Solutions user to logon at the same time. Against the operator code you configure the number of logons that must be allowed within a 30 minute time period. This effectively becomes the number of concurrent users (i.e. the number of licenses consumed when the operator is licensed to use a functional area). *Guest/anonymous user* licensing is available for both business objects and the *Web-Based Applications*. **Figure 2-18** shows the operator being configured as a *Guest/anonymous user* with 3 *Concurrent users*.

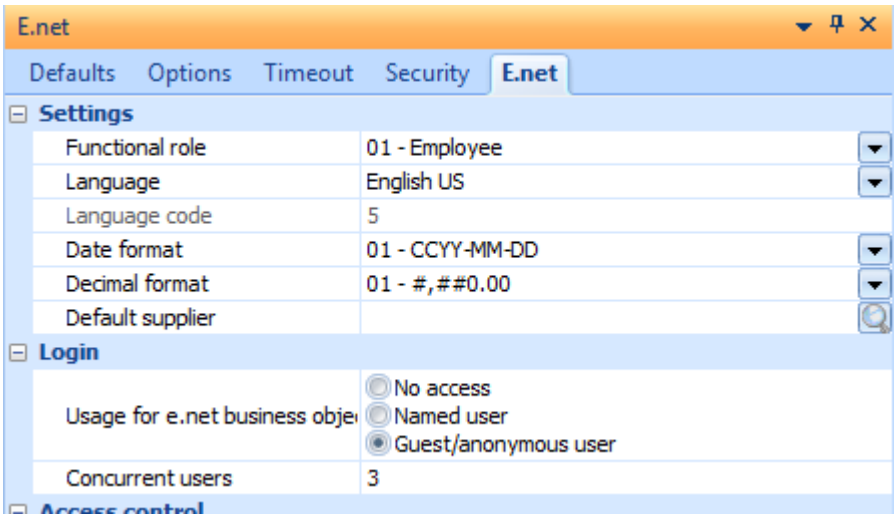


Figure 2-18: Configuring a *Guest/anonymous user*

The idea of the guest/anonymous user is that you do not need one SYSPRO operator for each person that requires the same access to e.net Solutions. For example, if a SYSPRO company wants to provide an application to 300 of their customers that allows them to capture sales orders remotely and upload them using web services, they may not want to provide a license for each one, particularly since some may order daily, some only order monthly and some order infrequently. By establishing how often their customers order they can work out the maximum number that could logon during any 30 minute time period. They could then create an operator to process these orders and set it to be a *Guest/anonymous user*. Against this they can set the number of concurrent users during a 30 minute time period.

Unlike a *Named user* operator, a *Guest/anonymous* operator does not need to *logoff* from e.net Solutions as their license will become available automatically for another person to use after 30 minutes of inactivity. However, they can start using the *UserID* again if it has not been consumed by someone else in the interim, as they are not automatically logged off after the 30 minutes of inactivity.

The following are some items to consider when looking at *Guest/anonymous* licensing:

- If *a* guest/anonymous user has a concurrent user count of three, then each functional area to which the guest/anonymous user is given access will consume three licenses.

- If a guest/anonymous user logs on to e.net Solutions, logs off and logs back on again within 30 minutes, then two licenses would have been consumed. Because this method could use more licenses than that consumed in a named user environment, we recommend that you only use this licensing method when appropriate.

- Because each of the users that are logging on with a guest/anonymous operator code are using the same operator code, you cannot place restrictions (or set defaults) individually. All users of the operator code will have the same defaults and restrictions. You also cannot tell (using job logging, or journals) exactly which person performed a task, or made a change, as it will all be logged under the same guest/anonymous operator code.

- A guest/anonymous user who successfully logs on to e.net Solutions is allocated a 34-character *UserID* which is placed in the `ADMSTATE` file, together with their state information. Each time this user uses a business object, a timestamp is updated against this *UserID* in the `ADMSTATE` file. Unlike a named user, when a guest/anonymous user logs off from e.net Solutions their *UserID* and matching state information is not removed from the `ADMSTATE` file, but the timestamp is updated. When another attempt is made to logon using the same guest/anonymous operator code, a check is made to see how many users of this operator code are already logged on.

  The user is allowed to logon if the number is less than the number of concurrent users allowed for this operator code. If the number is the same as the number allowed, then a check is made to establish which existing user of the operator code has the highest period of inactivity. If this period of inactivity is 30 minutes or greater, they are logged out and the new user logged on. If the highest period of inactivity is less than 30 minutes the new user is informed that this guest account has exceeded the maximum number of concurrent users, and prevented from logging on.

- If a guest/anonymous user is already consuming licenses for functional areas and the number of concurrent users against the operator code is increased using the *Operator Maintenance* program, it is possible that this will cause the number of licenses consumed to exceed those available. If this occurs, a message is displayed, indicating that you have apportioned more licenses to business objects than you are licensed to use. This happens as soon as an

operator attempts to either call a business object, or logon to e.net Solutions. An example of this message can be seen in **Figure 2-19**.

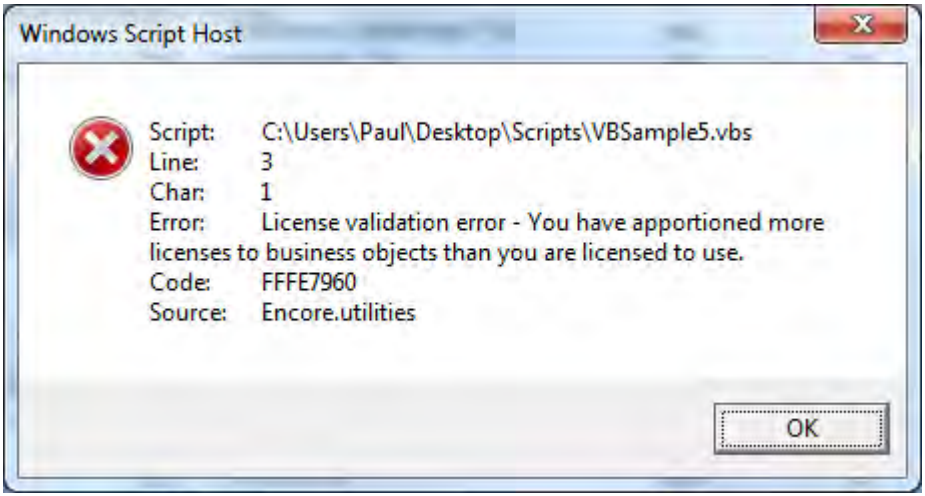


Figure 2-19: The error message that is displayed when you over-apportion *Guest/anonymous* licenses

Class Users

Class users are only available for use with the e.net Solutions *Web-Based Applications*. A class is a group of people performing the same or similar tasks that are licensed together in a true concurrency model. A SYSPRO operator can only be a member of one class, but a class can have almost any number of members.

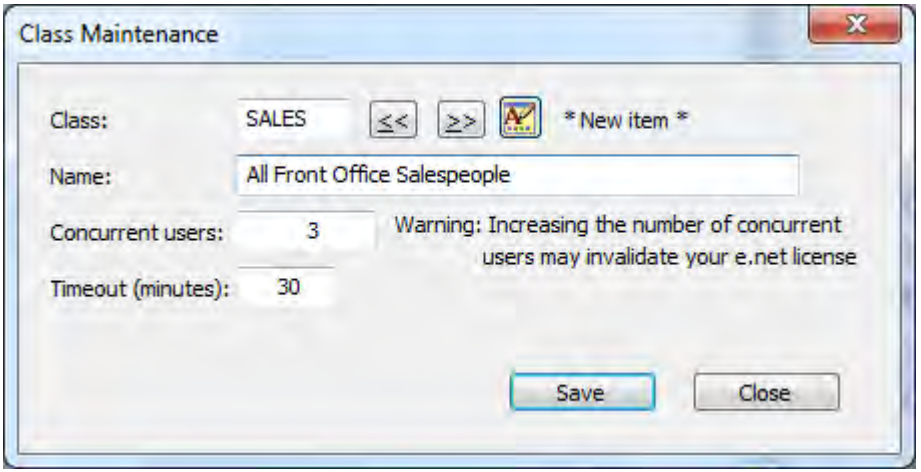


Figure 2-20: Adding an e.net Solutions *Web-Based Applications Class*

Unlike both *Named users* and *Guest/anonymous users*, a class is not a SYSPRO operator code. It is created using the same program that you used to import the `License.XML` file for e.net Solutions/apportion the licenses (*Ribbon bar* | *Setup* tab | *General Setup* dropdown | *System Setup* | *Configure e.net License* button | *Configure e.net Licenses* radio button | *Maintain e.net Classes* button).

Figure 2-20 shows a Web-Based Applications class being added that has three concurrent users. The timeout value specifies after how many minutes of inactivity must the operator be logged out of e.net Solutions.

When logging on to the e.net Solutions web-based application, operators use their own SYSPRO operator code. Because they are a member of a class, the Web-Based Applications will use the class license. It is not possible to logon using the class code.

When apportioning access to functional areas for a class, the class consumes the same number of licenses as the number of concurrent users defined against the class. It consumes the same number of licenses for all functional areas that the class is licensed to use.

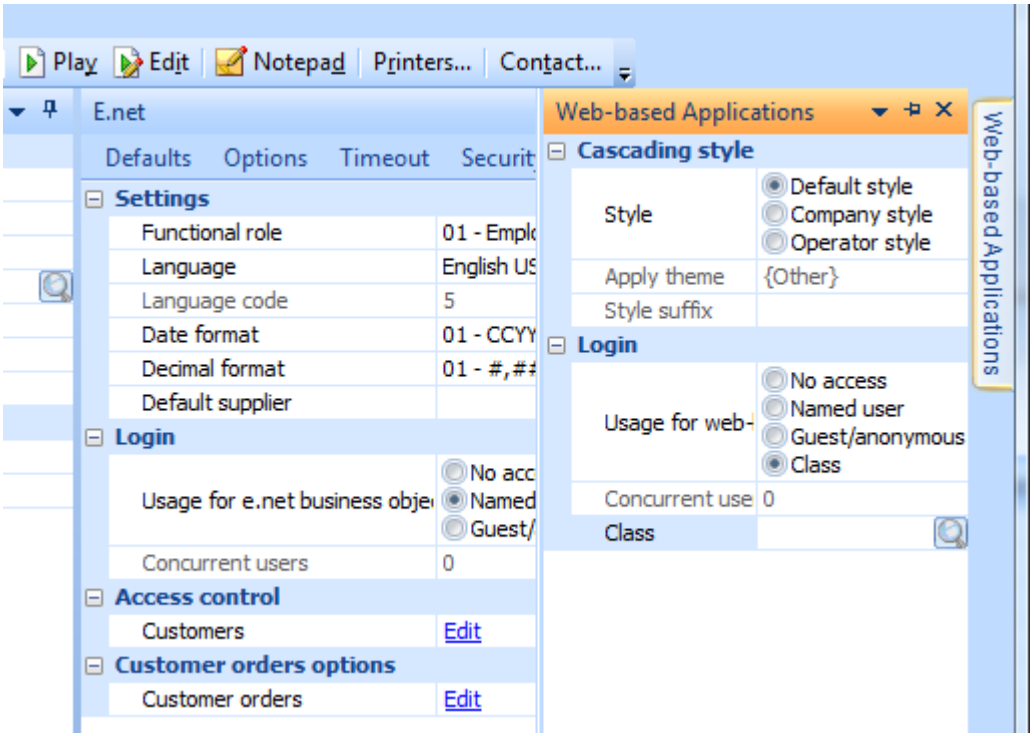


Figure 2-21: Setting the *Class* licensing for the *Web-Based Applications*

When a member of a class attempts a logon to the *Web-Based Applications*, the system checks the state file to establish the number of existing members of the class currently logged on. If it is less than the number of *Concurrent users* defined against the class, then the operator is assigned a *UserID* (which is added to the state file) and they are allowed to logon. Whenever this operator accesses a business object, a timestamp is updated against their entry in the `ADMSTATE` file. When operators that are members of a class logout, their *UserID* and state information is removed from the state file.

License Method	Benefit	Drawback
Named User	• You can track what each user has done • You can restrict what each user can do	• Uses one license for each user, regardless of whether they are logged on
Guest/anonymous User	• You can use less than one license per operator code	• You cannot track what individual users did as they are all logged in using the same operator code • You cannot restrict what individual users can do as they are all logged on as the same operator code • There is a 30 minute timeout to make the license available • Used incorrectly it can consume more licenses than Named User
Class User	• You can use less than one license per operator code • The administrator can set the number of concurrent users • The timeout value can be set by the administrator • You can track what individual users have done • You can restrict what individual users can do	• Only available for the Web-Based Applications • An operator can only belong to one class

Table 2-1: Summary of benefits and drawbacks for each license method

When a member of a class attempts a logon and the number of members for the class already in the state file equals the number of concurrent users allowed, the system establishes which member of the class has the greatest period of inactivity. If this period of inactivity is greater than the *Timeout* value set against this class, then the existing user is logged out and the new user is logged on. If the period

of inactivity is less than the timeout value, then the new operator is prevented from logging on and a message to this effect is displayed.

The *Web-Based Applications* have their own tab within the *Operator Maintenance* program for configuring the licensing as well as how the Web-Based Applications will appear to the operator. Unlike the other tabs, the *Web-Based Applications* tab is a minimized tab on the right of the screen. When you move your mouse pointer over this tab it will expand and you can either pin it in place, or make any changes and let it minimize again. **Figure 2-21** shows the *Web-based Application* tab after it has been expanded. The *Usage for Web-Based Applications* section has the *Class* radio button selected, but the *Class* hasn't yet been selected.

Table 2-1 shows the different licensing methods, along with the benefits and drawbacks of each.

XML

XML is an abbreviation for eXtensible Markup Language and is a markup language similar to HTML. However, unlike HTML it is designed to carry data, not display it, and you define your own element names, they are not predefined for you. Although designed for use by computers, an XML file is humanreadable and can be generated and modified with simple text editors such as Notepad

This section is not intended to go into great detail about XML; it is merely intended to give you some idea of what it is, and how you can use it.

Some Basic Syntax Rules

There are a few basic rules to explain before working through one of the SYSPRO XML files. When an XML file conforms to these syntax rules it is referred to as well-formed, and when it does not conform to the syntax rules it is referred to as being not well-formed. Just like HTML, XML has tags. An XML element is everything from the beginning of its opening tag to the end of its closing tag. This will include any other elements that fall within it, any values, and any attributes.

The following is an XML element that is being used to describe a stock code. The element name does not need to be *StockCode*, it is just good practice to use element names that are meaningful.

```
<StockCode>A100</StockCode>
```

You can see that the tag name is enclosed within less-than and greater-than signs. The element is closed using a tag of the same name, just preceded by a slash. Element names are case-sensitive, so if you were to do the following it would not be well-formed, and could not be used.

```
<StockCode>A100</Stockcode>
```

If an element is empty it can appear as in either of the following two examples. Alternatively, if it is not required it could be left out completely.

```
<StockCode></StockCode>

<StockCode/>
```

There are some restrictions about how you can name your elements, such as it cannot start with a number or punctuation mark, it cannot start with the letters *xml*, and it cannot contain spaces. So an XML file containing any of the following would not be considered well-formed.

<xmlStockCode>A100</xmlStockCode>

<2Items><Item1>AAAA</Item1><Item2>BBBB</Item2></2Items>

<%Variables><Variable1>A</Variable1><Variable2>B</Variable2></%Variables>

XML elements must be properly nested, meaning that an element that is opened within another element, must be closed before the first one can be closed. This one is well-formed:

```
<Item><StockCode>A100</StockCode></Item>
```

Whereas this one is not:

```
<Item><StockCode>A100</Item></StockCode>
```

Every XML document must have a root element, which is the parent to all other elements. In this case the *<SalesOrder>* element is the root element. Note: the XML has been displayed indented this way as it makes it easier to read, it would be just as acceptable for it to appear as one line.

```
<SalesOrder>
  <Header>
    <SalesOrderNumber>000100</SalesOrder>
    <Customer>0000001</Customer>
  </Header>
  <Details>
    <StockCode>A100</StockCode>
    <OrderQty>1.000</OrderQty>
  </Details>
</SalesOrder>
```

The XML syntax caters for values to be supplied using what are known as *attributes*. SYSPRO business objects do not use entities for consuming XML, but they do use them for the returned XML, so you do need to know about them. The attribute values must always appear in quotes. The following is an abbreviated representation of the opening root element of the output from the ***ARSQRY*** business object. It has been abbreviated otherwise it would wrap around the page multiple times.

```
<ARStatement Language='05' Language2='EN' CssStyle='' Version='6.1.017' >
```

This would be the equivalent of the business object returning the following:

```
<ArStatement>
  <Language>05</Language>
  <Language2>EN</Language2>
  <CssStyle></CssStyle>
  <Version>6.1.017</Version>
```

As you can see, attributes can save considerable space in the XML.

There are five special characters that you should not use as values within your XML data. These special characters can be replaced with a string of characters known as *Entity References*. These characters are the less-than and greater-than signs (because XML would not know where the elements start and finish), the apostrophe/single quote, the double-quote, and the ampersand (because the entity references start with an ampersand). All entity references start with an ampersand and finish with a semi-colon. **Table 2-2** lists the characters and their entity references.

Character	Entity Reference	Description
<	<	Less-than
>	>	Greater-than
‘	'	Apostrophe/single quote
“	"	Double quote
&	&	Ampersand

Table 2-2: A list of *Entity References*

Comment lines start with the `<!--` string, and continue until they are terminated with the `-->` string. Lines that are comment lines are ignored when the XML is processed.

Samples and Schemas
Within your SYSPRO `Base` folder is a folder called `Schemas`. This contains sample XML files and their matching schemas. The schemas have the same name as the XML file, but instead of having an XML suffix they have an XSD suffix. The schema is also a text file with tags, and describes what can and can’t appear within the XML file. The schemas do their best to describe the possible entries in the matching XML file, but because SYSPRO business objects are so flexible, there are some where the schema does not perfectly match everything that the business object allows. Therefore they should be treated as another form of documentation, and not a definitive way to check that the contents of the XML file is valid (if a schema is programmatically used to check the contents of an XML file and it passes, the XML file is referred to as valid. If it fails it is invalid).

The following is a simplified sample of the *ARSQRY.XML* file from SYSPRO’s `Base` folder. It is the sample for the customer query business object called ***ARSQRY***. Because of space limitations imposed by the width of this page, the line starting with *<Query xmlns* has wrapped around and appears over two lines.

```
<?xml version="1.0" encoding="Windows-1252"?>
<!-- Copyright 1994-2010 SYSPRO Ltd.-->
<!--
     This is an example XML instance to demonstrate
     use of the Customer Query Business Object
-->
<Query xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:noNamespaceSchemaLocation="ARSQRY.XSD">
  <Key>
    <Customer><![CDATA[000019]]></Customer>
  </Key>
  <Option>
    <MultiMediaImageType>GIF</MultiMediaImageType>
    <IncludeFutures>N</IncludeFutures>
    <IncludeTransactions>Y</IncludeTransactions>
    <IncludeCheckPayments>Y</IncludeCheckPayments>
    <IncludePostDated>Y</IncludePostDated>
    <IncludeZeroBalances>N</IncludeZeroBalances>
    <IncludeCustomForms>N</IncludeCustomForms>
  </Option>
</Query>
```

The first line in this file is the XML declaration and defines that this is using version 1.0 of XML, and the *Windows-1252* character set. Windows-1252 is a single byte character set that includes all the English characters, and most of those used in Western Europe. This is SYSPRO's default.

The line containing *Copyright* is a standalone comment line because it starts with the opening comment string and ends with the closing one. The four lines immediately after this one are also comment lines (because the first line starts with the opening string, and the fourth line finishes with the closing string). So the following XML file without the comment lines would be processed by the business object in the same way as the one above.

```
<?xml version="1.0" encoding="Windows-1252"?>
<Query xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:noNamespaceSchemaLocation="ARSQRY.XSD">
  <Key>
    <Customer><![CDATA[000019]]></Customer>
  </Key>
  <Option>
    <MultiMediaImageType>GIF</MultiMediaImageType>
    <IncludeFutures>N</IncludeFutures>
    <IncludeTransactions>Y</IncludeTransactions>
    <IncludeCheckPayments>Y</IncludeCheckPayments>
    <IncludePostDated>Y</IncludePostDated>
    <IncludeZeroBalances>N</IncludeZeroBalances>
    <IncludeCustomForms>N</IncludeCustomForms>
  </Option>
</Query>
```

The next line (that starts with *<Query*) is the root element. Every XML document must have one root element that is the parent of all other elements. This is used to define the start and end of the XML document, and wraps around the whole document, excluding the XML declaration (and any comment lines that might appear above it). In this case the root element also contains *attributes*, one specifies how *namespaces* are to be used and the other specifies that there is a matching *schema* file. See the *Namespaces* section below for more information.

Nested within the *<Query>* element is the *<Key>* element, and this has the *<Customer>* element nested within it. The *<Customer>* element from the XML appears below.

```
<Customer><![CDATA[000019]]></Customer>
```

This contains a *CDATA* section, but could have been represented with the following :

```
<Customer>000019</Customer>
```

The *CDATA* section starts with the `<![CDATA[` string and finishes with `]]>`. It tells the XML parser to ignore the contents of the *CDATA* section when it is parsing the document. There are several reasons for doing this, but the most common is where the data is being received from a third party and you know that the business object will accept it. So instead of working through the data, converting the illegal characters to *entity references*, and passing it to the business object, you use a *CDATA* section.

An example is where you are receiving data from someone, and this will be used to create a new customer in your system. You have an agreement that they will not pass any of the illegal XML characters in the customer account code, but there are several customers that will have ampersands in their customer names. By wrapping the customer name value within a *CDATA* section, the XML parser will not complain, and the ampersand is a valid character in a SYSPRO customer name. Therefore the value will be accepted and used as is.

The rest of the XML in the sample consists of elements and values, and finishes at the closing root element.

Namespaces

Namespaces were briefly covered in the logon parameters earlier in this chapter. They can be used where there are multiple XML elements with the same name, but different types of data, or from different sources. Many of the query business objects will retrieve data from multiple tables. As the element names typically use the column name of the data that is retrieved, there are cases where the element names would be duplicated. In these cases the business object will use namespaces to separate the data. This is probably better explained using an example below. The newer business objects have been written in a slightly different way so that they do not use namespaces, as some third party products cannot handle XML that contains namespaces. As the existing business objects that used namespaces could not be changed without the risk of breaking existing applications, the logon XML parameter *<NameSpaceReqd>* was added to remove the namespaces for people that did not want them, and use an alternative prefix.

A namespace example: The customer query business object ***ARSQRY*** retrieves data from many tables, depending on the input XML. One of these pieces of data is the customer's name, from the customer master table. The column that this is retrieved from is called *Name*, so that becomes its element name. The business object also needs to retrieve the salesperson's name from the salesperson master table. The salesperson name column in the salesperson master table is also called *Name*. What the business object does is check to see if any element names are duplicated. If they are, and any of these come from the primary table that the business object is accessing, these element names are used as is. In this example the customer master table is the primary table so its *Name* column header is used as the *<Name>* element. All other column names of *Name* will have a namespace.

When used, a namespace is added to the root element of the returned XML, and it is added as an XML *attribute* (see below, where the example has been abbreviated to stop it wrapping around the page).

```
<ARStatement Language='05' Language2='EN' xmlns:SALSLS="SALSLS">
```

The namespace is the section starting "`xmlns:`". Immediately after this is "`SALSLS`" (which is the *File Code* for the salesperson master table), and the prefix to use for the namespace (which is also `SALSLS`). Whenever there is a duplicate element name that originated from the salesperson master table, the business object will add the *SALSLS:* prefix to the element name. Here is an extract from the XML that this business object produced. Note that the opening tag for the salesperson's name element is now *<SALSLS:Name>* and the closing tag is *</SALSLS:Name>*.

```
  …
<SoldToAddr5>AUS</SoldToAddr5>
<SoldPostalCode>1001</SoldPostalCode>
<SALSLS:Name>Barry Jones</SALSLS:Name>
<StatementDate>2012-04-24</StatementDate>
<TermsCode>2</TermsCode>
<TBLART:Description>60 Days - Net</TBLART:Description>
<Currency>$</Currency>
<TBLCUR:Description>Local Currency</TBLCUR:Description>
<Contact>Sheryl Endstep</Contact>
  …
```

In this example there are also two elements called *Description*. One comes from the *Currency* table (`TBLCUR`) and the other comes from the *Accounts Receivable Terms* table (`TBLART`).
Later business objects do not use namespaces. Instead, their element names that would have had a namespace are prefixed with a meaningful description that puts the element name into context. Below is an extract from the *ASSQFE* business object where there are multiple *Description* elements that would previously have had namespaces. The element names have been created as *<AssestDescription>*, *<LocationDescription>*,*<AssetTypeDescription>*, and *<BranchDescription>*.

```
  ...
  <AlternateCodeSelectionFilterValue/>
</QueryOptions>
<AssetEntry>
  <AssetEntryHeader>
  <Asset>LB100</Asset>
  <AssetDescription>17 Baton Road, Erf 3577</AssetDescription>
  <Location>H/Office</Location>
  <LocationDescription>Head Office</LocationDescription>
  <AssetType>L</AssetType>
  <AssetTypeDescription>Land</AssetTypeDescription>
  <Branch>00</Branch>
  <BranchDescription>Head Office</BranchDescription>
  ...
```

Business Object Sample: Creating Your Own Sales Order

As mentioned previously, business objects can be thought of as the building blocks that enable you to put your own solution together. The following shows how you could use the business objects to build up the information required then create a sales order.

SORRSH

SORRSH is a *Build* method business object used to build the XML that you can use for the header of your sales order. The following is a sample of the XML that is provided to ***SORRSH***. Where elements contain the default value they have been removed to reduce the amount of space required.

```
<?xml version="1.0" encoding="Windows-1252"?>
<Build>
  <Parameters>
    <Customer>0000001</Customer>
  </Parameters>
</Build>
```

The XML that is returned contains information about this customer that can be used in the sales order header such as the salesperson, currency, and default delivery address. The values against the relevant elements should be stored.

SORRSL

SORRSL is also a *Build* business object. However, instead of building the information for the sales order header it is used to build the information for a sales order detail line. The *<StockCode>* and *<Customer>* elements are mandatory, but by supplying more data you can get back more meaningful information. For example, if you do not supply the *<OrderQuantity>* it will default to 1. This is fine if the order line is for a quantity of one of this stock code, but if the order is for more than one you should

supply the order quantity as the business object will take account of quantity discounts when returning the price and line value.

```
<Build>
  <Parameters>
    <StockCode>A100</StockCode>
    <Warehouse>E</Warehouse>
    <Customer>1</Customer>
    <OrderQuantity>11</OrderQuantity>
  </Parameters>
</Build>
```

You should check the returned values in case the item has been superseded, and to make sure that there is stock available. The values against the relevant elements should be stored. If there are multiple lines for this sales order you should repeat this process for each line, making sure that you store the details for each line.

SORQOV

The ***SORQOV*** business object is a *Query* class business object that is used to value a sales order. It can value an existing sales order, or if the header and detail lines are provided it can value this XML as if it were an order.

```
<?xml version="1.0" encoding="Windows-1252"?>
<Query>
  <Option>
    <OrderHeader>
      <Customer>000001</Customer>
      <DocumentType>O</DocumentType>
    </OrderHeader>
    <MerchandiseLine>
      <LineType>1</LineType>
      <MStockCode>A100</MStockCode>
      <MWarehouse>E</MWarehouse>
      <MOrderQty>11.000</MOrderQty>
      <MPrice>556.00</MPrice>
    </MerchandiseLine>
    <MerchandiseLine>
      <LineType>1</LineType>
      <MStockCode>A101</MStockCode>
      <MWarehouse>E</MWarehouse>
      <MOrderQty>3.000</MOrderQty>
      <MPrice>560.00</MPrice>
    </MerchandiseLine>
  </Option>
</Query>
```

Chapter 3 - Using e.net Diagnostics

What is e.net Diagnostics?

The e.net Diagnostics utility was designed to make it easy for developers to test their input to a business object, and see the XML output. It enables you to build your own XML, call a business object using this XML, and view the XML that is returned by the business object. There are two versions of the utility, namely *e.net Diagnostics* and *e.net Diagnostics Express*.

The e.net Diagnostics Express Version

The SYSPRO e.net Diagnostics Express version (from now will be referred to as "Express") was designed to be used without requiring an installation, as many System Administrators will not allow programs and utilities to be installed on their servers without extensive discussions and testing. Express consists of two files, `SYSPROEnetDiagnosticsExpress.exe` and `BusinessObjects.XML`, both of which must be placed in the SYSPRO `Base` folder.

If testing is performed directly on the SYSPRO application server, these two files are placed in the server's `Base` folder. Express is run by double-clicking on the *SYSPROEnetDiagnosticsExpress.exe* file. When run directly on the server, the sample XML files automatically become available (as they are kept in the server's `Base\Schemas` folder).

If testing is performed from a SYSPRO client machine the two files are placed in the client's `Base` folder. When Express is run the sample XML files should be present in the `Base\Schemas` folder, but if they are not, you should use the *Settings* option to point to their actual location.

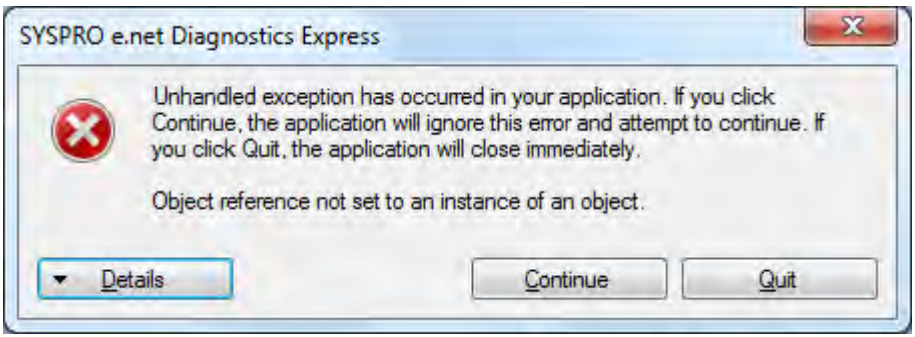


Figure 3-1: Running Express without `BusinessObjects.XML` being present

If Express is run without the `BusinessObjects.XML` file being present in the `Base` folder alongside `SYSPROEnetDiagnosticsExpress.exe`, an error message will be displayed (see **Figure 3-1**). This is resolved by locating the `BusinessObjects.XML` file and placing it in the `Base` folder.

When *Express* is loaded it consists of a menu, a toolbar, three tabs, and a status bar (see **Figure 3-2**).

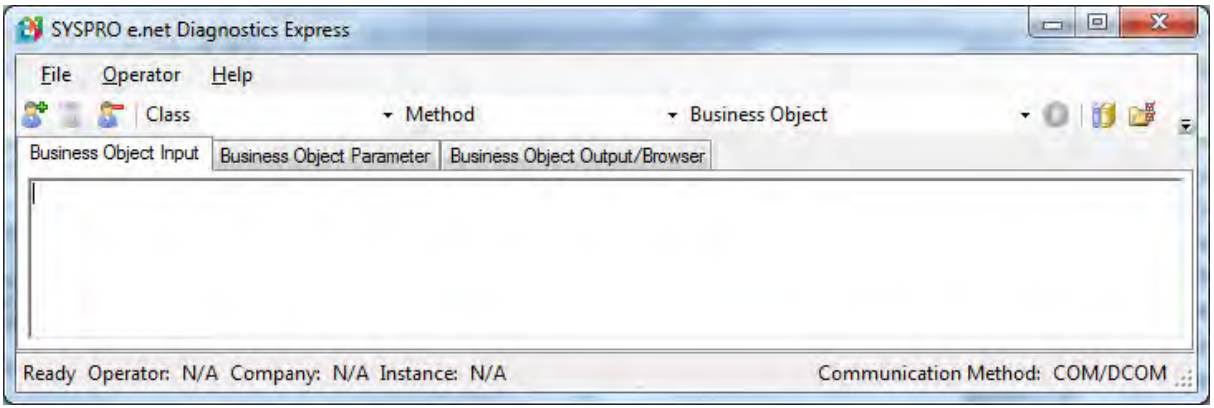


Figure 3-2: The *Express* version of e.net Diagnostics

The first two tabs accept text, and the third displays the results in a browser. The status bar reflects the current communication method, and this will default to *COM/DCOM* if this is the first time that you have loaded Express (as can be seen in **Figure 3-2**). There are currently two communication methods supported by Express, *COM/DCOM* and *Web Services*. The communication method can be changed using the radio buttons on the *System Settings* screen (*File* | *Settings*) that can be seen in **Figure 3-3.**

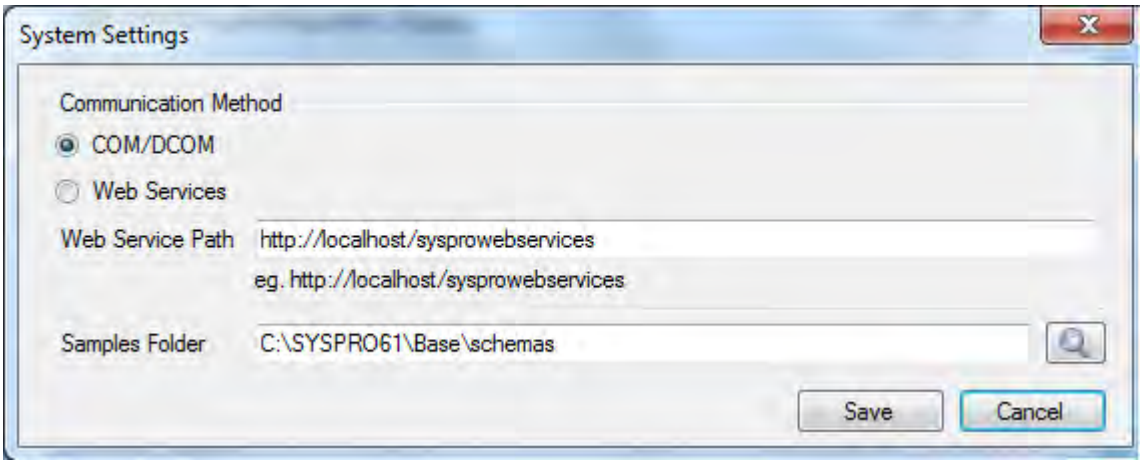


Figure 3-3: The *System Settings* screen

Also on the *System Settings* screen is the *Samples Folder* prompt that points to the location of the sample business object XML files, along with their matching schemas. The default location is determined by the registry, but it can be changed to point to an alternate location.

To logon to e.net Solutions, you can use either the *Logon to e.net solutions* button on the toolbar, or select *Operator* | *Logon* from the menu. In both cases the same *Logon* screen is displayed, which can be seen in **Figure 3-4**. Within the *Credentials* section are the prompts *Operator*, *Operator password*, *Company*, and *Company password*. The *Operator* prompt is mandatory, as is the *Operator password* if the operator has a password set against them. *Company* is optional if the operator has a default company set against them, and they are logging into the default company. If the company has a password set against it, *Company password* is also mandatory.

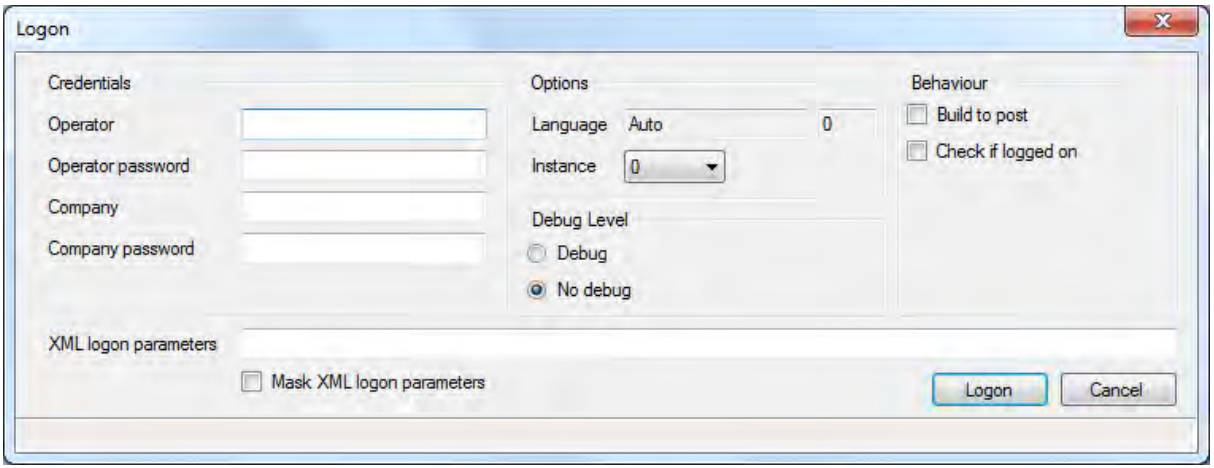


Figure 3-4: The e.net Diagnostics *Logon* screen

In Express, the Language option is hard-coded to *Auto*, so it picks up the language code held against the operator code in SYSPRO, and cannot be changed. The *Instance* has a dropdown list that defaults to zero, but allows you to select a number between 0 and 9 (where 0 will use the registry's *BaseDir* entry, 1 will use *BaseDir1*, 2 will use *BaseDir2*, etc.).

The *Debug Level* radio button enables you to select *Debug* or *No debug* mode. The *No debug* mode will allow any well-formed XML to be used as input to the business object, but will ignore all the elements that are not used by this specific business object. The concept of well-formed XML was covered in the *XML* section of Chapter 2.

In *Debug* mode, any element that is not used by this business object will cause an exception to be thrown containing the name of the element, and processing will stop. This mode is particularly useful when receiving XML from a third party that looks correct, but the business object does not appear to

be working as expected. Quite often there is a typo in the opening (and matching closing) element name, and no matter how hard you look you just cannot see it. Setting on *Debug* mode will throw an exception containing the offending element name.

Under the *Behavior* section are two checkboxes, *Build to post* and *Check if logged on*. If checked, the first will cause the logon to include the *<BuildToPost>* XML. If checked, the second will cause the logon to include the *<FailWhenAlreadyLoggedIn>* XML. You can read about how these two XML logon parameters affect processing by the business objects under *The e.net Solutions UserID and the Logon Process* section of Chapter 2.

The *XML logon parameters* section enables you to provide other XML parameters that can be seen in *The e.net Solutions UserID and the Logon Process* section of Chapter 2. An example appears in **Figure 3-5**.

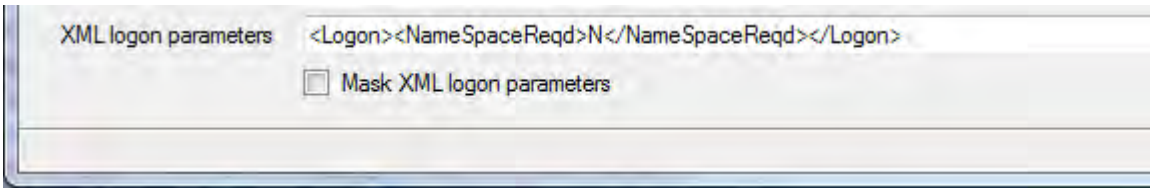


Figure 3-5: The *XML logon parameters* section of the *Logon* screen

The final option on this screen is the *Mask XML logon parameters* checkbox. When this is selected it replaces all the characters appearing against the *XML logon parameters* prompt with asterisks, so that the parameters cannot be read (see **Figure 3-6**). Unchecking the option will display them as normal text.

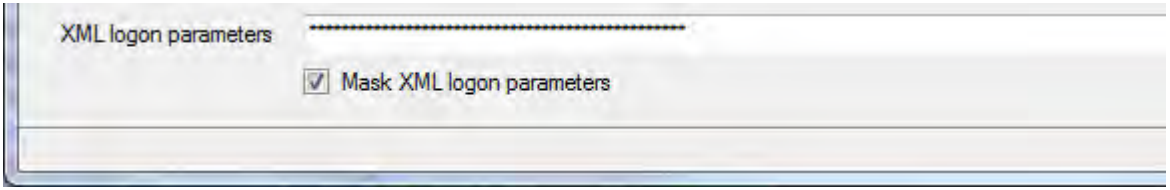


Figure 3-6: Masking the logon parameters

When you click on the *Logon* button, Express attempts to logon to e.net Solutions using the credentials supplied. If this is successful, the *Logon* screen will disappear and the returned *UserID* will appear against the *GUID* prompt on the main screen, in the top right-hand corner (see **Figure 3-7**). If the logon attempt fails, an error message will be displayed to explain the problem, and when you click on the *OK* button you will be returned to the *Logon* screen.

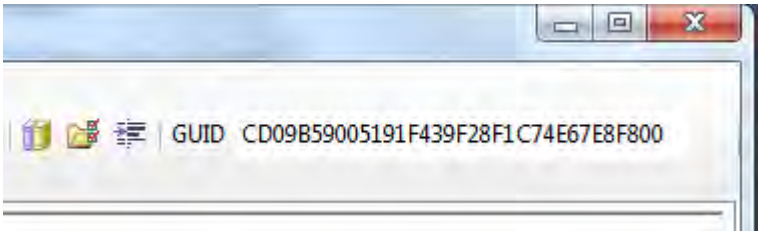


Figure 3-7: The displayed *UserID*

Next to the *Logon to SYSPRO e.net Solutions* button on the toolbar is the *Quick Logon* button. In Express, this will only be available if you have already logged on during this session. If this is clicked a logon will be attempted again using exactly the same credentials as the previous logon attempt. If successful, the only change that can be seen is that the *UserID* against the *GUID* prompt will change. If the logon fails for some reason (such as the XML parameter *<FailWhenAlreadyLoggedIn>* is set to *Y* and you did not logout) an error message explaining the problem will be displayed. Next to the *Quick Logon* button is the *Logoff* button, which will perform an e.net Solutions logoff.

Continuing along the toolbar there are three prompts with dropdown lists against them. The first is for the *Class*, and the options are *Query*, *Setup*, and *Transaction*. The second is *Method*, and the list of methods depends on the *Class* that was selected. The third is *Business Object*, and the list of business objects will be those in the `BusinessObjects.XML` file that match the selected *Class/Method* combination. Alternatively, you can type in the business object name (if it is known) without supplying the *Class*/*Method*.

Also on the toolbar is the *Business Object Library* button. When this is selected a list of all the business objects in `BusinessObjects.XML` is displayed, by functional area. If the required business object name is found you can highlight it and click on the *Select* button, or just double-click on it. The business object name will be inserted against the *Business Object* prompt on the main screen, and the *Class* and *Method* prompts will be updated accordingly. Next to this button is the *Load Sample XML* button. When this is clicked, Express looks in the folder specified against the *Samples Folder* prompt of the *Settings* screen to see if a sample XML file exists for this business object, and if one is found the *Business Object Input* tab is populated with its contents. For example, the inventory query business object ***INVQRY*** should have a sample called `INVQRY.XML`. There will be only one sample file because a *Query* class, *Query* method business object takes only one XML string. The *Inventory Warehouse Transfer In* business object ***INVTMI*** should have two sample files, `INVTMO.XML` and `INVTMODOC.XML`, as it is a *Transaction* class, *Post* method business object.

The next button on the toolbar is *Tidy XML Instance*. This is used to make the XML on the current tab easier to read, as well as parse the XML. It does not change the content of the XML, just indents each

level by two space characters. When used, this can make a big difference to the readability of the XML.

Although the *GUID* prompt is populated with the *UserID* when the logon happened, another *UserID* can be manually entered or pasted in. This is useful if you are attempting to duplicate a scenario using the *UserID* from another user who has not yet logged out (if they had logged out their *UserID* would have been removed from the state file, and no longer be valid).

The last button on the toolbar is the *Invoke Business Object* button. When clicked, this calls the selected business object and passes it the XML that has been provided. If the business object throws an exception, this is displayed in a message box, and processing stops. An exception occurs when a major issue is encountered that prevents the business object from continuing, or it is serious enough not to continue producing other error messages that relate to this error. An example of an error that would prevent the business object continuing is when the customer query business object is called and the XML contains an invalid customer code.

If the business object runs successfully, the results from the business object are displayed in the *Business Object Output/Browser* tab. This browser control allows sections of the output to be collapsed, or expanded. **Figure 3-8** shows the output from the business object used to add a new stock code. The minus signs at the beginning of each section show that that section can be collapsed. The plus sign at the beginning of the *<StatusOfItems>* element shows that it has been collapsed and can be expanded.

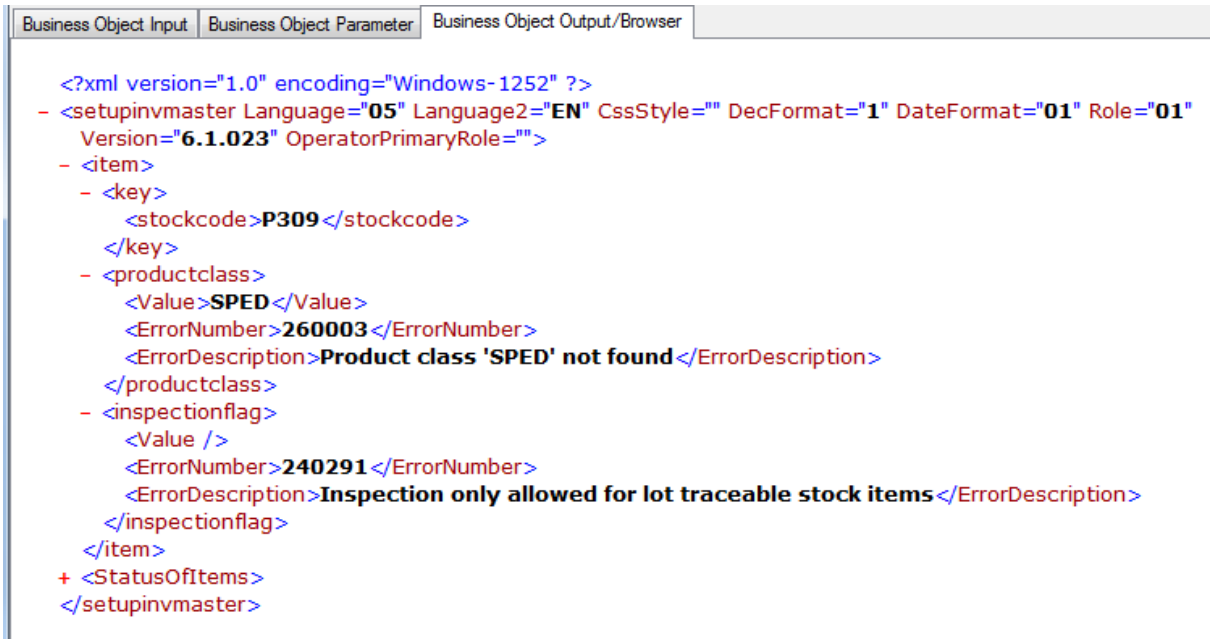


Figure 3-8: Business object output showing which sections can be collapsed and expanded

When using a *Transaction* or *Setup* class business object the results from the business object may contain error and/or warning messages. These will stop the transaction from completing, but will not cause an exception to be thrown. See the *Error Messages and Warnings* section in Chapter 2 for more information on how error messages are handled.

The final two options in Express enable you to save the XML that you have built up. They only appear under the *File* menu, not on the toolbar. The *Save As* option (*File* | *Save As*) is used to save the contents of the currently selected tab. It displays a browse so that you can find the location, and then supply the XML filename. The *Save All* option (*File* | *Save All*) uses a browse to find the location, and then saves the contents of all the tabs that currently contain values. If the *Business Object Input* tab contains XML, the content will be saved as `Document.xml`, the content of the *Business Object Parameter* tab will be saved as `Parameter.xml`, and the content of the *Business Object Output/Browser* tab will be saved as `Output.xml`.

The e.net Diagnostics Full Version

The full version of e.net Diagnostics needs to be installed before it can be used, and contains all of the functionality of the Express version, along with many additional features. It has a prerequisite of the Microsoft .NET Framework 2.0, but most machines have this already installed, as it is a prerequisite for *SYSPRO Reporting Services* (SRS). This diagnostics utility does not need to be installed on a machine that contains SYSPRO (either the client or the server), as the installation is self-sufficient.

The e.net Diagnostics utility contains the *File* menu, the *Tools* toolbar, the *Editing* toolbar, and the *Status* bar. It also contains a section with the *Business Object Input*, *Business Object Parameter*, and *Business Object Output* tabs. Another section contains the *Documentation* and *Additional information* tabs.

Figure 3-9 shows the e.net Diagnostics screen as it will appear when first loaded. At the very top is the *File* menu, directly under this is the *Tools* toolbar, and below that the *Editing* toolbar. The *Business Object Input*, *Business Object Parameter* and *Business Object Output* tabs are next, and these are where the XML is supplied to the business object, or returned by the business object.

The *Documentation* tab will display any documentation from the schema that relates to the currently selected field, and the *Additional information* tab will also display field specific information from the schema such as the maximum/minimum length, the default value, etc.

At the very bottom of the screen is the *Status* bar which contains information about the operator that is logged on, the company to which they logged on, the last business object executed, the position of the cursor within the XML, etc.

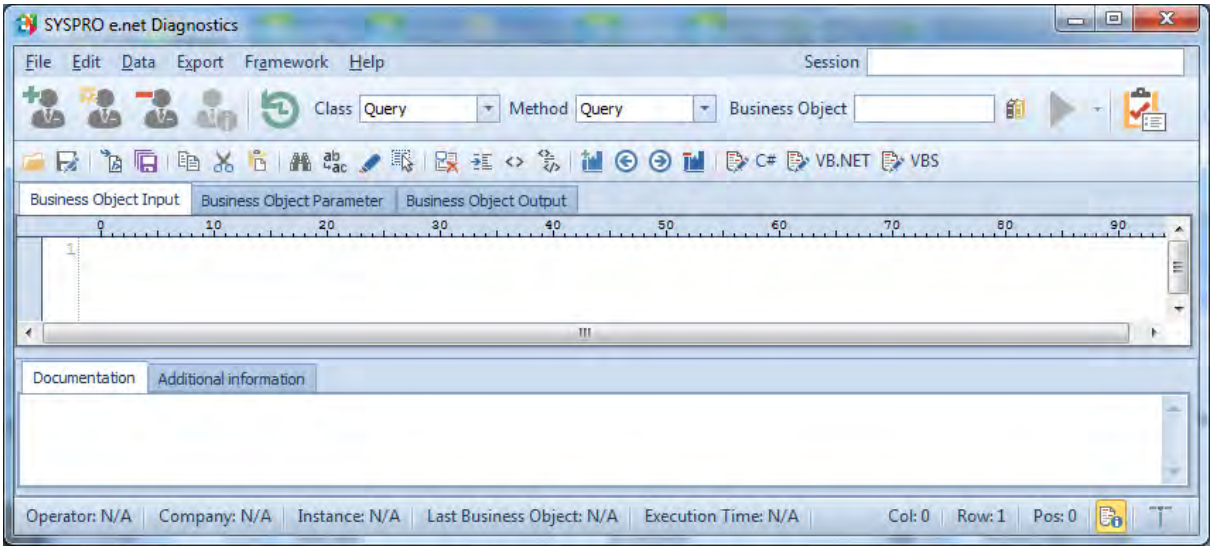


Figure 3-9: The e.net Diagnostics screen

The File Menu

The *File* menu consists of the *Open XML*, *Save XML*, *Load Sample XML*, *Save All XML*, *Settings*, and *Exit* options (see **Figure 3-10**). When either the *Business Object Input* or *Business Object Parameter* tabs are selected, the *Open XML* option will open a browse, allow you to select an XML file, and populate the currently selected tab with the contents of this file.

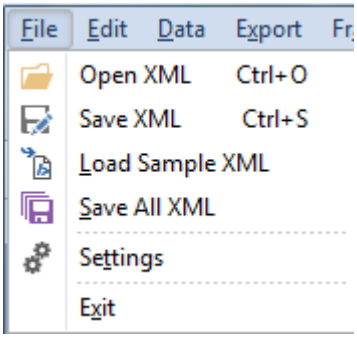


Figure 3-10: The *File* menu

The *Save XML* option will prompt for the location and filename, then save the contents of the currently selected tab to the supplied filename.

The *Save All XML* option prompts for a location, and then saves the contents of the *Business Object Input* tab to a file named *BusinessObject*_Document.XML and the contents of the *Business Object Parameter* tab to *BusinessObject*_Parameter.xml (where *BusinessObject* is replaced by the name of the business object in use in both cases). If one or both of the files already exist they will be overwritten without the operator being prompted. If the business object has been executed and the results are displayed in the *Business Object Output* tab, this is saved to the same location with the name *BusinessObject*_Output.xml.

If the XML was manually entered, and no business object name was supplied, the saved files would be called `_Document.xml` and `_Parameter.xml`.

SYSPRO ships with samples of the document and parameter XML for each business object, as well as schemas that explain the structure, what the XML can contain, and what it can't. These are stored in the SYSPRO `Base\Schemas` folder. Examples of the possible output from the business object are also included.

Some business objects such as queries only accept one XML input, known as the *Document* XML. The business objects that post transactions, and those that add/update semi-static information such as stock codes and customers, require two XML inputs. One contains the information to be processed, and the other contains information about how this will be processed. These are known as the *Document* XML and the *Parameter* XML respectively.

The *Load Sample XML* option is used to populate the *Business Object Input* tab with the sample document XML for the entered business object. If the business object requires a parameter XML the *Business Object Parameter* tab will be populated using the sample parameter XML.

If the business object name has not been provided at the time that the *Load Sample XML* option is selected, a message to this effect will be displayed.

Settings Option

The *Settings* option displays a screen containing two tabs, *Preferences*, and *Extensions* (see **Figure 3-11**). When checked, the *Auto Load XML* option causes the *Business Object Input* tab to be populated with the sample XML when a new business object name is entered and you tab off the field (or press the *Enter* key). If the business object requires a parameter XML entry, the *Business Object Parameter* tab is also populated. This option would typically be unchecked if you were pasting in XML examples from another system, or those that you had created for particular test scenario.

The *Specify Schema Path* option is used to supply the location containing the samples to be loaded. If this is left blank, the default location is assumed to be the `Schemas` folder under the SYSPRO *Base* folder.

The *Use UTF8 characters (COM/DCOM only)* checkbox is used to define whether the default *Windows-1252* character set is to be used or the *UTF-8* character set.

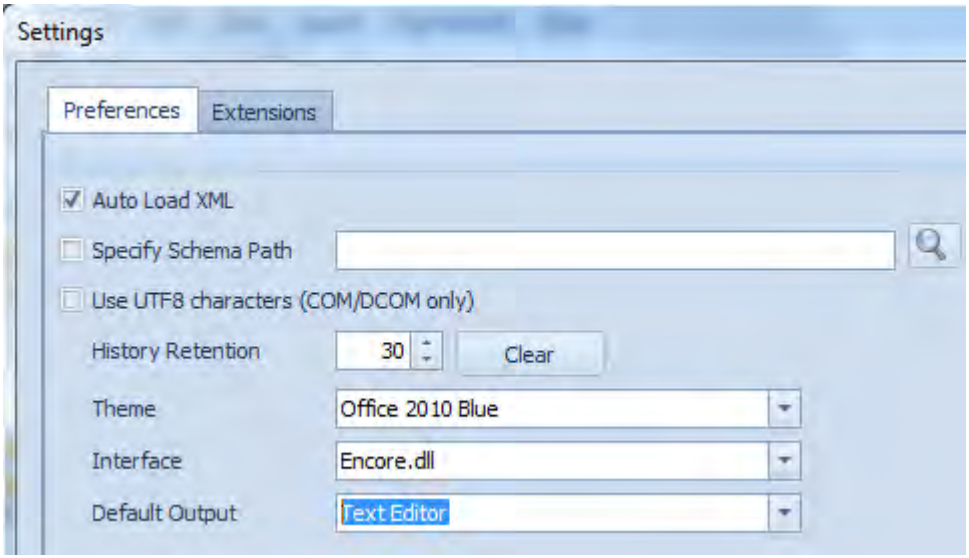


Figure 3-11: The *Preferences* tab of the *Settings* pane

Each time a business object is run, the date, time, business object name, business object description, business object version number, class, method, and any error message are stored. The *History retention* option specifies how many of these records are to be kept. The default is 30 records. The *Clear* button against this option is used to remove the history.

The *Theme* option allows you to choose from four colors/styles.

The *Interface* option specifies which COM object to use for e.net Solutions. The default is `Encore.dll` for backwards-compatibility. The decision of which communications type (COM/DCOM, web services, or WCF) is made at login time.

The *Default Output* option specifies the default type of output, but this can be changed when the business object is executed. The options are to display the output in a text editor, display the output in a web browser, or to write the output to an XML file. If the latter is selected you will be prompted for the filename and location at run time.

Although not in the version that shipped with SYSPRO 7, a new option has been added to this screen to automatically bring down the latest versions of the sample XML files and schemas. This option is *Auto Update Library* and will be available in a later version. The location to which this option must point is still to be confirmed.

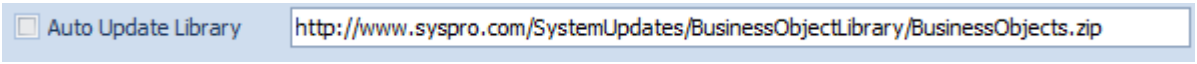


Figure 3-12: The *Auto Update Library* option that will be available in a later version

The *Extensions* tab allows you to add your own add-ins (or extensions) to the e.net Diagnostics menu. If any items are provided in the *Extensions* grid, a new *Tools* menu will appear between the *Framework* and *Help* menus. **Figure 3-13** shows the *Tools* menu after three items have been added to the *Extensions* tab.

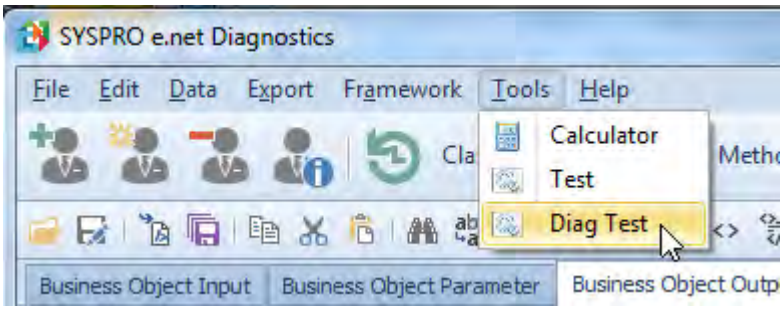


Figure 3-13: Items added to the *Extensions* tab appearing in the *Tools* menu

The *Extensions* tab consists of a grid containing columns for the *Name* and *Path* of the extension, a *Shortcut* column containing a checkbox, along with nine other columns containing checkboxes for parameters that are passed to the extension when the menu item is selected (see **Figure 3-14**).

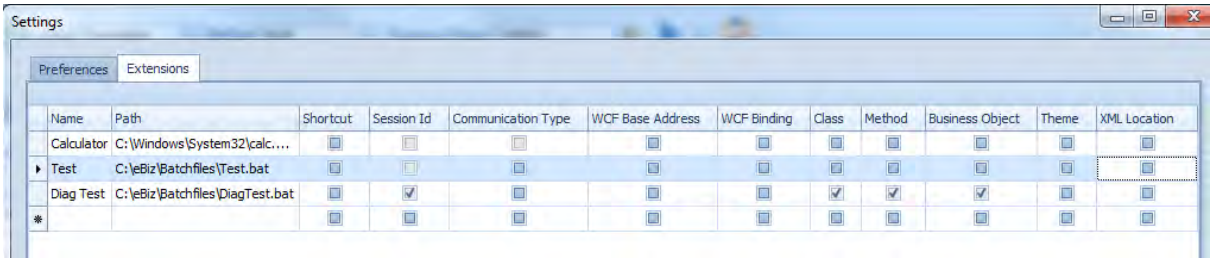


Figure 3-14: The *Extensions* tab of the *Settings* option.

The *Name* column should contain the name to appear in the menu. Although this can be a long, meaningful name, this is what will appear against the menu, so should be restricted to a reasonable length (see **Figure 3-15**).

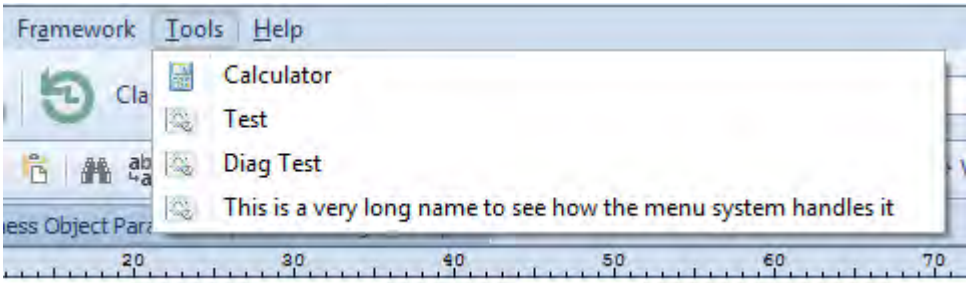


Figure 3-15: A long *Name* in the *Extensions* tab affecting the *Tools* menu

The *Path* column should contain the path, filename, and suffix of the executable to be run. There is a browse against cells in this column so that it is easier to locate and supply the entry. The default file type in the browse is .exe, so only files with this extension are displayed. However, you can enter the names of files with other suffixes by browsing to the relevant folder and typing in the full name of the file. For example, **Figure 3-14** contains three entries, the first has a suffix of .exe and the subsequent two have suffixes of .bat. Note that if the executable contains an embedded icon, this will be used alongside the text in the menu. The calculator icon was automatically added alongside the Calculator menu item.

If checked against an extension, the *Shortcut* checkbox adds a menu option to e.net Diagnostics called *Shortcuts*. This enables you to categorize your extensions (see **Figure 3-16** where the *Diag Test* option has the *Shortcut* checkbox checked, so appears against the *Shortcuts* menu).

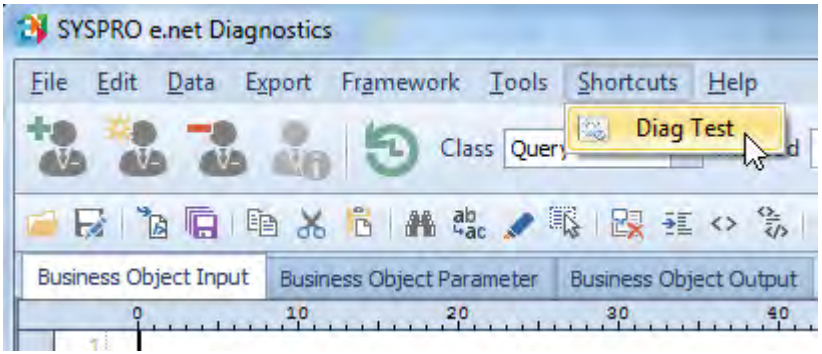


Figure 3-16: *Diag Test* extension appearing against the *Shortcuts* menu when *Shortcut* is checked

The nine columns containing the parameter checkboxes enable you to pass through parameters from the e.net Diagnostics program to the executable that is being invoked. Although these executables are being invoked from the e.net Diagnostics menu, they are being run standalone and have no knowledge of the e.net Diagnostics program, its environment, or any of its settings. These checkboxes

enable you to choose which settings are to be passed to the executable as parameters when you click on the menu item.

One of the benefits of this facility is that if SYSPRO wishes to add functionality to the e.net Diagnostics utility the functionality can be supplied as an extension, and this can be downloaded from the SYSPRO App Store.

The parameters that can be selected to be passed through to the extension are:

- *SessionID* – this contains the UserID that was returned when the user logged on to e.net Solutions. With this value the executable can also call business objects.
- *Communications Type* – this contains a 0 if the communications type is set to COM/DCOM, a 1 if the communications type is set to Web Services, and a 2 if the communications type is set to WCF Services.
- *WCF Base Address* – this contains the information about the configuration of the WCF Services (for example, it might contain something similar to `net.tcp://MyServer:501/SYSPROWCFService/Rest`).
- *WCF Binding* – this contains the WCF binding.
- *Class* – this contains the class that appears against the *Class* prompt on the e.net Diagnostics main screen at the time that the extension is invoked.
- *Method* – this contains the method that appears against the *Method* prompt on the e.net Diagnostics main screen at the time that the extension is invoked.
- *Business Object* – this contains the business object name that appears against the *Business Object* prompt on the e.net Diagnostics main screen at the time that the extension is invoked.
- *Theme* – this contains the theme in use by e.net Diagnostics, so if the other application has been written to use themes, the two applications can be made to look similar.
- *XML Location* – contains the location of the sample XML parameter and document files.

Only the parameters that are checked will be passed through, although the path and filename of the extension are always passed through. The sequence that the parameters are passed through is the sequence in which they appear in the *Extensions* tab. So if only the *SessionID* and *Business Object* columns are checked the *SessionID* will be parameter 1 and the name of the business object will be parameter 2.

A simple way to see exactly what is being passed through is to create a basic batch file similar to the one below. In this case it exists in the `C:\eBiz\Batchfiles` folder and is called *Diagtest.bat*.

This was added as an extension, with the *Name* of *Diag Test*, and can be seen as the third item in **Figure 3-14**. When extensions are added to the *Extensions* tab, they only appear in the menu the next time that e.net Diagnostics is run. Note that checking/unchecking the options against an extension that already appears in the menu will change which parameters are passed through immediately.

```
@echo Off
Echo %0 > C:\eBiz\Batchfiles\0.txt
Echo %1 > C:\eBiz\Batchfiles\1.txt
Echo %2 > C:\eBiz\Batchfiles\2.txt
Echo %3 > C:\eBiz\Batchfiles\3.txt
Echo %4 > C:\eBiz\Batchfiles\4.txt
Echo %5 > C:\eBiz\Batchfiles\5.txt
Echo %6 > C:\eBiz\Batchfiles\6.txt
Echo %7 > C:\eBiz\Batchfiles\7.txt
Echo %8 > C:\eBiz\Batchfiles\8.txt
Echo %9 > C:\eBiz\Batchfiles\9.txt
Pause
```

When this extension is run from the *Tools* (or *Shortcuts*) menu the batch file is executed. The name of path and filename of the extension is always available in parameter 0, and will be written to a file called `0.txt`.

The other files (`1.txt` to `9.txt`) will always be created, and will either contain the value of a parameter or the text *Echo is off* if there is no parameter. Using the example above where only the *SessionID* and *Business Object* columns are checked, the `0.txt` file will contain the path/filename of the extension, the `1.txt` file will contain the UserID, the `2.txt` file will contain the name of the business object, and `3.txt` to `9.txt` will contain the text *Echo is off*.

The Edit Menu

The *Edit* menu consists of *Copy*, *Cut*, *Paste*, *Find*, *Replace*, *Highlight*, and *Select All*. These options work as you would expect, so will not be covered here. The *Find* option locates and highlights the first occurrence of the required text. You use the *Next* button on the *Find* screen to locate and highlight the next occurrence. The *Highlight* option highlights all occurrences at the same time, as can be seen in **Figure 3-17**.

```
<IncludePostDated>Y</IncludePostDated>
<IncludeZeroBalances>N</IncludeZeroBalances>
<IncludeCustomForms>N</IncludeCustomForms>
```

Figure 3-17: Using the *Highlight* option

The Data Menu

The *Data* menu has two sections. The upper section contains options to manipulate the text in the currently selected tab. These are *Remove Comments*, *Tidy XML*, *Remove Inner text*, and *CDATA Section*. The lower section is for moving around within the text of the currently selected tab, typically

when there is a lot of text. These are *Toggle Bookmark*, *Previous Bookmark*, *Next Bookmark*, and *Clear Bookmarks*.

Remove Comments
Almost all of the sample XML files supplied with SYSPRO, and some that you will receive from other people, contain comment lines. Comment lines in XML start with <!-- and end with -->. The following is an example of a comment line, a variation of which will appear in all SYSPRO sample XML files:

```
<!-- Copyright 1994-2014 SYSPRO Ltd.-->
```

Figure 3-18 shows the top of the sample XML that is used with the ***ARSQRY*** business object to query a customer's information.

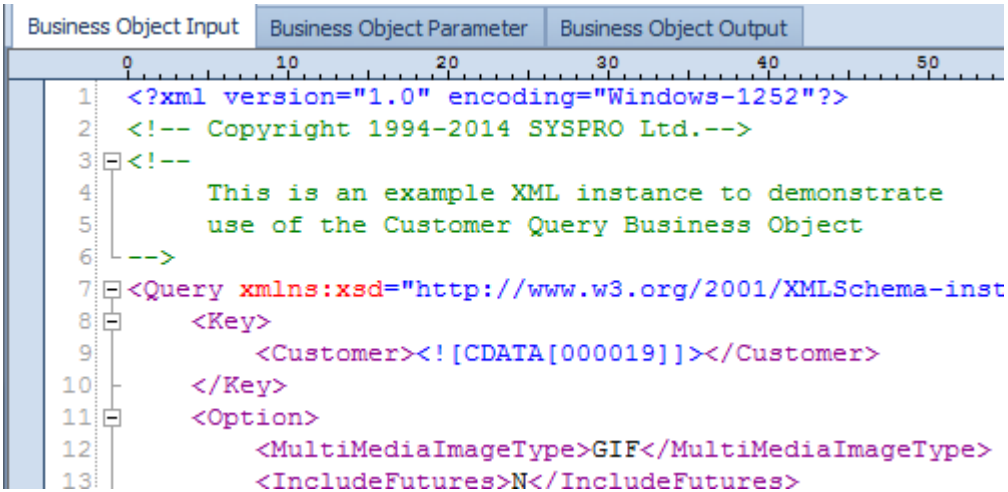


Figure 3-18: Sample XML for the ***ARSQRY*** business object containing comment lines

Although comment lines do not affect the processing of the XML, in some cases when there are many of them, it can make it difficult to follow the structure and content of the XML. The *Remove Comments* option strips out the comment lines in the XML on the currently selected tab. It does not affect other tabs.

Figure 3-19 shows the same sample XML as **Figure 3-18**, but after the *Remove Comments* option has been selected.

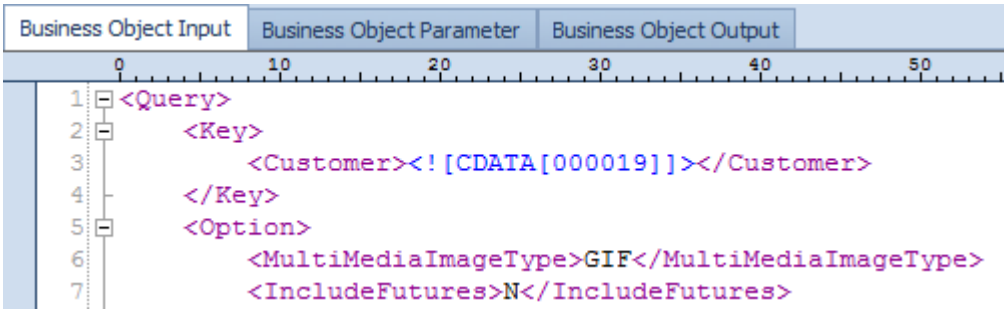


Figure 3-19: Sample XML for ***ARSQRY*** with the comments removed

Tidy XML

When XML appears in an unstructured format it can be very difficult to follow. The *Tidy XML* option takes the text in the currently selected tab and checks that it is well-formed XML (for more information on well-formed XML, see the sub-section *Some Basic Syntax Rules* under the XML section in Chapter 2). If the XML is well-formed, the display will be reformatted to make it easier to follow. This is useful when you receive XML from a third party and it is either one long string, or built up from information from different places.

Figure 3-20 shows an example where several elements appear directly after each other on the same line, and another element appears further to the left than its parent.

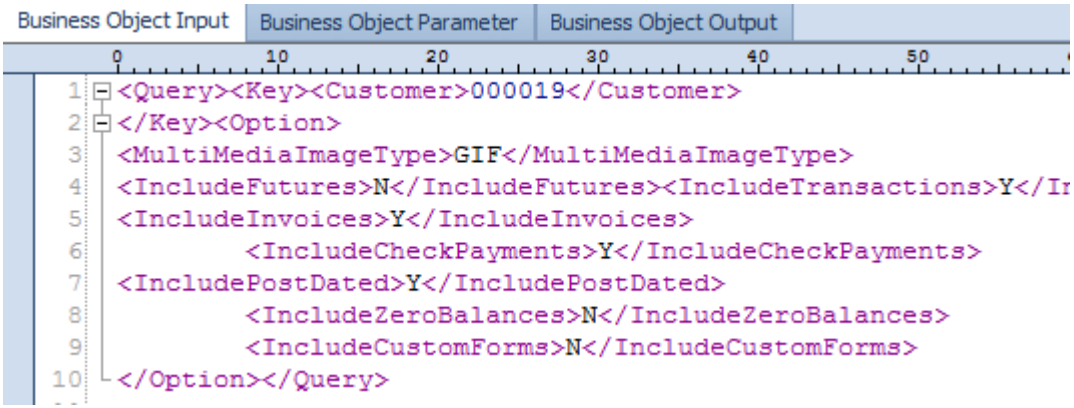


Figure 3-20: A well-formed, but not easy to follow, XML file

Figure 3-21 shows the same XML after the *Tidy XML* option has been used. The indentation is two characters. This XML is much easier to follow than that in **Figure 3-20**.

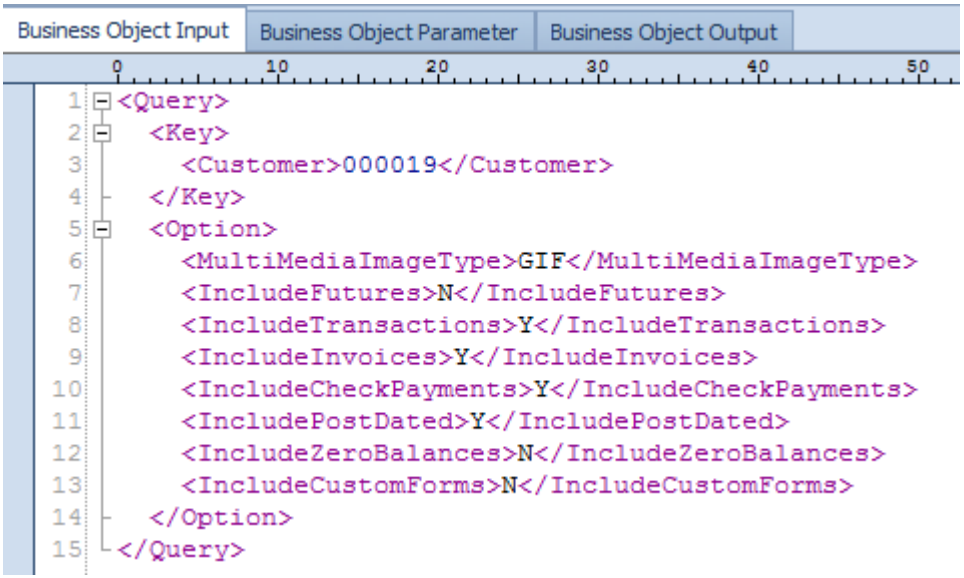


Figure 3-21: The same XML after the *Tidy XML* option has been used

Remove Inner Text

The *Remove Inner Text* option strips out all the values in your XML, leaving just the element names. This is useful if you have built up XML using values (that may be confidential), and now you need to send the XML structure to someone else (or keep a copy for yourself). If you need to also tidy up the XML, make sure that you tidy it up first, as doing this afterwards causes the opening and closing element names to be on their own line.

CDATA Section

The *CDATA Section* option wraps the highlighted value in a CDATA section. A CDATA section enables you to pass a value in your XML that contains a character that has a special meaning to XML, such as a less-than sign, a greater-than sign, or an ampersand. The list of XML special characters appears in **Table 2-2** in Chapter 2, and an explanation of CDATA sections appears in the *Samples and Schemas* section of the same chapter.

If you had the following element within your XML, and you need to keep the ampersand as it stands, you should highlight the value and select the *CDATA Section* option.

```
<MyKeyValue>A & B Together</MyKeyValue>
```

After selecting the *CDATA Section* option you would end up with the following XML:

```
<MyKeyValue><![CDATA[A & B Together]]></MyKeyValue>
```

Bookmarks
A bookmark is a marker placed against a line. You can move to the next or previous bookmark using options against the *Data* menu. Bookmarks become useful when you have a very large XML file and need to move backwards and forwards through the XML. **Figure 3-22** shows a section of XML that has bookmarks against three lines (they appear at the far left of the line). A bookmark does not affect the processing of the XML by a business object.

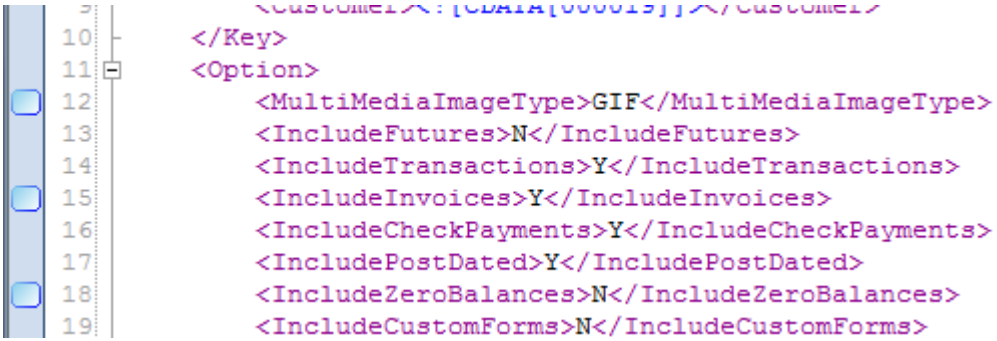


Figure 3-22: Three of the lines of XML have bookmarks against them

The *Toggle Bookmarks* option adds/removes a bookmark on the current line. The *Next Bookmark* option moves to the next bookmark down the XML file. If this bookmark is the last one in the file it goes to the top one in the file. The *Previous Bookmark* option goes to the next bookmark up the file. If the current bookmark is the highest one up the file, the *Previous Bookmark* option will start at the bottom one. The *Clear Bookmarks* option removes all bookmarks from the XML.

The Export menu

The *Export* menu contains the options *Export C#*, *Export VB.NET*, and *Export VB Script*. These options are used to wrap the XML with the relevant code to enable it to be used by the selected language.

The three options only work on the *Business Object Input* and *Business Object Parameter* tabs, and only if they contain text. No check is made to see that this is *well-formed* XML, or that this is *valid* (i.e. matches what is allowed by the schema). The output is displayed in a text editor.

The following is an example of XML that could appear in the *Business Object Input* tab:

```
<Query>
  <Key>
    <Customer>XXXXXX</Customer>
  </Key>
  <Option>
    <IncludeTransactions>Y</IncludeTransactions>
  </Option>
</Query>
```

If the *Business Object Input* tab is the tab currently in focus and the *Export VB Script* option is selected, the text editor will contain the following:

```
Dim Document

'Building Document content
Document = "<Query>"
Document = Document & "<Key>"
Document = Document & "<Customer>XXXXXX</Customer>"
Document = Document & "</Key>"
Document = Document & "<Option>"
Document = Document & "<IncludeTransactions>Y</IncludeTransactions>"
Document = Document & "</Option>"
Document = Document & "</Query>"
```

The Framework menu

The *Framework* menu contains the *Logon*, *Express Logon*, *Logoff*, *Get Logon Profile*, *History*, *Invoke Business Object*, and *Validate Schema* options.

Logon

As covered in Chapter 1, to be able to access an e.net Solutions business object outside of SYSPRO, you must first logon to e.net Solutions. Selecting the *Logon* option displays the screen that allows you to provide the information to be able to logon.

Figure 3-23 shows the *Logon Credentials* tab of the *Logon* screen. This has two main sections, one containing the means of connecting to e.net Solutions, and the other how to logon once the connection has been made.

The three means of connecting are *COM*/*DCOM*, *Web Services*, and *WCF Services*. Selecting the *COM*/*DCOM* radio button specifies that e.net Diagnostics should attempt to connect to the e.net Solutions COM object. If e.net Diagnostics is installed on the SYSPRO application server this will be via COM. If e.net Diagnostics is installed on a computer that does not contain SYSPRO, the DCOM client can be installed on this machine and if configured correctly, the COM request is routed via DCOM to the COM object on the SYSPRO application server.

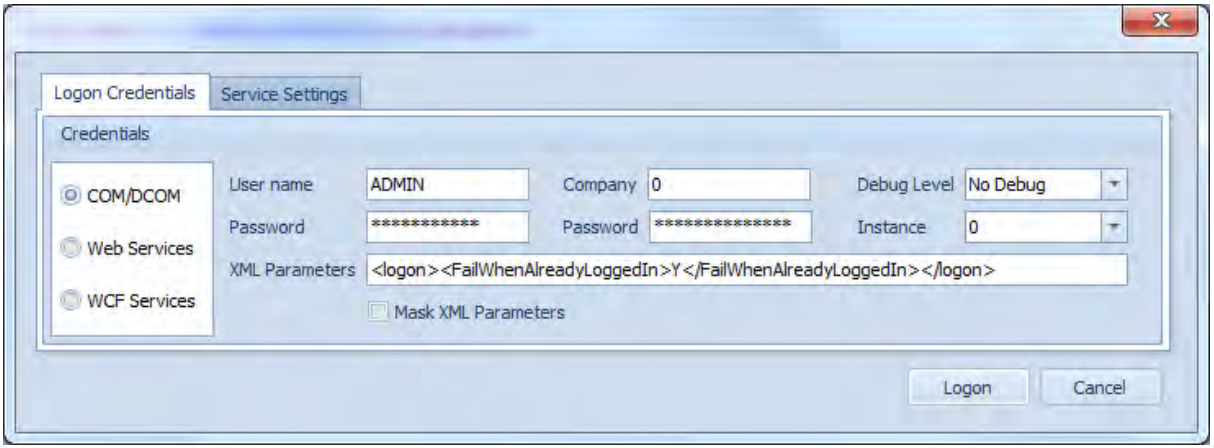


Figure 3-23: The *Logon Credentials* tab of the *Logon* screen

If the *Web Services* radio button is selected, the web service address configured against the *Service URI* prompt on the *Service Settings* tab (see **Figure 3-24**) to connect to the web service.

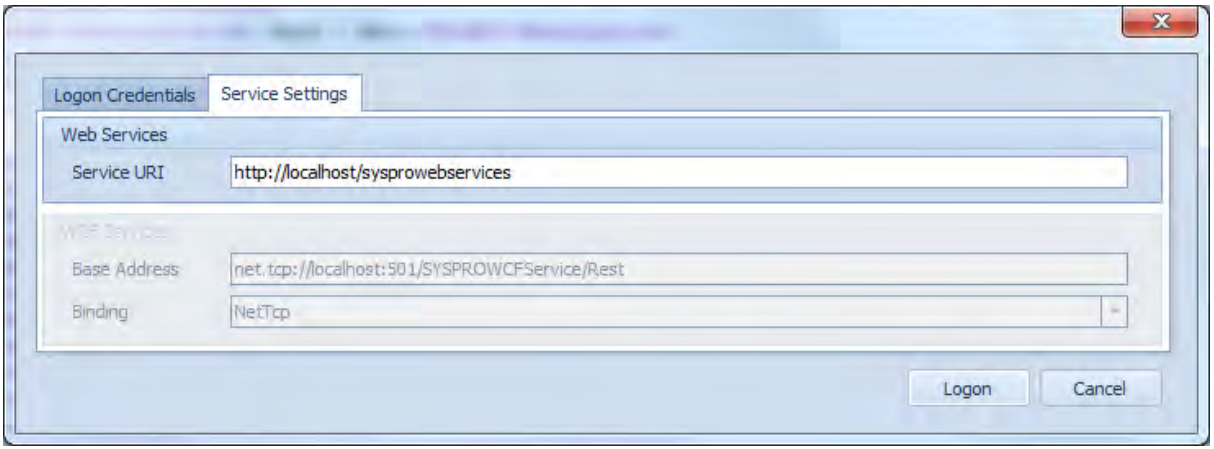


Figure 3-24: The *Service URI* option under the *Web Services* section on the *Service Settings* tab

If the *WCF Services* radio button is selected, e.net Diagnostics will use the settings under the *WCF Services* section on the *Service Settings* tab (see **Figure 3-25**) to connect to the WCF services.

The *User name*, *Password*, *Company* and *Password* options are as you would use to login to SYSPRO. If no password has been configured against the operator code or company ID these do not need to be provided. If the operator code has a default company ID configured against it within

SYSPRO, and this is the company that the operator wishes to logon to in e.net Solutions, this does not need to be provided either as it will default to this.

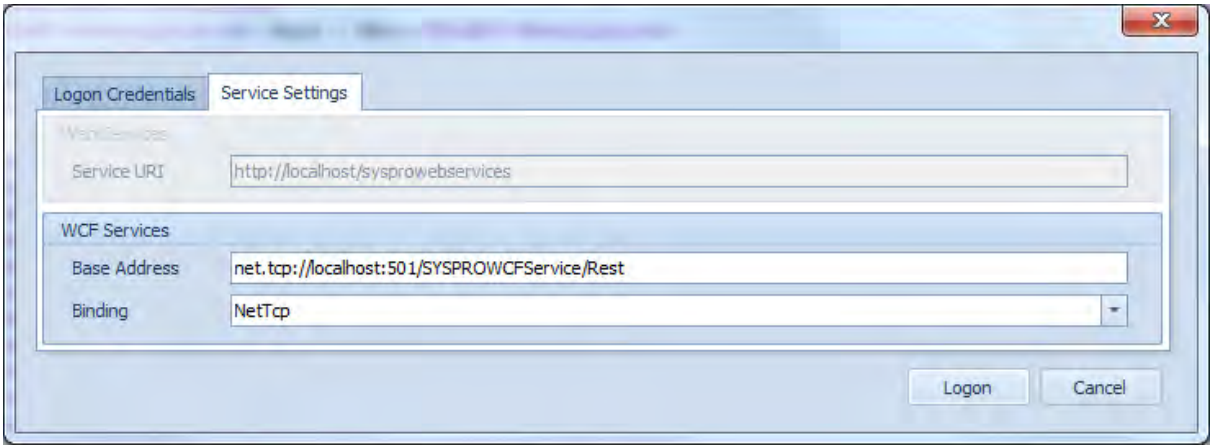


Figure 3-25: The *Base Address* and *Binding* options under the *WCF Services* section

The *Debug Level* option is used to highlight issues in the XML being used once the operator has successfully logged on. The options are *Debug* and *No Debug*. *No Debug* mode allows any *well-formed* XML to be used as input to the business object, but will ignore all the elements that are not used by this specific business object (*well-formed* XML was covered in the *XML* section of Chapter 2).

In *Debug* mode, any element that is not used by this business object will cause an exception to be thrown containing the name of the element, and processing will stop. This mode is particularly useful when receiving XML from a third party that looks correct, but the business object does not appear to be working as expected. Quite often there is a typo in the opening (and matching closing) element name, and no matter how hard you look you just cannot see it. Setting on *Debug* mode will throw an exception containing the offending element name. An example appears below.

If you had the following XML that you were using when calling the customer query business object (***ARSQRY***), but the results were still returning future invoices, you might be stumped. If you had logged on with the *No Debug* option set there would be no error message.

```
<Query>
  <Key>
    <Customer>1</Customer>
  </Key>
  <Option>
    <IncludeFuture>N</IncludeFuture>
    <IncludeTransactions>Y</IncludeTransactions>
  </Option>
</Query>
```

However, logging in again with the *Debug* option set, and then attempting to process the XML with the ***ARSQRY*** business object would cause an exception to be thrown (see **Figure 3-26**) which highlights that the *IncludeFuture* element is not one used by this business object (it should be *IncludeFutures*):

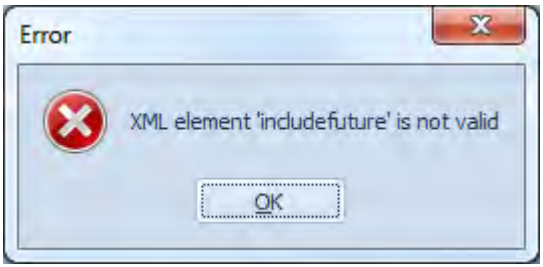


Figure 3-26: An exception being thrown when the element is not one used by this business object

The *Instance* option has a dropdown list against it containing 0 to 9. Most installations of SYSPRO will only have one instance, and this will default to 0. If multiple copies of SYSPRO exist on one SYSPRO application server, such as in a test environment, each of these can be configured as an instance. When logging on to e.net Solutions using e.net Diagnostics you can specify which instance to use, or use the default of zero. Instances are covered in more detail within the *The e.net Solutions UserID and the Logon Process* section of Chapter 2.

The *XML Parameters* section enables you to provide other XML parameters that can be seen in *The e.net Solutions UserID and the Logon Process* section of Chapter 2. An example appears in **Figure 3-23** where the logon will be prevented from happening if this operator code is already logged on.

The *Mask XML Parameters* checkbox appears immediately below the *XML Parameters* section. When this is selected it replaces all the characters against the *XML Parameters* prompt with asterisks, so that the parameters cannot be read. Unchecking the option will display them as normal text.

The language used during the logon process is *Auto*, which means that it will pick up the language code held against the operator code in SYSPRO.

The *Logon* button attempts to login to e.net Solutions using the supplied information, and the *Cancel* button will exit the logon screen. A successful login attempt will return a 34-character *UserID* (prior to SYSPRO 7 this was 33 characters). This *UserID* will be displayed against the *Session* prompt in the top right-hand corner of the e.net Diagnostics screen. If the logon attempt was not successful an error message will be displayed that attempts to convey the reason for the failure.

Express Logon
The *Express Logon* option will attempt a logon using the same credentials that were used for the last logon attempt. If the logon attempt fails, a message will be displayed that conveys the reason for the failure. If the logon attempt is successful the *Session* prompt will be updated with the new *UserID*.

Logoff
The *Logoff* option will log the current user off of e.net Solutions. The *UserID* against the *Session* prompt will be cleared and the user will need to login to be able to call a business object.

Get Logon Profile
The *Get Login Profile* option is only available if the user is logged on. This option calls the *GetLogonProfile* method of the *Utilities* class, and returns XML containing information about this operator, the company that they are logged on to, and the environment.

History
The *History* option works in conjunction with the *History Retention* option (*File* | *Settings* | *History Retention*). The *History Retention* option was covered in detail above (*The File Menu* section).

When the *History* option is selected the *History* pane is displayed above the tabs containing the business object XML. An example of the sort of information that is displayed appears in **Figure 3-28**. This has been grouped by date (the *Group by* field would appear above the information displayed in **Figure 3-28**, but has been omitted as it would have made the screen too wide to be displayed properly on the page).

Time	Busine...	Description	Version	Class	Met...	Error message
Date: 25/09/14						
05:54:04 AM	COMTFM	Business object...	7.0.001	Transaction	Post	
05:53:51 AM	COMTFM	Business object...	7.0.001	Transaction	Post	
Date: 22/10/14						
10:44:39 AM	ARSQRY	Customer Query		Query	Query	XML element 'includefuture' is nc
10:16:38 AM	ARSQRY	Customer Query	7.0.021	Query	Query	
10:16:26 AM	ARSQRY	Customer Query		Query	Query	XML element 'paul' is not valid
10:16:13 AM	ARSQRY	Customer Query	7.0.021	Query	Query	
10:15:33 AM	ARSQRY	Customer Query	7.0.021	Query	Query	
10:15:16 AM	ARSQRY	Customer Query		Query	Query	XML parsing error 'The end tag '

Figure 3-27: The main section of the *History* pane

Within this screen are the *Time*, *Business object*, *Description*, *Version*, *Class*, *Method*, and *Error message* columns. The *Time* column contains the time that the business object was invoked. The *Business object* column contains the name of the business object. The *Description* column contains the description for the business object, or if the description is not known it contains *Business object not found*. The *Version* column contains the version number of the business object. The *Class* and *Method* columns contain the class and method of this business object. The *Error message* column contains the content of any exception that was thrown, as opposed to error (or warning) messages returned in the XML.

The default *Group by* sequence is the *Date* column, in which case the *Date* column name appears above the grid (see **Figure 3-28**).

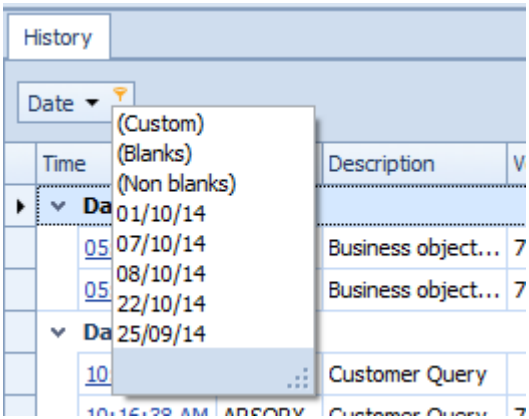


Figure 3-28: The *Group by* section showing the date option, and the ability to filter by date

Clicking on the *Date* column name toggles the sequence (ascending or descending). When the mouse pointer hovers over the *Date* column an icon appears that shows that a filter can be applied (this can also be seen to the far right of the *Date* column in **Figure 3-28**). This filter contains a list of all the unique dates, as well as *(Custom)*, *(Blanks)*, and *(Non blanks)* options. Clicking on one of these will restrict the display to items matching the filter value.

Back on the main *History* grid, the values against the *Time* column are hyperlinked. Clicking on one of these populates the *Business Object Input* tab with the document XML that was used when the business object was originally invoked, and the *History* pane is closed. If the business object requires a parameter XML, the *Business Object Parameter* tab is populated with the parameter XML that was supplied when the business object was originally invoked.

When the *History* pane is open, clicking on the *History* option closes the pane.

Invoke Business Object

The *Invoke Business Object* option is only available when the operator is logged on to e.net Solutions. The business object is called and the *UserID* and XML passed to it. If an exception is thrown, a message is displayed containing the error. If no exception is thrown focus is set on the *Business Object Output* tab and the output from the business object will appear there.

The *Class* and *Method* supplied must match the business object. **Figure 3-29** shows the exception that is thrown when attempting to call the ***SORTOI*** business object (which is uses the *Transaction* Class, *Post* method) with the settings of *Query* class and *Query* method.

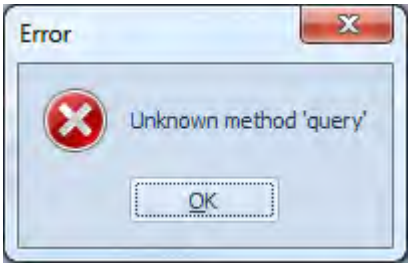


Figure 3-29: Calling a business object using the wrong method

Validate Schema

The *Validate Schema* option is used to ensure that the XML is *Valid* (which means that it conforms to the definition laid out in the schema). If the XML conforms to the schema, a message will be displayed to this effect. If the XML does not conform to the schema, a message will be displayed that attempts to explain the issue.

Figure 3-30 contains the error returned by the *Validate Schema* option when the *IncludeCustomForms* element contained a value of X, when the only valid options are N and Y, and the default is N if the element is not present.

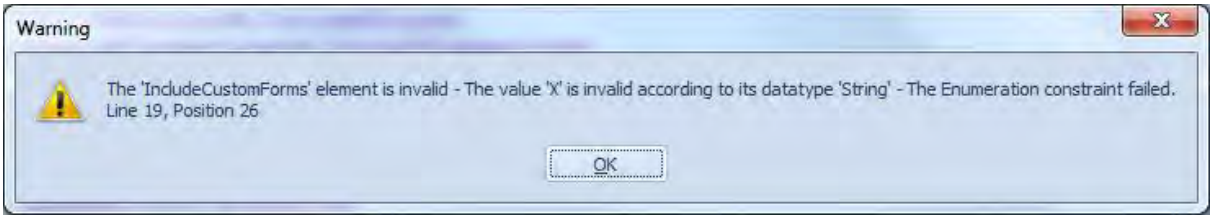


Figure 3-30: An error message returned by the *Validate Schema* option when the XML is not *Valid*

In most cases the schema can be used to validate the contents of your XML against the schema file. However, because e.net Solutions is so flexible there are a few cases where the schema validation may fail, even though the business object will successfully process the XML.

An example where the schema validation may incorrectly report a problem is the sales order import business object ***SORTOI***. Within the <*OrderDetails*> element there are four possible sub-elements, <*StockLine*>, <*CommentLine*>, <*MiscChargeLine*>, and <*FreightLine*>. The <*OrderDetails*> element can have up to 9999 detail lines in any combination of these four sub-elements, and in any sequence.

However, the schema language only caters for the sub-elements to be defined as either *All*, *Sequence*, or *Choice*. The *All option* would require at least one of each sub-element to be present, so even if you did not have a miscellaneous charge, you would have to enter one. The *Sequence* option would force the sub-elements to appear in a specified sequence, such as all the <*StockLine*> elements to appear first, followed by the <*CommentLines*>. The *Choice option* specifies that only one of the sub-element types can be present.

In the case of the ***SORTOIDOC*** schema, the type is set to *Sequence*, because this is the closest to what the business object allows. However, the business object will allow these sub-elements to be in any sequence.

The Help Menu

The *Help* menu contains the *About* option that is used to display the version number of e.net Diagnostics.

Tools Toolbar

The *Tools* toolbar sits just below the *File* menu at the top of the e.net Diagnostics screen, and can be seen in **Figure 3-31**. The purpose of this toolbar is to make the most commonly used functions of e.net Diagnostics available using a single click of the mouse.

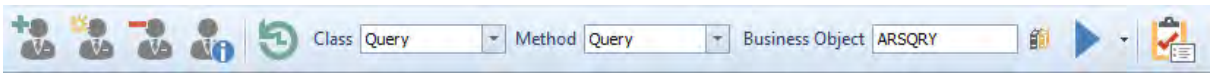


Figure 3-31: The *Tools* toolbar

The first five buttons on the toolbar are *Logon*, *Express Logon*, *Logoff*, *Get Logon Profile*, and *History*. These correspond to the matching items on the *Framework* menu, so have been covered already.

The *Class* prompt has a dropdown list alongside it containing the *Setup*, *Transaction*, and *Query* class names. The *Method* prompt also has a dropdown list against it, and this will be populated with the available methods for the class that is selected against the *Class* prompt.

A business object name can be entered against the *Business Object* prompt. If this business object exists in the library the values against the *Class* and *Method* prompts will automatically be changed to match those for this business object. When a *Setup* class business object is entered it can have up to three possible methods (*Add*, *Update*, and *Delete*) depending on the business object. It will default to the *Add* method, if this is available for this business object.

Next to the *Business Object* prompt is the *Business Object Library* button. If this is selected, a dropdown list option appears alongside the business object entry field. When this is selected, the library is opened, and consists of modules, with the functional areas within these. When the functional areas are expanded, a list of business objects for this functional area are displayed consisting of the name, the description, the class and method.

Figure 3-32 shows where the *Accounts Payable* module has been expanded to show the *AP Primary Posting* and *AP Primary Query* functional areas. The *AP Primary Query* functional area has been expanded so that some of the business objects can be seen.

Clicking on one of the business object names will populate the field against the *Business Object* prompt with this value, the *Class* and *Method* prompts will be updated to match, and the *Business Object Library* will be closed. To remove the dropdown list against the business object without selecting a business object from those available, click on the *Business Object Library* button again.

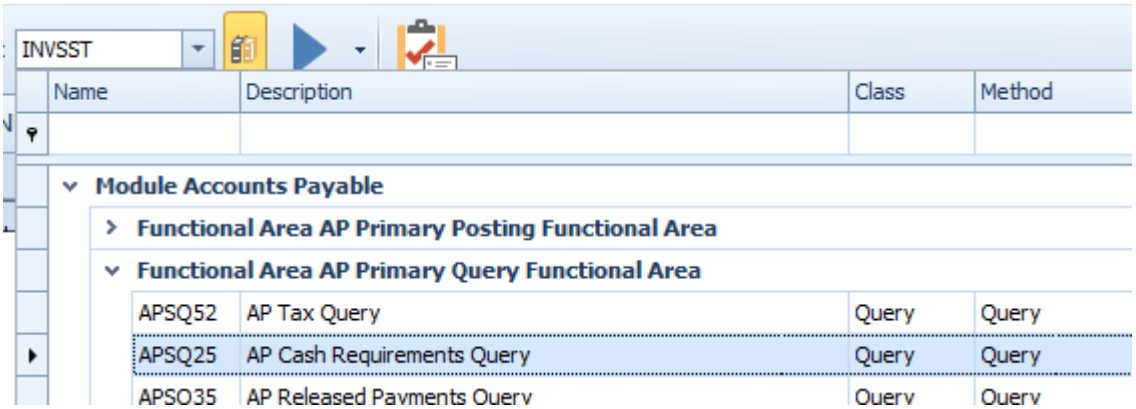


Figure 3-32: The *Business Object Library* showing modules, functional areas, and business objects

If the *Business Object* prompt is empty, when you click on the *Business Object Library* button to open it, as you start typing in the name of a business object against the prompt the library will display only the modules/functional areas/business objects that match what has been typed so far. If no module has been expanded it will only show the modules that contain business objects whose names match. If some modules have been expanded, it will also show the functional areas containing the matching

values. If some of the functional areas have been expanded, it will also show the business objects that match.

The *Invoke Business Object* button invokes the business object and passes it the XML file(s) and *UserID*. If the business object throws an exception, a message box displays this error message. If the business object does not throw an exception, the XML returned by the business object is displayed in the *Business Object Output* tab. The format of this tab is set against the *Default Output* option of the *Preferences* tab of the *Settings* option.

Alongside the *Invoke Business Object* button is a slim button that enables you to change the format of the *Business Object Output* tab. The button can be seen in **Figure 3-33**.

Figure 3-33: The option to change the format of the *Business Object Output* tab

When this button is selected a dropdown list appears containing three radio buttons; *Text Editor*, *Web Browser*, and *Output File* (see **Figure 3-34**). Selecting one of the first two options will change the format of the *Business Object Output* tab to match. If this already contains information it will appear in the new format. Selecting the third option will blank out the *Business Object Output* tab if it already contains values. The next time that a business object is invoked you will be prompted for the location where the file must be saved, and given the opportunity to change the filename from its default (subject to the business object not throwing an exception).

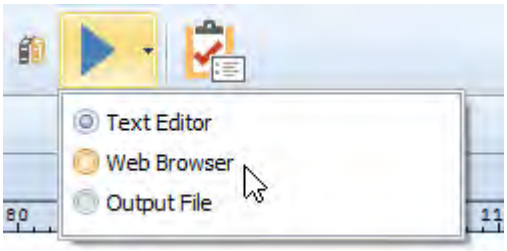


Figure 3-34: The radio buttons that can be used to change the *Business Object Output* tab

The *Validate Schema* button performs the same task as the option under the *Framework* file menu option.

Editing Toolbar

The *Editing* toolbar appears in **Figure 3-35**. All of the items on this toolbar tie up with items on the *File* menu, which were covered above.

Figure 3-35: The *Editing* toolbar

Documentation Tab

The *Documentation* tab and the *Additional information* tabs appear near the bottom of the screen (see **Figure 3-36**). The *Documentation* tab will only contain information if an element name is selected within either the *Business Object Input* or *Business Object Parameter* tabs. These two tabs can be hidden/shown using the *Schema Information* button on the status bar at the bottom of the screen.

Figure 3-36: The *Documentation* and *Additional information* tabs

When an element name is selected on one of these tabs the documentation text for this element in the matching schema is extracted and displayed in the *Documentation* tab. The name of the tab is changed to include the name of the element that was selected. Note that you must click on the opening or closing element names. If you click on the value between these, the documentation is not shown.

Figure 3-37 shows a section of the *Business Object Input* tab containing the ***SORTOI*** business object XML. The cursor has been placed on the *OrderActionType* element. The *Documentation* tab has been changed to *Documentation for OrderActionType*, and the tab contains the documentation for this element from the ***SORTOIDOC*** schema.

Note that where there are only a few options, and the element is not mandatory, the default will appear first in the documentation within braces. **Figure 3-38** shows the documentation for the *IncludeTransactions* element of the customer query business object ***ARSQRY***. Because the Y appears first within the braces, this is what will be used if the element is not included in the XML. In **Figure 3-37** the *OrderActionType* field was selected. Because this field is mandatory there is no default, and the valid options do not appear in braces.

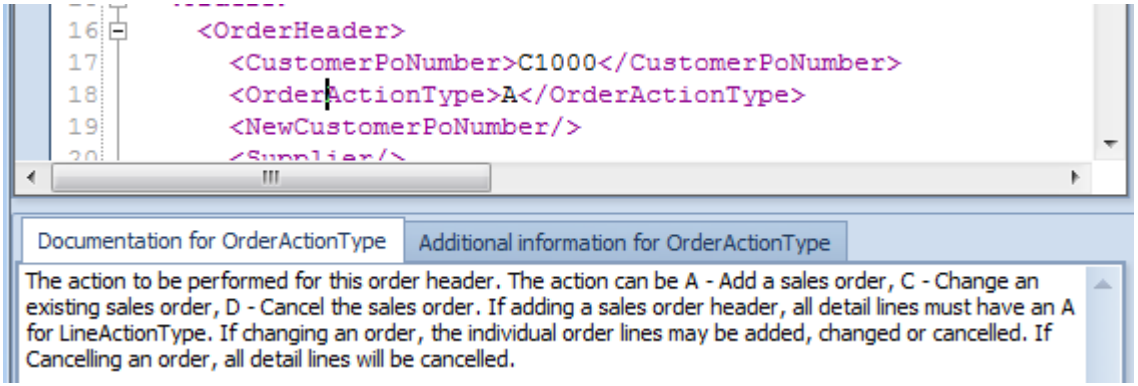


Figure 3-37: Using the *Documentation* tab

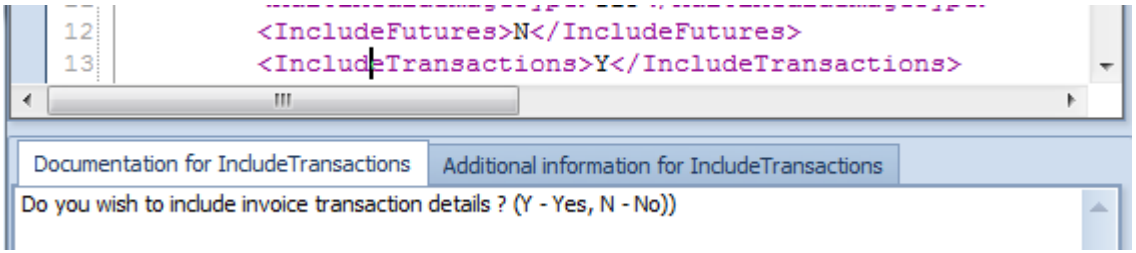


Figure 3-38: The default option (that would be used if the element is not present) appears first

Additional Information Tab

The *Additional information* tab contains additional information about the selected element, such as whether it is mandatory whether whitespace must be preserved, the default value, etc.

The Status Bar

The *Status Bar* appears at the bottom of the e.net Diagnostics screen. It will contain different information depending on the task being performed, and those that have already been performed. **Figure 3-39** shows the left hand side of the status bar. This displays the currently logged in operator code, the company to which they logged in, and the instance of e.net Solutions that they are using. If a business object has been invoked during this run of e.net Diagnostics, the name if the last business object to be invoked will also appear.

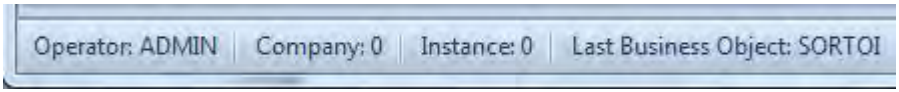


Figure 3-39: The left hand side of the status bar

Figure 3-40 shows the right-hand side of the status bar. If a business object has been invoked during this run of e.net Diagnostics, the time taken for the business object to execute will be displayed. In this case the value is ten one hundredths of a second (or 0.1 of a second). The *Col*, *Row*, and *Pos* values show the column, row, and character position of the cursor within the tab. The *Schema Information* button shows/hides the *Documentation* and *Additional information* tabs. Clicking on this button will toggle between them. The final piece of information shows the currently selected communication method.

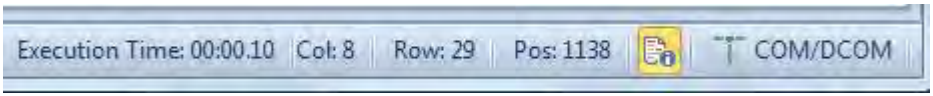


Figure 3-40: The right hand side of the status bar

Licensing Business Objects for use with e.net Diagnostics

As e.net Diagnostics (both the full and express versions) call the business objects from outside of SYSPRO, the functional area to which the business object belongs must be licensed for this operator to be able to invoke the business object. See the *Licensing* section of Chapter 2 for more information.

Figure 3-41 shows an example of the error that will be returned when the operator is not licensed to use the functional area to which the business object belongs. In this case the operator is trying to invoke the customer query business object (***ARSQRY***) and is not licensed to use the *Accounts Receivable Primary Query* functional area to which it belongs.

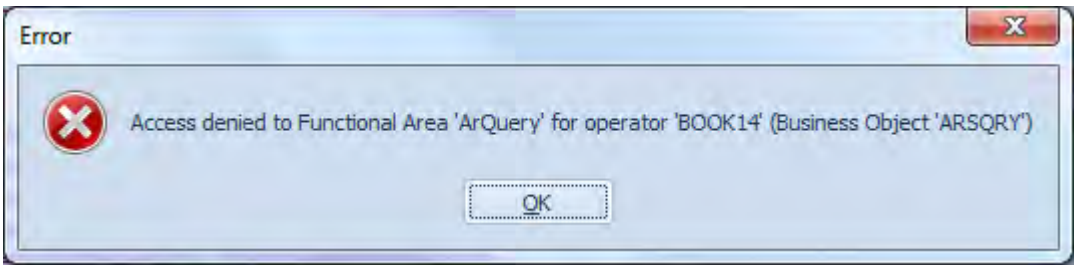


Figure 3-41: Operator code *BOOK14* is not licensed to use the *AR Primary Query* functional area

Chapter 4 - SYSPRO Program Components

Let's take a quick look at the user interface of a typical SYSPRO program. If you have read the first book called *Power Tailoring – A step by step guide*, this will be a refresher for you as it was covered in more detail in that book's Chapter 1. If you haven't, it will help define some of the components for you. The beginning of this chapter contains only an overview of what is available in the first chapter of the first book, so it is worth taking the time to go back and read it, if you can.

Main Window Components

Window
When referring to a window in SYSPRO, we are referring to the whole screen for that program. For example, the *Sales Order Entry* window contains a number of forms and listviews, but the screen that embraces these is referred to as the *window*. **Figure 4-1** shows the *Sales Order Entry* window.

Pane
A pane is a one of the components of a window, and can be a form, a listview, a data grid, customized pane, etc. In **Figure 4-1**, *Order Header*, *Customer Information*, *Order Totals*, *Entered Order Lines*, *Order Line Details*, *Stocked Line*, *Stocked Line Information* and *Stock Code Information* are all panes.

Forms
Forms are panes that are used to display and capture data. They comprise two columns - one containing the caption, and one containing the value. There are two types of forms - a display form, and an entry form.

Display Form
A display form, as the name suggests, is used to display information; you cannot add or maintain data within them. In **Figure 4-1**, *Customer Information*, *Order Totals*, *Order Line Details* and *Stock Code Information* are all display forms.

Entry Form
An entry forms allows the display, entry, and maintenance of data. Although entry forms allow the entry and maintenance of data, some fields cannot be changed. The captions for these will be greyed-out. In **Figure 4-1**, *Order Header*, *Stocked Line* and *Stocked Line Information* are all entry forms. In the *Order Header* form, the *Sales order* field is greyed-out and cannot be changed (because this company is using automatic sales order numbering).

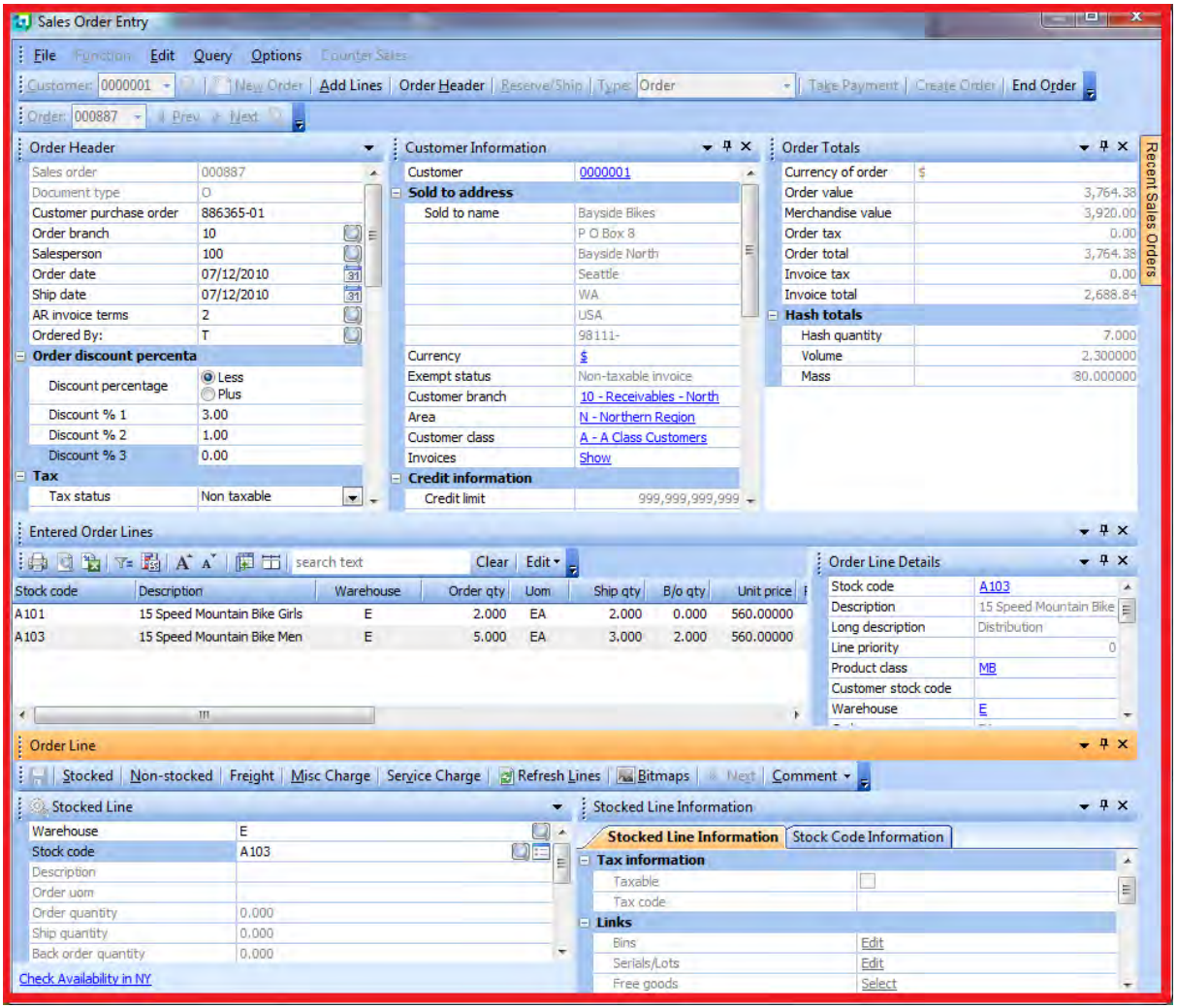


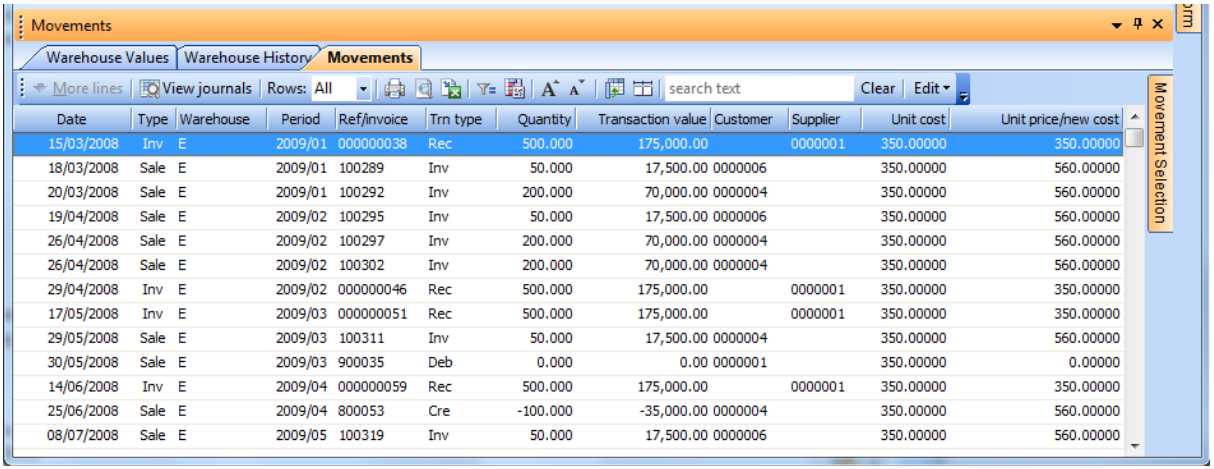
Figure 4-1: The *Sales Order Entry* window

Listview

A listview consists of cells in rows and columns, and is used for displaying data only. It appears similar to a spreadsheet. **Figure 4-2** shows the *Movements* listview from the *Inventory Query*.

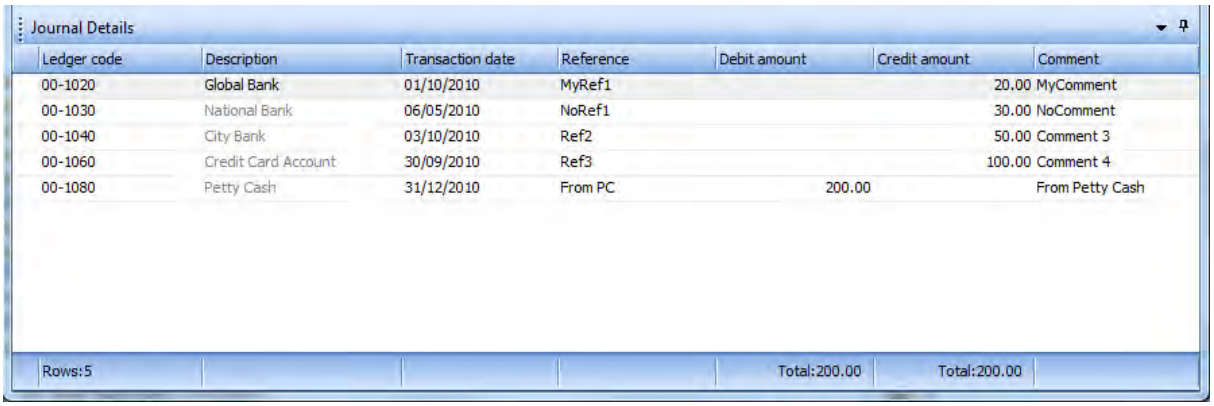
Data Grid

A Data grid is similar to a listview in that it has cells arranged in rows and columns. It differs from a listview in that you can use it to add/maintain data. **Figure 4-3** shows the *Journal Details* data grid from the *General Ledger Journal Entries* program.



Date	Type	Warehouse	Period	Ref/invoice	Trn type	Quantity	Transaction value	Customer	Supplier	Unit cost	Unit price/new cost
15/03/2008	Inv	E	2009/01	000000038	Rec	500.000	175,000.00		0000001	350.00000	350.00000
18/03/2008	Sale	E	2009/01	100289	Inv	50.000	17,500.00	0000006		350.00000	560.00000
20/03/2008	Sale	E	2009/01	100292	Inv	200.000	70,000.00	0000004		350.00000	560.00000
19/04/2008	Sale	E	2009/02	100295	Inv	50.000	17,500.00	0000006		350.00000	560.00000
26/04/2008	Sale	E	2009/02	100297	Inv	200.000	70,000.00	0000004		350.00000	560.00000
26/04/2008	Sale	E	2009/02	100302	Inv	200.000	70,000.00	0000004		350.00000	560.00000
29/04/2008	Inv	E	2009/02	000000046	Rec	500.000	175,000.00		0000001	350.00000	350.00000
17/05/2008	Inv	E	2009/03	000000051	Rec	500.000	175,000.00		0000001	350.00000	350.00000
29/05/2008	Sale	E	2009/03	100311	Inv	50.000	17,500.00	0000004		350.00000	560.00000
30/05/2008	Sale	E	2009/03	900035	Deb	0.000	0.00	0000001		350.00000	0.00000
14/06/2008	Inv	E	2009/04	000000059	Rec	500.000	175,000.00		0000001	350.00000	350.00000
25/06/2008	Sale	E	2009/04	800053	Cre	-100.000	-35,000.00	0000004		350.00000	560.00000
08/07/2008	Sale	E	2009/05	100319	Inv	50.000	17,500.00	0000006		350.00000	560.00000

Figure 4-2: The *Movements* listview in the *Inventory Query*



Ledger code	Description	Transaction date	Reference	Debit amount	Credit amount	Comment
00-1020	Global Bank	01/10/2010	MyRef1		20.00	MyComment
00-1030	National Bank	06/05/2010	NoRef1		30.00	NoComment
00-1040	City Bank	03/10/2010	Ref2		50.00	Comment 3
00-1060	Credit Card Account	30/09/2010	Ref3		100.00	Comment 4
00-1080	Petty Cash	31/12/2010	From PC	200.00		From Petty Cash

Figure 4-3: The *Journal Details* data grid in the *General Ledger Journal Entries* program.

Customized Pane

A customized pane is a user-defined pane which can be configured as one of several types. These are *Graph*, *Listview*, *Web browser*, *SRS/Crystal report*, *PDF viewer*, *Rich text notepad*, *.NET User Control*, *Form*, *Executive dashboard* or *Search window*. Depending on the type selected, you can use a combination of settings, VBScript, SWF files, and business objects to display/add/maintain the required data.

Custom Form

A custom form provides a mechanism to add extra data fields which are linked to certain master and transaction tables. A custom form is one that can be included in the relevant take-on or transaction program, to which the fields are added. These custom form fields can be alphanumeric, numeric or

date fields, and validation options can be set against them. Once created as custom form fields, these fields can be embedded in display and entry forms. **Figure 4-4** shows a custom form added to the inventory master table.

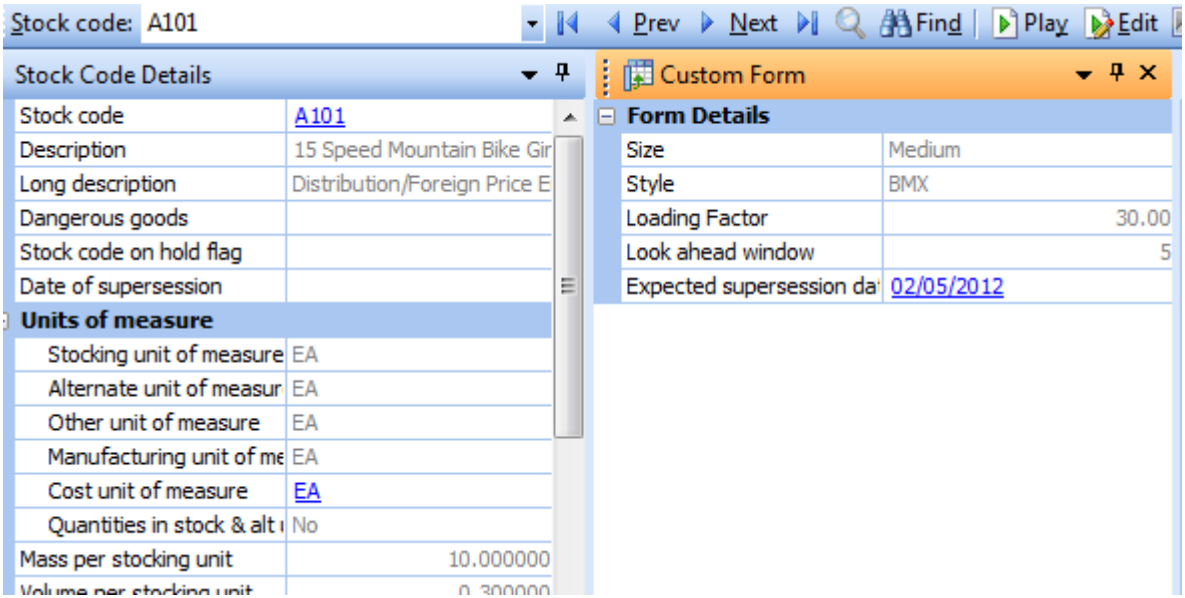


Figure 4-4: A custom form added to the inventory master table

Associated Pane

An associated pane is a pre-written customized pane that is specifically associated with a SYSPRO key field (i.e. a customized pane template that has been written for a specific SYSPRO key field such as stock code, customer, requisition, etc.). The associated pane can be a listview, a form, a graph, or any other format that is supported for a customized pane.

An associated pane differs from a customized pane in that when the value of the field with which it is associated changes, the associated pane automatically refreshes to match. **Figure 4-5** shows a listview associated pane that has been added to *Sales Order Entry*. It is associated with the *Customer* field and will reflect all sales orders for the current customer account code. If the customer code changes, the contents of the associated pane changes accordingly. However, if the next sales order is for the same customer code, the pane does not need to refresh, so will not.

Search Window

A search window is a special type of customized pane that contains two panes. One pane is used for the selection criteria, the other is used to display the results of the search. The search windows most commonly required have been supplied with SYSPRO and can be configured to be used instead of a

standard browse. You can also create your own customized pane search windows. **Figure 4-6** shows the supplied pre-defined *Stock Code Search*. The *Search statements* can be saved for later reuse.

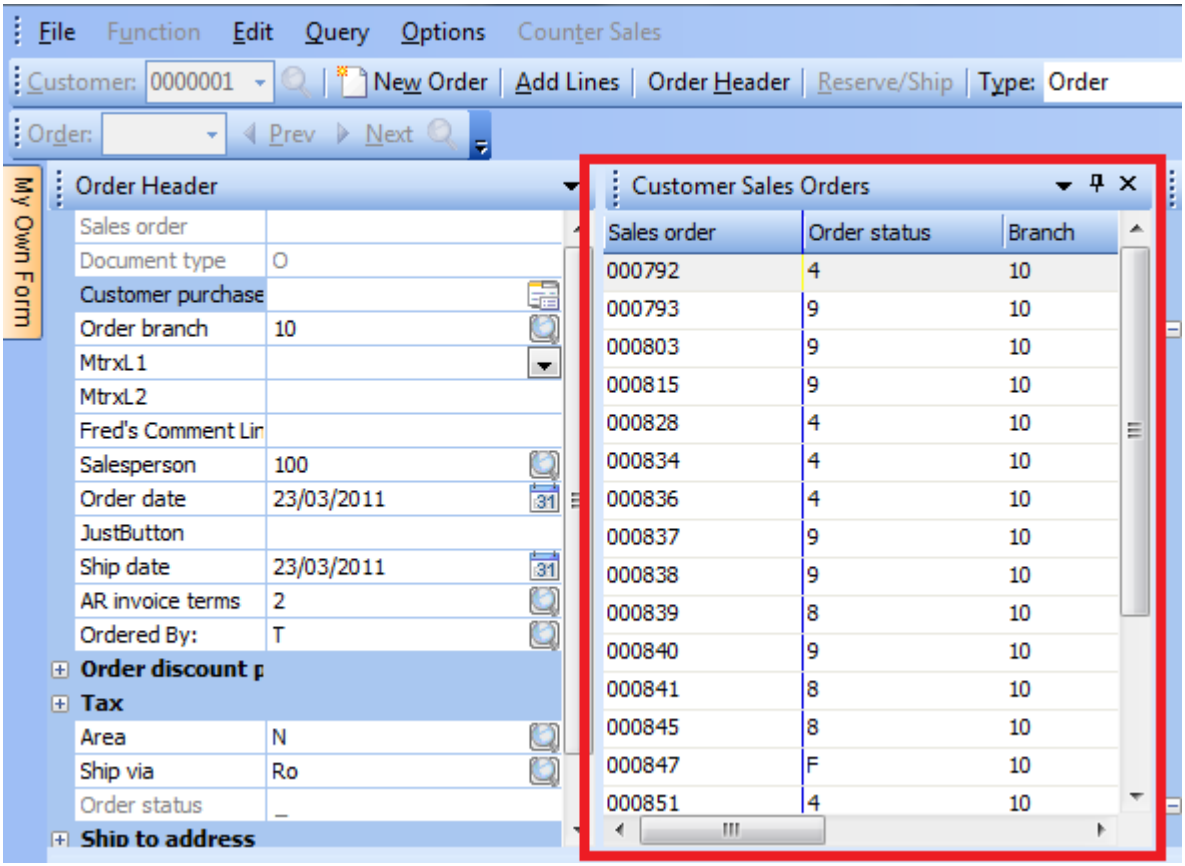


Figure 4-5: A listview *Associated Pane* that has been added to *Sales Order Entry*

Application Builder

The application builder enables you to build up to 20 sandboxed applications within SYSPRO. Each of these applications can contain one or more customized panes. As these applications are launched from within SYSPRO they can access e.net Solutions business objects without requiring additional licenses. Although each of these applications is sandboxed, if one contains multiple customized panes, these panes can talk to each other within the application.

The application builder program descriptions default *to Application x (IMPDHx)*, where *x* is replaced by the application number in the range 1 to 20. When you define the application you are able to change this description to something that is relevant to the application. **Figure 4-7** shows the list of

applications in a dropdown (that is accessed from the *Administration* tab of the ribbon bar) where the first application's description has been changed from *Application 1 (IMPDH1)* to *My First Application*.

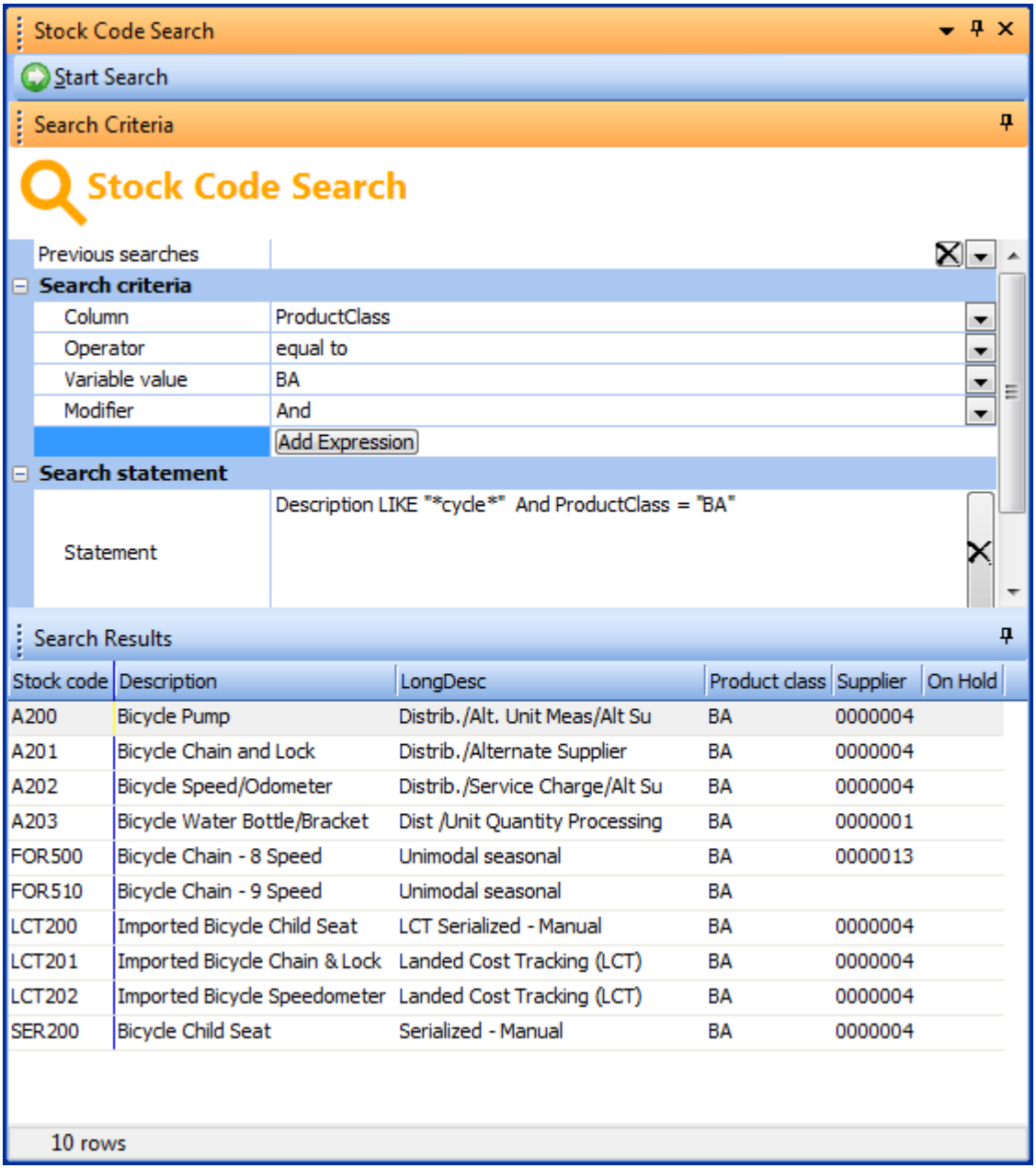


Figure 4-6: The supplied pre-defined *Stock Code Search*

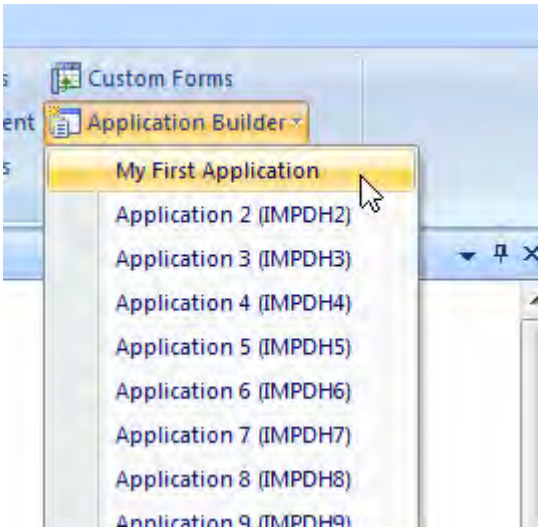


Figure 4-7: Launching one of the application builder applications

Form Components

Field
Fields appear on display forms and entry forms. A field comprises of a caption and its matching value (if a value currently exists for this field). In **Figure 4-8**, the label of *Warehouse* and its matching value of *E* make up the *Warehouse* field.

Caption
A caption is the label assigned to an item of data, or value. A caption, together with its value, forms part of a field. In **Figure 4-8**, *Warehouse* and *Stock code* are both captions.

Value
A value appears against a caption, making up the field. It is either blank, or contains data being displayed/input.

Display-Only Fields
On an entry form, some fields can be used to enter or modify data. Others appear greyed-out as they are only present to enhance the information displayed, or because the option is not relevant in the current context. In **Figure 4-8** the *Order uom* field is greyed-out as it cannot be changed, and the *Bin location* field is greyed-out because it is not relevant as the inventory control option for multiple bins is not enabled for this company (normally you would remove this field from the form if you do not use multiple bins).

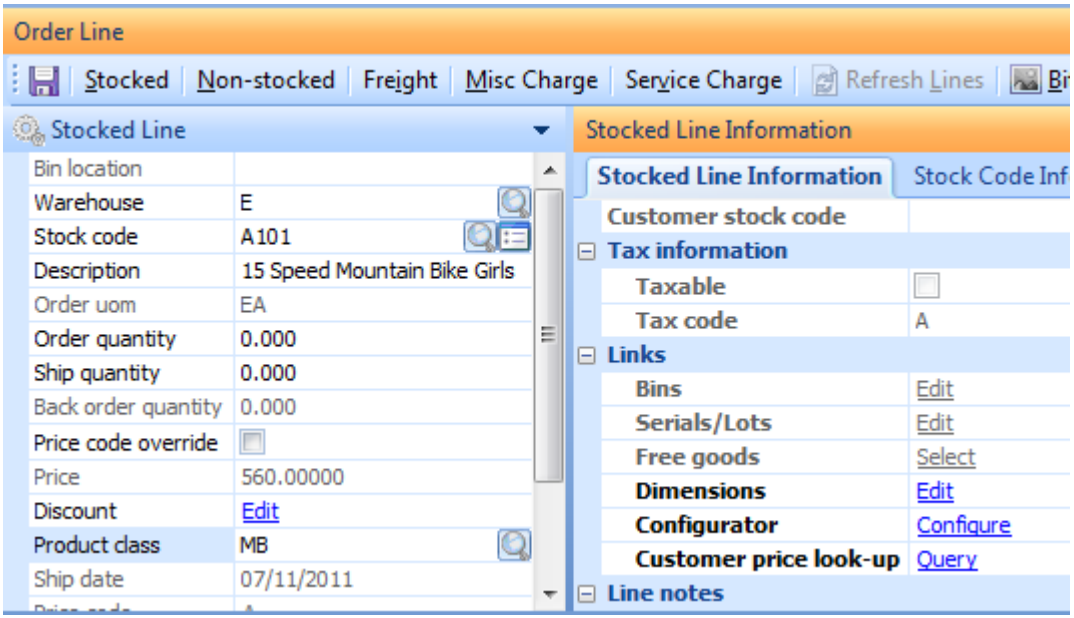


Figure 4-8: Form components showing *Fields*, *Captions*, *Values*, *Groups*, and *Display-Only Fields*.

Groups

Fields on a form can be grouped together, and these groups can be expanded to show all the fields, or contracted so that only the group heading appears. In **Figure 4-8**, *Tax information, Links* and *Line notes* are all groups.

Form Action

A form action is a hyperlinked option that is added to the bottom of a form. It can be configured to call another SYSPRO program or to call VBScript code (which could call a third party application). **Figure 4-9** shows the form action *Check Supplier Availability*.

Field Selector

The *Field Selector* screen allows you to see the fields that are available to be dragged onto a form. The dropdown list at the top allows you to select from the available types of fields (custom form fields associated with this form, fields from the master file associated with this form, or scripted fields that you have created). Not all field types are available for every form. For example, if this form has no association with custom forms, this option will not appear in the list.

Fields can be dragged from the field selector onto a form, and then the field name appears in the field selector in italics. **Figure 4-10** shows the field selector displaying the available custom form fields for a form associated with the customer master file.

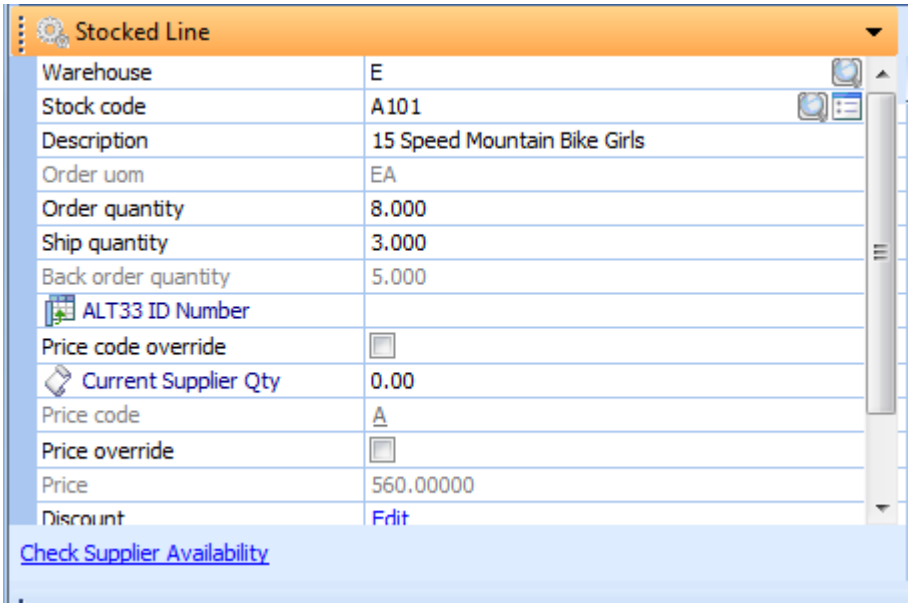


Figure 4-9: The *Check Supplier Availability* form action

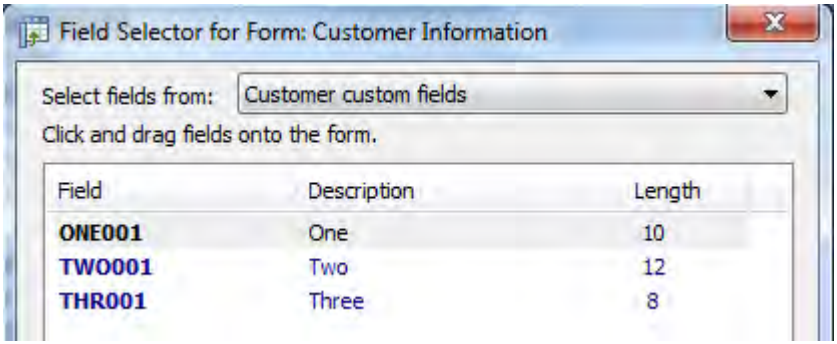


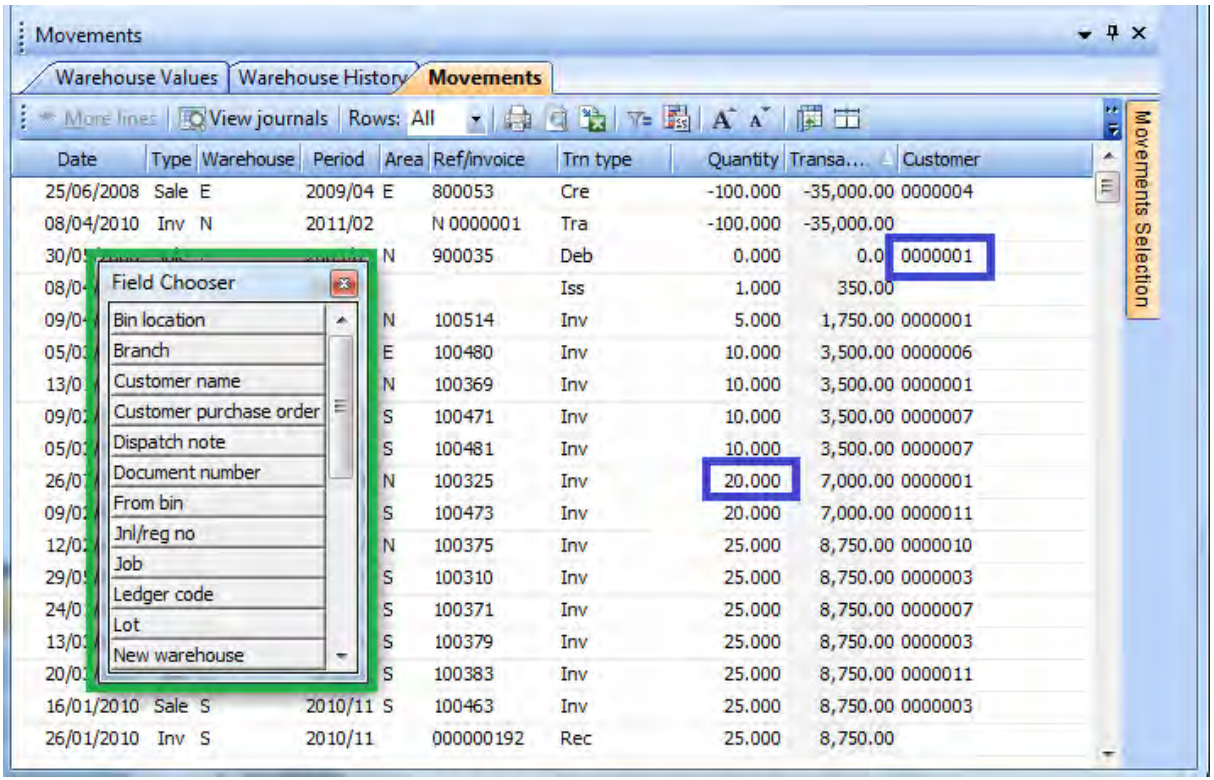
Figure 4-10: *The Field Selector* showing the *Customer custom fields*

Custom Form Fields

A custom form provides a mechanism of adding additional fields to certain master tables and transactions, through the master file take-on and transaction programs. Each of these fields can be defined as alphanumeric, numeric or date. Defaults and validation criteria can be set, and fields can be defined as mandatory. The individual fields can also be dragged onto forms that are related to this master table.

Scripted Fields
A scripted field is a container that can be placed on a form (both display and entry), and is available to be modified using VBScript. During its creation you specify the type of field (e.g. alphanumeric, numeric or a date) as well as its size. Scripted fields are global. Once they have been created they are available for use on all forms, although their contents are only available within the current program.

Cells
A listview is made up of many cells. Two of these cells have been highlighted in **Figure 4-11**, and contain the values *0000001*, and *20.000*.



Date	Type	Warehouse	Period	Area	Ref/invoice	Trn type	Quantity	Transa...	Customer
25/06/2008	Sale	E	2009/04	E	800053	Cre	-100.000	-35,000.00	0000004
08/04/2010	Inv	N	2011/02		N 0000001	Tra	-100.000	-35,000.00	
30/0				N	900035	Deb	0.000	0.0	0000001
08/0						Iss	1.000	350.00	
09/0				N	100514	Inv	5.000	1,750.00	0000001
05/0				E	100480	Inv	10.000	3,500.00	0000006
13/0				N	100369	Inv	10.000	3,500.00	0000001
09/0				S	100471	Inv	10.000	3,500.00	0000007
05/0				S	100481	Inv	10.000	3,500.00	0000007
26/0				N	100325	Inv	20.000	7,000.00	0000001
09/0				S	100473	Inv	20.000	7,000.00	0000011
12/0				N	100375	Inv	25.000	8,750.00	0000010
29/0				S	100310	Inv	25.000	8,750.00	0000003
24/0				S	100371	Inv	25.000	8,750.00	0000007
13/0				S	100379	Inv	25.000	8,750.00	0000003
20/0				S	100383	Inv	25.000	8,750.00	0000011
16/01/2010	Sale	S	2010/11	S	100463	Inv	25.000	8,750.00	0000003
26/01/2010	Inv	S	2010/11		000000192	Rec	25.000	8,750.00	

Figure 4-11: A listview with two cells and the *Field Chooser* highlighted

Field Chooser
A listview and data grid can be specified to only display some of the available columns by default. It is also possible for an operator to remove columns they do not want displayed, by dragging the column heading from the listview/data grid.

If the removed columns need to be put back, or columns need to be displayed that do not show by default, this is done using the field chooser. The field chooser shows a list of all the available column

headings that are not currently displayed in the listview/data grid. These fields can be dragged onto the column heading, and the column will be added to the listview/data grid. The next time that data is to be displayed, the added column will be populated. **Figure 4-11** shows the field chooser highlighted (the largest of the three highlighted items).

Chapter 5 - Macro Events

Introduction

A *Macro* is a sequence of commands that can be processed, and they will always be processed in this listed sequence. In SYSPRO, macros are written using VBScript (or Microsoft Visual Basic Scripting Edition, to use its full name).

Events are triggers built into SYSPRO programs that fire when the program is performing specific tasks, for example, when the content of a form is refreshed. When these events fire they run a macro that is linked to this event if one exists, hence the name *Macro Events*.

Every form, data grid, listview, and customized pane has one or more events associated with it, and every field on a form has its own list of events. Each of these program components (form, listview, etc.) can have their own VBScript file, and this VBScript will contain *Functions*. These functions are linked to the events mentioned above.

Using the *AR Customer Query* as an example, the program's name is ***ARSPEN***, and its *Customer Information* form is ***ARSPENL6***. This form can have two events against it (*OnLoad* and *OnRefresh*) and each field has its own *OnMenuSelect* event. If code is added to any of these events the ***ARSPENL6*** VBScript is created, and the function name associated with this event is created within the script.

When an operator enters a customer account code in the *Customer Query* program, the contents of the *Customer Information* form is refreshed with the details for the new customer. As the contents of this form are refreshed the form's *OnRefresh* event fires. If the *CustomerInformation_OnRefresh* function exists in the ***ARSPENL6*** VBScript, the lines of code against this function are run.

Macro events are available for the following standard program components, and each is covered in detail below:

- Forms (called Form events)
- Fields on forms (called Field events)
- Listviews
- Data grids
- When you Login/Logout
- Form Actions
- Toolbars

Form Events

As a program loads that contains a form, the empty form is displayed. At this point no macro events have fired. The first time that the form is going to be populated with data (for example, when you enter a customer code in the *AR Customer Query*) the form's *OnLoad* event is fired. This is followed by its *OnRefresh* event as the data is populated. The *OnLoad* event for a form only fires once for each run of the program. The *OnRefresh* event for the form fires each time that the contents of the whole form must be refreshed. Using the same *AR Customer Query* example as above, the first time that the customer code is entered the *OnLoad* event fires, followed immediately by the *OnRefresh* event. If a second customer code is entered, only the *OnRefresh* event is fired. When an event fires, if VBScript code exists within the *Function* associated with this event, the code is run.

Where a program contains more than one form, each of the forms will go through the same process. When the form's *OnLoad* event fires, its *OnRefresh* event will fire straight after. So form A's *OnLoad* and *OnRefresh* events will fire followed by form B's *OnLoad* and *OnRefresh* events, etc.

At the top of each form is its titlebar, which contains the form's description. No two forms within a program can have the same description. **Figure 5-1** shows the *AR Customer Query* screen where two forms can be seen, *Customer Information*, and *Contact Details*.

When a form is unpinned by clicking on the *Auto Hide* icon, it will appear as a tab. The tabs of two other forms can be seen, and their names appear on their tabs (*Customer History* and *Extra Customer Details*: one at the top and one on the right). The form events for these forms fire in exactly the same way as forms that are displayed.

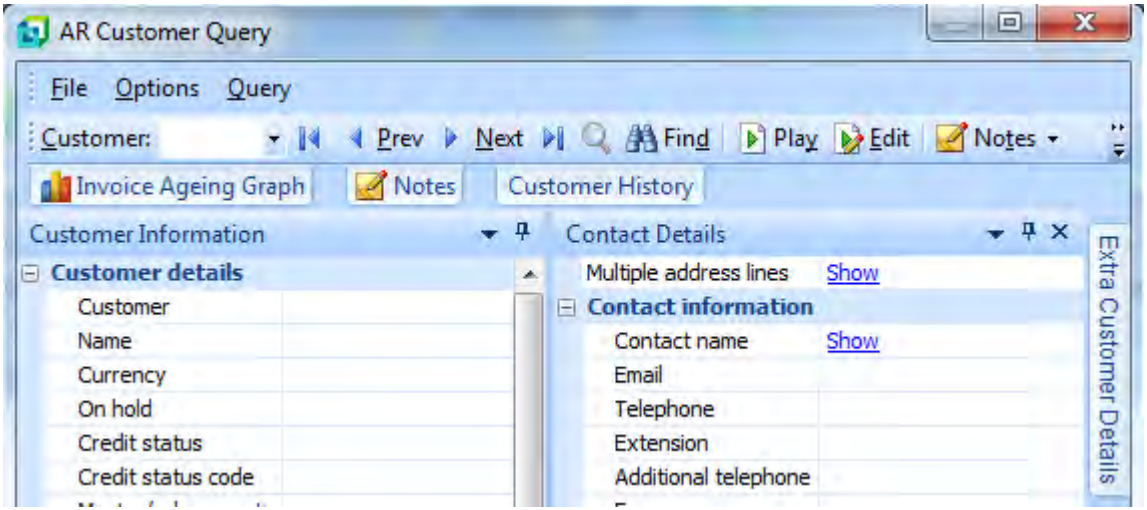


Figure 5-1: The *AR Customer Query* showing multiple display forms

If a form has been deleted from a program its behaviour is specific to the program from which it was deleted. Typically for a query program the events will not fire.

Each form event is associated with a function contained in a VBScript, and the function name is made up of the name of the form and the type of event. In the case of the *Customer Information* form, its *OnLoad* event is *CustomerInformation_OnLoad*, and its *OnRefresh* event is *CustomerInformation_OnRefresh*.

Adding/Editing a Form Event

The VBScript functions are added to a form by either right-clicking anywhere on the form and selecting the *Macro for:* option from the displayed menu (see **Figure 5-2**), or using the *Alt+F8* shortcut keys.

The eight character name that appears alongside *Macro for:* in this menu is the program's name for this form. This is made up of the six character program name (as opposed to its description) and two characters that are specific to this form. In the example in **Figure 5-2**, the name against the *Macro for:* entry is ***ARSPENL6***. Where ***ARSPEN*** is the program name for *AR Customer Query*, and ***L6*** refers to its *Customer Information* form.

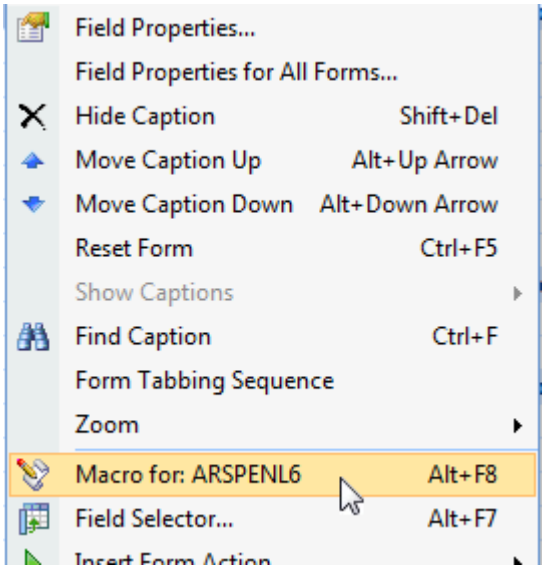


Figure 5-2: The context-sensitive menu that is displayed when you right-click on a form

When the *Macro for:* option is selected (or the *Alt+F8* shortcut keys used) the *VBScript Editor* screen is displayed. **Figure 5-3** shows the *VBScript Editor* screen, which is made up of a toolbar, *Options* section, and *Available Events* section. The toolbar contains buttons to save your changes, cancel out without saving your changes, and to edit the VBScript. In this figure it can be seen that the operator

that is adding the form/field event is not doing so for a role, as the *Current role* field within the *Options* section is blank.

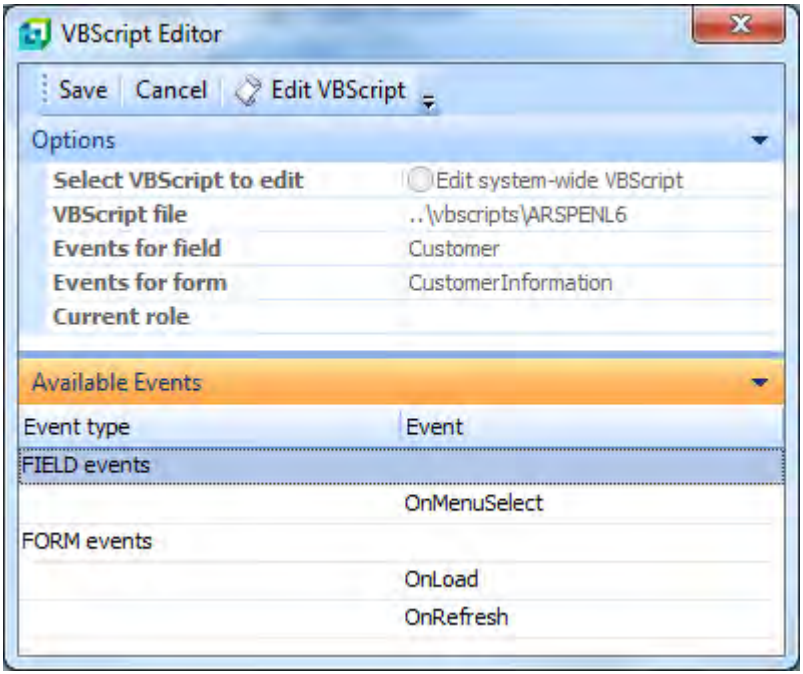


Figure 5-3: The *VBScript Editor* screen

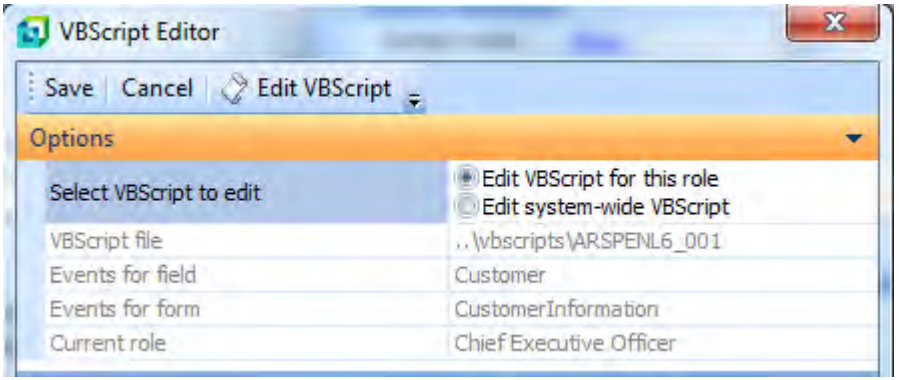


Figure 5-4: The *VBScript Editor* when in design role layout mode

The *Options* section of the *VBScript Editor* screen displays information such as the field on which focus was set when the editor was invoked, the name of the form, and the name of the script being

edited. If the form is being edited for a role (and you are in design role layout mode) the role description is displayed, along with radio buttons to select whether to edit the system-wide script, or the one specific to this role. **Figure 5-4** shows the *Options* section of the *VBScript Editor* when in design role layout mode, containing both the option to select the script to be edited, and the role description.

The *Available Events* section contains two event types, those related to *Field events* and those related to *Form events*. The form events for a display form appear at the bottom of **Figure 5-3**. The only two form events for display forms are *OnLoad* and *OnRefresh*. An entry form has these two events plus *OnStartEdit*, *OnStopEdit*, *OnSubmit*. In addition, some entry forms have an *OnAfterSubmit* event, where it was decided that this would be of benefit. These events can be seen in **Figure 5-5** which shows the form events for the *Stocked Line* entry form in the *Sales Order Entry* program, which is one of the entry forms where the *OnAfterSubmit* event is implemented.

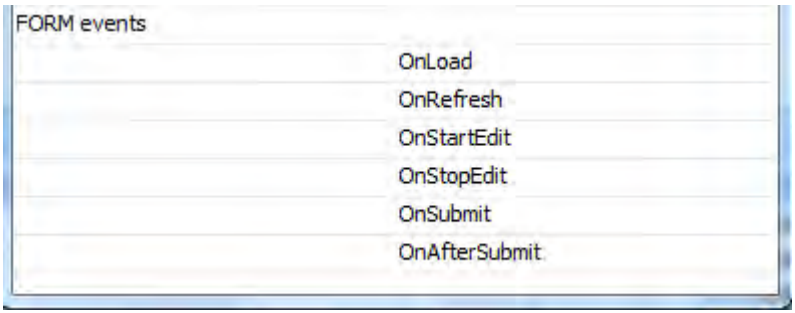


Figure 5-5: The available form events for an entry form

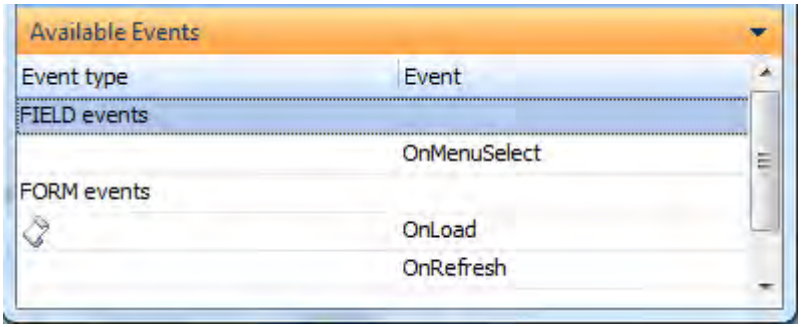


Figure 5-6: The icon shows that an event already exists

If a function is already present in the VBScript for an event, a script icon will appear against the event name. **Figure 5-6** shows a section of the same *VBScript Editor* screen as in **Figure 5-3**, but this time its *OnLoad* function is present in the VBScript, so the *OnLoad* event has a script icon against it.

To add a new function/modify an existing function against an event, either highlight the event and click on the *Edit VBScript* button on the toolbar, or double-click on the event name. A screen will be displayed where the script is created (or modified, if one already exists). Against the titlebar of the screen is *VBScript for:* followed by the description of the form that is being edited. If the form's name includes space characters, these will be stripped out. **Figure 5-7** shows the screen that is displayed when adding a function to the *Customer Information* form of *the AR Customer Query* program. As the *CustomerInformation_OnLoad* function did not already exist, it was created and the cursor placed between the start of the function and the end of it.

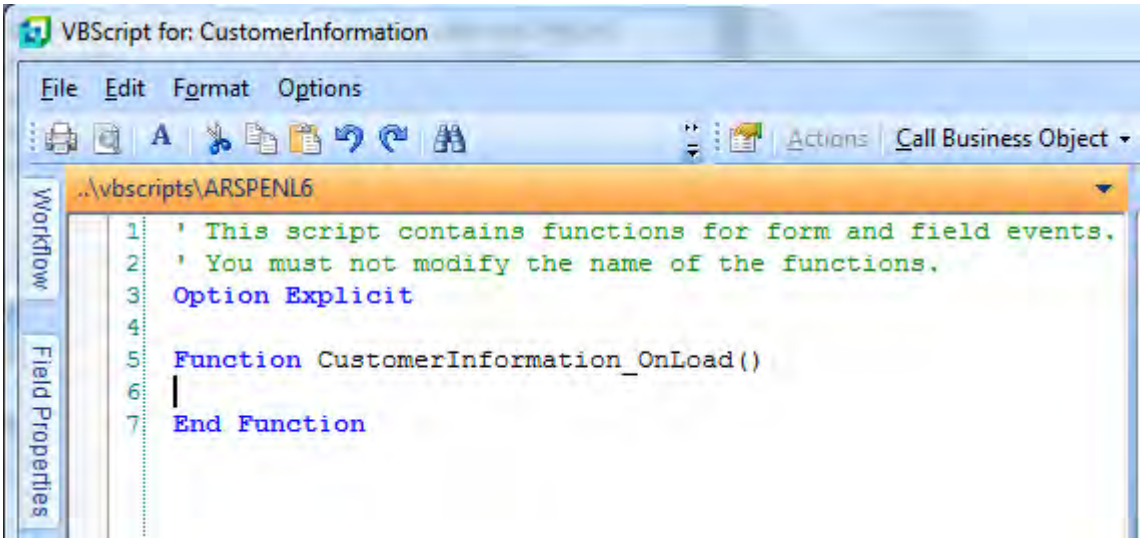


Figure 5-7: The *CustomerInformation_OnLoad* function being created

The simplest way of seeing when an event fires is to cause a message box to be displayed when it does fire. This is done by adding the following text between the start and end of the function, then copying the function name between the two double-quote signs (excluding the parentheses).

```
msgbox ""
```

For the example in **Figure 5-7** the message box command would be:

```
msgbox "CustomerInformation_OnLoad"
```

Once this has been done your script should appear similar to that shown in **Figure 5-8**.

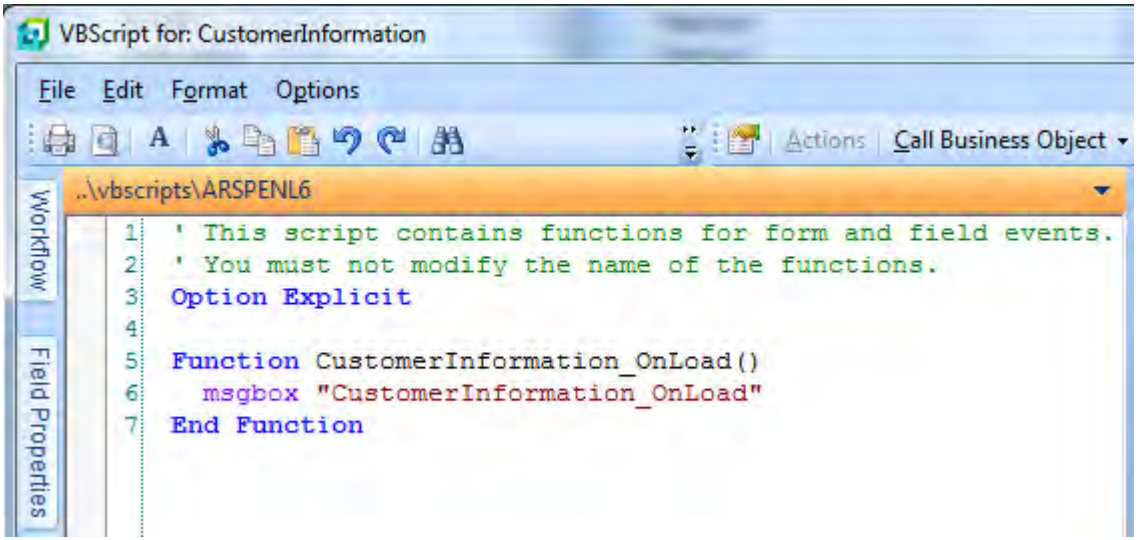


Figure 5-8: Adding a message box statement to a function

Exit this screen by using either *File* | *Exit* from the toolbar, or clicking on the big *X* at the top right of the screen. You will be taken back to the *VBScript Editor* screen, and the *OnLoad* event will now have a script icon against it. Now add a message box to the *OnRefresh* event in the same way as you did for the *OnLoad* event. Start by double-clicking on the event name, adding the *msgbox* statement, and copying the function name. **Figure 5-9** shows how the screen will appear with both the *OnLoad* and *OnRefresh* events containing their message box statements.

Exit back to the *VBScript Editor* screen and click on the *Save* button on the toolbar. You will be taken back to the *AR Customer Query* program. To be able to see the *OnLoad* event fire you should close this program and run it again. As explained above, when the program initially loads, no events are fired. When you enter a customer code and tab off this field the *OnLoad* event for the *Customer Information* form fires, which causes the *CustomerInformation_OnLoad* function to be invoked. When this is invoked the script against this function is run, which in this case contains the line to display the message box (see **Figure 5-10**).

The script will wait at this point until the operator acknowledges the message box, after which it will continue. When this function has finished running its script the next event fires, which is the *OnRefresh* event for the *Customer Information* form. This invokes the *CustomerInformation_OnRefresh* function, which runs its script containing the message box statement, and the message box is displayed (see **Figure 5-11**). The script waits for the operator to acknowledge this message box before continuing. When all of the events with functions against them have fired, focus is set on the primary form for this program and the operator is allowed to continue using the program.

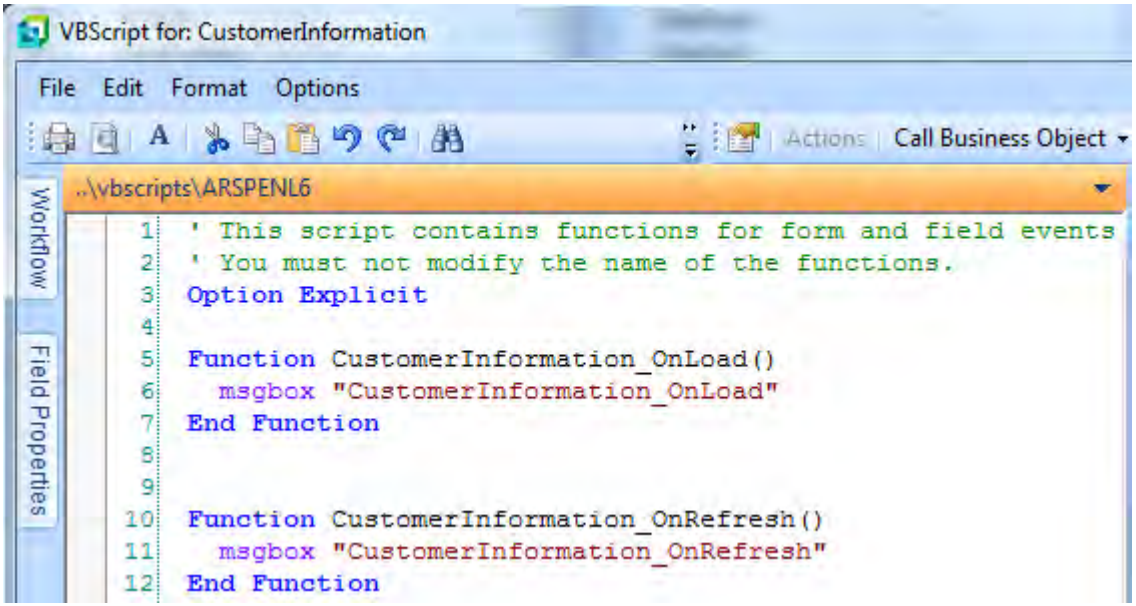


Figure 5-9: Both the *OnLoad* and *OnRefresh* events with message box statements against them

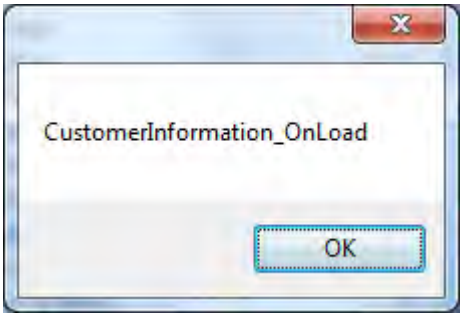


Figure 5-10: The message box displayed when the *CustomerInformation_OnLoad* event fires

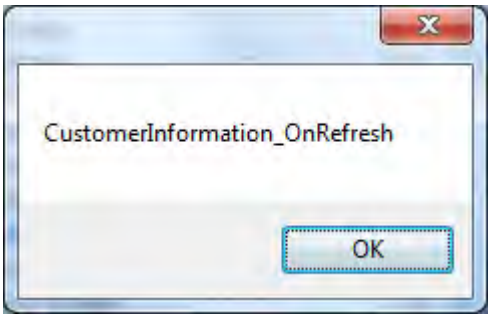


Figure 5-11: The message box displayed when the *CustomerInformation_OnRefresh* event fires

There are some exceptions to these rules. Some programs that contain forms that have no requirement to be refreshed fire both their *OnLoad* and *OnRefresh* events as the program loads. An example is the *AR Period End* program (most period end programs work this way) where the *Information* form contains all the selection criteria, and the *Report* listview that can contain the results after the program has run. This *Information* form's *OnLoad* and *OnRefresh* events fire as the program loads.

Another exception is when a program has a customized pane associated with it. When the program loads, the customized pane's *OnLoad* event is fired. Some of the predefined customized panes (known as templates) contain code within their *OnLoad* function to fire the *OnRefresh* function so that data is populated immediately. Although this is not strictly an exception to the rule, it is something you should be aware of.

On the topic of customized panes, there is an option that affects how the *OnLoad* event fires. When creating/editing a customized pane, within the *Refresh details* section of the *Pane Properties* form there is the *Ignore OnLoad VBScript* option. If this is checked, code against the customized pane's *OnLoad* function is ignored (see **Figure 5-12**).

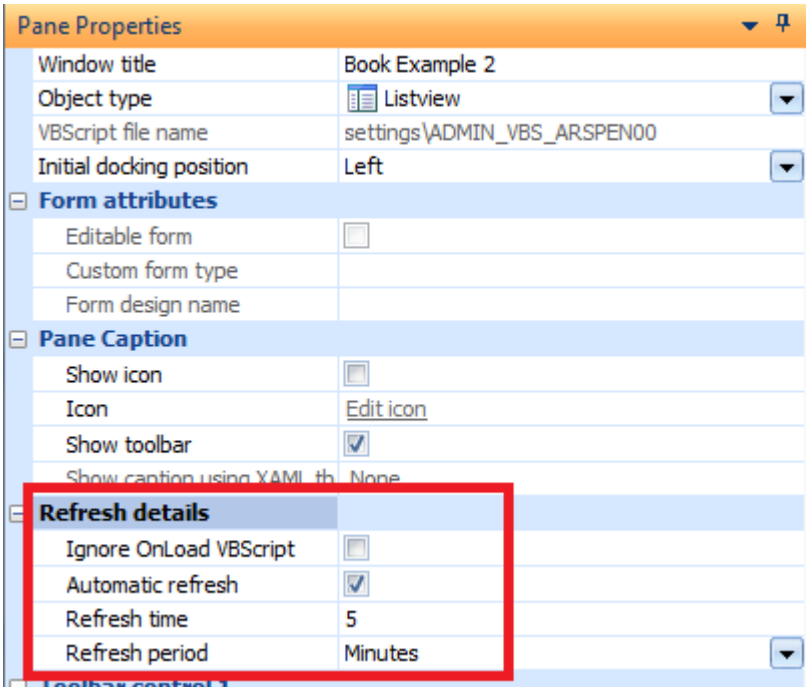


Figure 5-12: The *Refresh details* section of the *Customized Pane* properties

Also within the *Refresh details* section is an option to automatically fire the *OnRefresh* event. If this is checked the options to specify the *Refresh time* (and whether this is minutes or seconds) becomes available.

When do Form Events Fire ?

Display Forms have two form events, *OnLoad* and *OnRefresh*. All *Entry Forms* have at least five form events, *OnLoad*, *OnRefresh*, *OnStartEdit*, *OnStopEdit*, *OnSubmit*, and some also implement an *OnAfterSubmit* event.

OnLoad
The *OnLoad* event for a form fires only once for each run of the program. It fires the first time that the form is going to be populated with data, not as the program loads. An *OnLoad* event is typically used to invoke things that must remain for the whole run of the program, or to set an initial state. For example, if a field on the form needs to have a button against it, the definition of this button would appear against the form's *OnLoad* event. If the initial state of a field on an entry form must be read-only, this would also be set against the *OnLoad* event.

OnRefresh
The *OnRefresh* event for the form fires each time that the contents of the whole form must be refreshed. The first time that a form is going to be used, its *OnLoad* event fires followed immediately by its *OnRefresh* event. Whenever the program needs to refresh the contents of this form the *OnRefresh* event will fire. An example of this is when adding a stock detail line to a sales order. When you have completed adding the order header information and select to add detail lines, the *OnLoad* event for the *Stocked Line* form will fire (if this is the first time that this form has been used for this sales order) and this is followed immediately by its *OnRefresh* event. When you enter the stock code and tab off of this field, SYSPRO retrieves the details for this stock code/warehouse combination, and once it has populated the form it fires the *OnRefresh* event again.

The *OnRefresh* event is typically used for changing the state of a field, or causing a change on another form. For example, if you wanted to change the background color of a field to highlight that its value fell outside of a target range, you could do this when the form refreshes (and just as importantly, undo the change when the value falls back within the range).

Other events can also be used to fire a form's *OnRefresh* event; it doesn't have to be fired by the program. If you have created a *Customized Pane* to display some values, you could use the *OnRefresh* event of one of the standard forms of the program to fire the customized pane's *OnRefresh* event.

OnStartEdit
The *OnStartEdit* event is invoked as you start editing the first field on a form for the first time, or when the program has placed the cursor on the form for the first time. This will always be after the form has

fired its *OnLoad* and *OnRefresh* events, but not necessarily immediately after these have fired. This could be because other forms still have to fire their *OnLoad* and *OnRefresh* events first, or because focus is not automatically set on this form after its *OnLoad*/*OnRefresh* events have fired (maybe focus is set on another form).

When you have a program where you create the header information and then the detail lines (such as the *Sales Order Entry* program), you can have an *OnStartEdit* event against the *Order Header* form that fires as you start entering the header information. The *Stocked Line* form can have its own *OnStartEdit* event, and this would only fire when its form starts being edited.

Note: both the *OnStartEdit* and *OnStopEdit* events should be used sparingly, and you should always check that they fire at the appropriate times under all circumstances. They are considered "legacy" and are present only to provide backwards-compatibility.

OnStopEdit
The *OnStopEdit* event fires when the program knows that you have stopped editing the form and you will not be editing it again. The exact point that this is invoked is program-specific.

In the case of the *Sales Order Entry* example above, as you click on the *End Order* button on the *End Order* screen, the *Stocked Line* form's *OnStopEdit* event fires, followed by the *Order Header* form's *OnStopEdit* event. If you add/maintain another order while you are in the program the *OnStartEdit* event for the *Order Header* form will fire when focus is set on the *Order Header* form.

Note: both the *OnStartEdit* and *OnStopEdit* events should be used sparingly, and you should always check that they fire at the appropriate times under all circumstances. They are considered "legacy" and are present only to provide backwards-compatibility.

OnSubmit
The *OnSubmit* event is invoked when the operator attempts to save the data that has been entered on an entry form, and saving the data would be successful. An example is when saving a stocked line on a sales order. The *OnSubmit* event is invoked each time the form's data is saved, once for each detail line. If the *Save* would fail for any reason (e.g. it would not meet a specified minimum price percentage above the cost) the *OnSubmit* event will not be fired.

When saving the whole sales order, the *Order Header* form's *OnSubmit* event is fired as the sales order is ended. An *OnSubmit* event can be used to validate data (that is on this form, another form, or externally) and if not present/within requirements, it can prevent the *OnSubmit* function from completing by setting the *OnSubmit* function status to *false*.

OnAfterSubmit
The *OnAfterSubmit* event has only been implemented in certain places such as when adding/maintaining a detail line in the *Sales Order Entry* program. Whereas an *OnSubmit* event is

invoked while the sales order line is still displayed and before the data is saved to the server; the *OnAfterSubmit* is invoked after the server program has saved the data, but before the form is cleared.

The sequence of events followed is the *OnSubmit* event invokes first, a check is made that all custom form fields are valid, the business logic checks the validity of the fields, the data is saved, the *OnAfterSubmit* event is invoked and then, if successful, the form clears.

Sequence of Form Events when Adding a Sales Order
The following is a list of form events that fire when creating a sales order with one stocked and one non-stocked line, within the *Sales Order Entry* program. Other events may fire, or these events may fire in a different sequence, depending on the setup options and user preferences set.

Load Sales Order Entry Program

Enter customer number and tab off field
- OrderHeader_OnLoad
- OrderHeader_OnRefresh
- CustomerInformation_OnLoad
- CustomerInformation_OnRefresh
- OrderTotals_OnLoad
- OrderTotals_OnRefresh
- RecentSalesOrders_OnPopulate
- OrderHeader_OnStartEdit

Enter Customer PO number and tab off field

Click on Add Lines button on toolbar
- StockedLine_OnLoad
- StockedLine_OnRefresh
- StockCodeInformation_OnLoad
- StockCodeInformation_OnRefresh
- StockedLineInformation_OnLoad
- StockedLineInformation_OnRefresh
- StockedLine_OnRefresh
- StockCodeInformation_OnRefresh
- StockedLine_OnRefresh
- StockedLine_OnStartEdit

Enter warehouse and tab off field

Enter stock code and tab off field
- StockedLine_OnRefresh
- StockedLineInformation_OnRefresh
- StockCodeInformation_OnRefresh

Enter quantity and tab off field
- StockCodeInformation_OnRefresh
- StockedLine_OnRefresh
- StockedLineInformation_OnRefresh

Click on Save button to save stocked line
- StockedLine_OnSubmit
- StockedLineInformation_OnStopEdit
- StockCodeInformation_OnRefresh

Click OK when prompted “OK to add this line?”
- StockedLineInformation_OnAfterSubmit
- StockedLine_OnAfterSubmit
- OrderTotals_OnRefresh
- EnteredOrderLines_OnPopulate
- OrderTotals_OnRefresh
- LineInformation_OnLoad
- LineInformation_OnRefresh
- StockedLine_OnRefresh
- StockCodeInformation_OnRefresh
- StockedLineInformation_OnRefresh
- StockedLine_OnRefresh
- StockCodeInformation_OnRefresh
- StockedLineInformation_OnRefresh
- StockedLine_OnRefresh
- LineInformation_OnRefresh

Click on Non-stocked tab
- NonstockedLine_OnStartEdit
- NonstockedLine_OnLoad
- NonstockedLine_OnRefresh

Enter Nonstocked code and tab off field

Enter Description and tab off field

Enter Unit of measure and tab off field
- NonstockedLine_OnRefresh

Enter quantity and tab off field
- NonstockedLine_OnRefresh

Enter Product Class and tab off field

Click on Save button to save line
- NonstockedLine_OnSubmit

Click on OK button when prompted “OK to add this line ?”
- NonstockedLine_OnAfterSubmit
- OrderTotals_OnRefresh
- EnteredOrderLines_OnPopulate
- OrderTotals_OnRefresh
- LineInformation_OnRefresh
- NonstockedLine_OnRefresh
- LineInformation_OnRefresh

Click on End Order button on toolbar
- OrderHeader_OnSubmit

Click on End Order button on End Order screen
- StockedLine_OnStopEdit
- StockedLineInformation_OnStopEdit
- NonstockedLine_OnStopEdit
- OrderTotals_OnRefresh
- LineInformation_OnRefresh
- OrderHeader_OnStopEdit

Close Sales Order Entry program

Field Events against a Form, and when they Fire

All fields on forms have their own set of events, and the list of available field events is dependent on whether it is a display or entry form. Fields on a display form only have an *OnMenuSelect* event. Those on an entry form have *OnAfterChange*, *OnBeforeChange*, *OnButtonClick*, *OnGainFocus*, *OnLostFocus*, *OnMenuSelect*, and *OnLinkClicked* events.

OnMenuSelect
The *OnMenuSelect* event requires that a menu be defined for this field first. This is normally defined against the form’s *OnLoad* or *OnRefresh* functions. Only one menu option can be defined for a field. If a menu is defined for a field in both the *OnLoad* and *OnRefresh* events, the one against the *OnRefresh* event will be displayed as its definition will overwrite the *OnLoad* event, because it fires later in the sequence.

Once defined, the menu becomes available when you move the mouse pointer to the left of the value against the field, and click on the *Smartlink Information* icon that appears.
Figure 5-13 shows the *Smartlink Information* icon appearing against the *Currency* field on the *Customer Information* form of the *Customer Query* program. **Figure 5-14** shows the menu that is displayed after clicking on the *Smartlink Information* icon.

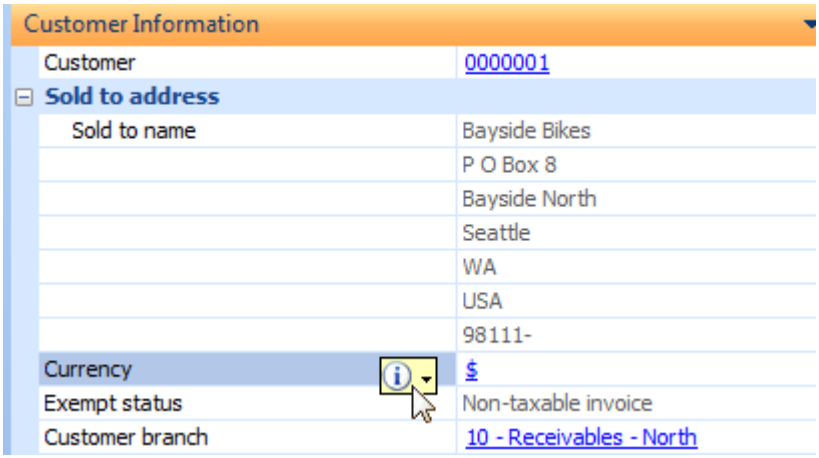


Figure 5-13: The *Smartlink Information* icon against the *Currency* field

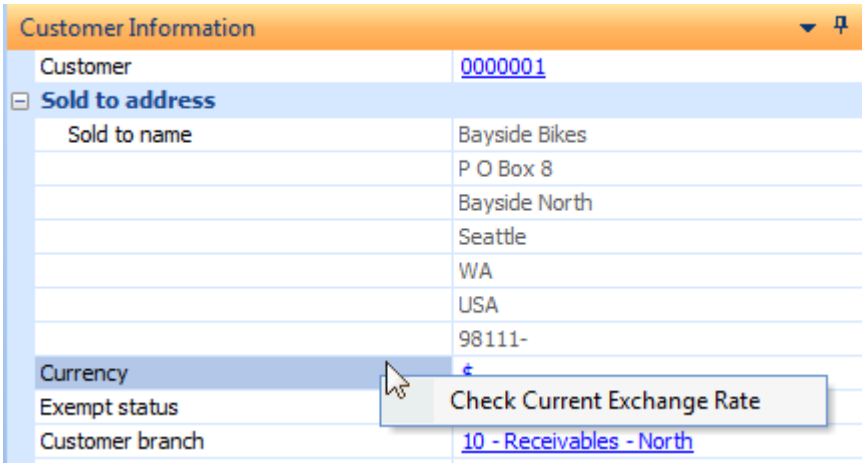


Figure 5-14: The menu displayed after clicking on the *Smartlink Information* icon

The code used to define the menu appearing in **Figure 5-14** is shown in **Figure 5-15**. This code is against the *Customer Information* form's *OnLoad* function, and specifies the menu text for the *CurrencyValue* variable.

The code after the variable name and equal sign was inserted using the *Field Properties* option when editing the VBScript. It has been displayed separately below to make it easier to read. The *Field Properties* option will be covered in a later chapter.

```
"<Field MenuText='Check Current Exchange Rate' > </Field>"
```

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function CustomerInformation_OnLoad()
CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current Exchange Rate' > </Field>"
End Function
```

Figure 5-15: The VBScript against the *OnLoad* function to define a menu

When the displayed menu item is clicked, the *OnMenuSelect* event is fired. This invokes the field's *OnMenuSelect* function within this form's VBScript, and this runs its VBScript code. It is possible to put conditional logic in the code against the *OnRefresh* function so that, for example, the menu option will only appear if the currency code is anything other than $. The code for this example appears in **Figure 5-16**. The lines of code that appear in **Figure 5-16** have been indented to make it easier to follow the logic. This form of indentation does not affect the way the script runs.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function CustomerInformation_OnRefresh()
  If CustomerInformation.CodeObject.CurrencyValue = "$" then
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='' > </Field>"
  Else
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current Exchange Rate' > </Field>"
  End If
End Function
```

Figure 5-16: The code to display the menu depending on the currency code

To perform a task, the *OnMenuSelect* function against the *Currency* field must contain the VBScript code to be run once the menu option has been selected. This VBScript is added in the same way as was done for the form events, except that the *Currency* field must be highlighted before right-clicking on the form and selecting the *Macro for:* option from the displayed context-sensitive menu.

Once the *VBScript Editor* screen is displayed, the *OnMenuSelect* field event should be highlighted and the *Edit VBScript* button clicked. The screen where the VBScript is edited will be displayed, and the *CurrencyValue_OnMenuSelect* function will be created for you. Code should be added that performs the required task between the *Function* line, and the *End Function* line. **Figure 5-17** shows the code to display a message box, but the function could contain the code to call a web service on an external site that would retrieve the exchange rate, and then perform some task with the result.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function CustomerInformation_OnRefresh()
  If CustomerInformation.CodeObject.CurrencyValue = "$" then
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='' > </Field>"
  Else
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current Exchange Rate' > </Field>"
  End If
End Function

Function CurrencyValue_OnMenuSelect()
  msgbox "What is the exchange rate ?",, "Check Exchange Rate"
End Function
```

Figure 5-17: The code against the *OnMenuSelect* function

OnGainFocus

The *OnGainFocus* event is fired when focus is set on the value portion of the specified field. Each time that focus is set on the value portion of this field, this event will fire (i.e. after focus has been set somewhere else, then back on the value portion of this field again).

Care should be exercised when using both the *OnGainFocus* and *OnLostFocus* events. It is possible to cause a loop where the one event causes the other event to fire, which causes the first to fire, etc. Where possible rather use the *OnAfterChange* event.

OnLostFocus

The *OnLostFocus* event is fired when focus was set on the value part of a field, and you click anywhere else or tab off of the field. It is possible that if a new value is entered, it is overwritten if the program returns data from the database on the server, and this is one of the fields that the program refreshes from the database.

Care should be exercised when using both the *OnGainFocus* and *OnLostFocus* events. It is possible to cause a loop where the one event causes the other event to fire, which causes the first to fire, etc. Where possible rather use the *OnAfterChange* event.

OnBeforeChange

The *OnBeforeChange* event is invoked after the operator has changed the value against a field and has moved off the field, but before the data has been changed in memory. This allows for the script to validate that the entered value is acceptable (i.e. falls between a certain range of values/is greater than the contents of another field, etc.) and to revert back to the previous value if it does not match this requirement by setting the *OnBeforeChange* function to *false*. The *OnBeforeChange* event fires before the *OnLostFocus* event.

OnAfterChange
The *OnAfterChange* event fires after the data has been changed in memory, and is invoked after the *OnLostFocus* event. Like the *OnLostFocus* event, it is possible that a change to the value may be overwritten by data returned by the program after a call to the server.

OnButtonClick
The *OnButtonClick* event will only fire if you have created a button on the form for this field. When this button is clicked the event is fired. This does not relate to a browse button that appears automatically against either a master field that is present by default, or one that has been added to this form.

The code to set up the button can be added to either the form's *OnLoad* or *OnRefresh* function. When the event fires (and the VBScript against the function is run) the button appears against the form. If a button (such as a browse button) already exists against this field, this button will appear alongside it, and there will be two buttons for this field.

Figure 5-18 shows the code that creates a button against a scripted field called *JustButton* on the *Order Header* form of the *Sales Order Entry* program. Everything to the right of the equal sign was added using the *Field Properties* option, which is covered in a later chapter. The text that appears on the button is *Click to check*. The button is created large enough to display the supplied text.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function OrderHeader_OnLoad()
  OrderHeader.CodeObject.JustButton = "<Field><Button>Click to check</Button></Field>"
End Function
```

Figure 5-18: The code to create a button against the *JustButton* field

The *Click to check* button appears in **Figure 5-19** alongside the *JustButton* field caption. When this is clicked the *OnButtonClick* event fires that invokes the *JustButton_OnButtonClick* function. The contents of the *JustButton_OnButtonClick* function appear in **Figure 5-20**. In this case it just displays a message that it has fired.

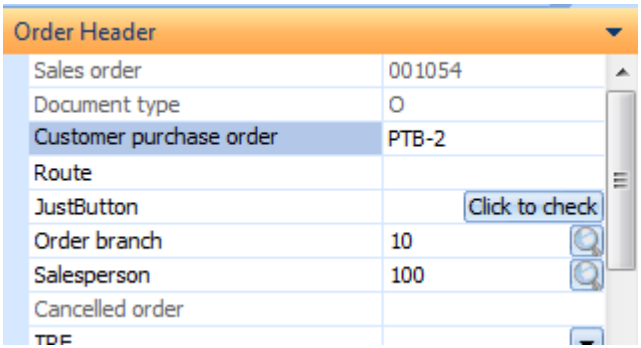


Figure 5-19: The *Click to check* button

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function OrderHeader_OnLoad()
  OrderHeader.CodeObject.JustButton = "<Field><Button>Click to check</Button></Field>"
End Function

Function JustButton_OnButtonClick()
  Msgbox "The JustButton was clicked",, "JustButton"
End Function
```

Figure 5-20: The *JustButton_OnButtonClick* function

OnLinkClicked

The *OnLinkClicked* event fires when you click on a predefined hyperlink on an entry form. Although you can define that hyperlinks appear on a display form, *OnLinkClicked* events are not available on this form type, so this event only relates to fields that are hyperlinked automatically by the program.

Hyperlinked fields on entry forms are used to call up separate programs. If code appears against the *OnLinkClicked* function for a field, the event is fired before the other program is called. **Figure 5-21** shows the *Order Header* form from *the Sales Order Entry* program. The *Sales order notes* field has a hyperlink against it. When this hyperlink is clicked the *Notepad* program is called. If the *notes3_OnLinkClicked* function contains code to display a message box, when this hyperlink is clicked the message box is displayed, and only once this code has been acknowledged by the operator is the *Notepad* program displayed.

The *OnLinkClicked* function can be prevented from completing by setting it to false, in the same way as with the *OnSubmit* function mentioned above, so conditional logic can be added. Using the

example of the *Sales order notes*, the line of code that appears below would prevent the program associated with the hyperlink from being run.

```
notes3_OnLinkClicked = false
```

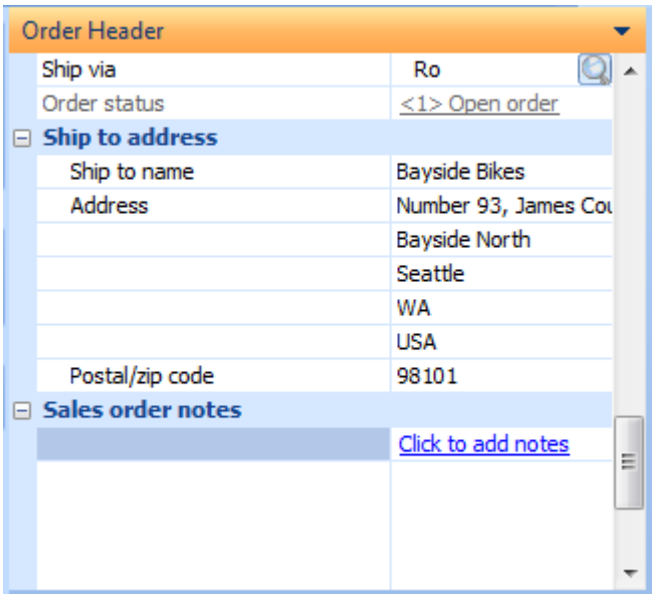


Figure 5-21: A hyperlinked field on an entry form

When do the Listview Events Fire ?

Listviews look similar to spreadsheets in that they are made up of rows, columns, and cells. They are the equivalent of display forms in that you cannot type values into a listview. This section covers standard listviews, not those that have been created with *Customized Panes*. The available events for a customized pane listview are different and are covered in the chapter on *Customized Panes*. Depending on which listview event is firing, you can programmatically change the displayed values, or retrieve the contents of the row that was selected. Listviews have three events; *OnPopulate*, *OnRowSelected*, and *OnDblClick*. The macro events for a listview are added/modified by right-clicking on one of the listview's column headers, and selecting *Customize*, followed by the *Macro for:* option from the context-sensitive menu (see **Figure 5-22**).

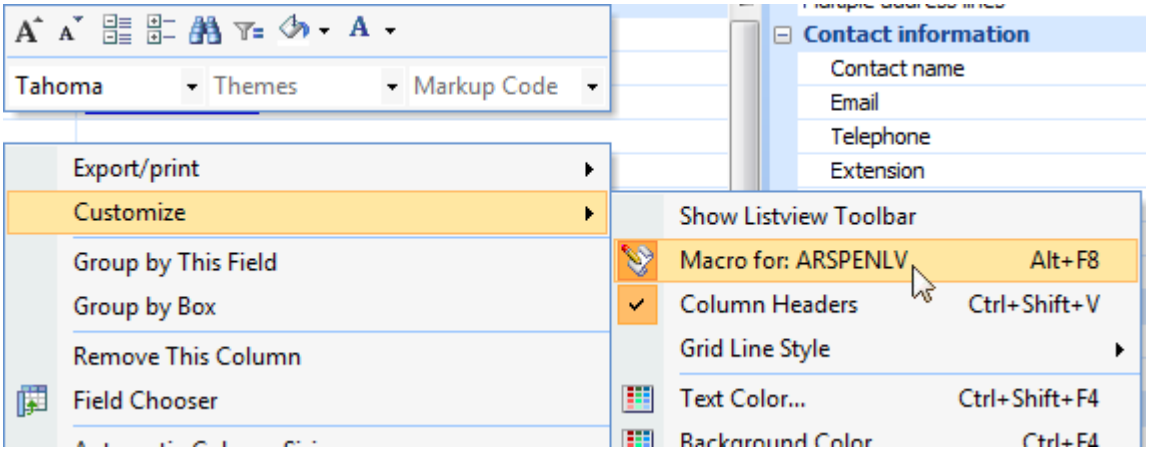


Figure 5-22: Selecting the *Macro for:* option from the context-sensitive menu

OnPopulate

The *OnPopulate* event fires after the data has been displayed in the listview, but before giving control back to the operator. At this point the values in the cells of the listview are available in the form of a two-dimensional array that can be accessed programmatically. A two-dimensional array stores all the values held within the listview in memory, and these can be accessed using their row and column numbers. You can reference the values in this array and display them, use them in condition logic, or pass them to other forms/programs.

A second two-dimensional array with the same name (but with the suffix of *_OUT*) also exists. Values can be written to the cells of the second array, or *Field Properties* can be set against them such as setting the background color, or changing the font. Where the contents of the cell in the *_OUT* array have not been changed, the original values and field properties will be used in the listview display. Cells that have been changed will have these changes applied to the listview display before control is given back to the operator. Note, however, that it is only the displayed value that has changed, not the value in the SYSPRO database.

CustomerInvoices.CodeObject.Array is an example of the array name for the first two-dimensional array and *CustomerInvoices_OUT.CodeObject.Array* is its matching second array name.

The column and row numbers in an array start at zero (instead of starting at one for the listview). When you address the array (either the standard one to retrieve information, or the *_OUT* one to change the displayed information) you need to subtract one from the column and row numbers. So if you need to access the value in column 3, row 5, you would use the address (002,4). Examples of how to interact with an array appear in a later chapter.

OnRowSelected
The *OnRowSelected* event fires when one of the rows in the listview is clicked. The contents of the selected row appears in a two-dimensional array (using the same name as used by the whole listview when the *OnPopulate* event fires). The values in the cells for this row can be accessed using the column number in the array (after subtracting 1), and the row number of zero.

You can change the values and field properties in the *_OUT* array in the same way as for the *OnPopulate* event. If any are changed they will replace those in the listview for this row before control is handed back to the operator. Note, however, that it is only the displayed value that has changed, not the value in the SYSPRO database.

OnDblClick
The *OnDblClick* event is fired by double-clicking on a listview row. It works in exactly the same way as the *OnRowSelected* event; it is only the way that it is invoked that is different.

When do Data Grid Events Fire ?

A data grid is similar to a spreadsheet in that it contains rows, columns, and cells. Unlike a listview it is not read-only, and is used to capture/change values. There are six data grid macro events, *OnPopulate*, *OnSubmit*, *OnRowSelected*, *OnDblClick*, *OnAfterChange*, and *OnLinkClicked*.

OnPopulate
The *OnPopulate* event fires after the data has been displayed in the data grid, but before giving control back to the operator. At this point the values in the cells of the data grid are available in the form of a two-dimensional array that can be accessed programmatically. A two-dimensional array stores all the values held within the data grid in memory, and these can be accessed using their row and column numbers. So you can reference the values in this array and display them, use them in condition logic, or pass them to other forms/programs.

A second two-dimensional array with the same name (but with the suffix of *_OUT*) exists. Values can be written to the cells of the second array, or *Field Properties* can be set against them such as setting the background color, or changing the font. Where the contents of the cell in the *_OUT* array have not been changed, the original values and field properties will be used in the data grid display. Cells that have been changed will have these changes applied to the data grid display before control is given back to the operator. Note that if the content of a cell is changed and this cell could not normally be changed manually (i.e. it is for display purposes only) it is only the displayed value that is changed. If the content is changed for a cell that could be changed normally, and the data grid is saved then this value will be changed in the database (providing that it does not contradict any rules for the contents of this cell).

The column and row numbers in an array start at zero (instead of starting at one for the data grid). When you address the array (either the standard one to retrieve information, or the *_OUT* one to

change the displayed information) you need to subtract one from the column and row numbers. So if you needed to access the value in column 3, row 5, you would use the address (002,4). Examples of how to interact with an array appear in a later chapter.

GLJournalEntry.CodeObject.Array is an example of the array name for the first two-dimensional array and *GLJournalEntry_OUT.CodeObject.Array* is its matching second array name.

OnRowSelected
The *OnRowSelected* event fires when one of the rows in the data grid is clicked. The contents of the selected row appears in a two-dimensional array (using the same name as the whole data grid when the *OnPopulate* event fires). The values in the cells for this row can be accessed using the column number in the array (after subtracting 1), and the row number of zero.

You can change the values and field properties in the *_OUT* array in the same way as the *OnPopulate* event, but with the row number of zero. If any are changed they will replace those in the data grid for this row before control is handed back to the operator. Note, however, that if the content of a cell is changed and this cell could not normally be changed manually (i.e. it is for display purposes only) it is only the displayed value that is changed. If the content is changed for a cell that could be changed normally, and the data grid is saved then this value will be changed in the database (providing that it does not contradict any rules for the contents of this cell).

OnDblClick
The *OnDblClick* event is fired by double-clicking on a data grid row. It works in the same manner as the *OnRowSelected* event.

OnLinkClicked
The *OnLinkClicked* event fires when a hyperlinked value is clicked in a cell within the data grid.

OnAfterChange
The *OnAfterChange* event fires when a value in the data grid has been changed manually. The contents of the row containing the changed value appears in a two-dimensional array (using the same name as used by the whole data grid when the *OnPopulate* event fires). The values in the cells for this row can be accessed using the column number in the array (after subtracting 1), and the row number of zero (as there is only one row, it is always zero).

A second two-dimensional array with the same name (but with the suffix of *_OUT*) exists. Values can be written to the cells of the second array, or *Field Properties* can be set against them such as setting the background color, or changing the font. Note, however, that if the content of a cell is changed and this cell could not normally be changed manually (i.e. it is for display purposes only) it is only the displayed value that is changed. If the content is changed for a cell that could be changed normally, and the data grid is saved then this value will be changed in the database (providing that it does not contradict any rules for the contents of this cell).

OnSubmit
The *OnSubmit* event is invoked when the operator attempts to save the data that has been entered in the data grid, and saving the data would be successful. If the *Save* would fail for any reason, such as some portion of the data fails a validity check, the *OnSubmit* event will not be fired.

An *OnSubmit* event can be used to validate data in this data grid, another form, or externally. If not present or within requirements, it can prevent the *OnSubmit* function from completing by setting the *OnSubmit* event status to *false*.

Global Login/Logout VBScript

The *Global Login/Logout VBScript* option is available from the *Customization Tools* dropdown of the *Administration* tab of the ribbon bar (or *Home* tab for version prior to SYSPRO 7) as indicated in **Figure 5-23**. When this option is selected the *VBScript Editor* screen is displayed, and it contains two macro events, *OnLogin*, and *OnLogout*. If the functions for these two events are created they will be stored in a file called *IMPMENXX* in the `vbscripts` folder on the SYSPRO application server, and executed by everyone that performs a login to SYSPRO.

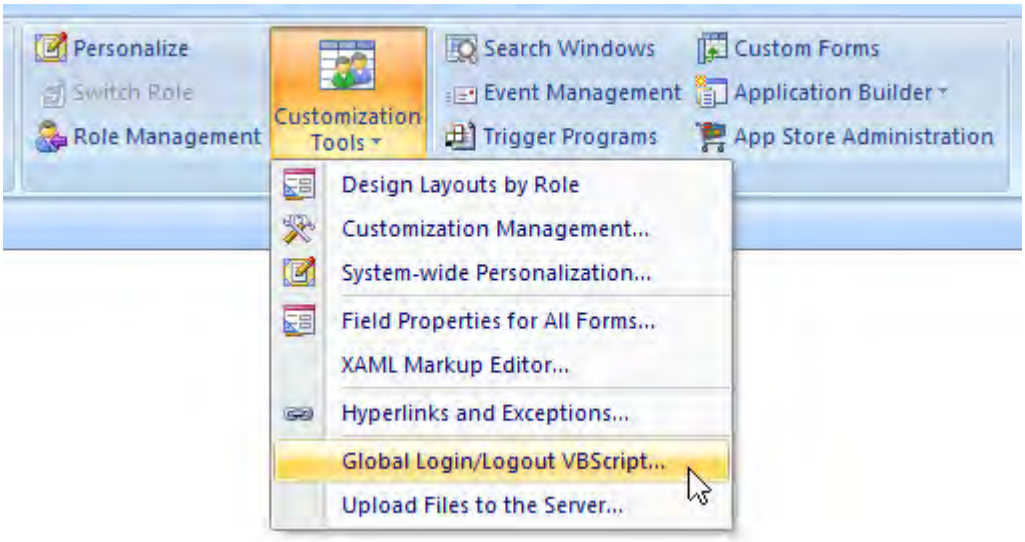


Figure 5-23: The *Global Login/Logout VBScript* option

OnLogin
The *OnLogin* event fires just as you complete the login process (i.e. supplied the operator code, company, etc., and clicked on the *OK* button), but before the main menu is loaded. This means that

checks can be performed to ensure that items are in place, such as drive mappings and other settings, before the operator starts work. If these checks fail the operator can be prevented from logging in by setting *MainMenu_OnLogin* to *false*.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function MainMenu_OnLogin()
  If SystemVariables.CodeObject.Company = "S" Then
    If SystemVariables.CodeObject.Operator = "BOOK" then
      MainMenu_OnLogin = false
    End If
  End If
End Function
```

Figure 5-24: Login script to prevent operator *BOOK* from accessing company *S*

A simple *OnLogin* script appears in **Figure 5-24** that prevents operator *BOOK* from logging in if company *S* is selected. Setting the *MainMenu_OnLogin* variable to *false* logs the operator out. As they have not completed the login process, the *MainMenu_OnLogout* event does not fire.

As the login script runs before the main menu is loaded, you cannot call other SYSPRO programs or interact with any customized panes (whether associated with the main menu or a specific program). However, the script can call e.net Solutions business objects. Only the *System Variables* are available to VBScript against the *OnLogin* function. Some of these such as *SYSPROProgramToRun* will not work, as the main menu is not loaded.

OnLogout

The *OnLogout* event fires when an operator logs out of SYSPRO. Although most of the *System Variables* are available at this point, you cannot use those such as *SYSPROProgramToRun* as they will not load the program. The e.net Solutions business objects can still be used at this point. If you wanted to keep an external copy of who logged in/out you would probably implement the logout part here.

Form Action Events

A *Form Action* is an extremely powerful user-definable hyperlink that resides on a section at the bottom of a form. The hyperlink can be configured to call a SYSPRO program and pass it parameters, or to fire a VBScript event. You can have multiple form actions defined per form, and they can be any combination of these types. Chapter 4 of the first *Power Tailoring* book described how to configure a form action to call a SYSPRO program and pass it parameters, as this requires no development skills.

This section will cover the basics of creating a VBScript form action, as these fire macro events that call functions within a VBScript.

A form action is added using the *Insert Form Action* option from the context-sensitive menu that appears when you right-click on a form. This option can be used to add a form action that will use VBScript, a form action that will call a SYSPRO program (and optionally pass it a parameter) or add a form action template, which is a predefined sample.

Figure 5-25 shows how to add a form action using the *Insert Form Action* option of the context-sensitive menu that is displayed when you right-click on a form. As this form action is going to fire an event (as opposed to call a SYSPRO program) only the description is added, in this case *Check Available Delivery Times*. When you press the *Enter* key the hyperlink is added to the form and the list of available events for this form is shown in the *VBScript Editor* screen.

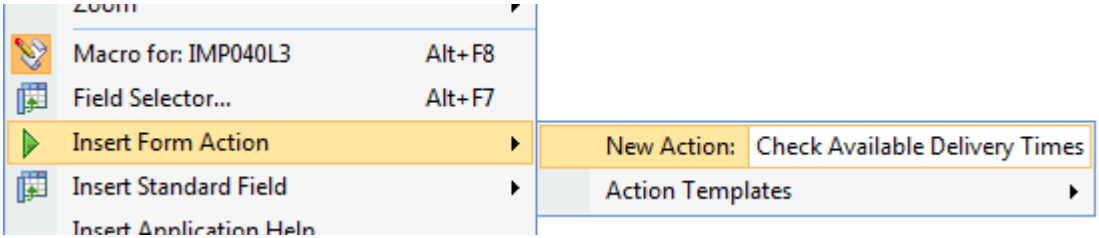


Figure 5-25: Adding a *Form Action* that will fire a VBScript function

Figure 5-26 shows the *Available Events* section of the *VBScript Editor* screen where the O*nCheckAvailableDeliveryTimes* event has been automatically added under a section called *ACTION events*.

The function is added by either highlighting this event and clicking on the *Insert VBScript* button, or double-clicking on this event name. In the same way as any other function, the function name will be made up of the form name and the event name. As the *Check Available Delivery Times* form action was added to the *Sales Order Entry* program's *Order Header* form, the function name is *OrderHeader_OnCheckAvailableDeliveryTimes*.

Figure 5-27 shows the *OrderHeader_OnCheckAvailableDeliveryTimes* function after a message box statement has been added. The message box is used to show that the event is firing when the hyperlink is clicked.

When you exit the screen where the VBScript was edited, you are returned to the *Sales Order Entry* program, where you can see the *Check Available Delivery Times* hyperlink at the bottom of the *Order Header* form (see **Figure 5-28**). When you click on the hyperlink, the event fires and the code against the function is run, in this case displaying a message box.

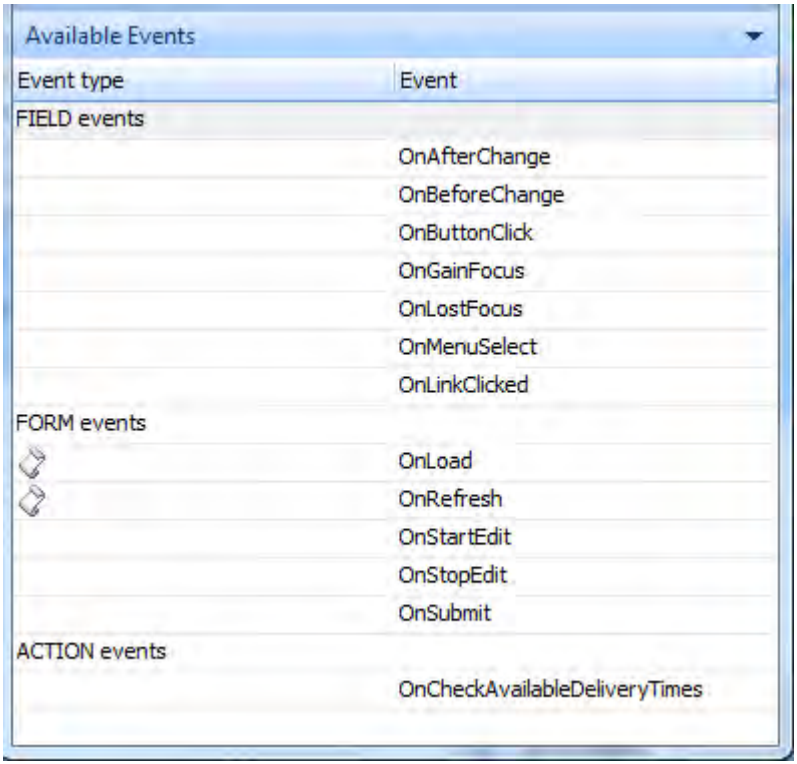


Figure 5-26: The list of available events including the *ACTION events*

```
Function OrderHeader_OnCheckAvailableDeliveryTimes()
  msgbox "OrderHeader_OnCheckAvailableDeliveryTimes Clicked"
End Function
```

Figure 5-27: Adding a message box statement to the function to show that it is firing

The *Insert Form Action* option also displays the currently available form actions (see **Figure 5-29**). These form actions can be removed by clicking on them, and you will be prompted before they are removed. The removal process includes removing *Action event* from the *VBScript Editor* screen, but does not remove the function and code from the VBScript. This needs to be done manually.

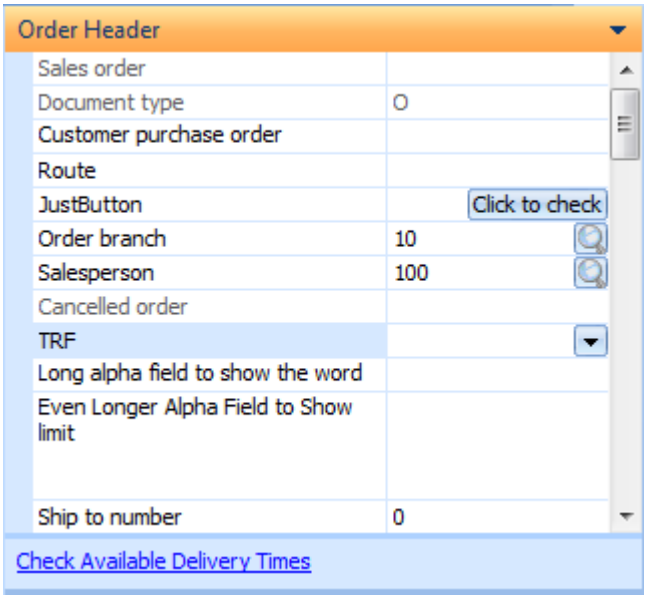


Figure 5-28: The *Check Available Delivery Times* hyperlink on the *Order Header* form

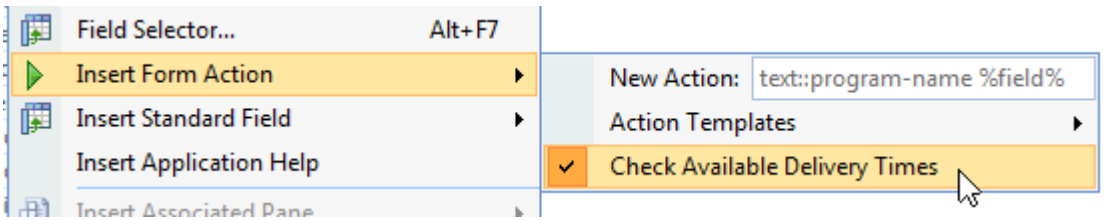


Figure 5-29: Removing the *Check Available Delivery Times* form action

Toolbar VBScript Events

Toolbar buttons can have VBScript functions configured against them. When these have been configured, and the toolbar button is clicked its code will be run (if there is a matching function within the VBScript).

Just like the *Form Action* events above, there is no standard name for a function as it is user-definable; the function name is supplied by the operator. This is best explained with an example.

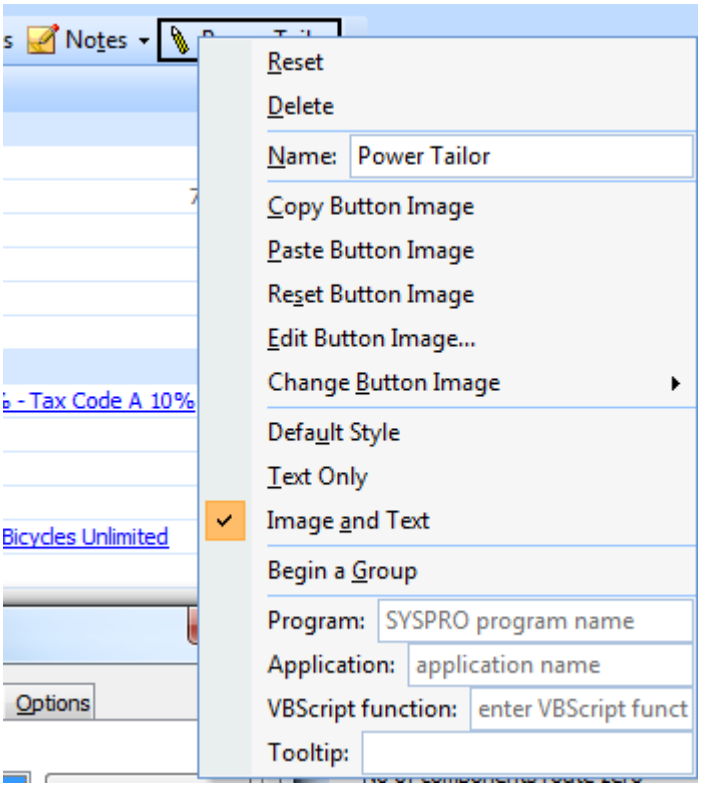


Figure 5-30: Viewing the properties of a toolbar button

Within the *Inventory Query* program, a button has been added to the main toolbar at the top of the screen called *Power Tailor* (typically this is done by copying an existing button and changing the settings against this button). The toolbar's name is *INVPENTB*, which is made up of the six character program name and the letters *TB*. While still in the toolbar's *Customize* mode after adding the button, it is possible to right-click on the button and see the button's properties (see **Figure 5-30**).

The last two options at the bottom of **Figure 5-30** are *VBScript function* and *Tooltip*. If you supply a function name against the *VBScript function* field and press the *Enter* key while focus is still set on this field, the function name will be saved against this button. This will clear out the contents against the *Program* or *Application* fields if present. The function name cannot contain space characters, so if these are supplied they will be stripped out when you press the *Enter* key.

Figure 5-31 shows the same toolbar button after the *VBScript function* and *Tooltip* have been added, but before these have been saved. Note that the supplied function name contains a space character.

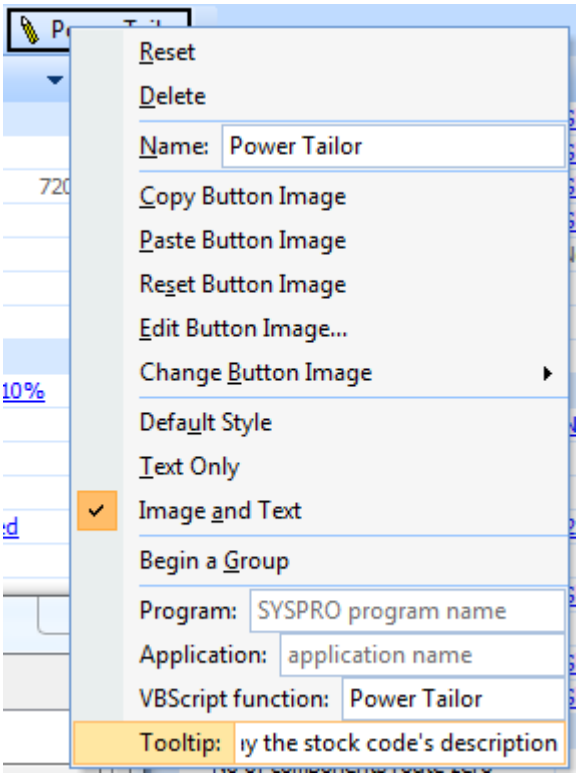


Figure 5-31: Adding the *VBScript function*

Figure 5-32 shows the same button's properties but after they have been saved and then redisplayed. Note that the name against the *VBScript function* field no longer contains a space character. This will be important to know later on.

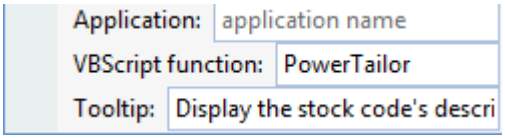


Figure 5-32: The *VBScript function* name after being saved and redisplayed

After exiting from the toolbar customization mode, when you right-click on the toolbar there will be options to display/hide the toolbar, return into customize mode, and to configure macros (**see Figure 5-33**).

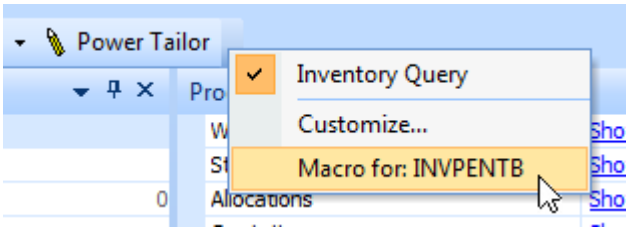


Figure 5-33: Selecting the *Macro for:* option to add the function

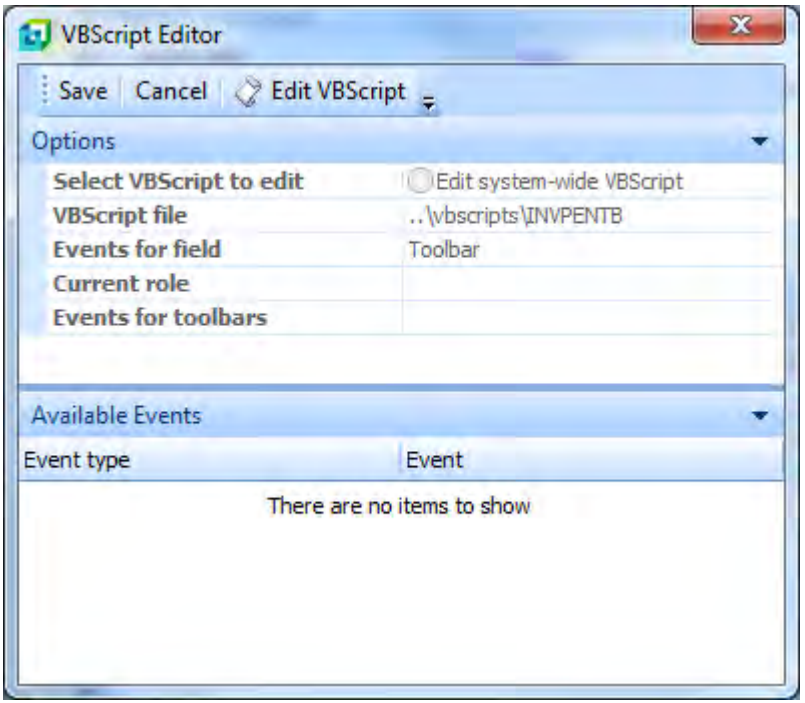


Figure 5-34: The *VBScript Editor* screen with no events in the *Available Events* section

When you select the option to configure macros the *VBScript Editor* is displayed, but no events appear within the *Available Events* section. This is because there are no standard event names for a toolbar; they must all be created manually (see **Figure 5-34**).

When the *Edit VBScript* button is selected the screen where you edit the VBScript is displayed. Unlike the other instances where the function hasn't already existed and it has been created for you, with a toolbar macro the editor is empty. Even the *Option Explicit* statement isn't present (see **Figure 5-35**).

Figure 5-35: The completely empty screen where the VBScript is edited

The first step is to add the *Option Explicit* statement, as this forces variables to be explicitly defined (using a *DIM* statement) before they are used. This is always good practice with a VBScript so that you will be informed if you misspell a variable name, instead of just wondering why your script is not behaving as expected.

Then you need to create the statement that starts your function. This starts with the word *Function*, followed by the name of the function, and two parentheses. This is why it was important to know that the space characters are removed from your function name. The function name used in this example is *PowerTailor*, so your function statement will be:

```
Function PowerTailor()
```

It is also good practice to add the statement that ends the function at this point, so that it doesn't get forgotten. This statement is:

```
End Function
```

If the button was copied, under some circumstances it will still attempt to perform its original task, as well as run the code within this function. To prevent this from happening, add a line of code to set this function name to false. In this example the code will be:

```
PowerTailor = false
```

Next add the code to perform the button's new task. In this example the code displays the stock code's description from the *Stock Code Details* form in a message box (see **Figure 5-36**).

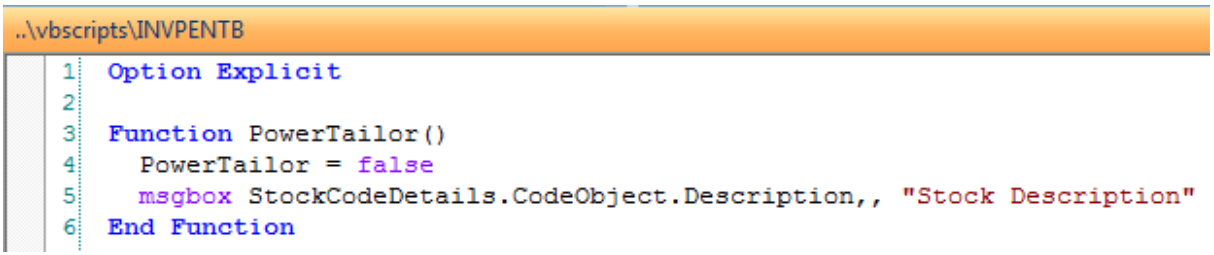


Figure 5-36: The completed function

Exit back to the *VBScript Editor* screen and click on the *Save* button. You will be returned to the *Inventory Query* program. Once you have selected a stock code and click on the *Power Tailor* button, the code within your *PowerTailor* function will be run and the message box displayed. Note: in this example the button name and the name of the function are the same, but they do not have to be the same.

The message box displayed by the *PowerTailor* function when you click on the *Power Tailor* button appears in **Figure 5-37**.

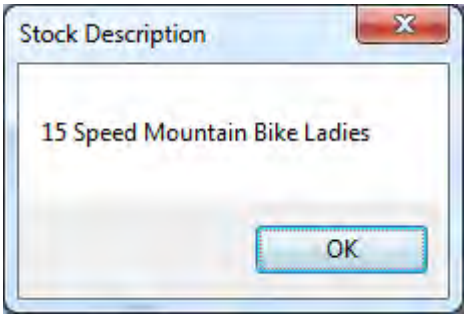


Figure 5-37: The message box displayed by the *PowerTailor* function

Comparing Macro Events / VBScript to SYSPRO Triggers

As mentioned in the first Power Tailoring book, triggers are still available in SYSPRO, despite the huge improvement in flexibility by using VBScript. It is worth noting a few ways that VBScript techniques differ from triggers:

- Trigger programs cannot return values to the SYSPRO forms: A trigger program is invoked or executed at a single point within the SYSPRO application. Once invoked, the trigger program cannot send a result back to the SYSPRO application. VBScript, however, can launch an application and then apply the results to one or more fields on a form.

- Trigger programs run asynchronously: When a trigger program is launched, the SYSPRO application continues to run, even though the trigger program has yet to complete its task. This can cause application timing issues, since the trigger program may assume that certain files or tables have been updated (or not) at the time the trigger program is executed.

- Using VBScript, the script code executes synchronously (i.e. the code must complete its task before it returns control to the SYSPRO application).

- Macros can only be added or modified if the security *Activities* against the operator code allow it. If the operator security *Activity 'Forms – Customization by operator*' is denied the operator will not be able to add or maintain a macro, although the macros will be run if they are present.

Chapter 6 - Using the VBScript Editor

VBScripts are created/edited within a screen that has the title *VBScript for:* followed by the name of the form, listview, customized pane, or data grid that is being edited. **Figure 6-1** shows the default layout of the *VBScript for:* screen where the VBScript associated with the *Customer Information* form in the *AR Customer Query* program is being edited.

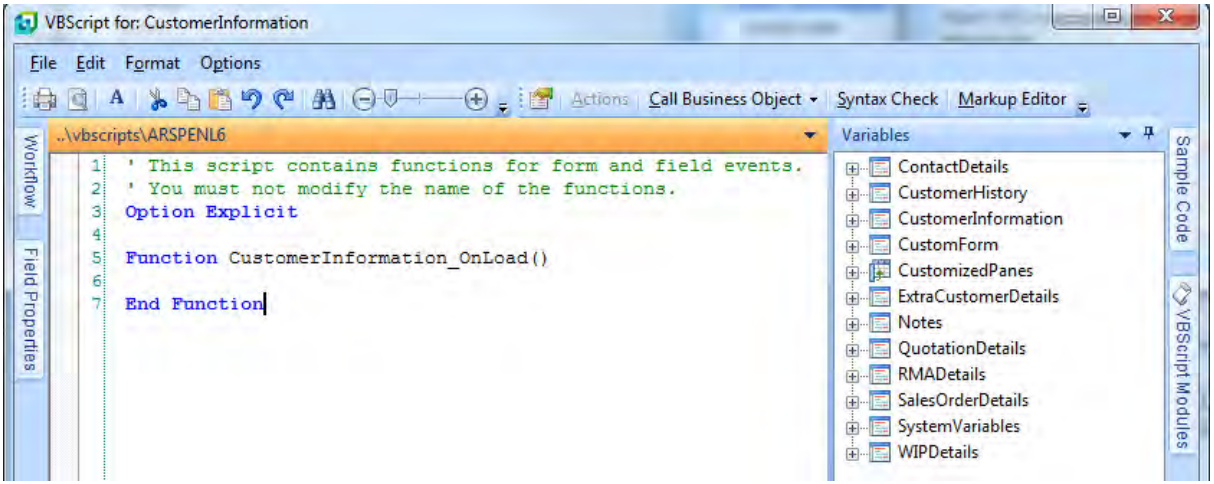


Figure 6-1: Editing the script associated with the *Customer Information* form

Menu Bar

At the top of the screen is the *Menu Bar*, which consists of *File*, *Edit*, *Format*, and *Options* (see **Figure 6-2**). Most of these contain the usual *Print*/*Copy*/*Paste* type of options that you would find on a typical menu bar, so there is no point in covering them here. Instead, only the options specific to this program will be covered.

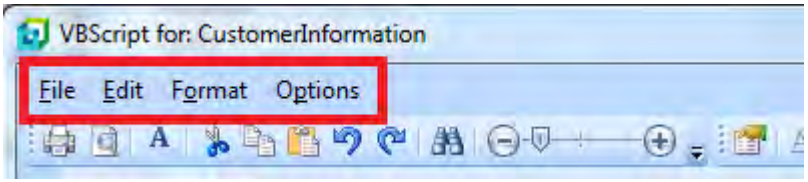


Figure 6-2: The *Menu Bar*

Bookmarks

Bookmarks are typically only useful when you have large scripts, as they enable you to move around between the different sections of the script. The three bookmark options are under the *Edit* section of the menu bar, and are *Toggle Bookmark*, *Next Bookmark*, and *Prev Bookmark*. The *Toggle Bookmark* option (which can also be selected using the *Ctrl+T* shortcut keys) enables you to place a bookmark against the currently selected line. The *Next Bookmark* option (also available using the *F3* shortcut key) enables you to move to the next bookmark down the screen. If there are no more bookmarks below this one you are taken back to the first one. The *Prev Bookmark* option (also available using the *F4* shortcut key) takes you to the next bookmark up the screen. If there are no more bookmarks above your current line, it will start again from the bottom of the screen.

..\vbscripts\ARSPENL6

```
 1  ' This script contains functions for form and field events.
 2  ' You must not modify the name of the functions.
 3  Option Explicit
 4
 5  Function CustomerInformation_OnLoad()
 6    Msgbox "Display START message"
 7    Dim VariableA, VariableB, VariableC, MyXML
 8    VariableA = CustomerInformation.CodeObject.Name
 9    VariableB = CustomerInformation.CodeObject.Customer
10    VariableC = CustomerInformation.CodeObject.Salesperson
11
12    ' Build the required XML
13    MyXML = "<MyRoot><Options>"
14    MyXML = MyXML & "<CustName>"
15    MyXML = MyXML & VariableA
16    MyXML = MyXML & "</CustName>"
17
18    ' Validate Current Company
19    If SystemVariables.CodeObject.Company = "0" then
20      If SystemVariables.CodeObject.Operator = "PAUL" then
```

Figure 6-3: Using *Bookmarks*

In **Figure 6-3** there are three bookmarks, these are against lines 7, 12, and 19. From the top of the screen the *Next Bookmark* option was selected. The border of the bookmark against line 7 became highlighted and the cursor placed at the beginning of this line. Pressing *F3* would move the cursor to the beginning of line 12, or pressing *F4* would take the cursor to the bottom of the script and find the bookmark nearest the bottom.

Bookmarks are removed by highlighting the line containing the bookmark and selecting the *Toggle Bookmark* option. Bookmarks only exist during the time that you are editing this script. So when you close this screen back to the *VBScript Editor* and either *Save*, or *Cancel*, the bookmarks will disappear. Exiting back to the *VBScript Editor* screen (without exiting it) and clicking the *Edit VBScript* button will take you back to the *VBScript for* screen and the bookmarks will still be present.

Show Line Numbering

Under the *Options* menu item is the *Show Line Numbering* option. This is used to display/hide the column that contains the line numbers. This column can be seen in **Figure 6-3**, and is the second column from the left. There are several benefits to using the line number columns. The most useful is when you receive an error message regarding the VBScript. A pre-processor works through the script looking for any syntax errors, and reports them along with the line number that is causing the error.

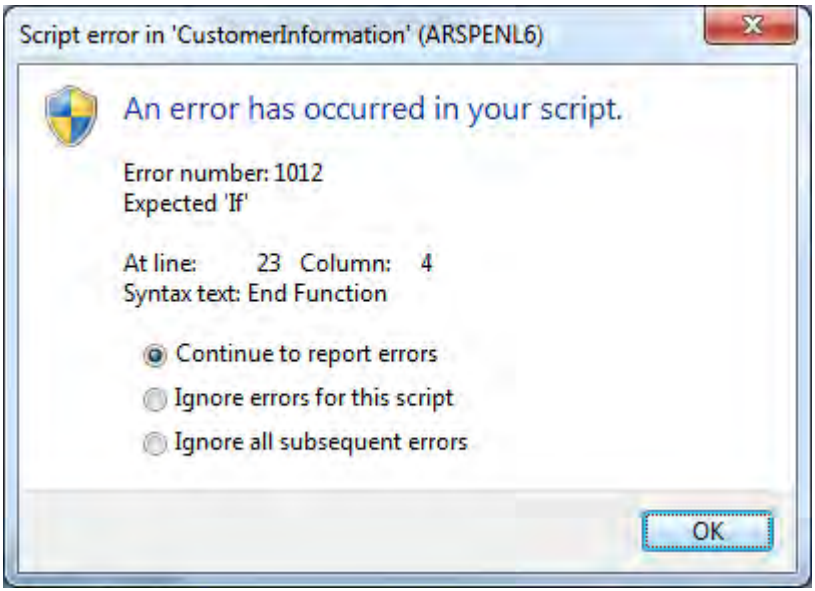


Figure 6-4: The pre-processor reporting a syntax error, including the line number

Figure 6-4 shows the error message that was displayed when a syntax error was detected. The line number that is supplied gives you an idea of where to start looking in your script to find the reported

problem. Prior to this functionality being implemented, you would need to count the lines to find the location.

If this option is unchecked the whole column is removed. This will give you a marginal benefit if screen real estate is at a premium.

Show Selection Margin
The *Show Selection Margin* option is used to show/hide the column that is used to display the bookmarks in **Figure 6-3**. If the margin is not present, the bookmarks are still there, you just can't see them. They also work in exactly the same was as they would do if they were visible. If this option is unchecked the whole column is removed. This will give you a marginal benefit if screen real estate is at a premium.

Enable Virtual Space
The *Enable Virtual Space* option logically inserts space characters at the end of each line. This is useful when you need to use the keyboard's arrow keys to move around, and the lines of code are of different lengths. For example, you have three lines of code. The first line is 40 characters long, the second is 10 characters long, and the third is 38 characters long. If you were positioned towards the end of the first line, and used the arrow key to move down, the cursor would jump to the last character of the second line. When you used the arrow key to move down one more line the cursor would jump to the right as it moved to the third line.

With the *Enable Virtual Space* option checked, the cursor will move down from its position on the first line to exactly the same position on the second line, even though the code did not extend that far. This also enables you to see if text lines up, even if they are separated by many lines.

e.net Logon Method
The *e.net Logon Method* was important prior to SYSPRO 6.0 Issue 010 Service Pack 2. It was used to specify whether the e.net Solutions login was performed using *COM*/*DCOM*, or using *Web Services*. Before this version, if you wanted to use an e.net Solutions business object you would access a *System Variable* called *enetGUID* to retrieve the 34-character *UserID*, and you would supply this *UserID* whenever you accessed a business object using SYSPRO's VBScripting. This *UserID* is only valid for the duration of the execution of the current script and cannot be used outside of the script environment.

When the *enetGUID System Variable* was accessed for the first time by this operator during this run of SYSPRO, an attempt was made to logon to e.net Solutions using whichever *e.net Logon Method* was selected (see **Figure 6-5**). This meant that in a client server environment, either *DCOM* (along with its permissions) had to be configured correctly, or the SYSPRO *Web Services* had to be present and working before e.net Solutions business objects could be used within SYSPRO's scripting. Although interacting with the business objects can still be done this way (so that existing power tailoring continues to work when customers upgrade from prior versions) enhancements were made so that interacting with the business objects from within SYSPRO is much easier.

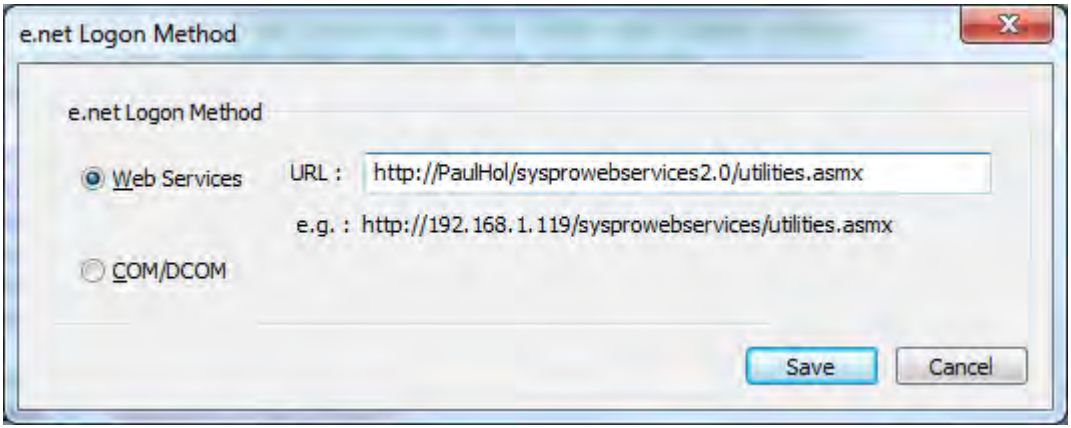


Figure 6-5: The *e.net Logon Method* screen

The *enetGUID* is no longer required when using the *Call Business Object* option from the toolbar (introduced in SYSPRO 6.0 Issue 010 Service Pack 2), which will be covered later. All you need to know about the *e.net Logon Method* is that it allows you to choose whether to use *COM*/*DCOM* or *Web Services* if you are not using the default *Call Business Object* way of interacting with business objects.

Although set within a screen used to edit a VBScript, the *e.net Logon Method* is a global setting. It doesn't matter which operator sets it, or from which company it is done. It affects all operators that login from that point onwards, for all companies.

VBScript Timeout

The *VBScript Timeout* is particularly useful when writing new code and running it the first few times. It sets the amount of time (in seconds) that the script can run before it is interrupted, and the operator given the option to terminate the script, or to let it continue. If the script finishes within this timeout period, everything continues as normal. If there is a fault in the VBScript code that causes it to go into a loop, this timeout option allows the operator to exit from the script gracefully (i.e. without having to terminate SYSPRO).

Figure 6-6 shows the screen where the timeout is configured, and its default value is 10 seconds. Apart from setting a timeout in seconds (with a maximum of 999) it can also be set to never timeout. Setting this *Unrestricted timeout* option is probably not a wise thing to do, as the maximum of 999 seconds is already over 16 minutes, and an operator is unlikely to wait that long before attempting to terminate a process.

Using the example of the default 10 seconds timeout value, if the script is still running after 10 seconds a message will be displayed asking the operator whether the script should continue, or if it should be aborted. The script will stop processing at this point and wait for input from the operator.

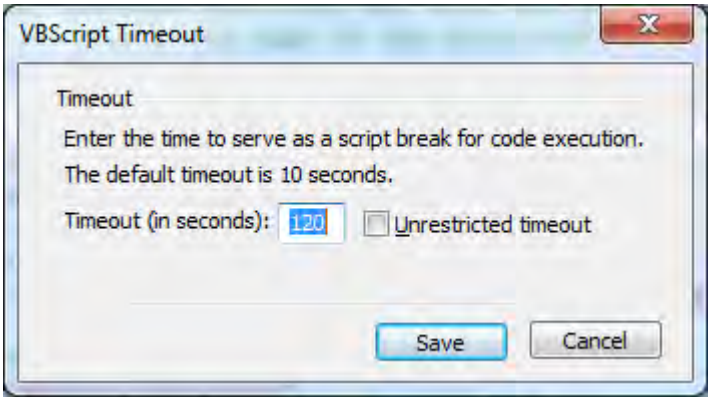


Figure 6-6: Setting the *VBScript Timeout*

The message to the operator asking if the script should be ended or continue appears in **Figure 6-7**. If the operator selects the *Continue* button, the script will resume from where it paused. It will continue either for another 10 seconds, or until the script finishes, whichever comes first. This cycle continues until the script finishes or the operator selects the *End* button.

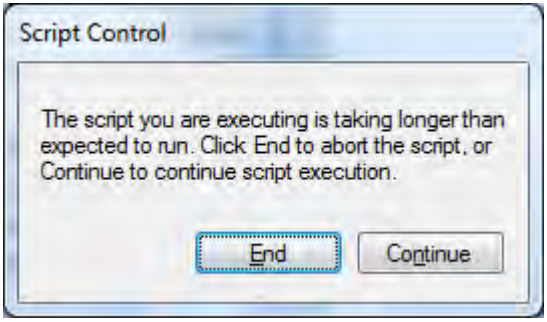


Figure 6-7: The VBScript being interrupted by the timeout

If the operator selects the *End* button and the script is terminated, a message to this affect is displayed showing the line number that the script was on when it was terminated (see **Figure 6-8**). If you have the *Show Line Numbering* option checked (as mentioned above) you can use this line number to narrow down the search for the fault in the script.

It is advised to set this timeout value to a reasonable length of time (such as 120 seconds) because if the script finishes within this time it will not affect the operator in any way. If the script takes the full time of this value, there is likely a problem with the script, or something that the script is calling (such as a remote web service, database, or other remote machine).

The *VBScript Timeout* value is global and affects all operators and all companies.

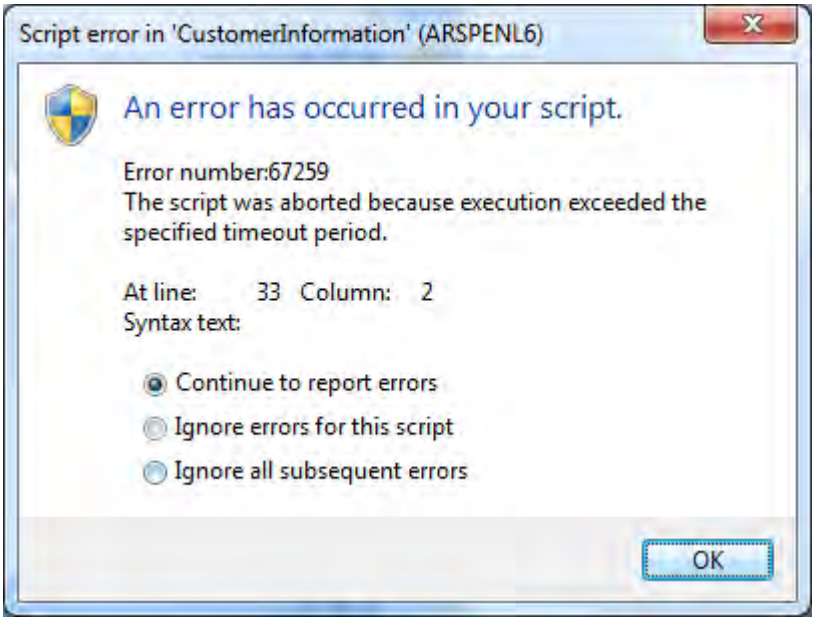


Figure 6-8: The message received if you terminate a script

Toolbar

The *Toolbar* appears just below the menu bar in the *VBScript for:* screen (see **Figure 6-9**). In the same way as was handled in the menu bar section, the usual *Copy*/*Paste*/*Print* type of options will not be covered here.

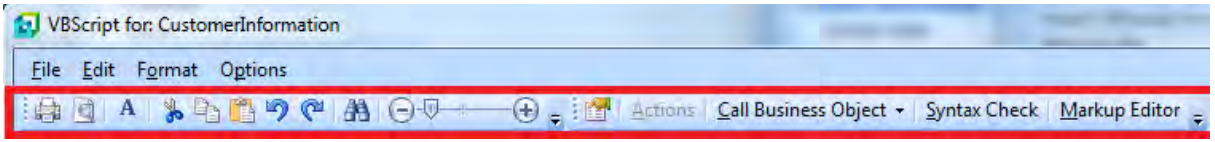


Figure 6-9: The standard *Toolbar*

When you move your mouse pointer over the buttons on the toolbar, a tooltip is displayed, so if the button's function is not clear from its image, you can see what the button does.

Show Field Properties

Although the *Field Properties* will be covered later in this chapter, the *Show Field Properties* button will be covered here. The *Field Properties* pane can be in one of three states. It can be pinned in place

using the *Auto Hide* option, it can appear as a tab because it is still resident on the screen but not pinned in place, or it could have been deleted by the operator if they clicked on the *X* in the top right of its pane.

If the *Field Properties* pane is pinned in place and you click on the *Show Field Properties* button, the cursor will be positioned on the last field property that you accessed, or if this is the first time that you have used the field properties the cursor will be positioned on the first field within this pane.

If the *Field Properties* pane is not pinned in place (so appears as a tab) and you click on the *Show Field Properties* button, the pane will expand and the cursor will be positioned on the last field property that you accessed, or if this is the first time that you have used the field properties the cursor will be positioned on the first field within this pane. The *Field Properties* pane will not be pinned in place, so if you click anywhere else within the *VBScripts for:* screen the *Field Properties* pane will revert back to a tab.

If the *Field Properties* pane has been deleted, clicking on the *Show Field Properties* button will reinstate the pane, and pin it in place on the left of the window. The cursor will also be placed on the pane.

Actions

An *Action* enables you to programmatically invoke a section of the code of the SYSPRO program from your VBScript. The *Actions* button is only enabled where a form implements *Actions*, and the available actions are specific to the form being edited. **Figure 6-10** shows the *Actions* screen that is displayed if the *Actions* button is clicked when editing a script against the *Sales Order Entry* program's *Stocked Line* form. This example has five actions.

The *DoRefreshLines* action causes the *Entered Order Lines* listview (that contains all the current detail lines of this sales order) to be refreshed. An example of when this might be used is if you are using the ***SORTOI*** business object to add detail lines to the sales order while you are still maintaining it within SYSPRO. As the business object is adding the sales order lines without using the SYSPRO user interface, the list of detail lines in the *Entered Order Lines* listview is not being updated. When this action is invoked it performs exactly the same task as if you right-clicked on the listview's column header and selected the *Refresh Lines* option.

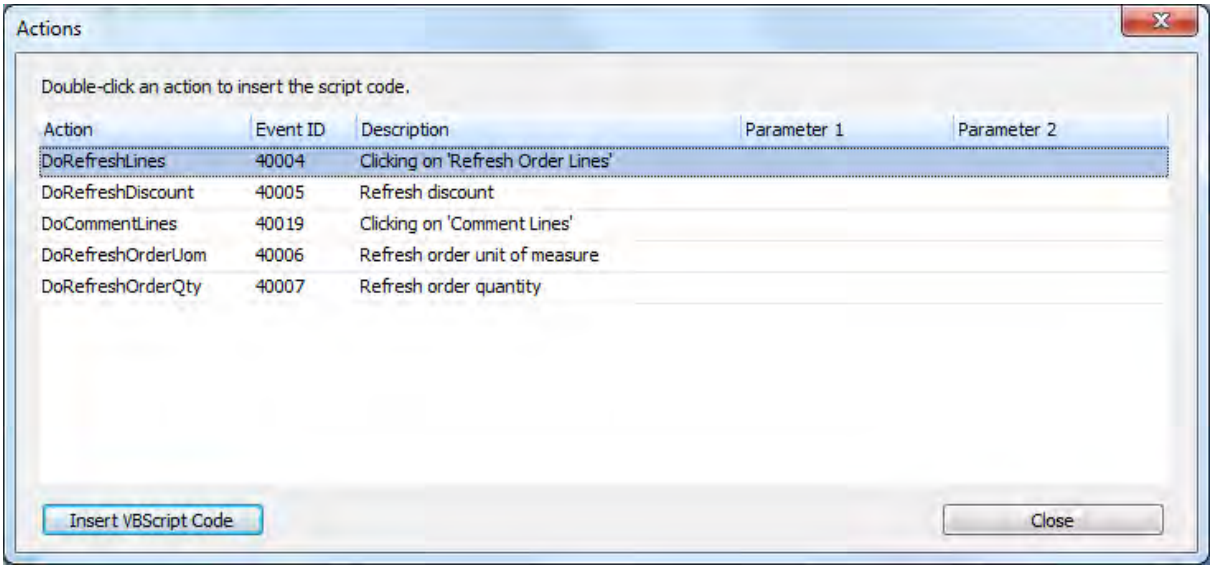


Figure 6-10: The *Actions* screen

An action is added to your code by highlighting the required line and clicking on the *Insert VBScript Code* button. **Figure 6-11** shows where the *DoRefreshLines* action from **Figure 6-10** has been added to a function in the VBScript code of the *Stocked Line* form. When this part of the code is run the program will retrieve all the current details lines from the server and populate the listview with the data. The listview's *OnPopulate* event will also fire, and any code against the *EnteredOrderLines_OnPopulate* function will be run.

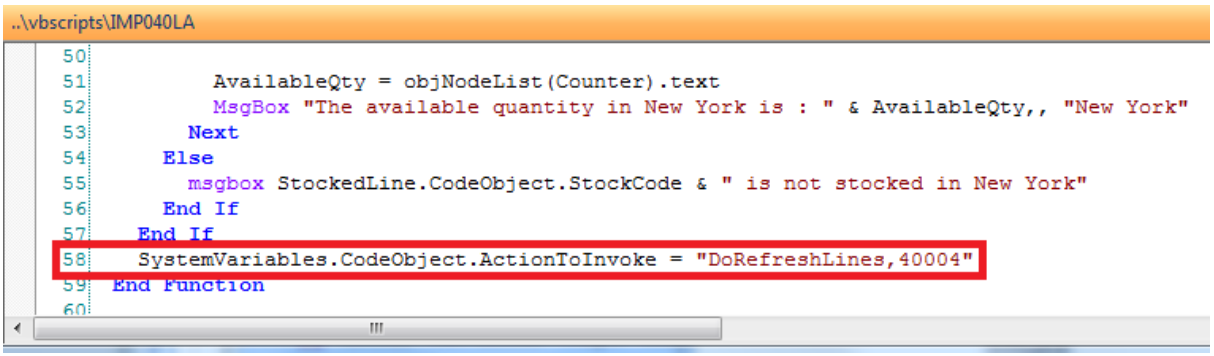


Figure 6-11: The *DoRefreshLines Action*

Only one action can be invoked within a function. If there are multiple actions in a function, only the last one will be run. In the sample of code that appears below there are two *ActionToInvoke*

statements. The first one refreshes the contents of the *Entered Order Lines* listview, and the second calls up the *Comment Line* screen where you can enter free format comment lines (as if you had clicked on the *Comment* button on the *Order line* toolbar). Because only one action can be run per function, and any later one will overwrite an earlier one, the action to call up the free format comment lines screen will be invoked. If these two lines appeared the other way around, the *Entered Order Lines* listview would be refreshed.

```
Function JustButton_OnButtonClick()
  SystemVariables.CodeObject.ActionToInvoke = "DoRefreshLines,40004"
  SystemVariables.CodeObject.ActionToInvoke = "DoCommentLines,40019"
End Function
```

A complete list of the available *Actions* can be found in a file called `IMPVBS.IMP` in the SYSPRO `Programs` folder. This includes the program name, forms, and the actions associated with these forms.

Note: *Actions* have by and large been superseded from SYSPRO 7 onwards, as you can use the *ToolbarButton* system variable action to develop code to manipulate any buttons in any toolbar.

Call Business Object

The functionality behind the *Call Business Object* button replaces the need to configure *DCOM* or *Web Services*, and to use the *enetGUID* system variable. When a business object is called after being configured using this wizard, the business object is called using the SYSPRO architecture, not using the web services or DCOM.

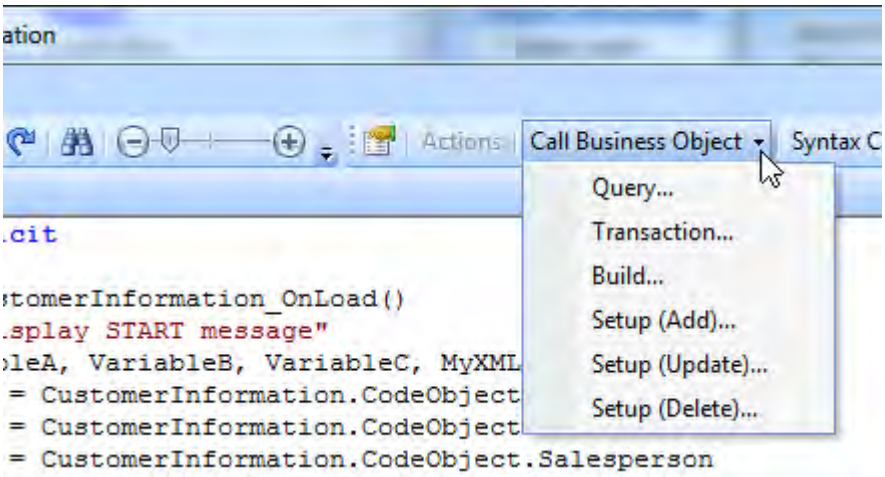


Figure 6-12: The *Call Business Object* button

If the text portion of the *Call Business Object* button is clicked (the portion containing the text *Call Business Object*), the *Call a Query Business Object* screen is displayed. Clicking on the inverted triangle icon alongside of the text on the *Call Business Object* button displays a dropdown list containing all the different types of business objects that can be called from this button (see **Figure 6-12**). If one of these options is selected the screen displayed will match the selected business object type.

Call Business Object - Query

The *Call a Query Business Object* screen consists of a toolbar, a *Parameters* pane (with both *Parameters* and *Parameters XSD* tabs) and a *Sample XMLOUT* pane. The toolbar has a *Query Business Object* prompt where the business object name can be entered, selected from a dropdown list, or the browse button can be used to find the correct business object (see the top of **Figure 6-13**).

If the business object name is entered manually, when you tab off the business object name the *Sample XMLOUT* pane, *Parameters* tab, and *Parameters XSD* tab will be populated with the contents of the sample XML, sample XMLOut, and the schema, if they are present. If the business object name is selected from the browse or dropdown list, the schema and samples will be displayed without the need to tab off the field.

The XML that is displayed in the *Parameters* tab should be changed to reflect the XML that you want to supply to the business object as closely as possible. In **Figure 6-13** the business object name supplied is ***ARSQRY***, which is the business object that performs a customer query. As the default values for most of the XML parameters will return the required results from this business object, the XML has been reduced to the absolute minimum.

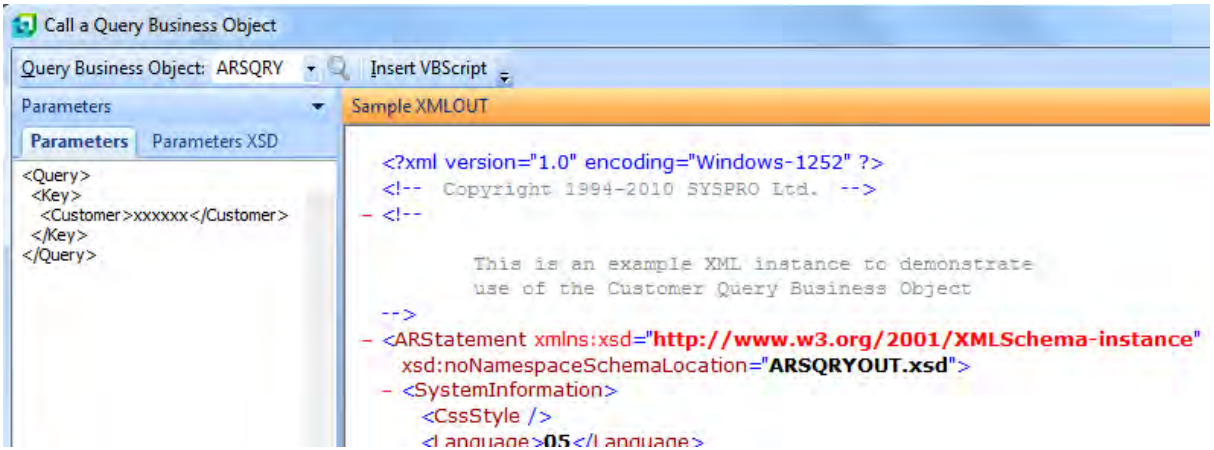


Figure 6-13: Using the *Call a Query Business Object* wizard

Note that sometimes it is better to supply slightly more XML parameters to the business object so that it returns significantly less XML. This can be useful in low bandwidth environments, or where you only need small amounts of the returned data, which then needs to be put into the Document Object Model (DOM) and the data extracted from it. If in doubt, rather return a smaller quantity of XML, providing that it contains everything that you need.

In **Figure 6-13** the value that appeared against the *Customer* element in the *Parameters* tab has been replaced with a string that is known to be an invalid customer account code. This way, if you later forget to replace the customer account code with the variable that you are using for the customer code, the XML will report that there is a problem.

If you were to leave the customer code from the sample XML, and it happens to be a valid customer code on your system, you may not notice that the results from the business object are not what should be returned for the currently selected customer. Using a customer code such as *xxxxxx* means that it is more obvious that something is amiss when you scan through your code, and if you are consistent in using the same string when requiring a variable to replace it, you can perform a search to see if this string exists in the script.

..\vbscripts\ARSPENL6

```
22
23  dim XMLOut, XMLParam
24
25  XMLParam = XMLParam & " <Query>"
26  XMLParam = XMLParam & "   <Key>"
27  XMLParam = XMLParam & "     <Customer>xxxxxx</Customer>"
28  XMLParam = XMLParam & "   </Key>"
29  XMLParam = XMLParam & " </Query>"
30  on error resume next
31  XMLOut = CallBO("ARSQRY",XMLParam,"auto")
32  if err then
33    msgbox err.Description, vBCritical, "Calling Business Object"
34    exit function
35  end if
36  ' Switch on error handling
37  on error goto 0
38
39
40  End Function
```

Figure 6-14: The inserted VBScript code to call the *Query* business object

Once you have manipulated the XML in the *Parameters* tab to your requirements, you click on the *Insert VBScript* button on the toolbar. This builds and inserts the VBScript code that contains the XML

and calls the specified business object. This is inserted at the point where your cursor was positioned within the editor window before calling the wizard.

Figure 6-14 shows the VBScript code that was built from the business object name and XML supplied in **Figure 6-13**. The first line of code (that appears against line 23) defines the name of two variables that will be used in the code. The *Call a Query Business Object* will always use the variable names *XMLOut* and *XMLParam*, although these can be changed manually after they have been inserted.

Care should be taken that these variable names have not already been used in this function,either by coincidence, or where this is the second business object that is being called in this function, and the wizard was used to build both VBScripts.

The *Name redefined* error that will appear under these circumstances is shown in **Figure 6-15**. The names of these two variables will need to be changed, along with wherever these are used within this section of code.

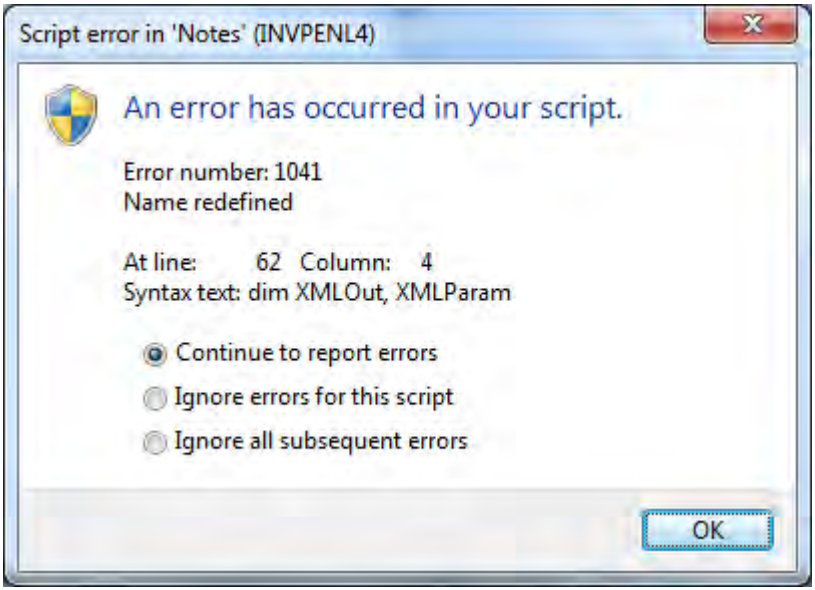


Figure 6-15: The error message that is displayed when attempting to use a variable name twice

The next section of code (lines 25 to 29) is building up the XML in the variable *XMLParam* that was declared above. When it is declared it does not contain anything. Each of the lines below performs the equivalent of “make the variable *XMLParam* contain the existing contents of this variable, plus whatever exists within the quotes of this line”. These lines appear below.

```
XMLParam = XMLParam & "<Query>"
XMLParam = XMLParam & "  <Key>"
XMLParam = XMLParam & "    <Customer>xxxxxx</Customer>"
XMLParam = XMLParam & "  </Key>"
XMLParam = XMLParam & "</Query>"
```

When these five lines have been processed the *XMLParam* variable will contain the following string:

```
<Query>  <Key>    <Customer>xxxxxx</Customer>  </Key></Query>
```

When processing XML, the business objects strip out the space characters that appear outside of the elements. You could have stripped out the spaces from your XML within the wizard, but that makes it harder to follow when you need to work out what is happening in the code.

The command on line 30 is “on error resume next”, and is used to force the script to continue if there is a non-critical issue when calling the business object. This is followed on line 31 by the line that calls the business object, which has been duplicated below.

```
XMLOut = CallBO("ARSQRY",XMLParam,"auto")
```

This line calls the business object using *CallBO* (which is for query business objects) and passes it the business object name, contents of the *XMLParam* variable, and e.net Solutions instance of *“auto”*. Apart from if the business object throws an exception, whatever is returned by the business object is placed into the *XMLOut* variable.

The contents of lines 32 to 37 are used to handle exceptions thrown by the business object. For example, if you are calling the ***ARSQRY*** business object and you supplied an invalid customer code, the business object would throw an exception. This section of code will detect this and exit without continuing. If this section was not present, your code may attempt to load the contents of the *XMLOut* variable into the DOM and start processing it.

Call Business Object - Build

Although the *Build* option of the *Call Business Object* button works the same way as *Query*, the line that calls the business object is different in that it uses *CallTrn*, instead of *CallBO*. *CallTrn* is used to call both *Build* and *Post* method business objects, which belong to the *Transaction* class. *Build* method business objects are used to build information that will be used by the *Post* business objects. A *Build* business object only accepts one XML string (referred to as the document string) whereas a *Post* business object expects both a parameter and document XML string.

The line to call a *Build* business object appears below. This statement contains *CallTrn* and passes it the business object name. Because this is a build business object the next parameter that is passed to *CallTrn* is an empty string (because there is no parameter XML string). This is followed by the contents of the variable containing the document XML string, the text to tell the business object that this is a *Build* method business object, and the e.net Solutions instance of “*auto*”.

```
XMLOut = CallTrn("SORRSH"," ",XMLParam,"Build","auto")
```

Call Business Object - Transaction

The wizard associated with the *Transaction* option of the *Call Business Object* button is used to call a *Transaction* class, *Post* method business object. It is different from the two mentioned above as it must cater for two XML strings.

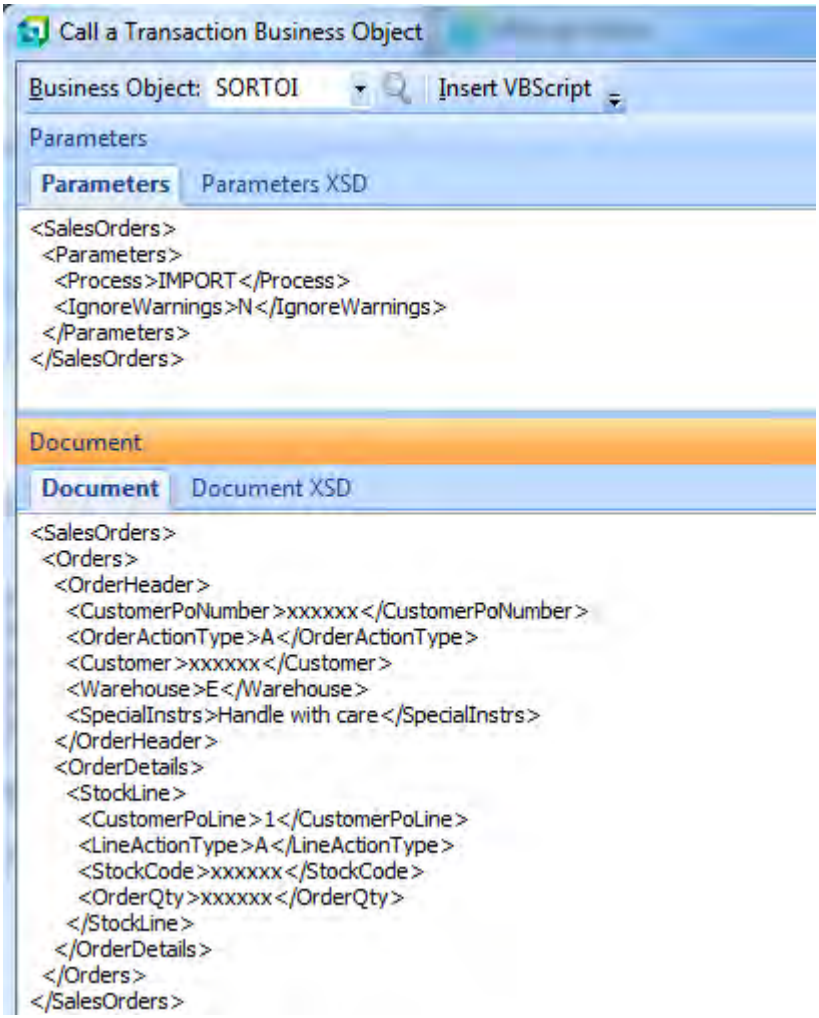


Figure 6-16: The *Call a Transaction Business Object* wizard

This wizard has three panes instead of two, *Parameters*, *Documents*, and *Sample XMLOUT*. The *Parameters* pane has both *Parameters* and *Parameters XSD* tabs, and the *Document* pane has both *Document* and *Document XSD* tabs (see **Figure 6-16**).

Within the wizard you must supply the business object name, and the tabs on the *Parameters* and *Document* panes will be populated with the sample XML. Update the XML files to suit your requirements, and click on the *Insert VBScript* button on the toolbar. The VBScript code to build both the parameter and document XML strings, along with the statement to call the business object will be created and inserted for you. These will be inserted where your cursor was positioned before calling the wizard.

Figure 6-17 shows the code that was inserted by the *Call a Transaction Business Object* wizard when the ***SORTOI*** business object was selected. This business object is used to create, update, or delete a sales order. Line 6 defines the variables to be used in this code and contains three variable names, whereas the other examples shown so far only contain two. *Transaction* class, *Post* method business objects require two XML strings (referred to as the parameter string and the document string). Lines 8 to 13 build up the parameter XML string in a variable called *XMLParam*, and lines 15 to 33 build up the document XML string in a variable called *XMLDoc*.

Line 35 is where the business object is called, and the results passed to the *XMLOut* variable. *CallTrn* expects to receive the business object name, parameter XML string, document XML string, the method (which is *Post* in this example), and the e.net Solutions instance of *"auto"*. The error handling is as mentioned previously.

\vbscripts\INVPENL1

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function StockCodeDetails_OnRefresh()
6  dim XMLOut, XMLParam, XMLDoc
7
8  XMLParam = XMLParam & " <SalesOrders>"
9  XMLParam = XMLParam & "   <Parameters>"
10 XMLParam = XMLParam & "     <Process>IMPORT</Process>"
11 XMLParam = XMLParam & "     <IgnoreWarnings>N</IgnoreWarnings>"
12 XMLParam = XMLParam & "   </Parameters>"
13 XMLParam = XMLParam & " </SalesOrders>"
14
15 XMLDoc = XMLDoc & " <SalesOrders>"
16 XMLDoc = XMLDoc & "   <Orders>"
17 XMLDoc = XMLDoc & "     <OrderHeader>"
18 XMLDoc = XMLDoc & "       <CustomerPoNumber>xxxxxx</CustomerPoNumber>"
19 XMLDoc = XMLDoc & "       <OrderActionType>A</OrderActionType>"
20 XMLDoc = XMLDoc & "       <Customer>xxxxxx</Customer>"
21 XMLDoc = XMLDoc & "       <Warehouse>E</Warehouse>"
22 XMLDoc = XMLDoc & "       <SpecialInstrs>Handle with care</SpecialInstrs>"
23 XMLDoc = XMLDoc & "     </OrderHeader>"
24 XMLDoc = XMLDoc & "     <OrderDetails>"
25 XMLDoc = XMLDoc & "       <StockLine>"
26 XMLDoc = XMLDoc & "         <CustomerPoLine>1</CustomerPoLine>"
27 XMLDoc = XMLDoc & "         <LineActionType>A</LineActionType>"
28 XMLDoc = XMLDoc & "         <StockCode>xxxxxx</StockCode>"
29 XMLDoc = XMLDoc & "         <OrderQty>xxxxxx</OrderQty>"
30 XMLDoc = XMLDoc & "       </StockLine>"
31 XMLDoc = XMLDoc & "     </OrderDetails>"
32 XMLDoc = XMLDoc & "   </Orders>"
33 XMLDoc = XMLDoc & " </SalesOrders>"
34 on error resume next
35 XMLOut = CallTrn("SORTOI",XMLParam,XMLDoc,"Post","auto")
36 if err then
37   msgbox err.Description, vBCritical, "Calling Business Object"
38   exit function
39 end if
40 ' Switch on error handling
41 on error goto 0
42
43 End Function
```

Figure 6-17: The VBScript code generated by the *Call a Transaction Business Object* wizard

Call Business Object – Setup (Add) / Setup (Update) / Setup (Delete)
The *Call Business Object* button has three options to cater for *Setup* class business objects, one for each method. The three wizards that are called by these options contain the same three panes as the *Call a Transaction Business Object* wizard, and they are used in the same way. When the *Insert VBScript* button is selected the only difference is the line that calls the business object. This line uses *CallSetup* and expects the business object name, parameter XML string, document XML string, the method, and the e.net Solutions instance of *"auto"*. The methods are *Add*, *Update*, and *Delete*. Below are examples of the lines to call business objects for each of these methods.

```
XMLOut = CallSetup("INVSST",XMLParam,XMLDoc,"Add","auto")

XMLOut = CallSetup("INVSST",XMLParam,XMLDoc,"Update","auto")

XMLOut = CallSetup("INVSST",XMLParam,XMLDoc,"Delete","auto")
```

SYSPRO Instance of "auto"
When SYSPRO is run, it knows the location of all of its components because it is started from a shortcut, or from an executable in its `Base` folder. When e.net Solutions is called it needs to know where all of its components reside, as there is no shortcut or executable from which it was run. The Windows registry contains an *e.net solutions* registry key to facilitate this. On 64-bit machines the e.net Solutions registry key is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\SYSPRO\e.net solutions
```

On 32-bit machines the *e.net solutions* registry key is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SYSPRO\e.net solutions
```

Within this key is a string value called *BaseDir*, which is the default entry that is used by e.net Solutions to find the SYSPRO `Base` folder. Once this is known, it uses the `IMPWRK.INI` file in this folder to locate the SYSPRO `Work` folder, and then uses the `IMPACT.INI` configuration file within the `Work` folder to locate all of the SYSPRO components.

In a test environment (or when a later version of SYSPRO is installed in parallel to an existing version) there can be a requirement to have multiple instances of e.net Solutions on one machine. To facilitate this you can specify up to 10 different base folders against the e.net solutions entry in the registry. These are named *BaseDir*, *BaseDir1*, *BaseDir2*, up to *BaseDir9*. **Figure 6-18** shows an e.net registry key that contains seven *BaseDir* entries.

If you are logging on to e.net Solutions from outside of SYSPRO, one of the parameters that must be supplied is the SYSPRO instance.

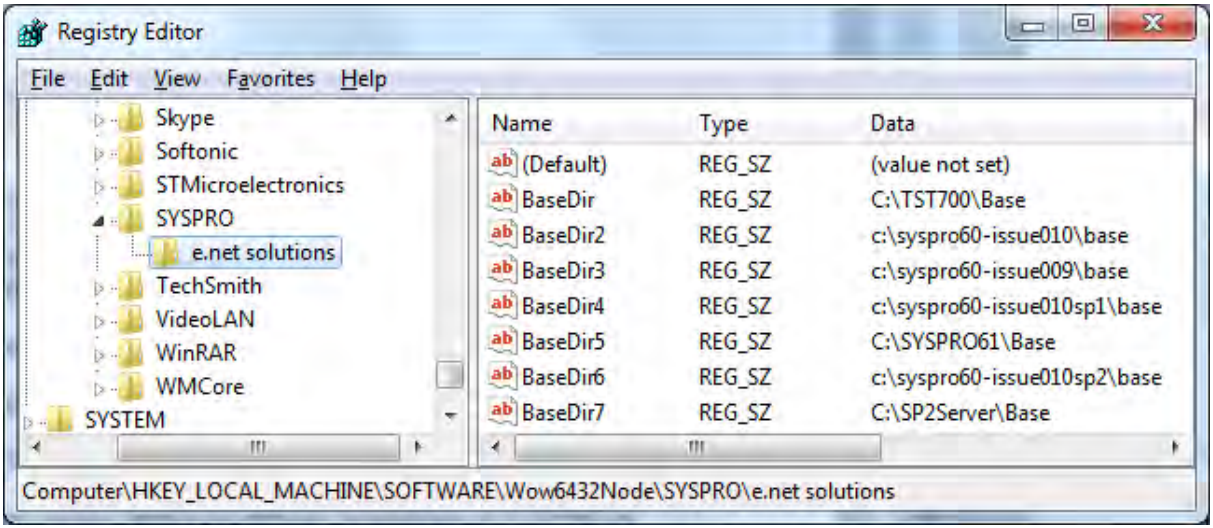


Figure 6-18: The *e.net solutions* registry key containing all the *BaseDir* entries

The *SYSPRO Instance* is the second to last item passed to the *Logon* method during the logon process. Below is an excerpt from a VBScript that performs a logon to e.net Solutions from outside of SYSPRO. The third line is the one that performs the logon, and the *Instance* contains the value of 2. Instance 2 will use the *BaseDir2* entry to look up the SYSPRO `Base` directory.

```
Dim uid, Syspro
Set Syspro = createobject("Encore.Utilities")
uid = Syspro.Logon("ADMIN","OPPASS" ,"S" ,"PASS", "00", 0, 2,"")
```

Once e.net Solutions knows which *BaseDir* entry to use it can access the `IMPWRK.INI` file to find the SYSPRO `Work` folder, and use the `IMPACT.INI` file within this folder to find all the other components that it needs.

The *Call Business Object* wizard is used to build the code to call a business object from within SYSPRO using VBScript. This uses the SYSPRO architecture to call the business object, so you do not need to set up DCOM, Web Services, or the SYSPRO WCF Service to access the business object (which resides in the `Program` folder on the SYSPRO application server).

As you are already logged into SYSPRO, you do not need to logon to e.net Solutions to invoke a business object, as this is handled for you. The code that is built by the *Call Business Object* wizard uses either the *CallBO*, *CallTrn*, or *CallSetup* methods, depending on the business object to be called. Below is a line of code that was created using this wizard, and uses *CallBO* to call the *Customer Query* business object.

```
XMLOut = CallBO("ARSQRY",XMLParam,"auto")
```

The last parameter in this line of code is an e.net Solutions instance of "*auto*". This specifies that SYSPRO must use its `Base` folder (which it knows) to locate the matching *BaseDir* entry in the registry, and then use this instance number associated with this for e.net Solutions. Note that there must be a matching registry entry for this to work.

As SYSPRO has worked out the instance number you do not need to supply it when calling the business object. It also means that your code can be used at other sites without having to make any changes. This is particularly useful for developers who can write code on their machine, export it, and have it imported by a System Administrator without any tweaking.

In theory, the Instance of "*auto*" can be replaced with an instance number if so required, and an example using the instance number of 2 appears below. However, there is no direct benefit to this, and may cause issues at a later date if the sequence of the *BaseDir* entries in the registry are changed by the Administrator.

```
XMLOut = CallBO("ARSQRY",XMLParam,2)
```

Syntax Check

The *Syntax Check* button looks through all the code in the current VBScript and attempts to find any issues with the syntax of the commands. It will find items such as defining a variable name for the second time, any *IF* statements that aren't nested properly, issues with incorrect use of quotation marks, etc.

What it won't find are logic problems; such as calling SQL Server with the wrong command string or, calling a business object using the wrong class/method. These types of issues will only show up when the script is used.

The *Syntax Check* button performs the same checks as happen when you exit back to the *VBScript Editor* screen (which is also the same as is done before the script runs). An example of a syntax error appears in **Figure 6-19**. This was caused by an *IF* statement not being completed before the *End Function* statement. Looking at the code there are actually two *IF* statements that are not completed before the *End Function* statement, one on line 19, and one on line 20. The error is *Description: Expected If*, and is reported on line 23. This is because an *IF* statement is completed using an *End IF* statement, and the script has read the *End* part of *End Function* and is reporting that the next word should be *If*, not *Function*.

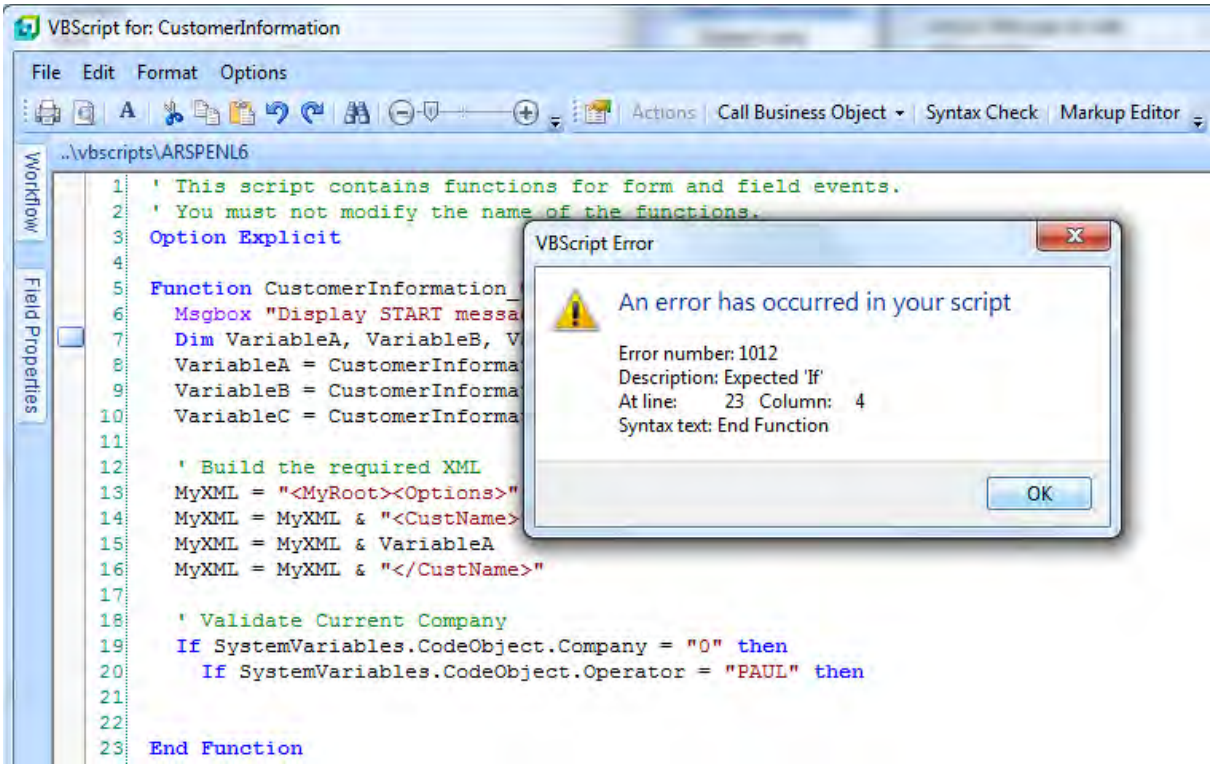


Figure 6-19: A syntax error caused by not ending an *IF* statement.

Markup Editor

The *Markup Editor* button calls the *XAML Markup Editor* program. XAML is an abbreviation of *eXtensible Application Markup Language* and is a display markup language that is used to display information in different formats. XAML and the *Markup Editor* will be covered in detail in a later chapter.

There are over 30 XAML markup templates that ship with SYSPRO, and these can be applied to listview columns and customized panes. However, each template expects to receive a certain number of data items as parameters, and if insufficient are supplied, or they are of the wrong type, the display might not work as expected.

Figure 6-20 shows an associated pane where two columns have XAML associated with them. The warehouse column (*Wh*) uses XAML to display both the warehouse name and warehouse description in one column, saving space. This uses the *Address Plain* XAML template that will accept and display between one and seven parameters.

The *Sales History (Quantity)* column is using XAML to display the contents of the 12 columns that appear immediately after it in a graph (known as a *trendline*). By default only the Wh and Sales History columns are displayed in this listview, with the columns containing the values only being displayed if the operator clicks on the button that shows these columns. They are shown here so that you can see some of the values.

Displaying the data this way saves both space and makes the information more easily understood for those who prefer visual information. This *trendline* uses the *TLLineChart12* XAML template that expects to receive 13 pieces of data to plot the graph, so you should not apply this template to a column that only contains one value.

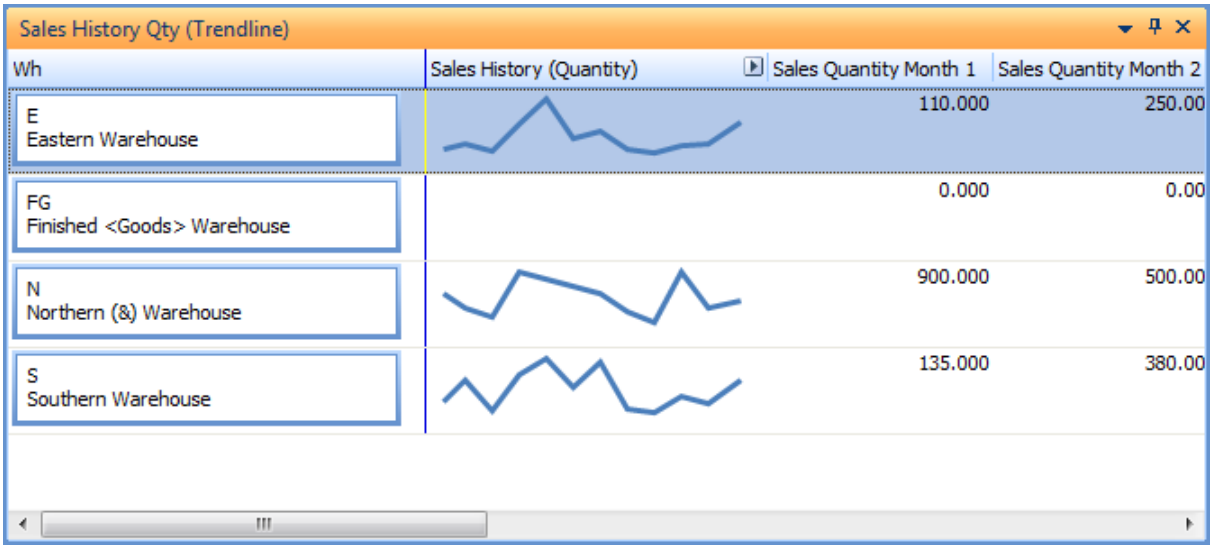


Figure 6-20: An associated pane with two columns displayed using XAML markup

Script Area

The *Script Area* is the main section of the *VBScript for:* screen that contains the code that you enter. It is the pane that has the title *..\vbscripts\XXXXXXXX* (where *XXXXXXX* is replaced by the name of the script). An example can be seen in **Figure 6-17**. If the option *Show Line Numbering* is checked the *Script Area* will also contain the line numbers.

This section typically starts with three lines that are created by SYSPRO when you edit the VBScript for the first time. These lines appear below. The first two lines are comment lines and are not required. The third line specifies that all variables must be declared before they are used. If you attempt to use a variable without declaring it, and this line is present, you will receive an error message.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit
```

After these lines your script will typically be made up of functions that are called when their matching events fire. An exception to this is when loading one of the *VBScript Modules* (which will be covered later in this chapter) where the statement to do this does not appear within a function.

Variables

The *Variables* pane contains entries for each pane that is displayed in the program from which you loaded the *VBScript Editor*, along with two for the *System Variables*. **Figure 6-21** shows the *Variables* pane for the *Customer Query* program in its default mode where all the entries are collapsed.

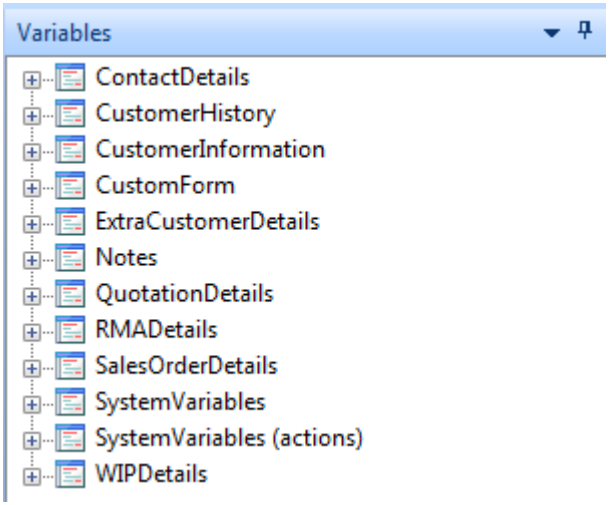


Figure 6-21: The *Variables* pane containing the list of panes, and the two *System Variables*

Each of the entries for the panes can be expanded to show the fields within this pane. **Figure 6-22** shows the entry relating to the *Customer Information* pane (*CustomerInformation*) expanded so that the fields are visible.

Variable names cannot contain spaces so the spaces are removed from the names of both the forms and the fields. To conform to the requirements of the XML standards (where element names cannot start with a numeric character), the 30 Days, 60 Days, and 90 Days fields have also been prefixed with the letter “a”.

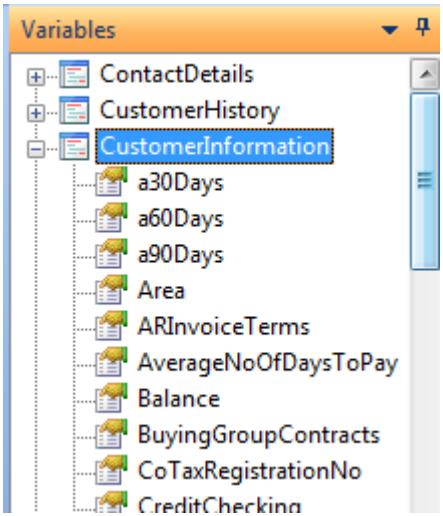


Figure 6-22: The entry for the *Customer Information* pane expanded to show the fields

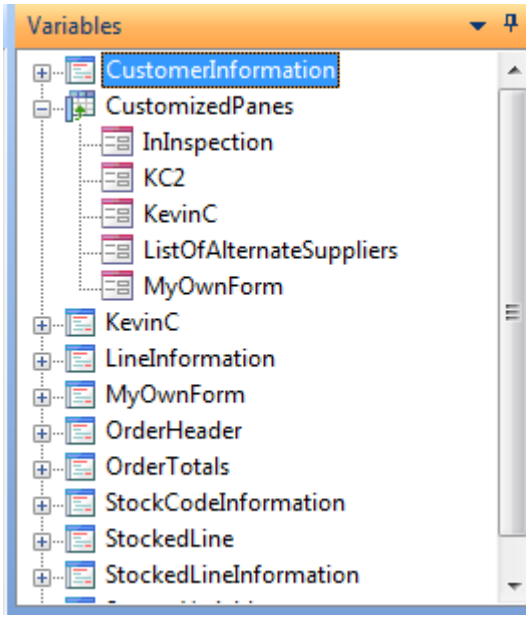


Figure 6-23: The list of customized panes in the *CustomizedPanes* section

If a program contains *Form*, *Search Window*, or *Listview* customized panes, they will appear within a *CustomizedPanes* section (see **Figure 6-23**). This section will also contain any *Associated Panes*.

If the program contains a *Custom Forms* pane, there will be a section for this in the *Variables* pane.

Figure 6-24 shows the *Variables* pane that is displayed when in the *Sales Order Query* program. In this example there are two custom forms, one for the sales order header, and one for the sales order details

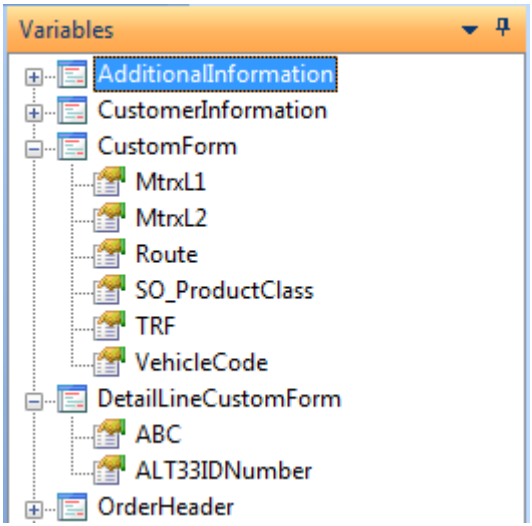


Figure 6-24: Two *Custom Form* panes configured for sales orders, one header, and one detail line

System Variables

Prior to SYSPRO 7, all the system variables were held within a section called *System Variables*. From SYSPRO 7 these were separated into those that are property variables (that just contain information), and those that will prompt for additional information. The latter now exist under a section called *System Variables (actions)* which is covered below. The *System Variables* and *System Variables (actions)* are common to the *Variables* pane for all programs.

The majority of the variable names that exist under the *System Variables* section are read-only variables that contain information about the current environment at run time, such as the company ID, system date, operator code, program name, default warehouse, etc. The exceptions to this are the *Global Variables* that can store user information at run time, and *Cancel Standard Action* which prevents standard program actions from happening.

Global Variables

Each script and each function within a script exists in isolation. They do not know about each other and cannot communicate between themselves. For example, even though the *Customer Information* form's *OnLoad* and *OnRefresh* functions are run straight after each other, when the *Customer Query* is

supplied a customer number for the first time, they have no knowledge of each other and cannot communicate.

However, there is a way of storing information that is available between functions and scripts, and this is called a global variable. There are four global variables available within the *System Variables* section (*GlobalVariable1* to *GlobalVariable4*). They allow you to store up to 256 characters of information each, and they retain this information until they are cleared out programmatically, their values overwritten, or the operator logs out of SYSPRO. Using the example above, placing a value in *GlobalVariable1* during this *OnLoad* event makes it available for all functions in all programs, not just the *OnRefresh* function of this script.

Care must be exercised that you do not overwrite the contents of a global variable that you still need (either in this program, or another), and that you do not consume the global variables contents when you are not certain that it contains the value that you need (i.e. that the same global variable has not been overwritten by the VBScript in another program).

Cancel Standard Action

Within an entry form there may be a scenario where you want to change a value on the screen but when you make this change, the application overrides your change. To get around this the *CancelStandardAction* variable can be set to *true* against this field's *OnAfterChange* event, so when the value is changed the form will ignore any built-in actions.

In this example, the option *Show description of keys in form caption* has been checked on the *Forms* tab of the *Customize* program. **Figure 6-25** shows an example of this where the *Ordered By:* field's label also contains the description of *Order placed by telephone* that matches the field's value of *T*. If the value is changed to *S*, the system would look up the description that matches this value and change the displayed description to *Order given to Salesperson*.

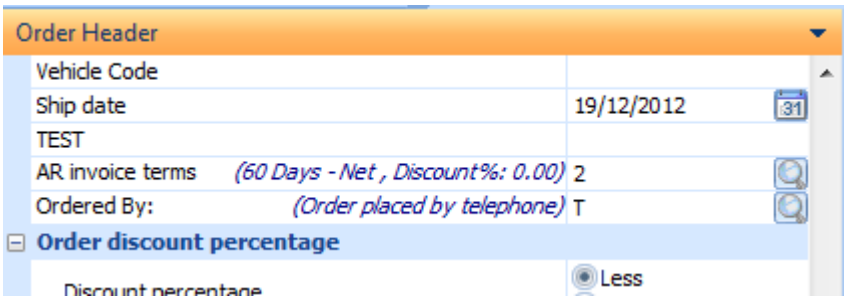


Figure 6-25: The description against the *Ordered By: field*

However, if against the *OrderType_OnAfterChange* function the *CancelStandardAction* system variable is set to *true*, when the value is changed to *S* the description will remain as *Order placed by telephone*. This code appears below.

```
Function OrderType_OnAfterChange()
  SystemVariables.CodeObject.CancelStandardAction = true
End Function
```

ScriptName
The *ScriptName* variable is a read-only variable that returns the name of the script being run. The script name is the same as the name of the pane from which it is being run, and is made up of the six character program name plus two characters that identify this pane within the program. For example, the program name of the *Customer Query* program is ***ARSPEN***. The *Customer Information* pane is identified within the program using the two characters ***L6***, so the pane's name is ***ARSPENL6***, as is the script name.

ScriptFunction
The *ScriptFunction* variable is a read-only variable that returns the name of the function being executed. Using the same example as in the *ScriptName* section above, if the function being executed was the script's *OnRefresh* function, the *ScriptFunction* variable would contain *customerinformation_onrefresh*.

PostChangeEvent
This variable is a write-only variable that is used to flag the current form as having changed. This means that if it has a *Save* button associated with it, the *Save* button will become enabled.

System Variables (actions)
The system variables that exist under the *System Variables (actions)* section all display an additional screen prompting for information. These are *ActionToInvoke*, *SYSPROBrowseToRun*, *SYSPROProgramToRun*, and *ToolbarButton*.

ActionToInvoke
The *ActionToInvoke* variable loads the same screen as the *Actions* button from the toolbar of the *VBScript for:* screen. If *Actions* are available the same screen will be displayed as if you had clicked on the *Actions* button. An example of this can be seen in **Figure 6-10**.

The *Actions* button on the toolbar is only enabled if there are *Actions* available for this pane. However, the *ActionToInvoke* variable name is always present, so if you attempt to use this and there are no *Actions* available for this pane, a message is displayed to this effect.

SYSPROBrowseToRun
If you double-click on the *SYSPROBrowseToRun* variable name the *Run SYSPRO Browse Program* screen is displayed. This contains an entry field for the name of the browse, a browse button that displays a list of SYSPRO modules and programs, and an *Insert VBScript* button. If you know the name of the browse you can enter it at the prompt, otherwise use the browse functionality to locate the name of the required browse program. The *Insert VBScript* button will insert the required code for you.

```
82
83  Function JustButton_OnButtonClick()
84    SystemVariables.CodeObject.SYSPROBrowseToRun = "INVBST"
85  End Function
```

Figure 6-26: The code to call the *Stock Code Browse*

Line 84 of **Figure 6-26** shows the code that will call the *Inventory Stock Code Browse* when a button associated with the *JustButton* field is clicked. The code will call the browse and start at the beginning. If the *JustButton* field already contains a value this will be passed to the browse and the displayed list will start from this value. If an item is selected in the called browse its value will be returned, and this field will be populated with it.

SYSPROProgramToRun

Adding the code associated with the *SYSPROProgramToRun* variable is similar to the *SYSPROBrowseToRun* variable covered above. When double-clicked it displays the same screen with the same options, but with the title *Run SYSPRO Program*. However, the *SYSPROProgramToRun* code can accept a variable that is passed to the program that is being called, and this is delimited with a space character. This only calls the program and passes this variable; it cannot return anything back to the field. This is probably best explained with an example.

A new button is added to the *Sales Order Query* toolbar, and a VBScript function called *MaintSO* added to it. Against the *Macro for: SORPENTB* option a function is created as appears in **Figure 6-27**. This function states that when the button is clicked, *SYSPROProgramToRun* must be invoked with the *Sales Order Entry* program name of ***IMP040***, followed by the contents of the *Sales Order* field on the *Order Header* form (which is present in the *OrderHeader.CodeObject.SalesOrder* variable). When the *Sales Order Entry* program is called this way it is put into maintenance mode automatically, and expects to receive a sales order number to be processed as a parameter. The sales order is opened and the cursor placed on the first entry field on the primary form.

```
Function MaintSO()
  SystemVariables.CodeObject.SYSPROProgramToRun = "IMP040 " & OrderHeader.CodeObject.SalesOrder
End Function
```

Figure 6-27: Calling the *Sales Order Maintenance* program and passing the sales order number

ToolbarButton

When selected, the *ToolbarButton* variable calls up the *Modify Toolbar Buttons* screen. This screen contains options to enable/disable a button, make a button visible/invisible, make a button's checkbox checked, cause the action against a button to be executed, change the value of a button that accepts input, or any combination of these options.

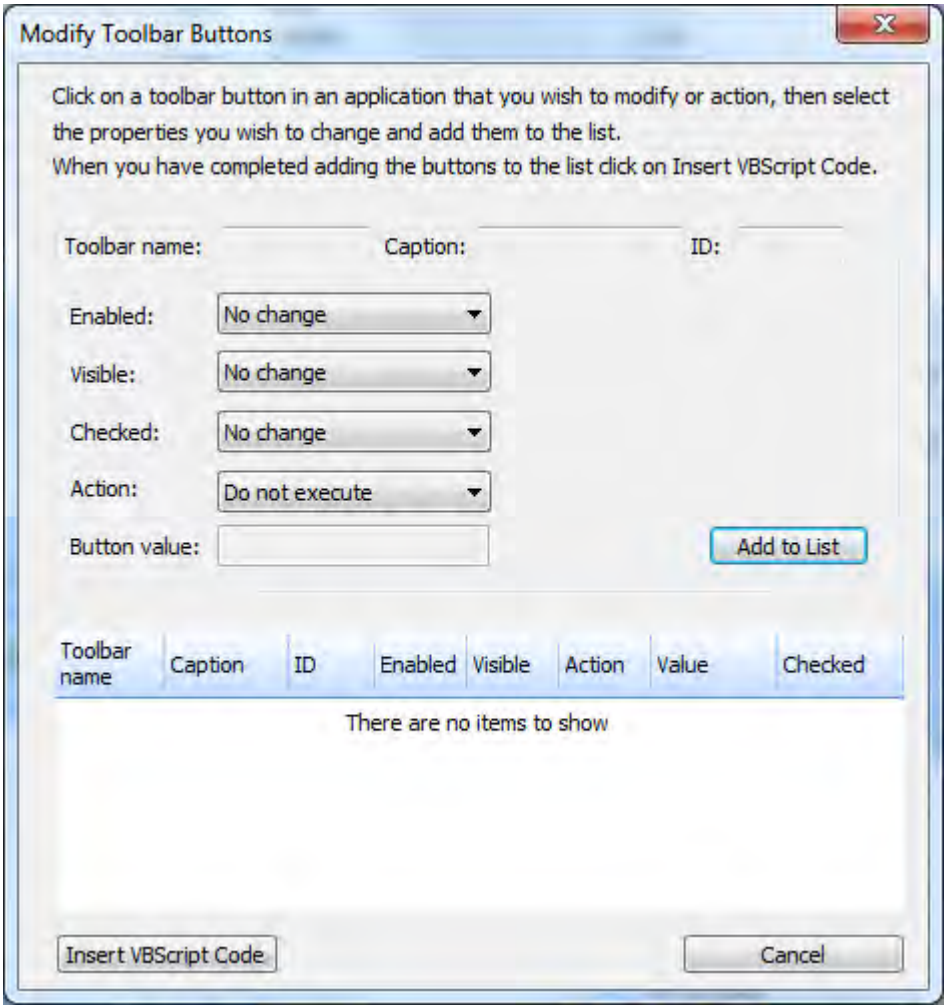


Figure 6-28: The *Modify Toolbar Buttons* screen in its default state

Figure 6-28 shows the *Modify Toolbar Buttons* screen in its default state, just after the variable name has been selected. At this time the VBScript editing screen disappears from behind the *Modify Toolbar Buttons* screen, leaving it in front of the program from which the VBScript editor was called. This happens so that you can select the button to be modified by clicking on it. When the button is selected the button's ID and other details are passed to the *Modify Toolbar Buttons* screen and used to populate the *Toolbar name*, *Caption*, and *ID* fields.

Figure 6-29 shows a portion of the screen after the button to create a sales order has been selected within the *Sales Order Entry* program. The prompts for *Toolbar name* (*IMP040T1*), *Caption* (*Crea&te*

Order), and *ID* (39000) are all populated automatically on the *Modify Toolbar Buttons* screen when the button is selected.

The button to be selected has to be enabled at the time it is selected; otherwise the details will not be taken across to the *Modify Toolbar Buttons* screen. After clicking on the required button, if you click on another button on the main program before making your selections and clicking on the *Add to List* button, the button's details at the top of the *Modify Toolbar Buttons* screen will be overwritten with those of the new button.

Selecting *Execute* from the dropdown list against the *Action* prompt and clicking the *Add to List* button will add an entry to the listview at the bottom of this screen. (Note: all the normal SYSPRO listview functionality is available within this listview). At this point the prompts that were prepopulated when you clicked on the original program's button are cleared, and the other items are reset to their default values. Multiple options can be selected before clicking on the *Add to List* button.

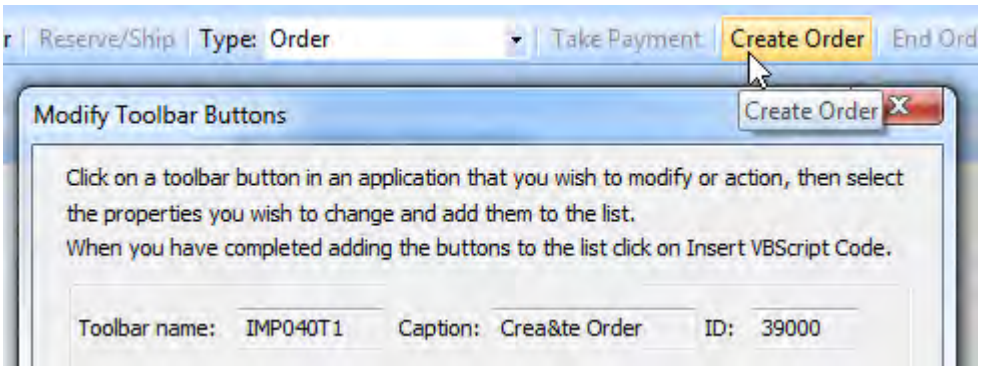


Figure 6-29: Selecting button and the prompts being populated on the *Modify Toolbar Buttons* screen

While this screen is displayed, other buttons can be selected from the main program. The same process is followed to add them to the listview (see **Figure 6-30**). Once all the required entries exist in the listview, clicking on the *Insert VBScript Code* button creates the code to perform all of the button changes that appear in this listview. This code is added to your VBScript at the place within the function where your cursor resided before clicking on the *ToolbarButton* variable name.

The code that is added to the script is the full name of the *ToolbarButton* variable followed by an equal sign and XML containing the button information. The XML is split over multiple lines to make it easier to follow/modify. At the end of each line (except the last one) there is an ampersand (&) and an underscore (_) that tells the script that the code continues on the next line.

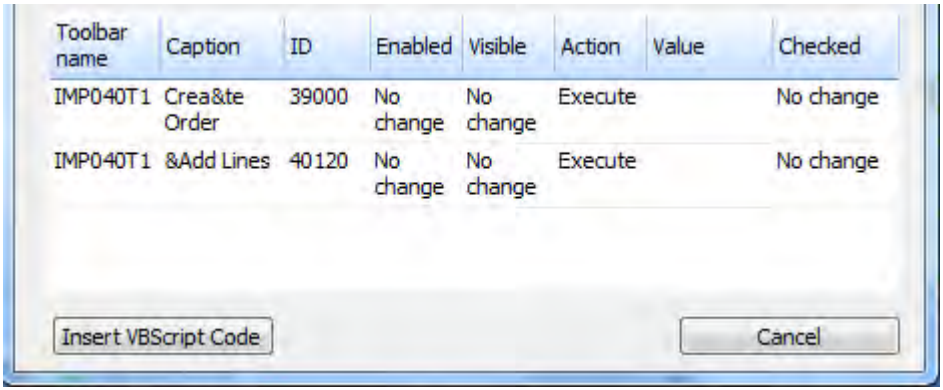


Figure 6-30: Multiple entries added to the listview

The following example is the code to execute the *Create Order* button on the toolbar of the *Sales Order Entry* program. It contains the toolbar name, button ID, and the button's caption. The first two are required, whereas the last one is present so that you know to which button it refers. You can change the content of the *Caption* attribute to be more meaningful to you, as it is not used by SYSPRO.

```
SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
"<Toolbar Name='IMP040T1' Id='39000' Action='Execute' Caption='Create Order'/>" & _
"</Toolbars>"
```

The following is an example of where you can use the *ToolbarButton* variable. In this example you have your biggest customer who always starts their customer purchase order number with the string *BigCo*. When they phone, the telesales person knows that this is a real order and must be created (as opposed to some people that start the order, then exit when the price of the first line is displayed, so the sales order never gets created within SYSPRO). The customer needs the sales order number as soon as possible to enter into their system.

SYSPRO's default behavior is to only create the sales order once the header has been completed and the operator saves the first detail line. At this point the sales order number is generated if automatic sales order numbering is configured. After the customer account code has been added, and before the first detail line is complete, there is a *Create Order* button on the toolbar that, if clicked, will create the order and generate the sales order number.

Because you know that this customer will always place the order, once you have the customer's purchase order number that begins with *BigCo*, you can automate the clicking of the *Create Order* button which will generate the sales order number.

To add the code within the *Sales Order Entry* program you need to get SYSPRO to the point where the button is available. In this case that means adding the customer code, and tabbing off the field. At this point, right-click on the label of the *Customer Purchase Order* field and select *Macro for:IMP040L3*

from the context-sensitive menu. On the *VBScript Editor* screen highlight the *Field event* called *OnAfterChange*, and click on the *Edit VBScript* button on the toolbar. If it did not already exist, the *CustomerPurchaseOrder_OnAfterChange* function will be added for you and the cursor placed within it.

On the *Variables* pane, expand the *SystemVariables (actions)* section and double-click on the *ToolbarButton* variable name. The *Modify Toolbar Button* screen is displayed and the screen where you modify the script will disappear. Click on the *Create Order* button and the *Toolbar name* prompt will be populated with ***IMP040T1***, the *Caption* prompt will be populated with *Crea&te Order*, and the *ID* prompt will be populated with *39000*. Use the dropdown list alongside the *Action* prompt to select *Execute*, and click on the *Add to List* button. This will be added to the listview at the bottom of the screen (see **Figure 6-31**).

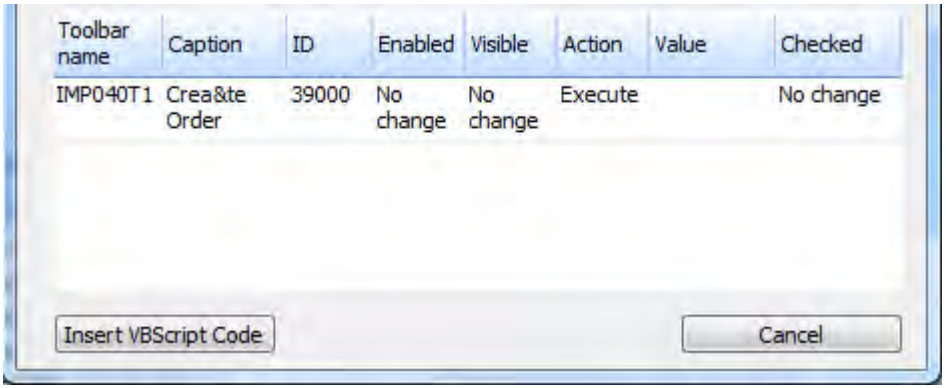


Figure 6-31: The listview containing the *Execute* action for the *Create Order* button

Click on the *Insert VBScript Code* button and the code will be added to the *CustomerPurchaseOrder_OnAfterChange* function (see **Figure 6-32**).

..\vbscripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerPurchaseOrder_OnAfterChange()
6    SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
7    "<Toolbar Name='IMP040T1' Id='39000' Action='Execute' Caption='Create Order'/>" & _
8    "</Toolbars>"
9  End Function
```

Figure 6-32: The script to execute the *Create Order* button

Although the code now exists to execute the *Create Order* button, this will happen every time that the *Customer Purchase Order* value is changed, which is not the required behavior. Code needs to be added to the function so that the button is only executed when the customer purchase order number starts with *BigCo*.

Add a new line between the line *Function CustomerPurchaseOrder_OnAfterChange* and the line that starts defining how the button will act. On this new line add the following code to create a variable called *FirstFive*. This variable will be populated with the first five characters of the customer purchase order number, converted to uppercase for comparison purposes.

```
Dim FirstFive
```

Below this line, code needs to be added to populate this variable. Type in the variable name, followed by a space character, and equals sign, and another space character. Immediately after this add the word *Left*. "Left" refers to a function within VBScript where you specify a variable name and a number. The function returns a string of characters from this variable, starting from the left, for the specified number of characters.

Continuing with the code, immediately after the word *Left*, add an opening parenthesis. Within the *Variables* pane, expand the *OrderHeader* section and double-click on the *CustomerPurchaseOrder* variable name. The full name of this variable will be added to your code. After the variable name, add a comma, the number *5*, and the closing parenthesis. This line will take the first five characters of the customer purchase order field and use it to populate the *FirstFive* variable.

Add a new line directly below the one that you just created. The line that you are adding will be used to convert the contents of the *FirstFive* variable to uppercase for comparison purposes. This will cater for the operator entering *BigCo* in any combination of upper or lowercase characters.

Start by entering the *FirstFive* variable name, and follow this with a space character, an equal sign, and another space character. Add the text *UCase*, which refers to a VBScript function to convert a string (or content of a variable) to uppercase. This is followed by an opening parenthesis, the *FirstFive* variable name and a closing parenthesis.

At this point the *FirstFive* variable contains the first five characters of the customer purchase order, converted to uppercase. You need to add an *IF* statement around the section that executes the button, so that the button only gets executed if this variable contains the text *BIGCO*.

To add this *IF* statement, create a new line between the line containing the *UCase* function, and the line specifying the *ToolbarButton* variable. Start the line with the word *If* followed by a space character, and the variable name *FirstFive*. Follow this with a space character, and equal sign, another space character, a double-quote, the text *BIGCO* (which must be in uppercase), another double-quote, a space character, and the word *Then*. Add another new line directly after the end of the toolbar button

XML (which should end with the text “</Toolbars>”). Add the text *End If* to close off your *IF* statement. Your code should match that in **Figure 6-33**.

..\vbscripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerPurchaseOrder_OnAfterChange()
6    Dim FirstFive
7    FirstFive = Left(OrderHeader.CodeObject.CustomerPurchaseOrder,5)
8    FirstFive = UCase(FirstFive)
9
10   If FirstFive = "BIGCO" then
11     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
12     "<Toolbar Name='IMP040T1' Id='39000' Action='Execute' Caption='Create Order'/>" & _
13     "</Toolbars>"
14   End If
15
16 End Function
```

Figure 6-33: Code to execute the *Create Order* button depending on the Customer’s PO number

Exit the screen where you edited the VBScript, back to the *VBScript Editor* screen, and use the *Save* button to save your code. Exit the *Sales Order Entry* program. The next time that this program is used, when the operator types in a customer purchase order beginning with *BigCo* and tabs off this field, it will be as if the operator had clicked on the *Create Order* button.

You can have multiple toolbar button actions being actioned at the same time. If this example was changed to click on both the *Create Order* and *Add Lines* buttons the only difference to the code would be the XML below. The *Modify Toolbar Buttons* screen would look the same as **Figure 6-30** before the *Insert VBScript Code* button is clicked.

```
SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
"<Toolbar Name='IMP040T1' Id='39000' Action='Execute' Caption='Create Order'/>" & _
"<Toolbar Name='IMP040T1' Id='40120' Action='Execute' Caption='Add Lines'/>" & _
"</Toolbars>"
```

With this updated XML, when the operator adds a customer purchase order number beginning with *BigCo* the sales order is created, the sales order number is generated automatically, the *Order Line* form is activated and the cursor place against the *Stock code* field.

Another option within the *Modify Toolbar Button* screen is to check/uncheck a checkbox against a button. This example changes the *Reset Credit Status* checkbox in the *AR Invoice Posting* program depending on which customer’s information is being posted (see **Figure 6-34**).

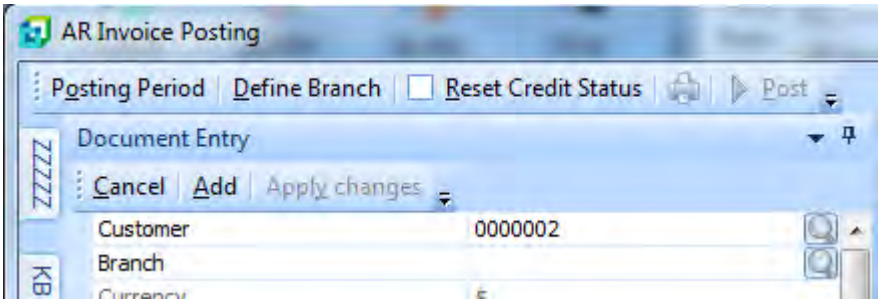


Figure 6-34: The *Reset Credit Status* checkbox unchecked

Call up the *AR Invoice Posting* program (***ARSPIN***) and right-click on the *Customer* field's label. Select *Macro for:ARSPINL1* from the context-sensitive menu. On the *VBScript Editor* screen, highlight the *OnAfterChange* Field event and click on the *Edit VBScript* button. The code is added to this event because you want to check the contents of the *Customer* field when it is changed, and set the checkbox accordingly. If it did not already exist, the *Customer_OnAfterChange* function will be created for you, and the cursor placed within it.

Within this function, add an *IF* statement to check the contents of the *Customer* field. This is done by typing *IF*, followed by a space character. Locate the *DocumentPostingEntry* section of the *Variables* pane and expand it. Locate the *Customer* variable and double-click on it. The full name of the *Customer* variable will be added to your code. Immediately after this add a space character, an equal sign, and another space character. Add the customer account code surrounded by double-quotes, and follow this with a space character and the word *Then*.

Add a new line directly below the *IF* statement, locate the *SystemVariables (actions)* section on the *Variables* pane and expand it. Double-click on the *ToolbarButton* variable name. The *Modify Toolbar Button* screen will appear and the screen where you were adding the VBScript will disappear. On the *AR Invoice Posting* screen, locate the *Reset Credit Status* button and click on it. On the *Modify Toolbar Button* screen the *Toolbar* prompt will be populated with *ARSPINTB*, the *Caption* with *&Reset Credit Status*, and the *ID* with *01021*. From the dropdown list against the *Checked* prompt, select *True* and click the *Add to List* button to add this to the listview. As this is the only button that needs to be modified, click on the *Insert VBScript Code* button to add this setting to your code.

Back on the screen where you are adding your script, add a new line directly underneath the one that closes the *Toolbars* element. This line should contain the word *ELSE*, as everything that appears after this is what should happen if the customer code does not match the one specified. Add a new line directly under the one containing *Else*.

With the cursor placed on the new line, locate the *ToolbarButton* system variable again and double-click it. The *Modify Toolbar Button* screen should be displayed and the screen containing the VBScript should disappear, just as before. Click on the *Reset Credit Status* button again, select *False* from the dropdown list against the *Checked* prompt, and click on the *Add to List* button. Click on the *Add VBScript Code* button to add this setting to your code.

Back on the screen where you are adding your script, add a new line directly underneath the one that closes the *Toolbars* element. This line should contain the word *End If* to finish off your code. Your code should be similar to that in **Figure 6-35**. Exit the VBScript editing screen and click *Save* to save your script. Exit the *AR Invoice Posting* program. The next time that a customer code is entered within this program, a check will be made, and the checkbox set accordingly.

```
5  Function Customer_OnAfterChange()
6    If DocumentPostingEntry.CodeObject.Customer = "000000000000001" then
7      SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
8      "<Toolbar Name='ARSPINTB' Id='01021' Checked='True' Caption='Reset Credit Status'/>" & _
9      "</Toolbars>"
10   Else
11     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
12     "<Toolbar Name='ARSPINTB' Id='01021' Checked='False' Caption='Reset Credit Status'/>" & _
13     "</Toolbars>"
14   End If
15 End Function
```

Figure 6-35: The code to check/uncheck the *Reset Credit Status* checkbox

..\vbscripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerPurchaseOrder_OnAfterChange()
6    Dim FirstFive
7    FirstFive = Left(OrderHeader.CodeObject.CustomerPurchaseOrder,5)
8    FirstFive = UCase(FirstFive)
9
10   If FirstFive = "BIGCO" then
11     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
12     "<Toolbar Name='IMP040TB' Id='40001' Enabled='False' Caption='Non-stocked'/>" & _
13     "</Toolbars>"
14   Else
15     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
16     "<Toolbar Name='IMP040TB' Id='40001' Enabled='True' Caption='Non-stocked'/>" & _
17     "</Toolbars>"
18   End If
19 End Function
```

Figure 6-36: The code to enable/disable the *Non-stocked* button in *Sales Order Entry*

Following on from the *BigCo* example above, you have been informed by BigCo's Financial Director that you are not to allow their staff to order non-stocked items. As the selecting of non-stocked items is controlled using a button, you can use similar logic to disable this button when the customer purchase order is *BigCo*. The code to do this appears in **Figure 6-36**.

The process of creating this is the same as the other *BigCo* example, but instead of using the *Execute* option against the button you set the *Enabled* option to *False*. Make sure that the code sets the *Enabled* option back to true if this sales order is not for *BigCo*. **Figure 6-37** shows the *Non-stocked* button after it has been disabled.

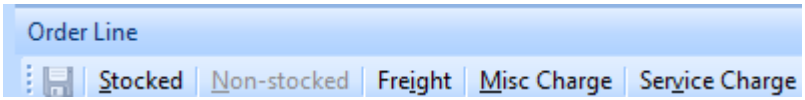


Figure 6-37: The disabled *Non-stocked* button

Alternatively, you could remove the button from the toolbar altogether by setting the *Visible* option to *False*, and back to *True* if this order is not for *BigCo*. **Figure 6-38** shows the same toolbar as appears in **Figure 6-37**, however instead of the *Non-stocked* line being disabled it is invisible.

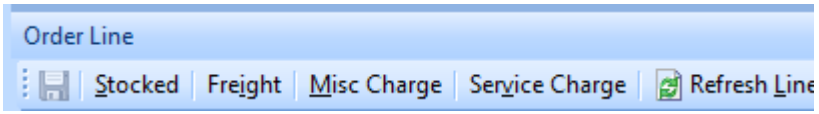


Figure 6-38: The *Non-stocked* button does not appear as it has been set to be invisible

.

..\vbscripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerPurchaseOrder_OnAfterChange()
6    Dim FirstFive
7    FirstFive = Left(OrderHeader.CodeObject.CustomerPurchaseOrder,5)
8    FirstFive = UCase(FirstFive)
9
10   If FirstFive = "BIGCO" then
11     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
12     "<Toolbar Name='IMP040TB' Id='40001' Visible='False' Caption='Non-stocked'/>" & _
13     "</Toolbars>"
14   Else
15     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
16     "<Toolbar Name='IMP040TB' Id='40001' Visible='True' Caption='Non-stocked'/>" & _
17     "</Toolbars>"
18   End If
19 End Function
```

Figure 6-39: The code to make the *Non-stocked* button visible/invisible

Figure 6-39 shows the same code as in **Figure 6-36** except that the button is set to be invisible instead of disabled

The final option on the *Modify Toolbar Buttons* screen is *Button value*, and this is used to populate a button that accepts input from the operator, such as the *Customer* prompt on the *Customer Query* toolbar. The *Button value* option only becomes enabled after you have selected a button that can accept input. This is best explained with an example.

Your operators have stated that when viewing a customer account that is a sub-account in the *Customer Query*, they need to be able to click on something to display the details of the master account. This is possible using a *Form Action* (covered in Chapter 5), the ***ARSQRY*** business object, a little VBScript code, and the *Button value* option.

The first step is to call up the *AR Customer Query* program and add a *Form Action*. Right-click on one of the captions on the *Customer Information* pane, move your mouse pointer over the *Insert Form Action* option on the context-sensitive menu, type in the *Form Action* name of *Go to Master Account* and press *Enter* (see **Figure 6-40**).

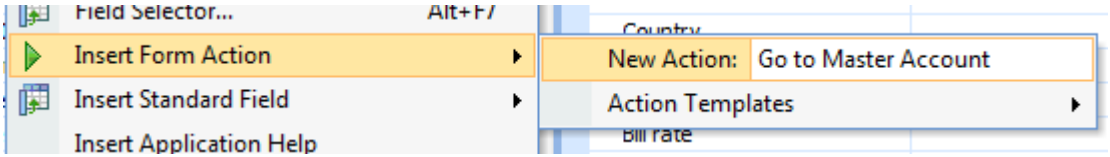


Figure 6-40: Adding the *Form Action*

The *Go to Master Account* form action will appear at the bottom of the *Customer Information* pane, and the *VBScript Editor* screen will appear with an *Action event* of *OnGoToMasterAccount*. Double-click on this event name and the *CustomerInformation_OnGoToMasterAccount* function will be created for you, and the cursor placed within it.

As you only want this code to be run if this is a sub-account you will use an *IF* statement to check the contents of the *Master/sub-account* field (using the *MastersubAccount* variable) to see if it contains the text *This is a sub account*. Start by typing *If* followed by a space character. Then locate the *MastersubAccount* variable name under the *CustomerInformation* section of the *Variables* pane, and double-click it. This will insert the full name of this variable. Follow this with a space character, an equal sign, another space character and *"This is a sub account"* (including the quotation marks), another space character and the word *Then*. When you have finished the line should match that below.

```
If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
```

Add a blank line immediately below this and place the cursor on it. Click on the *Call Business Object* button on the toolbar, supply the business object name of ***ARSQRY***, and press the *Tab* key. The *Parameters* tab will be populated with the sample XML for this business object. Replace the XML in the parameters tab with the XML that appears below.

```
<Query>
  <Key>
    <Customer>xxxxxx</Customer>
  </Key>
  <Option>
    <IncludeFutures>N</IncludeFutures>
    <IncludeTransactions>N</IncludeTransactions>
  </Option>
</Query>
```

Click on the *Insert VBScript* button on the toolbar and the code to call the business object with this XML will be inserted for you (see **Figure 6-41**).

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnGoToMasterAccount()
6    If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
7      dim XMLOut, XMLParam
8
9      XMLParam = XMLParam & " <Query>"
10     XMLParam = XMLParam & "   <Key>"
11     XMLParam = XMLParam & "     <Customer>xxxxxx</Customer>"
12     XMLParam = XMLParam & "   </Key>"
13     XMLParam = XMLParam & "   <Option>"
14     XMLParam = XMLParam & "     <IncludeFutures>N</IncludeFutures>"
15     XMLParam = XMLParam & "     <IncludeTransactions>N</IncludeTransactions>"
16     XMLParam = XMLParam & "   </Option>"
17     XMLParam = XMLParam & " </Query>"
18     on error resume next
19     XMLOut = CallBO("ARSQRY",XMLParam,"auto")
20     if err then
21       msgbox err.Description, vBCritical, "Calling Business Object"
22       exit function
23     end if
24     ' Switch on error handling
25     on error goto 0
26
27 End Function
```

Figure 6-41: The code to call the ***ARSQRY*** business object

Line 7 defines that there are two variables that are available to be used by the script, *XMLOut* and *XMLParam*.

```
dim XMLOut, XMLParam
```

To interrogate the XML that will be returned by the business objects you need two more variables, which will be called *XMLDoc* and *MastAcc*. Modify line 7 to include the variables by making it match the line below.

```
dim XMLOut, XMLParam, XMLDoc, MastAcc
```

Line 11 (that appears below) includes the *Customer* element, which currently contains the value *XXXXXX*. This must be changed to pick up the current customer's account code.

```
XMLParam = XMLParam & "     <Customer>xxxxxx</Customer>"
```

As the string *XXXXXX* is only present to highlight where the change must occur, remove it. Between the opening and closing *Customer* tags, add a double-quote character, a space character, an ampersand, and another space character. Then locate the *Customer* variable within the *CustomerInformation* section on the *Variables* pane, and double-click it. The full name of the *Customer* variable will be added.

Between the end of the *Customer* variable name and the closing *Customer* tag, add a space character, an ampersand, another space character, and a double-quote. This line should now match the one that appears below. Note that this is one line, and has wrapped around on this page.

```
XMLParam = XMLParam & "     <Customer>" & CustomerInformation.CodeObject.Customer &
"</Customer>"
```

Add the following three lines of code between the last line of the code (which is *on error goto 0*) and the *End Function* statement

```
Set XMLDoc = createobject("MSXML2.DOMDocument")
XMLDoc.async = false
XMLDoc.LoadXML(XMLOut)
```

These three lines create an instance of the Document Object Model (DOM) and place the output from the business object (contained in the *XMLOut* variable) into the DOM. Directly below this add the following line of code.

```
MastAcc = XMLDoc.SelectSingleNode("//ARStatement/Header/MasterAccount").Text
```

This line of code looks through the XML in the DOM starting at the root element (*ARStatement*), then into the *Header* section, and extracts the contents of the *MasterAccount* element. This contains the

account number of this sub-account's master account. The content of this element is placed in the *MastAcc* variable.

Add a blank line directly below this line and place the cursor on it. Locate the *ToolbarButton* variable within the *SystemVariables (actions)* section on the *Variables* pane, and double-click it. The script containing the script will disappear and the *Modify Toolbar Buttons* screen will appear.

Click on the *Customer* prompt on the *AR Customer Query* screen. On the *Modify Toolbar Buttons* screen the *Toolbar name* prompt will be populated with *ARSPENTB*, the *Caption* prompt will be populated with *&Customer:*, and the *ID* prompt will be populated with *38000*. The *Button value* prompt will also become enabled. Add in a button value of *XXXXXX* and change the *Action* dropdown to reflect *Execute*. Click on the *Add to List* button so that these settings appear in the listview at the bottom of the screen, then click on the *Insert VBScript Code* button (see **Figure 6-42**).

```
25      on error goto 0
26
27      ' Load the output from the business object in to the DOM
28      Set XMLDoc = createobject("MSXML2.DOMDocument")
29      XMLDoc.async = false
30      XMLDoc.LoadXML(XMLOut)
31
32      MastAcc = XMLDoc.SelectSingleNode("//ARStatement/Header/MasterAccount").Text
33
34      SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
35      "<Toolbar Name='ARSPENTB' Id='38000' Action='Execute' Value='xxxxxx' Caption='Customer:'/>" & _
36       "</Toolbars>"
37  End Function
```

Figure 6-42: The code added to interrogate the XML and modify the button

The entered string of *XXXXXX* appears within the code to modify the button (on line 35 in **Figure 6-42**). This must be modified to reflect the contents of the *MastAcc* variable (that contains the account code of the master account). The section to be modified appears below.

```
Value='xxxxxx'
```

Replace the string *XXXXXX* with a double-quote, a space character, and ampersand, another space character, the variable name *MastAcc*, a space character, an ampersand, another space character and a double-quote. When you have finished this section should match the entry below.

```
Value='" & MastAcc & "'
```

The last item is to add the *END IF* statement between the line containing the closing *Toolbars* element, and the *End Function* statement. The completed code appears in **Figure 6-43**.

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnGoToMasterAccount()
6    If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
7      dim XMLOut, XMLParam, XMLDoc, MastAcc
8
9      XMLParam = XMLParam & " <Query>"
10     XMLParam = XMLParam & "   <Key>"
11     XMLParam = XMLParam & "     <Customer>" & CustomerInformation.CodeObject.Customer & "</Customer>"
12     XMLParam = XMLParam & "   </Key>"
13     XMLParam = XMLParam & "   <Option>"
14     XMLParam = XMLParam & "     <IncludeFutures>N</IncludeFutures>"
15     XMLParam = XMLParam & "     <IncludeTransactions>N</IncludeTransactions>"
16     XMLParam = XMLParam & "   </Option>"
17     XMLParam = XMLParam & " </Query>"
18     on error resume next
19     XMLOut = CallBO("ARSQRY",XMLParam,"auto")
20     if err then
21       msgbox err.Description, vBCritical, "Calling Business Object"
22       exit function
23     end if
24     ' Switch on error handling
25     on error goto 0
26
27     ' Load the output from the business object in to the DOM
28     Set XMLDoc = createobject("MSXML2.DOMDocument")
29     XMLDoc.async = false
30     XMLDoc.LoadXML(XMLOut)
31
32     MastAcc = XMLDoc.SelectSingleNode("//ARStatement/Header/MasterAccount").Text
33
34     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
35     "<Toolbar Name='ARSPENTB' Id='38000' Action='Execute' Value='" & MastAcc & "' Caption='Customer:'/>" & _
36     "</Toolbars>"
37   End IF
38 End Function
```

Figure 6-43: The completed code

Exit from the script to the *VBScript Editor* screen and click on the *Save* button. Exit the *AR Customer Query* back to the menu, and call up the program again. The *Go To Master Account* form action will appear at the bottom of the *Customer Information* pane. If you select a customer that is not a sub-account, when you click on the *Go To Master Account* form action, nothing will happen. If you select a customer that is a sub-account, when you click on the *Go To Master Account* form action the master account code for this customer will be placed against the *Customer* prompt and the query will be actioned for this account code.

Sample Code

The *Sample Code* pane contains VBScript code snippets that you can import into your script, and modify if required. Each snippet displays a short description (that appears within braces) and the first three lines of the code. The snippets will be inserted at the point where the cursor last resided in the

Script Area if you highlight the required snippet and click on the *Insert Code* button on the pane's toolbar (or double-click on the snippet).

VBScript Modules

The *VBScript Modules* functionality enables you to create a library of frequently used functions, and reference these in your VBScript. The benefit of this over adding the code into your script is that if a change needs to be made it only needs to be made in one place, and is automatically available in all scripts that reference it.

VBScript Modules are VBScript files that have a suffix of `.vbs` and each contains one or more functions. These modules are stored in a folder called *Vbsmodules* that exists under the SYSPRO `Work\Vbscripts` folder on the SYSPRO application folder.

When you add/maintain a VBScript through SYSPRO's *VBScript editor* the *VBScript Modules* pane will appear. Its default location is as a tab on the right of the screen, but if it has been pinned using the *Auto Hide* option it can appear anywhere on the screen.

The filename and first three lines of each script are displayed within the *VBScript Modules* pane, so it is advisable to add a meaningful description to the top of the script. If you move the mouse pointer over one of the modules, a tooltip will display the contents of the first 18 lines.

Figure 6-44 shows a *VBScript Modules* pane that contains three modules (*AMessageCall.vbs*, *BasicSqlInterfaceClass.vbs*, and *Common.vbs*). These are displayed in alphabetic sequence. The mouse pointer has been positioned over the *BasicSqlInterfaceClass* module, so the first 18 lines appear in a tooltip.

At the top of the pane is a toolbar that contains buttons to *Insert Code*, add a *New* module, *Edit* an existing module, or *Delete* a module. The *New* button displays a blank VBScript editing pane and you can add your code (don't forget the meaningful description at the beginning, and the *Option Explicit* statement). When you click on this pane's *Save* button you are prompted for a name for this script.

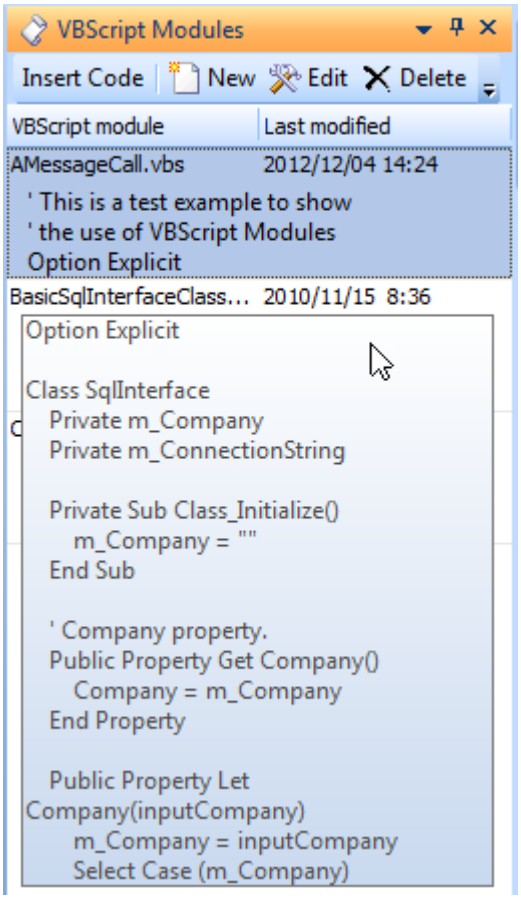


Figure 6-44: The *VBScript Modules* pane

The *Edit* button opens a new VBScript editing pane and places all the code of the currently selected module within it. After modifying the script the existing one will be overwritten when you click on the pane's *Save* button.

The *Delete* button is used to delete the currently selected module. Before this is deleted you are prompted to make sure that you want to delete it, and warned that if any of your scripts use it they will report errors and not function correctly.

The *Insert Code* button does not insert the contents of the currently selected module into your code. Instead, it inserts a line of code that causes this module to be loaded into memory when the main script is run.

cripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  LoadModule("AMessageCall.vbs")
6
7  Function OrderHeader_OnLoad()
8    OrderHeader.CodeObject.JustButton = "<Field><Button>Click to check</Button></Field>"
9    call (Mymessage)
10 End Function
```

Figure 6-45: Loading the module, and calling the function

Line 5 of **Figure 6-45** shows the line of code that is inserted if the *Insert Code* button is clicked when the module *AMessageCall.vbs* is highlighted. Only a link to this module is inserted, not the whole module. Note that this is not inserted as part of a function, and this has been inserted near the top of the script.

The VBScript module *AMessageCall.vbs* contains a function called *MyMessage*. Line 9 of **Figure 6-45** shows how to call this function in the module from within the *OrderHeader_Onload* function.

Procedures

The *Procedures* tab contains a list of all the functions that are available within the script currently being edited, whether these are standard SYSPRO functions or custom-created ones (see **Figure 6-46**). Double-clicking the function name will place the cursor on the first line of that function within the *Script Area*.

When you only have a couple of small functions this will not be of much benefit to you. However, when you have many functions (or a very large script) you will find that using the *Procedures* pane to navigate around the script will save quite a lot of time.

The list in the pane is updated as you call up the editor. If you selected to add a new function as you called up the editor (because you selected an event name that did not exist and clicked on the *Edit VBScript* button, for instance) this will be present because it was created as the editor was loaded. However, if you manually create a function it will not appear in the *Procedures* pane until you next call up the editor.

The *Procedures* pane can be used even if it is not pinned in place, and appears as a tab. By simply moving the mouse pointer over the tab it will expand the pane, you can double-click on the required function name in the pane, then move the mouse pointer away from this pane. The cursor will be placed in the correct location

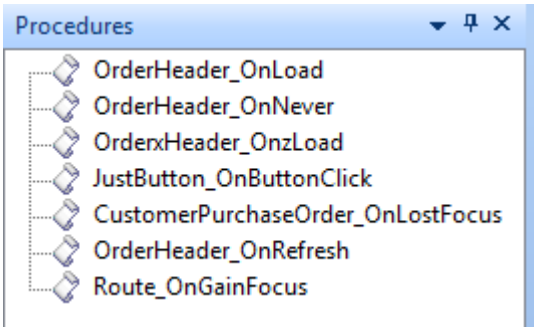


Figure 6-46: The *Procedures* pane

Field Properties

Before covering how to use the *Field Properties* pane it is worth explaining how field properties work, and some of the things that you can do with them. Field properties consist of one or more XML attributes that exist within an XML string, and this XML string is placed against the field for which the properties are to be set.

To set field properties against a field you should first supply the field's variable name, followed by an equal sign, and then the field properties XML. The field's variable name is selected by double-clicking on the variable name within the *Variables* pane (or you can manually type in its name). The example below will display the value against the *Customer Purchase Order* field in bold type.

```
OrderHeader.CodeObject.CustomerPurchaseOrder = "<Field IsBold='true'  > </Field>"
```

One of the simplest things is to set a property against a field so that the field is highlighted, which will draw the operator's attention to it. **Figure 6-47** shows that the *Clearing stock flag* field in the *Inventory Query* has its background color set to light turquoise, and **Figure 6-48** shows the code that performs this, which has been placed against the *StockCodeDetails_OnRefresh* function.

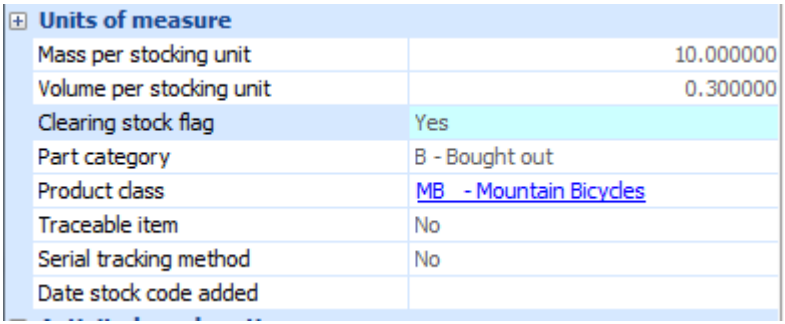


Figure 6-47: Setting the background color field property

bscripts\INVPENL1

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function StockCodeDetails_OnRefresh()
6    StockCodeDetails.CodeObject.ClearingStockFlag = "<Field Background='204255255'  > </Field>"
7  End Function
```

Figure 6-48: The code to set the background color of the *Clearing stock flag* to light turquoise

Once a field property is set it remains until it is changed. This can be by the program (in the case of values because new values are retrieved and redisplayed), or your code changes the field property, or you exit the program.

The problem with the above code is that if something is always highlighted operators eventually don't notice it, so it would be better to only highlight the field when there is an exception. The code in **Figure 6-49** does a check on the value of this field, and only changes the background color if it contains the text *Yes*.

bscripts\INVPENL1

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function StockCodeDetails_OnRefresh()
6    If StockCodeDetails.CodeObject.ClearingStockFlag = "Yes" then
7      StockCodeDetails.CodeObject.ClearingStockFlag = "<Field Background='204255255'  > </Field>"
8    Else
9      StockCodeDetails.CodeObject.ClearingStockFlag = "<Field Background=''  > </Field>"
10   End If
11 End Function
```

Figure 6-49: Using conditional logic to set/clear field properties

Just as importantly, if the value does not contain the text *Yes*, the background property is removed. If you did not remove this property, once you had viewed a stock code that had the clearing code set to *Yes*, all the other stock codes would display this field with a light turquoise background, no matter what their field contained.

Another example of what can be done with field properties appears in **Figure 6-50**. Here a button has been added to a scripted field called *Just Button* that was dragged onto the *Order Header* form.

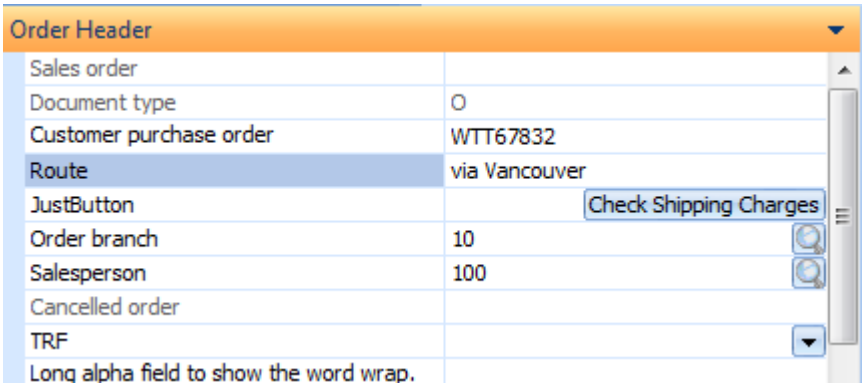


Figure 6-50: Creating a button on an Entry form

Within the *OrderHeader_OnLoad* function the button was created by simply adding text against the *Button wording* field property, and when the *Insert VBScript Code* button was clicked the *Button* field property was added to this field. The code for this appears in **Figure 6-51**. Code would then be added to the *JustButton_OnButtonClick* function to specify what should happen when this button is clicked.

vbscripts\IMP040L3

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function OrderHeader_OnLoad()
6    OrderHeader.CodeObject.JustButton = "<Field><Button>Check Shipping Charges</Button></Field>"
7  End Function
```

Figure 6-51: The code to create the *Check Shipping Charges* button when the form loads

Field Properties Pane

The *Field Properties* pane simplifies the process of setting properties against a field. You could type the same code manually, but using the *Field Properties* pane makes the process so much easier. **Figure 6-52** shows the *Field Properties* pane with its default settings.

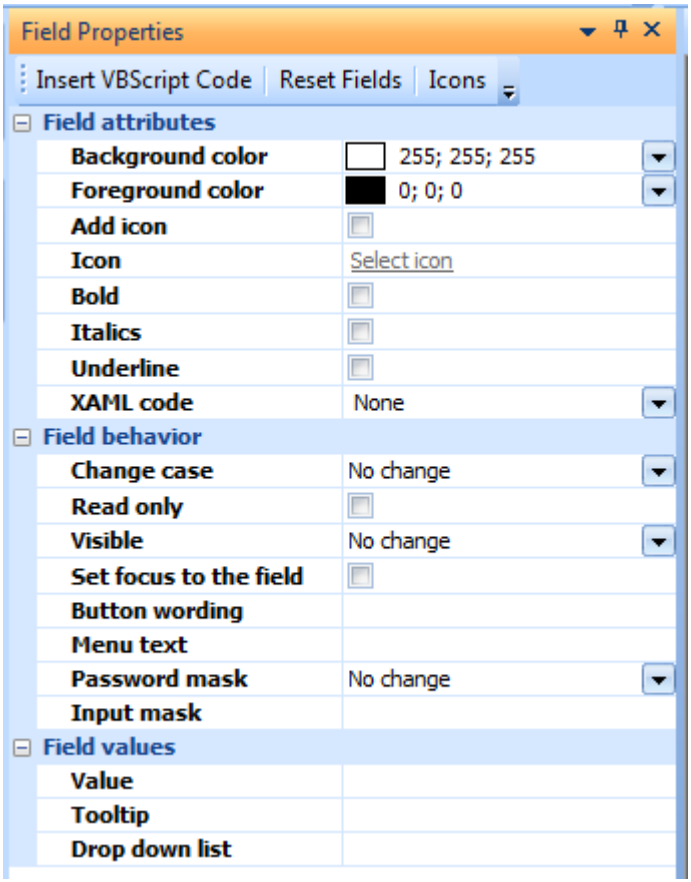


Figure 6-52: The *Field Properties* pane

This pane consists of a toolbar and a form containing three sections, *Field attributes*, *Field behaviour*, and *Field values*. The toolbar consists of three buttons*, Insert VBScript Code*, *Reset Fields*, and *Icons*.

The *Insert VBScript Code* button looks through all the settings in the *Field Properties* pane to see which do not contain their default values, builds up the XML to set all of these field properties, and inserts this XML where the cursor was last positioned in the VBScript pane.

The XML root element is `<Field>`. If there are no field properties containing values other than their defaults, the XML output when you click on the *Insert VBScript Code* button will be:

```
"<Field  > </Field>"
```

If you check the *Bold* option to set the text to be in bold, when you click on the *Insert VBScript Code* button the output XML will contain:

```
"<Field IsBold='true'  > </Field>"
```

The *Reset Fields* button sets all the properties back to their default values. While you are still in the *VBScript Editor*, the changes that you make to the *Field Properties* pane are cumulative. So if you check the *Bold* option and use the *Insert VBScript Code* button, the bold flag does not become unchecked. If you then check the *Italics* option and click on the *Insert VBScript Code* button you will get the XML to set the field to *Bold* and *Italic*.

This is the XML that you will get when both the *Bold* and *Italics* checkboxes are checked:

```
"<Field IsBold='true' IsItalic='true'  > </Field>"
```

This cumulative effect is useful when you are building up field properties in your code for a specific field, but if you are going to start setting the field properties for another field it is advisable to click on the *Reset Fields* button first.

The *Icons* button displays the *Change Icon* screen containing all the icons, but in display mode so that you can see which ones are available, along with a checkbox to display these as small icons. You cannot select these for use from the *Icons* button.

Field Attributes

The majority of the field properties in the *Field Attributes* section of the *Field Properties* pane are used to manipulate the font of the displayed value. The other field properties in this section are used to highlight the field in some way. **Figure 6-53** shows all the options under the *Field attributes* section.

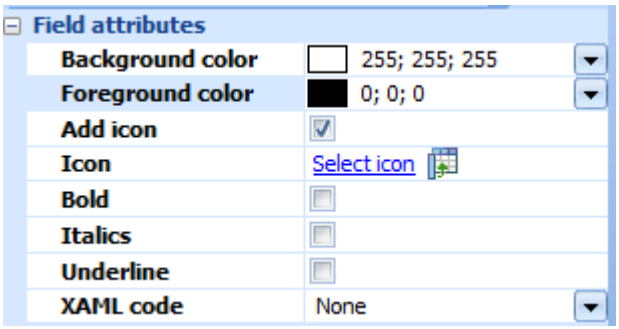


Figure 6-53: The *Field attributes* section of the *Field Properties* pane

The first field property is used to set the *Background color* of the field's value, and the default is white. Next to the displayed sample block of this color are three numeric values, separated by semi-colons.

This is the *RGB* value, which represents the amount of the Red, Green, and Blue colors making up this color. Each of these values can be in the range 0 to 255, where 0 means that it contains none of this color, and 255 means that it contains the maximum amount of this color. White is represented as 0;0;0 because it contains no red, green, or blue. Black is represented as 255;255;255 as it contains the maximum amount of red, green, and blue. It is this combination of numbers that is inserted against the relevant XML attribute when the *Insert VBScript Code* button is clicked.

An example can be seen in **Figure 6-49** where the color *Light Turquoise* is made up the the RGB values 204;255;255, and when the *Insert VBScript Code* button was clicked the following XML was inserted.

```
"<Field Background='204255255'  > </Field>"
```

Alongside the RGB values is an icon showing that there is a color chooser available. When this is selected the chooser is displayed. At the top of the chooser is the option to reset to the default value for this field, using the *Automatic color* option. There are also 40 color blocks that represent the more common colors, and their names are displayed in a tooltip if you hover your mouse pointer over them. **Figure 6-54** shows the color chooser with the mouse pointer hovering over the *Light Turquoise* color block.

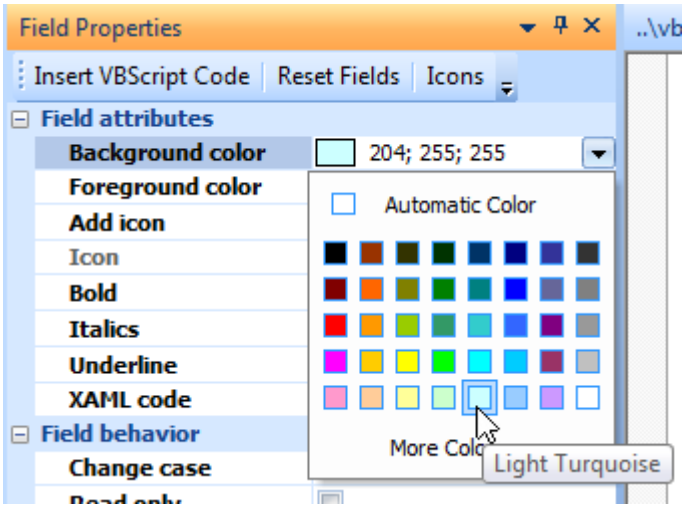


Figure 6-54: The color chooser with the mouse pointer hovering over the *Light Turquoise* color block

At the bottom of the color chooser is the *More Colors* option. If this is selected a new screen is displayed that has both *Standard* and *Custom* tabs, that let you choose from more colors, or design your own.

The *Foreground color* works in exactly the same way as the *Background color* option.

The *Add icon* and *Icon* options work in conjunction to display an icon against the value portion of the field. An example can be seen in **Figure 6-55** where the icon has been inserted against the value portion of the *Customer purchase order* field.

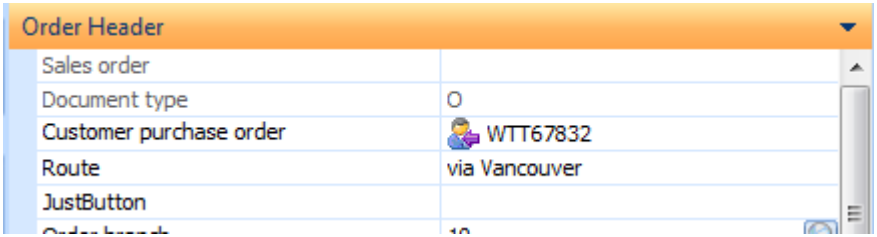


Figure 6-55: An icon inserted against the value portion of the field using the Field Properties

When the *Add icon* option is checked the *Select icon* hyperlink against the *Icon* field attribute is enabled. When the *Select icon* hyperlink is selected the *Change Icon* screen is displayed, which contains the 236 available large icons. If the *Small icons* checkbox is checked the same icons are displayed but in a smaller format. At the bottom left of the screen the reference number of the currently selected icon is displayed.

If you select an icon and click on the *OK* button the small representation of this icon will appear alongside the *Select icon* hyperlink (see **Figure 6-56**).

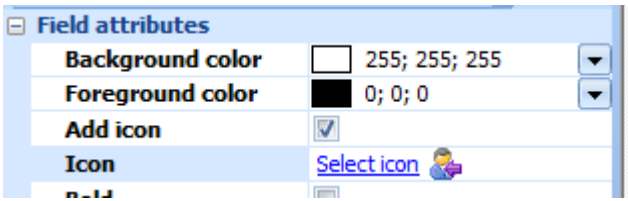


Figure 6-56: The selected icon

When the *Insert VBScript Code* button is clicked the code is inserted that contains this icon number.

```
"<Field Icon='196'  > </Field>"
```

The checkboxes for *Bold*, *Italics*, and *Underline* properties all behave as you would expect. They can be used in any combination. The following is the field property XML that is inserted if all are selected.

```
"<Field IsBold='true' IsItalic='true' IsUnderline='true'  > </Field>"
```

The *XAML code* option enables you to assign a file containing XAML markup code to an alphanumeric cell in a listview. This can be to highlight the contents of a cell by changing its color, or placing a border around it. This is different from right-clicking on the column header and selecting the XAML theme in that it is associated with a specific cell rather than all the cells in the column.

However, depending on which XAML code file is selected, it can also enable you to display more than one piece of information in a cell. **Figure 6-57** shows how the *Warehouses* browse ***IMPBWH*** has been enhanced to show the warehouse's description and its full address in the *Description* cell instead of just the warehouse's description. This technique can be useful when screen width is limited (because it allows you to reduce the number of columns displayed by placing the contents of these columns within the cells of other columns) or as in this case, where the information was not available within the listview, using a business object to return the values and displaying them in a logical place.

The XAML files supplied with SYSPRO reside in the `Base\Samples` folder, and they all accept parameters that are used to display values. These parameters exist within the XAML markup code as numbers with a percentage sign before and after them. For example, the first parameter is *%1%*, the second is *%2%*, etc. At run time these parameter numbers are replaced by the values supplied.

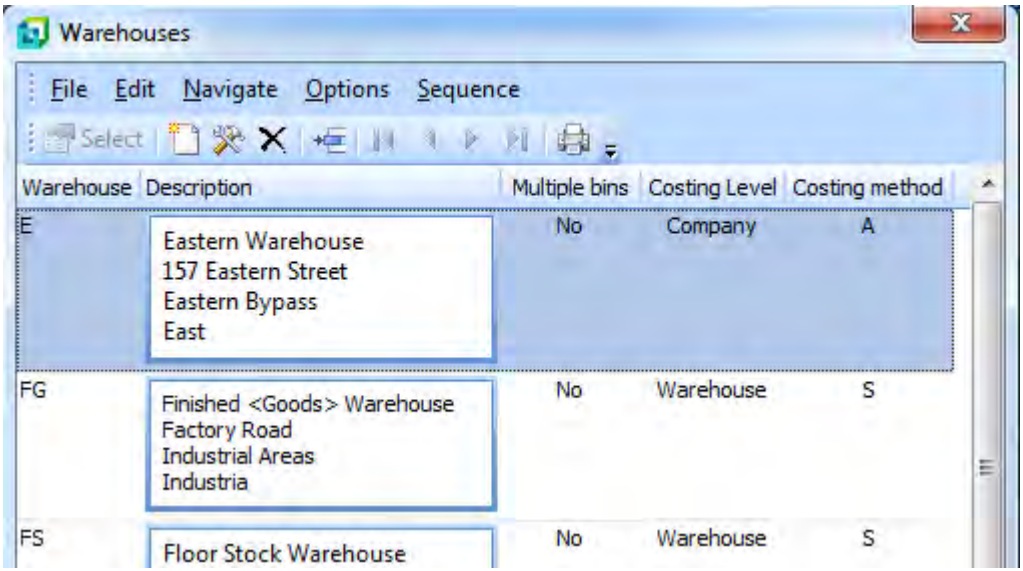


Figure 6-57: An example showing a cell populated with data that is not available in this browse

A large number of these XAML files accept only one parameter, but some accept up to 36. In most cases, if less parameters are supplied than the number of parameters that can be accepted, the parameter numbers are replaced with blank values. However, the graph and trendline XAML markup files require a minimum number of parameters to be able to render correctly. If insufficient parameters are supplied, the XAML will not be able to render properly and may display its source code in the cell.

At the top of these XAML markup files is a comment section (just like VBScript, XAML comments are defined as starting with the string *<!--* and ending with the string *-->*) and these comments explain the function of each parameter in this XAML file.

If you use the XAML code field property against a cell without specifying a value, the current content of this cell is automatically supplied to the XAML markup code as the first parameter (*%1%*). However, if you supply any parameters to the markup code the current content of this cell will not automatically be passed to the XAML file, and must be manually added to your parameters if you want to display it.

In this example the field properties were used to add the *AddressPlain* XAML markup to the cell that is the first row of the *Invoices* column of the *Invoices* listview in the *Customer Query*. This was done against the listview's *OnPopulate* function. The code below does not contain any mention of a value being passed through (this is one line that has wrapped around on this page).

```
CustomerInvoices_OUT.CodeObject.Array(000,0) = "<Field XAMLCode='AddressPlain' >
</Field>"
```

When the listview next displays values, the specified cell has the *AddressPlain* XAML markup applied to it and as no parameters are specified the invoice number appears within this as it is automatically passed through as the first parameter (see **Figure 6-58**).

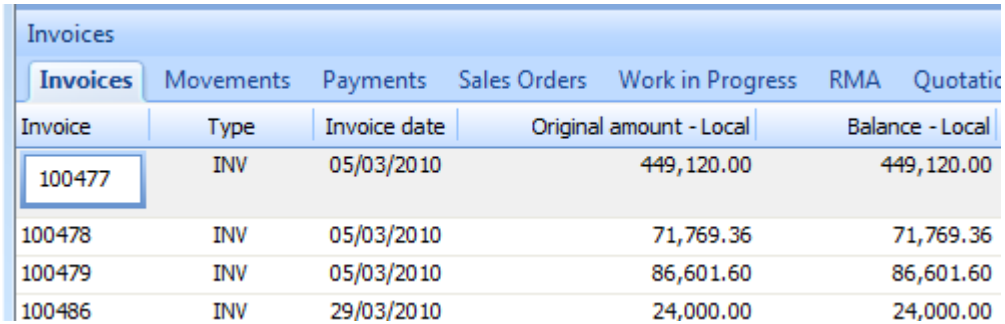


Figure 6-58: Applying just the *XAMLCode* without supplying a value

Note that in the code the field properties are applied to the "*_OUT*" array, as was covered in Chapter 5 on *Macro Events*.

If a single value is to be passed to the XAML to be displayed, this will replace the current content of the cell as being the first parameter passed to the XAML code. The value can be added at the same time as specifying the XAML code, under the *Field values* section of the *Field Properties* screen. Alternatively, this attribute can be added manually as per the example of code below. This is one line but has wrapped around because of the length of the line.

```
CustomerInvoices_OUT.CodeObject.Array(000,0) = "<Field XAMLCode='AddressPlain'
Value='9999999' > </Field>"
```

The supplied value is only displayed in place of the original value, it does not replace the original value. It also does not get written back to the table from where the original value was extracted.

If multiple values need to be passed to XAML the values are separated by pipe signs (|). Using the above example, if three parameters are 9999999, 88888888, and 7777777, the *Value* attribute would contain:

```
Value='9999999|8888888|7777777'
```

Going back to the Invoices listview example in **Figure 6-58** above, where a specific cell number was mentioned. Instead of just having the invoice number appearing in XAML for the first invoice number, you might want the invoice number, type, branch, area, and customer purchase order number to all appear within this cell. Taking this one step further, you might also want this for every cell under the *Invoice* column. This provides the opportunity to introduce a few new concepts, while still keeping it reasonably simple. **Figure 6-59** shows the end result. Once this is done the columns that are now being displayed in the XAML can be dragged from the listview to free up real estate (such as the *Type* column that appears alongside the *Invoice* column).

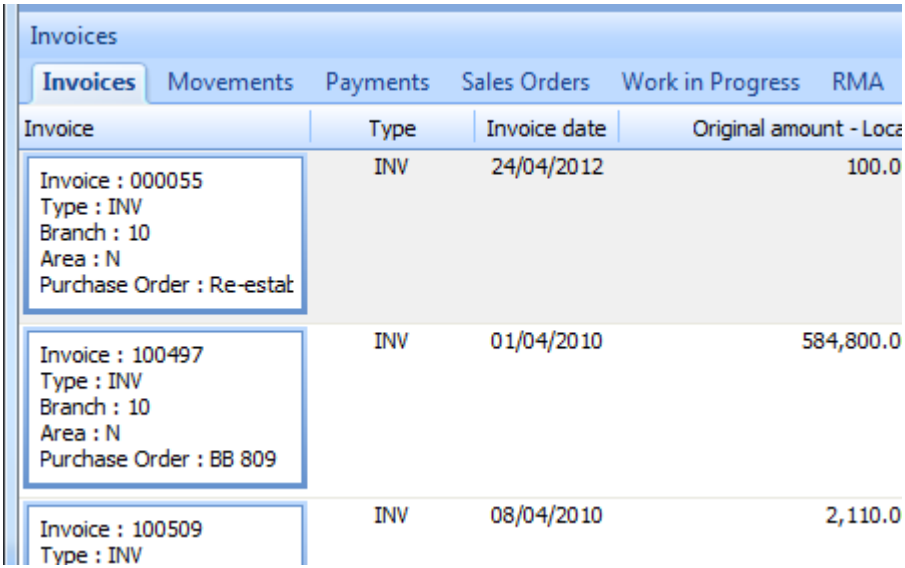


Figure 6-59: Displaying values from multiple columns in the *Invoice* cells

Figure 6-60 shows the code that makes this possible. Line 6 and 7 both use a *DIM* statement to define the names of variables that are going to be used within this function. The variable names are comma-

delimited. These two lines could have been made into one line, but were made into two separate lines to make it easier to view here.

bscripts\ARSPENLV

```
1  ' This script contains functions for list view events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInvoices_OnPopulate()
6    Dim RowNumber, MaxNumber, ReportArray, InvoiceNum, InvoiceType
7    Dim Branch, Area, CustomerPO
8    ReportArray = CustomerInvoices.CodeObject.Array
9    MaxNumber = UBound(ReportArray,2)
10
11   For RowNumber = 0 to MaxNumber -1
12     InvoiceNum = CustomerInvoices.CodeObject.Array(000,RowNumber)
13     InvoiceType = CustomerInvoices.CodeObject.Array(001,RowNumber)
14     Branch = CustomerInvoices.CodeObject.Array(010,RowNumber)
15     Area = CustomerInvoices.CodeObject.Array(011,RowNumber)
16     CustomerPO = CustomerInvoices.CodeObject.Array(008,RowNumber)
17     CustomerInvoices_OUT.CodeObject.Array(000,RowNumber) = "" & _
18       "<Field XAMLCode='AddressPlain' Value='Invoice : " & _
19       InvoiceNum & "|Type : " & InvoiceType & "|Branch : " & _
20       Branch & "|Area : " & Area & "|Purchase Order : " & _
21       CustomerPO & "' > </Field>"
22   Next
23 End Function
24
```

Figure 6-60: The code to render the XAML shown in **Figure 6-59**

Lines 17, 18, 19, 20, and 21 are treated as one line when the script is run. They use the technique of adding an ampersand and an underscore at the end of the line to tell VBScript that the line of code continues on the next line.

Back in Chapter 5 the topic of listviews was covered in some detail. There is an explanation that when the *OnPopulate* event fires the listview appears in two arrays, one contains the current values, and the other one accepts new values/field properties that must be changed before display. If you have not already read this section, please read it before continuing here.

On line 8 the variable name *ReportArray* is set to contain the contents of the first array (the one containing the current values).

Line 9 introduces the VBScript *UBound* function. When the *UBound* function is passed the contents of an array it will return either the highest row or highest column number in the array, depending on the parameter supplied. In this case, the content of the array is passed in the *ReportArray* variable, and the supplied parameter is "*2*". This tells the function to return the highest row number (the value "*1*", or not supplying this parameter at all as it is the default, tells the function to return the highest column number). The variable *MaxNumber* is populated with the returned value.

Line 11 starts a *For*/*Next* loop that runs to line 22. As covered in the *OnPopulate* section of Chapter 5, the numbering of rows and columns in an array starts at 0. So if there are 5 rows in the listview (numbered 1 to 5), in an array these would be numbered 0 to 4.

The loop must to be set to run from row zero to the highest row number in the array. The *MaxNumber* variable contains the number of rows, but this needs to have one subtracted from it to give the highest row number in the array, as the array starts with zero (see the explanation above).

Each time that processing goes through the *For*/*Next* loop, the *RowNumber* variable will contain the number of the current row of the array being processed (starting at zero).

Line 12 populates the variable *InvoiceNum* with the contents of the *Invoice* cell for the current row, as column zero in the array is the *Invoice* column. So the first time through the *For*/*Next* loop it will contain the values held in the *Invoice* cell for listview row 1/array row 0, the second time it will contain the values held in the *Invoice* cell for listview row 2/array row 1, etc.

Lines 13, 14, 15, and 16 are similar to line 12 except that they are for the *Type*, *Branch*, *Area*, and *Customer purchase order* columns respectively.

Line 17 is where this all comes together (remember that lines 17 to 21 are really one line, as it has only been split over five lines to make it fit on the page). If the line were to be put back together without the characters used to split it, this line would look like the following (bearing in mind that it is now wrapping):

```
CustomerInvoices_OUT.CodeObject.Array(000,RowNumber) = "<Field
XAMLCode='AddressPlain' Value='Invoice : " & InvoiceNum & "|Type : " & InvoiceType
& "|Branch : " & Branch & "|Area : " & Area & "|Purchase Order : " & CustomerPO &
"' > </Field>"
```

The part of the full line that appears on line 17 specifies that it is accessing column zero for the current row number of the *_OUT* array (which is the one that you make changes to so that these changes are reflected in the listview when it is displayed).

The part of the full line that appears on line 18 specifies that it should use the *AddressPlain* XAML markup file. It also starts the *Value* attribute (that finishes on line 21) that will be used to supply the pipe sign delimited values as parameters. The string making up the contents of the *Value* attribute

contains five labels and their matching variable names (the label also contains a space character, and colon, and another space character to make it easier to read). Between one label/variable pair and the next is a pipe sign.

Stripping out all the characters used to build this, the string against the *Value* attribute is the equivalent of the following, where the *InvoiceNum*, *InvoiceType*, *Branch*, *Area*, and *CustomerPO* variables will be replaced by the values from the appropriate cells at run time.

```
Invoice : InvoiceNum|Type : InvoiceType|Branch : Branch|Area : Area|Purchase Order
: CustomerPO
```

As an example, using the contents of the first cell in **Figure 6-59** (row 0, column 0) the string would be:

```
Value='Invoice : 000055|Type : INV|Branch : 10|Area : N|Purchase Order : Re-estab'
```

The *NEXT* statement says that this is the end of this row and to return to the *FOR* statement to see if there are any more rows in the array to process. If there are, it processes lines 12-21 for that row. If there aren't, it drops through the *For*/*Next* loop, in this case to the *End Function* statement and the listview is populated with the values.

Field Behavior

The *Field behavior* section is used to control how the field behaves. The first option is *Change case* and this has a dropdown list containing *No change*, *Upper*, *Lower*, and *Mixed*. If a field has *Upper* set against it, the contents of the field are automatically converted to uppercase as they are typed in.

In the following example the *Customer purchase order* number field on the *Order Header* form of *Sales Order Entry* has the case of *Upper* set against in the *Order Header* form's *OnRefresh* function. As soon as the operator starts typing the customer purchase order number the letters are automatically converted to uppercase. This isn't just for display purposes, the letters are changed to uppercase, and if the sales order is saved, these uppercase values are written to the database.

```
Function OrderHeader_OnRefresh()
  OrderHeader.CodeObject.CustomerPurchaseOrder = "<Field Case='Upper'  > </Field>"
End Function
```

The *Lower* option converts all the characters to lowercase, and the *Mixed* option sets it so that the characters appear as they are typed.

The *Read only* checkbox is used to set a field on an entry form to be read-only. This can be set against the form's *OnLoad* event (if it must always be read-only), or conditional logic can be used to make the field toggle between read-only, and read-write.

The following example checks the value of the *Branch* field on the *Sales Order Entry* program's *Order Header* form. If the field's value contains *10* the field is set to read/write. If it is any other value the field is set to read-only and the contents of the field cannot be changed.

```
Function OrderBranch_OnAfterChange()
  If OrderHeader.CodeObject.OrderBranch <> "10" then
    OrderHeader.CodeObject.Route = "<Field IsReadOnly='true'  > </Field>"
  Else
    OrderHeader.CodeObject.Route = "<Field IsReadOnly='false' > </Field>"
  End If
End Function
```

The next option is *Visible*, and goes one step further than the read-only option in that it hides/displays fields. The available choices from the dropdown list against the *Visible* option are *No change*, *True*, and *False*. If *True* is set for a field, the field will be displayed. If *False* is set, the field is removed from the screen until it is either set to *True* again, or the next time that the program is loaded. The following code will remove the *Route* field from the screen if the *Branch code* is changed to anything other than *10*, and return it to the screen when the *Branch code* becomes *10* again.

```
Function OrderBranch_OnAfterChange()
  If OrderHeader.CodeObject.OrderBranch <> "10" then
    OrderHeader.CodeObject.Route = "<Field Visible='false'  > </Field>"
  Else
    OrderHeader.CodeObject.Route = "<Field Visible='true'  > </Field>"
  End If
End Function
```

Figure 6-61 shows that when the *Branch code* is *10* the *Route* field is visible, and **Figure 6-62** shows that after the *Branch* has been changed to *20* the *Route* field disappears from the form. If the *Branch* was changed back to *10* the *Route* field would appear again on the form in the same location as it was previously.

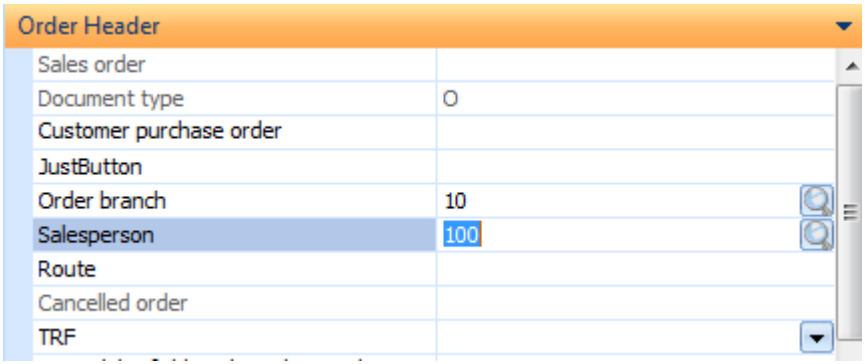


Figure 6-61: The *Branch* is *10* so the *Route* field is displayed

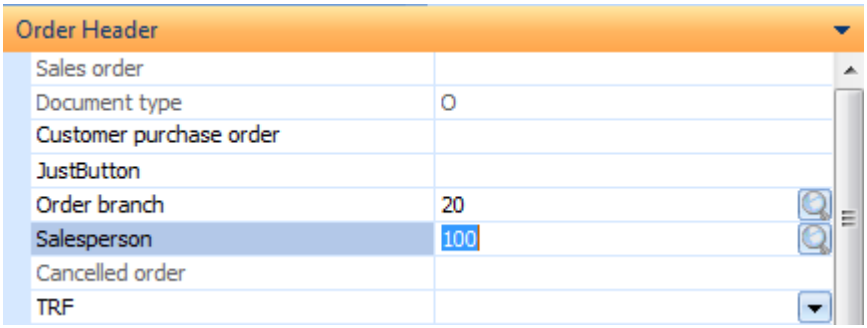


Figure 6-62: The *Branch* is changed to *20* so the *Route* field disappears from the form

The next item under the *Field behaviour* section is *Set focus to the field*. When this *Field property* is set the cursor will be placed on the value portion of the specified field, unless the program needs to put is somewhere else. The following example places the cursor on the *Route* field if the *Branch* contains any value other than *10*. If it contains *10* then the normal tabbing sequence is observed.

```
Function OrderBranch_OnAfterChange()
  If OrderHeader.CodeObject.OrderBranch <> "10" then
    OrderHeader.CodeObject.Route = "<Field Focus='true'  > </Field>"
  End If
End Function
```

Being able to set the focus is particularly useful when you are performing your own field verifications against a form's *OnSubmit* function. Under these circumstances, if your verification fails you can prevent processing from continuing by setting the *OnSubmit* function to *false*, and then use this field property to place the cursor on the field that failed the verification.

The *Button wording* field property causes a button to be created against the value portion of the specified field. This is only applicable to entry forms. If the field already has a button against it (because the field has a browse or other system generated button), this option causes a second button to be created alongside the first. **Figure 6-63** shows where button wording has been added to the *JustButton* and *Order branch* fields. In the case of the *Order branch* field, as this already had a browse button, a second button was added to its right. The code to do this appears below. Note that the line for *JustButton* has wrapped around.

```
Function OrderHeader_OnLoad()
  OrderHeader.CodeObject.JustButton = "<Field><Button>Click to check</Button>
</Field>"
  OrderHeader.CodeObject.OrderBranch = "<Field><Button>My Button</Button></Field>"
End Function
```

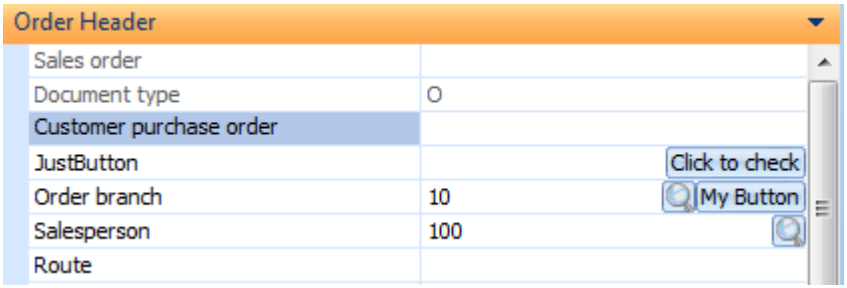


Figure 6-63: Buttons added to both the *JustButton* and *Order branch* fields

Within the VBScript Editor screen, each field on an entry form has the option to create an *OnButtonClick* function, regardless of whether a button exists against the field. If one does exist, and it is clicked, any code against this function will be executed.

The *Menu text* field property enables you to add a menu item to a field on a display and entry form. This menu item appears against the *Smartlink Information* icon. **Figure 6-64** contains the code to add the menu item to the *Currency* field.

```
Function CustomerInformation_OnLoad()
CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current Exchange Rate' > </Field>"
End Function
```

Figure 6-64: The code to set up the menu

Figure 6-65 shows the *Smartlink Information* icon that appears against a field with a menu set up against it, when the mouse pointer is moved over the field's label. **Figure 6-66** shows the menu option being displayed. If this menu option is selected and the *Currency_OnMenuSelect* function contains code, it will be executed. In this example it may link to an external web service to validate the current exchange rate.

The *Passwork mask* field property is used to prevent the display of the values being entered into, or existing values being displayed by a field. As characters are entered they are displayed against the field as an asterisk. If the field already contains a value this is also displayed as asterisks. In both cases this only affects the display, not the real values held against the field. **Figure 6-67** shows where the *Route* field has the *Password mask* set to *True* and the characters are displayed as asterisks.

The available options against the *Password mask* field property are *No change*, *True*, and *False*. These enable you to use conditional logic to set the mask on, remove it, or leave it in whichever state it currently resides. The following is the code to invoke the *Password mask* as appears in **Figure 6-67**.

```
OrderHeader.CodeObject.Route = "<Field PasswordMask='true' > </Field>"
```

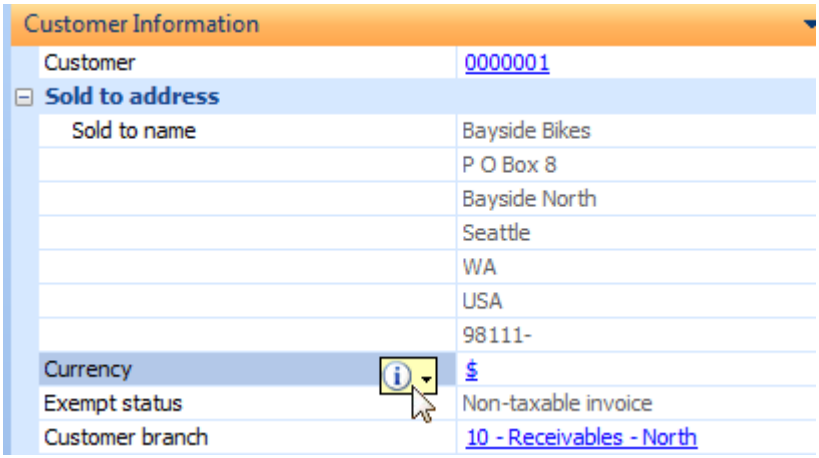


Figure 6-65: The *Smartlink Information* icon

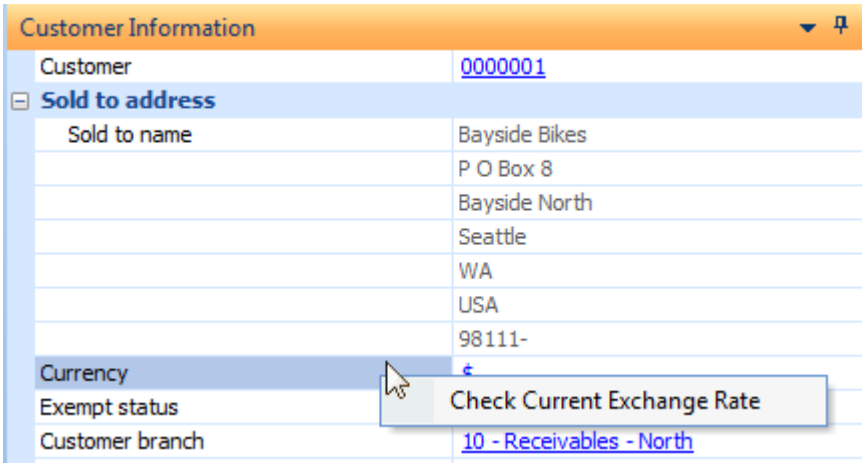


Figure 6-66: The menu displayed after clicking on the *Smartlink Information* icon

Figure 6-68 shows the same *Order Header* form as in **Figure 6-66**. However, a button has been added that shows the current value of the *Route* field. When this button is clicked it displays a message box to show the real contents of the *Route* field, to show that the field does not contain asterisks.

The *Input Mask* is set against a field and has two uses; it becomes a template to force the operator to input values in a consistent format, and it can be used to display values in a specific format. For example, you may want operators to always enter a telephone number with the area code before the number.

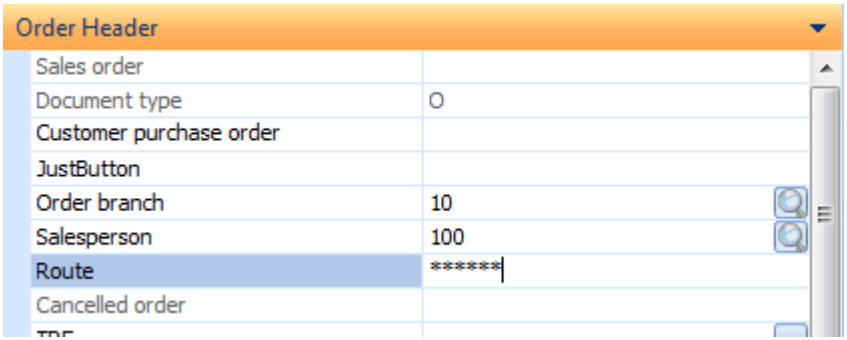


Figure 6-67: Entering a value against the *Route* field with the *Password mask* set

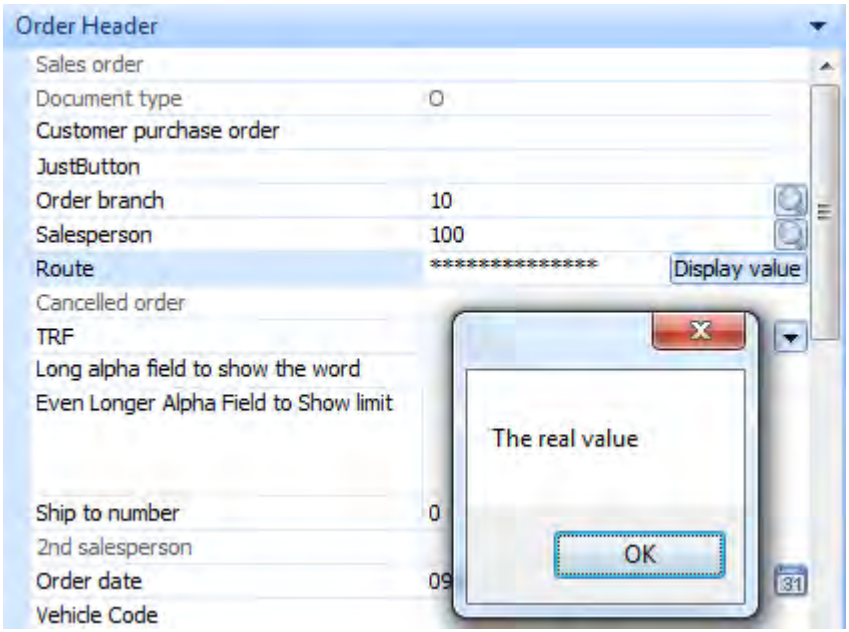


Figure 6-68: Displaying the contents of the *Route* field to show that it does not contain asterisks

Figure 6-69 shows an example where the input mask has been defined so that the telephone number must be entered this way. Because the template is shown, the operator knows what must be done, and they are also forced to enter it this way. The template is built up using a combination of characters and underscores. An underscore represents a position and the character immediately after it represents a range of numbers/letters, and their case. Any characters in the template that do not immediately follow an underscore will appear as themselves.

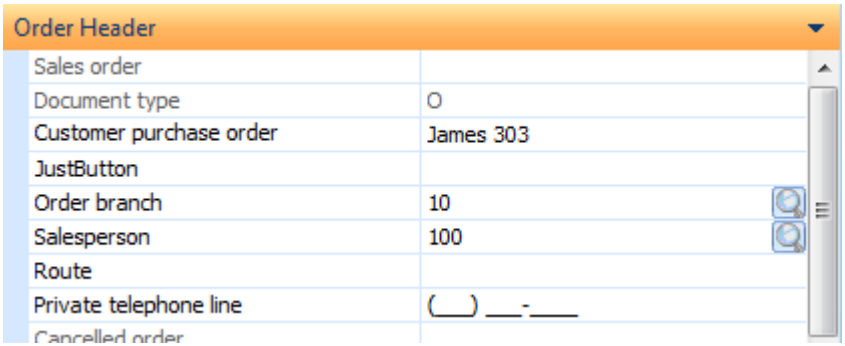


Figure 6-69: Using the *Input Mask* field property to define a template

The following is a list of the characters and their special meanings.

0 – Numeric (0-9)
9 – Numeric (0-9) or space
– Numeric (0-9) or space or + or -
L – Alpha (a-Z)
? – Alpha (a-Z) or space
A – Alphanumeric (0-9 and a-Z)
a – Alphanumeric (0-9 and a-Z) or space
& – All printable characters
H – Hex character (0-9 and A-F)
X – Hex character (0-9 and A-F) or space
> – Forces characters to uppercase (A-Z)
< – Forces characters to lowercase (a-z)

To force the telephone number to be entered you specify the entry of only numbers without spaces, so you would use underscores followed by zeros, which denote that 0-9 are the only characters allowed. The input mask to produce the telephone example above is (_0_0_0) _0_0_0-_0_0_0_0. As the parentheses are not immediately preceded by underscores, they are just for display purposes, and do not get written to the database. The same is true of the space character immediately after the second parenthesis.

The string above is added to the *Input Mask* of the *Field Properties* screen (see **Figure 6-69**). When the *Insert VBScript Code* button is clicked the following is code is added to the script where the cursor is currently positioned (which should be immediately after the variable name).

```
= "<Field InputMask='(_0_0_0) _0_0_0-_0_0_0_0' > </Field>"
```

If this was added to the form's *OnLoad* function, the next time that the program is run the template will appear (see **Figure 6-70**).

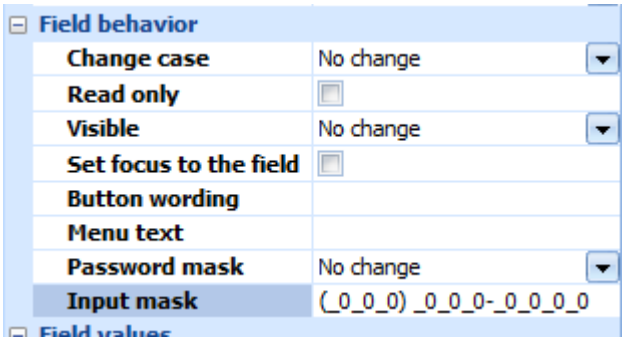


Figure 6-70: Adding the *Input mask* within the *Field Properties* screen

When focus is set on this field (by clicking on it, or tabbing to it) the cursor will be placed on the first position within this field. As the operator starts typing in numbers they will populate the template. After the third digit the cursor will jump past the parenthesis and space character to position four. If they attempt to type in a letter, it will not be allowed and an audible warning will be heard.

Figure 6-71 shows the entered telephone number. However, if this field is written back to a database, the value written is only 7144371000 as the parentheses, space, and dash are not saved.

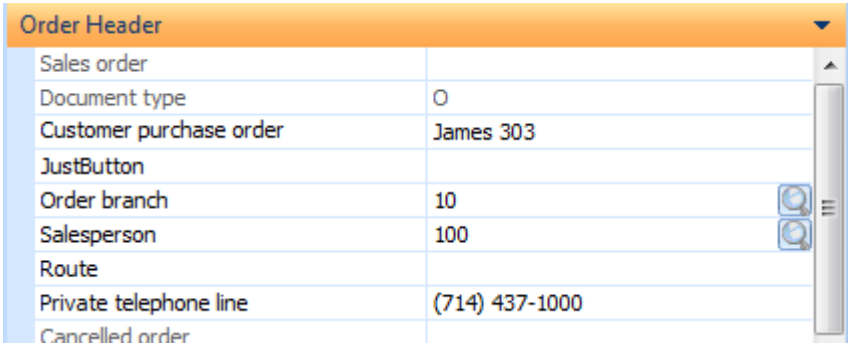


Figure 6-71: The entered telephone number.

Although this example was for capturing a value, the displaying of a value in a display or entry form works the same way.

Field Values

The first field property within the *Field values* section is *Value*. This is used to specify the value of a field in a form, or a cell in a listview. Under some circumstances it is possible to set the value against a field by just specifying the value within quotation marks, but this does not work in all cases. It is much better to add it using the *Value* field property.

The following is an example of using the *Value* field property against the *Sales Order Entry* program's *Order Header* form. If the *Salesperson* code is changed, a check is made to see if it is now *200*, and if it is, the address lines are changed.

```
Function Salesperson_OnAfterChange()
  If OrderHeader.CodeObject.Salesperson = "200" then
    OrderHeader.CodeObject.Address = "<Field Value='C/o Speedys' > </Field>"
    OrderHeader.CodeObject.address2 = "<Field Value='13 London Street' > </Field>"
    OrderHeader.CodeObject.Address3 = "<Field Value='Oxford' > </Field>"
    OrderHeader.CodeObject.address4 = "<Field Value='Oxfordshire' > </Field>"
  End If
End Function
```

Bear in mind that if you write a value to a field it may be overwritten by SYSPRO's normal processing, and that just because the value is displayed on an entry form does not mean that it will be written back to the database when the form is saved.

An example of the value not being written back is when entering a sales order in the *Sales Order Entry* program, and you have automatic sales order numbering configured. After adding the first detail line to the sales order the next sales order number is automatically allocated to this order, and appears against the *Sales order* field. If you programmatically change this number on the form, this only affects the display. If you then save the order, the sales order number is still the automatically generated number, no matter what you change on the display.

The *Tooltip* field property enables you to provide a short explanation to the operator of what is required to be entered into a field. Using the telephone example above, the *Tooltip* field property was added to this field. When the mouse pointer is placed over either the label or value portion of this field, the tooltip appears. The tooltip can be seen in **Figure 6-72**, and the code to do this appears below it. Note that this is one line of code that has wrapped around.

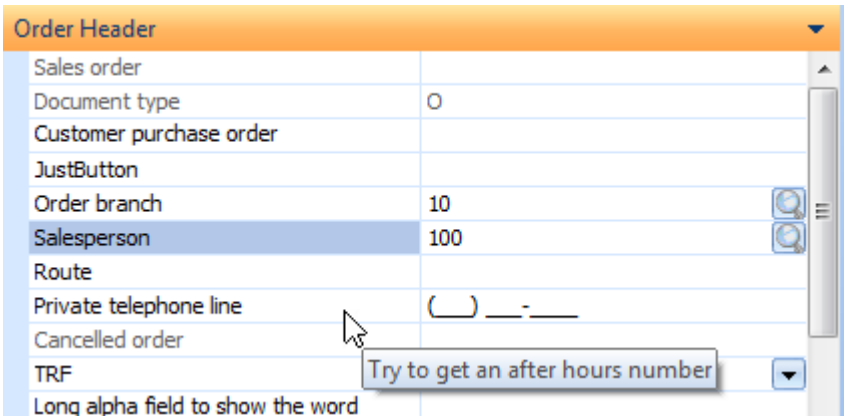


Figure 6-72: Applying a *Tooltip* to provide additional information

```
OrderHeader.CodeObject.PrivateTelephoneLine = "<Field InputMask='(_0_0_0) _0_0_0-
_0_0_0_0' Tooltip='Try to get an after hours number'  > </Field>"
```

The final field property is *Drop down list*, which enables you to create a dropdown list of values against a field on an entry form. This can also be configured against a scripted field on a display form, but this does not provide much benefit.

The items to appear in the dropdown list are added to the *Drop down list* field property, separated by semi-colons (see **Figure 6-73**).

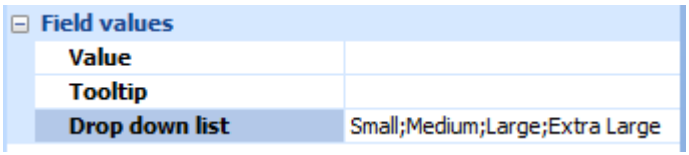


Figure 6-73: Adding the items to populate the dropdown list

When the *Insert VBScript Code* button is clicked the code is added to the chosen field, in this case the field called *TRF*. This is one line, although it has wrapped around here.

```
OrderHeader.CodeObject.TRF = "<Field List='Small;Medium;Large;Extra Large' >
</Field>"
```

The dropdown list against the *TRF* field can be seen in **Figure 6-74**. Only values that appear within the dropdown list can be selected, other entries cannot be entered manually. If you type a character into the field, if there are any entries from this list start with this character, the first item from the list starting with this character will be displayed. Once one entry is present, you can use the up and down arrow keys to move through the list.

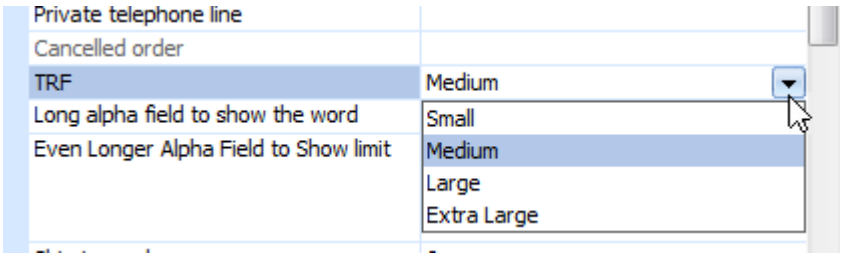


Figure 6-74: Using the dropdown list

To further enhance the dropdown list, icons can be specified for each item in the dropdown list. The list of available icons can be seen by clicking on the *Icons* button on the *Field Properties* toolbar. As

each icon is highlighted within this screen its icon number is displayed at the bottom left of the screen. This is the number that is applied to each item in the dropdown list field property (see **Figure 6-75**).

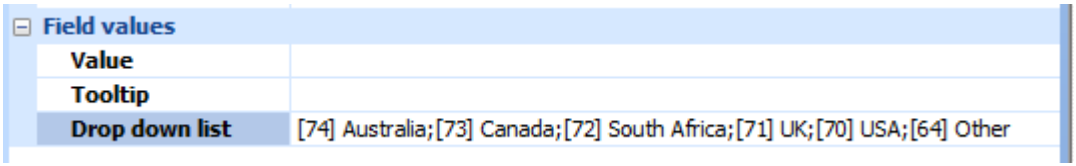


Figure 6-75: Adding items to populate the dropdown list, including icons

When the *Insert VBScript Code* button is clicked these entries are added to the selected field. The following line of code is one line that has wrapped around.

```
OrderHeader.CodeObject.TRF = "<Field List='[74] Australia;[73] Canada;[72] South
Africa;[71] UK;[70] USA;[64] Other'  > </Field>"
```

When the form is used the dropdown list appears with the options and matching icons (see **Figure 6-76**). The value of the selected item is placed against the field without the icon.

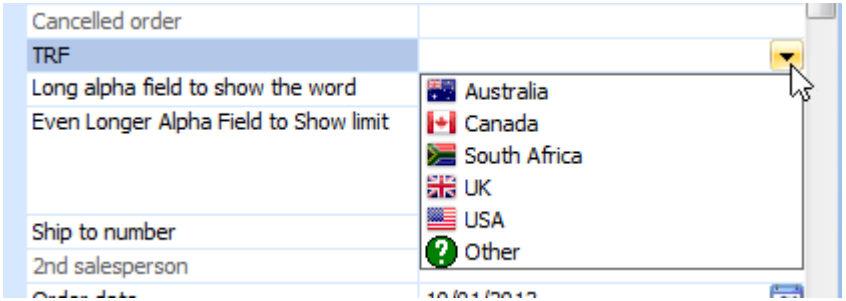


Figure 6-76: The dropdown list with values and icons

When a dropdown list already exists against a field on this form, if the *Drop down list* field property is applied to this field the existing dropdown list will be replaced. This can be useful if you want to restrict the usage of some of the items in the original dropdown list. Bear in mind that if you add extra items they may prevent the program from saving the data because of a standard SYSPRO validation check that only allows certain values in this field. Also, even if the value is saved there may be implications in other programs that do not understand this value.

Saving Changes

When you have finished modifying a script you will need to save these changes, or they will be discarded. Before exiting the screen where you edit the VBScript, it is advisable to click on the *Syntax Check* button on the toolbar. This performs a syntax check, and an explanation appears in the *Syntax Check* section above.

There is no *Save* button on the screen where you edit the VBScript. To save the changes you must first exit this screen back to the screen titled *VBScript Editor*. This can be done by selecting *File* | *Exit* from the menu bar, or just closing the window.

On the *VBScript Editor* screen are buttons for *Save*, *Cancel*, and *Edit VBScript*. Clicking on the Save button will save the changes and exit the *VBScript Editor* screen. Clicking on the *Cancel* button (or just closing the window) will prompt whether you want to save the changes and exit, not save the changes and exit, or return to the *VBScript Editor* screen.

Completely Removing a Script

There are occasions when there is a script against a form and the script is no longer required (or you just want to remove it so that you can start again). Instead of editing the script, manually removing all the lines of code, and saving it, you can click on the *Clear* button in the *VBScript Editor* screen (see **Figure 6-77**).

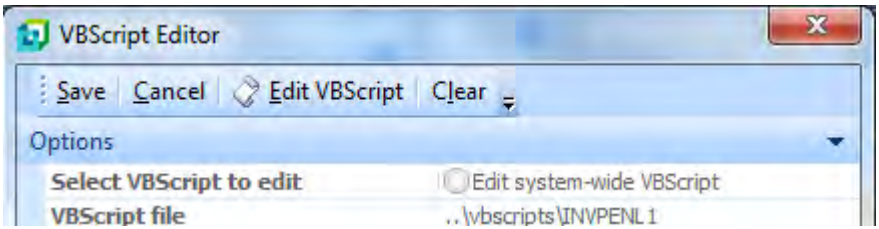


Figure 6-77: The Clear button on the *VBScript Editor* screen

The *Clear* button will prompt to make sure that the script must be removed before removing all the lines of script against this form (including any against fields). You can then either add more code, or click on the *Save* button to save the empty script.

Basic Debugging and Writing out Values to a File

One of the most important things to do when writing a script is to document your script while it is still fresh in your mind. This will help you, or someone else, to understand decisions that were made, and why they were made. You should also consider adding other details such as the date that the script was created, who created it, and an overview of what it should be doing (and not doing).

You should also use descriptive variable names, and meaningful names for any functions that you create. This can make it so much easier to follow the script at a later date.

Writing Out a Message Box

One of the simplest ways of debugging is to write out messages so that you can see the route that you are taking through the script. You can also display the contents of variables with a message box so that you know that they contain meaningful values, as can be seen in **Figure 6-68**. You can also see if you are entering *IF* statements, how many times you loop through *For/Next* statements, etc.

However, there is a limit of 1023 characters that can be displayed by a message box. So if you intend to display the XML output from a business object, unless the output is something basic like confirming that a sales order was created, the message is likely to be truncated. For example, if you wanted to view the output of the customer or stock query business object to see if a particular element or node is present, the 1023 character limit means that the message box will be truncated before any detail information is displayed.

To be able to review larger amounts of data it is advisable to write out the contents of the variable, or output from the business object, to a text file. Below is a sample code snippet that will write out the contents of the *XMLOut* variable to a file called `Testfile.xml` in a folder called `C:\Messages`.

This snippet can be added to your VBScript to write out these values. However, as VBScript that is embedded in SYSPRO is run on the client machine, you must make sure that the nominated folder exists, and that SYSPRO has permission to write to this folder. If you need to write out multiple files in one script you can reuse the code, just make sure that you change the variable names (so that you don't get a warning that you are trying to reuse a variable) and the name of the file being output (so that you don't overwrite the file).

Also remember to comment out this code, or delete it from your script, when you have finished, as you don't want this overhead when you do not need the information.

```
Dim fs, a
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.CreateTextFile("c:\messages\testfile.xml", True)
a.WriteLine (xmlout)
a.Close
```

Other Suggestions

The following are other suggestions that will make it easier to debug your script.

Always use *Option Explicit* at the beginning of your script. It may seem like extra effort to have to declare variables before you can use them, but you will very soon find typos in variable names that you had not noticed.

Use an apostrophe to comment out any *On Error Resume Next* statements as you want to see which lines are causing the original error. If you comment them out it is easy to reinstate them later.

Invoking a Debugger

There are some occasions where you need to be able to use an external debugger and there are many documents on the internet explaining how to call a debugger when a script fails. Many of these use either *Microsoft Visual Studio* or *Microsoft Script Debugger*.

Microsoft used to ship the *Microsoft Script Debugger* as part of *Microsoft Office*, but this no longer appears to be included in *Office*. It is available from the following Microsoft web page, although it is no longer a supported product.

http://www.microsoft.com/en-us/download/details.aspx?id=22185

Instructions on how to configure this are available from the *Knowledge Base* article at the following address:

http://support.microsoft.com/kb/308364

To get the *Microsoft Script Debugger* to work on my Windows 7 machine I first installed it, then made the following changes to my registry. Note: if you are not familiar with making changes to the registry, do not perform these steps. Instead, get someone who is familiar to assist you. You are also strongly advised to make a backup of the registry before changing it in any way.
The document stated that the script looks at the following location to find out which debugger to use.

```
HKEY_CLASSES_ROOT\CLSID\{834128A2-51F4-11D0-8F20-00805F2CD064}\LocalServer32
```

When I got to the *CLSID* section there was no *Key* below it with this name. I created this *Key*, and then a *LocalServer32* one beneath that. Against the *Default* string value I entered the address to invoke the *Microsoft Script Debugger* on my machine. This is the full address to the executable, including the name of the executable (in my case *C:\Program Files (x86)\Microsoft Script Debugger\msscrdbg.exe*). The full address to the entry in the registry, and the entry that was created, can be seen in **Figure 6-78**.

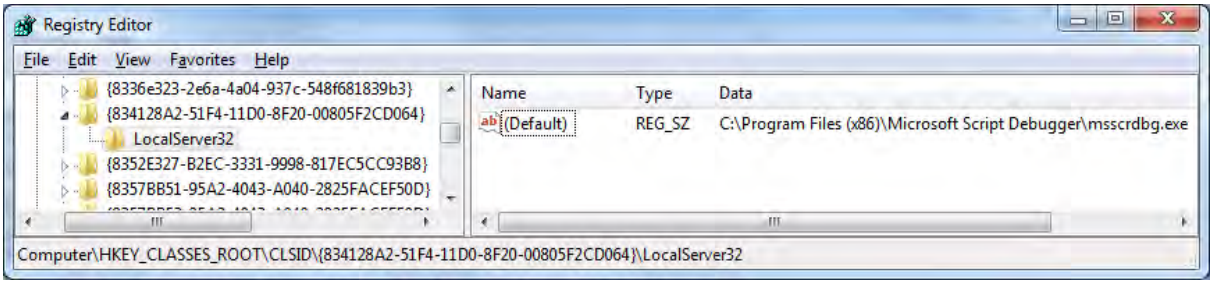


Figure 6-78: Adding the registry entry that points to the *Microsoft Script Debugger*

The next change to the registry was to enable *Just in Time debugging*. The registry entry is:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows Script\Settings\JITDebug
```

The default setting for this is 0, and must be changed to 1 to enable debugging (see **Figure 6-79**). To be able to use the debugger, it must be running on the client before SYSPRO is run. When SYSPRO hits a script error, a new window is opened in the debugger and populated with the script. The line causing the problem is highlighted (see Figure **6-80**).

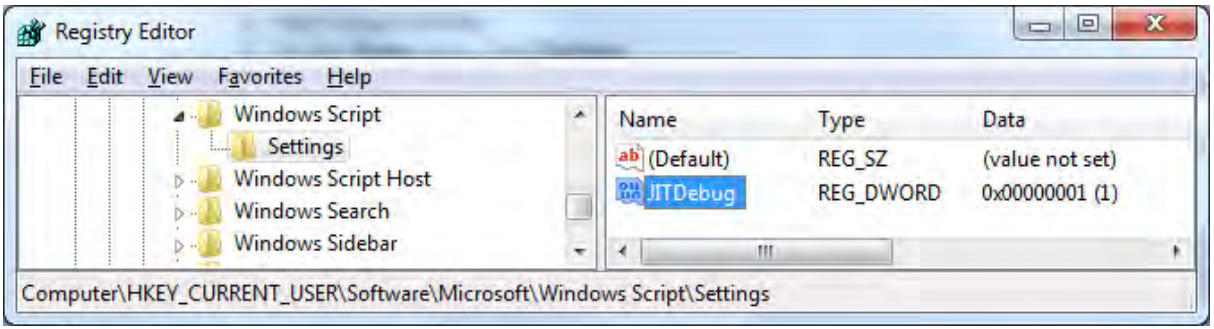


Figure 6-79: Changing the value of the *JITDebug* setting

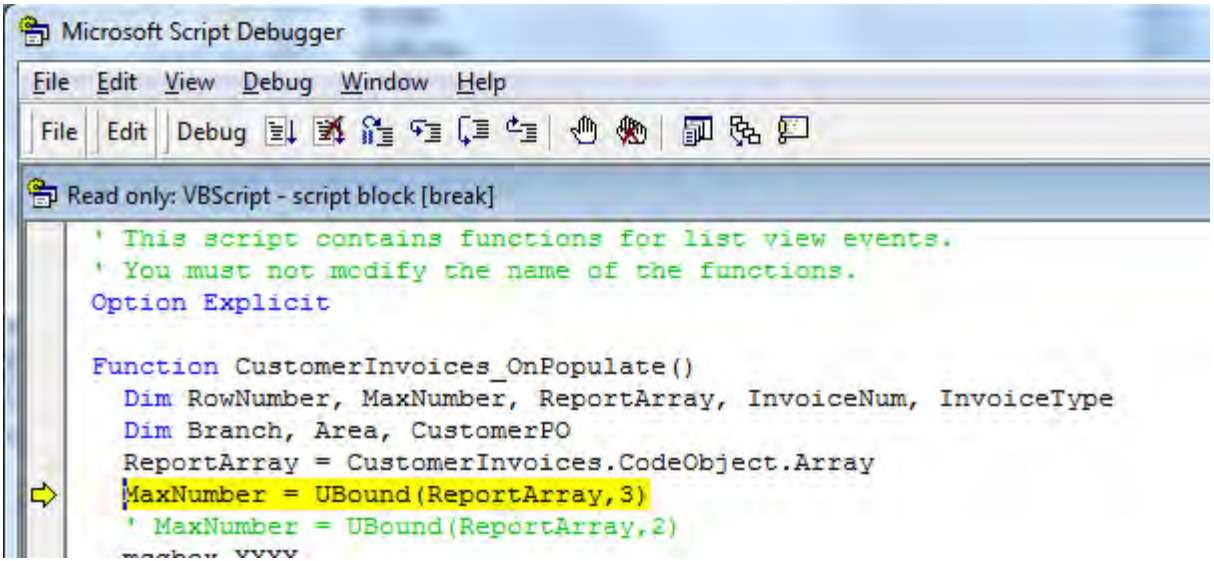


Figure 6-80: The *Microsoft Script Debugger* populated with the script from an *OnPopulate* function

Script Limitations

Prior versions of SYSPRO had size limitations for the VBScripting. Up to SYSPRO 6.1 the maximum size that a VBScript could be (including all *VBScript Modules* that are specified in the main script) was 100KB. For SYSPRO 6.1 Service Pack 1 this limit was increased to 200KB. From SYSPRO 7 there is no limit.

Chapter 7 - Forms

The power tailoring of forms using scripting is done using form and field events. Form and field events were covered in detail in Chapter 5. The scripting that is applied to these events enables you to change the look and feel of the captions and values associated with fields on a form, as well as the values themselves.

The captions cannot be changed using VBScript code, but they can be changed using either the *Field Properties* or *Field Properties for all Forms* screens. These screens are available from the context-sensitive menu that appears when you right-click on the form, and were covered in great detail in Chapter 4 of the first Power tailoring book. In addition, the captions can be changed using the language translation facility. The translation facility was covered in detail in Chapter 8 of the first Power Tailoring book.

Form Events

Display Forms have two form events, *OnLoad* and *OnRefresh*. All *Entry Forms* have at least five form events, *OnLoad*, *OnRefresh*, *OnStartEdit*, *OnStopEdit*, *OnSubmit*, and some also implement an *OnAfterSubmit* event.

OnLoad
The *OnLoad* event for a form typically fires only once for each run of the program. It fires the first time that the form is going to be populated with data, not as the program loads. An *OnLoad* event is typically used to invoke things that must remain for the whole run of the program, or to set an initial state. For example, if a field on the form needs to have a button against it, the definition of this button would appear against the form's *OnLoad* event. Also, if the initial state of a field on an entry form must be read-only, this would be set against the form's *OnLoad* event.

OnRefresh
The *OnRefresh* event for the form fires each time that the contents of the whole form must be refreshed. The first time that a form is going to be used its *OnLoad* event fires, and this is followed immediately by its *OnRefresh* event. Whenever the program needs to refresh the contents of this form the *OnRefresh* event will fire again.

An example of this is when adding a stock detail line to a sales order. When you have completed adding the order header information and select to add detail lines, the *OnLoad* event for the *Stocked Line* form will fire (if this is the first time that this form has been used for this sales order) and this is followed immediately by its *OnRefresh* event. When you enter the stock code and tab off this field, SYSPRO retrieves the details for this stock code/warehouse combination, and once it has populated the form it fires the *OnRefresh* event again.

The *OnRefresh* event is typically used for changing the state of a field, or causing a change on another form. For example, if you wanted to change the background color of a field to highlight that its value fell outside of a target range, you could do this when the form refreshes (and just as importantly, undo the change when the value falls back within the range).

Other events can also be used to fire a form's *OnRefresh* event; it doesn't have to be fired by the program. If you have created a *Customized Pane* to display some values, you could use the *OnRefresh* event of one of the standard forms of the program to fire the customized pane's *OnRefresh* event. *Customized Panes* are covered in more detail in a later chapter.

OnStartEdit
The *OnStartEdit* event is invoked as you start editing the first field on a form for the first time, or when the program has placed the cursor on the form for the first time. This will always be after the form has fired its *OnLoad* and *OnRefresh* events, but not necessarily immediately after these have fired. This could be because other forms still have to fire their *OnLoad* and *OnRefresh* events first, or because focus is not automatically set on this form after its *OnLoad*/*OnRefresh* events have fired (maybe focus is set on another form).

When you have a program where you create the header information followed by the detail lines (such as the *Sales Order Entry* program), you can have an *OnStartEdit* event against the *Order Header* form that fires as you start entering the header information. The *Stocked Line* form can have its own *OnStartEdit* event, and this would only fire when its form starts being edited.

Note: both the *OnStartEdit* and *OnStopEdit* events should be used sparingly, and you should always check that they fire at the appropriate times under all circumstances before using them. They are considered "legacy" and exist only to give backwards-compatibility.

OnStopEdit
The *OnStopEdit* event fires when the program knows that you have stopped editing the form and you will not be editing it again. The exact point that this is invoked is program-specific.

In the case of the *Sales Order Entry* example above, as you click on the *End Order* button on the *End Order* screen, the *Stocked Line* form's *OnStopEdit* event fires, followed by the *Order Header* form's *OnStopEdit* event. If you then add/maintain another order while still in the program, the *OnStartEdit* event for the *Order Header* form will fire when focus is set on the *Order Header* form.

Note : both the *OnStartEdit* and *OnStopEdit* events should be used sparingly, and you should always check that they fire at the appropriate times under all circumstances before using them. They are considered "legacy" and exist only to give backwards-compatibility.

OnSubmit
The *OnSubmit* event is invoked when the operator attempts to save the data that has been entered on an entry form, and saving the data would be successful. An example is when saving a stocked line on a sales order, the *OnSubmit* event is invoked each time the form's data is saved, once for each detail line. If the *Save* would fail for any reason (i.e. it would not meet a specified minimum price percentage above cost) the *OnSubmit* event will not be fired.

Another example is when saving the whole sales order, the *Order Header* form's *OnSubmit* event is fired as the sales order is ended (unless the *Save* would be unsuccessful, in which case it would be invoked when the *Save* is retried and would be successful). An *OnSubmit* event can be used to validate data (that is on this form, another form, or externally) and if not present/within requirements, it can prevent the process from completing by setting the *OnSubmit* function status to *false*.

Setting the *OnSubmit* function to false does not prevent the code against this function from running. The code against this function will complete before control is handed back to the program, at which time the *false* status will prevent the *Sales Order Entry* program from ending the order.

OnAfterSubmit
The *OnAfterSubmit* event has only been implemented in certain places such as when adding/maintaining a detail line in the *Sales Order Entry* program. Whereas an *OnSubmit* event is invoked while the sales order line is still displayed, and before the data is saved to the server, the *OnAfterSubmit* is invoked after the server program has saved the data, but before the form is cleared. Unlike the *OnSubmit* event, there is no way to set the *OnAfterSubmit* event to false to prevent it from happening, as the data has already been written away.

The sequence of events followed is the *OnSubmit* event invokes first, a check is made that all custom form fields are valid, the business logic checks the validity of the fields, the data is saved, the *OnAfterSubmit* event is invoked and then, if successful, the form clears.

Field Events

All fields on forms have their own set of *Field Events*, and the list of available field events depends on whether it is a display or entry form. Fields on a display form only have an *OnMenuSelect* event. Those on an entry form have *OnAfterChange*, *OnBeforeChange*, *OnButtonClick*, *OnGainFocus*, *OnLostFocus*, *OnMenuSelect*, and *OnLinkClicked* events.

OnMenuSelect
The *OnMenuSelect* event requires that a menu be defined for this field first. This is normally defined against the form's *OnLoad* or *OnRefresh* functions. Only one menu option can be actioned for a field. If a menu is defined for a field in both the *OnLoad* and *OnRefresh* events, the one against the *OnRefresh* event will be displayed as its definition will overwrite that of the *OnLoad* event (because it fires later in the sequence).

Once defined, the menu becomes available when you move the mouse pointer to the left of the value against the field, and click on the *Smartlink Information* icon that appears. The *OnMenuSelect* event fires when you click on the displayed menu item.

A section called *Menu Sample* appears later in this chapter and contains an example of how to configure a menu option.

OnAfterChange
The *OnAfterChange* event fires after the data has been changed in memory, and is invoked after the *OnLostFocus* event. Like the *OnLostFocus* event, it is possible that a change to the value may be overwritten by data returned by the program after a call to the server. The *BeforeChangeValue* system variable contains the value of the field before the operator changed it.

OnBeforeChange
The *OnBeforeChange* event is invoked after the operator has changed the value against a field and has moved off the field, but before the data has been changed in memory. This allows for the script to validate that the entered value is acceptable (i.e. falls between a certain range of values/is greater than the contents of another field, etc.) and to revert back to the previous value if it does not match this requirement by setting the *OnBeforeChange* function to *false*. The *OnBeforeChange* event fires before the *OnLostFocus* event.

OnButtonClick
The *OnButtonClick* event will only fire if you have created a button on the form for this field. When this button is clicked the event is fired. This does not relate to a browse button that appears automatically against either a master field that is present by default, or a master field that has been added to this form.

The code to set up the button can be added to either the form's *OnLoad* or *OnRefresh* function. When the event fires (and the VBScript against the function is run) the button appears against the form. If a button (such as a browse button) already exists against this field, the button that you added will appears alongside the standard one, and there will be two buttons for this field.

A section called *Button Sample* appears later in this chapter that contains an example of how to configure a button.

OnGainFocus
The *OnGainFocus* event is fired when focus is set on the value portion of the specified field. Each time that focus is set on the value portion of this field, this event will fire (i.e. after focus has been set somewhere else, then back on the value portion of this field again).

Care should be exercised when using both the *OnGainFocus* and *OnLostFocus* events. It is possible to cause a loop where the one event causes the other event to fire, which causes the first to fire, etc. Where possible rather use the *OnAfterChange* event.

OnLostFocus
The *OnLostFocus* event is fired when focus was set on the value portion of a field, and you click anywhere else. This can be on the caption portion of the same field, another field on this form, another form on this window, another window altogether, or tab off of the field. It is possible that if a new value was entered, it is overwritten if the program returns data from the database on the server, and this is one of the fields that the program refreshes from the database.

Care should be exercised when using both the *OnGainFocus* and *OnLostFocus* events. It is possible to cause a loop where the one event causes the other event to fire, which causes the first to fire, etc. Where possible rather use the *OnAfterChange* event.

OnLinkClicked
The *OnLinkClicked* event fires when you click on a predefined hyperlink on an entry form. Although you can define that hyperlinks appear on a display form, *OnLinkClicked* events are not available on this form type, so this event only relates to fields that are hyperlinked automatically by the program.

Hyperlinked fields on entry forms are used to call up other programs. If code appears against the *OnLinkClicked* function for a field, this event is fired before the other program is called. For example, the *Sales order notes* field on the *Order Header* form of the *Sales Order Entry* program has a hyperlink against it. When this hyperlink is clicked the *Notepad* program is called. If the *notes3_OnLinkClicked* function contains code to display a message box, when this hyperlink is clicked the message box is displayed, and only once this has been acknowledged by the operator is the *Notepad* program displayed.

The *OnLinkClicked* function can be prevented from completing by setting it to false, in the same way as with the *OnSubmit* function mentioned above, so conditional logic can be added. Using the example of the *Sales order notes*, the line of code that appears below would prevent the program associated with the hyperlink from being run.

```
notes3_OnLinkClicked = false
```

Adding Fields

There are several different types of fields that can be added to standard display and entry forms. These are *Standard Fields*, *Custom Form fields*, *Related fields*, and *Scripted fields*. Apart from the *Standard Fields* which have their own option, the other three field types are added using the *Field Selector*.

In addition, fields that would be present on the form by default, and have been removed by an operator or the administrator, can be reinstated on the form using the *Show Captions* option on the context-sensitive menu that is displayed when you right-click on the form.

Standard Fields

The list of Standard Fields is the same for all form types. It consists of the *Company date*, *Company name*, *Current printer*, *Group*, *Operator*, and *Role*. The *Standard Fields* are added to a form using the *Insert Standard Field* option of the context-sensitive menu that appears when you right-click on a form (see **Figure 7-1**).

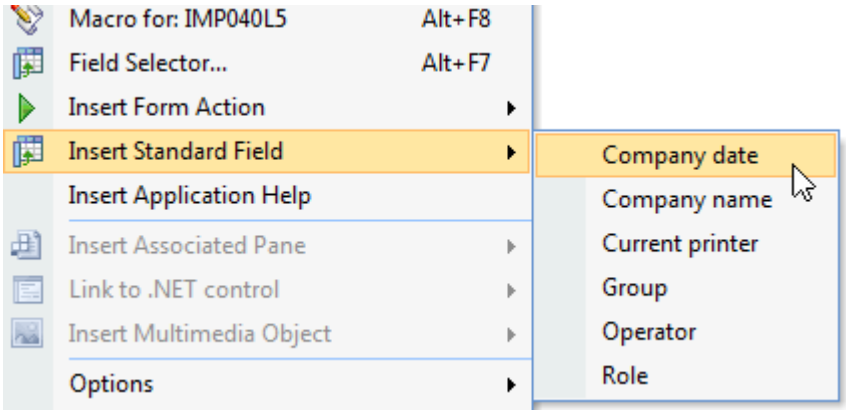


Figure 7-1: The *Insert Standard Field* option on a form's context-sensitive menu

Adding multiple *Standard Fields* at the same time is simple as the list of fields remains on the screen as you click on them. A *Standard Field* that has already been added to a form will appear greyed-out with a checkmark against it if the *Insert Standard Form* option is selected again. As the values for these fields are not specific to the form/data being displayed they are populated as the form loads, so are available immediately (see **Figure 7-2**).

If you just need to use the contents of the *Standard Fields* in your form's VBScript you do not need to add these fields to your form, as their contents are all available in variables within the *System Variables* section of the *Variables* pane.

Standard Fields can be removed from the form using the *Hide Caption* option from the context-sensitive menu that appears when you right-click on the form, or by using the *Shift+Del* shortcut keys. As soon as they are removed from this form they no longer appear greyed-out, or with a checkmark against them in the *Insert Standard Fields* list, and are available for use again.

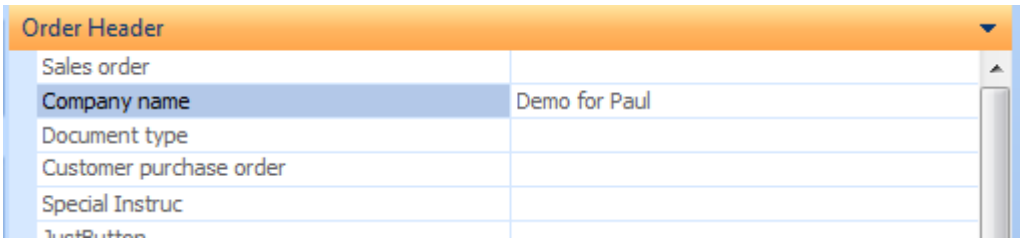


Figure 7-2: The *Company name* field being populated as the form loads

Custom Form Fields

A *Custom Form* provides a mechanism of adding fields to certain master and transaction tables. The fields are not added to this table, but are added to other tables, and associated with this record in this table. Custom form fields can be a combination of alphanumeric, numeric, or date, and each table can be associated with up to 150 fields. Prior to SYSPRO 7 this was limited to 100 fields.

If there is a custom form type linked to the information on this form (such as *Sales Order* that is linked to both the header section of the *Sales Order Entry*, and *Sales Order Query* programs) the custom form fields associated with this custom form can be added directly onto the display or entry form. These custom form fields appear under the *Field Selector* option of the context-sensitive menu (or by using the *Alt+F7* shortcut keys directly from the form) and can be dragged onto the form.

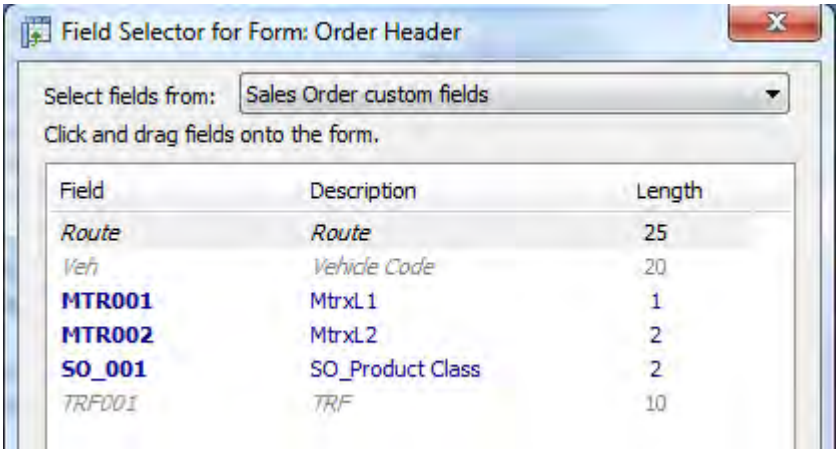


Figure 7-3: The *Field Selector* screen

Figure 7-3 shows the *Field Selector* screen that appears when it is called from the *Sales Order Entry* program's *Order Header* form. The *Field Selector* screen's title specifies the screen from which it was called. The dropdown list defaults to the custom form fields associated with this form, if a custom form is linked to the form from which the field selector was called. The other possible options for this dropdown list are the *Related Fields* and *Scripted Fields*. Previously-defined custom form fields will be shown on this screen. If any of these fields already reside on the form, its field name and description is italicised (such as *Route*, *Vehicle Code*, and *TRF*).

An in depth explanation of how to add custom form fields, the different types, lengths, and validation properties, appears in Chapter 4 of the *Power Tailoring Book 1*.

Related Fields

When a program is run for the first time, each form will display the fields that SYSPRO Development (hereafter referred to as the developer) has pre-defined that will most likely be required by the user. If there are only a small number of fields in the table, the form will contain all of these fields. If the table contains a significant number of fields, the developer will typically make those not being displayed on the form available via the *Show Captions* option from the context-sensitive menu that is displayed when you right-click on the form. If the main table for the form contains a lot of fields, the developer will make the most significant ones available on the form, and the other fields are made available as *Related Fields*.

The dropdown list at the top of the *Field Selector* screen contains the name of the master table associated with the form from which the *Field Selector* was called. In **Figure 7-4** the related fields entry contains *Sales Order Master fields*, as the *Field Selector* was called from the *Order Header* form of the *Sales Order Entry* program, and the *Sales Order Master* table is the primary table associated with this form.

The list of fields from this table appears in a listview below the dropdown list, along their matching descriptions and field length. Any field that already appears on the form (or against the *Show Captions* option) will be displayed italicised. Any of the fields that appear in this list that is not italicised can be dragged from this list onto the form on which you resided before calling up the *Field Selector*. When you release the mouse button the field will be inserted on the form and its matching entry in the *Field Selector* will become italicised.

The ability to add related fields to a form is so that the contents of the master file can be displayed. If a related field is added to an entry form, the field (both caption and any value that is displayed) will appear as a read-only field. Even if you manage to change the displayed value programmatically via VBScript, the changed value will not be written back to the master file.

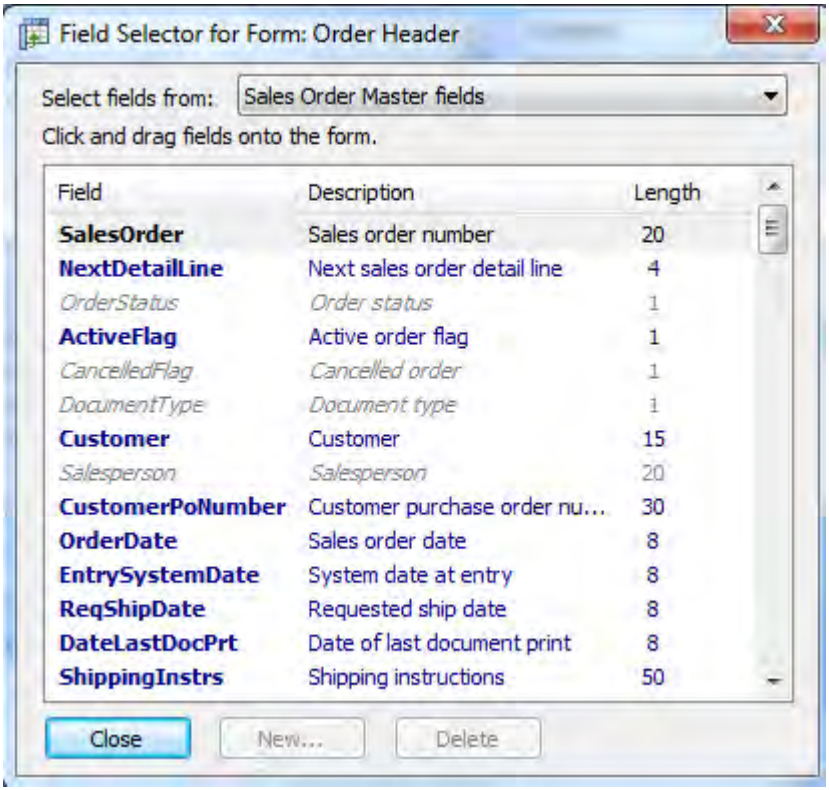


Figure 7-4: The *Field Selector* showing the list of *Related Fields* this form

Scripted Fields

Scripted Fields are placeholders that are used for displaying data, and their values are typically updated via VBScript. Unlike *Custom Form Fields* and *Related Fields*, *Scripted Fields* are not specific to a form. When the *Scripted fields* option is selected in the *Field Selector*, the displayed list of fields could have been created against any form.

Figure 7-5 shows the *Field Selector* screen that was called from the *Order Header* form of the *Sales Order Entry* program. The fields listed in the listview were created against various different forms, but are all available for use against the *Order Header* form. Those that appear italicized are already present on the *Order Header* form. *Scripted Fields* are added to the *Order Header* form by dragging them from the *Field Selector* to the required location on the form and releasing them. As they are added to the *Order Header* form they will appear italicized in the *Field Selector*. New *Scripted Fields* are added by clicking on the *New* button at the bottom of the Field Selector, which calls the *Scripted Field* screen (see **Figure 7-6**).

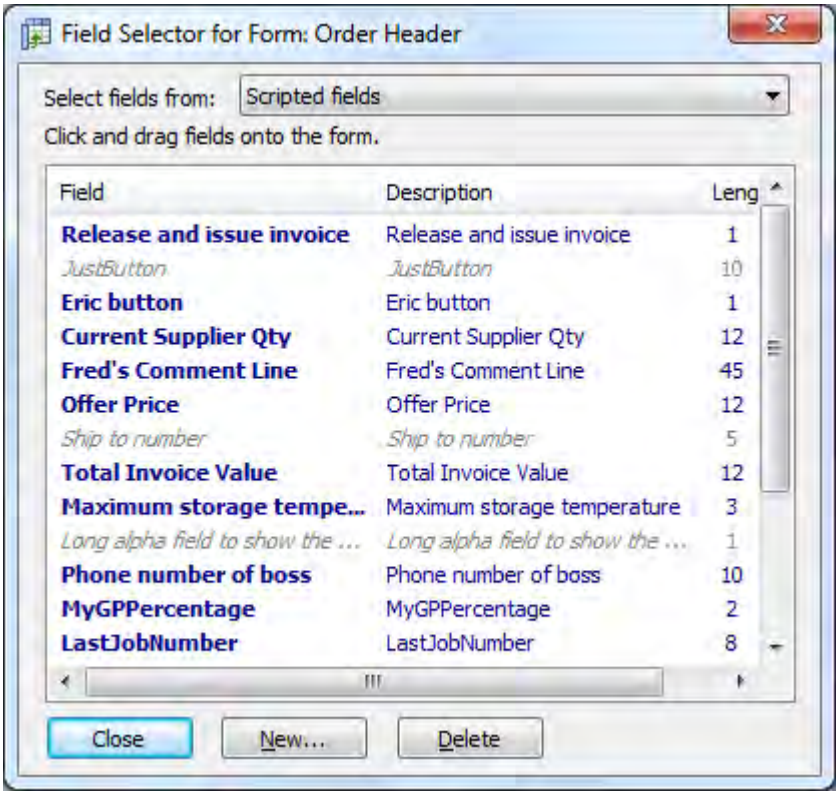


Figure 7-5: The *Field Selector* showing the *Scripted fields*

Figure 7-6: Adding a new *Scripted Field*

When adding a scripted field you must supply a description and the field type. The field's type can be defined as *Alpha*, *Numeric* or *Date*. An alpha field can be up to 100 characters long, a numeric field can be up to 12 digits to the left of the decimal place with up to 6 decimal places, and a date will always be 8 digits long. The scripted field can also be specified as *read-only*, meaning that the value cannot be overwritten by the operator using the keyboard. **Figure 7-6** shows the adding of a numeric scripted field called *Orders This Week* that has a Length of 2 and no decimals. This means that the value of this field can be between 0 and 99.

A scripted field can be removed from the list of scripted fields in the field selector by highlighting it and clicking on the *Delete* button. As this list of scripted fields is not specific to this form, once deleted this scripted field will not appear in the list of scripted fields against any form. Before a scripted field is deleted a warning message and prompt are displayed (See **Figure 7-7**).

If you delete a scripted field from the *Field Selector*, and this scripted field is still in use on a form, this scripted field will remain on this form and continue to work.

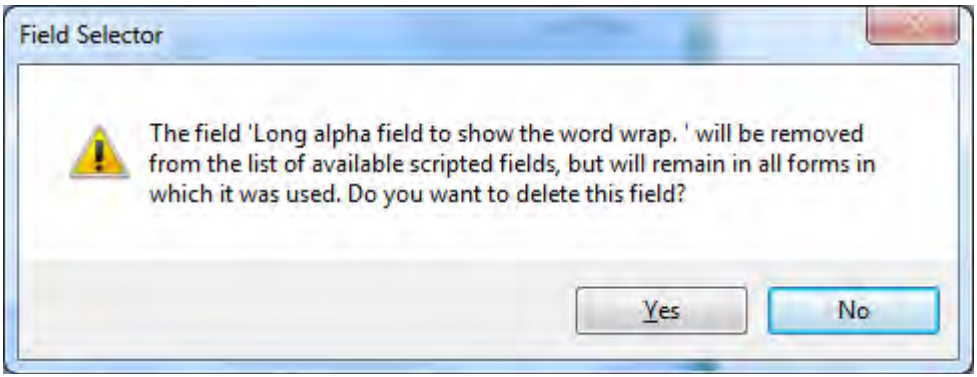


Figure 7-7: The warning message displayed when deleting a *Scripted Field*

Changing the Appearance of a Field's Value

The appearance of a field's value can be changed very simply in VBScript by applying a *Field Property* to the variable name associated with the field. There are 23 field properties that can be set, some are specific to the field's value, some are specific to a field's caption, and the rest affect the whole field. Within the list of those that affect a field's value there are differences between those that can be used with an entry form and those that can be used with a display form.

For a complete list of which field properties affect field values and captions on display and entry forms, see the section *Which Field Properties Can Be Applied to Field Values/Captions by Form Type* below.

Example 1 – Displaying a Customer's Name in Bold

In the following example there was a requirement to display the customer name in the *AR Customer Query* in a bold font. This is done using a *Field Property*. The first thing that must be decided is the appropriate time to set the field property. In this case, as it is required for all customers, the appropriate time is the form's *OnLoad* event.

After calling up the *AR Customer Query* program, right-click anywhere within the *Customer Information* form, and select *Macro for: ARSPENL6* from the displayed context-sensitive menu. The *VBScript Editor* screen is displayed. The same result can be achieved by highlighting the *Customer Information* form and using the keyboard shortcut *Alt+F8*.

Figure 7-8 shows the *VBScript Editor* screen that will appear if the operator is not associated with a security role. This screen has two panes: *Options*, and *Available Events*. The *Options* pane displays the name of the form, the field on this form that was highlighted when the operator called up the *VBScript Editor*, as well as the name and location of the script being edited. As this is a display form the *Available Events* pane shows that there is one *Field event* (*OnMenuSelect*), and two *Form events* (*OnLoad* and *OnRefresh*). As none of these events have a script icon against them, there are no functions present in the script.

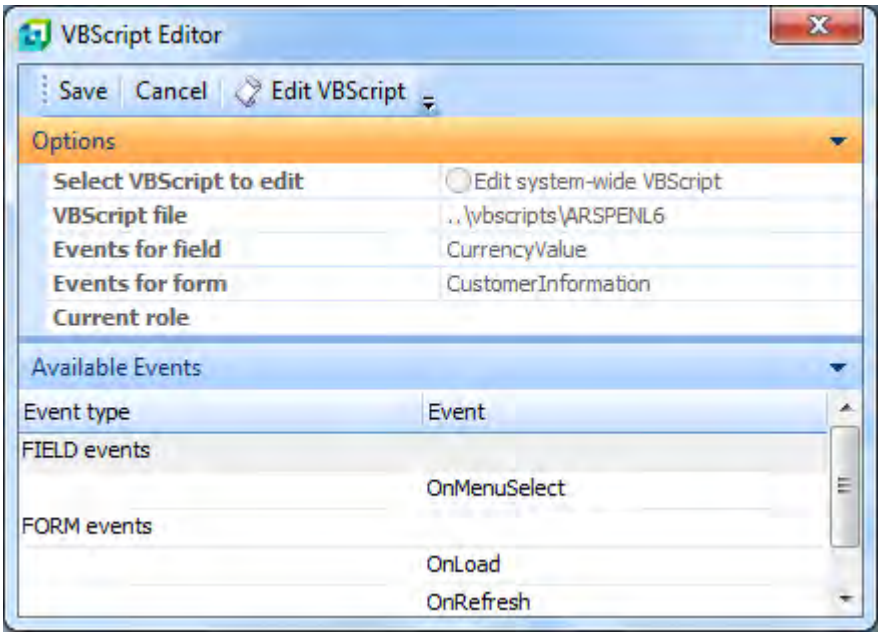


Figure 7-8: The *VBScript Editor* when logged in using an operator not associated with a role

Figure 7-9 shows the same *VBScript Editor* screen, but for an operator that is associated with a role. As only the *Options* pane is different from **Figure 7-8**, only this part appears here. The two differences are that the *Current role* is displayed, and the operator can choose to edit the script associated with this role, or the system-wide one.

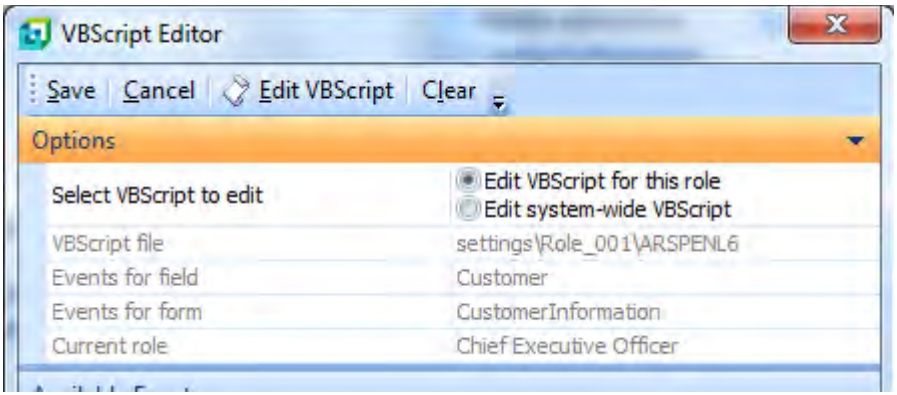


Figure 7-9: The *VBScript Editor* when logged in using an operator associated with a role

Highlight the *OnLoad* event and click on the *Edit VBScript* button (or double-click on the event name) and the screen will be displayed where you edit the script. The layout of the displayed screen has been covered in detail in *Chapter 6*, so will not be covered here. Within the main pane the function will be created for you, and the cursor placed within it (see **Figure 7-10**).

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnLoad()
6
7  End Function
```

Figure 7-10: The newly-created *OnLoad* function

Within the *Variables* pane, locate the *CustomerInformation* section and expand it by clicking on the plus sign alongside it (see **Figure 7-11**).

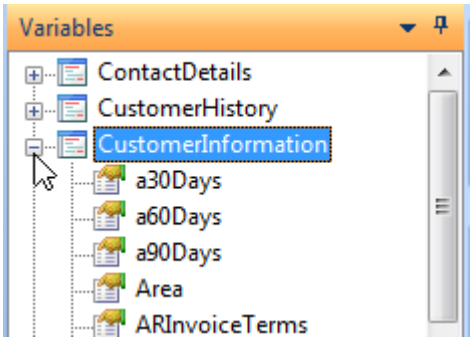


Figure 7-11: Locating and expanding the *CustomerInformation* section

Locate the *Name* variable within the *CustomerInformation* section, and double-click it. The full name of this variable will be added to your code at the last position of the cursor within this pane (see **Figure 7-12**).

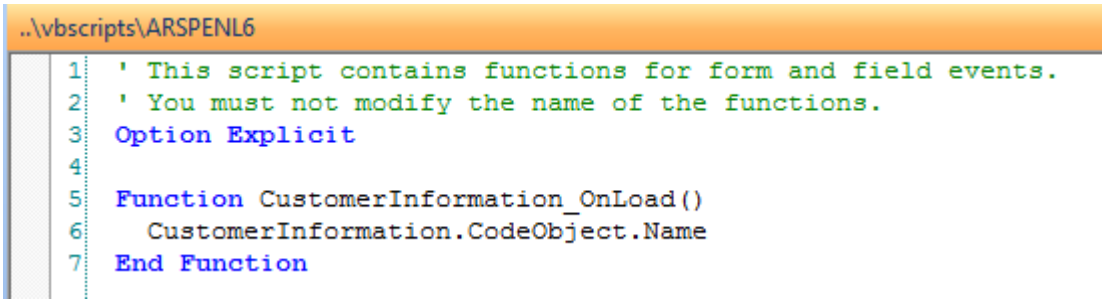


Figure 7-12: The full variable name added to the script

Immediately after the variable name, add a space, an equal sign, and another space. Within the *Field Properties* pane, locate the *Bold* option and check it, then click on the *Insert VBScript Code* button (see **Figure 7-13**).

The code to add this field property to the variable will be inserted at the last place the cursor was present on the script pane. The completed script appears below.

```
Function CustomerInformation_OnRefresh()
  CustomerInformation.CodeObject.Name = "<Field IsBold='true' > </Field>"
End Function
```

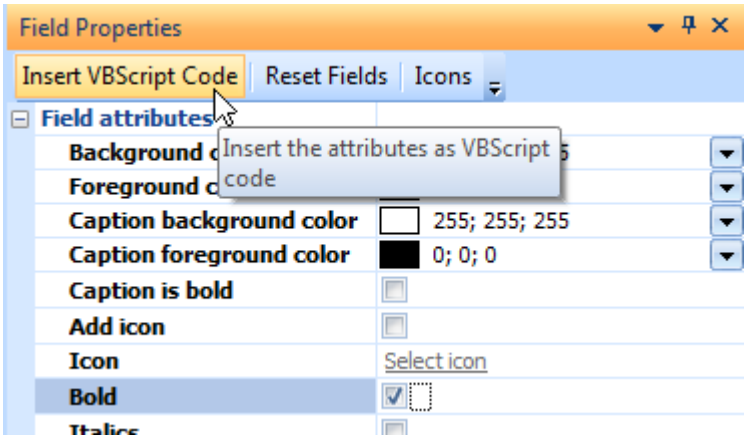


Figure 7-13: Using the *Insert VBScript Code* button to add the selected field property

Click on the *Syntax Check* button to make sure that there are no typos. Exit this screen back to the *VBScript Editor* and the *OnLoad* event should have a script icon against it. Click on the *Save* button, the changes to the script will be saved and you will be returned to the *AR Customer Query* program. As this code is against the *OnLoad* event, you must exit the program and reload it. When you next run it (as the code is against the *OnLoad* event which fires before a customer's details are displayed) the customer's name will appear in bold (see **Figure 7-14**).

Figure 7-14: The *Name* field with the *Bold* field property set

Example 2 – Changing a Field's Appearance Depending on a Value
This is also a simple appearance change, but using conditional logic. Because the conditional logic must perform a check each time the customer account code changes, the logical place for this code is against the form's *OnRefresh* event. The logic is that if the *Credit Status Code* is 6 or higher the field's background color must be changed. If this code is less than 6, the background must be the default color.

Call up the *AR Customer Query* program, right-click on the *Customer Information* form and select *Macro for:ARSPENL6* from the displayed context-sensitive menu. The *VBScript Editor* screen will be displayed. Double-click on the *OnRefresh* event name and you will be taken to the pane where you

edit the VBScript. If it does not already exist, the *CustomerInformation_OnRefresh* function will be created for you, and the cursor will be placed within this function.

To test the *Credit Status Code* you use an *IF* statement. The first thing to add is the word *If*, followed by a space. Locate the *CustomerInformation* section of the *Variables* pane and expand it by clicking on the plus sign alongside it. Within the *CustomerInformation* section locate the *CreditStatusCode* variable, and double-click it. This will add the full name of this variable to your code. After the variable name add a space character and the following to complete the line:

```
>= "6" Then
```

The completed line should match this.

```
If CustomerInformation.CodeObject.CreditStatusCode >= "6" Then
```

Add a new line directly below this one, locate the *CreditStatusCode* variable in the *Variables* pane again, and double-click on it to add the full name of the variable to the script. Immediately after the variable name add a space, an equal sign and another space. On the *Field Properties* pane, select the dropdown against the *Background color* option, and choose a color. In **Figure 7-15** the color chosen was Gold. Once you have selected a color, click on the *Insert VBScript Code* button on the toolbar to add the code to set this field property.

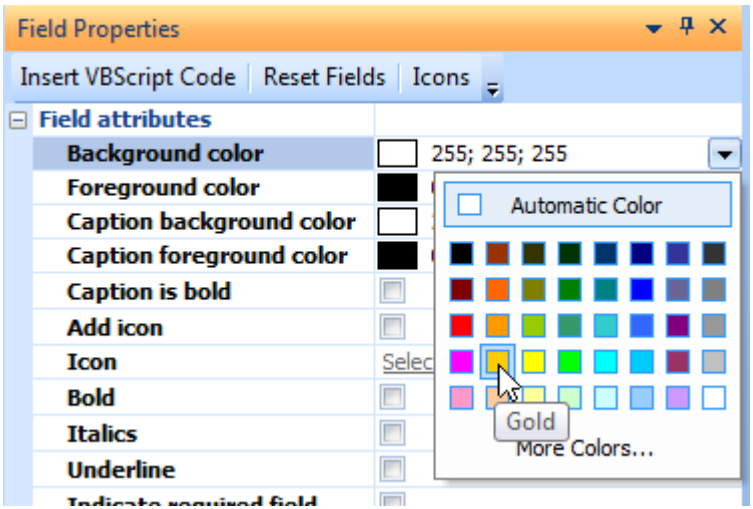


Figure 7-15: Choosing a *Background color*

Add a new line directly below the current one, and type the word *Else*. Add another new line directly below that one, and use the *Variables* pane to add the *CreditStatusCode* variable in the same way as

you did before. This is followed immediately by a space, and equal sign, and another space. The next steps are used to set the *Background color* field property to its default value.

As you have not exited the editing screen, the *Background color* field property on the *Field Properties* pane will still be set to the Gold color that you chose earlier. If you now set the *Background color* property back to the default (either manually or using the *Reset Fields* button) when you click on the *Insert VBScript Code* button the code below would be added. However, this code will not change any of the field properties against the variable as none are named within the XML, so the field will remain the current color.

```
"<Field  > </Field>"
```

The way to resolve this is to leave the *Background color* as it was previously set, use the *Insert VBScript* Code button to add the Gold *Background color* field property, and then manually change the code that was added.

Click on the *Insert VBScript Code* button on the *Field Properties* toolbar to add the Gold *Background color* field property to you code, then add a new line to the code and type in *END IF*, which will end the *IF* statement. The script should resemble that in **Figure 7-16**.

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnRefresh()
6    If CustomerInformation.CodeObject.CreditStatusCode >= "6" Then
7      CustomerInformation.CodeObject.CreditStatusCode = "<Field Background='255204000' > </Field>"
8    Else
9      CustomerInformation.CodeObject.CreditStatusCode = "<Field Background='255204000' > </Field>"
10   End If
11 End Function
```

Figure 7-16: The conditional logic before the final tweak

Locate the field property section of the line between the *ELSE* and *END IF* statements, it should resemble:

```
= "<Field Background='255204000' > </Field>"
```

Remove the numbers that specify the Red/Green/Blue color code, but leave the two single quotes. The code should now match that of **Figure 7-17**.

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnRefresh()
6    If CustomerInformation.CodeObject.CreditStatusCode >= "6" Then
7      CustomerInformation.CodeObject.CreditStatusCode = "<Field Background='255204000' > </Field>"
8    Else
9      CustomerInformation.CodeObject.CreditStatusCode = "<Field Background='' > </Field>"
10   End If
11 End Function
```

Figure 7-17: The completed script

Exit back to the *VBScript Editor*, and click on the *Save* button. Exit back to the menu and call up the *AR Customer Query* program again. Using the *Next* button, as you move through the customers, if the *Credit Status Code* is 6 or above, the background color will be set to the color that you chose (see **Figure 7-18**). If the *Credit Status Code* is below 6 the background color will be set back to the default.

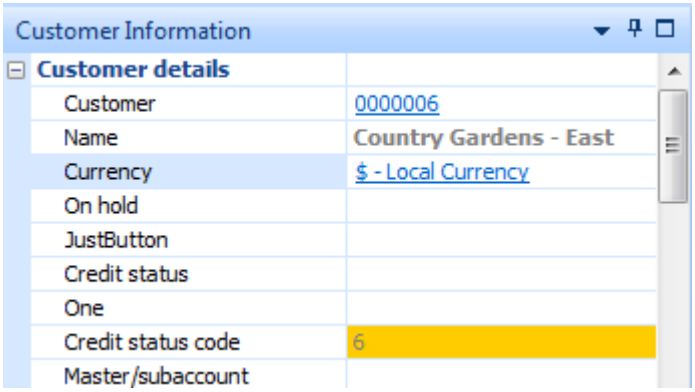


Figure 7-18: The *Credit Status Code* field's background color is set to gold

The field being checked and the field being changed do not need to be on the same form. When changing values on another form you do need to be careful about the sequence and timing of the updates, although this is more of an issue when changing the values.

In the example in **Figure 7-19**, the code is against the *Customer Information* form's *OnRefresh* function. This code performs the same check to make sure that the *Credit Status Code* is 6 or above, but instead of changing the field property of the *Credit Status Code* field on this form, it changes the field properties of the *Contact name*, *Telephone*, and *Extension* on the *Contact Details* form.

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnRefresh()
6    If CustomerInformation.CodeObject.CreditStatusCode >= "6" Then
7      ContactDetails.CodeObject.ContactName = "<Field Background='255204000' > </Field>"
8      ContactDetails.CodeObject.Telephone = "<Field Background='255204000' > </Field>"
9      ContactDetails.CodeObject.Extension = "<Field Background='255204000' > </Field>"
10   Else
11     ContactDetails.CodeObject.ContactName = "<Field Background='' > </Field>"
12     ContactDetails.CodeObject.Telephone = "<Field Background='' > </Field>"
13     ContactDetails.CodeObject.Extension = "<Field Background='' > </Field>"
14   End If
15 End Function
```

Figure 7-19: Changing the field properties of fields on another form

When the *AR Customer Query* is used for the first time, the *Customer Information* form fires both its *OnLoad* and *OnRefresh* events. This is followed by the *Contact Details* form firing its *OnLoad* and *OnRefresh* events. When subsequent customers are displayed the *Customer Information* form's *OnRefres*h event fires, followed by the *Contact Detail* form's *OnRefresh* event.

The code in **Figure 7-19** is against the *Customer Information* form's *OnRefresh* function, and is setting the field properties of fields on the *Contact Details* form. Even though the *Contact Details* form's *OnRefresh* event fires after these field properties have been set, they will not be affected by the form refreshing (unless the *ContactDetails_OnRefresh* function sets the field properties of any of these fields). However, if the code changed the values (as opposed to the field properties) the updated values would be replaced by those from the form's refresh.

Figure 7-20 shows the *Contact Details* form when the *Credit Status Code* is 6 or above on the *Customer information* form.

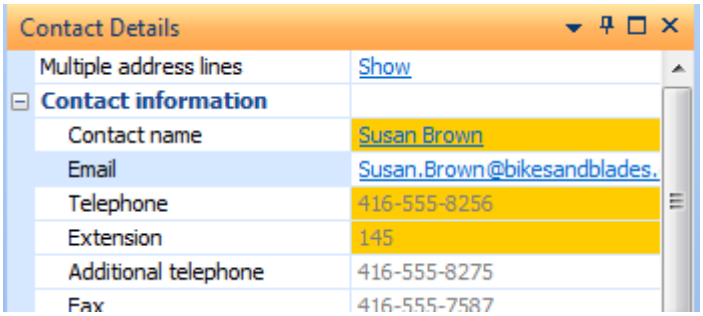


Figure 7-20: The background color is set when the *Credit Status Code* value is 6 or above

Care must be exercised if the code is written this way because someone may decide to extend its functionality at a later date to update a value, and this changed value may well be overwritten with data coming from SYSPRO when the *Contact Detail* form's *OnRefresh* event fires. Therefore, it is suggested that the logic be turned around, and the code placed against the *Contact Detail* form's *OnRefresh* event as this fires later. When this *OnRefresh* event fires the check is made of the *Credit Status Code* on the *Customer Information* form (which will already have been updated) and the field properties changed on the *Contact Details* form.

Because the full variable names that refer to fields on SYSPRO forms include the form name, exactly the same code was copied from the *CustomerInformation_OnRefresh* function and placed against the *ContactDetails_OnRefresh* function. After saving the script changes and exiting the program, the next time the program was run the relevant field's field properties were set. By comparing the code in **Figure 7-19** with that of **Figure7-21**, it can be seen that the code is the same, and only the function name is different.

..\vbscripts\ARSPENL7

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function ContactDetails_OnRefresh()
6    If CustomerInformation.CodeObject.CreditStatusCode >= "6" Then
7      ContactDetails.CodeObject.ContactName = "<Field Background='255204000' > </Field>"
8      ContactDetails.CodeObject.Telephone = "<Field Background='255204000' > </Field>"
9      ContactDetails.CodeObject.Extension = "<Field Background='255204000' > </Field>"
10   Else
11     ContactDetails.CodeObject.ContactName = "<Field Background='' > </Field>"
12     ContactDetails.CodeObject.Telephone = "<Field Background='' > </Field>"
13     ContactDetails.CodeObject.Extension = "<Field Background='' > </Field>"
14   End If
15 End Function
```

Figure 7-21: Exactly the same code, but against the *ContactDetails_OnRefresh* function

It is not intended that all the *Field Properties* that affect the display of a field's value be covered at this point, as they were covered in great detail within the *Field Properties* section of Chapter 6, and later in this chapter. However, one thing to remember is that when a field property is set that changes the appearance of the field's value, this will remain set until you change it or exit the program. It is always good practice to set the property back to its default value when it is no longer required to be set.

Changing the Appearance of a Field's Caption

The appearance of a field's caption can be changed in the same way as the field's value by applying a *Field Property* to the variable name associated with the field within the VBScripting. There are 23 field

properties that can be set, some are specific to the field's value, some are specific to a field's caption, and the rest affect the whole field. Within the list of those that affect a field's value there are differences between those that can be used with an entry form and those that can be used with a display form. For a complete list of which field properties affect field values and captions on display and entry forms, see the section *Which Field Properties Can Be Applied to Field Values/Captions by Form Type* below.

In the following example it is required that the *Route* field held on the *Sales Order Entry* program's *Order Header* form is compulsory for customer number 0000001, but not for other customers. The *Route* field is a scripted field that has been added to the *Order Header* form.

The *Indicate required field* field property is used to highlight to the operator that this is a compulsory field, and it was decided that this should only appear when it is compulsory (i.e. when working on a sales order for customer 0000001).

The customer account code is not stored on the *Order Header* form; it is stored on the *Customer Information* form. When the customer code is entered against the customer prompt on the toolbar and the operator presses *Tab* or *Enter* to accept this, the *Order Header* form fires its *OnRefresh* event (if this is the first order to be processed since the program was loaded it would fire the *Order Header* form's *OnLoad* before this), and only then does the *Customer Information* form fire its *OnRefresh* event.

At the time when the *Order Header* form's *OnRefresh* event fires the customer account code has not been updated on the *Customer Information* form (because its *OnRefresh* hasn't fired yet). Therefore, the code to perform this check cannot reside against the *Order Header* form's *OnRefresh* event. The logical place is for it to reside against the *Customer Information* form's *OnRefresh* event, as the field will have been updated at that time.

To configure this, right-click on the *Customer Information* form and select *Macro for: IMP040L4* from the context-sensitive menu. On the *VBScript Editor* screen, double-click on the *OnRefresh* form event. The *VBScript for:CustomerInformation* screen is displayed. If an *OnRefresh* function already exists against this form the cursor will be placed at the beginning of the first line of this function. If the function did not exist it will be created for you, and the cursor placed within it.

Start adding the code by typing *IF* followed by a space. Locate the *Customer* variable name within the *CustomerInformation* section of the *Variables* pane, and double-click it. The full name of this variable will be added to your code. Add to this a space, an equal sign, another space, and a double quote. Follow this with the customer code to be checked, another double quote, a space, and the word *THEN*. When you have finished the line should look similar to the one below.

```
If CustomerInformation.CodeObject.Customer = "000000000000001" then
```

Add a new line immediately after this one, and place the cursor at the beginning of it. Locate the *Route* variable within the *OrderHeader* section of the *Variables* pane and double-click on it to insert its full name. Add a space, and equal sign and another space. On the *Field Properties* pane, locate the *Indicate required field* entry and check it. Click on the *Insert VBScript Code* button on the toolbar and the XML code to apply the indicator will be inserted for you. Note that if you have already set any of the properties against the *Field Properties* pane during this run of the VBScript editor these properties may still be present and you should use the *Reset Fields* button on the *Field Properties* pane's toolbar to clear them before selecting the *Indicate required field* property.

```
OrderHeader.CodeObject.Route = "<Field Indicator='true' > </Field>"
```

Add a new line immediately below this one, and insert the word *ELSE*, and add another new line below that one. This line will contain the code specifying what should happen if the customer account code is not 0000001. The simplest way of supplying this information is to copy the line that specifies the indicator, and paste it here, then change the word *true* to be *false*. When finished the line should match the one below.

```
OrderHeader.CodeObject.Route = "<Field Indicator='false' > </Field>"
```

Finally, add another new line and close off the *IF* statement with *END IF*. When you have finished the code should match that below.

```
Function CustomerInformation_OnRefresh()
  If CustomerInformation.CodeObject.Customer = "000000000000001" then
    OrderHeader.CodeObject.Route = "<Field Indicator='true' > </Field>"
  Else
    OrderHeader.CodeObject.Route = "<Field Indicator='false' > </Field>"
  End If
End Function
```

Before exiting this screen, click on the *Syntax Check* button to check for spelling mistakes and typos. If there are none, exit the VBScript editor back to the *VBScript Editor* screen, select *Save*, and you will be returned to the *Sales Order Entry* screen.

Supplying the customer code of 0000001 should cause the indicator to appear against the *Route* field (see **Figure 7-22**). Supplying any other customer code should cause it to not be displayed.

Note: in the above example the presentation length of the customer field has been set to 7, but as this is a numeric value the check must be made against the full length of this field, so must include the preceding zeros.

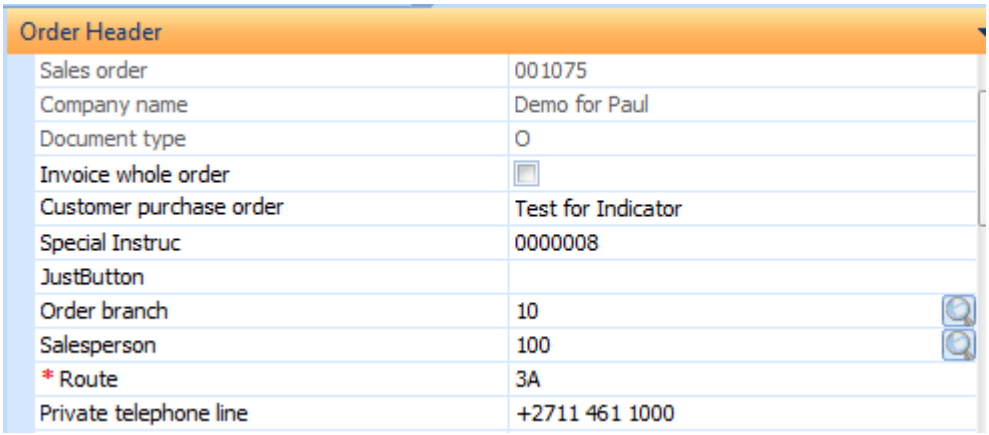


Figure 7-22: The indicator property (red asterisk) set against the *Route* field

The indicator is just a notification to the operator that the field is required, it doesn't stop the operator from ignoring it. With six more lines of code you can perform a check to see if this field has been populated, and prevent the operator from saving the order until it is.

This code must be supplied against the *Order Header* form's *OnSubmit* function. Note that unlike the code above (where it had to be added to the *Customer Information* form because of the timing of the form updates) this code is added to the *Order Header* form because the *Customer* field must have already been updated to be able to save the order.

Highlight any field on the *Order Header* form. This time, instead of right-clicking on the form to load the editor, press the *ALT+F8* shortcut keys, and the *VBScript Editor* will be loaded. Within the *Available Events* listview, locate the *OnSubmit* event name and double-click on it. If it does not already exist the *OrderHeader_OnSubmit* function will be created for you, and the cursor placed within it. Start adding the code by typing *IF* to start an *IF* statement, and follow this with a space. Locate the *Customer* variable within the *CustomerInformation* section of the *Variables* pane and double-click on it. The full name of the *Customer* variable will be added. Add to this a space, an equal sign, another space, and a double quote. Follow this with you customer account code to be checked, a double quote, a space, and the word *THEN*. You should have a line of code that is similar to this.

```
If CustomerInformation.CodeObject.Customer = "000000000000001" then
```

Add a new line after this and start it with the word *IF* followed by a space. Locate the *Route* variable name under the *OrderHeader* section of the *Variables* pane, and double-click it to add the full variable name to your code. After this add a space, an equal sign, another space, two double quotes, another space, and the word *THEN*. You line should match the one below. It specifies that what happens after this line should occur if the *Route* field is empty.

```
If OrderHeader.CodeObject.Route = "" Then
```

Start a new line, locate the *Route* variable name under the *OrderHeader* section of the *Variables* pane, and double-click it to add the full variable name to your code. Follow this with a space, an equal sign, and another space. On the *Field Properties* pane, click on the *Reset Fields* button to clear any existing properties. Locate the *Set focus* to the field property within the *Field behavior* section and check it. Click on the *Insert VBScript Code* button to add the XML to your code. This field property puts the cursor on the specified field, which makes it easier for the operator to see which field is missing, and supply the required information.

```
OrderHeader.CodeObject.Route = "<Field Focus='true' > </Field>"
```

Start a new line below this one and type in the following, which stops the *OrderHeader_OnSubmit* function from completing.

```
OrderHeader_OnSubmit = false
```

End the second *IF* statement by adding a line containing *END IF*, and follow this with another line containing *END IF* to finish of the first *IF* statement. When you have finished, your code should match this.

```
Function OrderHeader_OnSubmit()
  If CustomerInformation.CodeObject.Customer = "000000000000001" then
    If OrderHeader.CodeObject.Route = "" Then
      OrderHeader.CodeObject.Route = "<Field Focus='true' > </Field>"
      OrderHeader_OnSubmit = false
    End If
  End If
End Function
```

Click on the *Syntax Check* button to check for typos, and then exit back to the *VBScript Editor* screen. Click on the *Save* button to return to the *Sales Order Entry* screen.

When the operator attempts to save an order for customer 0000001, if the *Route* field contains a value, normal processing will occur. If the *Route* field does not contain a value, the processing will stop and the cursor will be placed on the *Route* field.

Changing a Field's Value

The field's value is changed using the *Value* field property (see **Figure 7-23**) which is applied to the field's variable name.

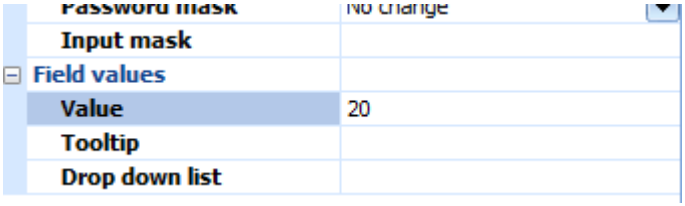


Figure 7-23: Setting a field's *Value* to 20

Below is a sample of code where the *Branch* field on the *Order Header* form of the *Sales Order Entry* program is set to the value *20*.

```
OrderHeader.CodeObject.OrderBranch = "<Field Value='20' > </Field>"
```

An example of where this can be used is where you need to set the *Branch* and *Salesperson* depending on the company that is being processed, and which operator is processing the order. In the sample code below, a check is made to see if this is company C. If it is, a check is made to see if this is operator JOE. If both criteria are met the branch code is set to 20 and the salesperson code set to 201.

```
Function OrderHeader_OnRefresh()
  If SystemVariables.CodeObject.Company = "C" then
    If SystemVariables.CodeObject.Operator = "JOE" then
      OrderHeader.CodeObject.OrderBranch = "<Field Value='20' > </Field>"
      OrderHeader.CodeObject.Salesperson = "<Field Value='201' > </Field>"
    End If
  End If
End Function
```

There are a couple of items to note regarding changing a value against a field. In most cases validation is not performed immediately. For example, populating the branch code with a 12 digit number (the branch code can only be 10 characters long) will cause the number to be displayed against the field. The validation will occur the next time that you attempt save the header record, save the sales order (or if no detail lines have been added yet) when you attempt to add the first detail line.

If you attempt to programmatically write a letter into a field that can only accept numeric values, the field will be set to zeros. If you had already manually entered a numeric value, and you attempt to write an alphanumeric character the existing number will be replaced with zeros. An example is the *Discount % 1* field on the *Order Header* form in *Sales Order Entry*. If a value of 5.00 had already been entered, and you programmatically write an X to the field, the value against this field will be reset to its default value of 0.00.

Another point to note is that SYSPRO may refresh the value of a field that you have already updated, in which case your change will get overwritten by the information being returned from the SYSPRO table. An example of this is when adding a *Stocked Line* in *Sales Order Entry*. The operator enters the

Warehouse and *Stock Code*. SYSPRO goes back to the server to retrieve information, and populates the *Description* field with the stock description from the stock master table. You could have code against the *Order quantity* field's *OnLostFocus* event that changes the *Description* depending on the *Order quantity* that works. If the operator realises that they have supplied the incorrect warehouse code and changes this, when they tab off the *Stock code* field SYSPRO will repopulate the *Description* field with the value from the stock master table, and your change will be overwritten.

To programmatically update a date field you must supply the date in the format CCYY-MM-DD or CCYYMMDD (for example, the 1st September 2013 would appear as 2013-09-01 or 20130901). If you programmatically supply alpha characters to a date it will clear the existing date (see **Figure 7-24**).

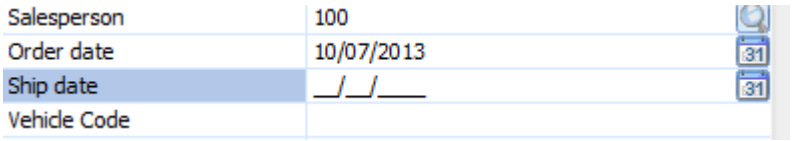


Figure 7-24: The result of programmatically adding an alphanumeric character to a date field

Types of Fields and those that may have Their Values Changed

The type of field that can have its value changed programmatically depends on the type of form on which it resides, either a display or an entry form.

Display Form

- You cannot change the value of a normal field that appears on the form (numeric, alphanumeric, date)
- You cannot change the value of a *Standard* field (*Company Name*, *Company Date*, *Current Printer*, etc.)
- You cannot change the value of a *Custom Form* field
- You can change the value of a *Scripted* field.

Entry Form

- You can change the value of a normal field that appears on the form, even if it has been set to read-only using a field property
- You cannot change the value of a *Standard* field
- You can change the value of a *Custom Form* field
- You can change the value of a *Scripted* field.

Which Field Properties Can Be Applied to Field Values/Captions by Form Type

There are 23 different *Field Properties* that can be set programmatically. Some of these change the properties of the value portion of a field, some change the properties of the caption portion of a field, and some can change the properties of both. In addition, which field properties work with captions and values depends on which form type they reside, either a display or an entry form. Below is a list of which field properties can be used against a field caption or value, by form type.

Field Properties Affecting Captions on a Display Form

- Caption background color
- Caption foreground color
- Caption is bold
- Indicate required field
- Field height – affects whole field
- XAML code
- Tooltip – appears when moving mouse pointer over both value and caption
- Menu text – the menu appears against the whole field
- Visible – shows/hides the whole field

Field Captions on an Entry Form

- Caption background color
- Caption foreground color
- Caption is bold
- Indicate required field
- Field height - affects whole field
- XAML code
- Read only – changes look of whole field
- Tooltip – appears against caption and value
- Menu text
- Visible – shows/hides whole field

Field Properties Affecting Values on a Display Form

- Background color
- Foreground color
- Icon
- Bold
- Italics
- Underline
- Field height – affects whole field
- Password Mask
- Input mask – used only to format the display of the data. It would be better to use the Field Properties for All Forms option to achieve this.

- Drop down list
- Visible – shows/hides the whole field
- Menu text – the menu appears against the whole field
- Tooltip – appears when moving mouse pointer over both value and caption

Field Properties Affecting Values on an Entry Form

- Background color
- Foreground color
- Icon
- Bold
- Italics
- Underline
- Field height – affects whole field
- Change case
- Read only - changes look of whole field
- Visible – shows/hides whole field
- Set focus to the field
- Button wording
- Menu text – the menu appears against the whole field
- Password mask
- Input mask
- Value
- Tooltip – appears when moving mouse pointer over both value and caption
- Drop down list

Form Actions

A *Form Action* is an extremely powerful user-definable hyperlink that sits on a section at the bottom of either a display or an entry form. The hyperlink can be configured to call a SYSPRO program and optionally pass it parameters, or to fire a VBScript event. You can have multiple form actions defined per form, and they can be any combination of these types. Chapter 4 of the first *Power Tailoring* book described how to configure a form action to call a SYSPRO program and pass it parameters, as this requires no development skills. This *Form Actions* section will cover creating a VBScript form action, as these fire macro events that call functions within a VBScript, and do require development skills.

A form action is added using the *Insert Form Action* option from the context-sensitive menu that appears when you right-click on a form. This option can be used to add a form action that will use VBScript, a form action that will call a SYSPRO program (and optionally pass it a parameter) or add a form action template, which is a predefined sample.

Figure 7-25 shows how to add a form action using the *Insert Form Action* option of the context-sensitive menu. As this form action is going to fire an event (as opposed to call a SYSPRO program) only the description is added, in this case *Go To Master Account*. When you press the *Enter* key the hyperlink is added to the form and the list of available events for this form is shown in the *VBScript Editor* screen.

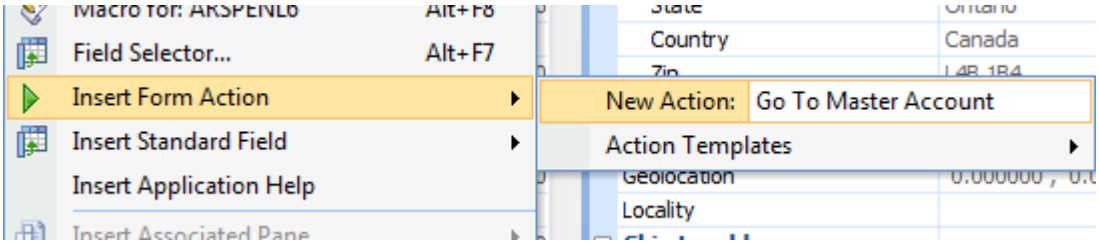


Figure 7-25: Adding a *Form Action* that will call a VBScript function

Figure 7-26 shows the *Available Events* section of the *VBScript Editor* screen where the *OnGoToMasterAccount* event has been automatically added under a section called *ACTION events*. Behind this you can see a portion of the *Customer Information* form, and right at the bottom you will see that the *Go To Master Account* hyperlink has already been created.

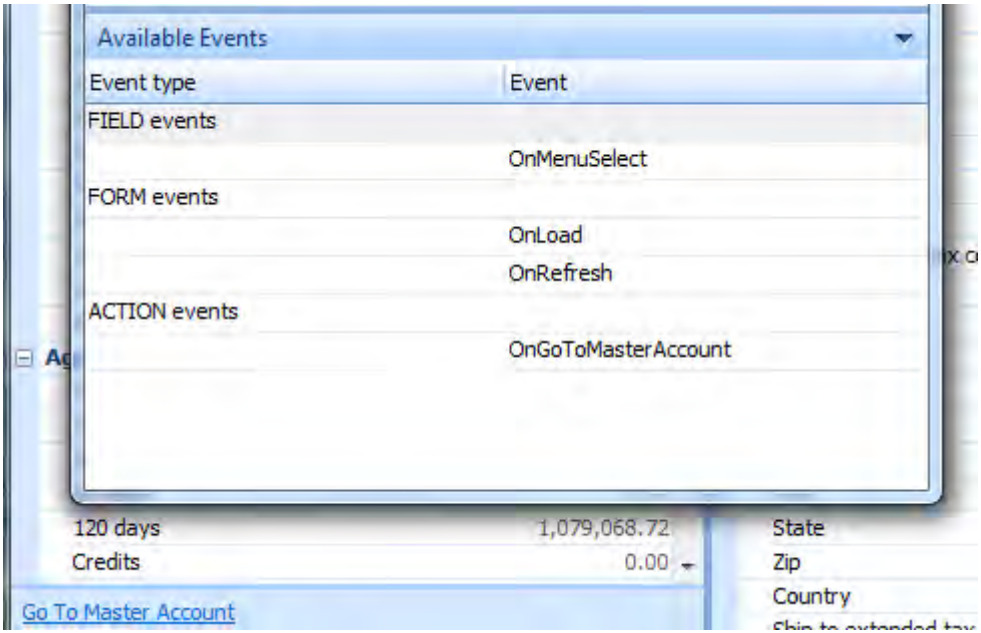


Figure 7-26: The hyperlink added to the form, and the *OnGoToMasterAccount* event

The function is added to the VBScript by either highlighting this event and clicking on the *Insert VBScript* button, or double-clicking on this event name. In the same way as any other function, the function name will be made up of the form name and the event name. As the *GoToMasterAccount* form action was added to the *Customer Query* program's *Customer Information* form, the function name is *Customer Information _OnGoToMasterAccount*. **Figure 5-27** shows the *Customer Information _OnGoToMasterAccount* function after it has been added.

..\vbscripts\ARSPENL6

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomerInformation_OnGoToMasterAccount()
6
7  End Function
```

Figure 5-27: The *CustomerInformation_OnGoToMasterAccount* function

The next part of this section will build up the code that will be run when the *Go To Master Account* hyperlink is clicked on the *Customer Information* form. The code needs to check to see if the currently selected customer account code is a sub-account of a master-account. If it is a sub-account it must call the customer query business object (***ARSQRY***) for this account, load the results into the Document Object Model, and extract the corresponding master account from the XML. Once this information is available it must use the *Toolbar Button* functionality to pass this account number to the *Customer* button on the toolbar, and execute it. The *Toolbar Button* functionality was covered in detail in Chapter 6.

If a customer account is a master-account the *Master/sub-account* field will contain the text *This is a master account*. If it is a sub-account it will contain the text *This is a sub account* (see **Figure 7-28**). If it is neither a master nor sub-account the field will be empty.

Credit status code	0
Master/subaccount	This is a sub account
Credit limit	0
AR invoice terms	1 - 30 Days - 2.5%

Figure 7-28: The text against the *Master/subaccount* field for a sub-account

The first line of code must perform a check to see if the currently selected account is a sub-account. Make sure that the cursor is on the blank line inside the function. Add an *IF* statement and follow this

with a space. Locate the *MastersubAccount* variable name within the *CustomerInformation* section of the *Variables* pane, and double-click it. The full name of this variable will be added to your code. Follow this with a space, an equal sign, another space, and a double-quote. Add the text *This is a sub account* and follow this with another double-quote, a space, and the word *Then*. When you have finished the line should match that below.

```
If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
```

Add a new line directly below this one and place the cursor on it. Click on the *Call Business Object* button on the toolbar which will start the *Call a Query Business Object* wizard. At the *Query Business Object* button type in ***ARSQRY*** and press the *Enter* key. The *Parameters* tab will be populated with the sample XML for this business object. Replace this sample XML with what appears below.

```
<Query>
  <Key>
    <Customer>XXXXXX</Customer>
  </Key>
  <Option>
    <IncludeFutures>N</IncludeFutures>
    <IncludeTransactions>N</IncludeTransactions>
  </Option>
</Query>
```

Once the XML has been replaced, click on the *Insert VBScript* button on the toolbar (see **Figure 7-29**), and the code to build this XML and call the business object will be added for you.

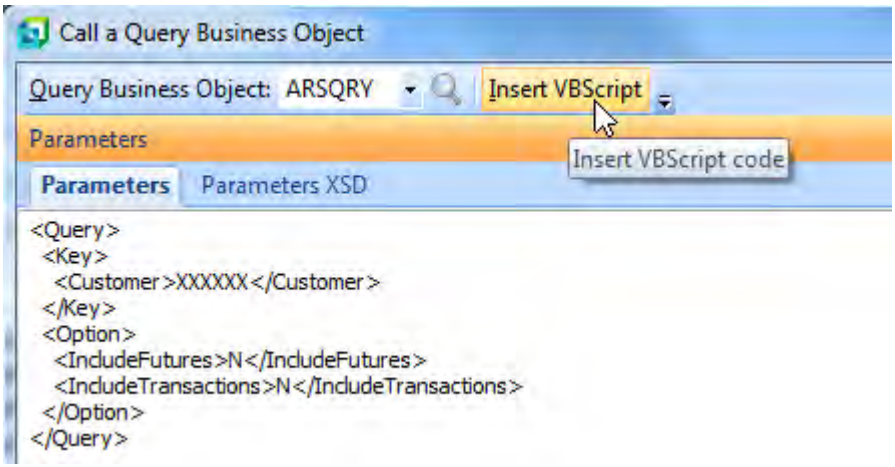


Figure 7-29: Completing the *Call a Query Business Object* wizard

Your code should now match the code below (note that the indentations have been added to make it easier to follow, and that these do not affect the functionality).

```
Function CustomerInformation_OnGoToMasterAccount()
  If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
    dim XMLOut, XMLParam

    XMLParam = XMLParam & "<Query>"
    XMLParam = XMLParam & "  <Key>"
    XMLParam = XMLParam & "    <Customer>XXXXXX</Customer>"
    XMLParam = XMLParam & "  </Key>"
    XMLParam = XMLParam & "  <Option>"
    XMLParam = XMLParam & "    <IncludeFutures>N</IncludeFutures>"
    XMLParam = XMLParam & "    <IncludeTransactions>N</IncludeTransactions>"
    XMLParam = XMLParam & "  </Option>"
    XMLParam = XMLParam & "</Query>"
    on error resume next
    XMLOut = CallBO("ARSQRY",XMLParam,"auto")
    if err then
      msgbox err.Description, vBCritical, "Calling Business Object"
      exit function
    end if
    ' Switch on error handling
    on error goto 0

End Function
```

The *Customer* element contains the value *XXXXXX* which must be replaced with the variable that contains the currently selected customer account code. The two elements within the *Options* section have been included because these default to *Y*, and could return a significant amount of extra XML that is not required, and it is more efficient to prevent this XML from being returned.

Two variables are created within the *DIM* statement. These are used to build the XML to be sent to the business object, and to store the XML that is returned by the business object. To complete the code you need to add two more variables. Either these can be added with their own *DIM* statement, or the two variable names can be added to the existing *DIM* statement. The two variables are called *XMLDoc* and *MastAcc*. In this example they are added to the existing *DIM* statement. Note that there are commas between each of the variable names. The *DIM* statement must be changed from this:

```
dim XMLOut, XMLParam
```

To this :

```
dim XMLOut, XMLParam, XMLDoc, MastAcc
```

The next step is to replace the six *X* characters with the variable containing the currently selected customer account code. Locate the line containing the six *X* characters, which should finish with the following:

```
"     <Customer>XXXXXX</Customer>"
```

Note that the space characters between the double quote and the less-than sign are only present because the XML was indented to make it easier to read.

Immediately after the greater than sign that closes off the opening *Customer* element, add a double quote, a space, an ampersand, and another space. Then locate the *Customer* variable within the *CustomerInformation* section of the *Variables* pane. Double-click this variable and its full name will be added to your code. Follow this with a space, an ampersand, another space, and a double quote. Then remove the six *X* characters. The end of this line should now match the following:

```
"      <Customer>" & CustomerInformation.CodeObject.Customer & "</Customer>"
```

You have completed the required changes to the existing code, but there are still a few lines that must be added to the end. Add a new line between the last line of code (containing *on error goto 0*) and *End Function*. Add the following three lines of code. These instantiate the Document Object Model, and load the output from the business object into it.

```
Set XMLDoc = createobject("MSXML2.DOMDocument")
XMLDoc.async = false
XMLDoc.LoadXML(XMLOut)
```

The next line of code extracts the contents of the *MasterAccount* element and places it in the *MastAcc* variable.

```
MastAcc = XMLDoc.SelectSingleNode("//ARStatement/Header/MasterAccount").Text
```

Add a new line directly under this one. Locate the *ToolbarButton* variable within the *SystemVariable (actions)* section of the *Variables* pane, and double-click it (see **Figure 7-30**).

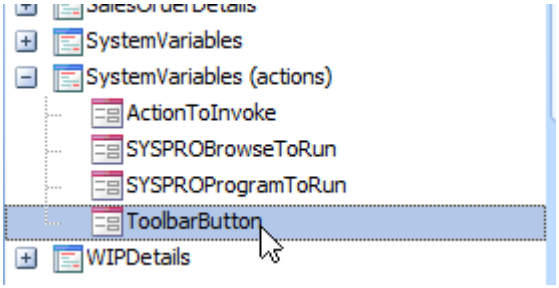


Figure 7-30: The *ToolbarButton* variable

Double-clicking the *ToolbarButton* variable name calls up the *Modify Toolbar Buttons* screen. The VBScript editing screen will disappear, leaving the *AR Customer Query* screen behind the *Modify Toolbar Buttons* screen. Click on the *Customer* button on the *AR Customer Query* toolbar (that appears behind the *Modify Toolbar Button* screen). The information about this toolbar button will populate the first three fields on the *Modify Toolbar Buttons* screen (see **Figure 7-31**).

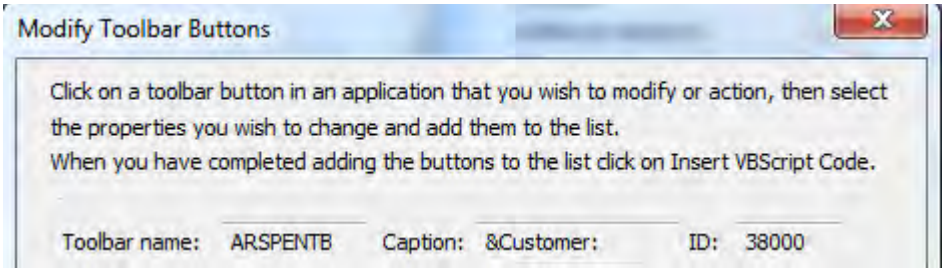


Figure 7-31: The *Modify Toolbar Buttons* screen populated with the *Customer* button information

On the *Modify Toolbar Buttons* screen, change the *Action* dropdown to say *Execute*, add a *Button value* of *XXXXXX*, and click on the *Add to List* button. The listview at the bottom of the *Modify Toolbar Button* screen will be populated with these settings (see **Figure 7-32**).

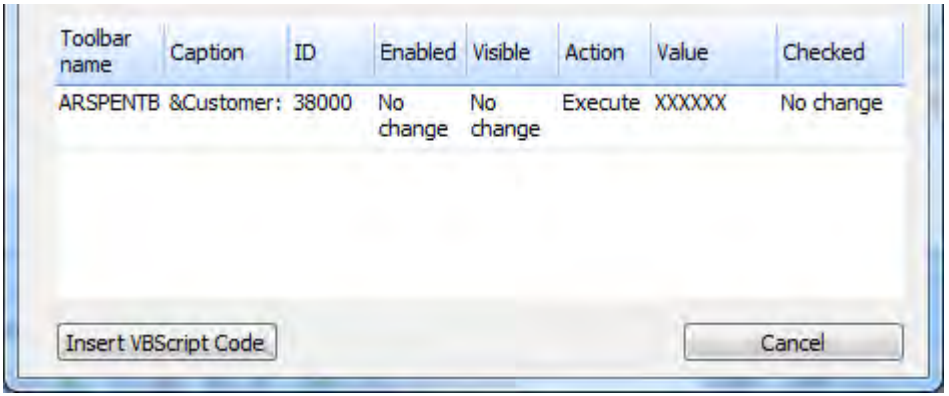


Figure 7-32: The populated listview at the bottom of the *Modify Toolbar Button* screen

Click on the *Insert VBScript Code* button to insert the code to populate the button and execute it. The inserted code will contain three lines. Locate the second line which contains the following segment that needs to be modified.

```
Value='XXXXXX'
```

The six *X* characters must be replaced with a double quote, a space an ampersand, another space, the variable name *MastAcc*, another space, another ampersand, another space, and a double quote. When completed the change should look like this:

```
Value='" & MastAcc & "'
```

All that is left to do is to close off the *IF* statement by adding *END IF* between the last line of code and the *End Function* statement. The completed code appears is **Figure 7-33**.

Exit the program where you were editing the script back to the *VBScript Editor* screen, click *Save* and return to the *AR Customer Query* program.

If the currently selected customer in the query is a sub-account, when you click on the *Go To Master Account* form action, the displayed customer will change to the master account to which the sub-account belongs. If the currently selected customer is not a sub-account nothing will happen when you click on the form action.

```
5  Function CustomerInformation_OnGoToMasterAccount()
6    If CustomerInformation.CodeObject.MastersubAccount = "This is a sub account" Then
7      dim XMLOut, XMLParam, XMLDoc, MastAcc
8
9      XMLParam = XMLParam & " <Query>"
10     XMLParam = XMLParam & "   <Key>"
11     XMLParam = XMLParam & "     <Customer>" & CustomerInformation.CodeObject.Customer & "</Customer>"
12     XMLParam = XMLParam & "   </Key>"
13     XMLParam = XMLParam & "   <Option>"
14     XMLParam = XMLParam & "     <IncludeFutures>N</IncludeFutures>"
15     XMLParam = XMLParam & "     <IncludeTransactions>N</IncludeTransactions>"
16     XMLParam = XMLParam & "   </Option>"
17     XMLParam = XMLParam & " </Query>"
18     on error resume next
19     XMLOut = CallBO("ARSQRY",XMLParam,"auto")
20     if err then
21       msgbox err.Description, vBCritical, "Calling Business Object"
22       exit function
23     end if
24     ' Switch on error handling
25     on error goto 0
26
27     Set XMLDoc = createobject("MSXML2.DOMDocument")
28     XMLDoc.async = false
29     XMLDoc.LoadXML(XMLOut)
30
31     MastAcc = XMLDoc.SelectSingleNode("//ARStatement/Header/MasterAccount").Text
32
33     SystemVariables.CodeObject.ToolbarButton = "<Toolbars>" & _
34     "<Toolbar Name='ARSPENTB' Id='38000' Action='Execute' Value='" & MastAcc & "' Caption='Customer:'/>" & _
35     "</Toolbars>"
36   End If
37 End Function
```

Figure 7-33: The completed *Form Action* code

VBScript Actions

A VBScript *Action* enables you to programmatically invoke a section of the code of the SYSPRO program from your VBScript. The *Actions* button is only enabled where a form implements *Actions*, and the available actions are specific to the form being edited. **Figure 7-34** shows where the *Actions* button resides on the toolbar within the screen where you edit your scripts.

Figure 7-34: The *Actions* button on the toolbar

If the *Actions* button is enabled it means that there are some *Actions* available for this form. Not all forms contain *Actions*. A complete list of the programs containing *Actions* (along with which panes within that program contain which *Actions*) can be found in a text file called `IMPVBS.IMP` which can be found in the `Programs` folder on the SYSPRO application server.

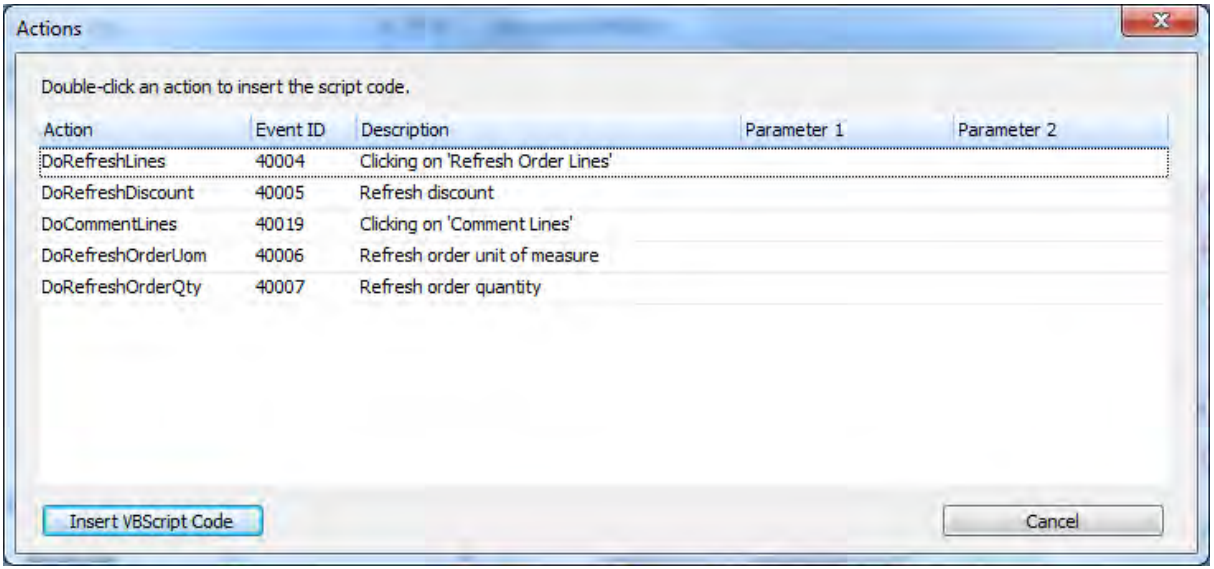


Figure 7-35: The list of *Actions* for the *Stocked Line* form of the *Sales Order Entry* program

The *Stocked Lines* pane within the *Sales Order Entry* program contains five *Actions*. These are *DoRefreshLines*, *DoRefreshDiscount*, *DoCommentLines*, *DoRefreshOrderUom*, and *DoRefreshOrderQty*. These can be seen in **Figure 7-35**.

To add this functionality, highlight the relevant option and click on the *Insert VBScript Code* button. The code to invoke the *Action* will be added to your script. The *Actions* screen will remain open. Click on the *Cancel* button to exit this *Actions* screen.

The following is the code that was added when the *DoRefreshLines Action* was selected. This causes the *Entered Order Lines* listview to refresh so that all detail lines are included. In this case it was added to the *OnAfterSubmit* event of the *Stocked Line* form.

```
Function StockedLine_OnAfterSubmit()
  SystemVariables.CodeObject.ActionToInvoke = "DoRefreshLines,40004"
End Function
```

A scenario where this could be used is if you automatically added extra detail lines to the currently open sales order when stock codes belonging to a specific product class are ordered. For example, you have a special on girl's and boy's bicycles that includes a helmet. The detail line for the helmet is automatically added to the order using the ***SORTOI*** business object as the bicycle is added to the order. Unless the *Entered Order Lines* listview is refreshed it would not include the helmet, as it wasn't added by the operator. When the *DoRefreshLines Action* is invoked the listview is refreshed with all detail lines for this order. When the *Entered Order Lines* listview is refreshed in this way its *OnPopulate* event is also invoked, so any code against this function will also be run.

Only one *Action* can be invoked within a function. If there are multiple actions in a function, only the last one will be run. In the sample of code that appears below there are two *ActionToInvoke* statements. The first one is used to refresh the contents of the *Entered Order Lines* listview, and the second to call up the *Comment Line* screen where you can enter free format comment lines (as if you had clicked on the *Comment* button on the *Order line* toolbar). Because only one action can be run per function, and any later one will overwrite an earlier one, the action to call up the free format comment lines screen will be invoked. If these two lines appeared the other way around, the *Entered Order Lines* listview would be refreshed.

```
Function JustButton_OnButtonClick()
  SystemVariables.CodeObject.ActionToInvoke = "DoRefreshLines,40004"
  SystemVariables.CodeObject.ActionToInvoke = "DoCommentLines,40019"
End Function
```

Note: *Actions* have been mostly superseded from SYSPRO 7 onwards, as you can use the *ToolbarButton* system variable action to develop code to manipulate any buttons in any toolbar.

Input Mask Sample

The *Input Mask* is set against a field and has two uses: it becomes a template to force the operator to input values in a consistent format, and it can be used to display values in a specific format.

For example, you may want operators to always enter a telephone number with the area code before the number. **Figure 7-36** shows an example where the input mask has been defined so that the telephone number must be entered this way. Because the template is shown, the operator knows what must be done, and they are also forced to enter it this way.

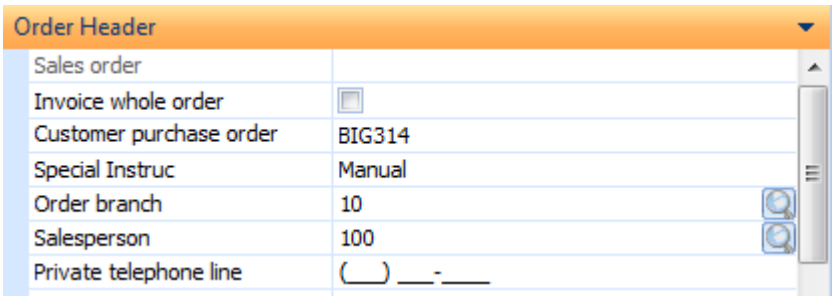


Figure 7-36: Using the *Input Mask* field property to define a template

The template is built up using a combination of characters and underscores. An underscore represents a position and the character immediately after it represents a range of numbers/letters, and their case. Any characters in the template that do not immediately follow an underscore will appear as themselves.

The following is a list of the characters and their special meanings.

0 – Numeric (0-9)
9 – Numeric (0-9) or space
– Numeric (0-9) or space or + or -
L – Alpha (a-Z)
? – Alpha (a-Z) or space
A – Alphanumeric (0-9 and a-Z)
a – Alphanumeric (0-9 and a-Z) or space
& – All printable characters
H – Hex character (0-9 and A-F)
X – Hex character (0-9 and A-F) or space
> – Forces characters to uppercase (A-Z)
< – Forces characters to lowercase (a-z)

To force the telephone number to be entered correctly you would specify the entry of only numbers without spaces. To do this you use underscores followed by zeros, which denote that 0-9 are the only characters allowed.

The input mask to produce the telephone example above is (_0_0_0) _0_0_0-_0_0_0_0. As the parentheses are not immediately preceded by underscores, they are just for display purposes, and do not get written to the database. The same is true of the space character immediately after the second parenthesis. The string above is added to the *Input Mask* of the *Field Properties* screen (see **Figure 7-37**).

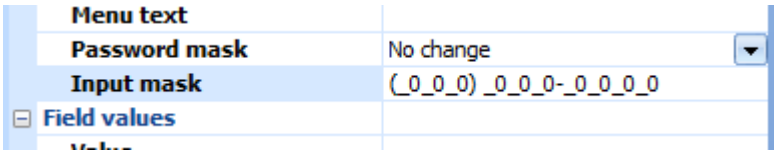


Figure 7-37: Adding the string to the *Input Mask* section of the *Field Properties*

When the *Insert VBScript Code* button is clicked, the following code is added to the script where the cursor was last positioned in the script (which should be immediately after the relevant variable name, a space, an equal sign, and another space).

```
"<Field InputMask='(_0_0_0) _0_0_0-_0_0_0_0' > </Field>"
```

If this was added to the form's *OnLoad* function, the next time that the program is run the template will appear (see **Figure 7-38**). If this were applied to a normal *Order Header* field (as opposed to a *Scripted* field as per this example) only the value of 7144371000 would be written to the column in the database. If the sales order were to be displayed without this *Input Mask* being present, the value would be displayed without any formatting.

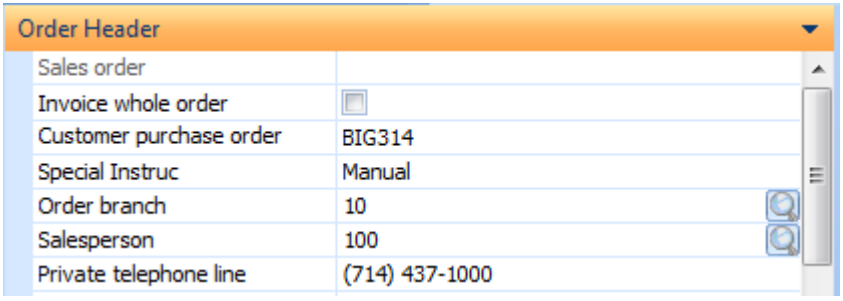


Figure 7-38: The operator adding a telephone number using the *Input Mask*

Drop Down List Sample

The *Drop down list* field property is used to apply a dropdown list to a field, and only items from this list can be chosen. The operator cannot supply a value that is not in this list. The available options are added to the *Drop down list* field property delimited with semi-colons. **Figure 7-39** shows the *Drop down list* field property populated with three values, *1 Year*, *2 Years*, and *3 Years*. It may seem obvious, but I am going to state this, the sequence that the items appear against the *Drop down list* field is the sequence in which they will appear in the dropdown list.

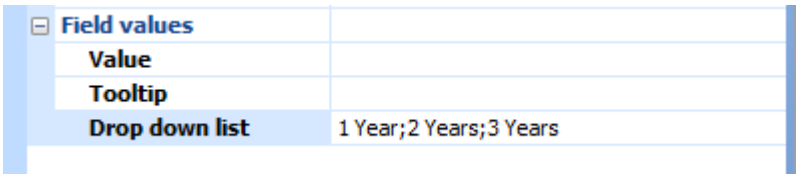


Figure 7-39: The *Drop down list* field property containing three values

In this example the dropdown list is applied to a field within the *Stock Code Maintenance* program. The field in question is *User defined 1*, and its name is being displayed as *Warranty period* using the *Field Properties for All Forms* functionality that was covered in Chapter 4 of the first *Power Tailoring* book.

Figure 7-40 shows the *Field Properties for All Forms* screen where the caption of the *User defined 1* field is changed to *Warranty period*. After this change has been made, the *User defined 1* caption is replaced with the *Warranty period* caption (see **Figure 7-41**).

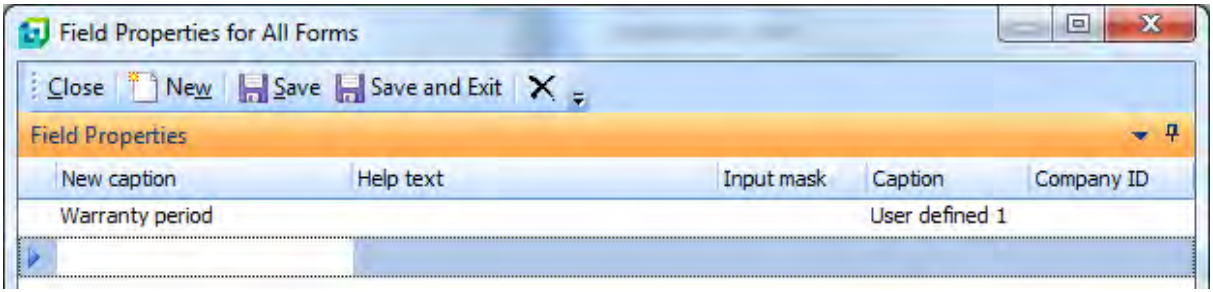


Figure 7-40: Changing the field's caption using the *Field Properties for All Forms* program

The *User defined 1*/*Warranty period* field resides on a form called *Other* in the *Stock Code Maintenance* program. To add the dropdown list to this field you need to add VBScript code to the *Other* form's *OnLoad* function.

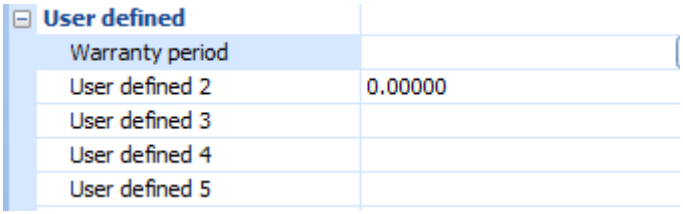


Figure 7-41: The updated caption displaying *Warranty period*

Highlight a field on the *Other* form and use the *Alt+F8* shortcut keys to load the *VBScript Editor* screen. Double-click the *OnLoad* event name, and the function will be created for you if it doesn't already exist. The cursor will be placed within this function. Locate the *User defined 1* variable name within the *Other* section of the *Variables* pane (note that this remains as *User defined 1* as the *Field Properties for All Forms* functionality does not affect the variable names). Double-click this variable name and the full name of this variable will be added to your code. Add a space, an equal sign and another space.

Against the *Drop down list* caption in the *Field Properties* pane, add the options *1 Year*, *2 Years*, and *3 Years* with semi-colons between them (as appears in **Figure 7-39**). Next, click on the *Insert VBScript Code* button on the toolbar of the *Field Properties* pane, and the XML to build your dropdown list will be added to your code. The line should now match the following.

```
Other.CodeObject.UserDefined1 = "<Field List='1 Year;2 Years;3 Years' > </Field>"
```

Click on the *Syntax Check* button on the toolbar to highlight any issues. If none are found, close the screen where you edit the VBScript and you will be taken back to the *VBScript Editor* screen. Click on the *Save* button and you are taken back to the *Stock Code Maintenance* program. As you have added this code to the *Other* form's *OnLoad* function you will need to exit this program and call it up again. After doing so, when you enter a stock code to be added or maintained, the *Warranty period* field will have a dropdown list against it containing the three options that the operator can select (see **Figure 7-42**).

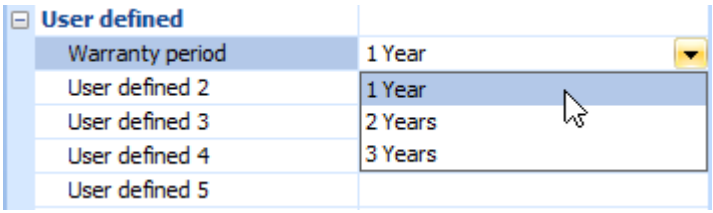


Figure 7-42: The *Drop down list* field property applied to the *Warranty period* field

The items in the dropdown list can also have icons placed against them. If this field were against a sales order detail line, instead of against stock code maintenance, you might want to encourage the salesperson to push the sale of an optional longer warranty. You could reinforce this by placing an icon against each item in the list (such as a traffic light).

When using the *Drop down list* option of the *Field Properties* pane to add the list of options, you can click on the *Icons* button on the toolbar at the top of the pane (see **Figure 7-43**) which will display the *Change Icon* screen.

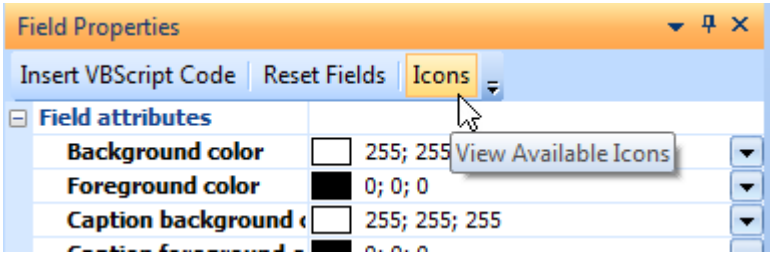


Figure 7-43: Calling up the *Change Icon* screen to view the available icons

The *Change Icon* screen contains a list of the available icons. Each icon has a number associated with it. **Figure 7-44** shows that the icon titled *Traffic[2]* has been highlighted, and its *Icon number* of *55* is displayed at the bottom of the screen.

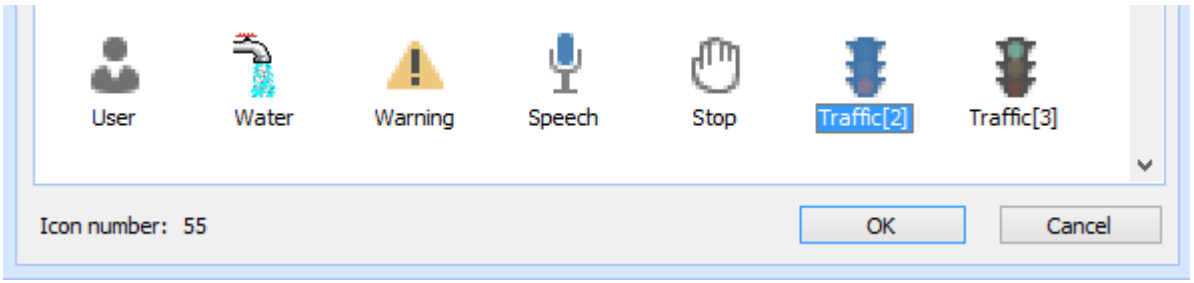


Figure 7-44: Displaying the *Icon number* for the highlighted icon

Make a note of this icon number, as well as the others that you want to use. In this case the red traffic light (055), amber traffic light (044), and green traffic light (056) will be used. The three digit icon numbers are added (within square brackets) before each of the options against the *Drop down list* prompt. **Figure 7-45** shows where the red light icon will appear against the *1 Year* option, the amber light icon will appear against the *2 Years* option, and the green traffic light will appear against the *3 Years* option.

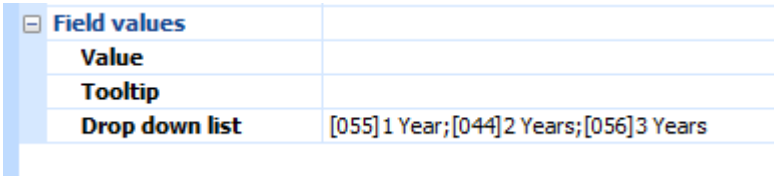


Figure 7-45: Add the icons to the list items

Once this has been inserted into your VBScript, saved, and the program called up again, the options will have icons against them (see **Figure 7-46**).

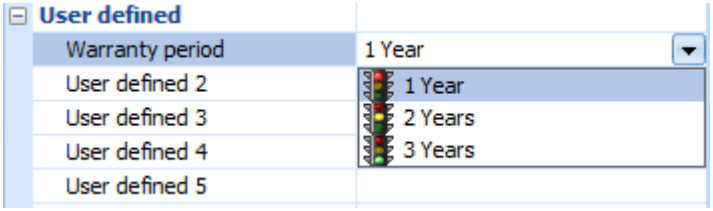


Figure 7-46: The icons appearing in the dropdown list

Going back to the original example where the dropdown list appears against the *Stock Code Maintenance* program. If this appeared against the *Other* form's *OnRefresh* function instead of the *OnLoad* function, the options in the dropdown list could be dependent on another value of the stock code, such as the *Product Class* field. For example, if the product class is *RB* the dropdown list could contain *1 Year*, *2 Years*, and *3 Years*, whereas if the product class is *SPEC* the list could contain *3 Months*, *6 Months*, *9 Months*, and *1 Year*.

Note: you would also need to add the same code against the *Product Class* field's *OnAfterChange* function to make sure that the dropdown list changed if the operator happened to change the contents of the product class field after the stock code had been added.

Visible Sample

The *Visible* field property is used to make a field visible/invisible. If the field must always be invisible it would be better for the Administrator to remove the field from the form by dragging it off. Setting the field to invisible is usually done when the field should/shouldn't be present depending on some other value or setting.

An example of where you might use this is in *Sales Order Entry*. When your largest customer places an order it will either be a rush order (which they will collect) or a normal order (which they will have delivered). When rush orders are placed for this customer you don't want the operator filling in the

Vehicle code, *Route*, or *TRF* fields so that there is no confusion about deliveries, and you don't consequently schedule a truck that is not needed. So in these circumstances you want to hide these fields so the operator cannot fill them in. This customer's purchase order numbers for a rush order always start with *RUSH*, so you can use this as a means of deciding when to hide/show these fields.

Note that *Vehicle code*, *Route*, and *TRF* are all custom form fields that have been added to the *Order Header* form.

On the *Sales Order Header* form, highlight the *Customer purchase order* field and use the *Alt+F8* shortcut keys to call up the *VBScript Editor*. Double-click on the *OnAfterChange* event name to create the *CustomerPurchaseOrder_OnAfterChange* function, and the cursor will be placed within it. Add the following three lines of code that create a variable, extract the first four characters of the customer purchase order number, and convert these characters to uppercase.

```
Dim FirstFour
FirstFour = Left(OrderHeader.CodeObject.CustomerPurchaseOrder,4)
FirstFour = UCase(FirstFour)
```

Add a new line directly below this that contains an *IF* statement that checks to see if the first four characters of the customer purchase order number contain the text *RUSH*.

```
If FirstFour = "RUSH" then
```

Place the cursor on a new line immediately after this. Locate the *Route* variable within the *OrderHeader* section of the *Variables* pane and double-click it. This will add the full name of the *Route* variable to your code. Follow this with a space, an equal sign, and another space. On the *Field Properties* pane, click the dropdown list alongside the *Visible* property and select *False*. Click the *Insert VBScript Code* button on the toolbar, and the XML to set this property will be added to your code. The line that you have just added should look the same as the line below.

```
OrderHeader.CodeObject.Route = "<Field Visible='false' > </Field>"
```

Perform the same tasks for the *Vehicle Code* and *TRF* fields. Immediately below these lines add a new line containing the word *ELSE*.

Place the cursor on a new line directly below the line containing the *ELSE* statement. Locate the *Route* variable name again and double-click it. Add a space, an equal sign, and another space. Use the dropdown list against the *Visible* field property to change it to *True*, then click on the *Insert VBScript Code* button to add the field properties XML. This line should match the one below.

```
OrderHeader.CodeObject.Route = "<Field Visible='true' > </Field>"
```

Perform the same tasks for the other two variables, making sure that these are also set to *True*. Close the *IF* statement by adding a line containing the text *END IF*. When you have finished your code should match the following.

```
Function CustomerPurchaseOrder_OnAfterChange()
  Dim FirstFour
  FirstFour = Left(OrderHeader.CodeObject.CustomerPurchaseOrder,4)
  FirstFour = UCase(FirstFour)

  If FirstFour = "RUSH" then
    OrderHeader.CodeObject.Route = "<Field Visible='false' > </Field>"
    OrderHeader.CodeObject.VehicleCode = "<Field Visible='false' > </Field>"
    OrderHeader.CodeObject.TRF = "<Field Visible='false' > </Field>"
  Else
    OrderHeader.CodeObject.Route = "<Field Visible='true' > </Field>"
    OrderHeader.CodeObject.VehicleCode = "<Field Visible='true' > </Field>"
    OrderHeader.CodeObject.TRF = "<Field Visible='true' > </Field>"
  End If
End Function
```

Click on the *Syntax Check* button on the toolbar to make sure that there are no typos, and exit the screen where you edit the VBScript. At the *VBScript Editor* screen click on the *Save* button. Back at the *Sales Order Entry* screen, when you change the value against the *Customer purchase order* field then *Tab* off it, if the first four characters of the *Customer purchase order* field are *RUSH* (in uppercase, lowercase, or any combination of upper and lowercase) the *Route*, *Vehicle Code*, and *TRF* fields will disappear from the *Order Header* pane. If the first four characters of the *Customer purchase order* field are anything else, these fields will appear on the screen.

XAML Sample

The *XAML code* field property is used to highlight the caption of a field. This could be because the value associated with it is outside a prescribed limit, or that this field needs to be populated because of the contents of another field.

The code in the example below appears against the *Customer Information* form in the *Sales Order Entry* program. When this form fires its *OnRefresh* event, a check is made to see if this is a sub-account (by checking if the *Master Account* field contains a value) and if it is, the caption against the *TRF* field has the *Cappucino* XAML theme applied to it. If it is not a sub-account, the XAML theme is removed by setting it to *none*.

```
Function CustomerInformation_OnRefresh()
  If CustomerInformation.CodeObject.MasterAccount <> "" Then
    OrderHeader.CodeObject.TRF = "<Field XAMLCode='Cappuccino' > </Field>"
  Else
    OrderHeader.CodeObject.TRF = "<Field XAMLCode='none' > </Field>"
  End If
Function CustomerInformation_OnRefresh()
```

Button Sample

Buttons can be added against the value portion of a field on an entry form. When the button is clicked an *OnButtonClick* event is fired, and if VBScript code appears against the *OnButtonClick* function for that field, the script is run. If a standard button already exists against this field (such as a browse, or a date picker) this field will have two buttons against it.

Adding a button is as simple as adding text against the *Button wording* field property, and applying it to a field's variable (see **Figure 7-47**).

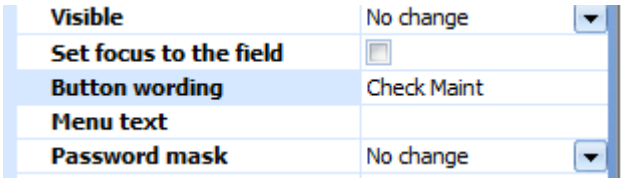


Figure 7-47: Adding the *Button wording* before clicking the *Insert VBScript Code* button

After adding the code it should appear similar to that below.

```
OrderHeader.CodeObject.VehicleCode = "<Field <Button>Check Maint</Button> </Field>"
```

In this example the *Vehicle Code* field (a scripted field on the *Order Header* form in *Sales Order Entry*) has both a dropdown list and a button against it. The dropdown list contains a list of available delivery trucks, and when the button is clicked a check is made against a maintenance schedule table in SQL Server to see if the vehicle is available to perform deliveries on the order's *Ship date*.

On the *Order Header* form in *Sales Order Entry*, highlight the *Vehicle Code* field and use the *Alt+F8* shortcut keys to call up the *VBScript Editor*. Double-click on the *OnRefresh* event to either create a new *OnRefresh* function, or open an existing one. Place the cursor on a new line between the end of any existing code and the *End Function* statement. Locate the *VehicleCode* variable name within the *OrderHeader* section of the *Variables* pane, and double-click it to add its full name to your code.

On the *Field Properties* pane, locate the *Button wording* property and add the *Check Maint* text (see **Figure 7-47**). Against the *Drop down list* property add the list of trucks, *1 Ton*, *3 Tons*, *5 Tons* and *7 Tons*. These must be separated with semi-colons. Click on the *Insert VBScript Code* button on the toolbar to add these field properties to your code. You have now added the code to create both the dropdown list and the button.

Exit back to the *VBScript Editor*, and double-click the *OnButtonClick* event to create the *VehicleCode_OnButtonClick* function. Against this *OnButtonClick* function, add the code to interrogate the SQL Server table to check the maintenance schedule (a snippet of which can be seen in Figure **7-48**). Alternatively you could just add a message box statement so that you can see the button has fired the *OnButtonClick* event. Once this has been completed, exit back to the *VBScript Editor* screen, and click on the *Save* button.

```
130
131 Function VehicleCode_OnButtonClick()
132 If OrderHeader.CodeObject.VehicleCode <> "" Then
133   ' Check to see if vehicle selected will be available on desired day
134   Dim VehicleLog, Server, DatabaseName
135   Dim objCN, strConnection
136   Set objCN = CreateObject("ADODB.Connection")
137
138   strConnection = "Driver={SQL Server};Server=Vehicle;Database=Maintenance;Trusted_Co
```

Figure 7-48: A snippet of the code that will verify the vehicle's maintenance schedule

The next time that the *OrderHeader_OnRefresh* event fires, the dropdown list and button will appear against this field (see **Figure 7-49**).

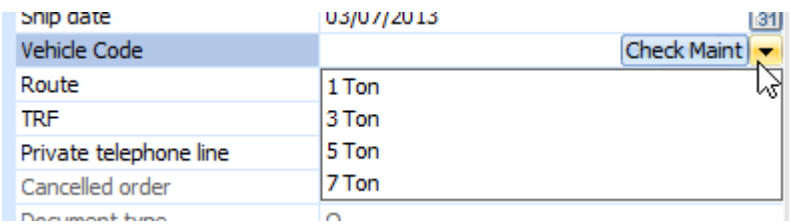


Figure 7-49: The dropdown list and button against the *Vehicle Code* field

Clicking on the button causes the code against the *VehicleCode_OnButtonClick* function to be executed (see **Figure 7-50**).

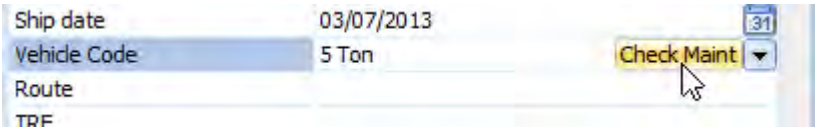


Figure 7-50: Clicking on the button to check the vehicle's maintenance schedule

Menu Sample

When a menu is added to a field against a form, an icon is displayed when the mouse pointer moves over this field. An example of the menu icon can be seen in **Figure 7-51**, where the mouse pointer moved over the *Currency* field.

Figure 7-51: The menu icon

A menu is added to a field by supplying text against the *Menu text* field property, and applying this to the field's variable name.

In this example the menu is added to the *Currency* field on the *Customer Information* pane of the *Sales Order Entry* program. When an order is added a check is made to see the currency code against this customer. If the currency code is anything other than US Dollar, the code is added for a menu item to look up the current exchange rate for that currency code. If the currency is US Dollar the menu item does not appear.

Call up the *Sales Order Entry* program, right-click on the *Currency* field and select *Macro for:IMP040L4* from the displayed menu. The *VBScript Editor* screen is displayed. Highlight the *OnRefresh* event and click on the *Edit VBScript* button on the toolbar. If the *CustomerInformation_OnRefresh* function already exists, the cursor will be placed at the beginning of it and you will need to start adding your code between the end of the existing code and the *End Function* statement. If this function did not already exist it will be created for you and the cursor placed on a blank line within it.

The first line of code performs a check to see which currency code is held against this customer. Start the line with *IF* followed by a space. Then locate the variable *CurrencyValue* within the *CustomerInformation* section of the *Variables* pane, and double-click it. The full name of this variable will be added to your code. Immediately after this, add a space, an equal sign, another space, a double-quote, a dollar sign, another double quote, a space, and the word *THEN*. When you have finished the line should match the following.

```
If CustomerInformation.CodeObject.CurrencyValue = "$" then
```

The next line of code describes what should happen if the currency code is a Dollar sign, which is not to display the menu. There is no way of adding a blank field property, because as it is blank, when you click on the *Insert VBScript Code* button the field property will be ignored. To get around this you will add the field property as if you were creating a menu item, and then edit the XML's contents once it has been added to your code.

On a new line below the one you have just added, locate the *CurrencyValue* variable in the *Variables* pane and double-click it. Follow this with a space, an equal sign, and another space. Locate the *Menu text* property within the *Field Properties* pane and add the text *Check Current Exchange Rate* against this (see **Figure 7-52**).

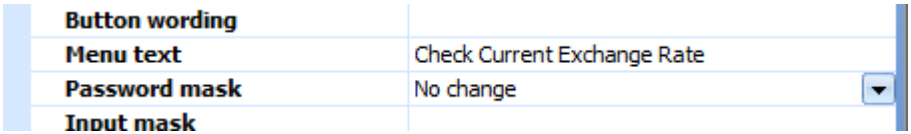


Figure 7-52: Adding text to the *Menu text* field property

Click on the *Insert VBScript Code* button on the toolbar to add the field property XML to your code. The line that you have just added will match the one below (which has wrapped around so appears as two lines but is one long line).

```
CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current
Exchange Rate' > </Field>"
```

Remove the text that appears against the *MenuText* XML attribute, so that you are left with just the two single quotes. When you have finished editing this line it should match the following.

```
CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='' > </Field>"
```

Add a new line below this one and add the statement *ELSE* to it. Add another new line below that one. Locate the *CurrencyValue* variable and double-click it, followed by a space, an equal sign, and another space. The *Menu text* field property should still contain the *Check Current Exchange Rate* text, so click on the *Insert VBScript Code* button to add the field property XML. Add a new line below this one that must contain the statement *END IF*. Once you have completed this your script should match that in **Figure 7-53**).

```
Function CustomerInformation_OnRefresh()
  If CustomerInformation.CodeObject.CurrencyValue = "$" then
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='' > </Field>"
  Else
    CustomerInformation.CodeObject.CurrencyValue = "<Field MenuText='Check Current Exchange Rate' > </Field>"
  End If
End Function
```

Figure 7-53: The completed code against the *OnRefresh* function

Click on the *Syntax Check* button to make sure that you have no typos. Exit back to the *VBScript Editor* screen and click the *Save* button. You will be returned to the *Sales Order Entry* program. Enter a customer code that does not have *$* as its currency code and tab off the field. Moving the mouse pointer over the *Currency* field should display the menu icon that appears in **Figure 7-51**. Clicking on this menu icon should display your menu text (see **Figure 7-54**).

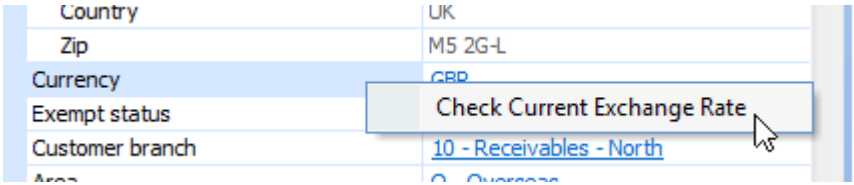


Figure 7-54: The displayed menu item

Clicking on the menu item will not perform any task, as none has been configured against it. To prove that your code is working correctly, click on the *New Order* button on the toolbar to start a new order, and supply a customer code that has the US Dollar as its currency code. When you move over the *Currency* field, no menu icon will be displayed.

The next step is to add the code that will be run if this menu option is selected. Highlight the *Currency* field on the *Customer Information* form and use the shortcut keys *Alt+F8* to call up the *VBScript Editor*. Highlight the *OnMenuSelect* event and click on the *Edit VBScript* button on the toolbar. The *CurrencyValue_OnMenuSelect* function will be created for you and the cursor placed within it. Whatever code you add here will be run when the menu item is clicked.

To prove that this is working you could just add a message box statement, in which case when you click on the menu item the message box will appear. **Figure 7-55** shows the start of a script that will perform a lookup of the current exchange rate for the currency code against this customer.

```
Function CurrencyValue_OnMenuSelect()
' Look up current exchange rate
Dim CurrSite, CurrResult

Select Case CustomerInformation.CodeObject.CurrencyValue
  Case "EUR"
    CurrSite = "http://www.xe.com/currencyconverter/convert/?Amount=1&From=EUR&To=USD"
  Case "GBP"
    CurrSite = "http://www.xe.com/currencyconverter/convert/?Amount=1&From=GBP&To=USD"
  Case "CAD"
```

Figure 7-55: The start of the script to check the exchange rate when the menu item is selected

Unsetting Field Properties

When a field property is set against a field it typically remains set until either this field property is set to something else, or the operator exits the program. Under some circumstances you may need to set a field property against a field back to its default setting. If you clear out the value against this field property in the *Field Properties* pane and click on the *Insert VBScript Code* button, the XML that is added will be blank, as per the example below:

```
CustomerInformation.CodeObject.Customer = "<Field > </Field>"
```

This will not change any of the currently set field properties. The way to set the field property back to its default setting is by setting it to another value on the *Field Properties* pane, and then once this XML has been added to your code, remove the value associated with it. This is probably best explained with an example.

If you had previously set the background color of a field to *Gold* using the code below, you may not know the setting of the default background color setting.

```
OrderHeader.CodeObject.Salesperson = "<Field Background='255204000' > </Field>"
```

At the point where you need to set this field property back to its default value, you need to supply the *Salesperson* variable name, and the *Background color* field property with any value. Remove the value that appears within the two single quotes, so you end up with the following, and the field property will be set back to its default value.

```
OrderHeader.CodeObject.Salesperson = "<Field Background='' > </Field>"
```

There are some exceptions to this rule. Where a field property has a checkbox against it the field property can only be *True* or *False*. If this property is set to *True*, and later is blank, the current status of this field property will remain in effect. For example, checking the *Bold* field property and applying it to the *Salesperson* field will display the contents of the *Salesperson* field in bold. The code for this appears below.

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold='true' > </Field>"
```

If you were to change this to remove the word *true* (as per the example below) the field's content would remain in bold.

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold='' > </Field>"
```

In this case you must set the *IsBold* field property to false.

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold='false' > </Field>"
```

Another exception is where you have used the *XAML code* field property. To remove this field property it must be set to *none*, as per the example below.

```
OrderHeader.CodeObject.Salesperson = "<Field XAMLCode='none' > </Field>"
```

If you have manipulated the height of a field using the *Field height* field property, you set this back to its default value by setting it to *-1*.

```
OrderHeader.CodeObject.Salesperson = "<Field Height='-1' > </Field>"
```

Accessing information on other forms

You can access the values on other forms using their variable names. However, you must consider the sequence in which the forms are updated, as the values that they currently contain may not be those you are expecting.

For example, in the *Sales Order Entry* program, after entering the customer account code the *Order Header* form is refreshed. This is followed by the *Customer Information* form, and then some others. When the *Order Header* form's *OnRefresh* event fires the *Customer Information* form has not yet been updated with the information for the new customer. If you have code against the *Order Header* form's *OnRefresh* function that checks/validates some information on the *Customer Information* form, you may not always get the required results as the values will belong to the previous customer.

If your code against the *OrderHeader_OnRefresh* function checks the currency code on the *Customer Information* form and then sets a value on its own form. This value would be checking the currency value for the previous customer, not the current one. In this case there is a simple solution in that you should put the code against the *Customer Information* form's *OnRefresh* function.

Checkboxes

Several forms contain checkboxes, such as the period end programs. A checkbox value is 0 for unchecked, and 1 for checked. This means that you can both read the current status of the checkbox, and set the value. The sample of code below is from the *Information* form of the *AR Period End* program, and shows both the reading and setting of the checkbox value.

Whenever the radio button against the *Function* field is changed, a check is made to see the current state of *the Close this application* checkbox. Whatever state it is currently in, it is changed to the opposite state.

```
Function FunctionA_OnAfterChange()
  If Information.CodeObject.CloseThisApplication = "0" then
    Information.CodeObject.CloseThisApplication = "<Field Value='1' > </Field>"
  Else
    Information.CodeObject.CloseThisApplication = "<Field Value='0' > </Field>"
  End If
End Function
```

Radio Buttons

A field that uses radio buttons must always have one button set, and only one of the buttons can be set at a time. The buttons against the field are numbered, staring with *000000000*. Just like with checkboxes, the contents of fields containing radio buttons can be both read, and set.

There may be more radio buttons against a field than first appears, as the logic of the program may prevent certain radio buttons from appearing. **Figure 7-56** shows the *Function* field of the *AR Period End* program. The *Balance* radio button is *000000000*, the *Month end only* radio button is *000000001*, the *Month end and purge* radio button is *000000002*, and the *Purge only* radio button is *000000005*. Because it is not an appropriate time to perform a year end, the radio buttons numbered *000000003* and *000000004* are hidden from the operator.

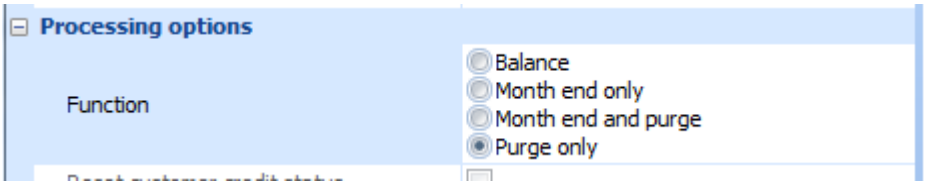


Figure 7-56: The radio buttons against the *Function* field in the *AR Period End* program

The *AR Period End* program also has a field called *Date entry* that contains radio buttons. The sample code below reads the current value of the *Function* field to see which radio button is selected. If the

Month end and purge radio button (000000002) is selected against the *Function* field, the *Day of month* radio button is set against the *Date entry* field. If the radio button against the *Function* field is anything else, the *Manual entry* radio button is set against the *Date entry* field.

```
Function FunctionA_OnAfterChange()
  If Information.CodeObject.FunctionA = "000000002" then
    Information.CodeObject.DateEntry = "<Field Value='000000001' > </Field>"
  Else
    Information.CodeObject.DateEntry = "<Field Value='000000002' > </Field>"
  End If
End Function
```

Form Sizes and Limitations

The maximum number of fields that can appear on a form is 150. This includes fields that appeared on the form by default, *Standard* fields, *Custom Form* fields, and *Scripted* fields. Prior to SYSPRO 7 the maximum number of fields that could appear on a form was limited to 100.

Custom Form fields can have validation rules configured against them (see **Figure 7-57**). Up to 100 custom form fields that have validation rules configured against them can be added to a SYSPRO form. Prior to SYSPRO 7 there was a limitation of 25 custom form fields with validation rules configured against them per form.

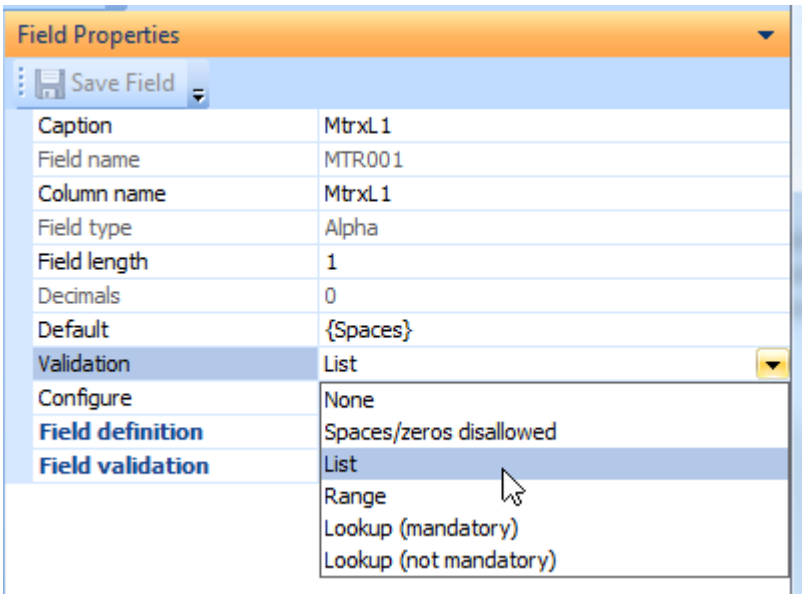


Figure 7-57: *Custom Form Field* validation rules

The maximum size of a multiline entry field is 1,750 characters. Prior to SYSPRO 7 each of these fields was limited to 100 characters.

The maximum length of an alphanumeric field against a form is 255 characters. Prior to SYSPRO 7 the maximum length was 100 characters. Note that the maximum size of an alphanumeric field on a customized pane form is 3,000 characters.

VBScript Changes Because of Increased Field Sizes in SYSPRO 7

SYSPRO 7 contains many fields that have increased in length. If you specifically check a value of a SYSPRO field in your code, you need to check that this code still works as expected in SYSPRO 7. This is more likely to happen when using numeric values.

For example, if you are using numeric customer account codes and your code under SYSPRO 6.1 checked to see if the current customer code was *0000001*, under SYSPRO 7 you will probably need to change this to check for *000000000000001*. In this case the *IF* statement may need to change from the example below, to the one that appears below that.

```
If CustomerInformation.CodeObject.Customer = "0000001" then

If CustomerInformation.CodeObject.Customer = "000000000000001" then
```

Chapter 8 - Listviews and Data Grids

A listview is a means of displaying data in columns and rows, in a similar way to a spreadsheet. SYSPRO contains three types of listviews: *Standard Listviews*, *Data Grids*, and *Customized Pane Listviews*. This chapter covers the *Standard Listviews* and *Data Grids*. It explains how you can extend the functionality of the *Listviews* and *Data Grids* using the macro events and VBScripting (the basics of using these are covered in Chapter 5 of the first Power Tailoring book).

The *Customized Pane Listviews* will be covered in detail in Chapter 11.

Standard Listviews

Standard Listviews (referred to as *Listviews*) are used for displaying data. They contain *Columns*, and are populated with *Rows* of data. A *Cell* contains the individual values that are displayed.

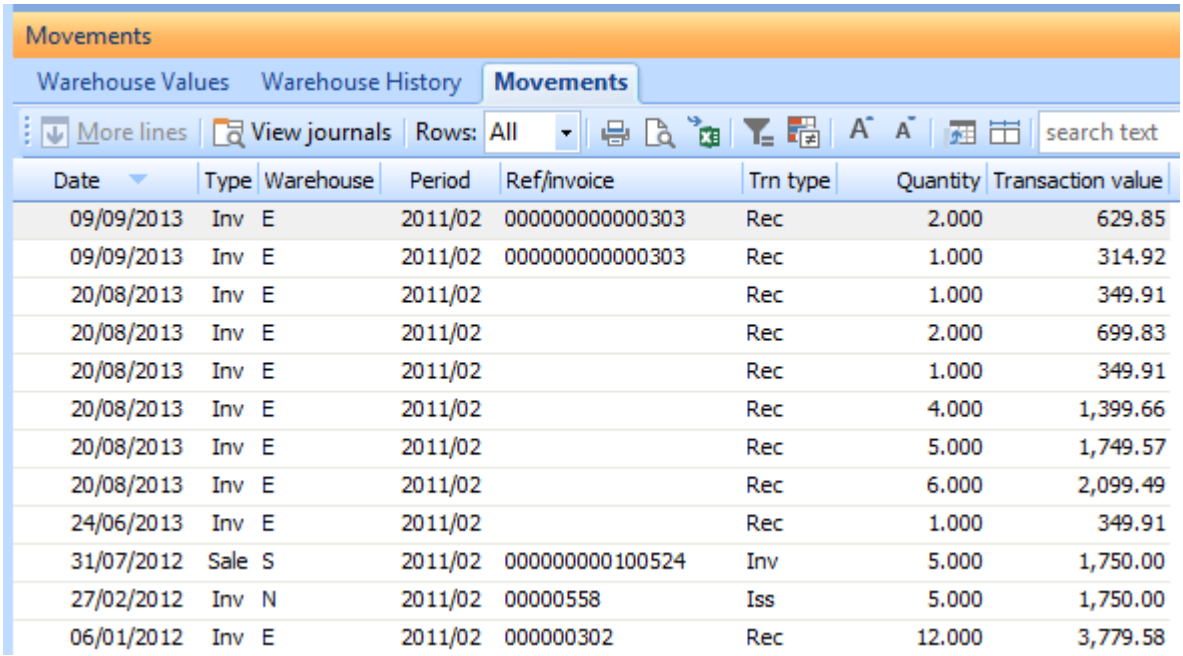
The columns that can be displayed in the listview are predefined by the SYSPRO developer (from now on referred to as the *developer*). Most listviews will display all of the available columns by default. However, some listviews have so many columns available that the *developer* selects those they consider to be the most likely to be required and displays these by default. The rest are added to a *Field Chooser* so that they are available to the Operator/Administrator to drag onto the listview.

The Operator/Administrator may decide not to use some fields that are present on the listview by default, in which case they can drag these from the listview and they will no longer appear. The fields that were dragged from the listview will then appear in the *Field Chooser*, so if they decide that they want them to appear on the listview again, it is a simple task to drag them back.

Listviews are read-only, so the operator cannot make changes to the data that is displayed. However, they can use listview features such as *Grouping*, *Filters*, and *Conditional Formatting* to change the way that the information is displayed, and which items are included.

Listviews have three *Macro events* associated with them: *OnPopulate*, *OnRowSelected*, and *OnDblClick*. Adding VBScript to these allows a developer to manipulate what is displayed, and how it appears. In addition, they can specify what should happen if the operator selects, or double-clicks, on a row.

Just like a form, each listview has a name that appears at the top of the pane. In many cases the listviews have been nested (to save on screen real estate) and each will appear as a tab. **Figure 8-1** shows the *Inventory Query* program which by default contains three listviews that have been nested. The name of the listview appears in the tab, and the name of the currently selected tab will appear in bold. The *Movements* listview is currently selected, and eight columns are visible (*Date*, *Type*, *Warehouse*, *Period*, *Ref/invoice*, *Trn type*, *Quantity*, and *Transaction value*). Twelve rows of data can also be seen. The other listviews (that appear as tabs) are *Warehouse Value*, and *Warehouse History*.



Date	Type	Warehouse	Period	Ref/invoice	Trn type	Quantity	Transaction value
09/09/2013	Inv	E	2011/02	000000000000303	Rec	2.000	629.85
09/09/2013	Inv	E	2011/02	000000000000303	Rec	1.000	314.92
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	2.000	699.83
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	4.000	1,399.66
20/08/2013	Inv	E	2011/02		Rec	5.000	1,749.57
20/08/2013	Inv	E	2011/02		Rec	6.000	2,099.49
24/06/2013	Inv	E	2011/02		Rec	1.000	349.91
31/07/2012	Sale	S	2011/02	000000000100524	Inv	5.000	1,750.00
27/02/2012	Inv	N	2011/02	00000558	Iss	5.000	1,750.00
06/01/2012	Inv	E	2011/02	000000302	Rec	12.000	3,779.58

Figure 8-1: The *Movements* listview showing eight columns and twelve rows

Starting the VBScript Editor

Invoking the *VBScript Editor* screen from a listview is performed slightly differently if it was invoked from a form. To start the *VBScript Editor*, right-click on one of the *Column Headers* and select *Customize* | *Macro for: xxxxxxxx*, where xxxxxxxx is replaced with the name of the listview (see **Figure 8-2**). *Column Headers* are the blocks that appear above the individual columns of the listview. In **Figure 8-1** the blocks containing the text *Date*, *Type*, *Warehouse*, etc., are the column headers.

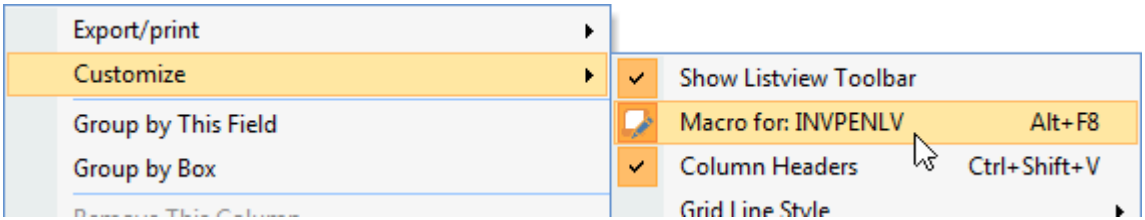


Figure 8-2: Using the context-sensitive menu to start the *VBScript Editor* for a listview

Alternatively you can highlight either a column header (or the area of the listview that can contain data) and use the *Alt+F8* shortcut keystrokes to invoke the editor. The *VBScript Editor* screen is displayed, and is split into two sections: *Options*, and *Available Events* (see **Figure 8-3**).

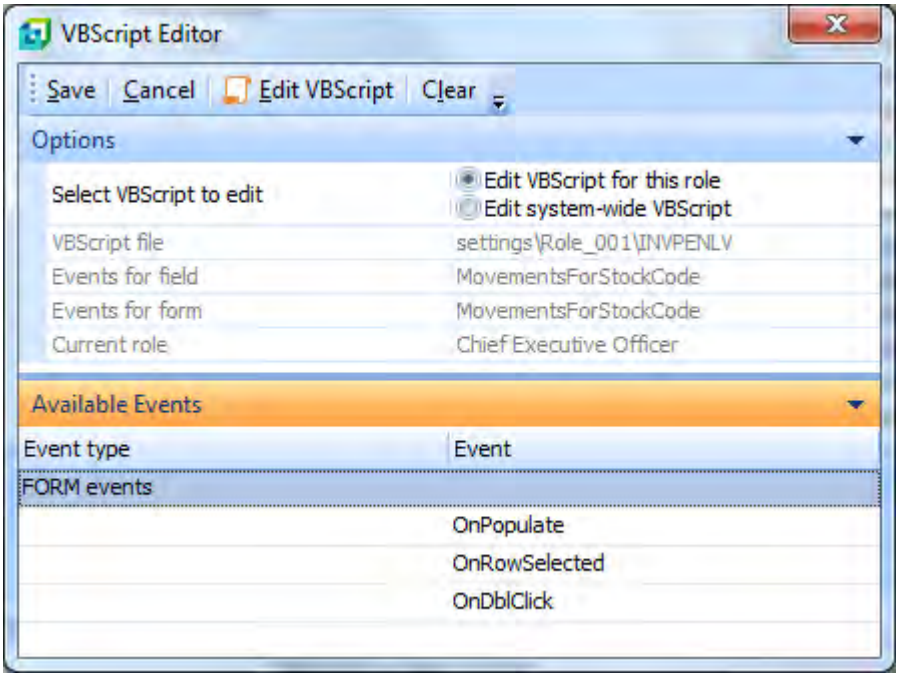


Figure 8-3: The *VBScript Editor* screen showing the available form events for a listview

The *Options* section of the *VBScript Editor* screen contains the name of the script being edited. If you are editing the VBScript for a role, the *Current role* will be displayed, as well as radio buttons so that you can edit either the VBScript for this role, or system-wide.

If you were editing the VBScript for a form, the *Events for form* field would contain the name of the form, and the *Events for field* field would contain the name of the field that was highlighted when the editor was started. As there is no notion of a field on a listview, both of these will contain the name of the listview.

The *Available Events* section contains the three form events: *OnPopulate*, *OnRowSelected*, and *OnDblClick*. Highlight the event to be created/updated and click on the *Edit VBScript* button on the toolbar.

Arrays

At various times, the content of the listview is stored in an array in memory. The array can be thought of as a spreadsheet that has both its columns and rows numbered from zero upwards. Looking back at **Figure 8-1**, the column *Date* would be column 0, *Type* would be 1, Warehouse 2, etc., up to *Transaction value* which would be 7. The row with the date of *24/06/2013* would be row 8, *31/07/2012* would be row 9, *27/02/2013* would be row 10, and *06/01/2012* would be row 11.

During the time that the listview is being populated, each listview consists of two arrays, one that contains the values supplied by the program, and one where you can modify the values or field properties before the listview in populated. The array containing the values supplied by the program has the same name as the listview referred to within the program. The array that you can modify has the same name, but with the suffix *_OUT*. The size of the arrays depends on how many columns are in the listview, and how many rows contain data.

This is best explained with an example. Within the *Inventory Query* program is the *Movements* listview, and within the program it is referred to as *MovementsForStockCode*. The array containing the values supplied by the program is called *MovementsForStockCode*, and the array that can have its values changed is called *MovementsForStockCode_OUT*. Each of these arrays can be accessed using its variable name that appears within the *Variables* pane. These can be seen in **Figure 8-4**.

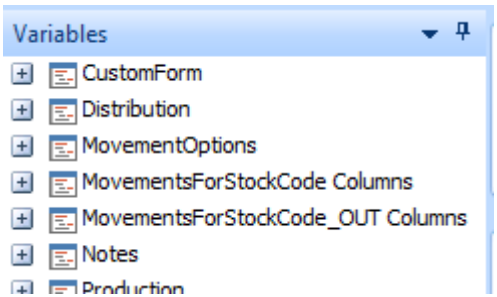


Figure 8-4: The array variable names in the *Variables* pane

Expanding either of these variable names reveals the list of columns that can appear within this listview, along with the column's three digit array number within braces (see **Figure 8-5**).

The column's array number relates to its default position in the listview. The default position for the *Date* column in this *Movements* listview is the first column, or column 1. As the columns in a listview always start at zero (instead of one) the *Date* column variable name will have 000 as its array number, so will appear as *Date(000)*. If the *Date* column is dragged to another position within the listview, its array number remains *000*. If array numbers did not work this way the VBScript code could be broken when an operator dragged columns around within the listview.

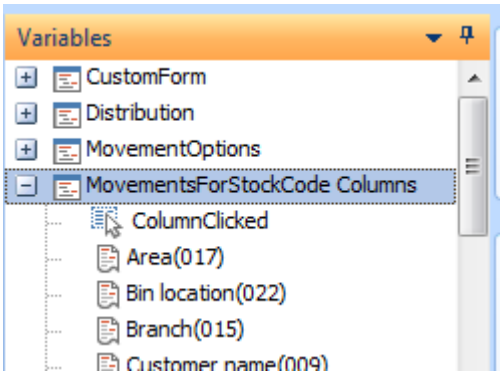


Figure 8-5: The expanded *MovementsForStockCode* section showing the columns

The column number is only one part of accessing a cell in an array; the other is the row number. To refer to a specific cell you need to provide both the column and row number, remembering that both the row and column numbers in an array start at zero. Looking back at **Figure 8-1**, the cell containing the date *24/06/2013* would be referenced as (000,8), and the one containing the date *27/02/2013* would be referenced as (000,10).

The fifth column is *Ref/invoice*. The value *00000558* appears in this column against the eleventh row, so the cell that contains it is referenced as (004,10) which equates to the fifth column, eleventh row.

Using the Variables to Retrieve/Display Values

As mentioned above, during the population of the listview there are two arrays for each listview. One has the same name as the listview, and the other has the same name but with the suffix of *_OUT*. Both of these will appear in the *Variables* pane (as can be seen in **Figure 8-4**). The one with the same name as the listview contains the values that are provided by the program to populate the listview, the one with the *_OUT* suffix is where you change the values or field properties that you want changed.

Expanding either of these sections within the *Variables* pane shows the columns with their array number.

If you expand the variable with the same name as the listview, and double-click on one of the variable names within it, the full name of that variable will be added to your code. Below is an example of what will be added when clicking on the *Ref/invoice(004)* variable name. Note that the column's array number of 004 is present, but the row's array number is always inserted as a zero.

```
MovementsForStockCode.CodeObject.Array(004,0)
```

Looking back at **Figure 8-1**, the fifth column, eleventh row, cell contains the value *00000558*. To retrieve the value in this cell you must provide both its column and row array numbers. Double-clicking on the *Ref/invoice* variable name in the *Variables* pane inserts the line of code that appears above. You need to change the row array number to contain the value 10 (for row 11). The code below displays this cell's value in a message box during the *OnPopulate* event.

```
Function MovementsForStockCode_OnPopulate()
  msgbox MovementsForStockCode.CodeObject.Array(004,10)
End Function
```

This code displays just the contents of this cell when the *OnPopulate* event fires (see **Figure 8-6**).

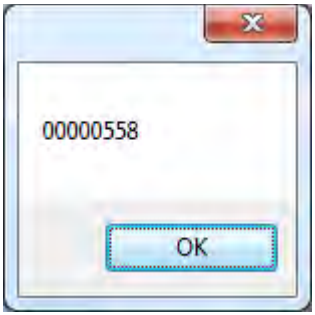


Figure 8-6: Display the contents of column 5, row 3 in a message box

Updating a Cell with a Value or Field Property

It is possible to change the displayed value or field property of a cell (or range of cells) within the listview when it is being populated. This is done by changing the value or setting the field property of the *_OUT* array. Not all *Field Properties* are available for use with a listview.

The following *Field Properties* can be configured against a specified cell:

- Background color
- Foreground color
- Italic
- Bold
- Underline
- Icon
- Tooltip
- Value
- XAML code

If the following *Field Property* is set against a cell it prevents the whole row from being displayed:

- Visible = false

The use of the field properties is probably best explained with a few simple examples.

Background Color

To change the *Background Color* of the cell used in the example above (that contained the value of 00000558) you could add the following code to the *MovementsForStockCode_OnPopulate* function. Note that this is one line of code even though it has wrapped around below. The results of this code can be seen in **Figure 8-7**.

```
MovementsForStockCode_OUT.CodeObject.Array(004,10) = "<Field Background='255153000'
> </Field>"
```

The line of code above was added by expanding the *MovementsForStockCode_OUT* section of the *Variables* pane, locating the *Ref/Invoice(004)* variable name, and double-clicking on it. This added the following line of code.

```
MovementsForStockCode_OUT.CodeObject.Array(004,0)
```

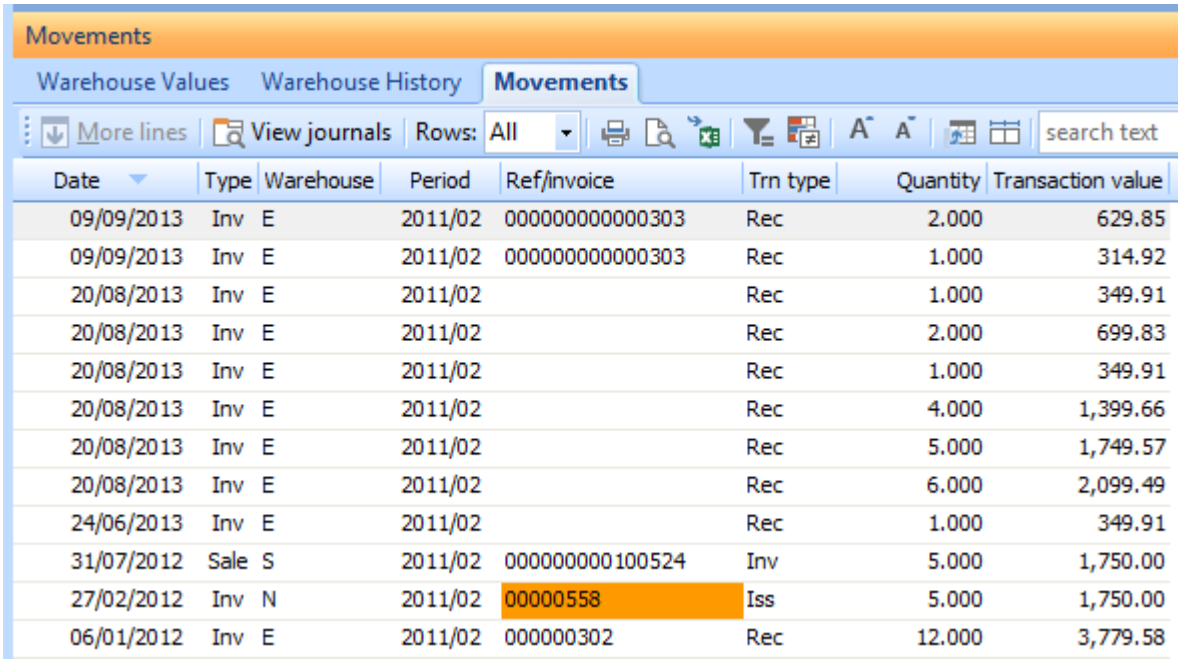
As a variable added this way will always point to array row zero, this had to be changed to point to array row ten (so that it would access listview row 11).

The next step was to add a space, and equal sign, and another space. At this point the code matched the following.

```
MovementsForStockCode_OUT.CodeObject.Array(004,10) =
```

Leaving the cursor after the last space, locate the *Background color* property within the *Field attributes* section of the *Field Properties* pane. Select a color from the color chooser, or supply the RGB (Red, Green, Blue) number (see **Figure 8-8**). Once the color has been selected, click on the

Insert VBScript Code button on the *Field Properties* toolbar to add the XML to your existing code to set this property.



Date	Type	Warehouse	Period	Ref/invoice	Trn type	Quantity	Transaction value
09/09/2013	Inv	E	2011/02	000000000000303	Rec	2.000	629.85
09/09/2013	Inv	E	2011/02	000000000000303	Rec	1.000	314.92
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	2.000	699.83
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	4.000	1,399.66
20/08/2013	Inv	E	2011/02		Rec	5.000	1,749.57
20/08/2013	Inv	E	2011/02		Rec	6.000	2,099.49
24/06/2013	Inv	E	2011/02		Rec	1.000	349.91
31/07/2012	Sale	S	2011/02	000000000100524	Inv	5.000	1,750.00
27/02/2012	Inv	N	2011/02	00000558	Iss	5.000	1,750.00
06/01/2012	Inv	E	2011/02	000000302	Rec	12.000	3,779.58

Figure 8-7: The results of setting the *Background Color* field property against a specific cell

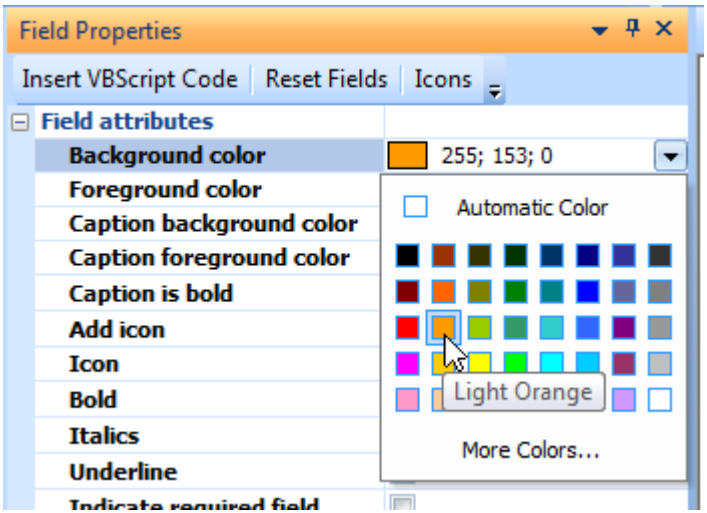


Figure 8-8: Adding the *Background color* field property

Value

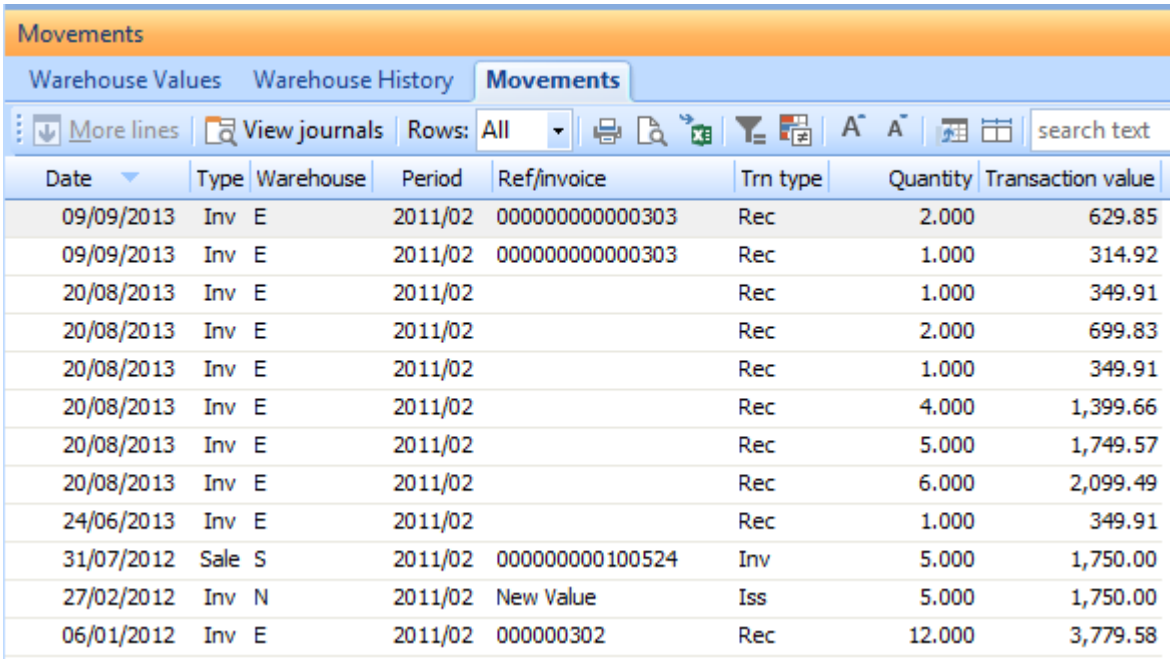
The value held within the cell can be replaced with another using the *Value* field property. This can be performed in one of two ways. The first is configured in a similar way to the *Background color* example above, just with a value field property instead of the color one. The line of code below will replace the contents of the cell in column 5, row 11 with the text *New Value*. Note that this is one line of text even though it has wrapped around on this page.

```
MovementsForStockCode_OUT.CodeObject.Array(004,10) = "<Field Value='New Value' >
</Field>"
```

The second is by just supplying the cell's new content as a string, as per the example below.

```
MovementsForStockCode_OUT.CodeObject.Array(004,10)= "New Value"
```

If either of these samples of code is added to the listview's *OnPopulate* function, the content of the cell is changed to *New Value*, (see **Figure 8-9**). Note that only the display changes; the back end data remains as it was.



Date	Type	Warehouse	Period	Ref/invoice	Trn type	Quantity	Transaction value
09/09/2013	Inv	E	2011/02	000000000000303	Rec	2.000	629.85
09/09/2013	Inv	E	2011/02	000000000000303	Rec	1.000	314.92
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	2.000	699.83
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	4.000	1,399.66
20/08/2013	Inv	E	2011/02		Rec	5.000	1,749.57
20/08/2013	Inv	E	2011/02		Rec	6.000	2,099.49
24/06/2013	Inv	E	2011/02		Rec	1.000	349.91
31/07/2012	Sale	S	2011/02	000000000100524	Inv	5.000	1,750.00
27/02/2012	Inv	N	2011/02	New Value	Iss	5.000	1,750.00
06/01/2012	Inv	E	2011/02	000000302	Rec	12.000	3,779.58

Figure 8-9: Replacing the cell's value with the text *New Value*

XAML Code

The *XAML code* field property enables you to use a XAML theme to highlight a particular cell, or range of cells. **Figure 8-10** shows the result of using the *XAML code* field property to highlight the contents of cell at column 8, row 4 of the listview. The XAML theme chosen is called *Blue*. The following is the code that caused the XAML to be applied in **Figure 8-10**. Note that it is one line even though it has wrapped around on this page.

```
MovementsForStockCode_OUT.CodeObject.Array(007,3) = "<Field XAMLCode='Blue' >
</Field>"
```

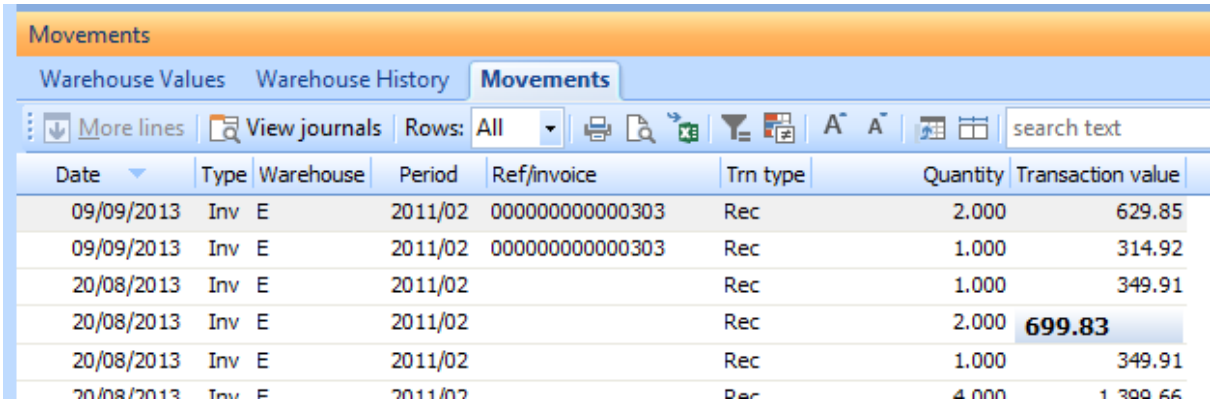


Figure 8-10: Using the *XAML code* field property to highlight a cell

Interacting with Listviews

Using the OnPopulate Event

The message box and field property examples above give you an idea of how the two listview arrays work when the listview's *OnPopulate* event fires. All of those examples specify what should happen to a specific cell, but life doesn't normally work that way. Not all stock codes have the same number of movements, and not all purchase orders have the same number of lines, so you need a means of dynamically working out which cells to access. If you attempt to read, or modify the field properties for a cell that doesn't exist, you will get an error similar to the one displayed in **Figure 8-11**.

Typically you need to look through the listview

- to highlight cells whose content are outliers to make sure that the operator notices them
- to programmatically start a process when an outlier is detected
- to set a field property for every line in the listview no matter how many lines there are

To perform any of these tasks, the first thing that you need to know is how many rows are in the two-dimensional array (a two-dimensional array is one that contains rows and columns). If you don't know

this you may have a problem similar to that encountered in **Figure 8-11** when you attempt to access the array.

The following steps occur when a listview is populated:

1. The program populates all of the columns for the first row in the array (the one with the same name as the listview). It then populates the columns of the second row of the array, etc.
2. The listview's *OnPopulate* event fires and any code against the *OnPopulate* function is run. This includes reading values from the first array, and setting any field properties in the second array (the one with the *_OUT* name).
3. A check is made to see if the listview's *OnPopulate* function has not been set to *false*.
4. The listview is populated with the contents of the first array, and if the *OnPopulate* function was not set to *false*, the field properties from the second array are applied to the cells.

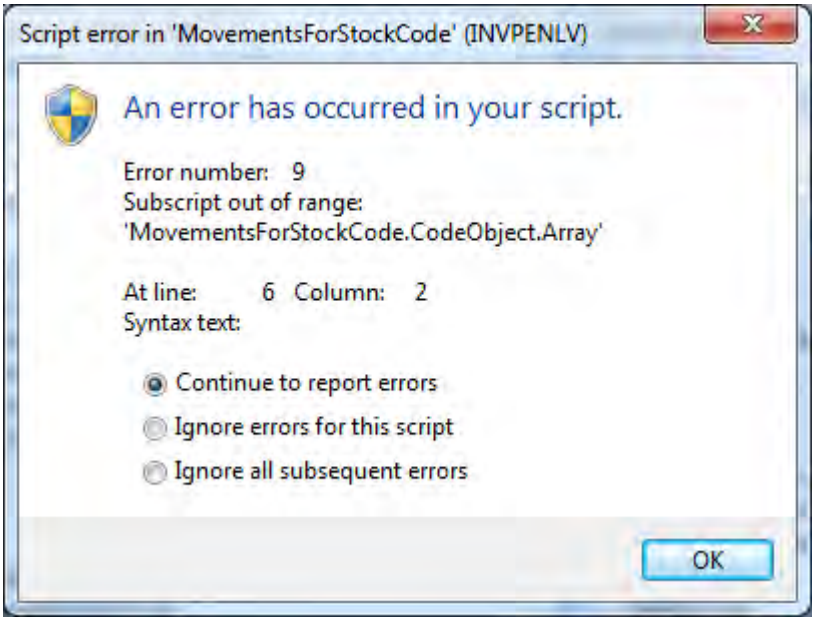


Figure 8-11: The error message when you attempt to access a part of the array that does not exist

Finding the Number of Rows in the First Array

The first array is a two-dimensional array. The columns are dimension one, and the rows are dimension two. You already know how many columns there are in the array, as it was specified by the *developer*. To be able to look through each row you need to know how many rows there are. This can be determined by looking at the upper boundary limit of the row dimension of the array, using VBScript's *UBound* function.

The following snippet of code (when added to a listview's *OnPopulate* function) will place the highest array row number into the *HighestArrayNum* variable. The lines starting with an apostrophe are comment lines and are ignored when processing occurs.

```
' Create the variables to be used
Dim  HighestArrayNum, MovementArray, Count, CurrentType

' Set the variable MovementArray to contain the whole first array
MovementArray = MovementsForStockCode.CodeObject.Array

' Work out the number of rows in the array, subtract one, and populate the
' HighestArrayNum variable with this value.
HighestArrayNum = UBound(MovementArray,2) -1
```

The first section creates the two variables that are going to be used in this piece of code (plus two more that will be used later).

The second section puts the contents of the first array into the variable called *MovementArray*.

The third section takes the contents of the *MovementArray* variable and passes it to VBScript's *UBound* function, along with the parameter of 2. The *UBound* function will find the upper boundary of the array. The number 2 states that it should find the upper boundary of the "rows" dimension. The -1 at the end states that one should be subtracted from this number (as we want the highest array row number, not the number of rows in the array). The result of this is placed in the *HighestArrayNum* variable.

For example, if there are five rows in the listview and ten columns, the *UBound* function works out that the upper boundary of the row dimension is 5, subtracts 1, leaving 4, and places this in the *HighestArrayNum* variable. The highest array row number is 4.

Iterating Through the Rows in a Listview

Now that you know how many rows exist in the listview, you can process one row at a time using a count, knowing that you will stop before causing an error. The count uses a *For*/*Next* loop to process the rows one at a time.

When you set up the count you need it to run from zero to the highest array row number (which you already have stored in the *HighestArrayNum* variable). By starting the count at zero (instead of one) the current value of the count is equal to the current array row number. This enables you to use this count number as the second parameter in the variable name when retrieving the information.

For example, assuming the count was called *Count*, the following code (if embedded within the *For*/*Next* loop) would enable you to change the contents of the *Type* column for the current line.

```
MovementsForStockCode_OUT.CodeObject.Array(001,Count)
```

If you started the count from one instead of zero, for each row you that was processed you would need to subtract one from the *Count* value before being able to use it in the code above. Failure to do this would mean that you were accessing the value in the array one row higher than you should.
The code snippet below is from the same *Movement* listview's *OnPopulate* function as the snippet above, and works in conjunction with that code. Because some of the lines wrap around, each line of code has a blank line between it and the next line of code to make it easier to follow. The lines are also indented within the *For*/*Next* statement, and within the *IF* statement, to make it easier to follow.

```
For Count = 0 To HighestArrayNum

  CurrentType = MovementsForStockCode.CodeObject.Array(001,Count)

  If CurrentType = "Inv" then

    MovementsForStockCode_OUT.CodeObject.Array(001,Count) = "<Field
Background='255204000' > </Field>"

  ElseIf CurrentType = "Sale" then

    MovementsForStockCode_OUT.CodeObject.Array(001,Count) = "<Field
Background='153204255' > </Field>"

  End If
Next
```

The first line of the code is a count that runs from zero to the highest row array number. This uses a *For*/*Next* loop that continues until it hits the *Next* statement at the end of this code snippet.

The second line of code uses the *Count* variable to access the contents of the *Type* column for the current row (the *Type* column is column 2, which is array column 1). The value is placed in the *CurrentType* variable.

The third line performs a test to see if the content of the *Type* column for the current row contains the text *Inv*. If it does, the fourth line of code is executed.

The fourth line uses the column array number of 001, and the *Count* variable to specify the row array number, to change the *Background color* field property of the *_OUT* array for this cell.

The fifth line of code performs a similar test to line three, just looking for the text *Sale*. If it does contain this value, the sixth line of code is executed.

The sixth line uses the column array number of 001, and the *Count* variable to specify the row array number, to change the *Background color* field property of the *_OUT* array for this cell.

The seventh line of code ends the *IF* statement.

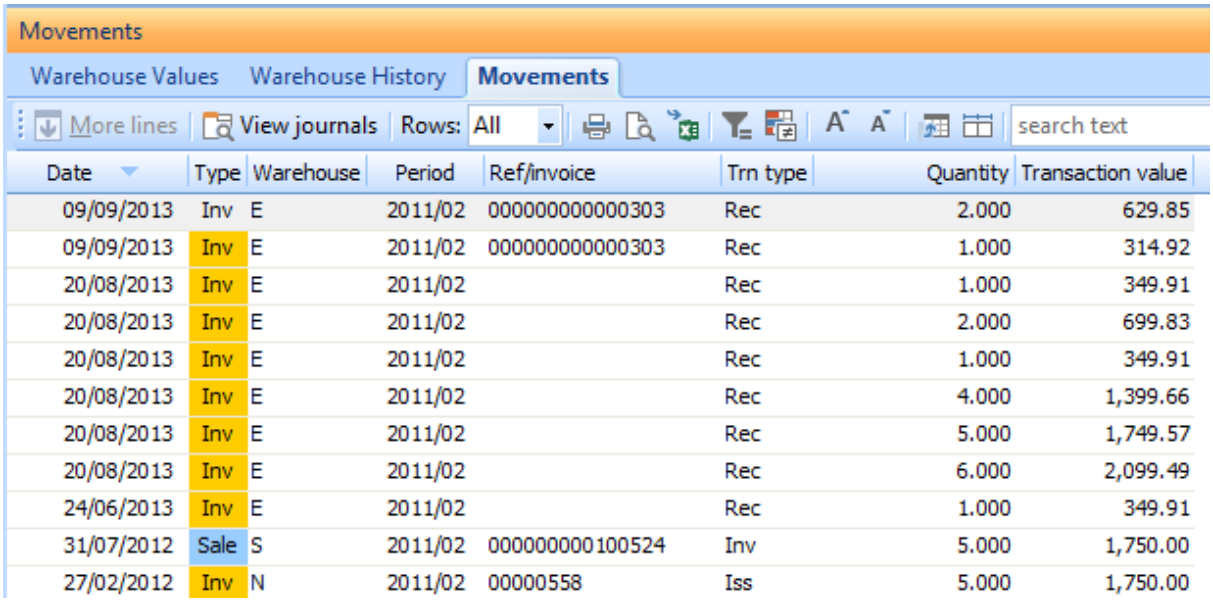
The eighth line of code contains a *NEXT* statement that starts processing of the next row, if there is one. If there isn't another row to process, the logic drops through to the next line of code.

```
Function MovementsForStockCode_OnPopulate()
  ' Create the variables to be used
  Dim  HighestArrayNum, Count, MovementArray, CurrentType

  ' Set the variable MovementArray to contain the whole listview array
  MovementArray = MovementsForStockCode.CodeObject.Array

  ' Work out the number of rows in the array, subtract one, and populate the NumberOfLines
  ' variable with this value.
  HighestArrayNum = uBound(MovementArray,2) -1

  ' Start a count that will run from zero to the highest array row number
  For Count = 0 To HighestArrayNum

    ' Populate the CurrentType variable with the contents of the Type
    ' column for this row
    CurrentType = MovementsForStockCode.CodeObject.Array(001,Count)

    ' Test to see if the Type column for this row contains the text Inv
    If CurrentType = "Inv" then
      MovementsForStockCode_OUT.CodeObject.Array(001,Count) = "<Field Background='255204000'  > </Field>"

    ' Test to see if the Type column for this row contains the text Sale
    ElseIf CurrentType = "Sale" then
      MovementsForStockCode_OUT.CodeObject.Array(001,Count) = "<Field Background='153204255'  > </Field>"

    End If
  ' Return back to the "For" statement
  Next
End Function
```

Figure 8-12: The completed code

The completed code appears in **Figure 8-12**.

Figure 8-13 shows the results when the listview's *OnPopulate* event fires and the code against the *OnPopulate* function is executed.



Date	Type	Warehouse	Period	Ref/invoice	Trn type	Quantity	Transaction value
09/09/2013	Inv	E	2011/02	000000000000303	Rec	2.000	629.85
09/09/2013	Inv	E	2011/02	000000000000303	Rec	1.000	314.92
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	2.000	699.83
20/08/2013	Inv	E	2011/02		Rec	1.000	349.91
20/08/2013	Inv	E	2011/02		Rec	4.000	1,399.66
20/08/2013	Inv	E	2011/02		Rec	5.000	1,749.57
20/08/2013	Inv	E	2011/02		Rec	6.000	2,099.49
24/06/2013	Inv	E	2011/02		Rec	1.000	349.91
31/07/2012	Sale	S	2011/02	000000000100524	Inv	5.000	1,750.00
27/02/2012	Inv	N	2011/02	00000558	Iss	5.000	1,750.00

Figure 8-13: The listview highlighting the contents of the *Type* column

Using the OnRowSelected Event

The *OnRowSelected* event is fired when the operator clicks on one of the rows containing data within a listview. When this event fires the content of the selected line is available in an array that contains just this row (i.e. where the row number is zero). The content of this row can be accessed using the array with the same name as the listview.

Using the *Movements* listview example as appears in **Figure 8-13**, if the operator clicks on the line containing the date *31/07/2012*, the values held within the individual cells of this row are available in the variables under the *MovementsForStockCode Columns* section of the *Variables* pane (see **Figure 8-14**).

Double-clicking on the *Ref/invoice(004)* variable name inserts its full name into your code (in this example the *msgbox* command has been manually inserted in front of the variable name).

```
Function MovementsForStockCode_OnRowSelected()
  msgbox MovementsForStockCode.CodeObject.Array(004,0)
End Function
```

The full name of this variable always contains a row number of zero, so can be used without any modification with the *OnRowSelected* event (as this uses a one-dimensional array). In the code snippet

above, the *msgbox* command has been added before the variable name so that the content of the variable is displayed to the operator when they click on a row (see **Figure 8-15**).

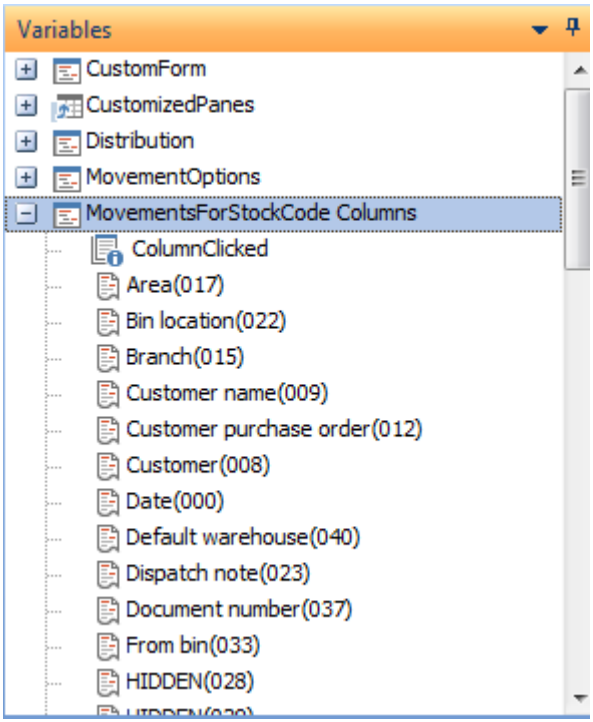


Figure 8-14: The variables available when a row is selected

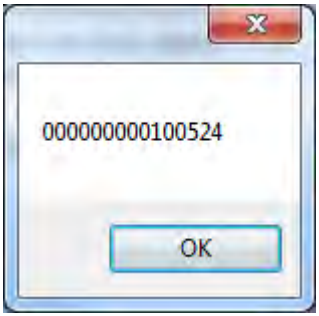


Figure 8-15: The content of the *MovementsForStockCode.CodeObject.Array(004,0)* variable

Only the contents of the first array (the one with the same name as the listview) can be accessed within the *OnRowSelected* function. You cannot modify the content or field properties of a cell by writing to the *_OUT* array. This feature is only available during the *OnPopulate* event.

The *OnRowSelected* event can also be used to perform more important tasks than just displaying the content of a cell in a message box. It can be used to call a third party application, update another database, or (as in **Figure 8-16**) populate a customized pane.

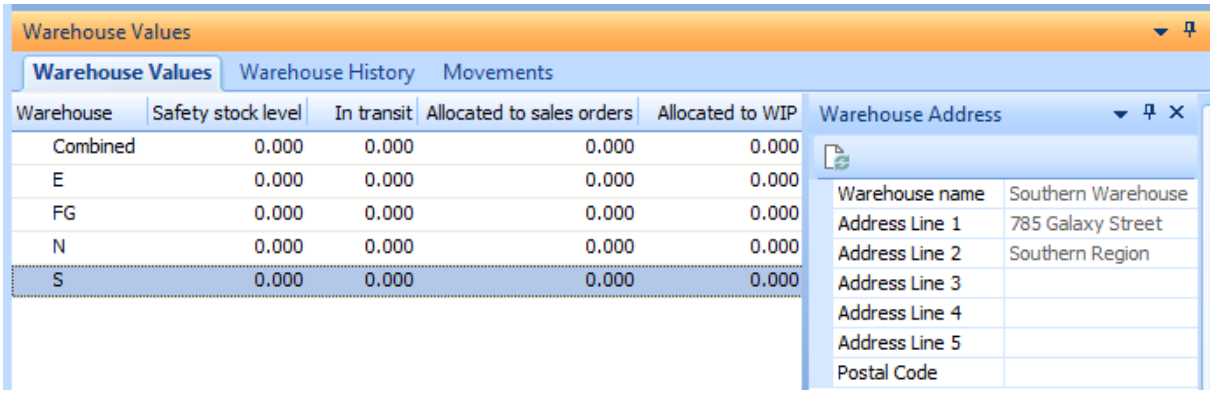


Figure 8-16: The *Warehouse Address* customized pane populated from the *Movements* listview

To populate this customized pane when the operator clicks on one of the lines in the *Warehouse Values* listview, the warehouse code must be passed through to the customized pane form, and the customized pane must be refreshed using this value.

Customized panes will be covered in more detail in Chapters 10 to 17, so the creation of the customized pane will not be covered here. However, it is important to understand what must be passed through to the customized pane from the listview, and how.

Within the VBScript editor, if there are any customized panes associated with this program, the *Variables* pane will have a *CustomizedPanes* section. When this is expanded, the names of the customized panes associated with this program will appear.

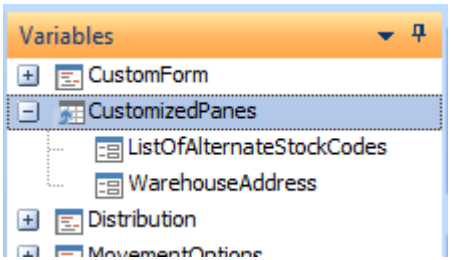


Figure 8-17: All customized panes associated with this program appear within *CustomizedPanes*

Figure 8-17 shows the two customized panes that are associated with the *Inventory Query* program in this example. *ListOfAlternateStockCodes* is an *Associated Pane* (a pre-written customized pane that is specifically associated with a SYSPRO key field). *WarehouseAddress* is a customized pane form that has been created to hold the warehouse's name and address lines. The creation of this customized pane will be covered in *Chapter 14*.

If you double-click on one of the customized pane names under the *CustomizedPanes* section the *Define Action for* screen is displayed for that customized pane (see **Figure 8-18**). This wizard enables you to specify what action should be performed, and will build the VBScript code to perform this for you. In this example the default settings are a good foundation for what is required.

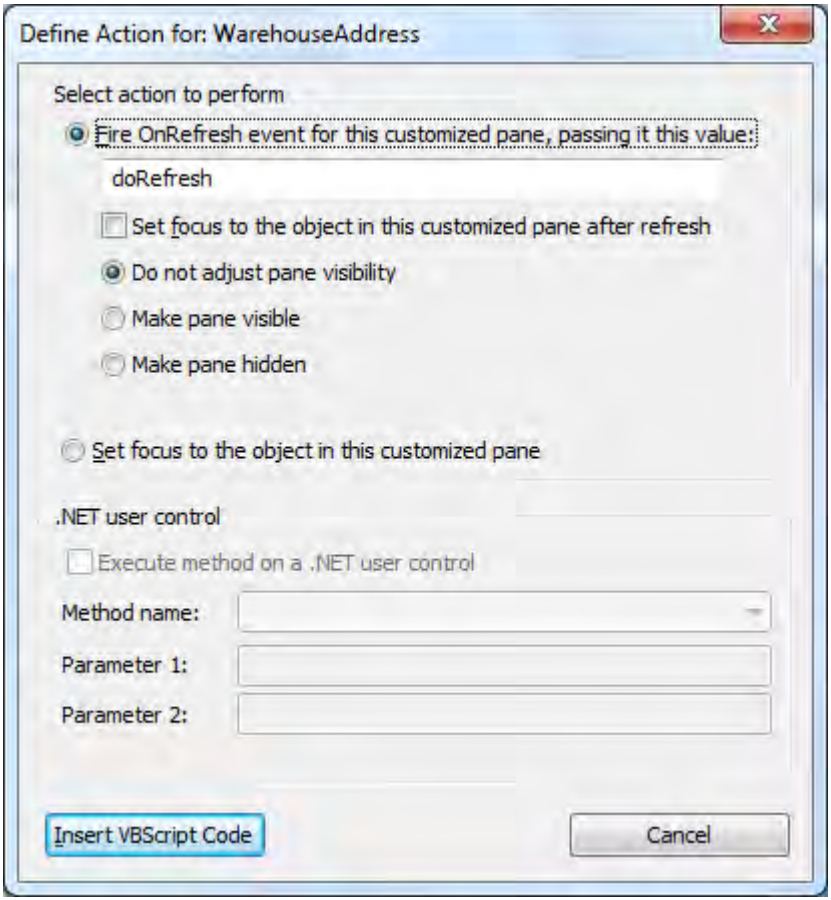


Figure 8-18: The *Define Action for* screen

Just under the *Fire OnRefresh event for this customized pane, passing it this value* radio button is the value *doRefresh*. When the customized pane's *OnRefresh* event is fired the *doRefresh* text will be passed to the customized pane.

Leaving the settings as they are, clicking on the *Insert VBScript Code* button will add the following code to refresh the customized pane.

```
CustomizedPanes.CodeObject.WarehouseAddress = "doRefresh"
```

The "*doRefresh*" text in the code above needs to be replaced with the warehouse code for the listview row that has been selected. When the row is clicked by the operator, the content of this row is added to the *WhValuesForStockCode Columns* single row array mentioned earlier. The warehouse code is in column zero of this array.

Highlighting the "*doRefresh*" text in the code and double-clicking on the *Warehouse(000)* variable within the *WhValuesForStockCode Columns* section will replace the "*doRefresh*" text with the full name of the warehouse code variable. The line of code should match the following (which has wrapped around below, but is one line of code).

```
CustomizedPanes.CodeObject.WarehouseAddress =
WhValuesForStockCode.CodeObject.Array(000,0)
```

When this line of code is executed the *Warehouse Address* customized pane will be refreshed, and passed the content of the *Warehouse* column for the currently selected row.

When editing the VBScript for a customized pane, the *Variables* pane contains a *CustomizedPane* section (as opposed to the *CustomizedPanes* section mentioned above). This section contains all the available variables for the type of customized pane that you are currently editing, in this case a *Form* (see **Figure 8-19**).

Any customized pane can be refreshed by another form, listview, or customized pane within the same program. When this happens the *RefreshValue* variable against the customized pane contains whatever value was passed by the other form, listview, or customized pane (the *doRefresh* text in **Figure 8-18**, or the warehouse code in the code above).

The content of the *RefreshValue* variable can be extracted and used in the VBScript, just like any other variable, as can be seen in the sample below. In this customized pane example the *RefreshValue* (which contains the warehouse code) is used to populate the *WH* variable. Later in the customized pane the *WH* variable will be passed to a business object that looks up the warehouse master information.

```
Dim WH
WH = CustomizedPane.CodeObject.RefreshValue
```

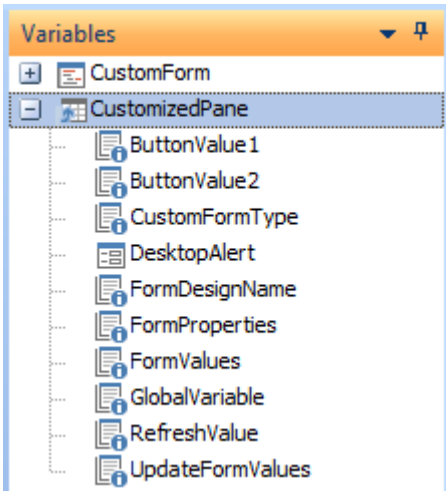


Figure 8-19: The available customized pane variables for a *Form*

Using the OnDblClick Event

The *OnDblClick* event is fired when an operator double-clicks on a row in a listview. It functions in exactly the same way as the *OnRowSelected* event.

If both the *OnRowSelected* and *OnDblClick* events are configured for a listview, only the *OnRowSelected* event will fire if the operator attempts to double-click on a row that has not already been selected. This is because the first click causes the *OnRowSelected* event to fire, so the *OnDblClick* event is ignored. However, if the operator double-clicks on a row that has already been selected, the *OnRowSelected* will not fire again, and the *OnDblClick* event will fire.

Interacting with XAML

This section is not intended to teach you XAML, merely explain how you can make use of the XAML themes provided with SYSPRO. XAML was briefly mentioned within the *XAML Code* field property section above, which explained how to use XAML to highlight specific cells.

There is a means of applying a XAML theme to a column by right-clicking on the column header and selecting a XAML theme. This applies to every cell in this column; not to a specific cell. **Figure 8-20** shows where the *Date* column header has been right-clicked and the list of XAML themes selected from the dropdown list. This works for columns of all data types (date, numeric, and alphanumeric).

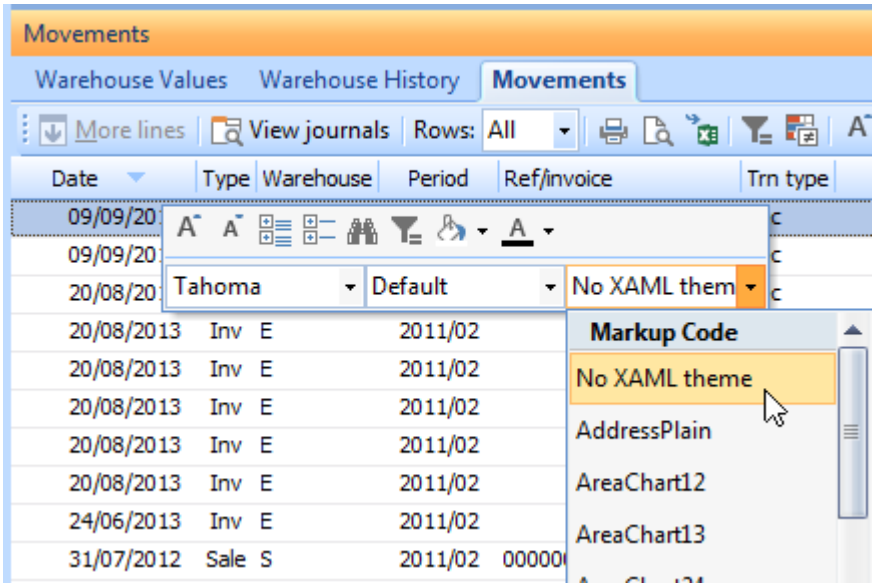


Figure 8-20: Applying a *XAML theme* to a listview column

The *AddressPlain* XAML theme can be seen in **Figure 8-20**. This XAML theme contains seven placeholders, so can contain up to seven pieces of information. This means that you can display multiple pieces of information in one cell (as can be seen in **Figure 8-21**).

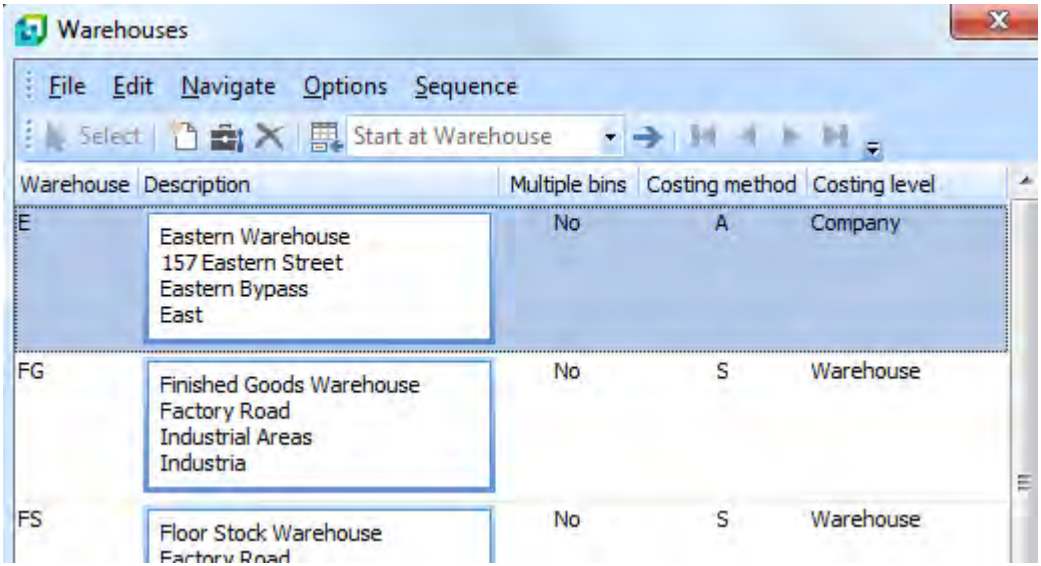


Figure 8-21: Example of using XAML to display multiple pieces of information in one cell

The XAML themes are stored in SYSPRO's `Base\Samples` folder, with the naming convention Form_*ThemeName*.xaml, where *ThemeName* is replaced with the name of the theme. The XAML files are text files, so you can look at the source, and even copy them to a new name and modify the copy. Do not modify the standard XAML file as it may be overwritten when SYSPRO is updated. If you create your own XAML theme and follow the SYSPRO naming convention, after logging out of SYSPRO, the next time that you browse on the XAML themes your custom XAML theme will appear in the list.

Placeholders are numbers surrounded by percentage signs. At the time that the data is displayed the placeholders are replaced with the data. Below is the section from the *AddressPlain* XAML theme that contains the placeholders. In the *AddressPlain* theme there are seven placeholders, so up to seven pieces of data can be displayed on separate lines within the cell that has this theme applied to it.

```
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%1%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%2%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%3%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%4%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%5%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%6%"/></TextBlock>
<TextBlock TextWrapping='Wrap' Foreground='Black'><Run Text="%7%"/></TextBlock>
```

These placeholders have been configured to appear of separate lines, but it is possible to have several placeholders appear alongside each other on the same line.

If there are specific requirements for using a XAML theme, these appear as comment lines at the top of the XAML file. For example, the XAML theme files that display graphs specify how many entries are required, and the minimum/maximum values (the scale) that can be used.

For those XAML themes that do not have specific requirements, if you do not supply a value, the value that would have normally appeared in that cell will be passed through to the XAML theme as variable one, and will be used with placeholder *%1%*.

When multiple pieces of data are required to be displayed in a cell, they are passed to the cell in the *_OUT* array, delimited by pipe signs. The first piece of data replaces placeholder *%1%*, the second replaces placeholder *%2%*, the third replaces placeholder *%3%*, etc.

The following line of code (which has wrapped around on this page) supplies three pieces of data to the *Inventory Warehouses* listview's *_OUT* array within the *Warehouses* browse. The three pieces of data are *AAA*, *BBB*, and *CCC*. These values are delimited with pipe signs.

```
InventoryWarehouses_OUT.CodeObject.Array(001,Count) = "<Field Value='AAA|BBB|CCC' >
</Field>"
```

When used in the *Warehouses* listview's *OnPopulate* function it causes the *Description* to contain these values (see **Figure 8-22**). Note that these appear vertically because that is how the placeholders were defined in the XAML theme.

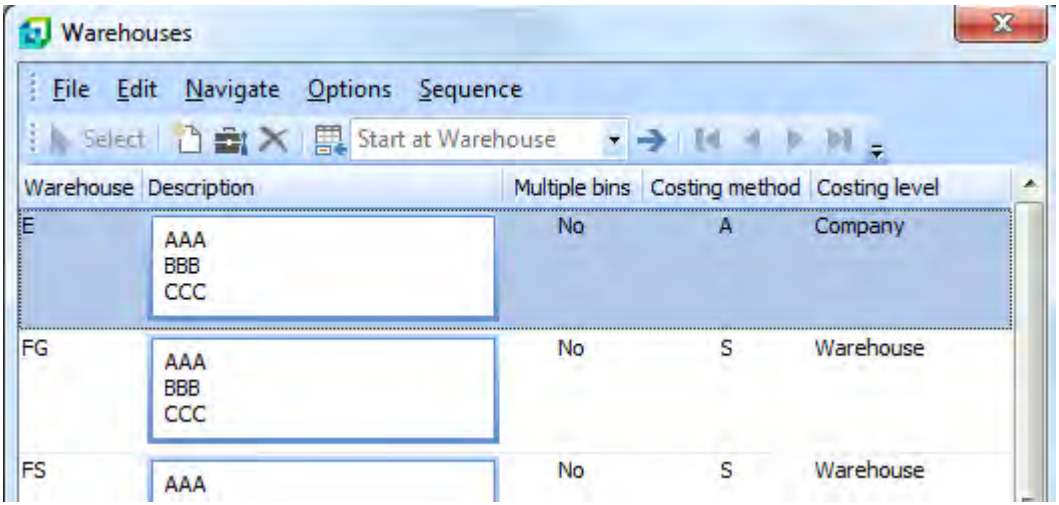


Figure 8-22: Supplying multiple pieces of data to a single cell using a XAML theme

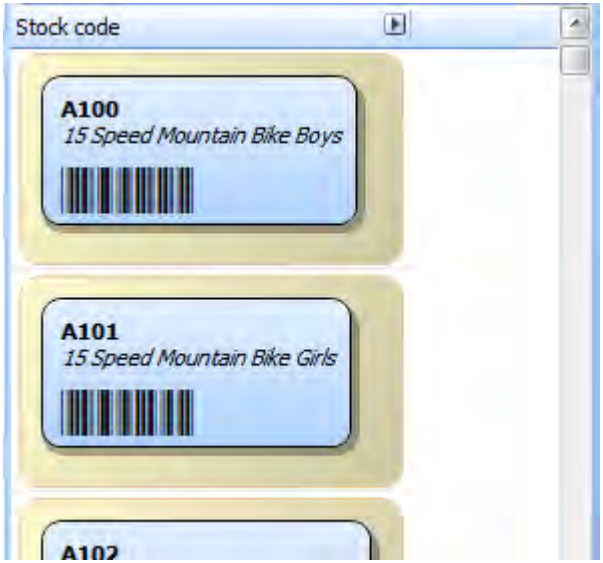


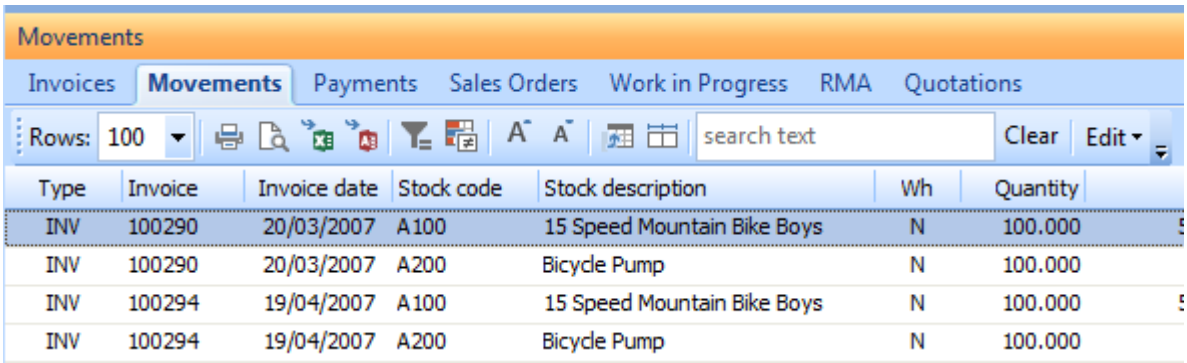
Figure 8-23: Using the *Label3of9* XAML theme to display barcodes

Another example of supplying multiple pieces of data appears in **Figure 8-23**. The *Label3of9* XAML theme has three placeholders. Placeholder *%1%* displays its value in bold, placeholder *%2%* displays its value in italic, and placeholder *%3%* displays its value as a *Code 3 of 9* barcode. In **Figure 8-23**, placeholders *%1%* and *%3%* both contain the stock code, and placeholder *%2%* contain the stock description.

Adding Custom Form Fields to a Listview

If you have a standard listview that has columns that contain key fields, it is an easy task to add custom form fields related to this key field. Note that this functionality is only available if the SYSPRO company is configured to use SQL Server for data storage, and the option to store the custom form field information separately for each custom form has been selected. This functionality is not available for *Data Grids*, which are listviews that are designed for data entry, and is covered in more detail in a later section.

Figure 8-24 shows the *Movements* tab from the *Customer Query* program, showing some of the columns that are available as standard.



Type	Invoice	Invoice date	Stock code	Stock description	Wh	Quantity
INV	100290	20/03/2007	A100	15 Speed Mountain Bike Boys	N	100.000
INV	100290	20/03/2007	A200	Bicycle Pump	N	100.000
INV	100294	19/04/2007	A100	15 Speed Mountain Bike Boys	N	100.000
INV	100294	19/04/2007	A200	Bicycle Pump	N	100.000

Figure 8-24: The *Movements* tab from the *Customer Query* program

Custom form fields that are associated with any of the key fields that appear within this listview can be added as columns. The *Movements* listview contains stock codes, so any custom form field associated with stock codes can be added as a listview column.

Figure 8-25 shows the same *Movements* listview after the *Style* and *Size* custom form fields have been added as columns. These are two of the custom form fields that are associated with stock codes for this company.

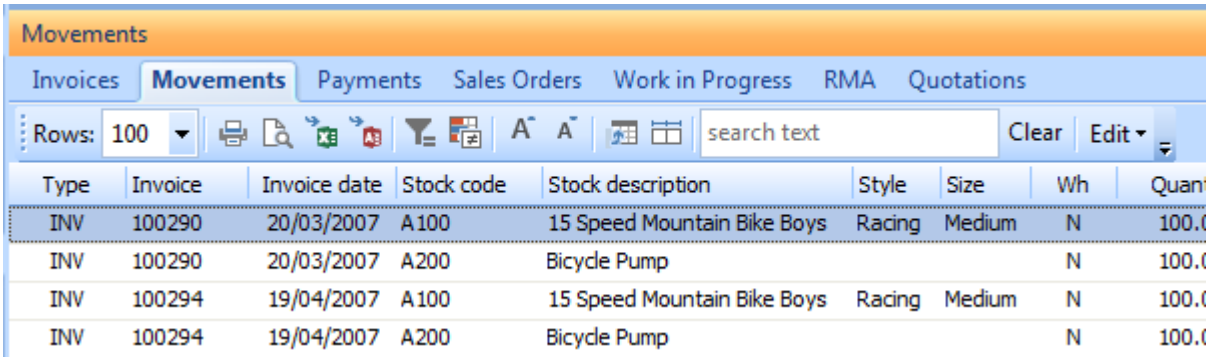


Figure 8-25: The Movements listview with the custom form fields Style and Size added as columns

The custom form fields are added by right-clicking on one of the column headers and selecting *Customize* | *Add Custom Column* from the displayed menu (see **Figure 8-26**). The *Custom Columns for Listviews* pane is displayed.

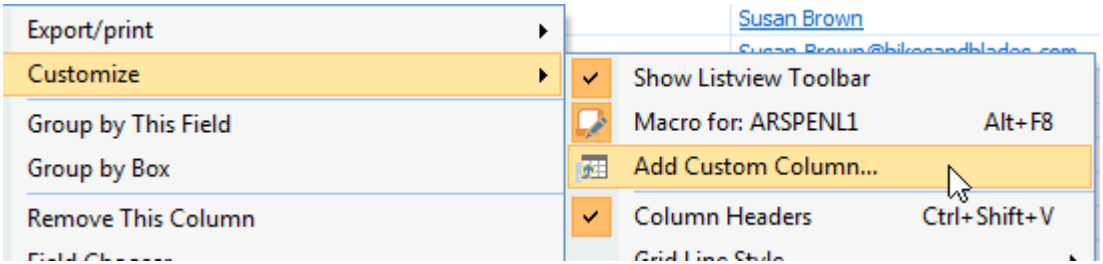


Figure 8-26: The *Add Custom Column* option

Figure 8-27 shows the *Custom Columns for Listviews* screen which contains the options to add a field from a custom form, add a blank column, or add a field from a related master table. In this case the *Custom form column* option has been selected. When this is selected the dropdown list against the *Custom form type* prompt contains the custom forms associated with the columns in this listview. The *stock code*, *sales order*, *customer invoice*, and *customer branch* custom forms are associated with columns in this listview and appear in the dropdown list. The *stock code* option has been selected.

The list of column names that can be matched appears in the `IMPQVW.IMP` file in the SYSPRO program folder. Columns that begin with a semi-colon in this file are comment lines. The matching will be performed against the field names that start in column 1 of this file.

Where the same field name is used in two completely different areas of SYSPRO, and each can have their own custom forms (such as *Branch* appears in both *Accounts Payable* and *Accounts Receivable*)

the field is re-labelled to include the area. In **Figure 8-27** it can be seen that *invoice* has been re-labelled as *customer invoice*, and *branch* as *customer branch*.

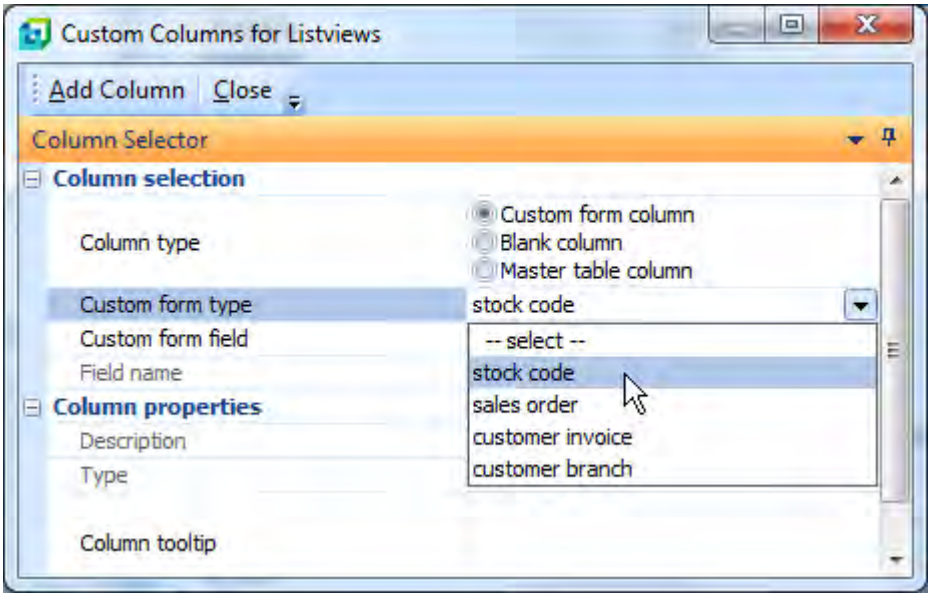


Figure 8-27: Selecting the *Custom form type* on the *Custom Columns for Listviews* screen

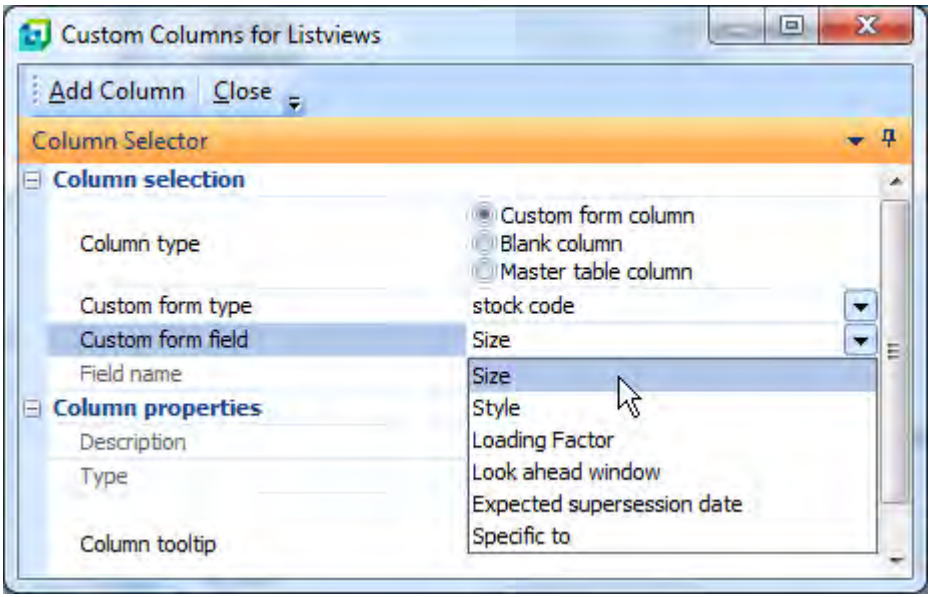


Figure 8-28: Selecting the *Custom form field* from the dropdown list

When the *Custom form type* is selected, the dropdown list alongside the *Custom form field* prompt is populated with all the fields for this custom form (see **Figure 8-28**). When one of the items in the *Custom form field* dropdown list is selected, the *Field name*, *Description* and *Type* fields are automatically populated using the custom form field's information. A tooltip can be added against the *Column tooltip* prompt, if required.

Clicking on the *Add Column* button on the toolbar will add this column to the listview, but leave you in the *Custom Columns for Listviews* screen so that you can add more columns if required. Clicking on the *Close* button on the toolbar will close the *Custom Columns for Listviews* screen and return you to the listview.

The listview will contain the added columns (see **Figure 8-25**), and these can be repositioned as required. By default the column is always added to the beginning of the listview, and in this case the *Style* and *Size* columns have been dragged to a more appropriate position.

Columns can be removed from the listview by dragging their column header from the column header section. This column name will appear in the *Field Chooser* (right-click on the *Column Header* | *Field Chooser*) and can be dragged back to the column header if this column is required again. **Figure 8-29** shows the *Field Chooser* after this column has been dragged from the listview.

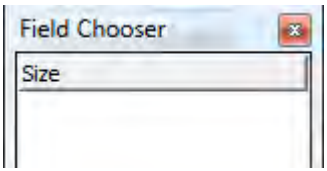


Figure 8-29: The *Field Chooser* showing the custom form field that was dragged from the listview

Adding Blank Columns to a Listview and Populating Them

A blank column can be added to a listview in a similar way to custom form fields. However, where the custom form fields automatically populate with the relevant values, VBScript code must be added to the blank column before anything will appear. Another difference is that you do not need to be using SQL Server for your data storage to be able to add blank columns and populate them. This functionality is not available for *Data Grids*, which are listviews that are designed for data entry.

Once blank columns have been added to a listview, two extra sections appear within the *Variables* pane within the VBScripting environment. This example uses the *Transactions* listview within the *Bank Query* program. **Figure 8-30** shows the *Variables* pane prior to a blank column being added. There is one section containing the variables for the columns as they would be displayed if no changes are made (*CashbookTransactions Columns*) and one where you post any changes that are required before the listview is populated (*CashbookTransactions_OUT Columns*).

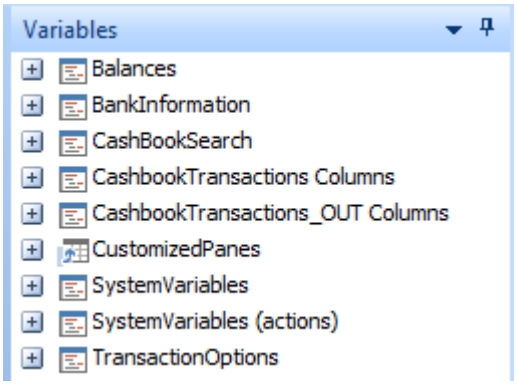


Figure 8-30: The *Variables* pane prior to adding a blank column

Once a blank column has been added, the two additional sections appear. One contains the variables for the blank columns as they would be displayed if no changes are made (*CashbookTransactions_Custom Columns*) and one where you post any changes that are required before this listview is populated (*CashbookTransactions_OUT_Custom Columns*). These can be seen in **Figure 8-31**.

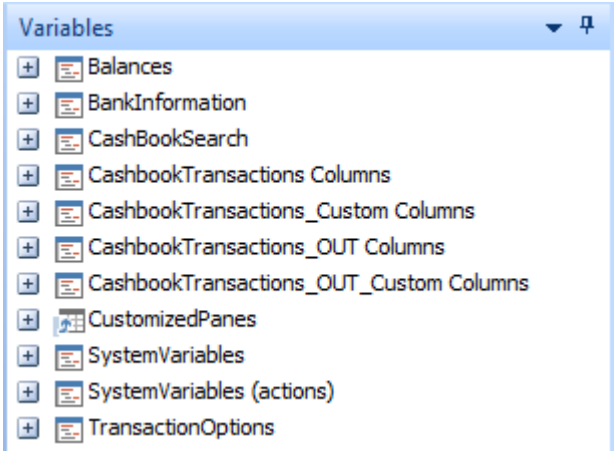


Figure 8-31: The extra sections that become available when blank columns are added to a listview

A blank column is added to a listview by right-clicking on one of its existing column headers and selecting *Customize* | *Add Custom Column* from the displayed menu. The *Custom Columns for Listview* screen is displayed. The first option on the screen is *Column type* where you select to add a *Custom form column*, or a *Blank column*.

Figure 8-32 shows the *Custom Columns for Listviews* screen where the *Blank column* radio button has been selected. The column *Description* is *Proof of Approval*, the *Type* is set to *Alpha*, and a tooltip is provided. When the *Add Column* button is clicked on the toolbar this column is added to the listview, but the *Custom Columns for Listviews* screen remains open. Once all the required blank columns have been added, the *Close* button is used to close the screen and you will be taken back to the listview.

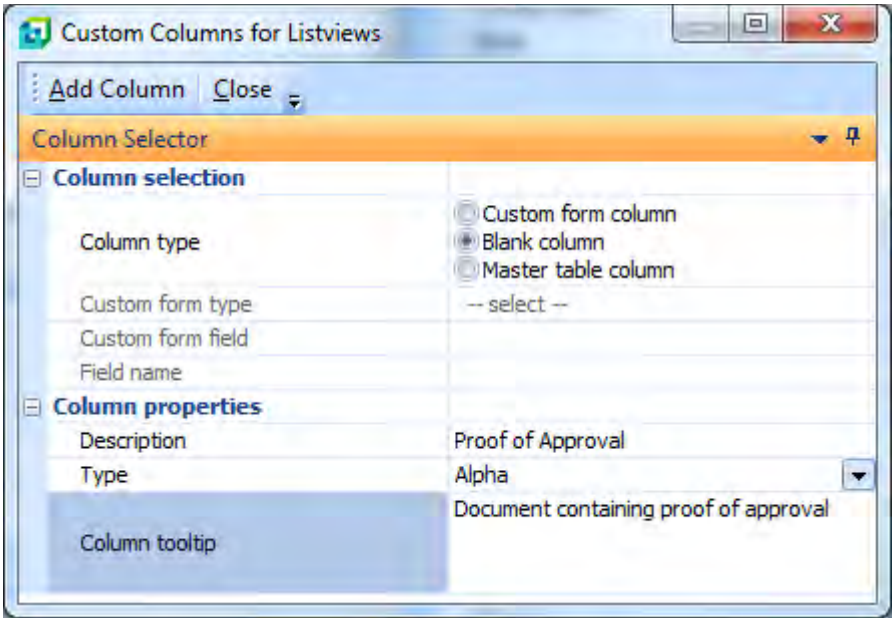


Figure 8-32: Adding a blank column called *Proof of Approval*

Each blank column that is added is inserted at the beginning of the listview. **Figure 8-33** shows where two blank columns have been added to the listview, the first was *Proof of Approval* and the second was *Cashier's initials*. These can be moved to any location within the listview.

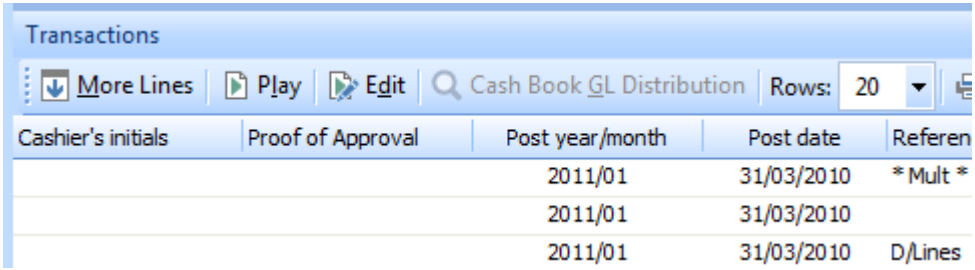


Figure 8-33: Two blank columns added to a listview

The blank columns can be populated using the listview's *OnPopulate* event using the variables within the *CashbookTransactions_OUT_Custom Columns* section that was mentioned above. **Figure 8-34** shows the two variables within the *CashbookTransactions_OUT_Custom Columns* section. The number within braces immediately following the variable name is the column number of the blank column. This matches up with the sequence in which they were added to the listview. If these blank columns are moved to another location within the listview this column number will not be affected.

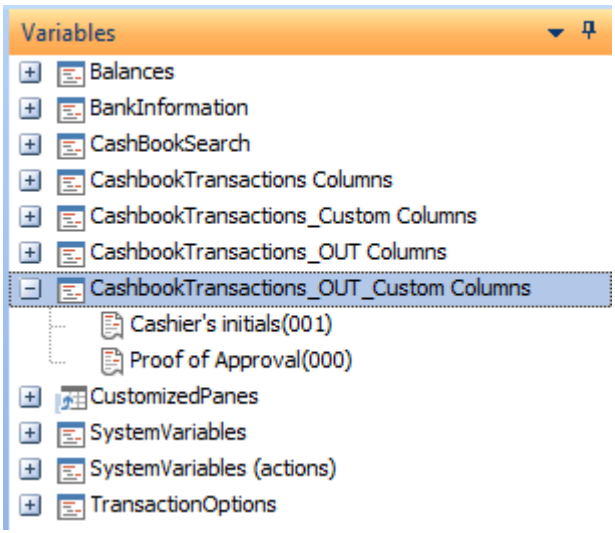


Figure 8-34: The blank columns available for populating

The cell of a blank column can be populated by double-clicking on its column name, which will add the full name of this variable to your code. Using the *Proof of Approval* column as an example the code that is added appears below. The 000 before the comma refers to the column number, and the 0 after the comma refers to the row number. As with all arrays, the first row is 0 and the first column is 0.

```
CashbookTransactions_OUT_Custom.CodeObject.Array(000,0)
```

You could populate the first row of the *Proof of Approval* column using either of the following two statements (note that the second of the two statements is one line, but has wrapped around on this page):

```
CashbookTransactions_OUT_Custom.CodeObject.Array(000,0) = "ABC123"

CashbookTransactions_OUT_Custom.CodeObject.Array(000,0) = "<Field Value='XYZ789'>
</Field>"
```

Although it is possible to use one of these to prove that the cell is populated, unless there is at least one row being displayed in the listview it will return an error message, as it will not have a cell to

populate. If you have not already done so, please read the sections above called *Arrays*, and *Interacting with Listviews*, which explain how to populate the listview.

If the operator drags a blank column from the listview, it becomes available within the *Field Chooser*. It can be dragged back to the listview at any time. If the blank column is no longer required it can be permanently removed using the *Remove This Column* option that is available when you right-click on one of the column headers (see **Figure 8-35**).

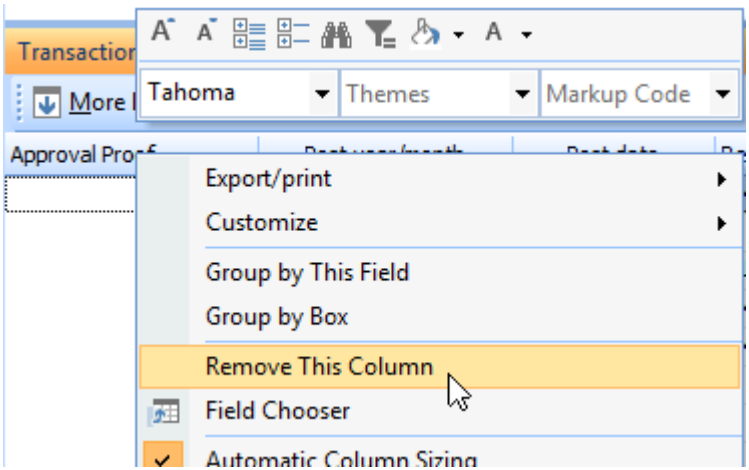


Figure 8-35: Permanently removing the blank column from a listview

Note that removing a blank column this way removes its column number from the *CashbookTransactions_OUT Columns* and *CashbookTransactions_OUT_Custom Columns* sections of the *Variables* pane. If a blank column was added after the one that is to be removed, the later one's column number will change. Any code to populate this column will need to be revisited.

Adding Master Table Fields to a Listview

Custom columns can be added to a listview that contain fields from master tables that are already used in this listview. Unlike adding custom form fields to a listview, this functionality does not require you to be using SQL Server for your data storage. This functionality is not available for *Data Grids* (which are listviews that are designed for data entry).

For example, the *Movements* listview from the *Customer Query* program appears in **Figure 8-36**. This allows you to add columns from the *Sales Analysis Branch* table, *Accounts Receivable Customer Invoice* table, *Sales Order Master* table, and the *Inventory Master* table.

Master field columns are added by right-clicking on the column header and selecting *Customize*, *Add Custom Columns* from the displayed menu. Against the *Column type* prompt select the *Master table column* radio button and the *Table Fields* screen is displayed (see **Figure 8-37**).

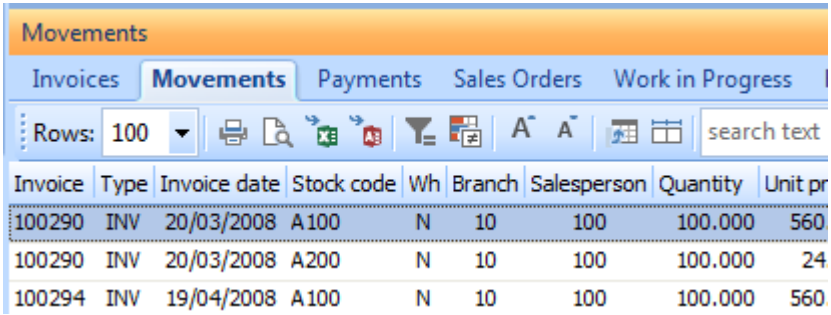


Figure 8-36: The *Movements* listview of the *Customer Query* program

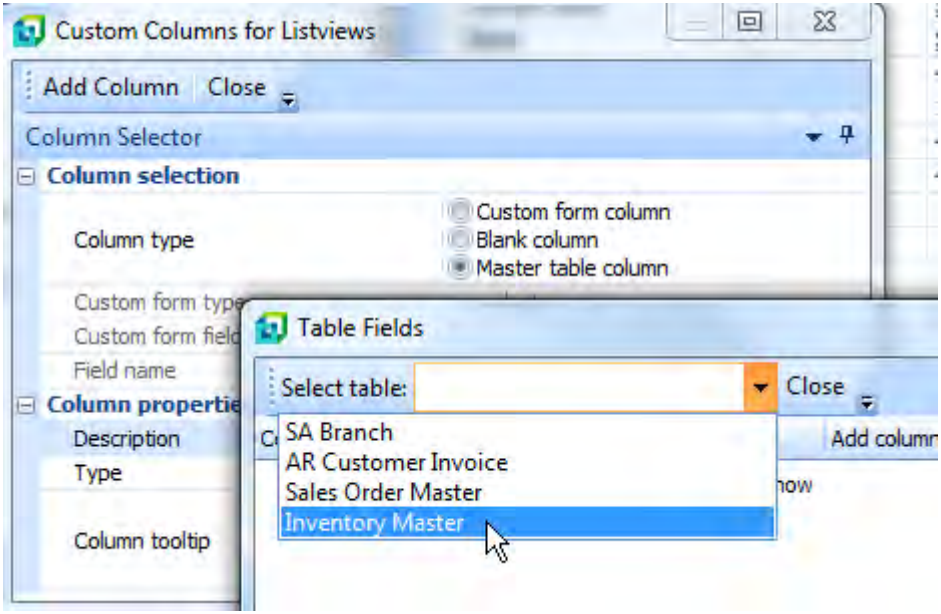


Figure 8-37: Selecting *Master table column* radio button, then selecting the required table

The *Select table* dropdown list will contain a list of tables that are used in the listview. Select the required table name and the *Table Fields* screen will be populated with the fields from the selected table (see **Figure 8-38**).

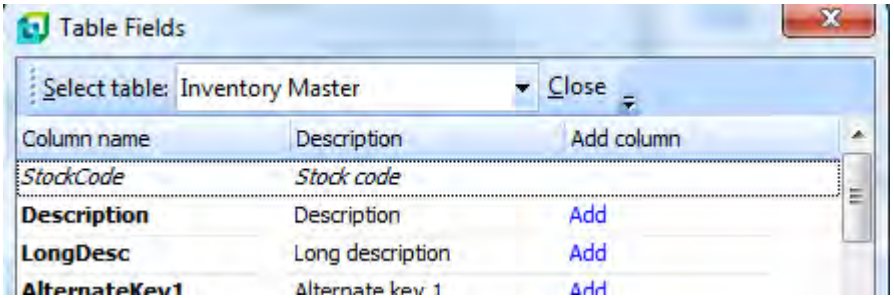


Figure 8-38: The fields from the *Inventory Master* table

The *Table Fields* screen contains three columns, *Column name*, *Description* and *Add column*. If the field already exists in the *Movements* listview the content of both the *Column name* and *Description* columns will appear in italics, and the *Add column* column will be blank. If the field does not already exist in the *Movements* listview the text will not appear in italics, and the *Add column* column will contain the hyperlinked text *Add*. Clicking on the *Add* hyperlink will add this column to the beginning on the *Movements* listview. The column will not contain any values until it is refreshed (usually by selecting another customer). **Figure 8-39** shows the same listview as in **Figure 8-36**, but after the *Description* column has been added. Note that it has not yet been populated.

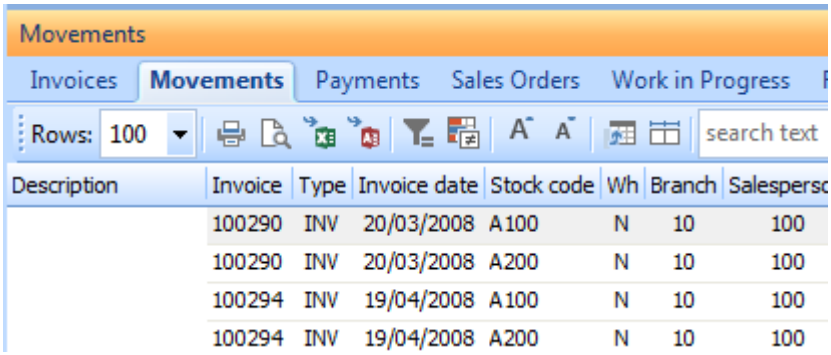


Figure 8-39: The *Description* column added to the *Movements* listview

Figure 8-40 shows the same listview after the *Description* column has been moved to a more logical place (after the *Stock code* column) and the data has been refreshed.

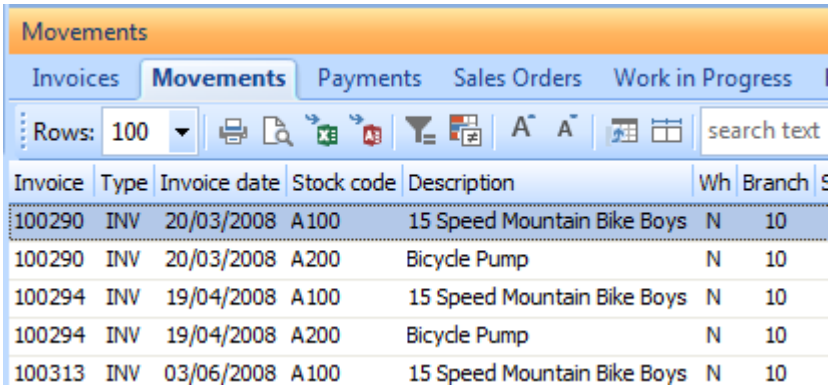


Figure 8-40: The *Description* column moved to its new location and the listview values refreshed

Variables for this column appear in both the *Movements columns* and *Movements_OUT columns* sections of the *Variables* pane within the VBScripting environment. This means that the stock description can be retrieved for a particular row, as well as changes made to it or its properties when the listview is populated.

Adding Master Table Fields that use a Compound Key

Where the column in the listview contains the key field for a table, SYSPRO will detect this automatically and add the table name to the dropdown list when you select the *Master table column* radio button (see **Figure 8-37**).

However, the keys of many SYSPRO tables contain more than one key field, called compound keys, and these will not be detected automatically. To be able to access a table with a compound key, all of the components of the key must exist as columns in the listview (they do not need to be displayed in the listview, they just need to be available in the *Field chooser* if they are not being displayed). A file called `IMPQVW.IMP` (that resides in your SYSPRO `Programs` folder) contains entries to build the compound keys for some tables, such as the *Multiple ship to address* table.

If you find that the table with the compound key that you want to use is not in the `IMPQVW.IMP` file, and all the parts of the key are contained in the listview, you can build your own equivalent of `IMPQVW.IMP`. The file must be called `CUSQVW.IMP`, and it needs to be located in your custom program folder. The location of your custom program folder can be found/set in your `IMPACT.INI` file on the SYSPRO application server. It should exist under the *[Customer Directories]* section and will start with *CUSPRG=*. The following is a sample of this entry:

```
[Custom Directories]
CUSPRG=C:\TST700\CUSTOM
```

When you use the dropdown list against the *Select table* prompt (when adding a field for a master table) the list of tables contains the master tables that SYSPRO has detected along with the relevant ones from the `CUSQVW.IMP` file.

The following is a working example of creating a compound key entry using the *Movements* listview in the *Inventory Query* program. The requirement is to display the salesperson's name against sales movements. The default view of the *Movements* listview does not contain the salesperson code, so the first step is to drag this onto the listview from the *Field chooser* (see **Figure 8-41**).

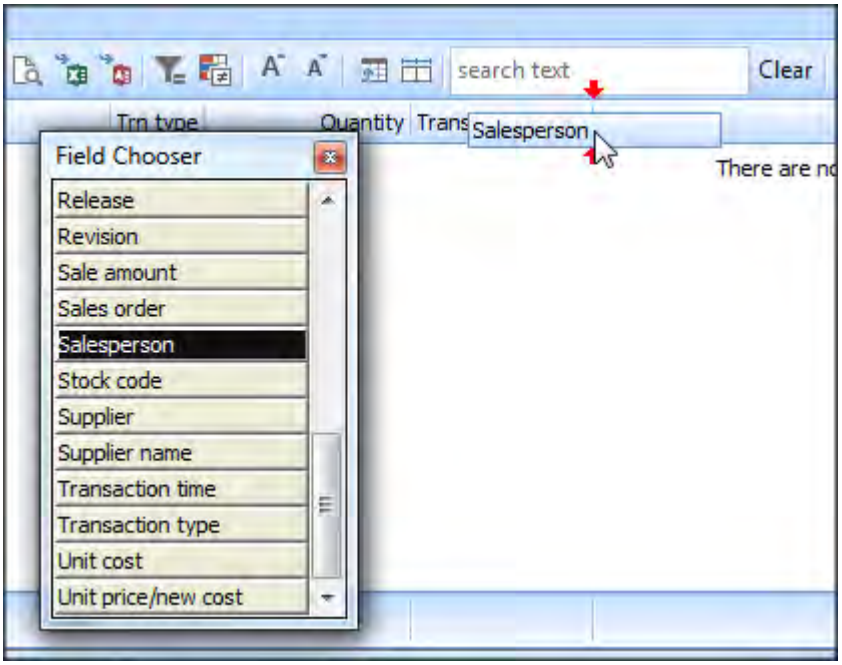


Figure 8-41: Adding the *Salesperson* column to the *Movements* listview

The salesperson's name is not available in any of the master tables that are linked to the *Movements* listview. The list of available master tables can be seen in **Figure 8-42**.

The *Sales Analysis Salesperson* table contains the salesperson's name. This table does not appear in the list in **Figure 8-42** because it has a compound key. This key consists of the *Branch* code and the *Salesperson* code. The *Salesperson* table does not exist in the *[Compound Keys]* section of the `IMPQVW.IMP` file, so a custom file needs to be built.

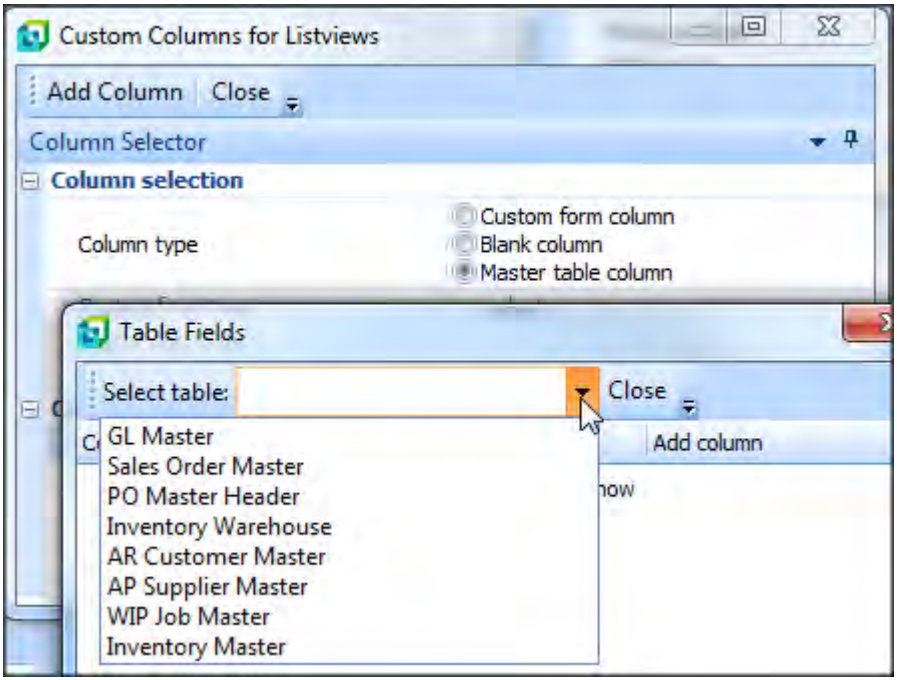


Figure 8-42: The available master tables linked to the *Movements* listview

The location of the custom programs folder can be found in the `IMPACT.INI` file. In this case the folder already existed so it did not need to be created. However, there was no `CUSQVW.IMP` file in this folder so one had to be created. This was done by copying the `IMPQVW.IMP` file from the SYSPRO *Program* folder to the custom program folder, and renaming it. The file was edited using a text editor (Notepad in this case) to remove the entries above the section starting with *List view Custom Column definitions using compound keys*.

Below is an extract from the `IMPQVW.IMP` file. The line containing the *BomStructure* entry has been truncated, otherwise it would have wrapped around. The section that would have appeared immediately above this contains information about the layout of the file, so has been omitted here.

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

ArMultAddress       ARSSHP Customer|Address code
BomStructure        BOMMAT Parent stock code|Parent Revision|Parent Release|Route

; End of IMPQVW.IMP
```

In the `CUSQVW.IMP` file these two entries must be removed, because they already exist in the `IMPQVW.IMP` file, and the entry for the *Salesperson* table must be added. The contents of this file is positional, which means that the *SQL Table name* must start as the first character of the row, the information for the *Custom* column must start as the 21st character of the row, and the first column names must start as the 30th character.

The simplest way of doing this is to remove the *BomStructure* entry, leaving the *ArMultAddress* entry. Then modify the *ArMultAddress* entry to contain the information for the *Salesperson* table. In the sample below the *BomStructure* entry has been removed (and the *End of IMPQVW.IMP* text has been changed *to End of CUSQVW.IMP*):

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

ArMultAddress       ARSSHP Customer|Address code

; End of CUSQVW.IMP
```

The name of the *Salesperson* table is *SalSalesperson*. The table name can be looked up using the *Data Dictionary Viewer* utility (see **Figure 8-43**).

The *Data Dictionary Viewer* can be loaded using *Ctrl+R* from the main menu, entering the program name ***DDSBFI*** and clicking on the *OK* button.

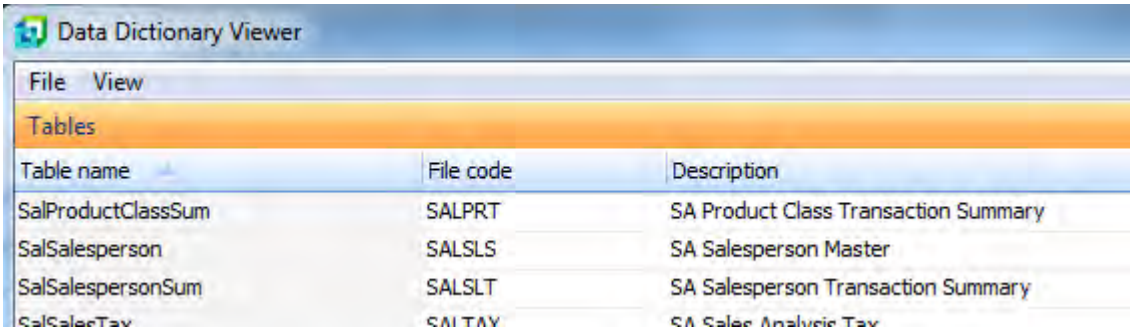


Figure 8-43: The *SalSalesperson* table name in ***DDSBFI***

Now that the *SQL Table name* is known, this can be entered into the `CUSQVW.IMP` file:

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      ARSSHP Customer|Address code

; End of CUSQVW.IMP
```

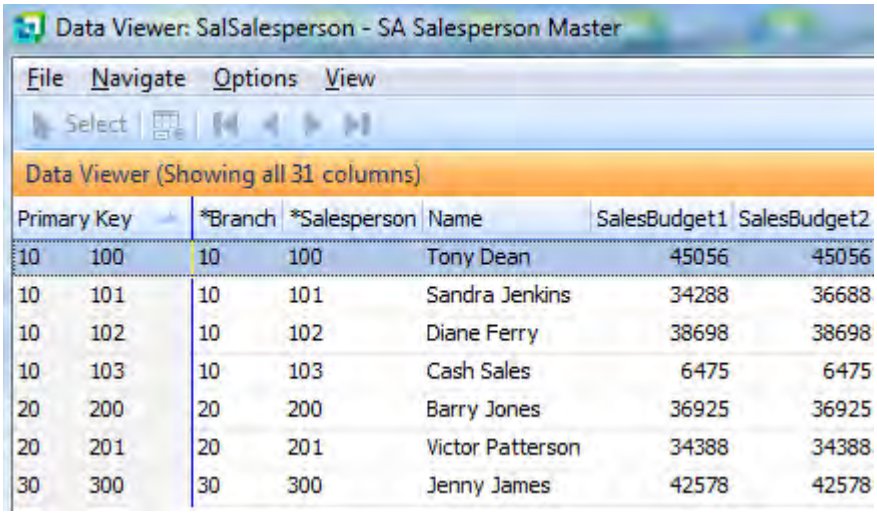


Figure 8-44: The structure of the *SalSalesperson* table

Double-clicking on the *SalSalesperson* entry in ***DDSBFI*** allows you to see the structure of the table and the content. **Figure 8-44** shows the *SalSalesperson* table. The column to the left of the vertical line is the *Primary key*. The columns to the right show the columns that appear in the table. Those marked with an asterisk are part of the key, in this case *Branch* and *Salesperson*.

This information can be added to your `CUSQVW.IMP` file (not that the column names are separated by pipe signs):

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      ARSSHP Branch|Salesperson

; End of CUSQVW.IMP
```

The last piece of information that is required is the name of the custom form for this table. This can be found in the `IMPCFM.IMP` file in the SYSPRO *Program* folder. The following is an extract from this file showing the relevant section. The *Salesperson* entry is the middle one, so the custom form entry required is *SLS*.

```
SORRD  Sales Order Route Detail              SorRouteDetail     -
SLS    Salesperson                           SalSalesperson     -
SERIAL Serial                                InvSerialHead      -
```

This information can be added to the `CUSQVW.IMP` file, as below:

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      SLS    Branch|Salesperson

; End of CUSQVW.IMP
```

The `CUSQVW.IMP` file is now complete, and you need to check that you have built this correctly. First, login to SYSPRO from scratch so that the new `CUSQVW.IMP` file is read in by SYSPRO. Call up the *Inventory Query*, right-click on one of the column headers in the *Movements* listview, select *Customize*, and then *Add Custom Columns*. Select the *Master table column* radio button and the *Table Fields* screen will be displayed. The dropdown list alongside the *Select table* prompt should contain the *SA Salesperson Master* table name (see **Figure 8-45**).

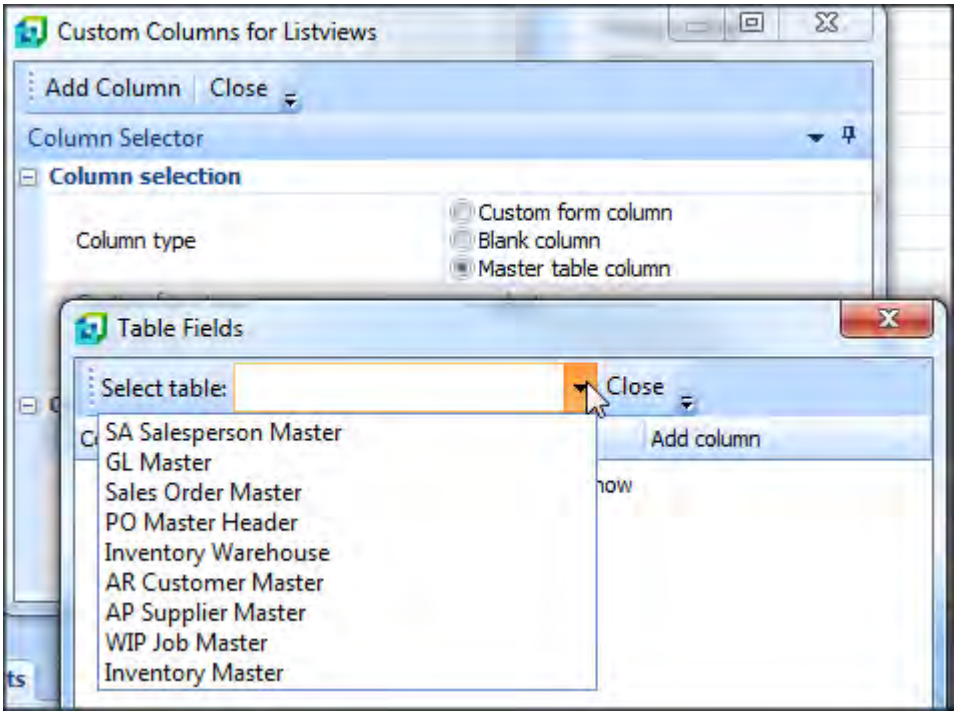


Figure 8-45: The *SA Salesperson Master* table name available in the dropdown list

Selecting this table name displays the fields from this table in the listview (see **Figure 8-46**). The fields from this table that are already in the *Movements* listview will appear italicized.

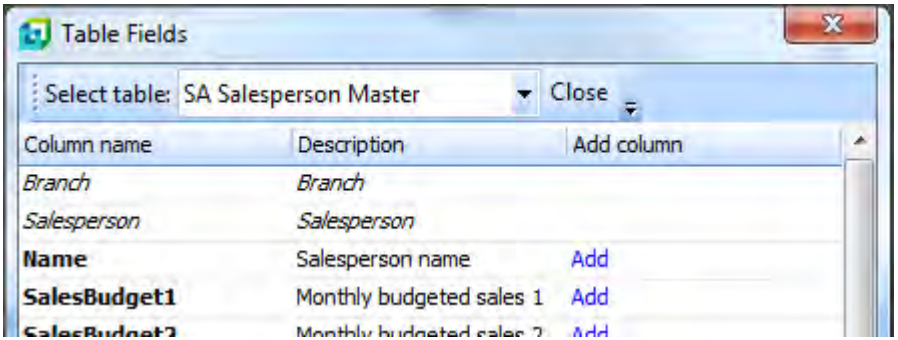


Figure 8-46: The fields from the *Salesperson* table

Select the *Add* hyperlink against the *Name* entry and the *Salesperson name* column is added to the *Movements* listview. This is always added as the first column of the listview (see **Figure 8-47**).

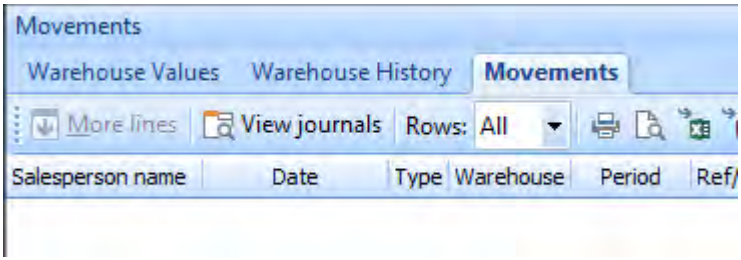


Figure 8-47: The *Salesperson name* column added to the *Movements* listview

This column header can be dragged to the required location, and the column will be populated with data the next time the listview is used (see **Figure 8-48**).

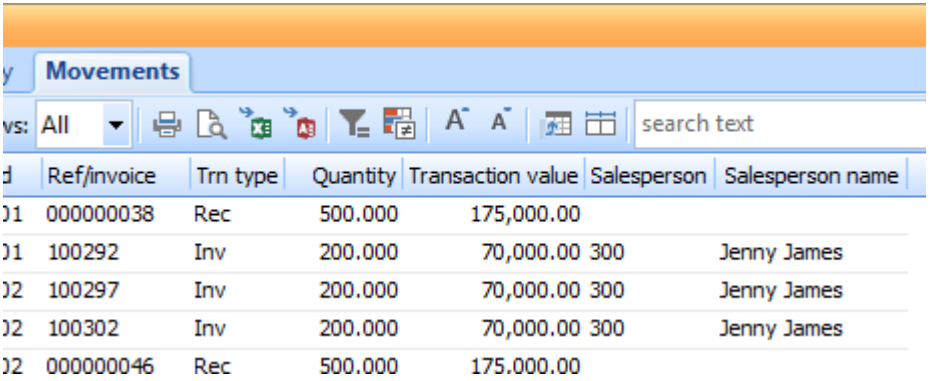


Figure 8-48: The *Salesperson name* column populated with data

Listview Restrictions

Custom columns (custom form fields, blank columns, and master table fields) can only be added to standard listviews. A maximum of 30 columns can be added to standard listviews. Listviews can have a maximum of 100 columns, so if the standard listview has 75 columns you can only add another 25 to this listview.

Both custom form fields and master table fields added to standard listview use a VBScript to retrieve the values to be used. This VBScript calls the ***COMCOL*** business object to access the data.

The operator will not be able to use the *Add Custom Columns* option if:

- they are a member of a role and are not in design mode
- the operator is denied access by either of these two security options being unchecked:
  Form – Customization by operator
  Form – Caption adjustments for group/company

Data Grids

Data grids are similar to listviews in appearance as they have rows, columns and cells. But whereas listviews are used to display data, data grids are used for the quick entry and maintenance of data. As detail lines are added, only the minimum of data validation is performed. On completion of the data entry, a *Save* or *Submit* button is selected and validation is performed on all of the data. **Figure 8-49** shows a data grid being used for entry of a sales order in the *Sales Order Entry Express* program.

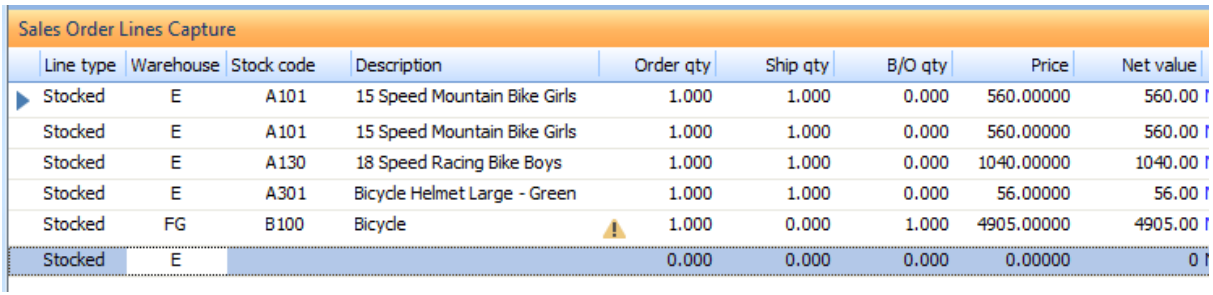


Figure 8-49: Entering sales order data using a *Data Grid*

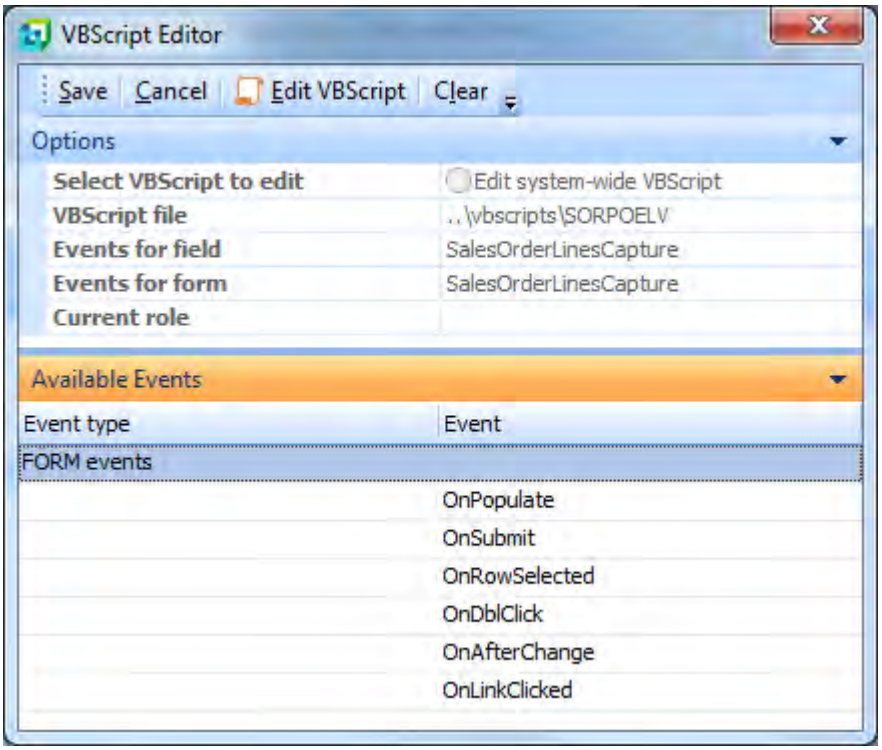


Figure 8-50: The macro events associated with a data grid

Data grids have macro events associated with them in the same way as listviews (see **Figure 8-50**). Three of the macro events (*OnPopulate*, *OnRowSelected*, and *OnDblClick*) are the same as the listviews, and work in mostly the same manner. Rather than cover these again in detail, refer to the *Using the OnPopulate Event*, *Using the OnRowSelected Event*, and *Using the OnDblClick Event* sections above. Only the differences will be covered below.

The other three events are specific to the data grids, and are *OnSubmit*, *OnAfterChange*, and *OnLinkClicked*.

OnPopulate
The *OnPopulate* event fires when the data grid is being used for the first time for this sales order, general ledger journal, etc. Unless the sales order/ledger journal is in maintenance mode, there will be nothing to populate the data grid at this time, but the event will still fire.

If the sales order/ledger entry is in maintenance mode, the data grid will be populated with the existing information. Within the *OnPopulate* function an array with the same name as the data grid, and one with the *_OUT* suffix will be present, in the same way as with the listviews. The *_OUT* array can be used to set field properties, so that any outliers are highlighted. If the *Value* field property is used to change the contents of a cell, this change is for display purposes only, and does not affect the value held in the database, even when the sales order/journal is saved.

OnAfterChange
The *OnAfterChange* event fires after the contents of a cell have been changed by an operator. This can be when changing the existing content of a row in maintenance mode, or when adding a row to a data grid. Each cell that is changed will cause this event to fire.

When this event fires the *ColumnClicked* variable against the array (with the same name as the data grid) will contain the column array number of the column that was changed. The *ColumnClicked* variable can be seen in **Figure 8-51**.

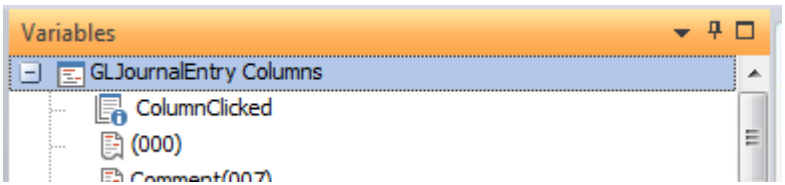


Figure 8-51: The *ColumnClicked* variable associated with the *GLJournalEntry Columns* array

The column number that is returned is the default one for the column that was changed. In the case of the *Sales Order Entry Express* program's data grid that can be seen in **Figure 8-49**, changing the *Order Quantity* against one of the rows would return the array column number of 7, even though this appears as the fifth column in the data grid/fourth column in the array, because the operator has

removed some columns. **Figure 8-52** shows the *OrderQty(007)* variable name within with the *SalesOrderlinesCapture Columns* section of the *Variables* pane.

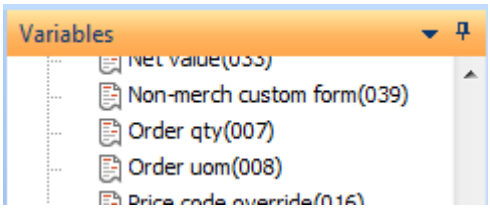


Figure 8-52: The *Order qty(007)* variable associated with the *SalesOrderlinesCapture Columns* array

When a cell is changed and the *OnAfterChange* event is fired, a one-dimensional array is available that contains all the columns for this row. The array name is the same as the name of the data grid, enabling you to use the variables within the *Variables* pane to retrieve these values.

The following uses the *Sales Order Line Capture* data grid that appears in **Figure 8-49** as an example. If you were to change the order quantity against the second row in the data grid from 1.000 to 2.000, the array called *SalesOrderLinesCapture* would contain all the values in the second line. The variables in **Figure 8-53** can then be used to access the values associated with the columns for this row (note that the row number in the variable must always be zero).

Double-clicking on the variable name will insert the full name of this variable into your code. The following is the code to display the stock code from the selected row (the *msgbox* statement has been manually inserted before the variable name).

```
msgbox SalesOrderLinesCapture.CodeObject.Array(004,0)
```

If you use the variable to access the value for the cell that was changed, the returned value is that from before it was changed. In this case the *Order qty(007)* variable would still contain the value 1.000.

When the *OnAfterChange* event fires, the *BeforeChangeValue* system variable contains the value of the cell before the operator changed it.

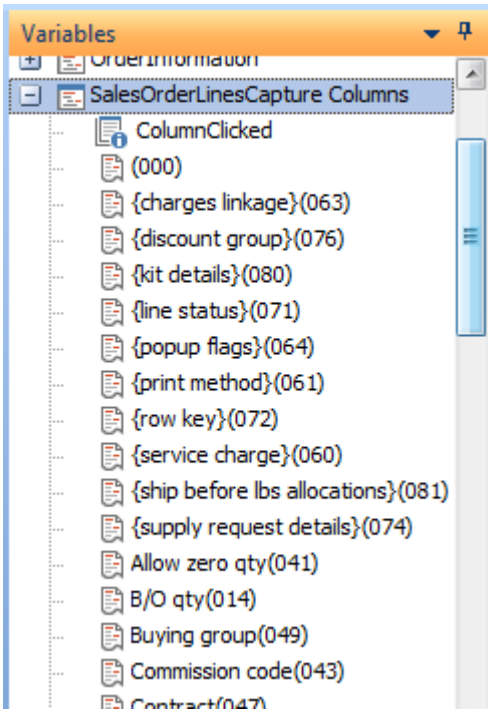
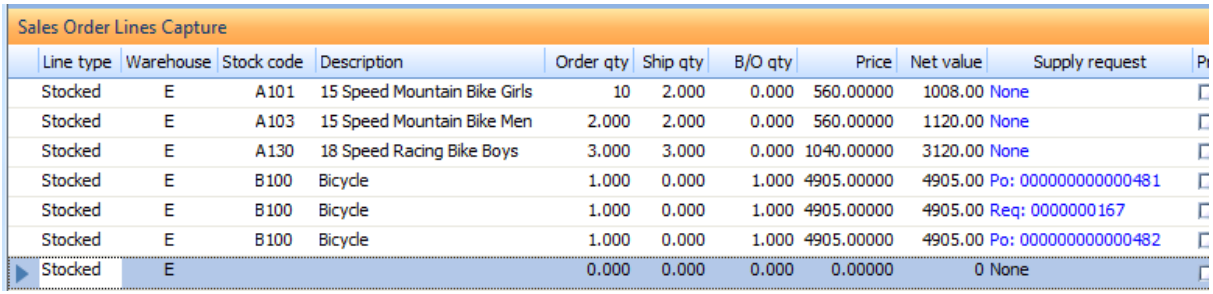


Figure 8-53: The *SalesOrderLinesCapture Columns* section

OnLinkClicked

The *OnLinkClicked* macro event is fired when you click on one of the hyperlinked values in a data grid. **Figure 8-54** shows the *Sales Order Lines Capture* data grid in the *Sales Order Entry Express* program. In this case the order has already been created, and the order is in maintenance mode.

The *Supply request* column contains hyperlinks. Clicking on one of these hyperlinks causes the *OnLinkClicked* event to fire. In the same way as the *OnAfterChange* macro event above, when this event fires the selected row is available within a one-dimensional array with the same name as the data grid. If code appears against the *OnLinkClicked* function, this code is executed before the hyperlink is actioned.



Sales Order Lines Capture

Line type	Warehouse	Stock code	Description	Order qty	Ship qty	B/O qty	Price	Net value	Supply request	Pr
Stocked	E	A101	15 Speed Mountain Bike Girls	10	2.000	0.000	560.00000	1008.00	None	
Stocked	E	A103	15 Speed Mountain Bike Men	2.000	2.000	0.000	560.00000	1120.00	None	
Stocked	E	A130	18 Speed Racing Bike Boys	3.000	3.000	0.000	1040.00000	3120.00	None	
Stocked	E	B100	Bicycle	1.000	0.000	1.000	4905.00000	4905.00	Po: 000000000000481	
Stocked	E	B100	Bicycle	1.000	0.000	1.000	4905.00000	4905.00	Req: 0000000167	
Stocked	E	B100	Bicycle	1.000	0.000	1.000	4905.00000	4905.00	Po: 000000000000482	
Stocked	E			0.000	0.000	0.000	0.00000	0	None	

Figure 8-54: The *Sales Order Lines Capture* data grid displaying order 0001084 in maintenance mode

OnSubmit

The data grid's *OnSubmit* event fires when the operator selects to save the sales order, ledger entry, sales order template, etc. The *OnSubmit* function can be configured to prevent the transaction from completing in SYSPRO by setting the function name to false. If this is done, all code within the function will be processed, but the sales order/ledger entry/template will not be updated, and will remain open.

The following example is from *Sales Order Lines Capture* data grid of the *Sales order Entry Express* program. Here a check is being performed to see if the *Geographic Area* field of the *Order Header* form contains the letter *N*, and if so, the *OnSubmit* function is set to false. If there was more code within this function it would also be processed, such as updating a non-SYSPRO SQL table. However, the data would not be saved, the data grid would remain populated, and focus would remain on the data grid.

```
Function SalesOrderLinesCapture_OnSubmit()
  If OrderHeader.CodeObject.Area = "N" then
    SalesOrderLinesCapture_OnSubmit = false
  End If
End Function
```

Using the Add Custom Columns Option

You cannot use the *Add Custom Column* option for a *Data grid* because it contains columns that are editable.

Chapter 9 - Toolbars, Form Actions, Login Script, Application Builder

This chapter covers the *Toolbars*, *Form Actions*, the login/logout scripts, and the *Application Builder*.

Toolbars

The vast majority of SYSPRO programs have menu bars and/or toolbars. These are located at the top of the screen immediately under the program's description. **Figure 9-1** shows a portion of the *Inventory Query* screen. The *Menu Bar* contains the three menu items of *File*, *Query* and *Preferences*. Immediately below this is the *Toolbar*, which starts with a toolbar item for entering the stock code.

Figure 9-1: The *Inventory Query* screen showing the *Menu Bar* and the *Toolbar*

Moving Toolbars and Menu Bars

At the far left of a toolbar and menu bar is a *Gripper*, which consists of four vertical dots. You can move a toolbar or menu bar to a new location by clicking on the gripper and dragging the bar to a new location within the section between the program description and the highest form/listview. During this process the mouse pointer will change to four radiating arrows (see **Figure 9-2**). Once it is in the correct location you release the mouse button and the bar will remain in this location. Alternatively you can drag it anywhere else on the screen and release it so that it becomes a floating menu/toolbar.

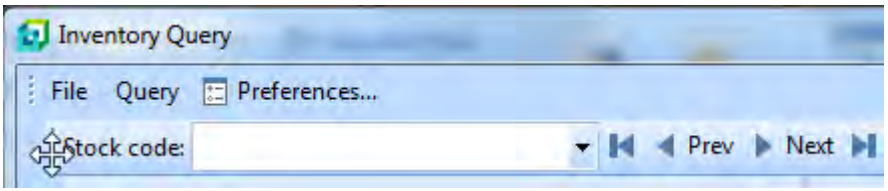


Figure 9-2: Dragging the toolbar to a new location

To restore the toolbar/menu bar to its original location you can either manually drag it back to its original position, or use the *Reset* option. This option will undo all changes that have been made to the

toolbar(s) and menu bar(s) in this program. This option is available by right-clicking anywhere on the toolbar or menu bar and selecting *Customize* (see **Figure 9-3**).

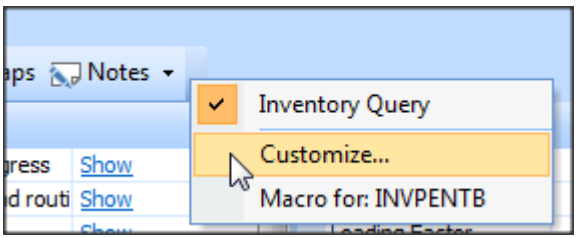


Figure 9-3: Selecting the *Customize* option from the toolbar

The *Customize* screen will be displayed, with the *Reset* button on the right (see **Figure 9-4**). If you select this button you are prompted that all changes to the toolbar(s) and menu bar(s) will be undone. If you select the *OK* button at the bottom of this screen, both the toolbar and menu bar will revert to their default settings. Use the *Close* button to exit from the customize mode.

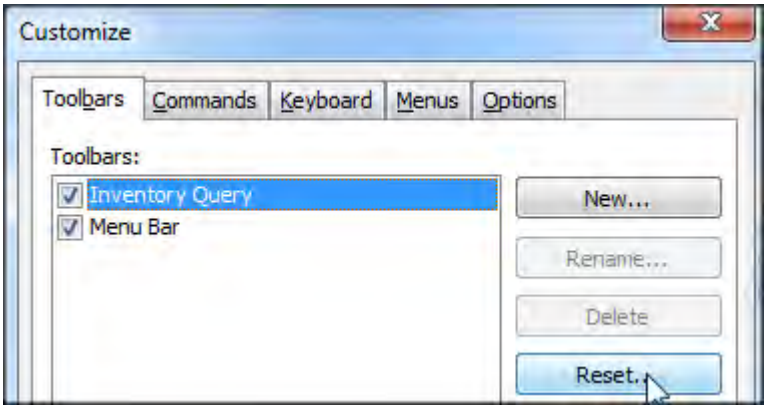


Figure 9-4: The *Reset* button on the *Customize* screen

Hiding/Showing Buttons on a Toolbar

On standard toolbars you can show or hide the standard buttons. If you click on the *Toolbar Options* button and select *Add or Remove Buttons*, a list of the standard toolbars for this program will be displayed. When one of these standard toolbar names is selected a list of the standard buttons is displayed (see **Figure 9-5**). The buttons that are currently being displayed will have a checkmark against them. If you uncheck one of the buttons, the matching button will be removed from the toolbar in real time. If you check one of the unchecked buttons, it will reappear.

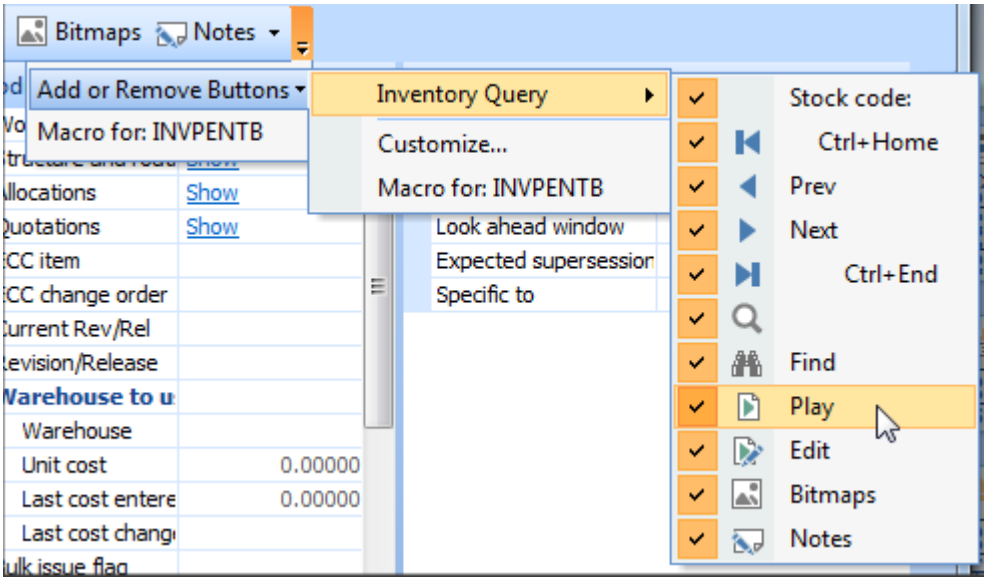


Figure 9-5: Showing/hiding the toolbar buttons

If buttons have been moved from their original toolbar to another, they will still appear in the original list but will be displayed unchecked. They will reappear on the original toolbar if their option is checked, as well as the other toolbar.

Buttons that have been renamed will appear in this list with their new names, and can be shown/hidden. Buttons that have been added by the operator or administrator will appear in this list, but the option to show/hide them is disabled. **Figure 9-6** shows the show/hide option for the buttons on the *Inventory Query* toolbar. The third to last button has been renamed from *Bitmaps* to *Images*. Because it is a standard button that has been renamed, it is available to be hidden/shown. However, the *My Button* button was created by the operator, therefore it is not available to be hidden/shown, and has been greyed-out.

Creating a New Toolbar

Most programs have one toolbar, but some have more than one. Some toolbars contain many buttons, such as the toolbar in *Inventory Query* that can be seen in **Figure 9-1**. When reducing the width of the screen to below the original width of the toolbar, the buttons to the right are no longer visible. These can be seen in a dropdown if you click on the *Toolbar Options* button (see **Figure 9-7**).

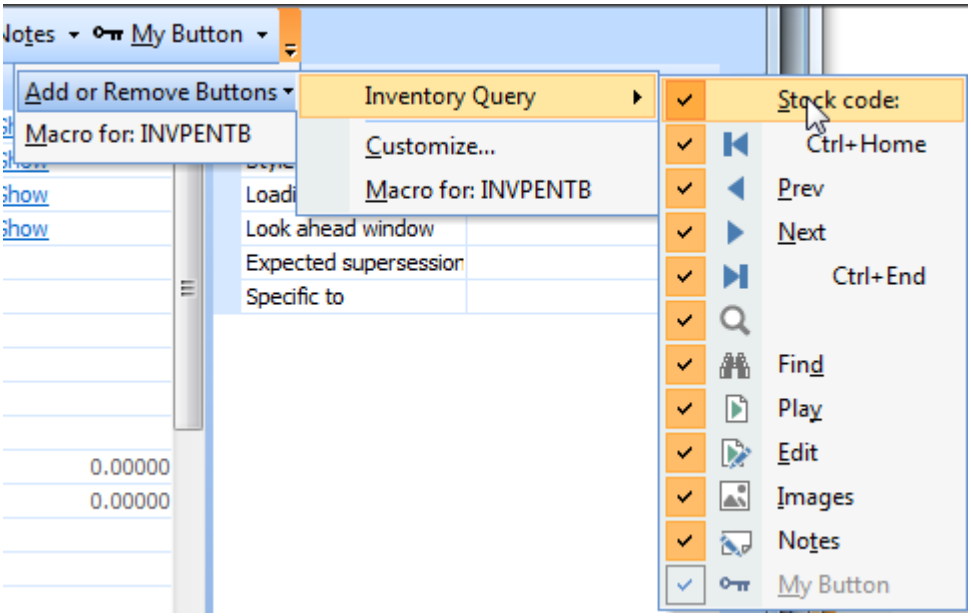


Figure 9-6: The list of buttons that can be shown or hidden

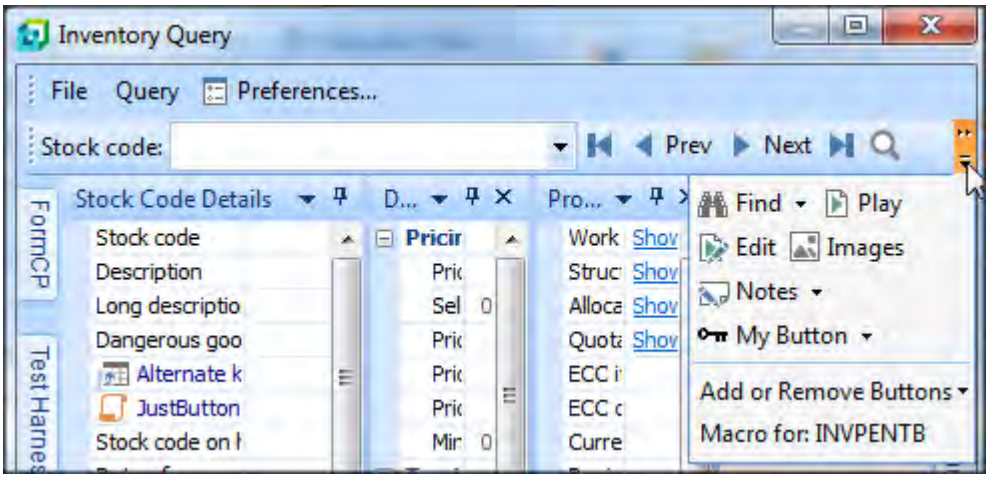


Figure 9-7: Viewing the buttons when the screen is narrower than the toolbar

If you prefer to see all the buttons on this reduced width screen you can create another toolbar and move some of the buttons from the original toolbar to the new one. To create a new toolbar you need to be in customize mode (i.e.right-click anywhere on the toolbar and select *Customize* from the

displayed menu). The *Customize* screen is displayed and the available toolbars will be listed. In most cases there is a toolbar with the program description, and a menu bar (see **Figure 9-8**).

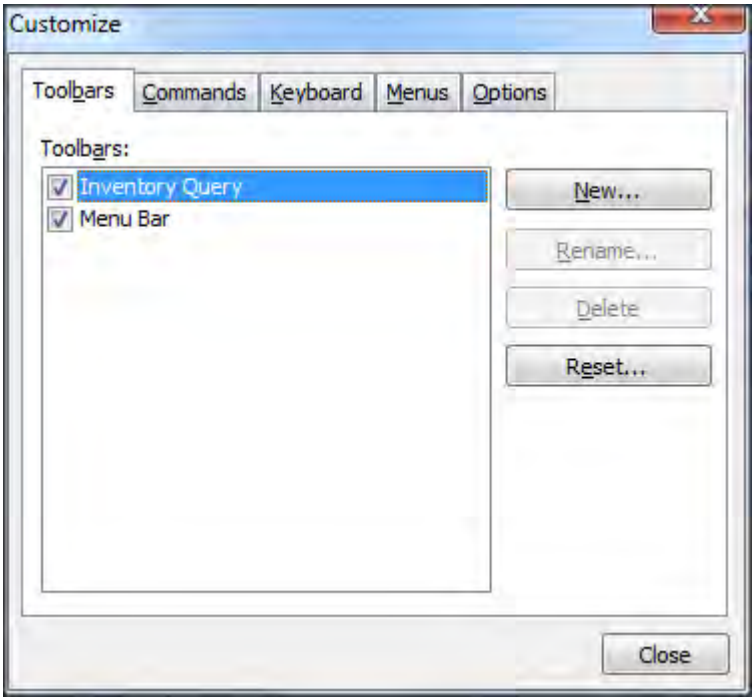


Figure 9-8: The toolbar with the program name, and the *Menu Bar*

To add a new toolbar, click on the *New* button. You will be prompted to supply a name for this toolbar. In this example the name *Lower Inventory Query* was used (see **Figure 9-9**). When you click on the OK button the toolbar name will be added to this list (see **Figure 9-10**), and an empty toolbar will be added to the toolbar area of the program.

Use the *Close* button to exit the customize *screen* and you will see the new toolbar below the existing toolbar (see **Figure 9-11**).

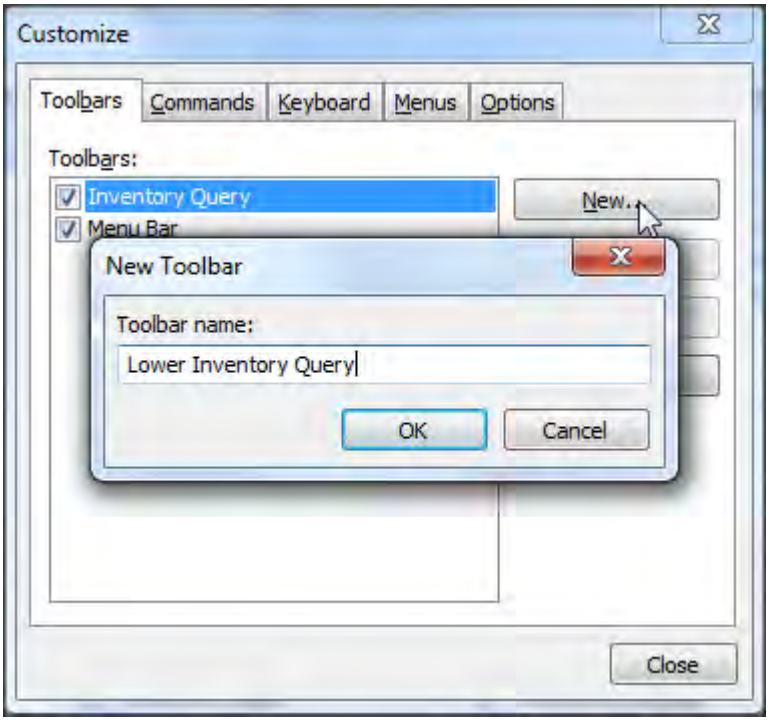


Figure 9-9: Adding a new toolbar

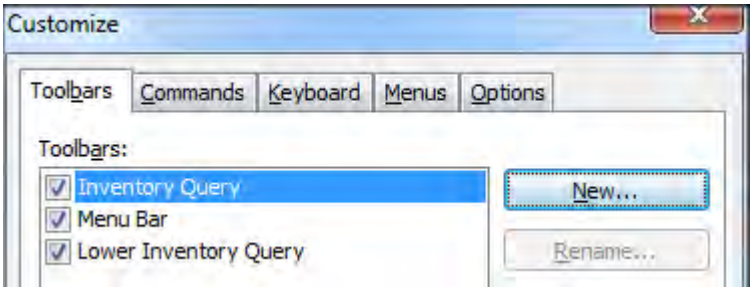


Figure 9-10: The new toolbar has been added

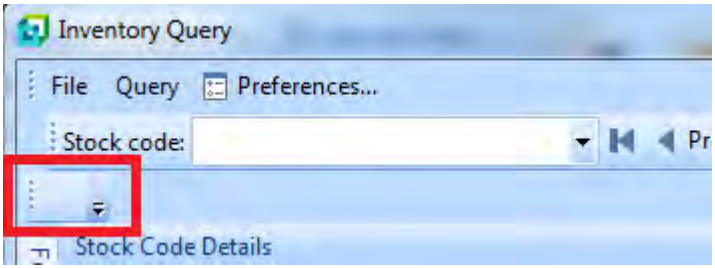


Figure 9-11: The newly-added empty toolbar

Moving Buttons from one Toolbar to Another

To move buttons from one toolbar to another you must be in customize mode (right-click on one of the buttons and select *Customize* from the menu). Click on one of the buttons on the *Inventory Query* toolbar and drag it to the *Lower Inventory Query* toolbar. A black vertical line will appear on the *Lower Inventory Query* toolbar at the point where the button will be placed if you release the mouse button. When you release the mouse button the toolbar button will be removed from its position on the original toolbar and appear in the new one. The button on the new toolbar will be highlighted. Repeat this process for all the other buttons that must be moved from the original toolbar to the new toolbar.

In **Figure 9-12** it can be seen that the *Play*, *Edit*, *Images*, and *Notes* buttons have already been dragged from the *Inventory Query* toolbar to the *Lower Inventory Query* toolbar. The *My Button* button is in the process of being dragged to the *Lower Inventory Query* toolbar.

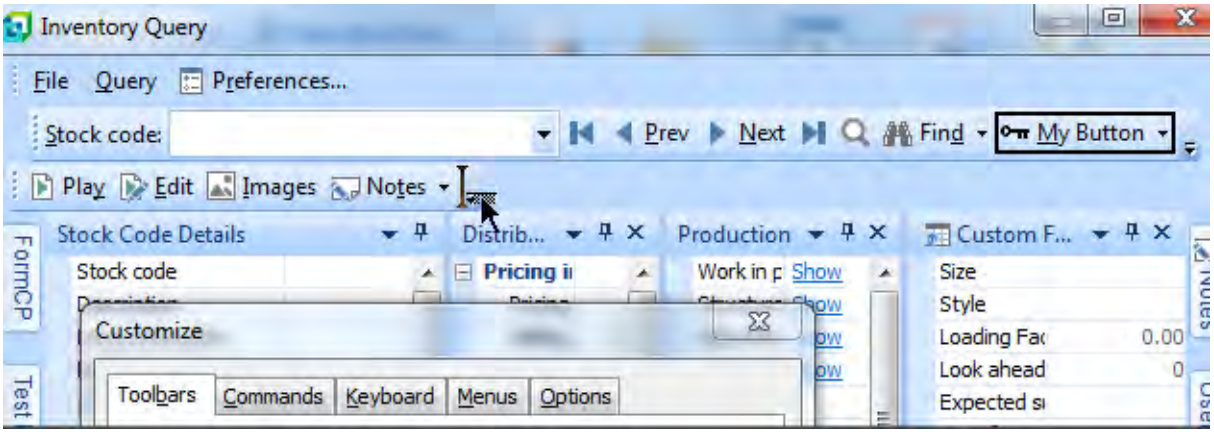


Figure 9-12: Dragging the *My Button* button to the *Lower Inventory Query* toolbar

Once the copy process has been completed, exit the *Customize* screen by clicking on the *Close* button. Because there are now two toolbars, all of the buttons can be seen on the reduced width screen (see **Figure 9-13**).

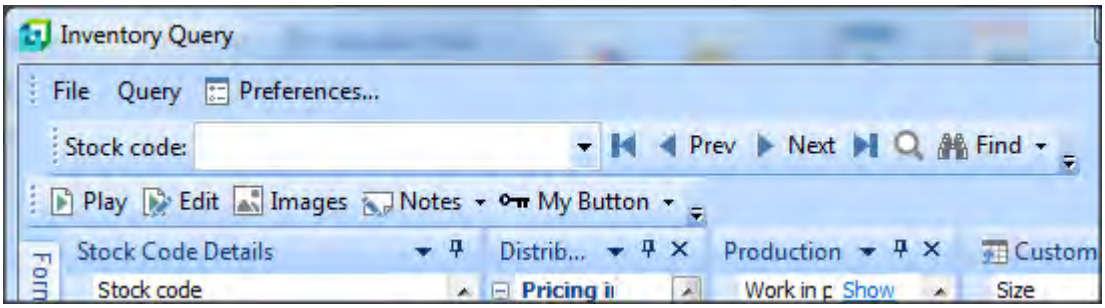


Figure 9-13: The completed *Lower inventory Query* toolbar

Modifying a Button's Name and Functionality

The name and action performed by an existing button can be changed. In customize mode, right-click on the relevant button to display the properties for the button (see **Figure 9-14**). The name is changed by overwriting the existing name and pressing the *Enter* key. The display of the button's properties will be closed, and the button name will be changed.

If you want to add a shortcut keystroke to the button, add an ampersand directly before the letter that is to be the shortcut. In **Figure 9-14** you can see that the ampersand has been added just before the letter M. The shortcut keystroke *Alt+M* invokes the action of this button.

To change any of the other values against the button, right-click the button again.

You can specify if an image must be displayed against this button (the image is made up of 16 x 16 pixels). You can copy an image from another button and paste it against this button, reset this button to its default image, select the image from a list of standard ones, or call up a button image editor and tweak an existing one/create your own.

The button can be configured to show the default style (which is to just show the icon that is configured for the button), to show only the button's name, or to show both the button's image and its name.

The button can be configured to call a SYSPRO program, call another application, or call a VBScript function. If more than one of these is entered, whichever of these was the last to be entered will be the one used, and the others will be discarded. The application name and path must be relative to the client machine, not the server.

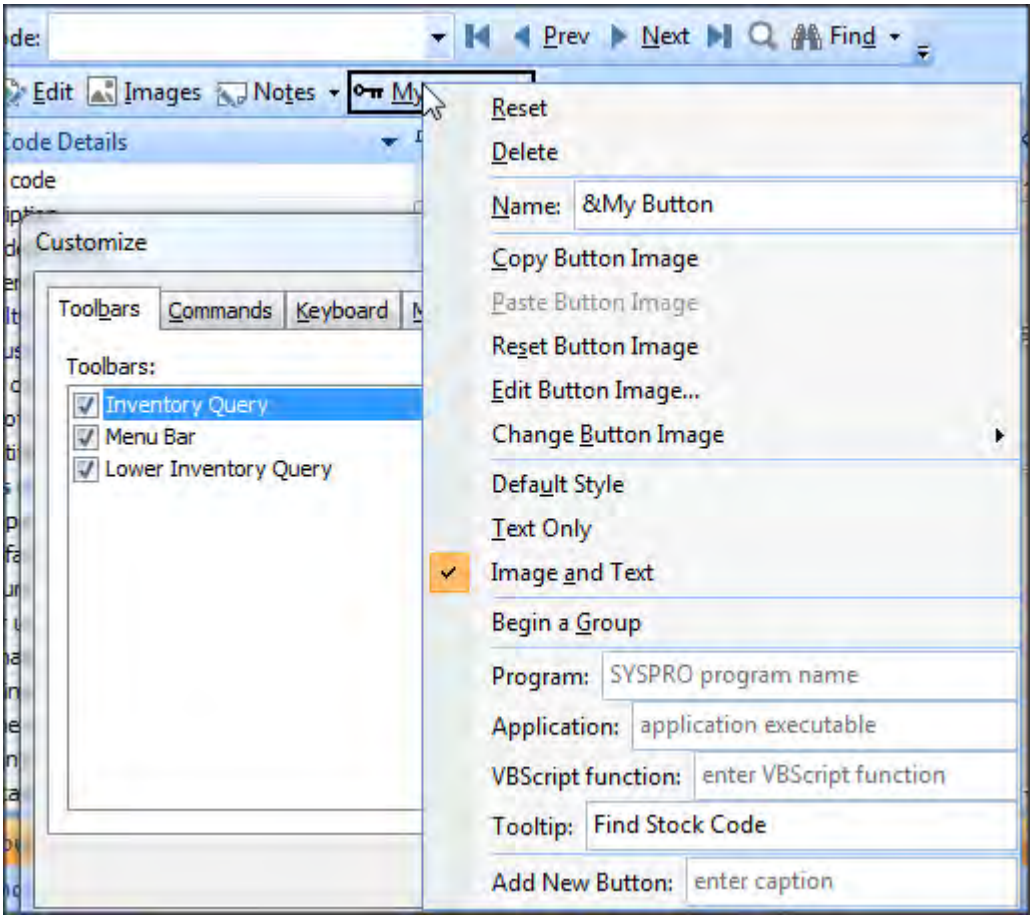


Figure 9-14: The properties of the *My Button* button

You can assign a tooltip to the button, which will be displayed when you hover over the button (see **Figure 9-15**).

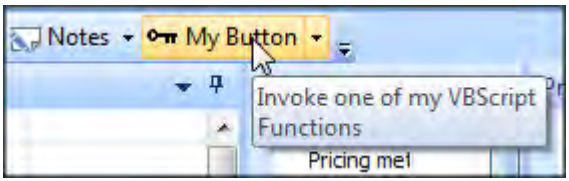


Figure 9-15: The tooltip added to the button

Adding VBScript to a Toolbar Button

As mentioned in the *Modifying a Button's Name and Functionality* section, a button can be configured to call a VBScript function. There are two parts to this, specifying the name of the VBScript function against the button, and creating the function in the *VBScript Editor*. If the name of the function has been added to the toolbar button but the function does not exist, when the operator clicks on the button, the calling of the function is ignored (and no error is displayed).

The name of the VBScript function is entered against the *VBScript function* prompt when displaying the properties of the button in customize mode. **Figure 9-16** shows the function name *My VBScript Code* entered against a button. When you press the *Enter* key the properties will be closed. Exit customize mode by clicking on the *Close* button on the *Customize* screen.

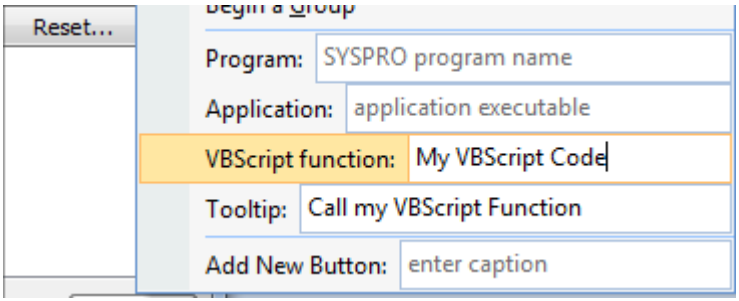


Figure 9-16: Entering the name of the VBScript function to use

Click on the *Toolbar Options* button at the far right of any of the toolbars, and select the option that starts with *Macro for:* followed by the name of the toolbar (see **Figure 9-17**).

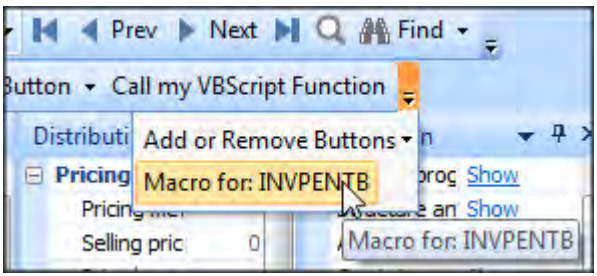


Figure 9-17: Selecting the *Macro for: INVPENTB* option to call up the *VBScript Editor* screen

The *VBScript Editor* screen will be displayed. This screen is different to most other *VBScript Editor* screens throughout SYSPRO in that it does not display the names of any *Macro Events* (see **Figure 9-18**).

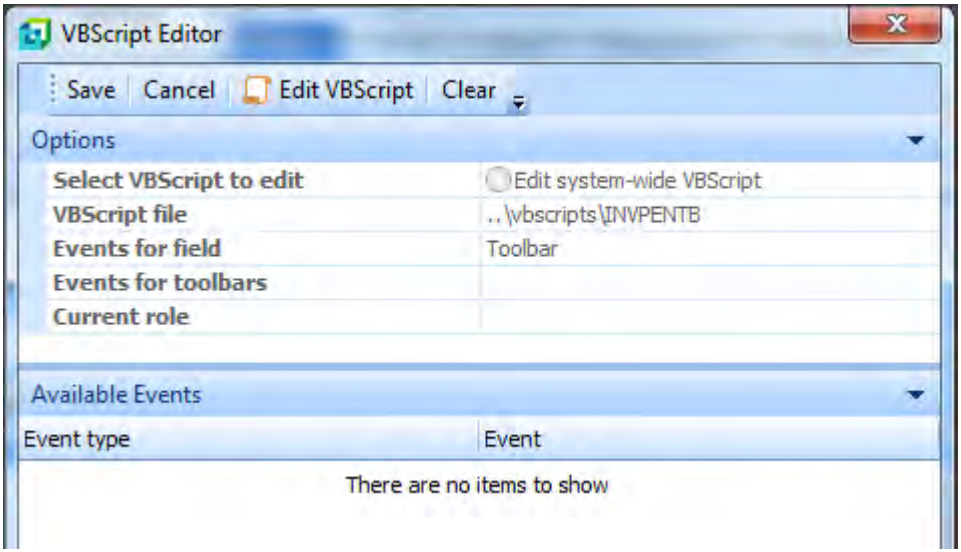


Figure 9-18: The *VBScript Editor* without any *Macro Events*

The function must be manually created. This is done by clicking on the *Edit VBScript* button, which loads the *VBScript for:* screen, where you add your VBScript code. Normally this would contain some comment lines and the *Option Explicit* statement. In this case it is completely empty (see **Figure 9-19**).

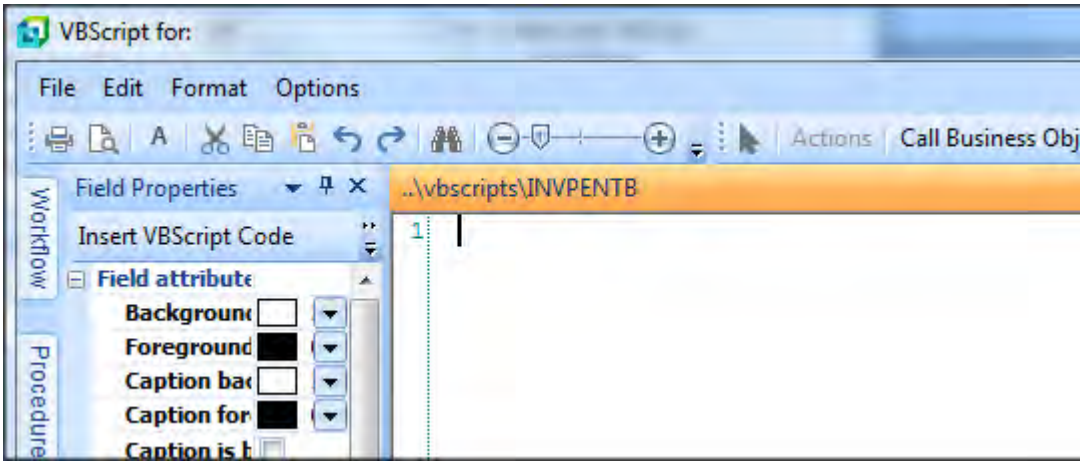


Figure 9-19: The completely empty *VBScript for:* screen

The first line of code to enter is the *Option Explicit* statement, which forces you to explicitly declare variables before you can use them.

The next line of code will be the start of the function itself. The name of the function that was supplied to the toolbar button was *My VBScript Code*, but function names cannot contain spaces. The name of the function is the same as supplied to the toolbar, but with the spaces removed. So the line of code to open the function is:

```
Function MyVBScriptCode()
```

The rest of the code to perform the function must then be entered, followed by the *End Function* statement. Below is a sample of code that displays the stock code description (because this toolbar is part of the *Inventory Query* program.

```
Option Explicit

Function MyVBScriptCode()
  MyVBScriptCode = false
  msgbox StockCodeDetails.CodeObject.Description,, "Stock Description"
End Function
```

The first line of code within the function doesn't need to be present in this case, because the button containing the VBScript function name was created from scratch. This line of code is useful when you have copied a button to create this one. It prevents the new button from performing its original task, or a task that was linked to the original button when it was copied.

When the function has been entered, exit the *VBScript for:* program in the normal way. At the *VBScript Editor* screen, click on the *Save* button, and you will be returned to the program containing the button.

Creating New Buttons on a Toolbar

The simplest way to add new buttons to an existing toolbar is by copying an existing button. This is done in customize mode. Hold down the *Ctrl* key on the keyboard and click on the button ypu want to copy, and drag the button to another location on the toolbar. As you hover over the toolbar, an area will be highlighted with a black vertical bar. This shows where the button will be placed if you release the mouse button. When you release the button a duplicate of the selected button will be added to the toolbar. You can change the name/functionality of the button by right-clicking on it whilst still in customize mode (see the section *Modifying a Buttons Name and Functionality* above). If you have copied a button so that you can use it to run a VBScript function you probably won't want the original task of the button to be actioned when it is clicked. Inside the VBScript function you should add the following line of code to tell SYSPRO to ignore the button's original task (assuming that the name of the VBScript function is MyVBScriptCode):

```
MyVBScriptCode = false
```

A new button can be added to a toolbar when in customize mode. Right-click on a toolbar button to display its properties. At the *Add New Button* option enter the name you want to assign to the and press *Enter*. The new button is created. This button is completely blank apart from the name that it was given.

The image in **Figure 9-20** shows where the properties of the *My Button* button are displayed, and the *Add New Button* prompt has *Call my VBScript Function* against it. When the operator presses the *Enter* key, a button with this name will be created. This is the button that was used in the *Adding VBScript to a Toolbar Button* section above.

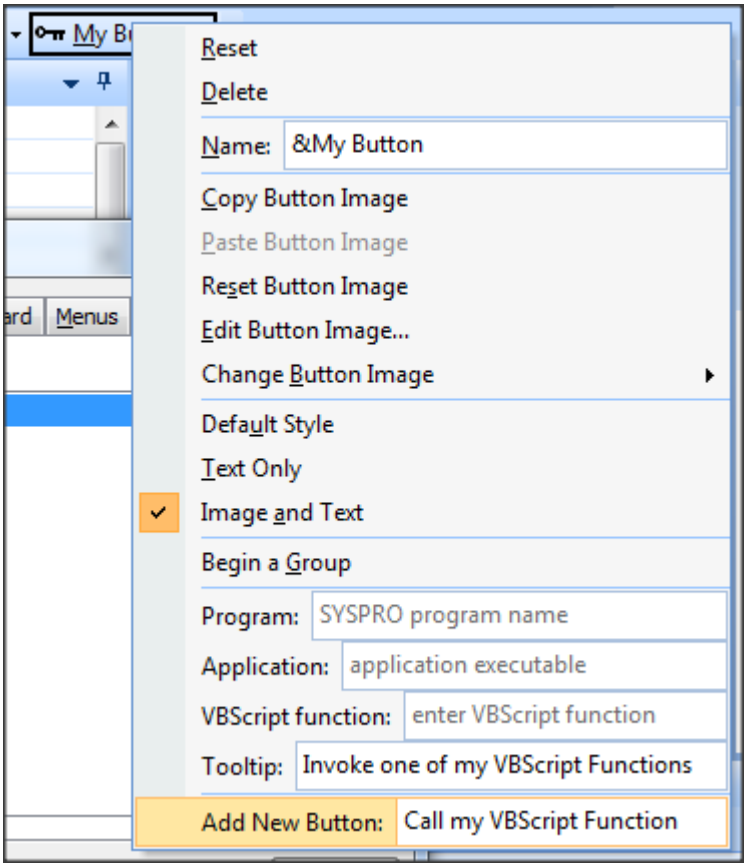


Figure 9-20: Adding a new button using the *Add New Button* option

Once the VBScript code has been added to the function for this button (*Function MyVBScriptCode*), when you click on the *Call my VBScript Function* button a message box is displayed that shows the stock code's description (see **Figure 9-21**).

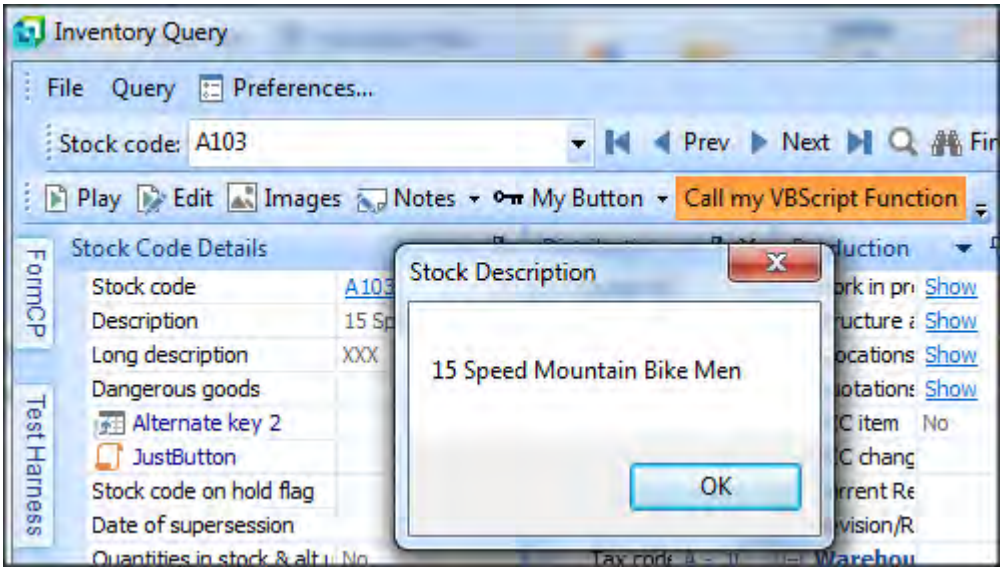


Figure 9-21: The message box displayed when you click the *Call my VBScript Function* button

Security Settings Affecting Toolbars

Administrators can restrict operators from modifying toolbars/menu bars using the operator activity: *Toolbars – Customization by operator*. (*SYSPRO Ribbon Bar* | *Setup* tab | *Operators* | select operator from the list | *Edit* | *Change* | *Security* tab). The *Toolbars – Customization by operator* activity can be seen in **Figure 9-22**.

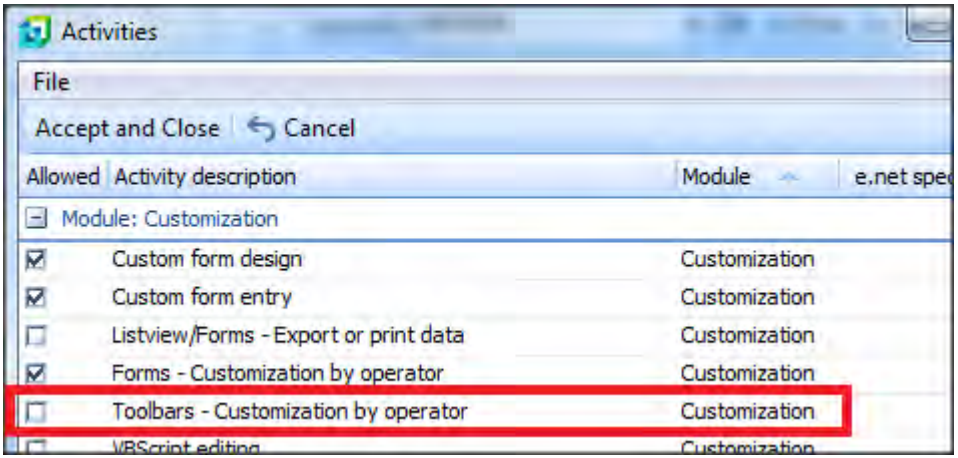


Figure 9-22: The *Toolbars – Customization by operator* activity

Toolbars and Roles
If your operator is a member of a role, the ability to go into the toolbar's *Customize* mode is disabled, unless you are in *Design Role Layout* mode before calling up the program containing the toolbar. This is configured using the *Role layout design* checkbox on the *Options* tab of the *Operator Maintenance* program.

Form Action Events

A *Form Action* is an extremely powerful user-definable hyperlink that is located at the bottom of a form. **Figure 9-23** shows where a form action called *Status of outstanding items* has been added to the *Job Details* pane of the *Job Query* program.

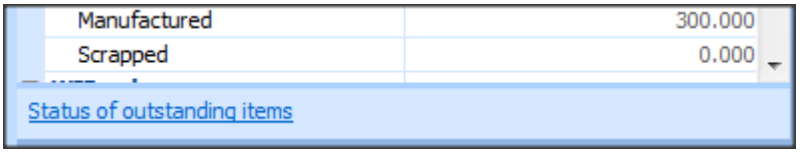


Figure 9-23: The *Status of outstanding items* form action in the *Job Query* program

The hyperlink can be configured to call a SYSPRO program and pass it parameters, or to fire a VBScript event. Multiple form actions can be defined per form, and they can be any combination of these two types. Chapter 4 of the first *Power Tailoring* book described how to configure a form action to call a SYSPRO program and pass it parameters (as this requires no development skills). This section covers the basics of creating a VBScript form action, as these fire macro events that call functions within a VBScript.

A form action is added using the *Insert Form Action* option from the context-sensitive menu that appears when you right-click on a form. This option can be used to add a form action that will use VBScript, a form action that will call a SYSPRO program (and optionally pass it a parameter) or add a form action template, which is a predefined sample.

The image in **Figure 9-24** shows how to add a form action using the *Insert Form Action* option of the context-sensitive menu that is displayed when you right-click on a form. As this form action is going to fire an event (as opposed to call a SYSPRO program) only the description is added, in this case *Status of outstanding items*. When you press the *Enter* key the hyperlink is added to the form and the list of available events for this form is shown in the *VBScript Editor* screen.

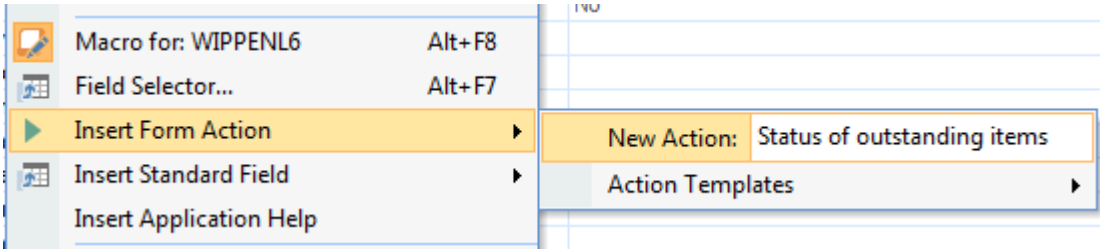


Figure 9-24: Adding the VBScript form action *Status of outstanding items*

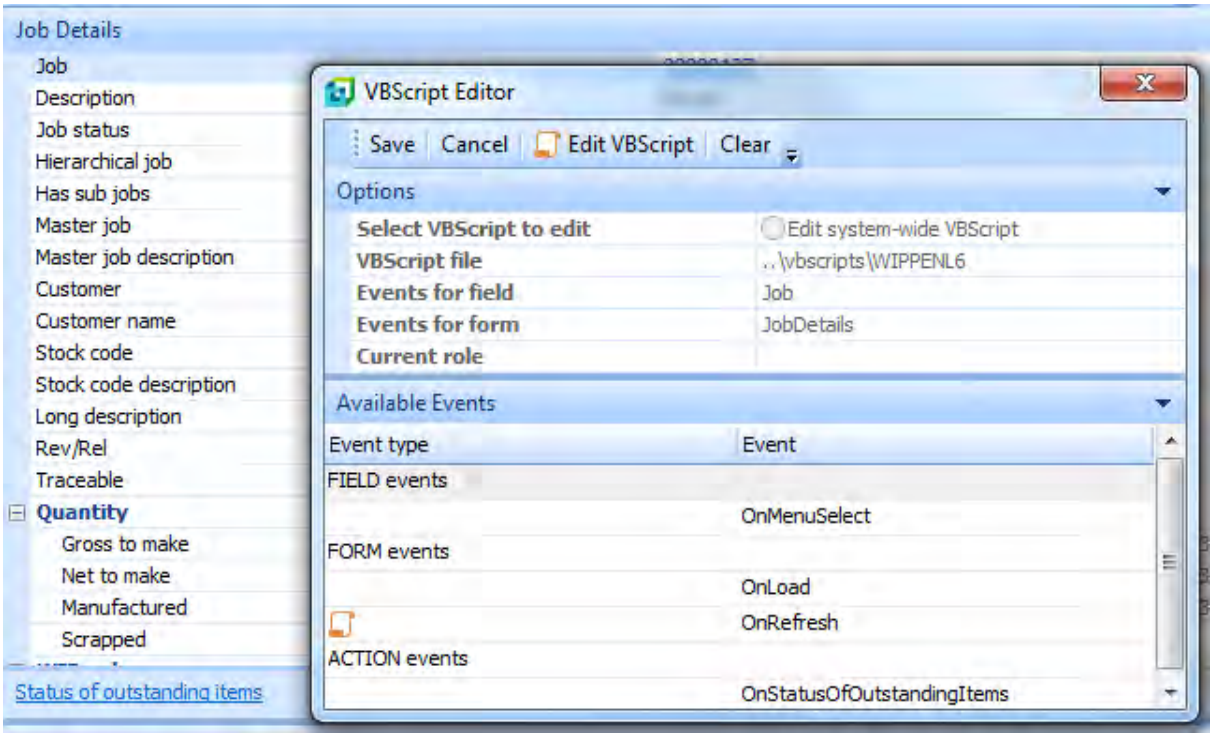


Figure 9-25: The form action is created and *VBScript Editor* screen is loaded

In **Figure 9-25** you can see the *Status of outstanding items* form action has been added to the bottom of the *Job Details* form, and the *VBScript Editor* screen where the *OnStatusOfOutstandingItems* event has been automatically added under a section called *ACTION events*.

The function is added by either highlighting this event and clicking on the *Insert VBScript* button, or double-clicking on this event name. In the same way as any other function, the function name will be

made up of the form name and the event name. Nearly all event names start with *On*, so this will be added to the function name. Spaces are removed, and the first letter of each word will be capitalized. As the *Status of outstanding items* form action was added to the *Job Query* program's *Job Details* form, the function name is *JobDetails_OnStatusOfOutstandingItems*.

Figure 9-26 shows the *JobDetails_OnStatusOfOutstandingItems* function after a message box statement has been added. The message box is used to show that the event is firing when the hyperlink is clicked. Obviously, this is just for testing that the macro event fires. The finished code could be made to look through all items that have not been issued to the job, check the availability in stock (if there is no stock), check the purchase orders, etc.

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function JobDetails_OnStatusOfOutstandingItems()
6    msgbox "JobDetails_OnStatusOfOutstandingItems fired"
7  End Function
```

Figure 9-26: The completed function

When you exit the screen where the VBScript was edited, you are returned to the *Job Query* program, where you can see the *Status of outstanding items* form action hyperlink at the bottom of the *Job Details* form. When you click on the hyperlink, the event fires and the code against the function is run, in this case displaying a message box (see **Figure 9-27**).

The *Insert Form Action* option also displays the currently available form actions (see **Figure 9-28**). These form actions can be removed by unselecting them. You will be prompted before they are removed. Although the removal process removes the *Action event* from the *VBScript Editor* screen, the function and code must be removed manually from the VBScript.

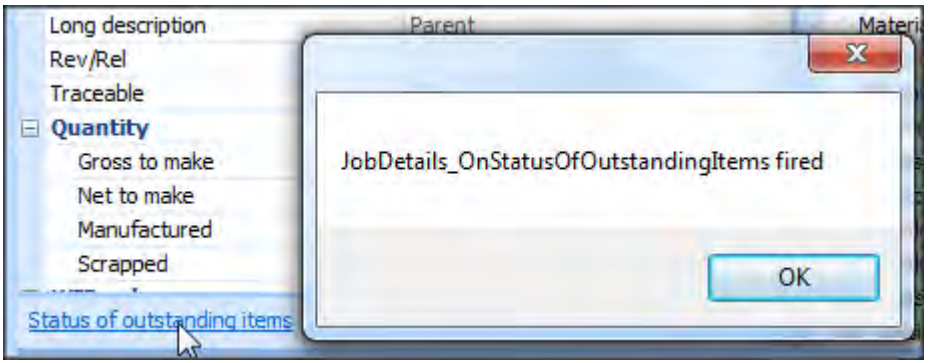


Figure 9-27: Clicking on the *Status of outstanding items* form action

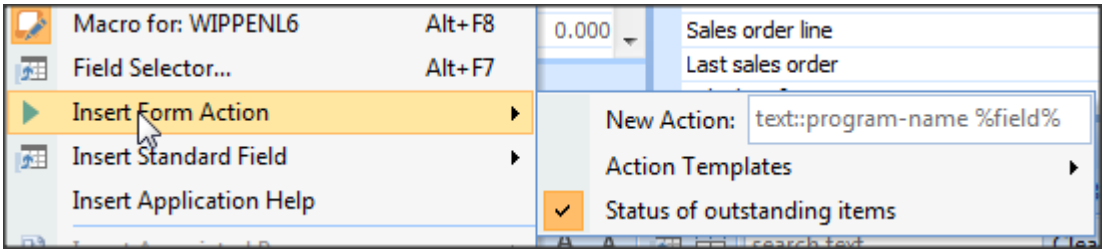


Figure 9-28: The available *Form Actions*

Global Login/Logout VBScript

The *Global Login/Logout VBScript* option is available from the *Customization Tools* dropdown of the *Administration* tab of the Ribbon Bar (*Ribbon bar* | *Administration* tab | *Customization Tools* button | *Global Login/Logout VBScript* option) (see **Figure 9-29**). It is available on the *Home* tab for versions prior to SYSPRO 7.

Figure 9-29: The location of the *Global Login/Logout VBScript* option

When this option is selected the *VBScript Editor* screen is displayed, which contains the macro events *OnLogin*, and *OnLogout* (see **Figure 9-30**). If the functions for these two events are created they will

be stored in a file called `IMPMENXX` in the `Work\vbscripts` folder on the SYSPRO application server, and executed by everyone that logs into, or logs out of, SYSPRO.

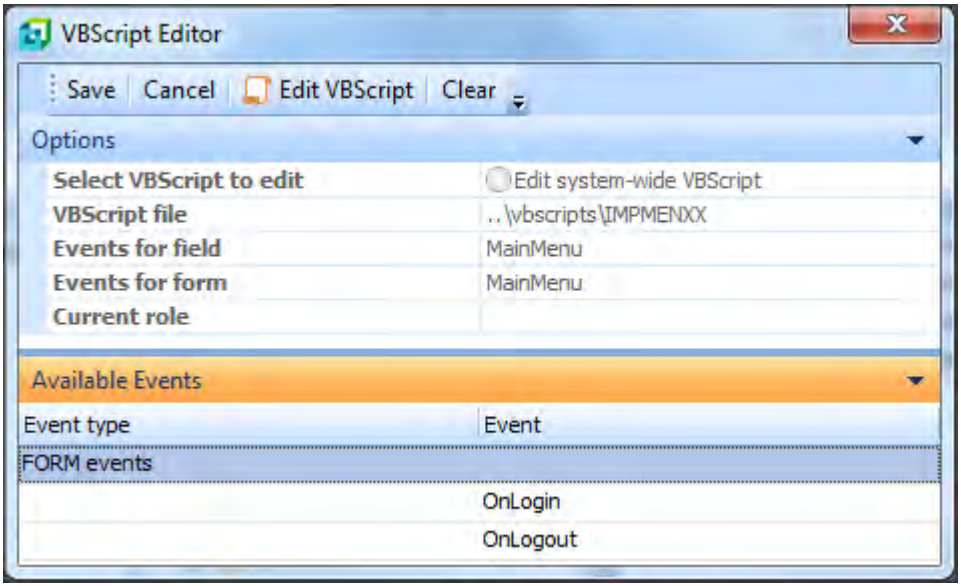


Figure 9-30: The *OnLogin* and *OnLogout* form events

OnLogin

The *OnLogin* event fires just as you complete the login process (i.e. supplied the operator code, company etc., and clicked on the *OK* button), but before the main menu is loaded. This means that checks can be performed to make sure that items are in place, such as drive mappings and other settings, before the operator starts work. If these checks fail, the operator can be prevented from logging in by setting:

```
MainMenu_OnLogin = False
```

A simple *OnLogin* script appears below that prevents operator *BOOK14* from logging in if they select company *0*:

```
Function MainMenu_OnLogin()
 If SystemVariables.CodeObject.Company = "0" then
   If SystemVariables.CodeObject.Operator = "BOOK14" then
     Msgbox "You are not permitted to login to this company",, "Access Denied"
     MainMenu_OnLogin = False
   End If
 End If
End Function
```

Setting the *MainMenu_OnLogin* variable to *False* logs the operator out. As they have not completed the login process, the *MainMenu_OnLogout* event does not fire.

As the login script runs before the main menu is loaded you cannot call other SYSPRO programs, or interact with any customized panes (whether associated with the main menu or a specific program). However, the script can call e.net Solutions business objects. Only the *System Variables* are available to VBScript code against the *OnLogin* function, although some of these such as *SYSPROProgramToRun* will not work, as the main menu is not loaded.

OnLogout
The *OnLogout* event fires when an operator logs out of SYSPRO. Although most of the *System Variables* are available at this point, you cannot use them (such as *SYSPROProgramToRun*) as they will not load the program. The e.net Solutions business objects can still be used at this point.

If you wanted to keep an external copy of who logged in/out you would probably implement the logout part against this event.

Application Builder

The *Application Builder* menu appears on the *Administration* tab of the SYSPRO Ribbon bar. It enables you to build up to 20 sandboxed applications per operator within SYSPRO, along with one application that is available for each role and one that is available system-wide.

Each of these applications can contain one or more customized panes. As these applications are launched from within SYSPRO they can access e.net Solutions business objects without requiring additional licenses. Although each of these applications is sandboxed, if one contains multiple customized panes, these panes can talk to each other within the same application.

The *Application Builder* program descriptions default *to Application x (IMPDHy)*, where *x* indicates by the application number in the range 1 to 20, and *y* indicates either the application number (if it is 1 to 9) or the letters A to K (if the application number is between 10 and 20).

When you modify the application you are able to change this description to something that is relevant to the application. **Figure 9-31** shows the list of applications in the *Application Builder* dropdown list where the first application's description has been changed from *Application 1 (IMPDH1)* to *My First Application*.

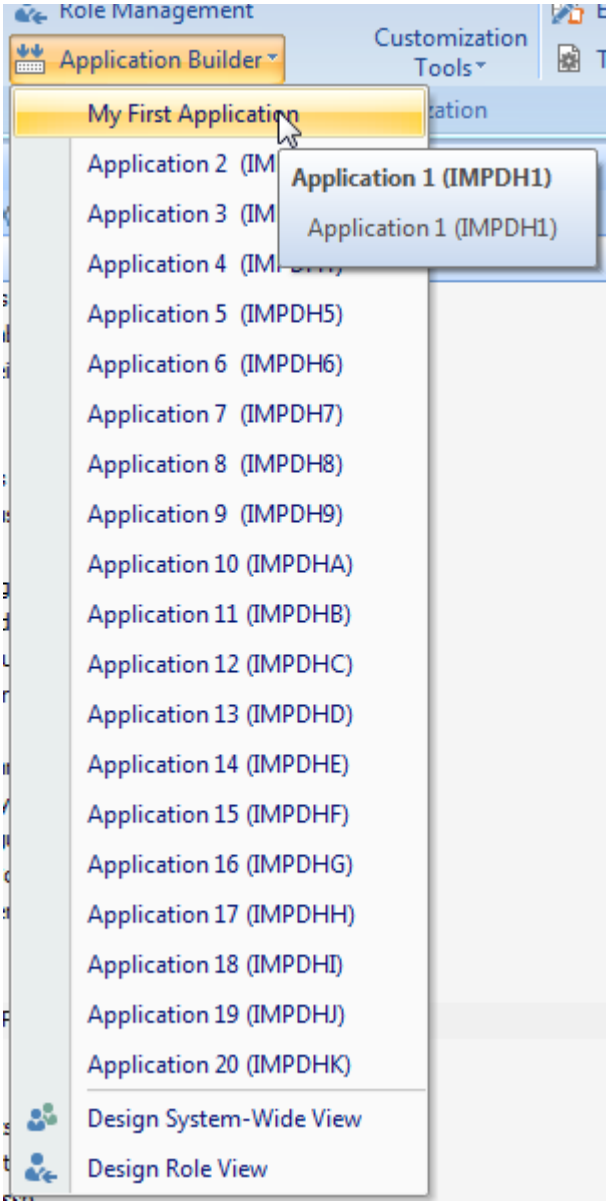


Figure 9-31: The *Application Builder* menu

Application Builder Program Access

Each application in the dropdown list, including the *Design System-wide View* and *Design Role View* options is controlled by a separate entry in the operator group security (see **Figure 9-32**). If you can only access applications 1 to 9 it may be that your security was configured when you were on a prior version of SYSPRO which only catered for nine applications, and did not have the concept of the role or system-wide views. In this case you will need to go to the *Operator Group Maintenance* program and allow access to the applications that you want to use (*SYSPRO Ribbon Bar* | *Setup* tab | *Groups* | select group to be maintained | *Edit* | *Change* | *System Admin & Shared Programs* section).

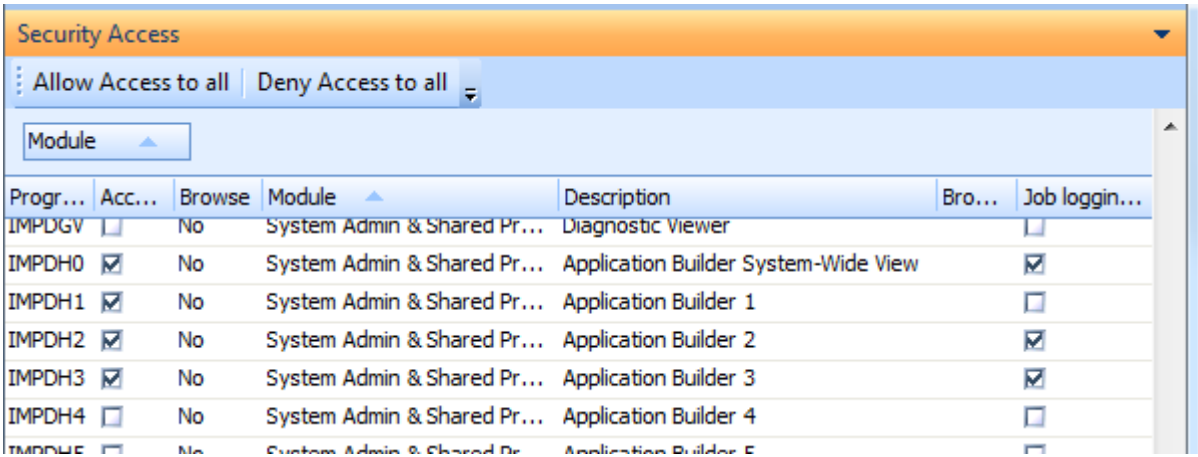


Figure 9-32: Security access by operator group

Required Permission to Modify an Application

All operators can call up applications within the *Application Builder* menu providing that they have been given program access (see the previous section). To be able to modify an application the operator needs to enable the operator activity *VBScript editing* (see **Figure 9-33**). This appears within the *Customization* section of the *Activities* program.

If the operator does not have this option checked, all of the buttons on the toolbar apart from the *Close* button will be disabled (see **Figure 9-34**). The application will still function, but the operator will not be able to change the functionality.

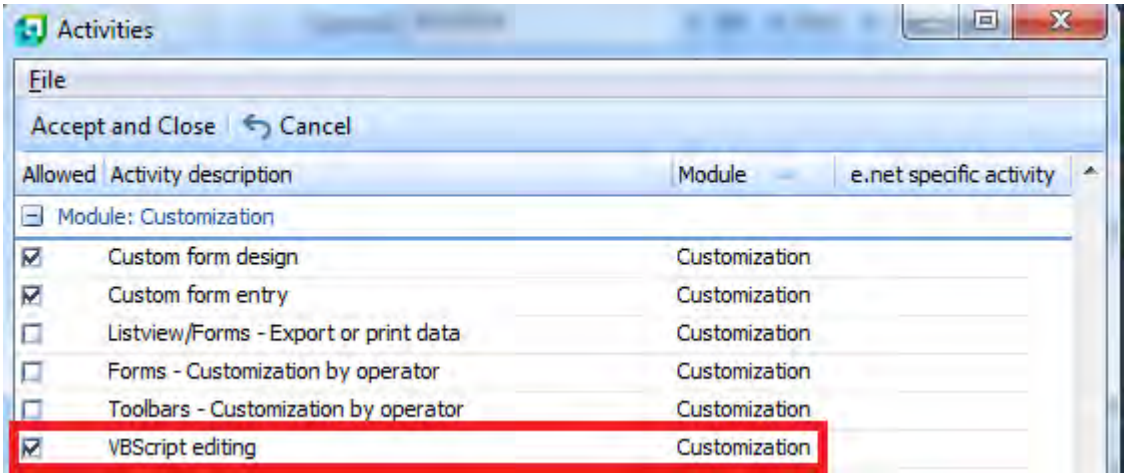


Figure 9-33: The *VBScript editing* operator activity

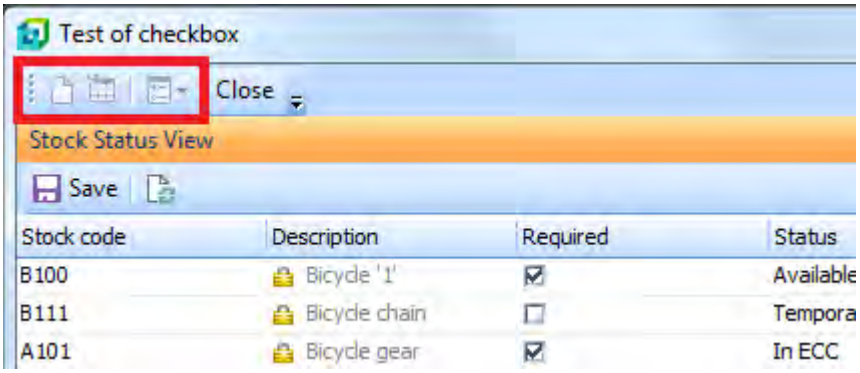


Figure 9-34: Only the *Close* button is available as the *VBScript editing* operator activity is unchecked

Modifying an Application

The application can consist of one or more customized panes. If the operator has permission to modify the application, the toolbar will comprise four buttons: *Add a Customized Pane*, *Import a Customized Pane*, *Options*, and *Close*.

The *Add a Customized Pane* button calls up the *Customized Pane Editor* screen. This works exactly the same as if it had been called up in another program (see *Chapter 10 – Customized Panes*).

The *Import a Customized Pane* button calls up the import program in the same way as if it had been called up from another program (see the *Importing a Customized Pane* section of *Chapter 10 – Customized Panes*).

There are two menu items that appear when you click on the *Options* button. The first is *Application Title* that allows you to change the title of the application. The second is *OnClose VBScript Event* that enables you to add VBScript code to a function that is run when the operator closes the program. Both of these can be seen in **Figure 9-35**.

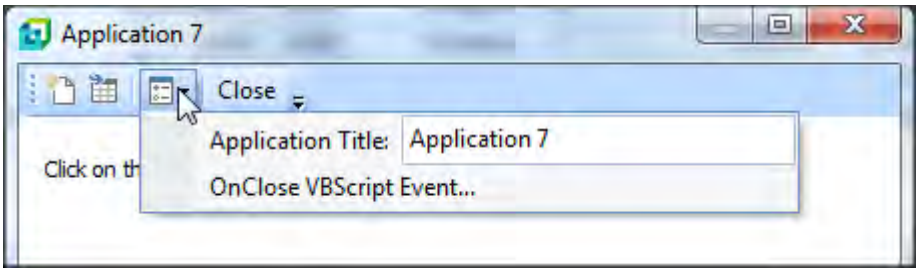


Figure 9-35: The options available under the *Options* button

Typing in a new *Application Title* and pressing the *Enter* key will immediately change the title at the top of the screen, and update the application builder menu. **Figure 9-36** shows the new *Application Title* being added, and **Figure 9-37** shows immediately after the operator has pressed the *Enter* key and the title of the pane changed.

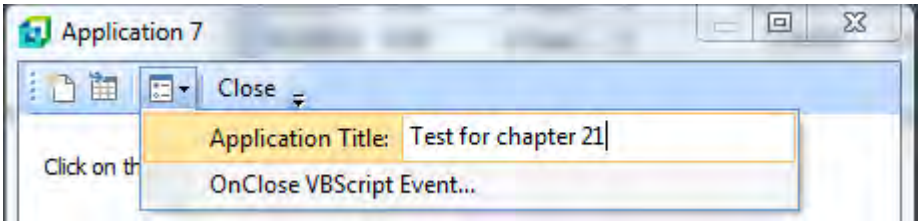


Figure 9-36: Entering the new title

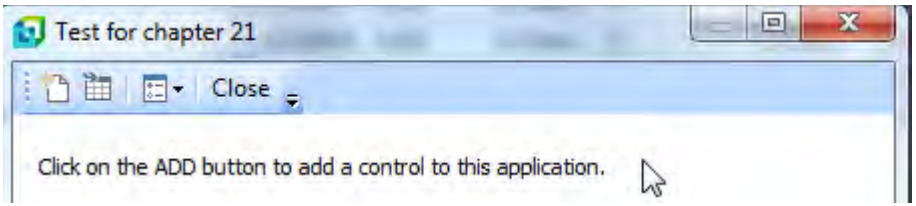


Figure 9-37: The updated title

At this point the entry in the *Application Builder* menu has also changed to reflect the new title.

The *OnClose VBScript Event* menu item calls up the *VBScript Editor* screen which contains a single *Form Event* called *OnClose* (see **Figure 9-38**). Selecting this creates the *AppBuilder_OnClose* function. This can be used to perform tasks such as writing out a log of what the operator has done, once the operator closes this application, or writing the information contained in a listview to a SQL Server table.

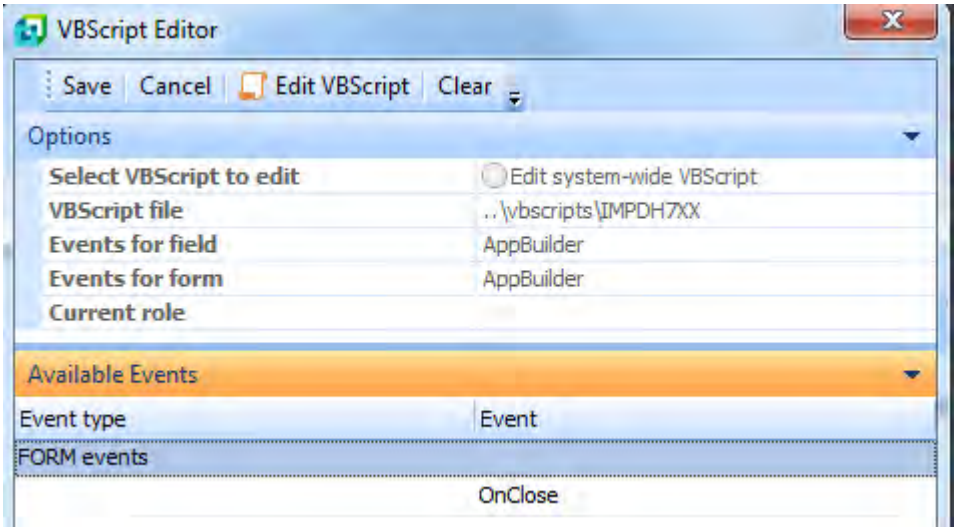


Figure 9-38: The *OnClose* form event

Each application in the *Application Builder* menu has its own *OnClose* script, so each can perform a different set of tasks. Within the *OnClose* function it is not possible to call another SYSPRO program or browse using either the *SYSPROProgramToRun* or *SYSPROBrowseToRun* variables.

Another use for the *OnClose* event is when you do not want the operator to exit the program until a certain task has been performed, or some other criteria has been met. The operator can be prevented from exiting the program by using the following line of code:

```
AppBuilder_OnClose = False
```

Below is some sample code to prevent the operator from closing the program if they have not populated the *Product Class* and *Alternate* fields:

```
Function AppBuilder_OnClose()
  If CostingForm.CodeObject.ProductClass = "" then
    msgbox "Please enter the Product Class"
    AppBuilder_OnClose = False
  End If

  If CostingForm.CodeObject.Alternate = "" then
    msgbox "Please enter the Alternate information"
    AppBuilder_OnClose = False
  End If
End Function
```

Adding Application Builder Programs to Other SYSPRO locations
Although you design the application builder applications under the *Application Builder* menu, you can access them from other locations. For example, they can be added to your *MyMenu* by specifying that a SYSPRO program is used, and supplying the application builder program name (***IMPDH1-9*** or ***IMPDHA-K***).

Application builder applications can also be added to the *Navigation pane* for an operator using the *Design Menus* button on the *Administration* tab of the SYSPRO Ribbon Bar.

An application builder application can also be called using the *SYSPROProgramToRun* variable within the VBScripting environment. This could be to capture batch information when the form originally loads/refreshes, or link to a button.

The application builder program name can also be added to a toolbar by supplying its program name.

Design System-wide View

The *Design System-wide View* option on the application builder menu enables you to design one application builder screen that will automatically be available to all operators. To be able to load the *Design System-wide View* program from the menu you must be in *Design Mode*. If you are not in design mode before calling up this program you will get the error message that appears in **Figure 9-39** that states that you are not in design mode at the system-wide level.

Design Mode is entered using the *Design UI Layouts* option on the *Administration* tab of the SYSPRO Ribbon Bar (see **Figure 9-40**). This displays the *Design Layout* screen that can be seen in **Figure 9-41**. The *System-wide* radio button must be selected on this screen, and the *Start Design Mode* button selected. When this is done the *Design Layout* screen is replaced by the *Design Mode in Progress* screen (see **Figure 9-42**).

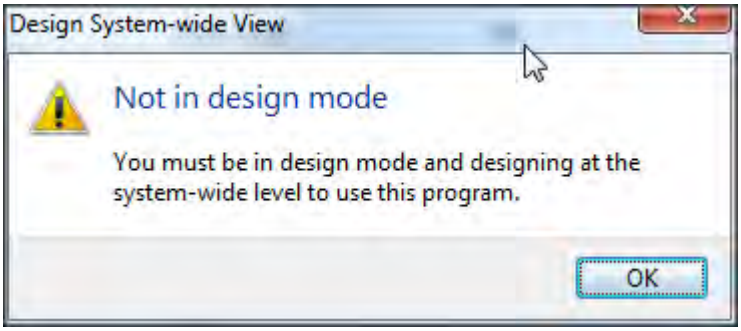


Figure 9-39: The message received when not in design mode

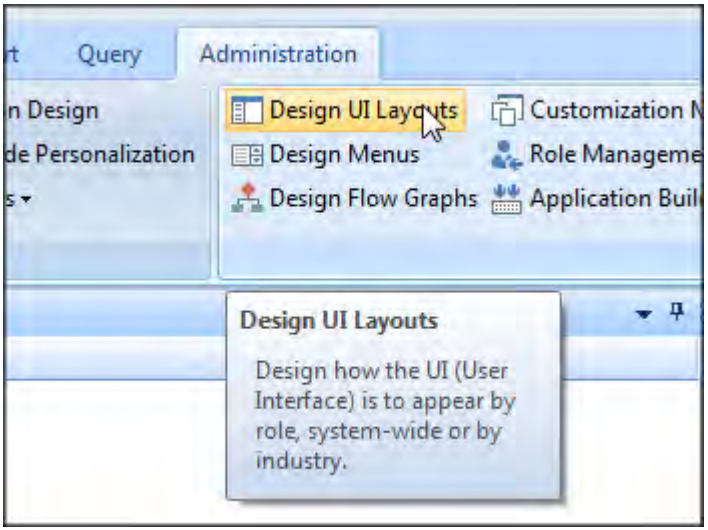


Figure 9-40: The *Design UI Layouts* button on the *Administration* tab of the Ribbon Bar

At this point you are in design mode, and you can select the *Design System-wide View* menu option. This calls up a blank application builder screen with the title *Application System-wide View*. The *Design Mode in Progress* screen will remain visible.

The *Application System-wide View* screen is the same as any other blank application builder screen except that it does not have the *Options* button on the toolbar. This means that you cannot change the applications title, and you cannot set VBScript code against an *OnClose* event.

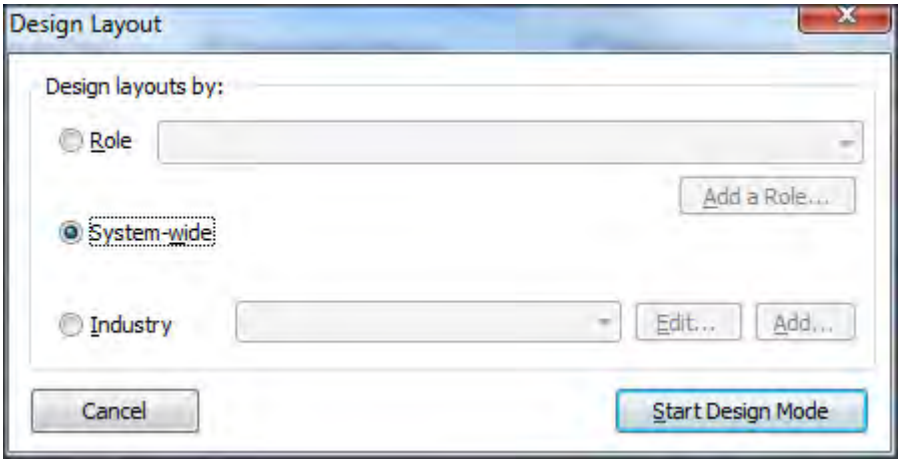


Figure 9-41: Selecting the *System-wide* radio button on the *Design Layout* screen

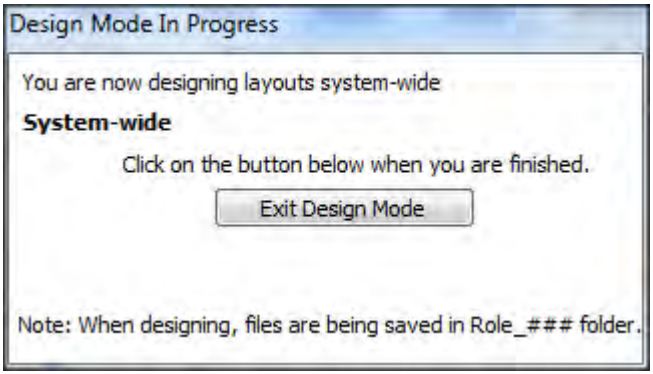


Figure 9-42: The *Design Mode in Progress* screen while in design mode

When you have finished designing the system-wide application you close it in the normal way, and then use the *Exit Design Mode* button on the *Design Mode in Progress* screen, which will close this screen. If you need to modify the system-wide application you will need to go back into design mode first.

The system-wide application will appear by default for existing and new operators. If the *Auto Hide* option is used so that the system-wide application is no longer on the desktop, it will minimize to a tab like any other screen.

If the system-wide application is closed by the operator, it can be re-opened using the *Show System-wide View* menu item on the *System-wide View* dropdown on the *Home* tab of the SYSPRO Ribbon bar (see **Figure 9-43**).

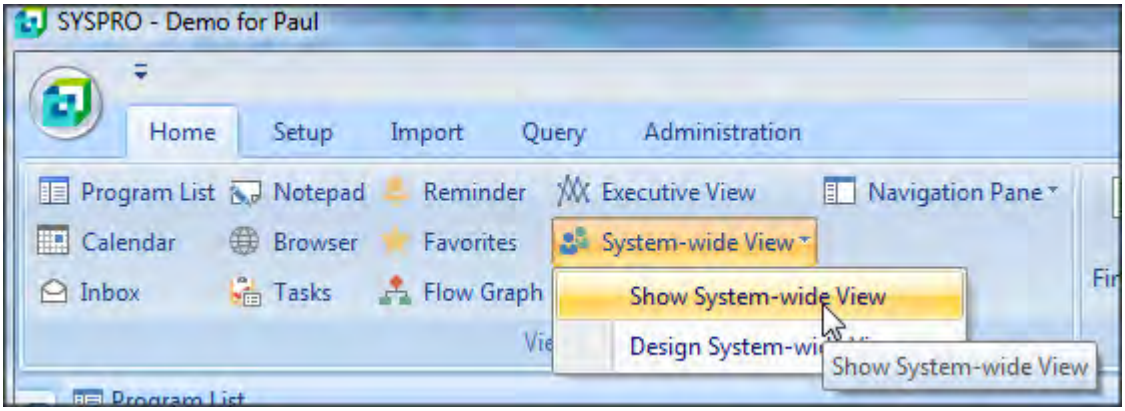


Figure 9-43: Re-opening the *System-wide View* after it was closed

An option exists against *System-wide Personalization* to force the *System-wide View* to be opened automatically when the operator logs in (*SYSPRO Ribbon Bar* | *Administration* tab | *Administration* section | *System-wide Personalization* button | *System-wide Personalization* pane | *Workspace* section). This can be seen in **Figure 9-44**.

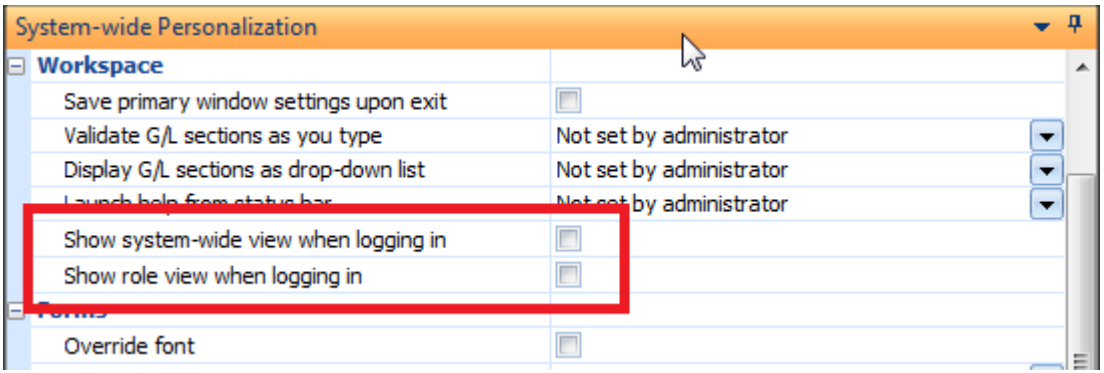


Figure 9-44: The options to show both the role and system-wide views when logging in

Alternatively it can be re-opened by selecting the *Menu* button on any pane that is open within the *SYSPRO Main Menu* and selecting *System-wide View* from the list of pane (see **Figure 9-45**).

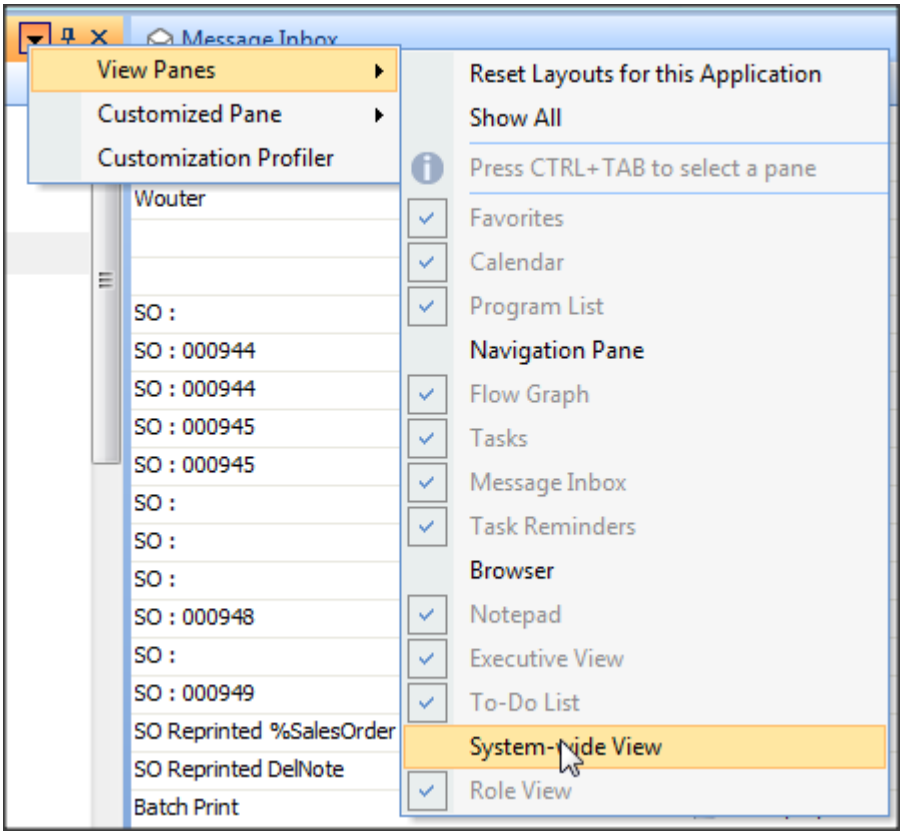


Figure 9-45: using the *Menu* button to re-open the *System-wide View*

Design Role View

The *Design Role View* option on the *Application Builder* menu works in a similar way to the *Design System-wide View* option, except that this is a single application that is common to all members of the selected role.

To be able to load the *Design Role View* program from the menu you must be in *Design Mode*. If you are not in design mode before calling up this program you will get the error message that appears in **Figure 9-46** that states that you are not in design mode at the role level.

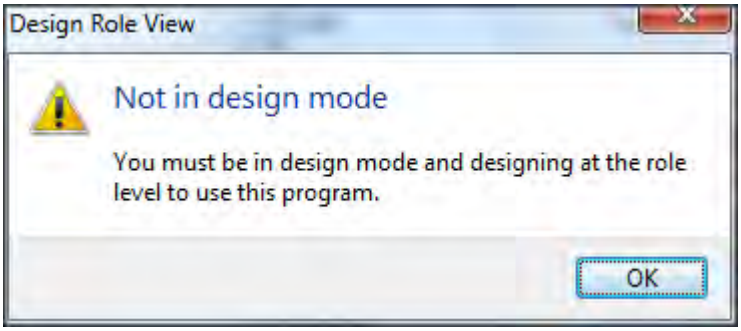


Figure 9-46: The message received when not in design mode

Design Mode is entered using the *Design UI Layouts* option on the *Administration* tab of the SYSPRO Ribbon Bar (see **Figure 9-40**). This displays the *Design Layout* screen that can be seen in **Figure 9-47**. The *Role* radio button must be selected on this screen, and the required role selected from the dropdown list alongside it. Click on the *Start Design Mode* button. When this is done the *Design Layout* screen is replaced by the *Design Mode in Progress* screen which displays the chosen role (see **Figure 9-48**).

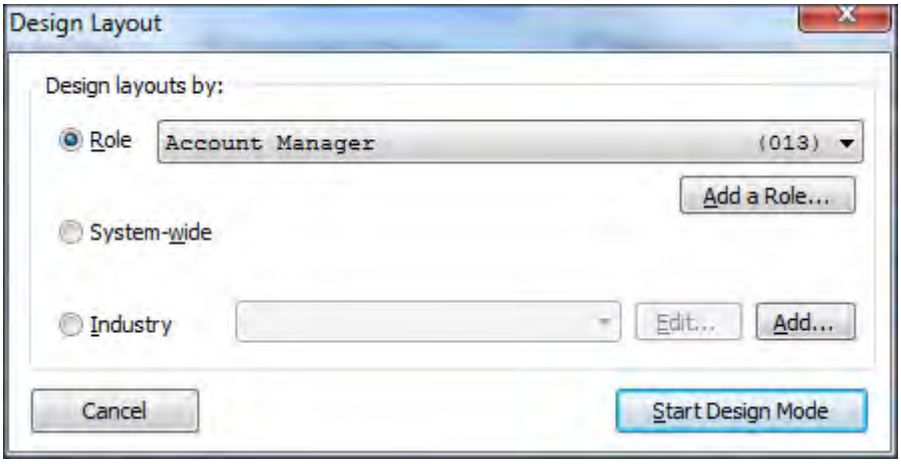


Figure 9-47: Selecting the *Role* radio button and role on the *Design Layout* screen

At this point you are in design mode, and you can select the *Design Role View* menu option. This calls up a blank application builder screen with the title *Application Role View*. The *Design Mode in Progress* screen will remain visible.

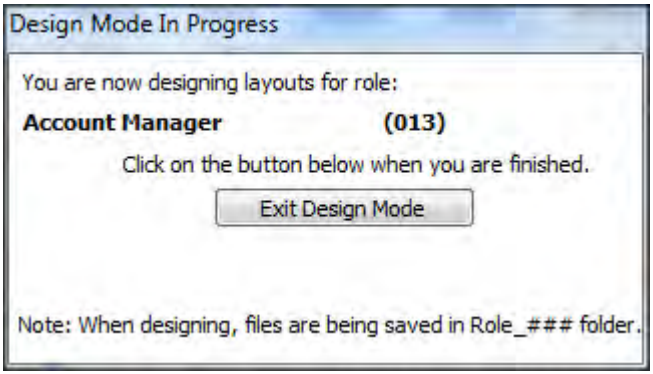


Figure 9-48: The *Design Mode in Progress* screen when designing for a role

The *Application Role View* screen is the same as the blank application builder screen for a system-wide application in that it does not have the *Options* button on the toolbar. This means that you cannot change the application's name, and you cannot set VBScript code against an *OnClose* event.

When you have finished designing the role application you close it in the normal way, and then use the *Exit Design Mode* button on the *Design Mode in Progress* screen, which will close this screen. If you need to modify the role application you will need to go back into design mode first.

The *Role View* application will appear by default for existing and new operators that are members of this role. If the *Auto Hide* option is used so that it is no longer on the desktop, it will minimize to a tab like any other screen.

If the role application is closed by the operator, it can be re-opened using the *Show Role View* menu item on the *Role View* dropdown on the *Home* tab of the SYSPRO Ribbon bar (see **Figure 9-49**).

An option exists against *System-wide Personalization* to force the *Role View* to be opened automatically when the operator logs in (*SYSPRO Ribbon Bar* | *Administration* tab | *Administration* section | *System-wide Personalization* button | *System-wide Personalization* pane | *Workspace* section). This can be seen in **Figure 9-44**.

Alternatively it can be re-opened by selecting the *Menu* button on any pane that is open within the *SYSPRO Main Menu* and selecting *Role View* from the list of pane (see **Figure 9-50**).

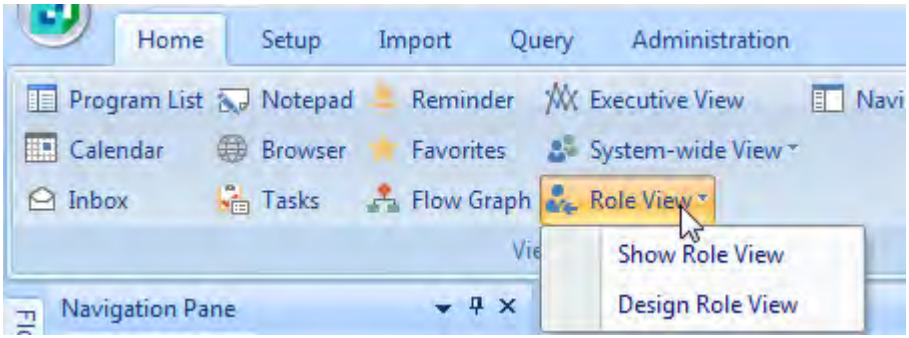


Figure 9-49: Re-opening the *Role View* application after it was closed

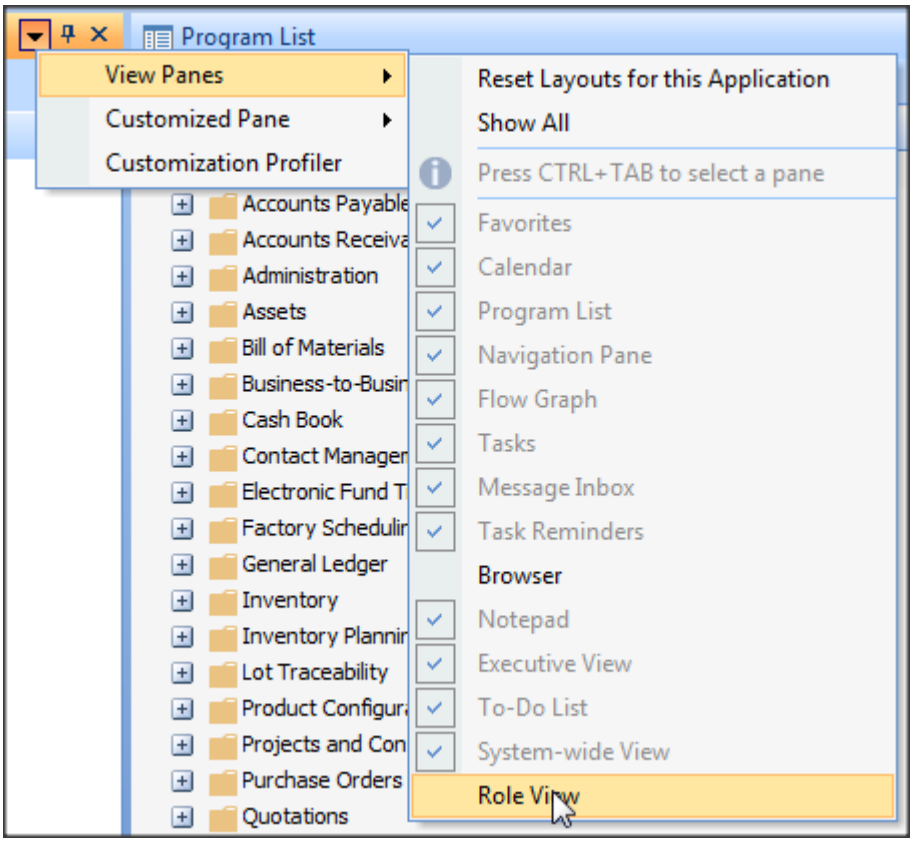


Figure 9-50: Re-opening the *Role View* application after it was closed

Volume 2 (Chapters 10 – 21)

Chapter 10 - Customized Panes

Customized Panes are a means of creating your own panes directly within a SYSPRO program. There are ten different pane types to choose from (*Graph*, *Listview*, *Web browser*, *SRS*/*Crystal report*, *PDF Viewer*, *Rich text notepad*, *.NET User Control*, *Form*, *Search Window*, and *Executive Dashboard*), and each will be covered in detail in a later chapter. This chapter covers the items that are common to all customized pane types.

Most customized panes will reside within a SYSPRO program. A single customized pane can be added to a program to perform a specific task, or many can be added so that they work in unison. They can be configured to integrate with the program in which they reside, or they can be completely independent, and just reside in the program because that is where the operator spends the majority of their time.

As a customized pane is a docking pane, it can be dragged from its current location and left to float in a more suitable location, or even minimized down to a tab. If they are embedded within a SYSPRO program they will be opened when the corresponding program opens, and closed when it closes.

Customized panes don't need to reside within a SYSPRO program; they can also reside within the *SYSPRO Main Menu* screen. If these are dragged from their current location they can be left floating anywhere on the operator's screen. The operator can interact with these floating customized panes, even when they have other programs open, and they will remain open until SYSPRO is closed or the operator closes them.

Customized panes can be easily transported between systems because they can be exported to a text file, and these text files can be imported into the new system. This means that an administrator can create a customized pane on a test system, and import it into their live system once they are happy with it. A third party developer can also develop one or more customized panes for a customer, and they (or the system's administrator) can import the pane on the customer's site. Note that customized panes are not necessarily able to be taken between versions of SYSPRO. A customized pane exported from a SYSPRO 6.1 system cannot be imported into a SYSPRO 7 system.

Each customized pane can have its own VBScript code associated with it, making it extremely flexible. The VBScripting can be used to call SYSPRO's business objects to process and retrieve information. When called correctly from within the customized panes, business objects do not need to be licensed. The ability to embed a *.NET User Control* into a customized pane means that skilled developers can create new functionality, and this can be done in such a way that it appears to the operators as if it is part of SYSPRO.

Using role-based security in conjunction with the customized panes, operators (or groups of operators) can have completely different working environments and different user experiences from each other. Note that role-based security is not applied to customized panes added to the SYSPRO Main Menu.

Customized Pane Templates

Of the ten types of customized panes, three (*Graph*, *Listview*, and *Executive Dashboard*) have the option of selecting from fully functional pre-built templates. As they do not integrate with any SYSPRO program by default, customized panes created using these templates can be created against either the *SYSPRO Main Menu* screen, or within any SYSPRO program.

Figure 10-1 shows one of the *Executive Dashboard* customized pane templates that has been created against the *SYSPRO Main Menu* screen. It was dragged from its default location and left to float to the side of SYSPRO.

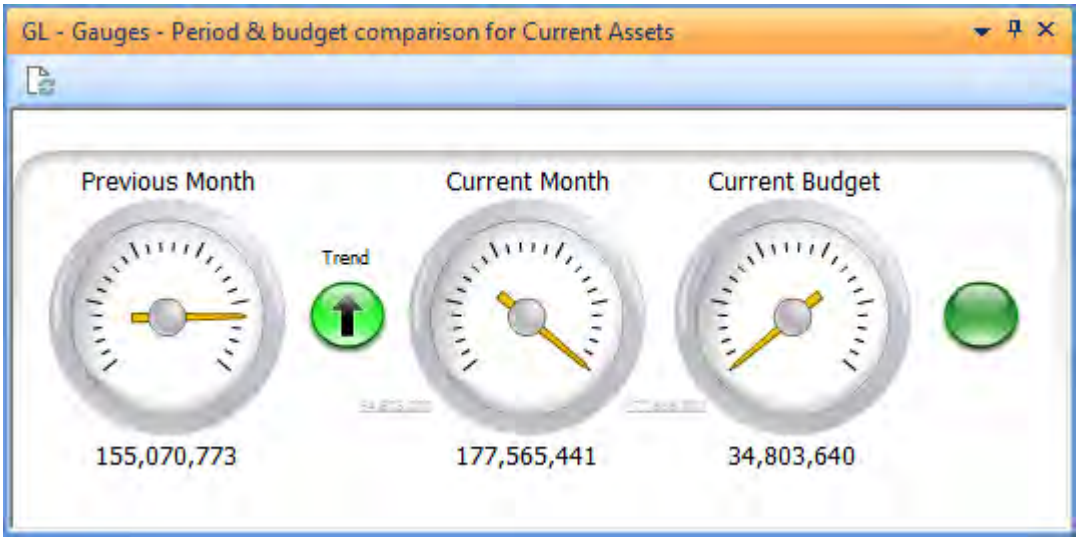


Figure 10-1: One of the *Executive Dashboard* customized pane templates

Adding a Customized Pane

Customized Panes can be added to a SYSPRO program, or the *SYSPRO Main Menu* screen. They are added by selecting the *Menu* button against one of the panes in the program, or the *SYSPRO Main Menu* screen, and selecting *Customized Pane* | *New* from the displayed menu (see **Figure 10-2**).

Of the other items on this menu, the *Create Editable Form* and *Create Display Form* options are covered within Chapter 14 on *Forms*, while the *Export Customized Panes* option will be covered within the *Exporting a Customized Pane* section below, and the *Import Customized Panes* option will be covered in the *Importing a Customized Pane* section below.

The *Add Enterprise Search* and *Add Tutorials Player* options are templates that create an enterprise search customized pane, and a tutorial player customized pane. Both of these use *.NET User Controls*.

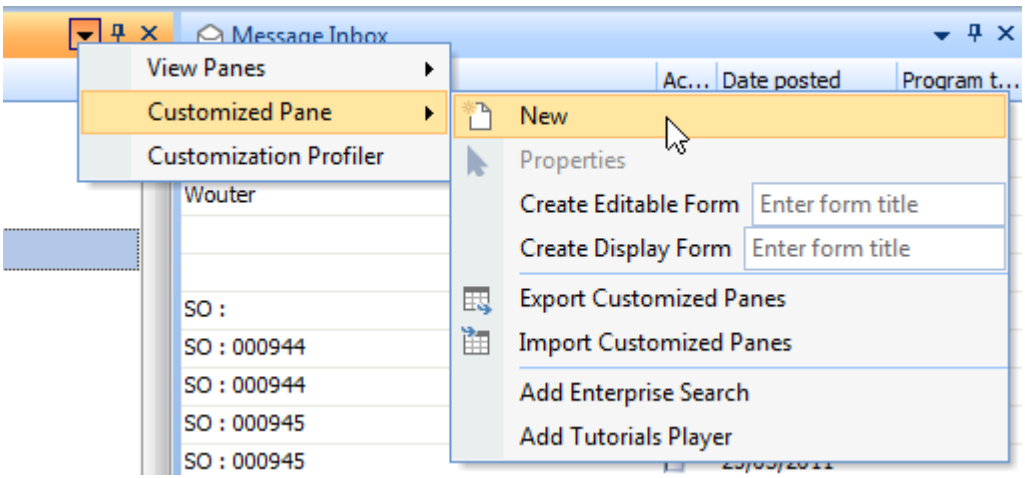


Figure 10-2: Adding a customized pane

When you add a new customized pane the *Customized Pane Editor* screen is displayed. By default the *Object type* will be *Graph*, so the screen will consist of a toolbar, a *Pane Properties* pane on the left, a *Preview* pane on the right (containing *Preview*, *VBScript*, and *Customized Panes* tabs), and a *Graph* pane at the bottom containing graph templates.

Toolbar

The *Toolbar* consists of six buttons, *Save*, *Save and Exit*, *Preview*, *Edit VBScript*, *Edit Window Title*, and *Design Form*. The *Save* button is used to save the customized pane, but remain within the *Customized Pane Editor*, so that you can create another one. The *Save and Exit* button is used to save the customized pane and exit the editor.

The *Preview* button has a dropdown list consisting of *Preview* and *Auto Save Templates* options. The *Preview* option enables you to see the effects of the changes that you have made to the customized

pane, without having to save these changes. It populates the *Preview* tab using the settings selected, after running the scripts associated with this pane.

The *Auto Save Templates* option is useful if you want to add multiple customized panes using the standard templates. If this option is checked, double-clicking on the template name will add the customized pane, leave you in the *Customized Pane Editor*, and position the cursor back on the *Window title* field so that you can continue adding customized panes. Double-clicking on a second template will repeat the process. The *Auto Save Templates* option is always unchecked when you load the *Customized Pane Editor*.

The *Edit VBScript* button calls up the *VBScript Editor* screen where you can add VBScript against the macro events. See the *VBScript Editor Screen* section for more information.

The *Edit Window Title* button is only enabled if you are editing an existing customized pane. It enables you change the title that appears against the customized pane. However, this is also the customized pane's name. When you attempt to change the pane's title (after it has been saved) you will receive a notification that changing the pane's title may require you to change other code that interacts with this pane (see **Figure 10-3**)

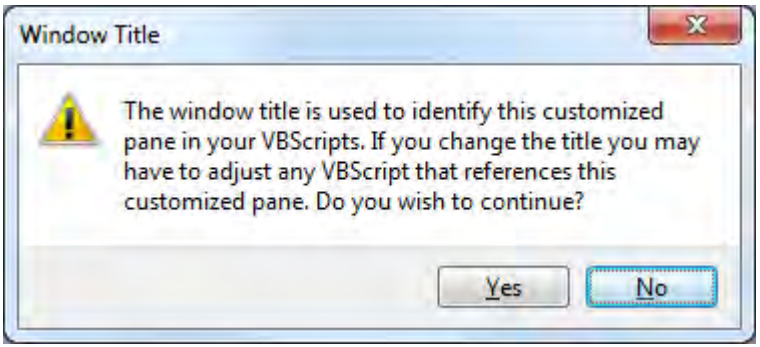


Figure 10-3: The notification when you attempt to change a customized pane's title

If any other panes contain a line of code that cause this customized pane to refresh, after changing this customized pane's title/name you will need to call up the other pane and edit its VBScript so that when it refreshes the customized pane it references it by its new name. Failure to make this change will display an error similar to that in **Figure 10-4** when the code attempts to refresh the pane using the old pane title.

The *Design Form* button is only enabled if the *Object type* for this customized pane is set to *Form* and calls up the *Design Form* screen where you can design your form. The *Design Form* screen will be covered in the *Form* section.

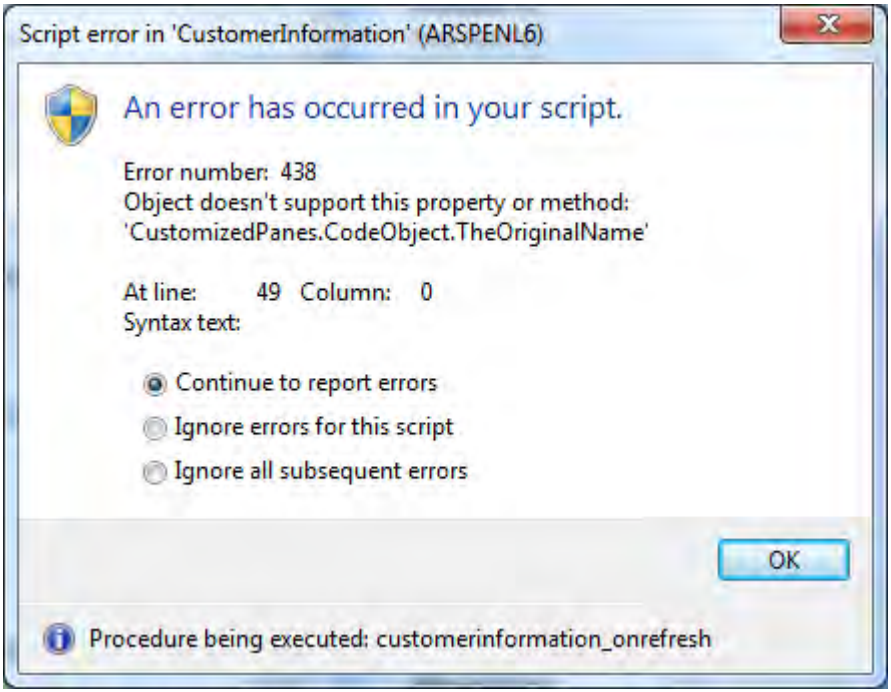


Figure 10-4: The VBScript error message when attempting to refresh a pane using the incorrect name

Pane Properties

The *Pane Properties* pane always contains the same options, no matter which *Object type* has been selected. However, in some cases options may be greyed-out/disabled.

A *Pane Properties* pane appears in **Figure 10-6**, and consists of four options at the top of the screen, and six sections. Each of these will be covered in detail below.

Window Title

The *Window title* is both the description that appears at the top of the customized pane, and the name that is used to programmatically access it. In **Figure 10-6** the *Window title* is *ROI per Stock Code*, and this can be seen as the title for the customized pane in **Figure 10-5**.

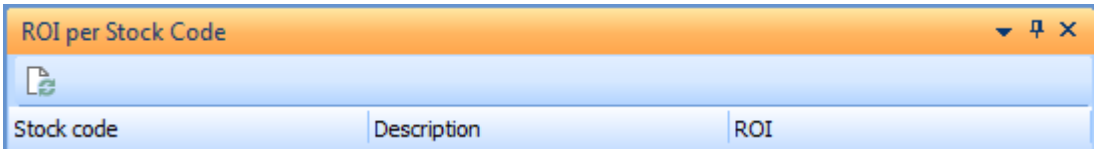


Figure 10-5: The *Window Title* against a customized pane

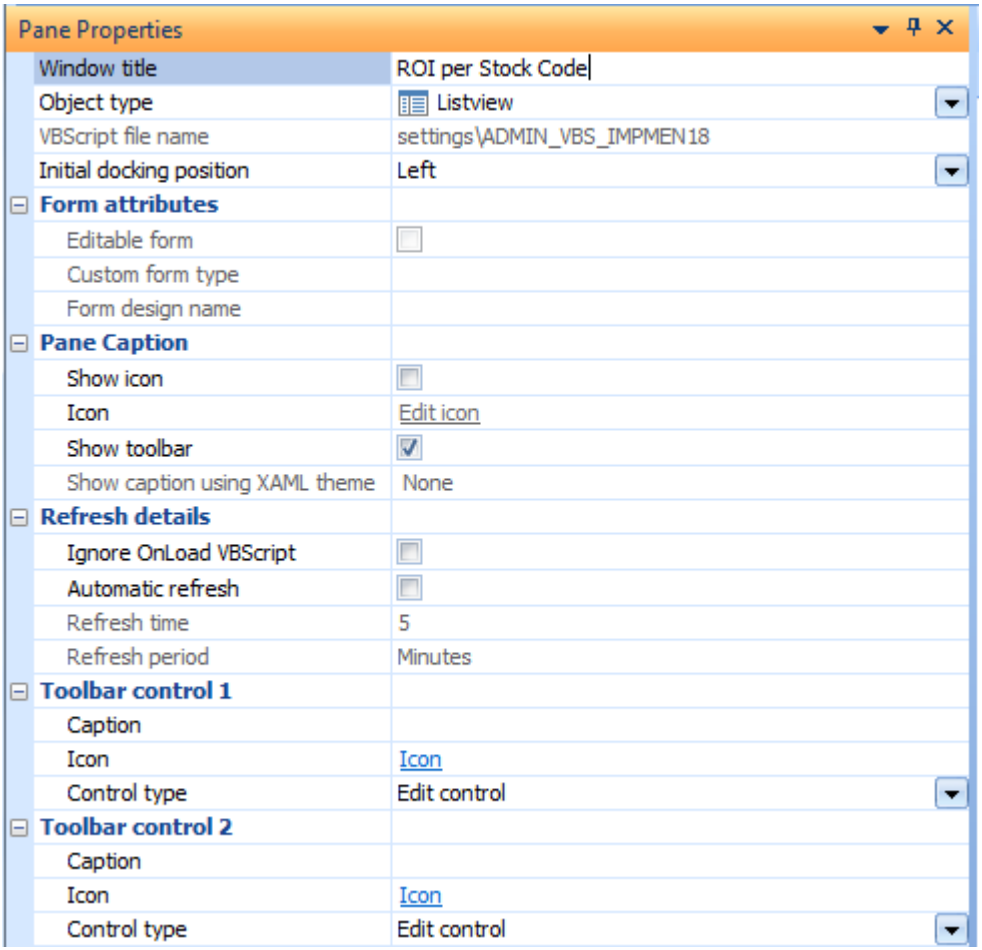


Figure 10-6: The *Pane Properties* pane

A customized pane can have its *OnRefresh* event fired by other panes within the same program. This is done using VBScript against the other pane (which could be another customized pane or a normal pane such as a display form, entry form, or listview). When editing the VBScript of the pane that is to cause the refresh event on the customized pane, expanding the *CustomizedPanes* section of its *Variables* pane lists all the customized panes associated with the current program. In **Figure 10-7** the customized pane name *ROIperStockCode* can be seen within the *CustomizedPanes* section.

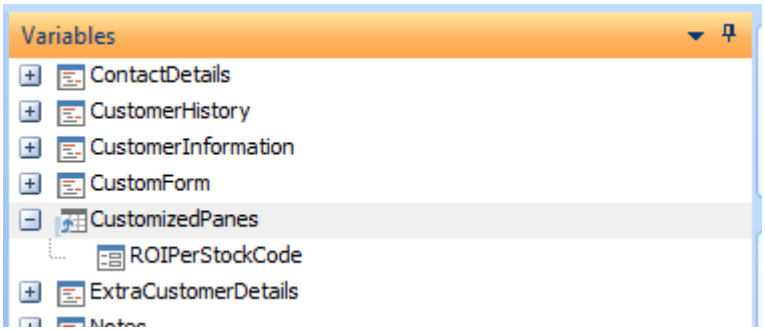


Figure 10-7: The *ROIperStockCode* customized pane variable name

Double-clicking on one of the customized pane names starts a wizard that ultimately inserts VBScript code for you. Using the *ROI for Stock Code* customized pane as an example, and the default settings of this wizard, the line of code that would be inserted appears below. This line of code specifies the name of the customized pane to be refreshed.

```
CustomizedPanes.CodeObject.ROIPerStockCode = "doRefresh"
```

Changing the *Window title* for a customized pane will display an information message informing that making this change may require other scripts to be changed (see **Figure 10-3**).

If the *Window title* of the customized pane is changed, any lines of code that cause this customized pane to refresh will attempt to refresh a customized pane name that no longer exists, and an error message to this effect will be displayed if the line of code is executed (see **Figure 10-4**).

Object Type
The *Object type* option contains a dropdown list of the ten possible *Object types* (see **Figure 10-8**). This dropdown list is available up until the customized pane is saved for the first time. Once a customized pane has been saved its *Object type* cannot be changed. The object type *of Executive Dashboard* will only appear if the *Executive Dashboards* have been licensed.

VBScript File Name
The *VBScript file name* displays the name and location of the VBScript file associated with this customized pane. It is a display only field, so the script name cannot be changed. The location of the VBScript file will depend on whether the customized pane is being created for an operator, or for a role. The first six letters of the filename relate to the SYSPRO program name in which it resides. For example, if the customized pane resides in the *Accounts Receivable Customer Query* program, the first six characters of the customized pane script name will contain the name of the *Customer Query* program, ***ARSPEN***. If the customized pane is associated with the *SYSPRO Main Menu* screen the first six characters of the script name will be ***IMPMEN***.

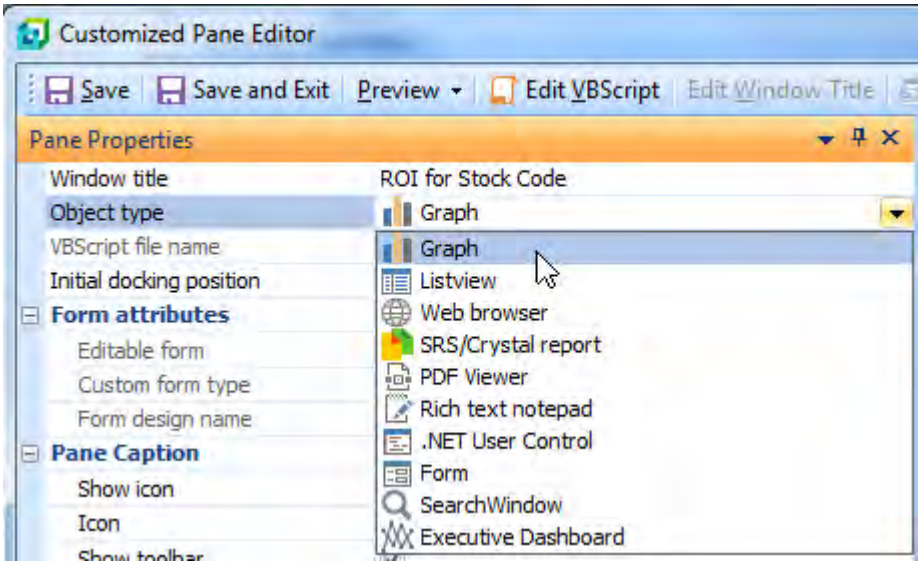


Figure 10-8: The *Object type* dropdown list

If this customized pane is being created for an operator, the VBScript file will be stored in the SYSPRO `Base\Settings` folder. If the customized pane is being created for a role the VBScript file will be stored in a role folder under the SYSPRO `Base\Settings` folder. Each role has a three digit number associated with it when it is created. The folder used for this VBScript will contain *Role_* followed by this role number. In **Figure 10-9** the customized pane is being added to the *Chief Executive Officer* role, which on this system is role number *001*. The VBScript is written to the `Base\Settings\Role_001` folder.

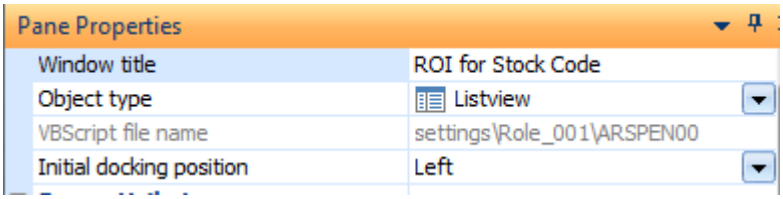


Figure 10-9: *Pane Properties* pane showing the VBScript file name for the *Chief Executive Officer* role

Initial Docking Position

The *Initial docking position* field has a dropdown list against it containing the options *Left*, *Right*, *Top*, and *Float*. This is the initial location where you would like the customized pane to appear. These relate to down the far left of the program, down the far right of the program, across the top of the program, or free floating. This only relates to the initial location when the customized pane is created, because once it has been created the operator can move the customized pane to a new location.

Once the customized pane has been saved this field is a display only field, and cannot be changed.

Form Attributes

The *Form attributes* section contains the *Editable form* checkbox, the *Custom form type* field, and the *Form design name* field. As the *Form attributes* section name infers, the options within this section are only relevant when the customized pane's *Object type* is *Form*.

The *Editable form* checkbox specifies whether this form will be used to just display information, or if the operator can also use it to capture information.

Custom Forms enable non-SYSPRO information to be stored against many of the key fields used in SYSPRO, such as against stock codes, customers, and sales orders. Each of these key fields has a custom code associated to it (*STK* for stock codes, *CUS* for customers, *ORD* for sales orders, etc.). An *Associated Pane* is available called *Custom form* that is used to display all of the custom form fields (and their contents) linked to this key field. Associated panes are customized panes that have been configured to automatically refresh when this key field changes its value.

If this *Custom form* associated pane has been inserted into a program, it can be edited in the same way as any other customized pane. When it is edited the *Custom form type* field will display the custom code to which it is linked, which cannot be changed. This code is also available using the *CustomFormType* variable within the *CustomizedPane* section of the *Variable* pane.

The design/layout of a form can be performed in several ways. It can be done programmatically in the VBScript, or it can be designed using the *Design Form* screen. In both cases XML is used to build/store the design. If the *Design Form* screen has been used to create the form, the *Form design name* field contains the name and location of the XML containing this design. This is maintained by the system and cannot be changed.

Pane Caption

The *Pane caption* section is used to specify the look and functionality of the area at the top of the customized pane. It contains four options, *Show icon*, *Icon*, *Show toolbar*, and *Show caption using XAML theme*.

The *Show icon* checkbox enables you to display an icon in the header of your customized pane. If this is checked the *Edit icon* hyperlink against the *Icon* field becomes enabled, and you use this to choose from the standard icons that are provided with SYSPRO.

The *Show toolbar* checkbox causes the customized pane's toolbar to be shown/hidden. By default the toolbar contains a *Refresh* button, but this can be configured to also contain customized buttons, checkboxes, and an entry field.

The *Show caption using XAML theme* option enables you to apply a XAML theme to the section immediately below the toolbar, if one is to be shown. Placeholder *%1%* will be populated with the contents of the *Window title* field. This option is only available for Form and Search Windows customized panes. For all other types this option is greyed-out.

Figure 10-10 shows a customized pane called *F G* with the entire range of *Pane caption* items configured. The *Window title* of *F G* appears right at the top of the customized pane, and alongside that is icon that appears against the *Icon* prompt. The toolbar appears below this, and has been configured to have an input field and a button on it. Each of these has been configured to have an icon. The default *Refresh* button is also present on the toolbar.

The next section down the screen uses the *Orange* XAML theme. The *F G* text is populating its *%1%* placeholder, and this happens automatically.

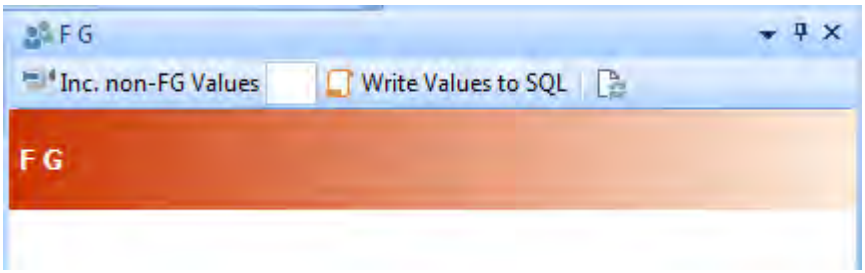


Figure 10-10: A customized pane showing the *Pane Captions* in use

Refresh Details

The *Refresh details* section contains four options*, Ignore OnLoad VBScript*, *Automatic refresh*, *Refresh time*, and *Refresh period*.

The *Ignore OnLoad VBScript* checkbox is used to prevent the *OnLoad* macro event from firing as the program loads. However, the *OnLoad* macro event will be fired when the *OnRefresh* macro event fires for the first time for this run of the program. The code against the *OnLoad* function will be run immediately before the code against the *OnRefresh* function is run. Typically this would be checked when the customized pane would take a significant time to load (because it is calling a business object, SQL query, etc. that is performing lots of processing) and the operator calls up this program frequently. If this were unchecked in this scenario, the operator would have to wait for the customized pane to finish loading before being able to continue working each time that the program was loaded.

The *Automatic refresh* checkbox is used to specify that this customized pane should automatically fire its *OnRefresh* event at a set interval. The *Refresh time* field can be configured with a value from 0 to 9999999, and the *Refresh period* can be set to *Minutes* or *Seconds*. It is strongly suggested that you

do not use a low value and *Seconds*, because the performance of your workstation may be adversely affected.

Toolbar Control 1 and 2

Toolbar control 1 and *Toolbar control 2* both work in the same way, so will be covered together. A customized pane can have two toolbar controls. These can be any combination of checkboxes, entry fields, or buttons. Each toolbar control has a *Caption* (without a *Caption* the control will be ignored), and optionally an *Icon*. The *Control type* is a dropdown list where you specify if this control is a *Button*, a *Checkbox*, or an *Edit control* (which is a text entry field).

Figure 10-11 shows a section of the *Pane Properties* pane that contains *Toolbar control 1* and *Toolbar control 2*. These are the settings that were used to create the customized pane that appears in **Figure 10-10**.

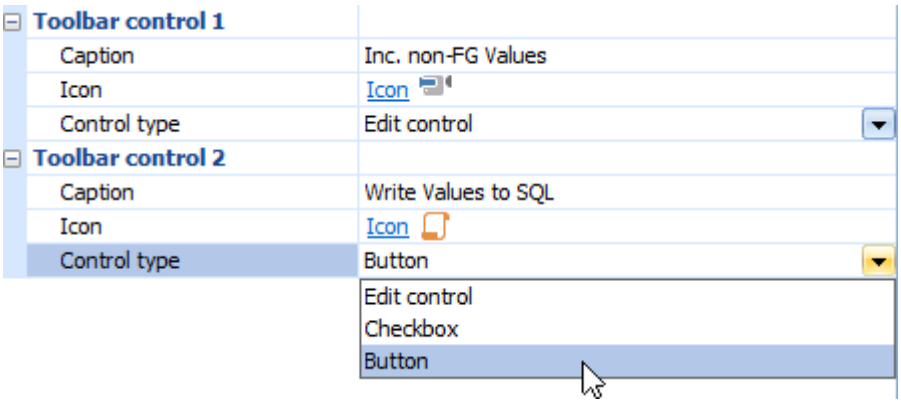


Figure 10-11: The Toolbar controls

Templates

Templates are fully functional customized panes, and are only available when the *Object type* is set to *Graph*, *Listview*, or *Executive Dashboard*. They use business objects to retrieve the required data. In nearly all cases the displayed values are from your data; they do not use sample data.

The title of the pane at the bottom of the screen will match the selected *Object type* within the *Pane Properties* pane. If the customized pane *Object type* is set to *Graph*, *Listview*, or *Executive Dashboard*, the pane at the bottom of the screen will become a listview, and contain all of the available templates for this object type. If the *Object type* is *Graph*, the title of the pane at the bottom of the screen is *Graph*, and the listview contains a list of all of the graph templates (see **Figure 10-12**).

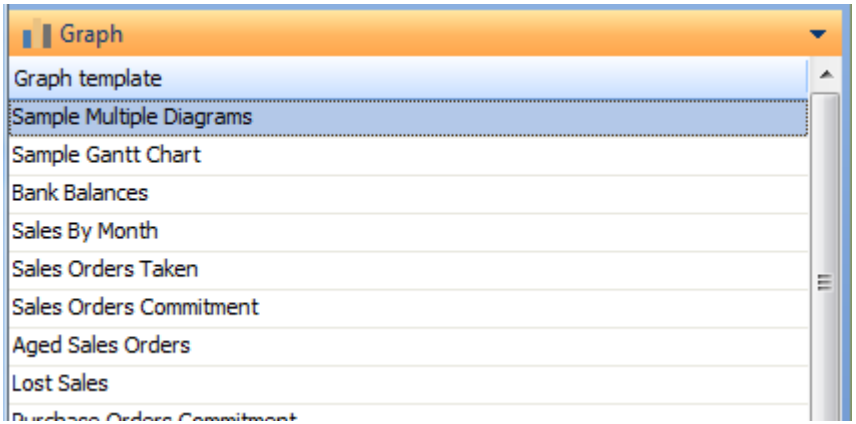


Figure 10-12: Some of the available templates when the *Object type* is *Graph*

These templates do not communicate with the program within which the customized pane resides, so can be added to any program, or the *SYSPRO Main Menu* screen.

Before a customized pane is saved for the first time, you can change the *Object type* for this customized pane, and the list of templates will change to match the new *Object type*. Once the customized pane has been saved you cannot change its *Object type*.

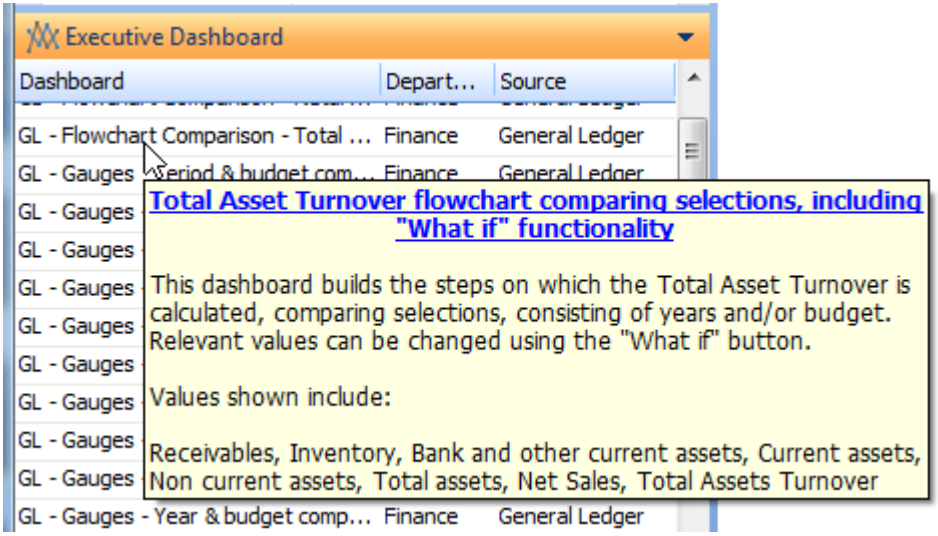


Figure 10-13: A template tooltip

Moving the mouse pointer over a template displays a tooltip that explains what the template does, and what it displays (see **Figure 10-13**).

When you double-click on one of these template names the VBScript for this customized pane is updated with the code from the *Template*. The *OnLoad* function of the script is executed, and the *Preview* tab shows the results using your data. Depending on what the code has to perform, and the amount of data associated with this, the preview may take a while to populate.

Most of the *OnLoad* functions of these templates contain a line of code to call their *OnRefresh* function. This is because the *OnLoad* function is used to define the look/feel/style of the output, and the *OnRefresh* populates it with the values.

By calling the *OnRefresh* function from the *OnLoad* function;

- The customized pane is always populated when it loads
- You don't need to have the code to retrieve/populate the values in two places, with the potential of making a change to one and not the other
- The pane can be refreshed (using the Refresh button, from another pane, or automatically after a certain period of time) without having to reload it

If you double-click on another *Template* name the VBScript will be overwritten with the code from this new template, this code will be executed, and the *Preview* tab will be populated with the results. You can do this as many times as you like. You can also change the *Object type* and double-click on a *Template* of this type. The same process will happen, where the code is executed and the *Preview* tab populated.

After using the *Edit VBScript* button to call up the *VBScript Editor* and you return back to the *Customized Pane Editor*, if you double-click on another template name you will receive a warning that continuing will cause your script changes to be lost (see **Figure 10-14**).

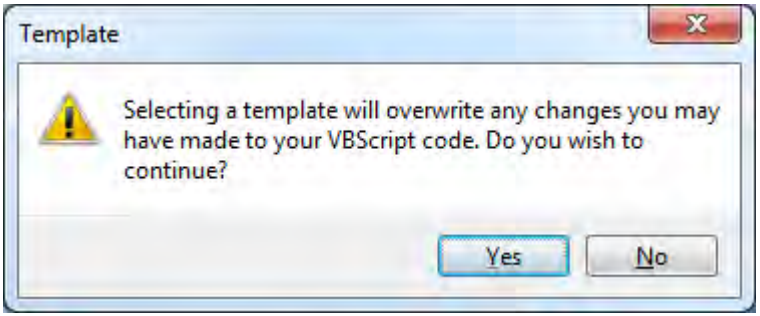


Figure 10-14: The message displayed if you double-click on a template after using the VBScript editor

Once you have saved your customized pane, and exited the *Customized Pane Editor* screen, if you edit the customized pane the templates are no longer available, and the listview will be greyed-out.

Preview Pane / Preview Button

When adding a new customized pane, or making changes to an existing one, the *Preview* tab provides a means of seeing the results of the changes without having to save the pane (and the consequent undoing of the changes if they are not what is required). It also provides a means of seeing the output of *Templates*, using your data, without having to save and exit the customized pane.

Clicking on the *Preview* button on the toolbar causes the *OnLoad* function of the VBScript to be executed, and uses the settings that have been configured against the *Pane Properties* pane.

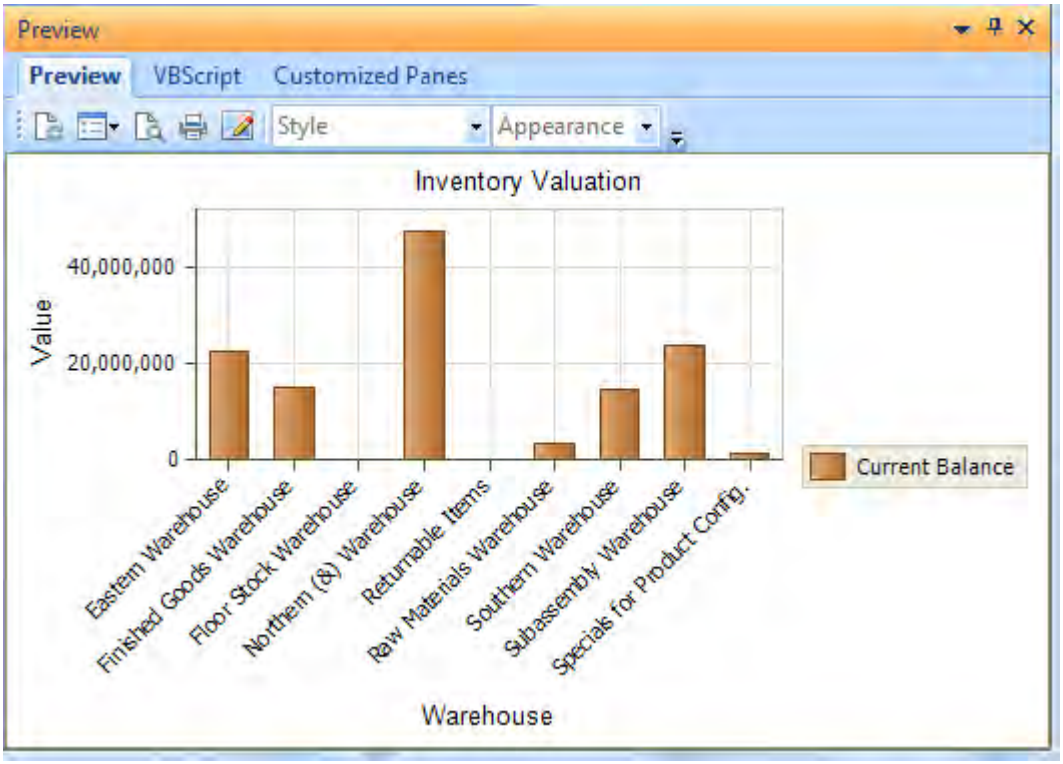


Figure 10-15: The *Preview* tab showing the output of the *Inventory Valuation* graph template

When adding a customized pane the templates are available for the *Object type* of *Graph*, *Listview*, and *Executive Dashboard*. If you double-click on one of these templates, any code against the *OnLoad*

function of the VBScript associated with this template is executed, and the *Preview* pane is populated with the output as if the customized pane had been saved and executed (see **Figure 10-15**).

The toolbar of the *Preview* pane will contain different options depending on the *Object type* selected. This is the same toolbar that will appear in the customized pane if the pane were saved, so will be covered within each *Object type's* section.

VBScript Tab

The *VBScript* tab displays the VBScript associated with the current customized pane. This is only a display, and the script cannot be edited in this tab. It enables you to view the script (or understand how a customized pane is working) without having to call up the editor, which means that there is no chance of making changes to the script by accident.

If you are adding the customized pane and double-click on a template, the *Preview* tab will be brought to the front and populated. If you click on the *VBScript* tab it will contain the script that matches the template.

Customized Panes Tab

The *Customized Panes* tab is a means of seeing all customized panes for this operator. To improve the speed of loading the *Customized Pane Editor*, the listview is not populated by default. The *Customized Panes* tab has its own toolbar. The first button on the toolbar is the *Refresh* button, and this retrieves the details about the customized panes and populates the listview.

Figure 10-16 shows the listview on the *Customized Panes* tab after the *Refresh* button has been selected. It contains the operator code, title of the pane, the type of customized pane, the name of the VBScript associated with this pane, and the program name in which it is used (where one starting with *IMPMEN* refers to one of the panes of the *SYSPRO Main Menu* screen).

Other items on the toolbar include a *Print* button to print the contents of the listview, and an *Export to Excel* button that creates an HTML file in SYSPRO's `Base\Settings` folder (using the operator code for its name) and opens the HTML file using *Excel*.

The *Search* button looks through all of the columns for all of the rows and filters out those rows that do not contain this value. If multiple values are added to the *Search*, all of these values must exist somewhere on the row for it to appear. The *Search* is not case-sensitive. Using **Figure 10-16**, entering the text *Form Z* would return multiple rows because the word *Form* appears in multiple rows, and *Z* appears in the *Used in* column for these rows. Changing the *Search* to *Form ZZ* would only return the

second row as this has the word *Form* in the *Object type* column, and *ZZZZZ* in the *Window title* column.

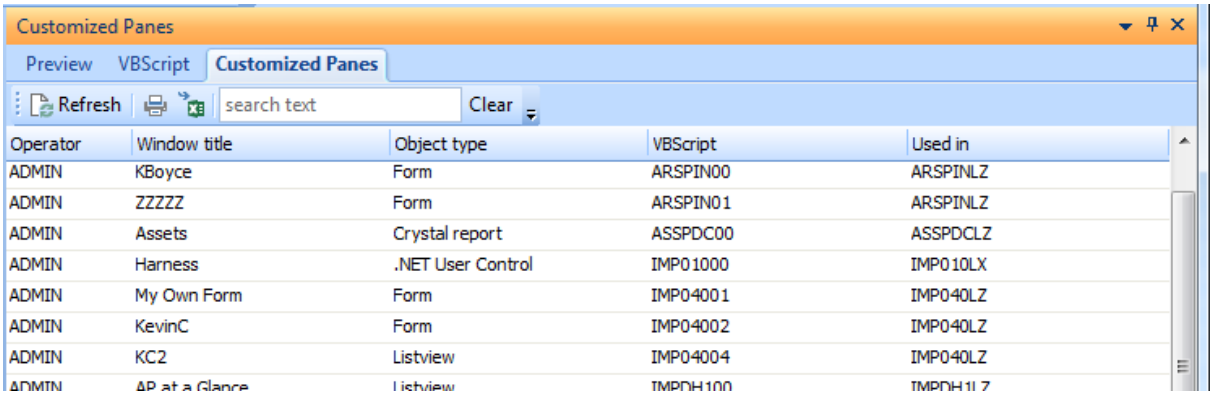


Figure 10-16: A populated *Customized Panes* tab

This listview also has the normal functionality of a listview, such as being able to click on a column header to sort the output by the contents of this column.

VBScript Editor Screen

When the *Edit VBScript* button is selected on the *Customized Pane Editor* screen's toolbar, the *VBScript Editor* screen is displayed. This screen is divided into two sections, *Options*, and *Available Events*. The *Available Events* section of the screen is specific to the object type of the customized pane being edited so will be covered within those sections.

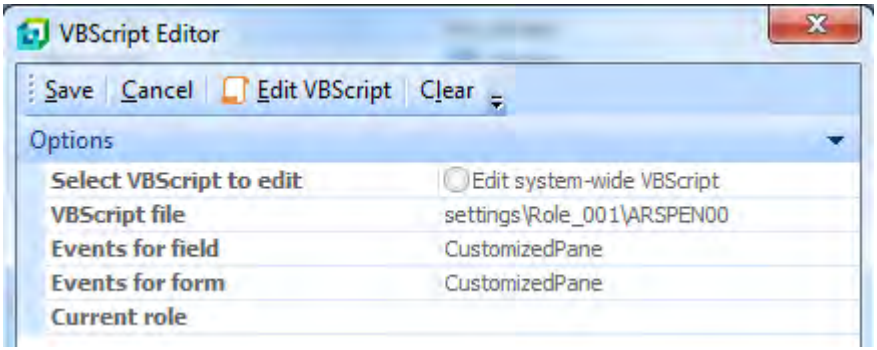


Figure 10-17: The *Options* section of the *VBScript Editor* screen

When editing a script for a standard SYSPRO form, one of the fields will already have been selected, so the *Events for field* and *Events for form* sections will be populated with the name of the highlighted field, and the form name respectively. A customized pane is always edited using the *Menu* button either against another form (in the case of adding a customized pane) or the customized pane's toolbar (when maintaining a customized pane). Therefore there is no notion of a field or form, so these fields contain the text *CustomizedPane* (see **Figure 10-17**).

When editing a form for a role you have the option to edit the system-wide script, or the one for the role. When editing a customized pane for a role, as the script is always kept per role you are not given this choice. This does not apply to customized panes added to the SYSPRO Main Menu as these are not part of the role design.

CustomizedPane Variables

When editing the VBScript for a customized pane, the *Variables* pane contains a *CustomizedPane* section. The content of this section is specific to the type of customized pane being edited.

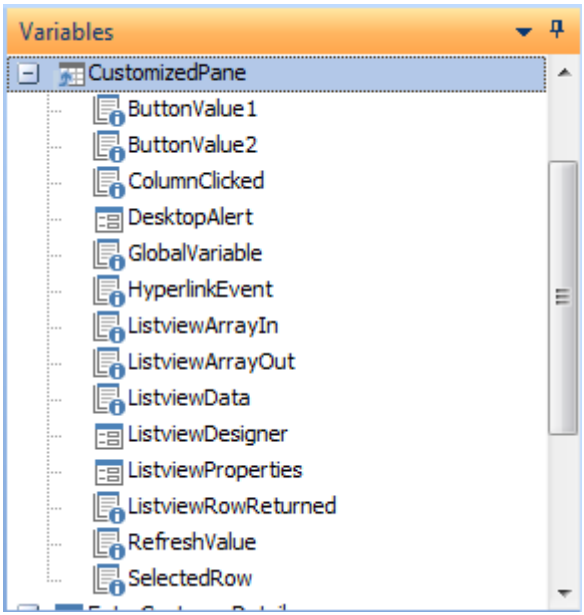


Figure 10-18: The *CustomizedPane* variables section for a listview

Figure 10-18 shows the *CustomizedPane* section of the *Variables* pane for a *Listview*. However, there are five variables within this section that are common to all customized pane types. These are *ButtonValue1*, *ButtonValue2*, *DesktopAlert*, *GlobalVariable*, and *RefreshValue*. These five variables

will be covered here, and those specific to a customized pane type will be covered in the section on that type of pane.

ButtonValue1 and ButtonValue2 Variables
Each customized pane can have up to two *Toolbar controls* configured against them. These toolbar controls can be any combination of *Edit control* (an entry field), *Checkbox*, or *Button*. The *ButtonValue1* and *ButtonValue2* variables are used to return the current value of these toolbar controls if they are configured to be either *Checkbox* or *Edit control*.

For an edit control the value returned will be whatever was entered in the edit control by the operator. If the entered value is longer than 240 characters, the value will be truncated and only the first 239 characters will be returned.

For a checkbox the returned value will be the current status of the checkbox, 1 for checked, and 0 for unchecked.

DesktopAlert Variable
The *Desktop Alert* variable is used to display a message on the operator's screen when a certain condition has been met. Double-clicking on the *Desktop Alert* variable name displays the *Desktop Alert Settings* screen where you can configure what the alert will contain. **Figure 10-19** shows the *Desktop Alert Settings* screen being populated. The *Duration* indicates how long the alert will stay on the screen unless you interact with it. The *Animation style* specifies how the alert will appear/disappear.

Against the *Heading line* section you can add a title for the alert against the *Text* prompt, and an icon can be chosen from a list to appear against this title. Against the *When clicked* prompt is a dropdown where you specify what will happen if the operator clicks on the alert's heading. The choices are *Do nothing*, *Launch SYSPRO program*, or *Run executable*. If either of the last two options are selected a browse enables you to locate the SYSPRO program or executable.

The *Subject line* section has similar options to specify the content of the main body of the alert, and the *Notes line* section has the same for the notes section at the bottom of the alert.

Figure 10-19 shows the *Desktop Alert Settings* screen where it has been configured to state that the WIP valuation is getting too high. If the operator clicks on either the heading or subject line, the *WIP at a Glance* query (***IMPATW***) will be run to show the detail values.

When you have finished configuring the alert, click the *Insert VBScript* button to add the code (see **Figure 10-20**).

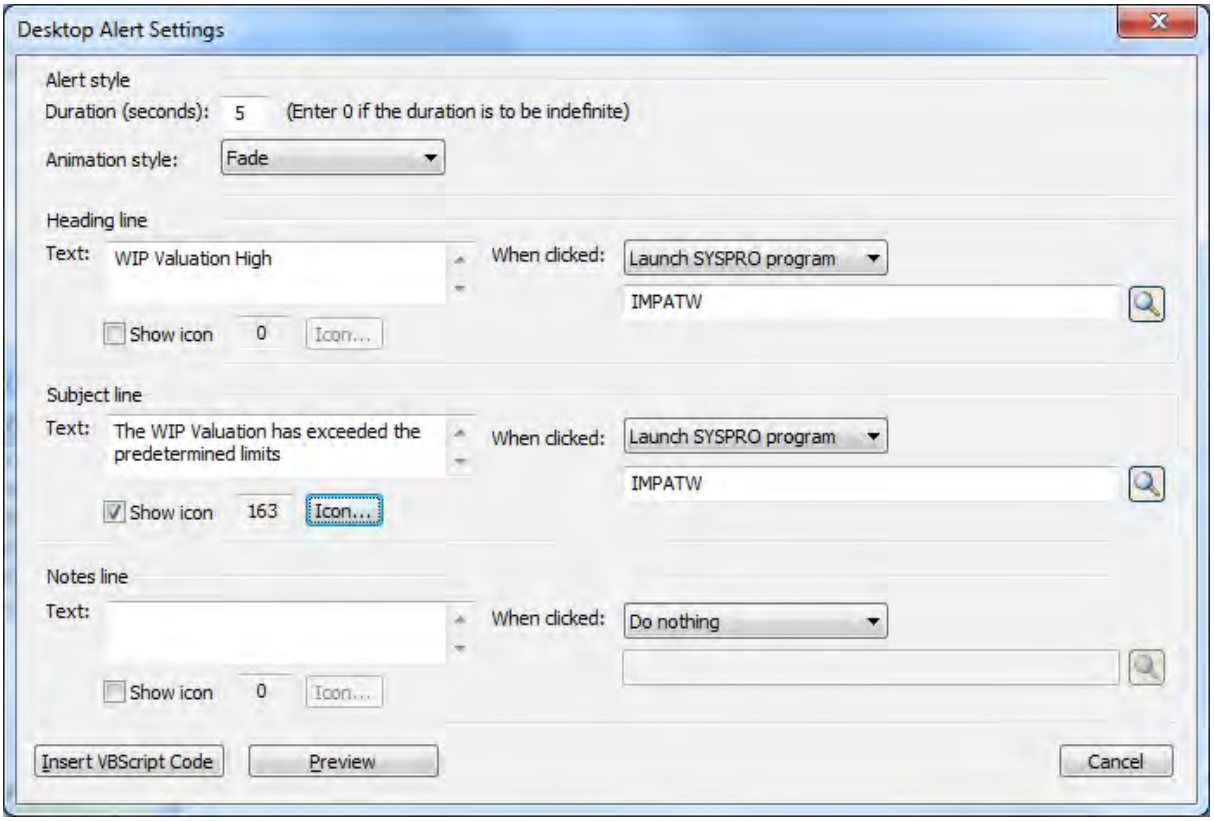


Figure 10-19: Configuring the *Desktop Alert Settings* screen

settings\ADMIN_VBS_ARSPEN01

```
1  ' This script contains functions for customized pane events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function CustomizedPane_OnRefresh()
6    dim Popup
7    Popup = Popup & "<Popup Duration='05' Animation='Fade'>"
8    Popup = Popup & "<Heading Text='WIP Valuation High' Icon='000'  SYSPROProgram='IMPATW'
9    Popup = Popup & "<Subject Text='The WIP Valuation has exceeded the predetermined limit
10   Popup = Popup & "</Popup>"
11  CustomizedPane.CodeObject.DesktopAlert = Popup
12 End Function
```

Figure 10-20: A section of the code used to configure a *Desktop Alert*

You would still need to add some code to check the current WIP values and compare these to predetermined values before firing the alert. **Figure 10-21** shows the *Desktop Alert* being displayed.

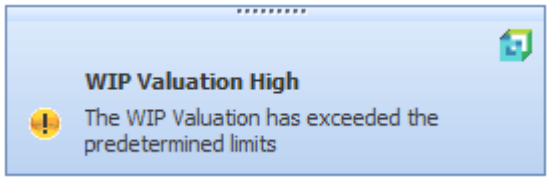


Figure 10-21: The *Desktop Alert* popup

GlobalVariable Variable

As each *Function* within a customized pane VBScript runs in isolation, setting a variable in one function means that this variable (and its content) expires when the function finishes running. So you could not create and set a variable in the customized pane's *OnLoad* function and access it from the customized pane's *OnRefresh* function.

The customized pane *GlobalVariable* remains available during the life of the customized pane. For example, if you populate this variable with a value during the customized pane's *OnLoad* function, this variable and its value remain available until the customized pane is closed (by closing the program within which it resides, or closing SYSPRO if the customized pane resides against the *SYSPRO Main Menu*).

RefreshValue Variable

A customized pane can have its *OnRefresh* event fired by other panes within the same SYSPRO program (or within the *SYSPRO Main Menu*, if that is where the customized pane resides). When this occurs, a string is passed through to the customized pane from the calling pane. The customized pane's *OnRefresh* function can access this string using the *RefreshValue* variable.

When editing the VBScript against any pane (including a customized pane), if there are customized panes configured within this program, the *Variables* pane of the screen where you edit the VBScript will contain a *CustomizedPanes* section. This section will have a list of all customized panes within this program. **Figure 10-22** shows the *CustomizedPanes* section for a program that has one customized pane within it, and this pane is called *Trial Listview*.

Double-clicking of the name of the customized pane will start the *Define Action for* wizard. **Figure 10-23** shows the *Define Action for* wizard after double-clicking on the *TrialListview* customized pane variable name. Under the section *Fire OnRefreshevent for this customized pane, passing it this value* is the default text *doRefresh*. This is the value that is passed through to the customized pane when it is refreshed by the code created using this wizard. This value can be changed.

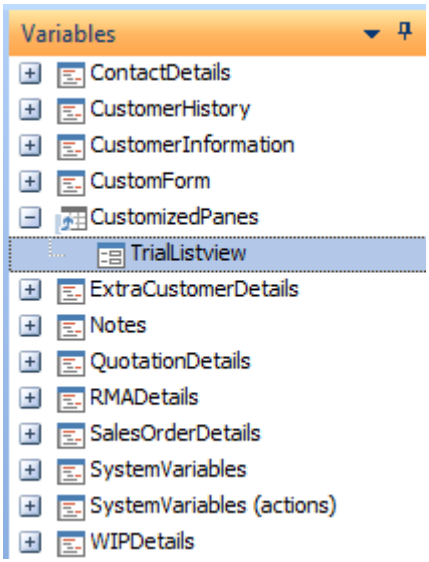


Figure 10-22: The *CustomizedPanes* section of the *Variables* pane

In this code snippet from the *Customer Query* program's *Customer Information* form, the text that will be passed through to the *Trial Listview* customized pane is *FromCI*.

```
Function CustomerInformation_OnRefresh()
  CustomizedPanes.CodeObject.TrialListview = "FromCI"
End Function
```

In the customized pane's *OnRefresh* function, the following code would display this value in a message box. Rather than just displaying the value this way, *IF* statements, or a *Case Select* statement could be used within the customized pane's *OnRefresh* function to perform different tasks depending on what value was passed through.

```
Function CustomizedPane_OnRefresh()
  msgbox CustomizedPane.CodeObject.RefreshValue,, "The Refresh Value"
End Function
```

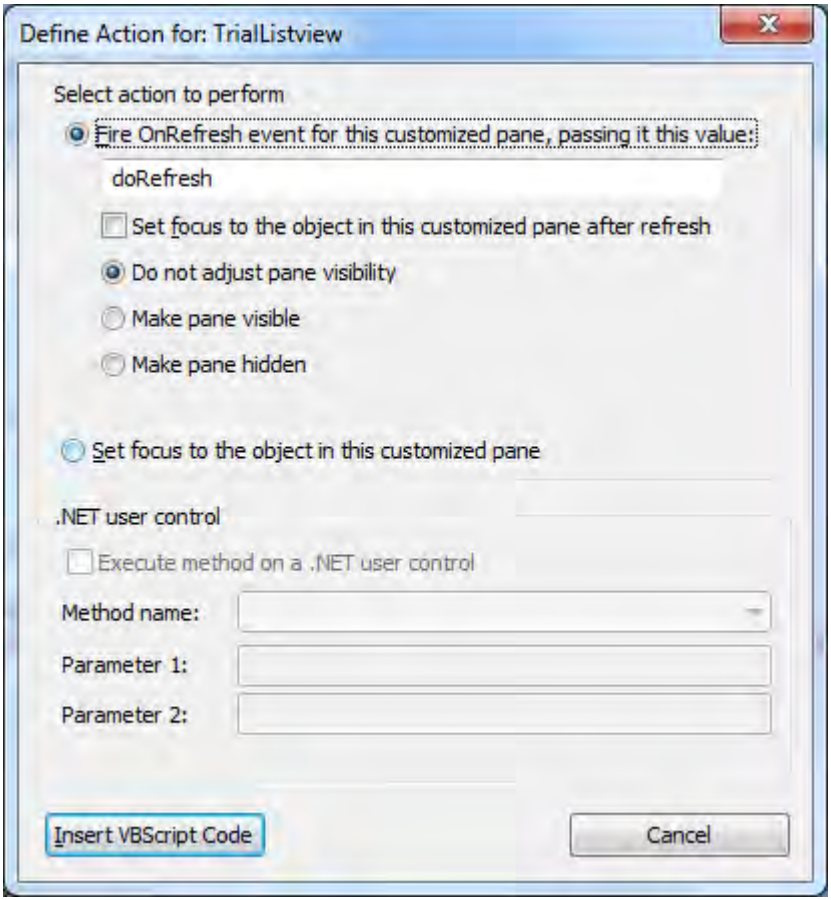


Figure 10-23: The *Define Action for* wizard

Exporting a Customized Pane

Customized panes can be exported so that they can be imported into another area of the same site, or another site. This could be an Administrator creating the same customized pane for another user/role, moving a customized pane from a test system to the live system, or a third party developer making their customized pane available for another site. A single `.txt` file is created containing the panes.

The *Export Customized Panes* program can be launched from the *Menu* button against any pane within the program containing the customized pane to be exported (*Menu* | *Customized Pane* | *Export Customized Panes*). This can be seen in **Figure 10-24**.

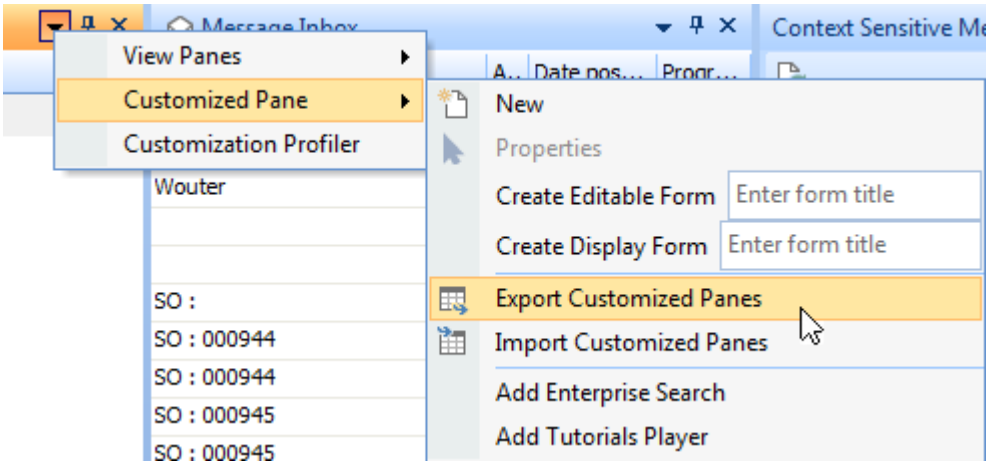


Figure 10-24: Using the *Menu* button to call up the *Export Customized Panes* screen

The *Export Customized Panes* screen consists of a toolbar and a listview. The listview contains all of the customized panes associated with the program from which it was launched (or the *SYSPRO Main Menu* if it was launched from there). **Figure 10-25** shows the *Export Customized Panes* screen where there are 14 customized panes associated with the program from which it was called. At the top of the screen is a toolbar containing the *Close* and *Export* buttons, the *Select* dropdown list, and the *Email the export file* checkbox.

The *Select* dropdown list contains the options *Select All* and *Unselect All*, making it easier to select which customized panes to export when there are many in the listview. Clicking on the *Export* button will display a browse where you can choose the location for the export file, and specify its name. If the *Email the export file* checkbox is checked, once the file has been exported the email program associated with this operator is called and an email opened that has the export file as an attachment. The operator can then provide the email address and click *Send*. The *Close* button enables the operator to exit without exporting.

In most cases, the export file contains everything required for the customized pane to be imported on the other system. For example, if the customized pane is a *Form* where the form structure was designed using the *Form Design* screen (and the form itself resides in a separate XML file), the export file will contain the structure of the form and will be built when the customized pane is imported. The exception is when a *.NET User Control* customized pane is exported; the user control and any dependencies must be taken across manually. Also note that if the customized pane must be refreshed by another pane, this will only occur if this change is made to the other pane on the new system.

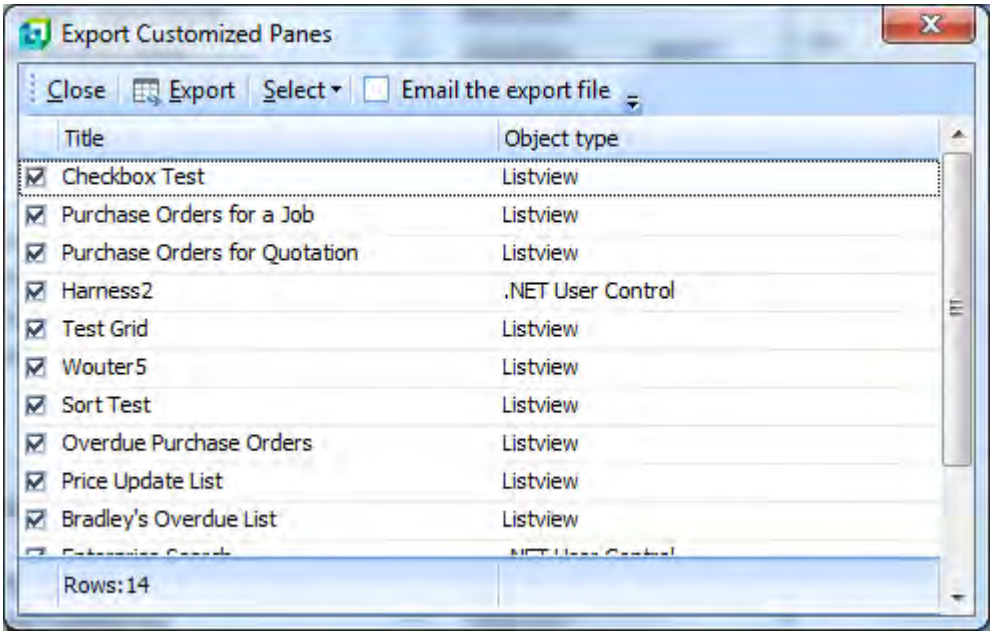


Figure 10-25: The *Export Customized Panes* screen

Importing a Customized Pane

Importing customized panes from an export file is performed by clicking on the *Menu* button of one of the panes within the program where the customized panes must reside (*Menu* | *Customized Pane* | *Import Customized Panes*). When this is selected a browse is opened that is used to locate the required export file. After locating and selecting the export file the *Import Customized Panes* screen is displayed which consists of a listview and a toolbar. The titles of the customized panes that resided in the export file will appear in the listview (see **Figure 10-26**).

The toolbar consists of a *Select* dropdown list, and the *Import* and *Close* buttons. The *Select* dropdown contains the *Select All* and *Unselect All* options, which make it easier to select the required customized panes to be imported when the export file contains a significant number of them. The *Import* button imports and creates the selected customized panes. The *Close* button enables the operator to exist the program without importing the customized panes.

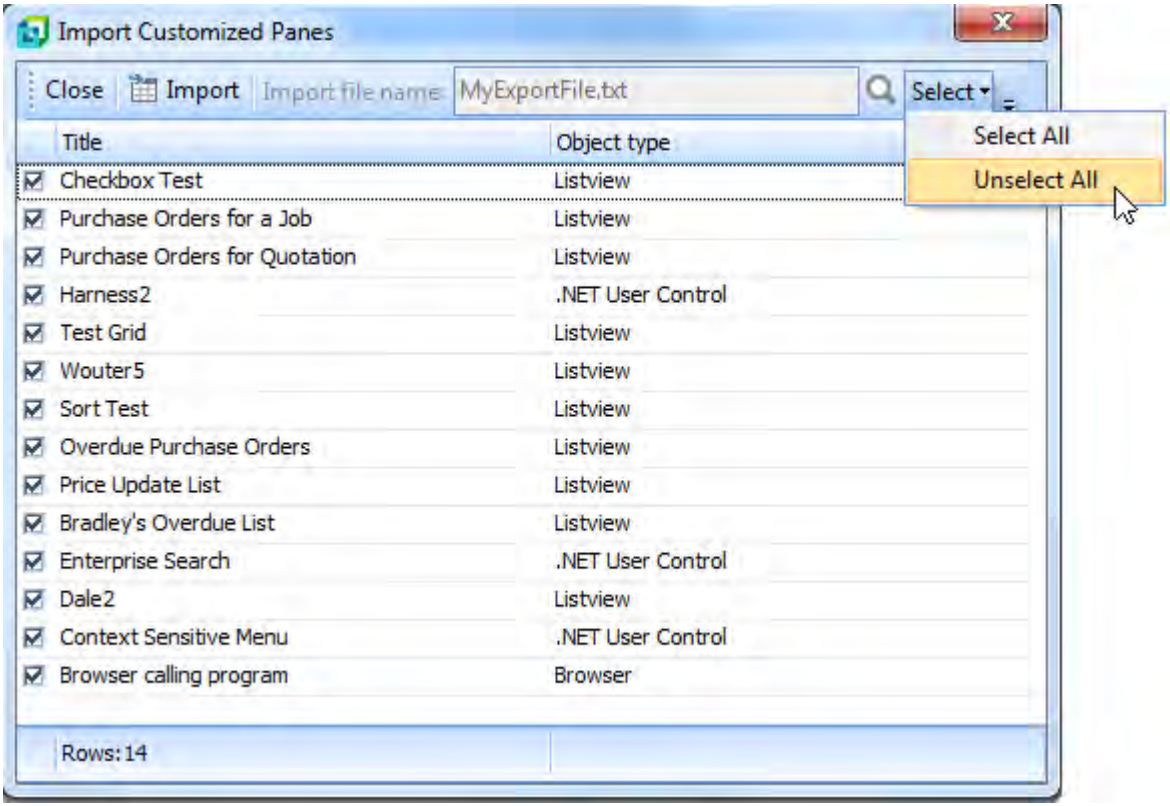


Figure 10-26: Using the *Unselect All* option in the *Import Customized Panes* program

Add Enterprise Search

The *Add Enterprise Search* option creates a *.NET User Control* customized pane that automatically populates the *Assembly name* with *SYSPROMA_Search.dll* and the *Assembly namespace* with *SYSPROMA_Search.SYSPROMA_MainSearch* (see **Figure 10-27**). No VBScripting is required.

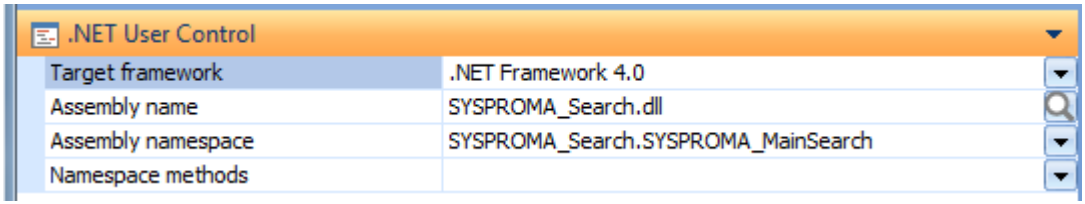


Figure 10-27: The *.NET User Control* settings for the *Enterprise Search*

The *Enterprise Search* is used to search all SYSPRO tables for the current company for records that match the supplied value. **Figure 10-28** shows the *Enterprise Search* in use. The search term used was CyCIE, and as can be seen, this is a case-insensitive search. The tag cloud to the right of the search results shows the total number of results, as well as how many came from each area of SYSPRO. In this example there are a total of 215 results with 140 coming from sales orders, 67 coming from stock codes, 4 coming from customers, and 4 coming from suppliers.

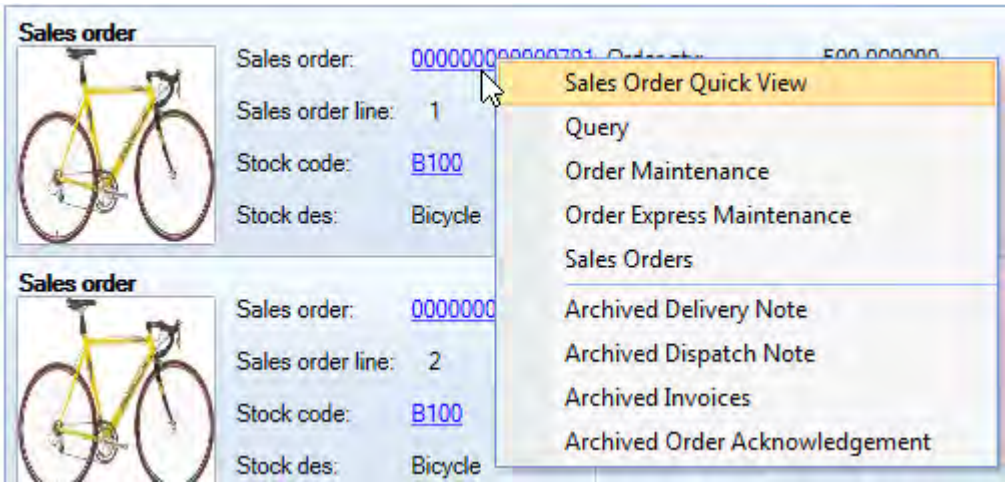


Figure 10-28: *Enterprise Search* results including the tag cloud

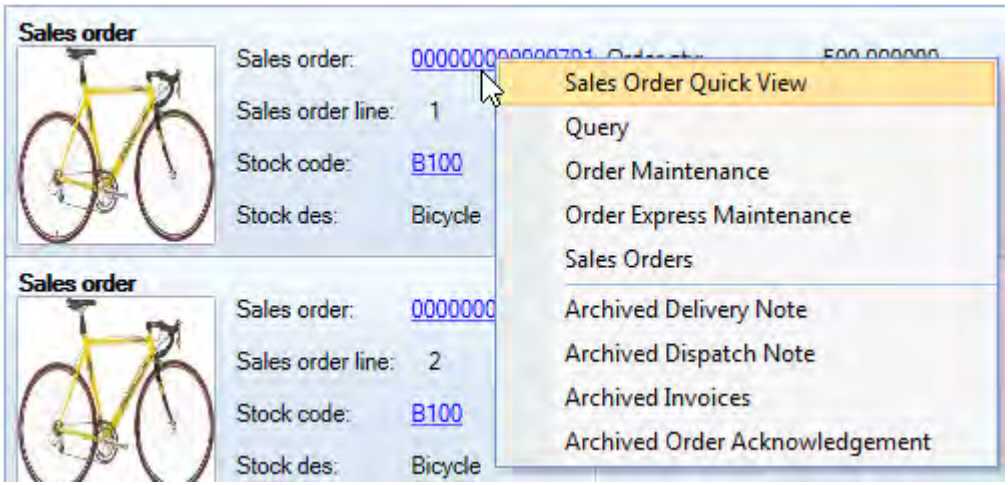


Figure 10-29: The context-sensitive menu allowing the operator to drill down

Within the results are hyperlinks enabling the operator to drill down to the individual records. When the operator clicks on one of these hyperlinks a context-sensitive menu is displayed, enabling them to choose how they want to drill down (see **Figure 10-29**).

Add Tutorials Player

The *Add Tutorials Player* option creates a floating customized pane using a *.NET User Control*. This can be used to play *Windows Media* (.wmv) files. SYSPRO ships with the *SYSPRO7 Introduction* movie. You can download others from the SYSPRO Support Zone or InfoZone, or create your own to go along with any customization/custom development that you have done. The player can be seen in **Figure 10-30**.

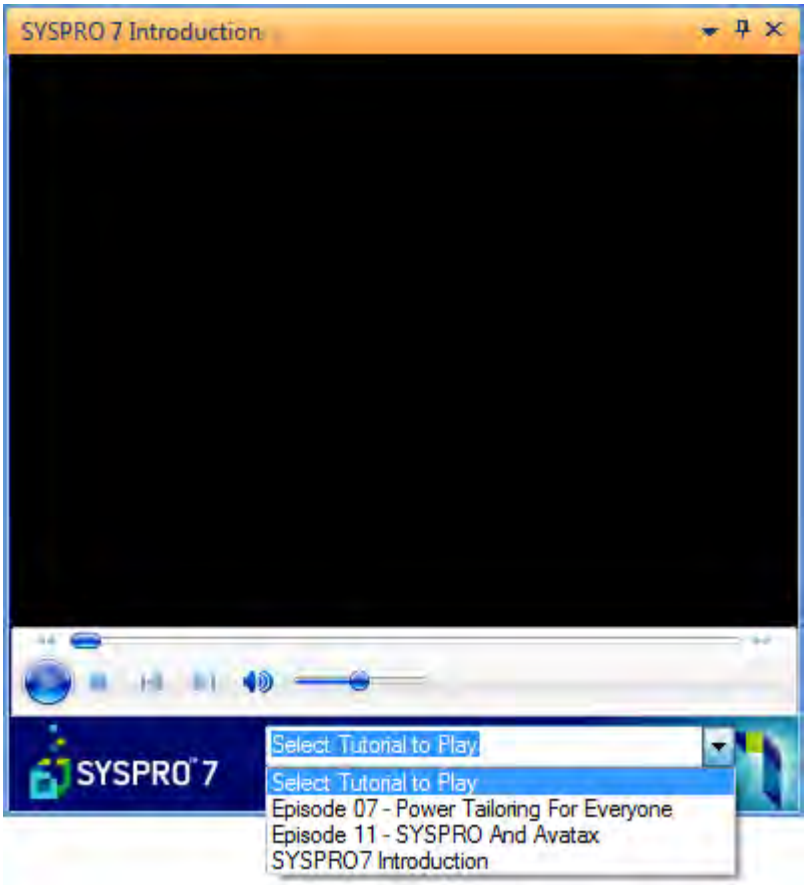


Figure 10-30: The *Tutorials Player* showing the available tutorials

To use a .wmv file within the *Tutorials Player* you must add it to the `Base\Samples` folder on the SYSPRO application server, and its name must be prefixed with *howto_* (note that the case is not important). The next time that the operator logs into SYSPRO on this workstation the .wmv file will be self-healed from the server to the client workstation, and be available for use in the *Tutorials Player*.

For example, if a file called *Episode 07 - Power Tailoring for Everyone.wmv* is downloaded from the InfoZone it must be renamed to *howto_Episode 07 - Power Tailoring for Everyone.wmv* and placed in the `Base\Samples` folder on the application server. When the operator next logs in on this workstation it will be available within the *Tutorials Player*. This can be seen as one of the files in the dropdown list at the bottom of the player in **Figure 10-30**.

Macro Events

There are thirteen macro events that are associated with customized panes. Four of these (*OnLoad*, *OnRefresh*, *OnToolbarButton1Clicked*, and *OnToolbarButton2Clicked*) are common to all customized panes. Six of the customized pane *Object types* only have these four events, these are:

- Web browser,
- SRS/Crystal report
- PDF Viewer
- Form
- Search Window
- Executive Dashboard

Table 10-1 shows the other *Macro Events* by *Object type*.

Macro Event	Graph	Listview	Rich text notepad	.NET User Control
OnPopulate		Yes		
OnDblClick	Yes	Yes		
OnLinkClicked		Yes		
OnAfterChange		Yes		
OnChecked		Yes		
OnRowSelected		Yes		
OnDELPressed		Yes		
OnClick	Yes			
OnSave			Yes	
OnAfterExecuteMethod				Yes

Table 10-1: The Macro Events

VBScript Editing Operator Security Activity

To be able to add or maintain a customized pane you need to have the operator activity *VBScript editing* checked. This appears within the *Customization* section of the *Activities* program (*Ribbon bar* | *Setup* tab | *Operators* button | select the operator | *Maintenance* button). The *Customization* section of the *Activities* screen can be seen in **Figure 10-31**.

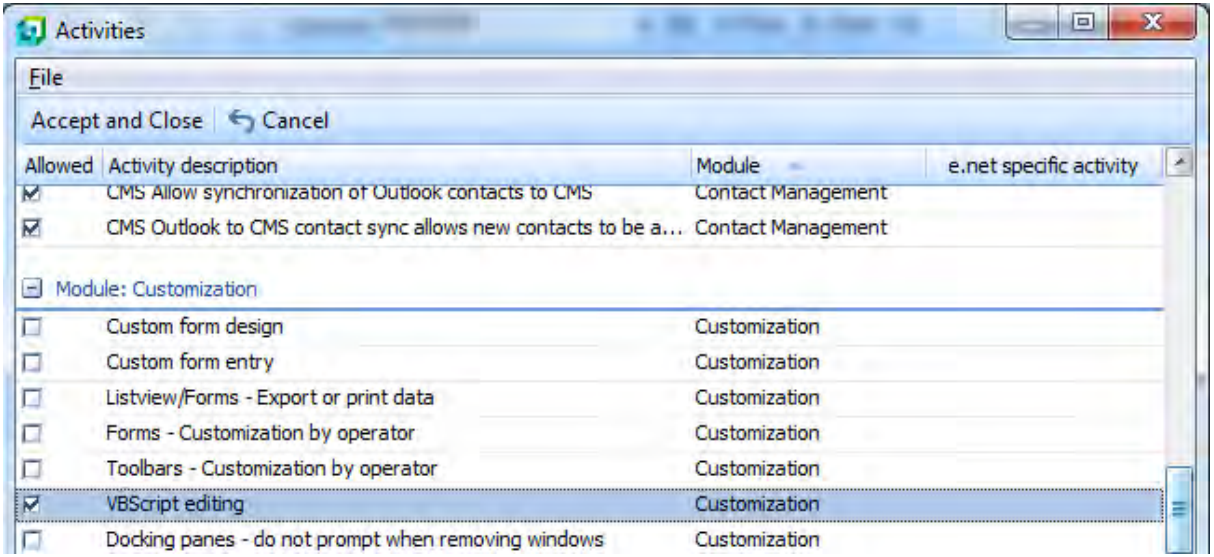


Figure 10-31: *VBScript editing* option of the *Activities* screen of the *Operator Maintenance* program

This option allows/disallows the adding or maintaining of customized panes. However, even if this is set to be disallowed, customized panes that are set up will still appear and work. If this option is changed it will only take effect the next time that this operator logs into SYSPRO.

The *Activities* section consists of the *Selection* and *Configure activities* options (see **Figure 10-32**). These options work together to set which activities the operator can perform. A dropdown list exists against the *Selection* prompt and contains the options *Default*, *All*, *None* and *List*.

If *Selection* is set to *Default*, certain of the activities will be checked, others will be unchecked, and the *Edit* hyperlink against the *Configure activities* option is disabled. The *VBScript editing* option is unchecked by default.

If *Selection* is set to *All*, all of the activities are checked and the *Edit* hyperlink against the *Configure activities* option is disabled.

If *Selection* is set to *None*, all of the activities are unchecked and the *Edit* hyperlink against the *Configure activities* option is disabled.

If *Selection* is set to *List* the current activity settings remain and the *Edit* hyperlink against the *Configure activities* option is enabled. Clicking on this hyperlink displays the *Activities* screen which contains all of the operator activities grouped by "module", which includes a group called *Customization*.

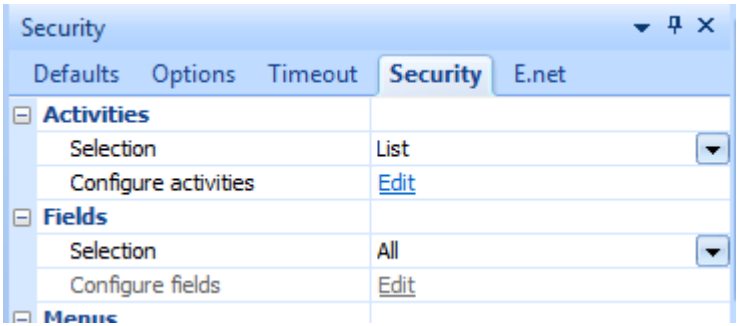


Figure 10-32: The *Activities* section of the *Security* tab of the *Operator Maintenance* program

Once the required options are checked/unchecked, the *Accept and Close* button can be used to return to the *Operator Maintenance* program, where the *Save* button will save these settings. These take effect the next time that the operator logs into SYSPRO.

Adding Customized Panes when a Member of a Role

If you are a member of a role, you can only add a customized pane to a SYSPRO program if you are in *Design Mode* for roles (*Ribbon bar* | *Administration* tab | *Design UI Layouts* | *Role* radio button | select role from dropdown list | *Start Design Mode* button). Note that you must also have the *VBScript editing* operator activity checked to be able to add a customized pane once you are in *Design Mode*.

You can add a customized pane to the SYSPRO *Main Menu* if you are not in *Design Mode* because the main menu is not under the control of roles.

Application Builder, System-wide View and Role Views

On the *Application Builder* menu (*SYSPRO Ribbon Bar* | *Administration* tab | *Customization* section | *Application Builder* menu) there are 20 application placeholders that you can use to build your own applications. Each of these can be configured to contain one or more customized panes, and these

are specific to this operator. The *Application Builder* is covered in detail in the *Application Builder* section of *Chapter 9* which covers *Toolbars*, *Form Actions*, the *Login*/*Logout* scripts and the *Application Builder*.

At the bottom of the *Application Builder* menu is a *System-wide View* option. This is also an application placeholder and can be configured to contain one or more customized panes. The difference between the *System-wide View* application and one of the *Application Builder* applications is that this application is common to all operators. There is also an option within the *System-wide Personalization* program to specify whether the *System-wide View* must appear by default for this operator when they login. The *System-wide View* is also covered in more detail in the *Design System-wide View* section of *Chapter 9*.

Below the *System-wide View* option on the *Application Builder* menu is the *Role View* option. This is similar to the *System-wide View* application, but for the role to which the operator belongs. There is also an option within the *System-wide Personalization* program to specify whether the *Role View* must appear by default for this operator when they login. The *Role View* is also covered in more detail in the *Design Role View* section of *Chapter 9*.

Chapter 11 - Listviews

The structure of a customized pane listview is defined using XML, and it is typically populated using XML that is supplied in the *ListviewData* variable.

The macro events available for a listview customized pane appear in **Figure 11-1**. An explanation of when each macro event type fires is covered in the *Macro Events* section.

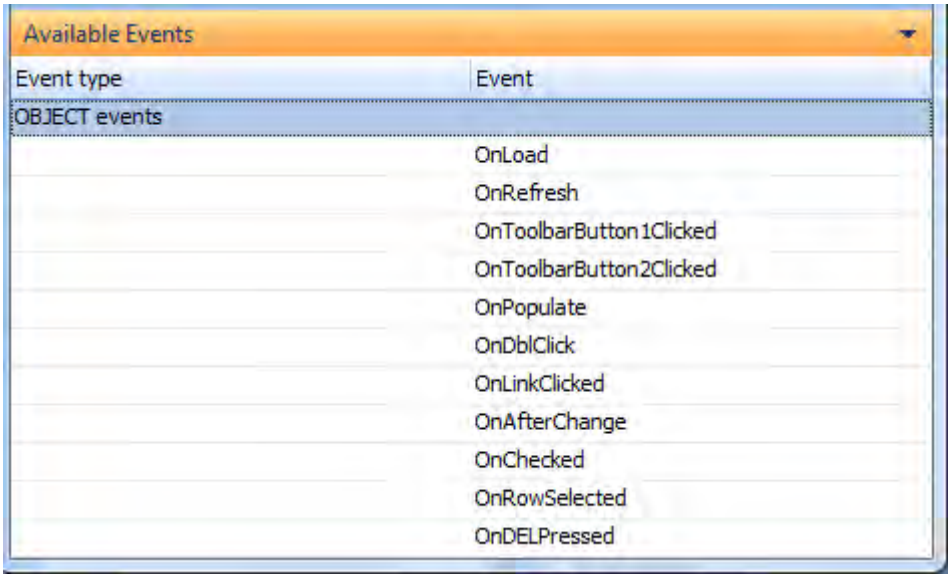


Figure 11-1: The list of available macro events for a listview customized pane

With a listview, the *OnLoad* event is used to define the structure of the listview (such as which columns are to appear, their type, etc.) and the *OnRefresh* event is used to populate the listview with data. The *OnPopulate* event can be used to set field properties, and manipulate the values before this data is populated. The *OnRowSelected* event is used to detect that the operator has clicked on a row within the listview, and the *OnDblClick* event is used to detect that the operator has double-clicked on an already selected row.

If a listview column is defined as being editable, and the operator changes its value, the *OnAfterChange* event will fire. If the editable column is defined as a checkbox, and the operator clicks on it, the *OnChecked* event fires (whether the checkbox's status is changing to checked or unchecked). A column can be defined as containing hyperlinks. When the operator clicks on this hyperlinked value within the column, the *OnLinkClicked* event will fire.

Defining the Listview Structure

Typically, the structure of the listview is defined within the customized pane's *OnLoad* function. The listview structure is defined using XML, and within this XML there will be one *Columns* root element, and one or more *Column* elements. Each *Column* element defines a column of the listview. To define the structure of a listview with five columns, you would build XML with one *Columns* element, and five *Column* elements. Once the XML to define the listview structure has been built it is passed to a variable called *ListviewProperties*, and this renders the structure.

This may sound a little daunting, but you have the choice of two wizards to build this XML for you. The first of these wizards assists you in designing/building your own listview structure, and the second assists you in designing/building your listview if you are using the output of a business object to populate the listview.

The following is an example of the XML to build a two column listview, consisting of a column called *Job*, and one called *Job description*.

```
<Columns PrimaryNode='Jobs' Style='DataGrid' AutoSize='true' FreezeColumn='0' >
  <Column Name='Job' Description='Job' />
  <Column Name='JobDescription' Description='Job description' Editable='true' />
</Columns>"
```

The Columns Element

There are five attributes that can be used against the *Columns* element, and the example above contains four of them. The available attributes are *PrimaryNode*, *Style*, *AutoSize*, *FreezeColumn*, and *AutoInsert*.

PrimaryNode Attribute

The *PrimaryNode* attribute is used to define from where in the supplied XML the data is to be extracted. In the two column example above, the *PrimaryNode* attribute is set to *Jobs*. If the XML sample below was supplied within the *ListviewData* variable, three nodes would be returned. These would ultimately become three rows in the listview. All of the repeating elements must be at the same level in the XML, they cannot be scattered at different levels throughout the XML.

```
<MyXML>
  <HeaderInfo>
    <Line1>My Job List</Line1>
  </HeaderInfo>
  <DetailInfo>
    <Jobs>
      <Job>123456</Job>
      <JobDescription>My Job</JobDescription>
      <QtyToMake>1</QtytoMake>
    </Jobs>
    <Jobs>
      <Job>3334456</Job>
      <JobDescription>Job number 2</JobDescription>
      <QtyToMake>5</QtytoMake>
    </Jobs>
    <Jobs>
      <Job>72318</Job>
      <JobDescription>Large Job</JobDescription>
      <QtyToMake>125</QtytoMake>
    </Jobs>
  </DetailInfo>
</MyXML>
```

Style Attribute

The *Style* attribute is used to specify whether this customized pane should resemble a spreadsheet (*DataGrid* style), or a form (*Form* style). A listview with its style attribute set to *DataGrid* contains one or more columns, and is populated with rows of data (see **Figure 11-2**).

Figure 11-2: A *DataGrid* style listview

A listview with its style attribute set to *Form* contains two columns, and is populated with caption/value pairs (see **Figure 11-3**).

Figure 11-3: A *Form* style listview

AutoSize Attribute

The *AutoSize* attribute is used to specify whether the width of the listview's columns will change if the whole listview is made wider or narrower. Setting this attribute to *False* means that if the width of the listview increases or decreases, the individual column widths will remain the same. If the listview's width is reduced to such an extent that the columns no longer fit on the listview, a slider bar appears at the bottom of the screen so that the operator can navigate right/left. **Figure 11-4** shows a listview where the *AutoSize* attribute has been set to *False*. When the listview was made narrower the columns remained the same size and the slider bar appeared at the bottom of the screen.

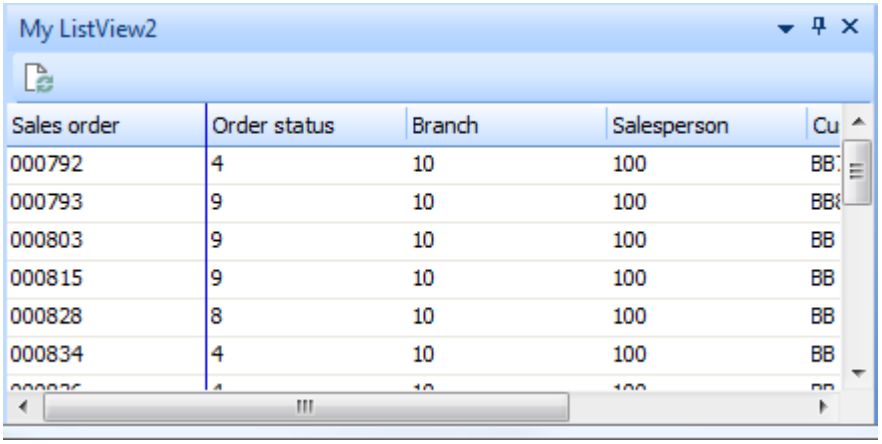


Figure 11-4: A listview with the *AutoSize* attribute set to *False*

Setting the *AutoSize* attribute to *True* means that if the width of the listview increases/decreases, the widths of the individual columns will change in size proportionally.

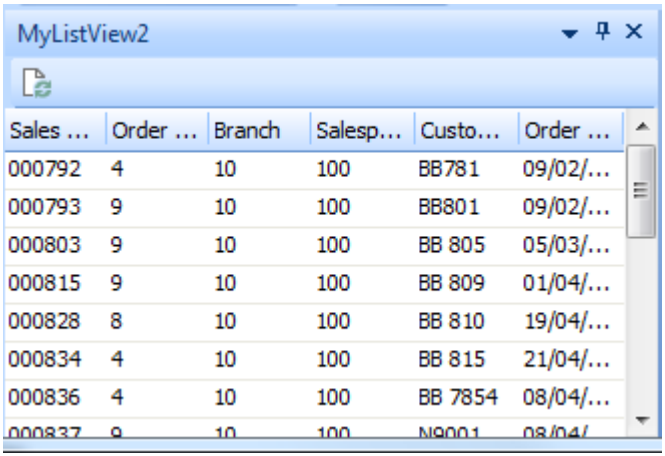


Figure 11-5: A listview with the *Autosize* attribute set to *True*

Because all the listview columns are displayed (even if the operator makes the listview really narrow) the slider bar will not appear. **Figure 11-5** shows a listview where the *AutoSize* attribute is set to *True*. When the listview is made narrower the columns are reduced in size proportionally.

FreezeColumn Attribute

The *FreezeColumn* attribute is used in prevent one or more columns from scrolling to the left when the *AutoSize* attribute is set to *False*, and the combined width of the listview columns is greater than the width of the listview. The possible values for the *FreezeColumn* attribute are 0 to 9. The listview that appears in **Figure 11-4** has its *FreezeColumn* attribute set to 1. The line delimiting the border between the first and second columns is slightly darker.

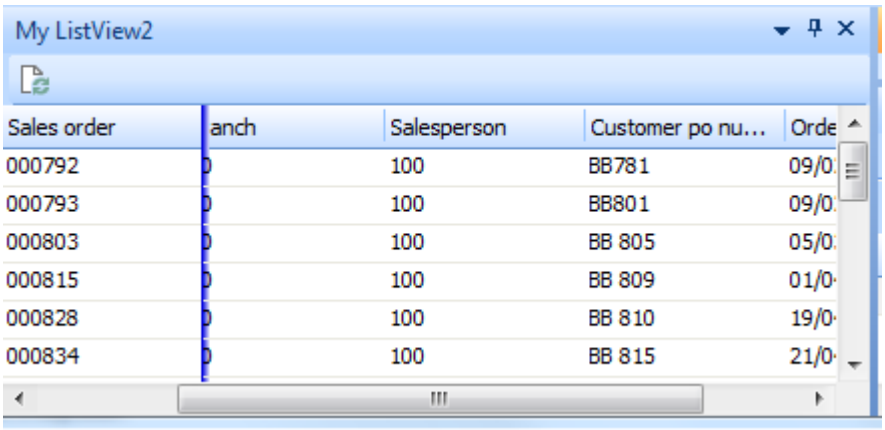


Figure 11-6: Using the *FreezeColumn* attribute to keep the first column in position while others scroll

Figure 11-6: shows the same listview where the slider bar at the bottom of the screen has been dragged to the right. The first column (*Sales order*) remains in position, and the other columns scroll to the left.

AutoInsert Attribute
The *AutoInsert* attribute (when set to *True*) enables the operator to add one row at a time to the end of a listview, providing that at least one column within the listview has been configured as *Editable*. The operator can only populate those columns within the row that have been configured as *Editable*. **Figure 11-7** shows a row being added, but as only the *Order status* column has been configured as *Editable*, it is the only column of the row that the operator can populate with data.

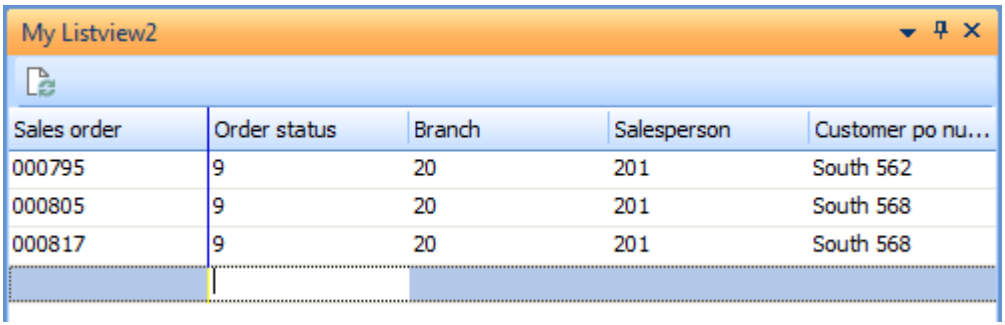


Figure 11-7: Adding lines to the bottom of a listview using the *AutoInsert* option

AllowDEL Attribute
When set to *True*, the *AllowDEL* attribute allows the operator to select a row and press the *Del* key to remove this row.

AllowUndo Attribute
The *AllowUndo* attribute works in conjunction with the *AllowDEL* attribute. When this is set to *True* it enables the operator to undo the deletion of rows using the *Ctrl+Z* shortcut keys. If multiple rows have been deleted using the *Del* key, each use of the *Ctrl+Z* shortcut keys will undo the previous deletion.

HighlightRow Attribute
The *HighlightRow* attribute changes the first column of the listview to a pointer to show which line is highlighted. If this attribute is added to an existing listview you will need to add an extra column to the beginning of the listview, otherwise the content of the first column will not be visible.

Figure 11-8 shows a listview with the *HighlightRow* attribute set to *False*. It consists of four columns, all populated with data. If the *HighlightRow* attribute is set to *True* the content of the first column is replaced with the pointer that shows which row is highlighted (see **Figure 11-9**). To be able to have all

the columns and the pointer, an extra column was added to the beginning of the row (see **Figure 11-10**). The following is the line of XML that was added to produce this column.

```
<Column Name='xxx' Description='Select' />
```

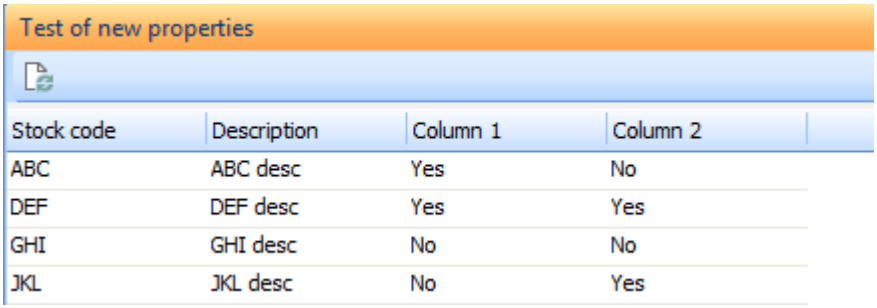


Figure 11-8: A listview with the *HighlightRow* attribute set to *False*

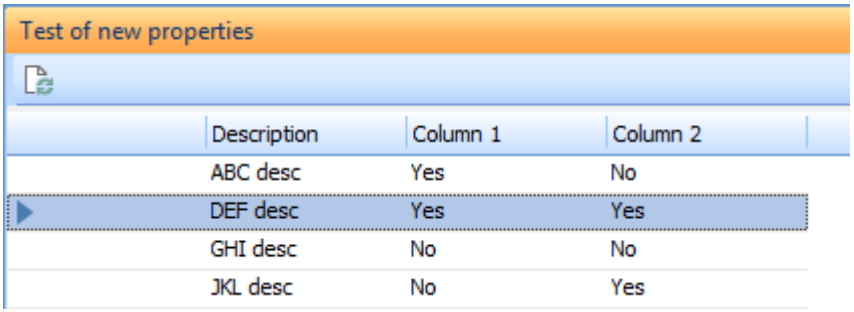


Figure 11-9: The same listview but with the *HighlightRow* attribute set to *True*

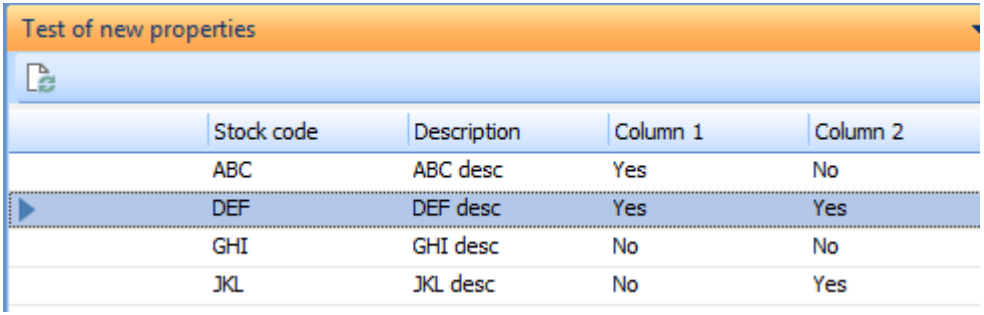


Figure 11-10: Adding an empty column to the beginning of the listview

If you check the *Auto row highlight* checkbox within the *Listview Designer* a message will be displayed informing you that the first column will be used for the row highlighting, and ask if you would like this row defined for you. If you select the *Yes* button, an extra column is inserted at the beginning called *autoRow*. If you select the *No* button, no extra column is added. When the listview is displayed your first column will be overwritten with the pointer column, and you will be unable to view the contents of your first column.

DragDrop Attribute

The *DragDrop* attribute enables the operator to drag rows up/down within the listview. In **Figure 11-11** the operator has highlighted the row containing the stock code *DEF* and dragged it to be the last row. A solid line highlights where the row will reside when the operator releases the mouse button.

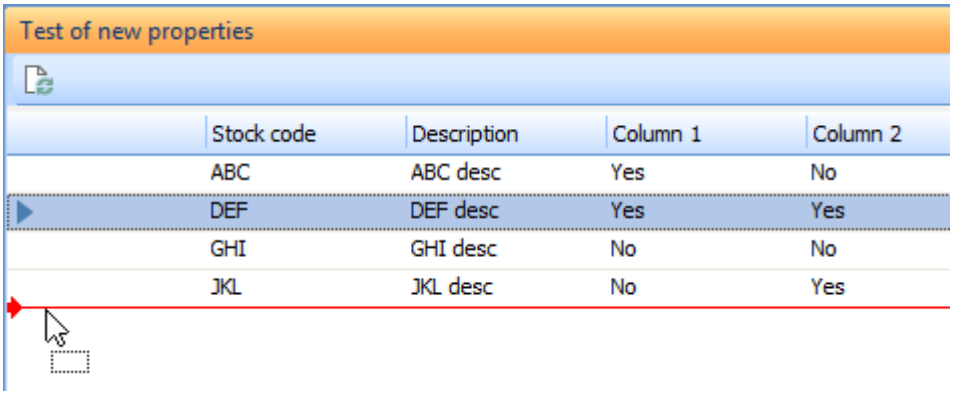


Figure 11-11: Dragging the row containing stock code *DEF* to be the last row

When the operator releases the mouse button the row is moved (see **Figure 11-12**).

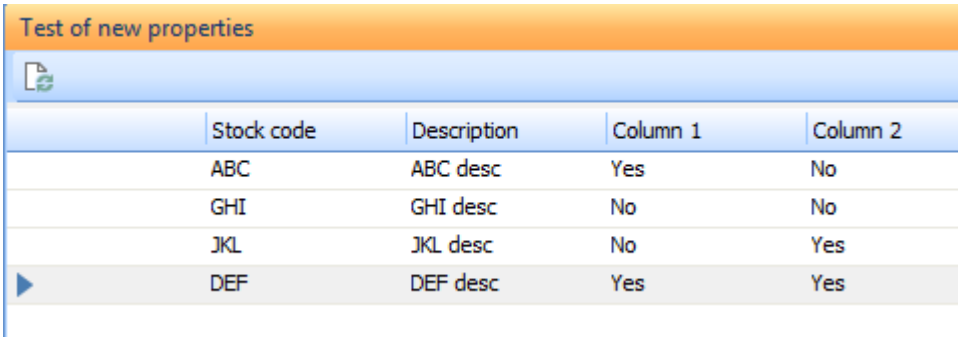


Figure 11-12: The listview after the operator releases the mouse button

PopulateVisible Attribute
The *PopulateVisible* attribute is used to improve the performance of a listview if there are many rows and many columns. When set to *True*, only columns that are visible will have their cells populated with data in the arrays. If a column is dragged from the listview before the listview is populated, when the column is dragged back onto the listview it will be blank, and it will populate when you refresh/reload the listview.

The Column Element

There are 21 attributes that can be used against each *Column* element. Some of these attributes can only be used when another attribute is present. For example, the *Decimals* attribute specifies how many decimal places to display and is used when the *Type* attribute is configured as *Numeric*.

Name Attribute
The *Name* attribute gives the column a name, and also specifies the element name from the XML that will be used to populate this column. Using the extracts below from the earlier example, the column *Name* is *Job*. This means that when the XML containing the data is provided to the listview, the *PrimaryNode* attribute against the *Columns* element is used to locate the recurring nodes containing the data, and the value against the *Job* element is used to populate the *Job* column. Note that this is case-sensitive, so in this example if the XML contained a <job> element instead of a <Job> element, the values would not be retrieved.

The following is an extract from column definitions section above.

```
<Column Name='Job' Description='Job' />
```

The following is an extract from XML containing the data.

```
<DetailInfo>
  <Jobs>
    <Job>123456</Job>
    <JobDescription>My Job</JobDescription>
    <QtyToMake>1</QtytoMake>
  </Jobs>
```

Description Attribute
The *Description* attribute is used to specify the title of the column that will appear in the column heading. There are many occasions where the element name against the data to be displayed would not be suitable for the column heading, or may use unfamiliar, and this allows you to display a more meaningful heading. In the example below the element in the XML that contains the data is *Journal*, but your preference is to use the term *Audit trail*. The column header will contain *Audit trail number*.

```
<Column Name='Journal' Description='Audit trail number' />
```

Type Attribute

The *Type* attribute enables you to specify whether the content of this column is *Numeric*, *Alpha*, *Date*, *Address*, *Checkbox*, or *Dropdown*. Unless specified using the *Type* attribute, SYSPRO will attempt to work out the column's *Type*, based on the value for this column from the first row.

It is best practice to specify what a column contains using the *Type* attribute. It is possible that the column should be set to *Alpha* because it can contain numbers and letters. However, if this is not specified, and the first row of this column contains a numeric value, the column will be assumed to be numeric and all values within this column displayed as if they were.

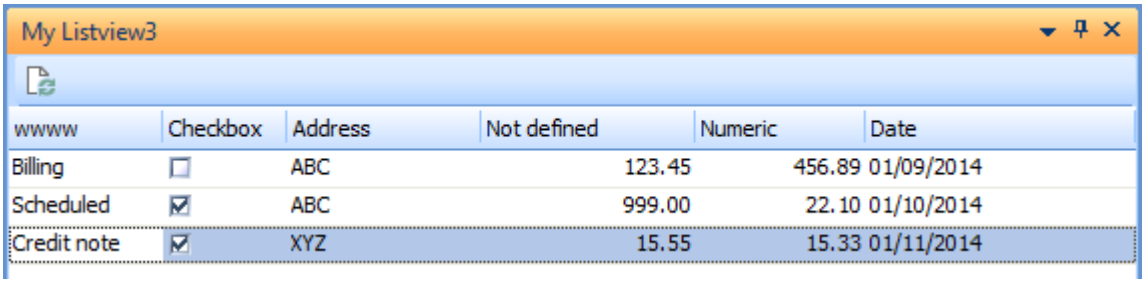


Figure 11-13: The fourth column has no *Type* definition so assumes it from the value of the first row

Figure 11-13 shows a listview where the *Type* of the fourth column is not defined. The XML for the first row contains the value 123.45, so the *Type* is assumed to be numeric with two decimal places. However, the XML for the second row contains the value ABC999, which is not numeric. As the column has been assumed to only contain numeric values (with two decimal places), it displays the value of ABC999 as 999.00.

Setting a column's *Type* to be *Alpha* will default to both the value and the column header being left-justified. There are attributes to set the justification of both the header and value.

Setting a column's *Type* to *Numeric* will default the value to be right-justified, and the header to be left-justified. The *Numeric* type works in conjunction with the *Decimals* attribute that is listed below. The fifth column in **Figure 11-13** has been set to *Numeric* with two decimals. The values supplied are 456.89, 22.1, and 15.333. The value 22.1 has automatically been padded to 22.10, and the value 15.333 has been truncated to 15.33. The code below shows the *Type* being set to *Numeric* and the *Decimals* to 2.

```
<Column Name='Num' Description='Numeric' Type='Numeric' Decimals='2' />
```

Reading back the contents from a cell in a numeric column will return the value of the cell. If the value was truncated in the display (such as in the 15.333 example above) the value returned will also be truncated. If the displayed value ends in one or more zeros as the least significant digits after the

decimal point, the returned value will not contain these zeros. For example, if a value of 19.30 is supplied to a numeric column defined with two decimal places, it will be displayed as 19.30. When reading back the value from the cell the value returned would be 19.3.

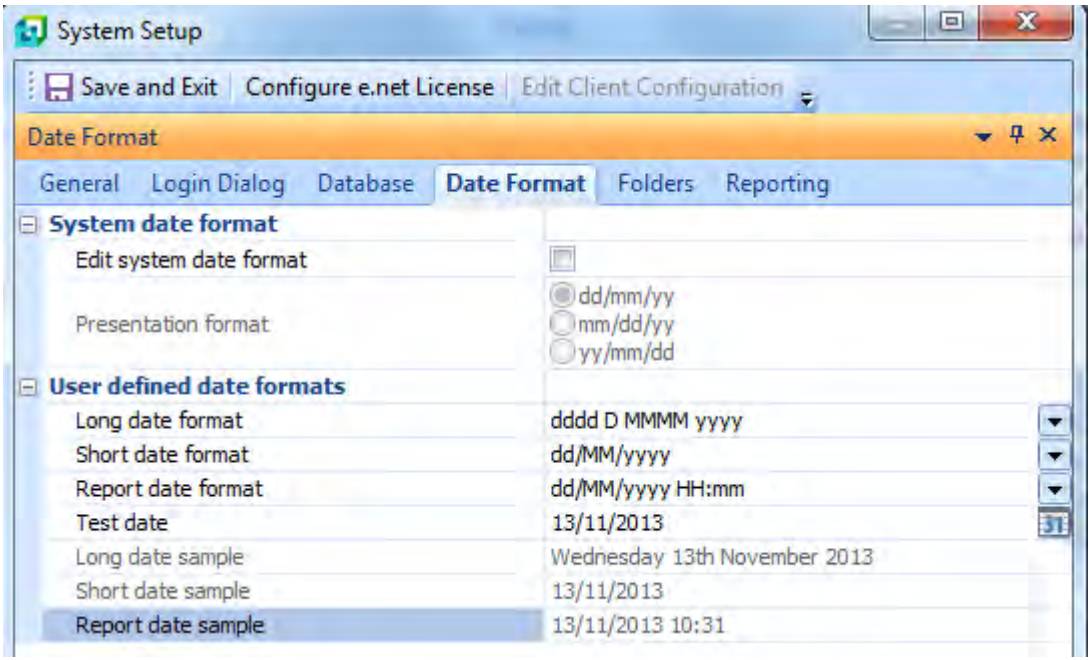


Figure 11-14: Viewing the *Short date format* on the *Date Format* tab of the *System Setup* program

Setting a column's *Type* to be *Date* will default both the heading and value to be left-justified. The date must be supplied in the format CCYY-MM-DD, CCYY/MM/DD, or CCYYMMDD. By default, the date is displayed in the format that you have configured as your *Short date format* on the *Date Format* tab in your *SYSPRO System Setup* (*Ribbon bar* | *Setup* tab | *General Setup* | *System Setup* | *Date Format* tab). The *Short date format* and *Long date format* can be seen in **Figure 11-14**.

As with any listview, the format of the date can be changed by the operator. This is done by right-clicking on the column header and selecting *Customize* | *Date Format*, then selecting from the available options. The default is SYSPRO's *Short Date Format* that can be seen in **Figure 11-14**. The alternatives are SYSPRO's *Long Date Format* which can also be seen in **Figure 11-14**, and two others that can be seen in **Figure 11-15**.

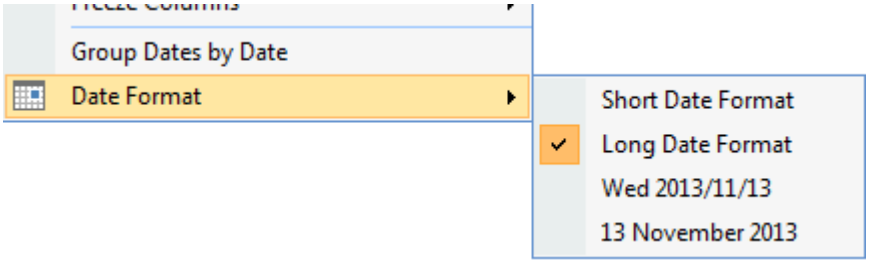


Figure 11-15: Overriding the date format against the listview

Setting a column's type to *Checkbox* will display a checkbox in all its populated cells. If the value supplied within the XML for this cell is 1 or Y, the checkbox will be checked. Any other value supplied in the XML will cause the checkbox to be unchecked.

However, if a value is supplied to a checkbox cell using the *_OUT* array during the *OnPopulate* function, it works differently. Supplying a 1 or Y as the value causes the checkbox to be checked. Supplying a 0 or N causes the checkbox to be unchecked.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='1' > </Field>"
End Function
```

Supplying a value other than 0, 1, Y, or N to the cell using the *_OUT* array causes this value to be displayed alongside the checkbox, and the checkbox status to remain as it was. **Figure 11-16** shows the results of passing the value *XXX* to the cell (using the code below).

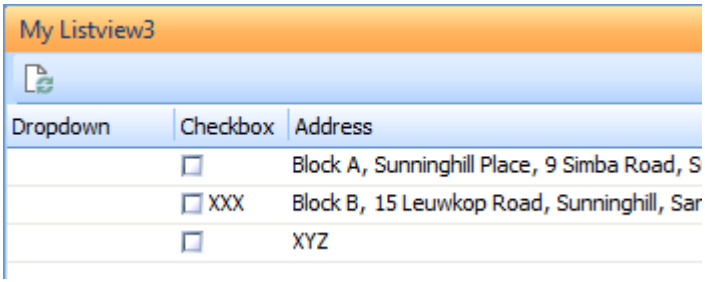


Figure 11-16: Supplying a value to appear alongside a checkbox

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='XXX' > </Field>"
End Function
```

It is possible to supply a value that must appear alongside the checkbox, and set the checkbox's status at the same time. This is done by providing the checkbox status in braces/curly brackets within the *Value* attribute. The example below checks the checkbox, and adds the value *XXX* alongside the checkbox (see **Figure 11-17**)

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='{1}XXX'  > </Field>"
End Function
```

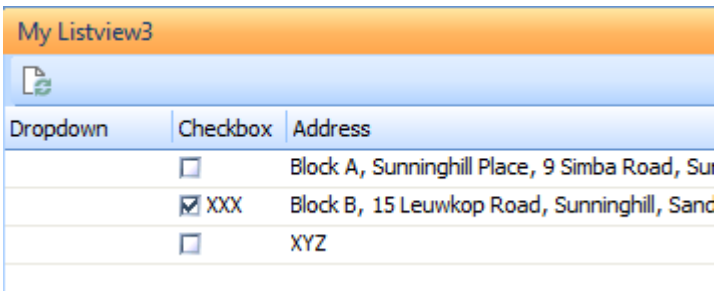


Figure 11-17: Setting the checkbox status and supplying a value to appear alongside

Reading back the value of a checkbox will always return a zero or a one, not the value that was originally supplied, even if it was supplied to sit alongside the checkbox.

Setting a column's *Type* to *DropDown* will add the same dropdown list to each cell in this column. An example can be seen in **Figure 11-18**.

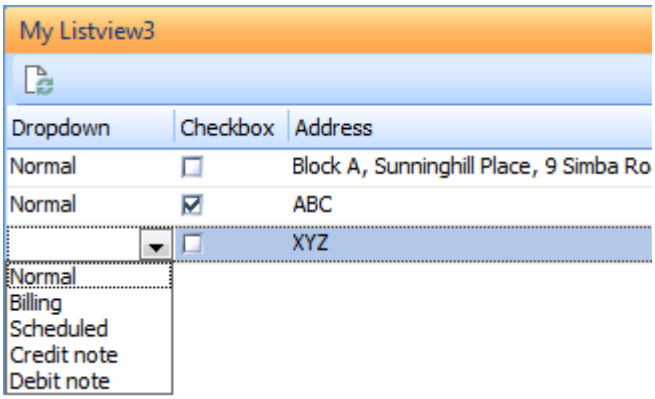


Figure 11-18: Configuring a column with the *Type* of *DropDown*

The choices for the listview are supplied in the *List* attribute. The following is the contents of the *List* attribute to produce the dropdown shown in **Figure 11-18**.

```
List='Normal;Billing;Scheduled;Credit note;Debit note'
```

The line of code below (which has wrapped around on this page) is the complete *Column* element to achieve this.

```
<Column Name='DD' Description='Dropdown' Type='DropDown' Editable='true'
List='Normal;Billing;Scheduled;Credit note;Debit note'/>
```

If the column is not defined as *Editable*, the dropdown list will not appear against the cell, because you can't use it.

If you read back the contents of a cell that contains a value from the dropdown list, you will receive the number of the entry, starting from zero. Using the code snippet above, if the cell contained *Normal* you would receive 0, if it contained *Debit note* you would receive 4.

It is possible to prepopulate the cell against this dropdown list with one of these values by supplying its entry number in the XML used to populate it. Alternatively the listview *_OUT* array can be passed the value within the listview's *OnPopulate* function. The code snippet below will set the first column, first row in **Figure 11-13** to *Billing*, because *Billing* is the second entry in the *List* attribute (which just like an array, starts at zero).

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(0,0)= "<Field Value='1' > </Field>"
End Function
```

When a column's type is set to *Address*, multiple pieces of information can be supplied and displayed in one cell, making up the whole address. The individual parts are displayed, separated by commas, as can be seen in **Figure 11-19**.

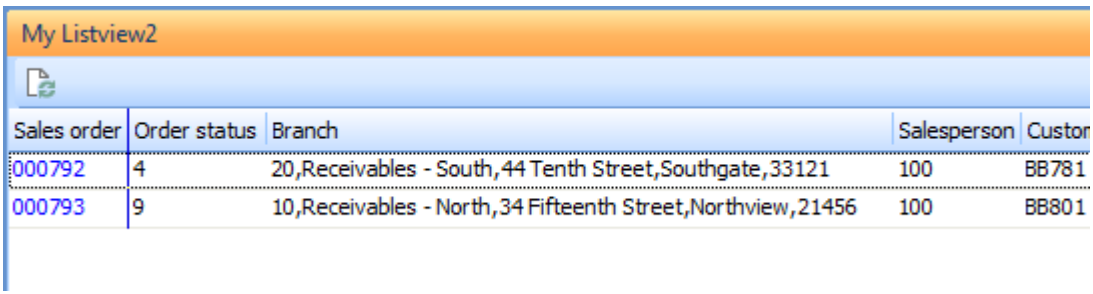


Figure 11-19: Setting the *Type='Address'* attribute against a column

The individual pieces of information are supplied to the cell delimited with *Carriage Return Line Feed* characters, which can be performed using the *VbCrLf* string constant. Below is what was supplied to the *Branch* element to build up the address for the first row in **Figure 11-19**.

```
<Branch>20" & VbCrLF & "Receivables - South" & VbCrLF & "44 Tenth Street" & VbCrLF
& "Southgate" & VbCrLF & "33121</Branch>
```

Alternatively, the address can be built up using information extracted from other locations, such as in the example below where it is extracted from another form. This is one line of code, although it has wrapped around on this page.

```
<Branch>" & ContactDetails.CodeObject.SoldAddress & VbCrLF &
ContactDetails.CodeObject.SOLD2 & VbCrLF & ContactDetails.CodeObject.SOLD3 & VbCrLF
& ContactDetails.CodeObject.SOLD4 & VbCrLF & ContactDetails.CodeObject.SOLD5 &
"</Branch>
```

If you attempt to extract this cell's content for use somewhere else, such as when using the *OnRowSelected* event, the content of this cell will just be text (without the carriage return line feed characters, and without the commas).

If the listview's *Style* is set to *Form* instead of *DataGrid*, there is no concept of a column, so the *Types* are ignored.

Decimals Attribute

The *Decimals* attribute is used in conjunction with the *Numeric* attribute to specify how many decimals must be displayed, and consequently cause the content of the column to be relative to the position of the decimal point.

If the number of decimal places for the supplied value is less than the amount set against the column, the value is padded to the right with zeros. If the number of decimal places for the supplied value is greater than the amount configured against the column, the least significant digits are dropped. For example, if the column is defined as using three decimal places (using the *Decimals='3'* attribute) and the value 10.5 is supplied, it will be displayed as 10.500. If 15.87932 is supplied the displayed value is 15.879. Reading back this value from the listview will return 15.879 as the rest of the number has been truncated.

Alignment Attribute

The *Alignment* attribute specifies where the value must appear within the cells of this column. The options are *Left*, *Right*, and *Center*. Because the position of the values within the cells of this column is stored within your operator preferences each time that you exit the program, you must either set this attribute when you create the listview, or use the *Reset View* (*Ctrl+F5*) option against the listview to see your change. If you attempt to set this attribute later without using the *Reset View* option, your operator preferences will override your new setting.

Figure 11-20 shows where the alignment for the *Sales Order* column has been set to *Left*, **Figure 11-21** shows where it has been set to *Center*, and **Figure 11-22** shows where it has been set to *Right*.

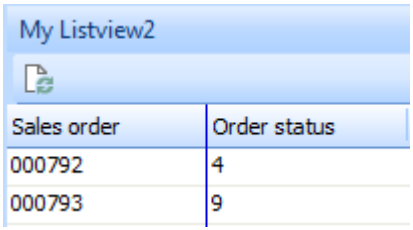


Figure 11-20: Using the *Alignment='Left'* attribute

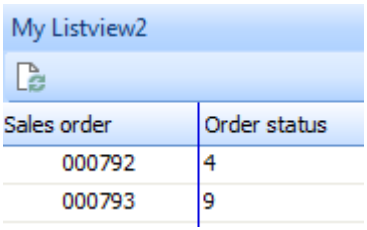


Figure 11-21: Using the *Alignment='Center'* attribute

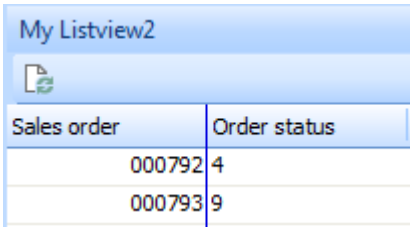


Figure 11-22: Using the *Alignment='Right'* attribute

HdrAlignment Attribute

The *HdrAlignment* attribute is used to specify the location of the column header. The options are *Left*, *Right*, and *Center*. Like the *Alignment*, the position of the column header is also stored in the operator preferences file, so has the same restriction to moving the header after the initial creation of the customized pane.

Hidden Attribute

Setting the *Hidden* attribute to *True* for a column will cause the column not to be displayed by default. The column will be available within the *Field Selector*, and can be dragged onto the listview. This is

useful when you want to create a listview customized pane that can be used by several operators/roles. You can add all the columns that they could possibly need, and then make only the most commonly required ones visible. Once the operator has used this listview, the settings will be saved as their preference.

If the operator is a not a member of a role, the saving of any listview changes is dependent on the *Save list view layout when using roles* operator *Activity* on the *Security* tab in the *Operator Maintenance* program. If the operator is a member of a role and the *Role configuration* is set that *Activities and fields* are *Configured by role*, this is configured using the browse against the *Activities and fields* option in the *Role Management* program (*Ribbon bar* | *Administration* tab | *Role Management* option | browse against *Activities and fields*).

Figure 11-23 shows the same listview customized pane as **Figure 11-7**, except that the *Branch* and *Salesperson* columns had their *Hidden* attribute set to *True*.

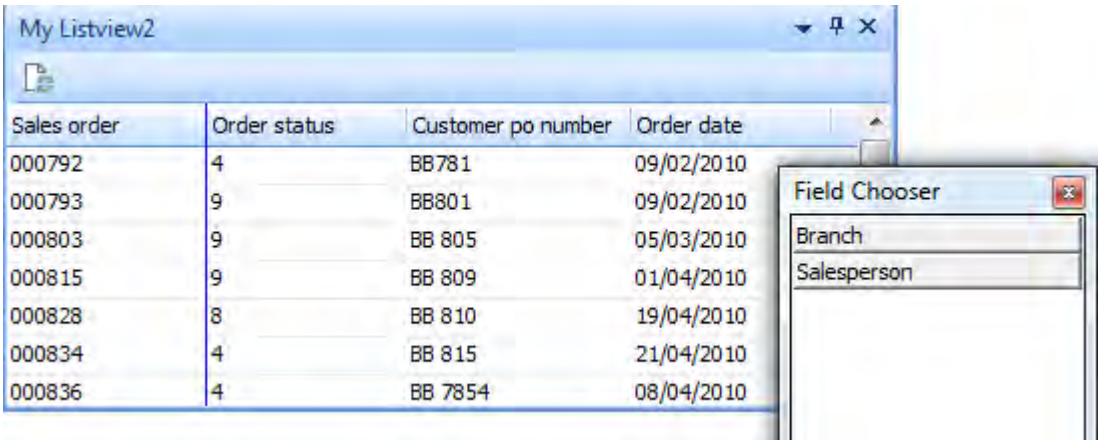


Figure 11-23: Using the *Hidden* attribute against the *Branch* and *Salesperson* columns

Editable Attribute

The *Editable* attribute is used to specify that the contents of this column are editable. Only those columns marked as *Editable* can have their values changed. Changing these values in the listview only changes them in the listview; it does not automatically change them at the source location. If you want to update the source location (or anything else), you can use the *OnAfterChange* event that fires when a value is changed. Within the *OnAfterChange* function you can access the *ListviewRowReturned* variable, which contains an array with the values in this row. The *ColumnClicked* variable contains the column array number of the cell that was changed so that you can pick out just that value, if required.

If one or more columns within the listview are configured as being dropdown lists, you need to set these columns to be *Editable*, otherwise the dropdown list will not work.

AllowRemove Attribute

The *AllowRemove* attribute is used to prevent operators from removing individual columns from the listview. An operator can drag column headers from the listview and they will appear in the *Field Chooser* so that the operator can put them back if they wish. **Figure 11-24** shows an operator dragging the *Salesperson* column header from the listview. The mouse pointer has changed to an X, and if the operator releases the mouse button the *Salesperson* column header will be removed from the listview, and will appear in the *Field Chooser*.

There are times that you don't want operators to be able to drag the columns from the listview, and for these columns you set the *AllowRemove* attribute to *False*.

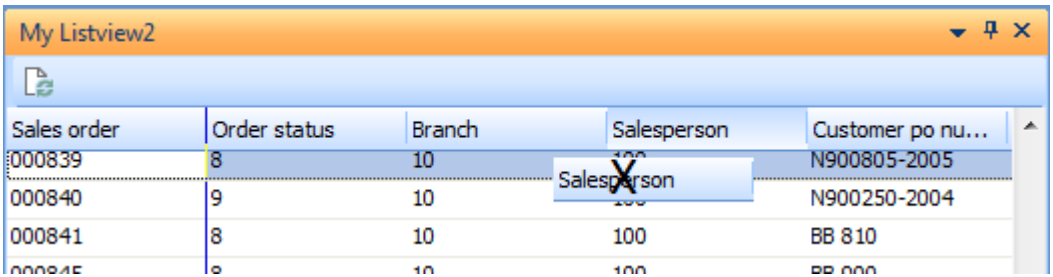


Figure 11-24: Dragging a column header from the listview

In **Figure 11-25** the *Customer po number* column has been configured with *AllowRemove='False'*. When the operator attempts to drag the column header from the listview the mouse pointer remains as it was, and when the operator releases the mouse button the column header returns to its location.

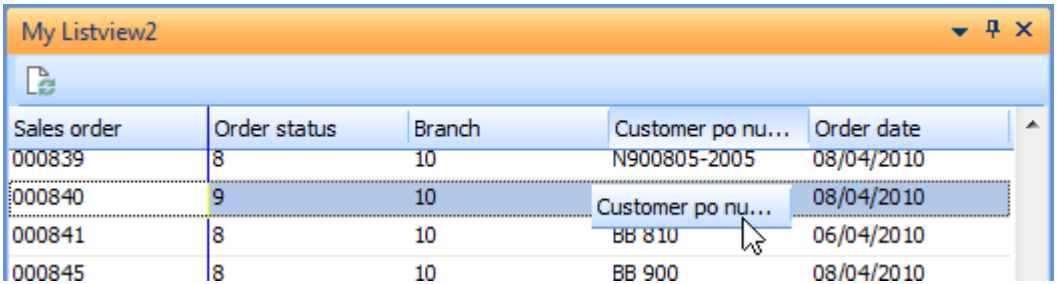


Figure 11-25: Dragging the column header from the listview when *AllowRemove='False'*

MaxLength Attribute
The MaxLength attribute is used to specify the maximum number of characters that can be entered in cells of a column if the column has been defined as editable. The following will limit the value that can be entered to five characters/digits:

```
MaxLength='5'
```

List Attribute
The *List* attribute is used in conjunction with the *Type* of *DropDown* to supply the values to be used in the dropdown list. An example of the *List* attribute appears below, which contains several sales order types.

```
List='Normal;Billing;Scheduled;Credit note;Debit note'
```

Tooltip Attribute
The *Tooltip* attribute is used to display up to 40 characters of text whenever the mouse pointer hovers over one of the cells of this column within the listview (see **Figure 11-26**). Hovering over the column header always displays the column header name as a tooltip, without requiring a tooltip attribute to be set.

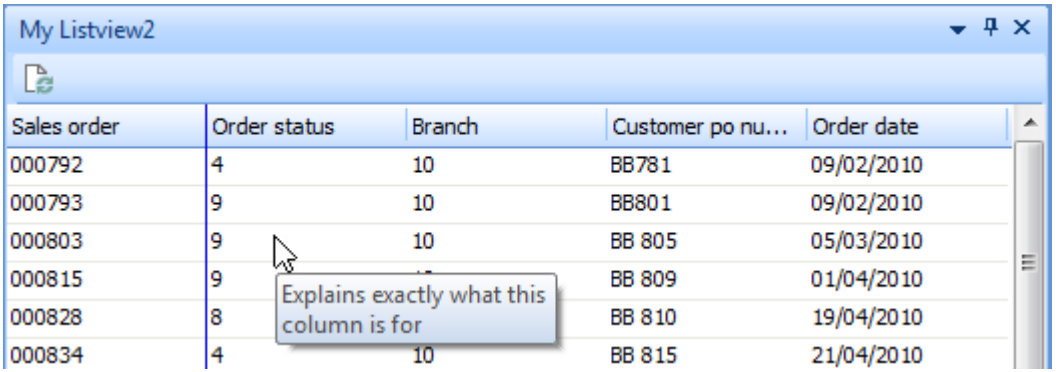


Figure 11-26: The *Tooltip* attribute displaying text when hovering over a cell

AutoColumnHide Attribute
The *AutoColumnHide* attribute is used to hide columns until the operator clicks on the icon against the column immediately before the hidden ones, at which point these hidden columns will all appear. An example of where this has been used is the *Sales by Product Class (incl. Trendline)* customized pane listview template that can be seen in **Figure 11-27**. This displays the *Branch*, *Product class*, and *Product class description* values, as well as a trendline showing the actual sales against budget. At the far right of the *Product class sales (Budget vs. Actual)* column header is a button.

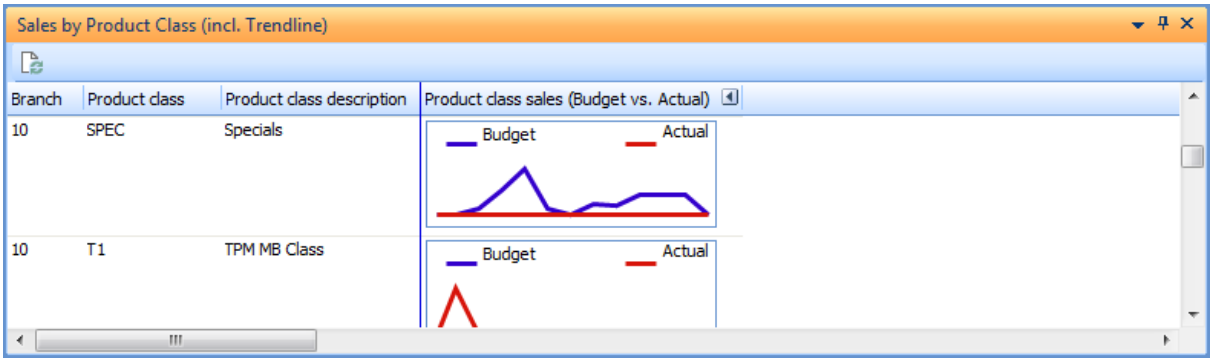


Figure 11-27: The *Sales by Product Class* template where the *AutoColumnHide* attribute is set

If the operator clicks this button the values from which the trendline was built are displayed to the right of this column (see **Figure 11-28**). Clicking on the button again will hide these values.

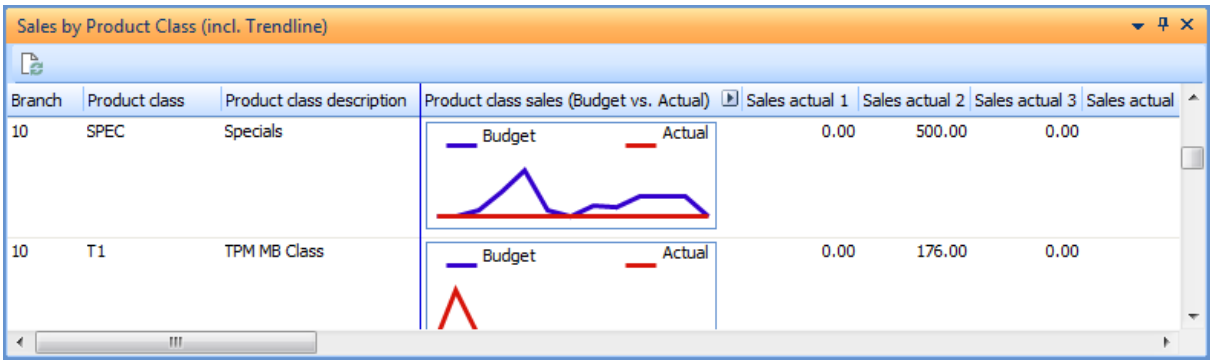


Figure 11-28: After the operator clicks on the button the other columns are displayed

The *AutoColumnHide* attribute specifies how many of the columns to the right of the current column must be hidden by default. In the case of the *Sales by Product Class* example, the *AutoColumnHide* attribute against the *GraphXAML* column (that contains the trendline) has its value set to 26, so the next 26 columns are hidden by default. The line of code making up this column appears below (which has wrapped around on this page).

```
<Column Name='GraphXAML' Description='Product class sales (Budget vs. Actual)'
XAMLCode='TLLineChart13_13' Width='210' AutoColumnHide='26' />
```

Sort Attribute

The *Sort* attribute is used to define the initial sort sequence of the listview. This attribute can be configured to ascending or descending sequence. Only one column can be defined with a *Sort*

attribute at a time. The example in **Figure 11-29** shows where the *Sort* attribute has been applied to the *Order status* column (which can be seen by the indicator alongside its column header).

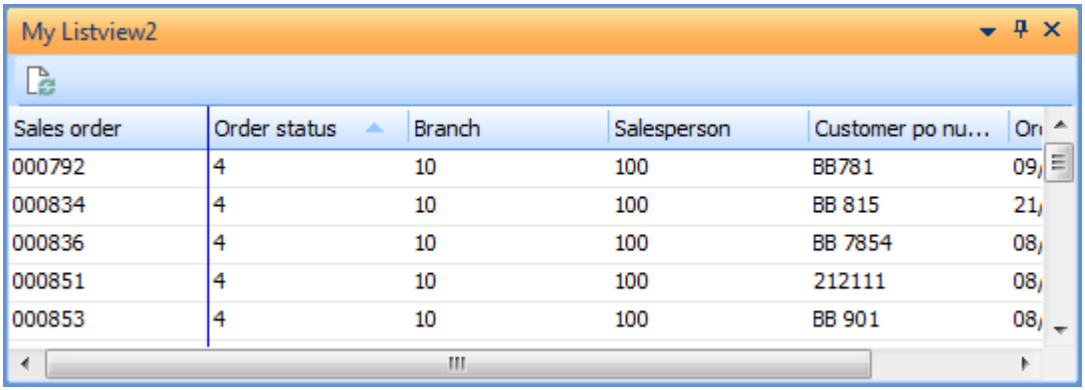


Figure 11-29: The *Sort* attribute applied to the *Order status* column

Total Attribute

The *Total* attribute is used to display a column total against specified totals, and is done by setting the attribute to *True* (see example below where the *Numeric* column is defined as having a total against it).

```
<Column Name='Num' Description='Numeric' Type='Numeric' Decimals='2'
Editable='true' Total='True' />
```

When this listview is displayed a total section appears at the bottom of the listview in an area called the footer, and any totals appear within it (see **Figure 11-30**).

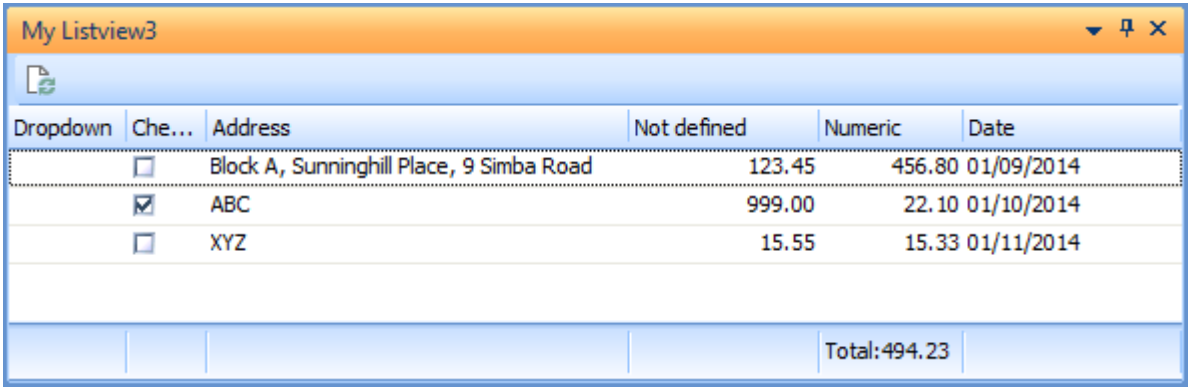


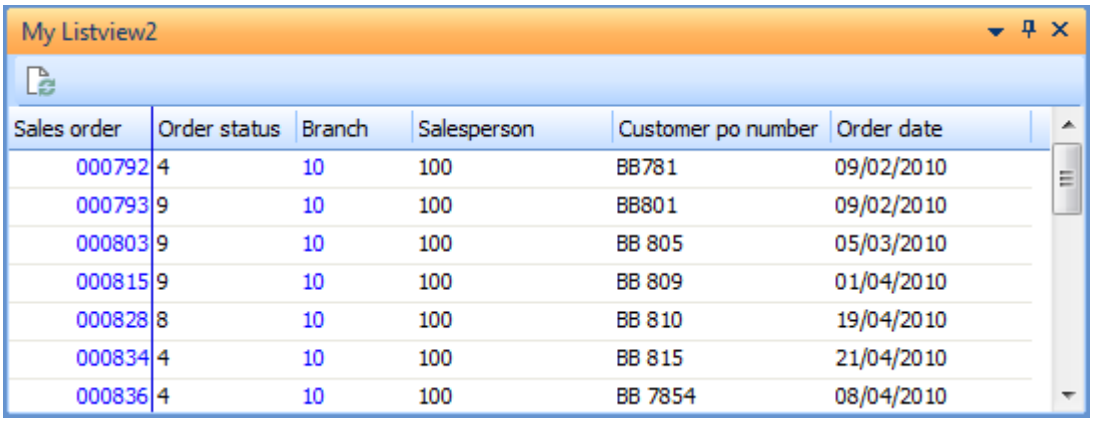
Figure 11-30: The *Total* attribute set to *True* for the column called *Numeric*

Columns defined as *Numeric* (or those that the listview assumes are numeric because of the value in the column's first row) can be defined as having a total. If a column defined as a *Checkbox* has a *Total* attribute against it, the total number of items checked will appear against the total. Setting any other column type to have a total against it may produce values (if the column contains numeric values) but these will likely be meaningless.

Link Attribute

The *Link* attribute is used to define that all the values in the rows of this column are hyperlinked. In **Figure 11-31** both the *Sales order* and *Branch* columns have been defined with the *Link='True'* attribute. Their color changes to that defined for hyperlinks. If the operator clicks on one of these hyperlinked values, the *OnLinkClicked* event will fire. If code exists against the *OnLinkClicked* function then this code will be run.

Against the *OnLinkClicked* function you can detect which row the clicked item belongs to as the contents of this row can be accessed using the *ListviewRowReturned* variable (from the *CustomizedPane* section of the *Variables* pane). This variable contains the contents of this row as a tab delimited list.



Sales order	Order status	Branch	Salesperson	Customer po number	Order date
000792	4	10	100	BB781	09/02/2010
000793	9	10	100	BB801	09/02/2010
000803	9	10	100	BB 805	05/03/2010
000815	9	10	100	BB 809	01/04/2010
000828	8	10	100	BB 810	19/04/2010
000834	4	10	100	BB 815	21/04/2010
000836	4	10	100	BB 7854	08/04/2010

Figure 11-31: The *Sales order* and *Branch* columns defined as hyperlinks

Using the listview in **Figure 11-31** as an example, the *OnLinkClicked* function contains the following code that displays the contents of the *ListviewRowReturned* variable.

```
Function CustomizedPane_OnLinkClicked()
  msgbox CustomizedPane.CodeObject.ListviewRowReturned
End Function
```

Clicking on either the sales order number or branch code hyperlinks on the third line would put the contents of this row into the variable, and the message box will display this (see **Figure 11-32**).

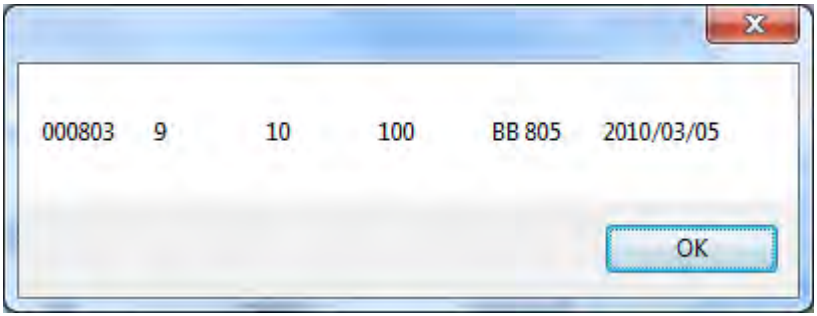


Figure 11-32: The contents of the *ListviewRowReturned* variable displayed in a message box

It is more likely that you will want to break up the contents of this list so that you can use one or more of the individual values. The following code creates a variable, and then uses the VBScript *Split* function to separate the individual values (that are delimited with tabs) and populate this variable as an array). The last line of the code displays the contents of the *Customer po number* column (which is the fifth column, but because arrays start at zero, this will be array entry 4). This is a single dimension array as it only contains a single row, so you only need to provide the column number, not the row and column number that you will see in other examples.

```
Function CustomizedPane_OnLinkClicked()
  Dim MyArray
  MyArray = Split(CustomizedPane.CodeObject.ListviewRowReturned, VBTab)
  msgbox MyArray(4)
End Function
```

In this example, more than one column in the row is hyperlinked. The *OnLinkClicked* event does not know which of the two columns were clicked within this row. However, you can use the *ColumnClicked* variable (under the *CustomizedPane* section of the *Variables* pane) to return the column number that was clicked. To return the array entry number you would need to subtract 1 from this column number to get the array entry number. Below is the modified code to display the value of the hyperlinked cell (note that 1 is subtracted from the *ColumnClicked* value against the array in the message box statement).

```
Function CustomizedPane_OnLinkClicked()
  Dim MyArray
  MyArray = Split(CustomizedPane.CodeObject.ListviewRowReturned, VBTab)
  msgbox MyArray(CustomizedPane.CodeObject.ColumnClicked - 1)
End Function
```

GroupBy Attribute

The *GroupBy* attribute is used to sort and group the contents of a column. Only one column can be associated with the *GroupBy* attribute. **Figure 11-33** shows a listview where the *GroupBy* attribute has been applied to the *Order status* column. This column is automatically sorted and the *GroupBy* logic

applied to it in the same way as if the *Group by This Field* option was selected from the listview's context-sensitive menu by the operator. The *GroupBy* attribute can only be applied when creating the customized pane, because once the pane has been used, the operator preferences takes over.

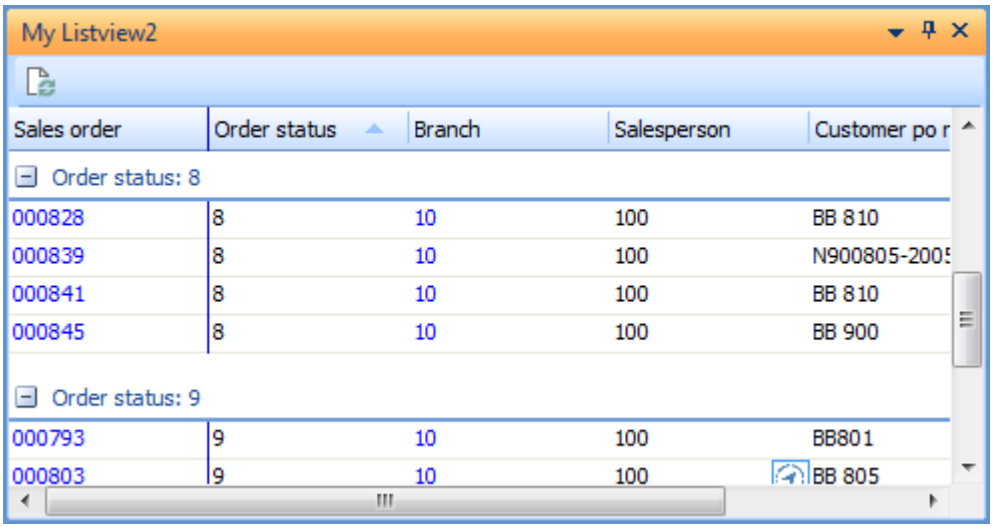


Figure 11-33: Using the *GroupBy* attribute to group by the contents of the *Order status* column

Footer Attribute

The *Footer* attribute enables you to add text to the footer section of the listview, as well as use certain variables to display values. These variables are surrounded by braces/curly brackets. The available variables are:

- *{rowcount}* displays the number of rows displayed in the listview
- *{total}* displays the column total
- *{checked}* displays the number of items that are checked in a checkbox column
- *{populated}* displays how many of the rows for this column are populated with data

The following five lines of code show examples of using the *Footer* attribute against the definition of the listview columns. Note that each of these lines has wrapped around on the page, and appears as if they are two lines. The first shows an example of displaying text in the footer.

```
<Column Name='HL' Description='Hyperlinked entry' Type='Alpha' Link='true'
Footer='Just text'/>
```

The second example defines the column as containing checkboxes, and the *Footer* attribute displays how many of the rows have their checkbox checked.

```
<Column Name='CB' Description='Checkbox' Type='Checkbox' Width='020'
Editable='true' Footer='Checked: {checked}' />
```

The third shows an example where the number of rows in the listview is displayed.

```
<Column Name='HL2' Description='Hyperlink 2' Type='Alpha' Link='true' Footer='Stock
codes: {rowcount}'/>
```

The fourth shows how to display how many rows are populated in this column.

```
<Column Name='Add' Description='Address' HdrAlignment='right' Width='050'
Editable='true' Type='Address' Footer='How many: {populated}' />
```

The fifth shows how to sum up the values for this column if it is defined as numeric.

```
<Column Name='Not' Description='Not defined' Editable='true' Footer='Total value:
{total}' />
```

The image in **Figure 11-34** shows how these will appear when the listview is rendered.

Hyperlinked entry	Checkbox	Hyperlink 2	Address	Not defined
A105	☐	B100	PO Box 77, Rivonia 2128,South Africa	123.45
A101	☑	B101	PO Box 123, Berea 2010, South Africa	999.00
A102	☐	B102		15.55

Just text	Checked:1	Stock codes:3		How many:2	Total value:1,138.00

Figure 11-34: The *Footer* attribute with text and *{rowcount}* variable against the *Order status* column

Width Attribute

The *Width* attribute is used to define the initial width of a column, in pixels. If you do not specify the *Width* attribute, the width of the column will be automatically determined by the listview when it is first displayed. The *Width* attribute only applies when the listview is created. Once the listview has been used by the operator the operator preferences will take over. An example of the *Width* attribute can be seen in the section covered in the *AutoColumnHide Attribute* section above.

Source Attribute

The *Source* attribute works in conjunction with the *PrimaryNode* attribute of the *Columns* element. The *PrimaryNode* attribute is used to specify the parent element name of the section to be displayed. Using the following simplified XML output from the *Inventory Query* business object (***INVQRY***), to display the details of the warehouses, the *PrimaryNode* would be configured as *WarehouseItem*.

```
<InvQuery>
  <StockItem>
    <StockCode>A100</StockCode>
    <Description>15 Speed Mountain Bike Boys</Description>
  </StockItem>
  <WarehouseItem>
    <Warehouse>E</Warehouse>
    <Description>Eastern Warehouse</Description>
    <QtyOnHand>10013.000000</QtyOnHand>
    <AvailableQty>-517.000000</AvailableQty>
  </WarehouseItem>
  <WarehouseItem>
    <Warehouse>FG</Warehouse>
    <Description>Finished Goods Warehouse</Description>
    <QtyOnHand>4997.000000</QtyOnHand>
    <AvailableQty>4997.000000</AvailableQty>
  </WarehouseItem>
</InvQuery>
```

The image in **Figure 11-35** shows a basic listview where the *PrimaryNode* has been set to *WarehouseItem*, and the *Warehouse*, *Description*, *Qty on hand*, and *Available qty* columns are being displayed.

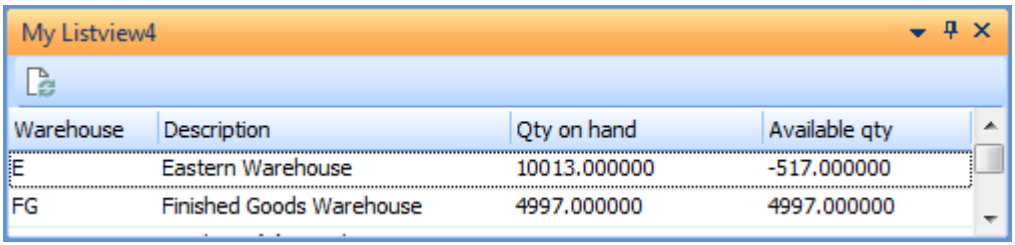


Figure 11-35: Using the *PrimaryNode* attribute to iterate through the warehouses

The XML to create this listview structure in **Figure 11-35** appears below. Note: some XML lines are wrapped to fit on the page.

```
<Columns PrimaryNode='WarehouseItem' Style='DataGrid' AutoSize='true'
FreezeColumn='0' AutoInsert='true' >
  <Column Name='Warehouse' Description='Warehouse' Type='alpha' />
  <Column Name='Description' Description='Description' Type='alpha' />
  <Column Name='QtyOnHand' Description='Qty on hand' Type='alpha' />
  <Column Name='AvailableQty' Description='Available qty' Type='alpha' />
</Columns>
```

If everything that you need for your listview falls within the *WarehouseItem* section of the XML, there is nothing else to do. However, if you also need to include information from an element that does not

appear within the *WarehouseItem* section of the XML, you can use the *Source* attribute to include this element.

Source Attribute

The *Source* attribute is used to provide the relative pathname to the element from the *PrimaryNode* that has been specified. For example, if you need to include the stock code in your listview, you could do so by setting the *Source* attribute to the relative location to the *StockCode* element. This is achieved by adding a *Column* element for *StockCode*, and including the following *Source* attribute.

```
Source='StockItem/StockCode'
```

The same can be done for the stock code's description. The two extra lines that are added to achieve this appear below, and the listview can be seen in **Figure 11-36**.

```
<Column Name='StockCode' Source='StockItem/StockCode' Description='Stock code'
Type='alpha' />

<Column Name='Description' Source='StockItem/Description' Description='Desc'
Type='alpha' />
```

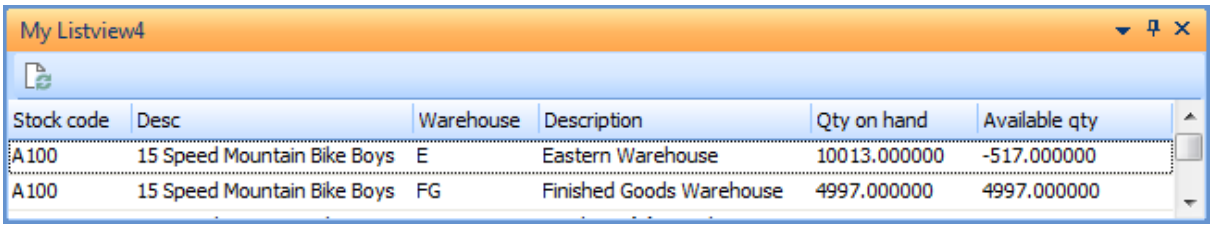


Figure 11-36: Using the *Source* attribute to add the *Stock Code* and *Description* to the listview

XAMLCode Attribute

The *XAMLCode* attribute is used to specify whether to apply a XAML theme to a column. Using one of the XAML themes enables you to display multiple pieces of information in one listview cell, change the appearance of the information, or highlight specific columns. **Figure 11-37** shows the same listview as **Figure 11-34**, but with the *AddressPlain* XAML theme associated with the *Branch* column. In addition, the address details of the branches have been injected into the XML to supply more information to the operator.

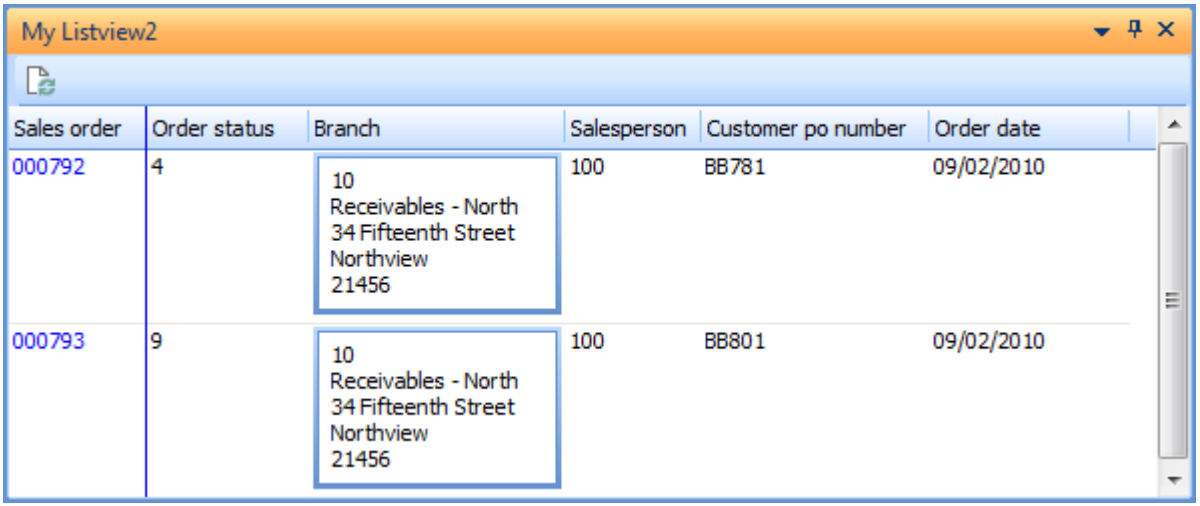


Figure 11-37: Using the *XAMLCode* field property to display multiple pieces of data in a cell

Using the Listview Designer

The section *Defining the Listview Structure* explained how to define the structure of the listview manually.The *Listview Designer* is a simple way to create the listview structure. The *ListviewDesigner* option is available from the *CustomizedPane* section of the *Variables* pane in the VBScript Editor. When this is selected the *Listview Designer* screen is displayed. This consists of a toolbar and three panes: *Properties*, *Column*, and *Columns*.

Properties Pane

The *Properties* pane consists of options that are used to build the attributes that will appear against the *Columns* element. **Figure 11-38** shows an empty *Properties* pane. Each of these options matches up with one of the attribute subsections under the *The Columns Element* section above, so are not covered in detail here. As each option is selected, a description of the attribute appears at the bottom of the pane. In this case the *Primary node* option has been selected, so the text at the bottom of the pane explains about this.

Column Pane

The *Column* pane is where you add columns that will appear as individual *Column* elements within the *Columns* node. Each of these options matches up with one of the attribute subsections under the *The Column Element* section above, so are not covered in detail here. As each option is selected, a description of the attribute appears at the bottom of the pane. In this case the *Max Length* option has been selected, so the text at the bottom of the pane explains about this.

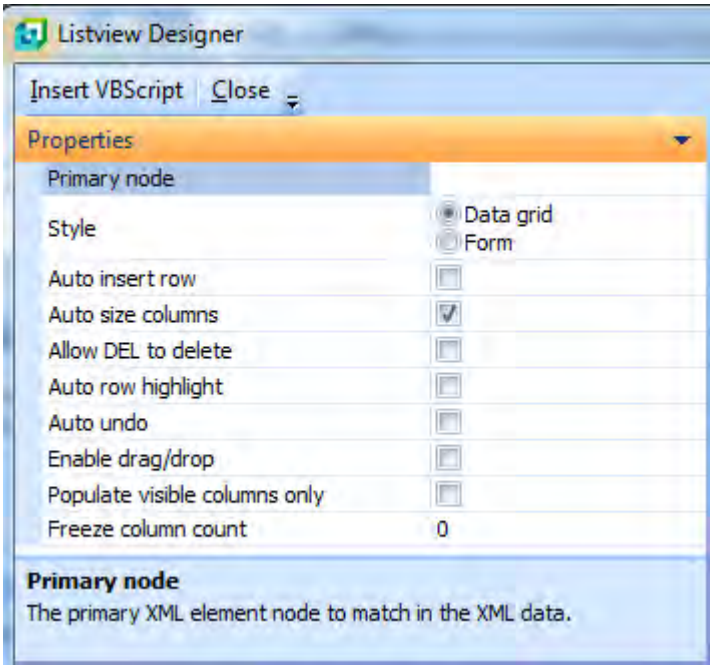


Figure 11-38: The *Properties* pane

Once you have supplied all of the information for this column you click on the *Add Column* button on the toolbar to add the column. Once it has been added, the populated options are cleared and this column will appear in the *Columns* pane.

Columns Pane

The *Columns* pane contains a list of all the columns that have been added while you have been in the *Listview Designer* screen (see **Figure 11-40**). As new columns are added using the *Column* pane they will be appended to this listview. The sequence can be changed by highlighting one of the rows and dragging it up/down the listview.

The options against a row cannot be changed within this listview. If a row contains incorrect information you will have to delete it (by highlighting it and using the *Del* key) and add it again.

When the list is complete, the *Insert VBScript* button on the *Listview Designer* toolbar adds the code to build this listview to your VBScript. It also adds sample code to populate it with a dummy row of data.

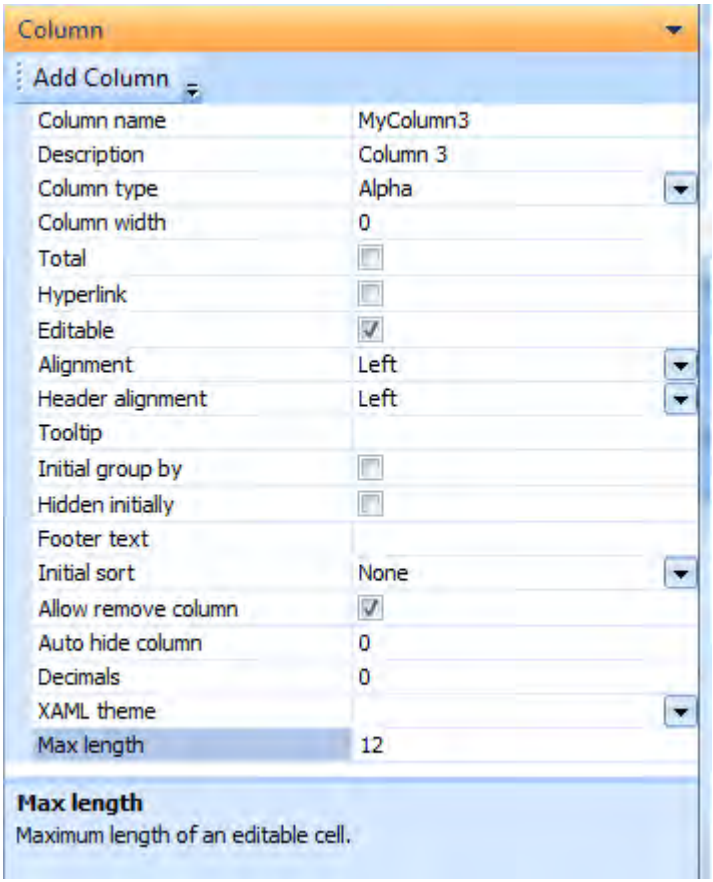


Figure 11-39: The *Column* pane

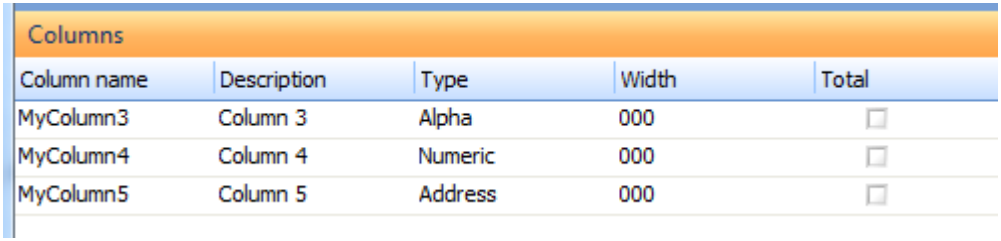


Figure 11-40: The *Columns* pane

Populating the Listview Using the Output from a Business Object

There are several ways that you can populate a listview with data. The most common is to use the output from a business object. Most *Query* class business objects contain repetitive information. This is held within a repeating section of the XML called a node.

```
<MyXML>
  <HeaderInfo>
    <Line1>My Job List</Line1>
  </HeaderInfo>
  <DetailInfo>
    <Jobs>
      <Job>123456</Job>
      <JobDescription>My Job</JobDescription>
      <QtyToMake>1</QtytoMake>
    </Jobs>
    <Jobs>
      <Job>3334456</Job>
      <JobDescription>Job number 2</JobDescription>
      <QtyToMake>5</QtytoMake>
    </Jobs>
    <Jobs>
      <Job>72318</Job>
      <JobDescription>Large Job</JobDescription>
      <QtyToMake>125</QtytoMake>
    </Jobs>
  </DetailInfo>
</MyXML>
```

The XML above is an example where there is information about several jobs. For each job there is a job number, a description and a quantity to make.

When defining the structure of a listview to display this data, the *PrimaryNode* would be set to *Jobs* within the *Columns* element, which would mean that the three *Jobs* nodes would be available. Each of the sub elements (*Job*, *JobDescription*, and *QtyToMake*) could then appear within a *Column* element, and they would appear as columns within the listview. This is typically configured within the listview's *OnLoad* function, as it will not change each time that the listview is refreshed.

Within the *OnRefresh* function the business object is called and its output passed to the *CustomizedPane.CodeObject.ListviewData* variable. The rest is handled by the customized pane, because it knows the structure of the listview, which is the primary node, and has the XML.

Using the Output from the Report Writer to Populate a Listview

The following assumes that you have licensed the use of the *Report Writer* module within SYSPRO. Providing that this is licensed, and the *Report Writer* business object is called from within SYSPRO, you will not need to license the *Report Writer* functional area.

The SYSPRO *Report Writer* can provide output in different formats, one of which is to XML. This makes it an ideal means of retrieving information from multiple sources and populating a listview.

Most of the work is done in the report itself. Some things must be done to make it work (such as setting the output to *XML*) while others can make the output better, or easier to achieve.

When you create the report, uncheck the *Create report with default headings* option (see **Figure 11-41**). This will save you time later on.

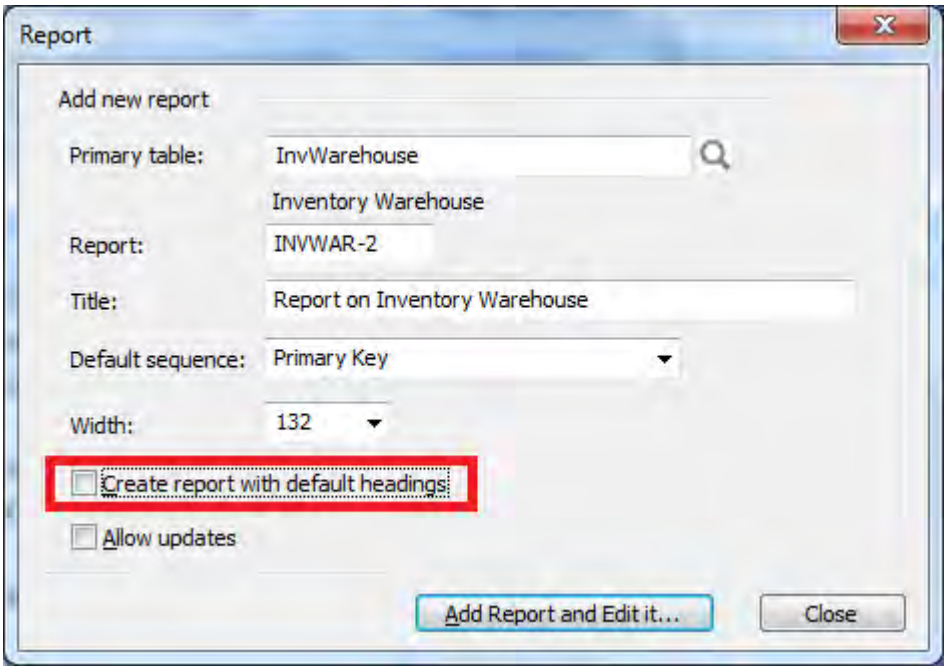


Figure 11-41: Unchecking the *Create report with default headings* option

Within the report (before you do anything else) you should call up the *Option* section. The second item in the list is *Output*, which defaults to *Report*. Double-clicking on this displays the *Report Options* screen which has a dropdown list against the *Report type* prompt. Select *XML* from this option (see **Figure 11-42**).

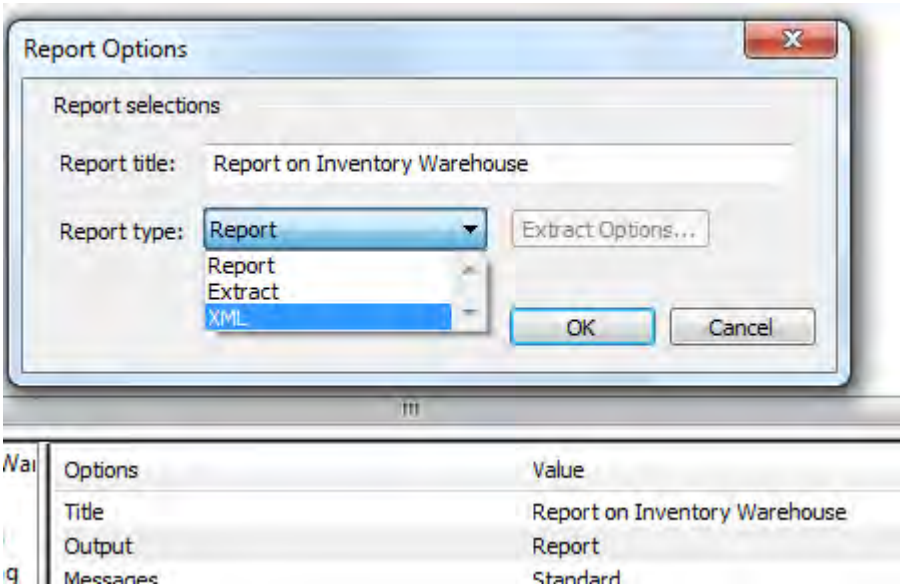


Figure 11-42: Selecting the *Report type* of *XML*

When the *Report type* is set to *XML*, the *XML Wizard* button becomes enabled. Clicking on this starts the wizard. The first screen is just for information, and the second screen is used to define the name of the file to hold the report output, and its location.

Some of the things that you can do with the XML output is apply a style sheet to improve its looks, output summary metadata, and drill down options so that you can display the summary information and then drill down within the report. Unless this report is going to be used for this function too, uncheck all of these options on the wizard's screen.

The next screen of the wizard enables you to define the element names such as the root element, the element containing the header information, the element containing the group information etc. You can leave these at their default settings, but to make it easier to follow you can replace the default setting for *Report detail* element to *Row*. When you finish the wizard the area at the top of the screen will become a browser, and it will be populated with the root element (see **Figure 11-43**). Click on the *OK* button to close the *Report Options* screen.

Still within the *Options* section, double-click on the *Column Heading* section and change the option *First line to be used for headings* in the heading section from the default of 4 to 0. Click on the *OK* option to close the *Report Headings* screen.

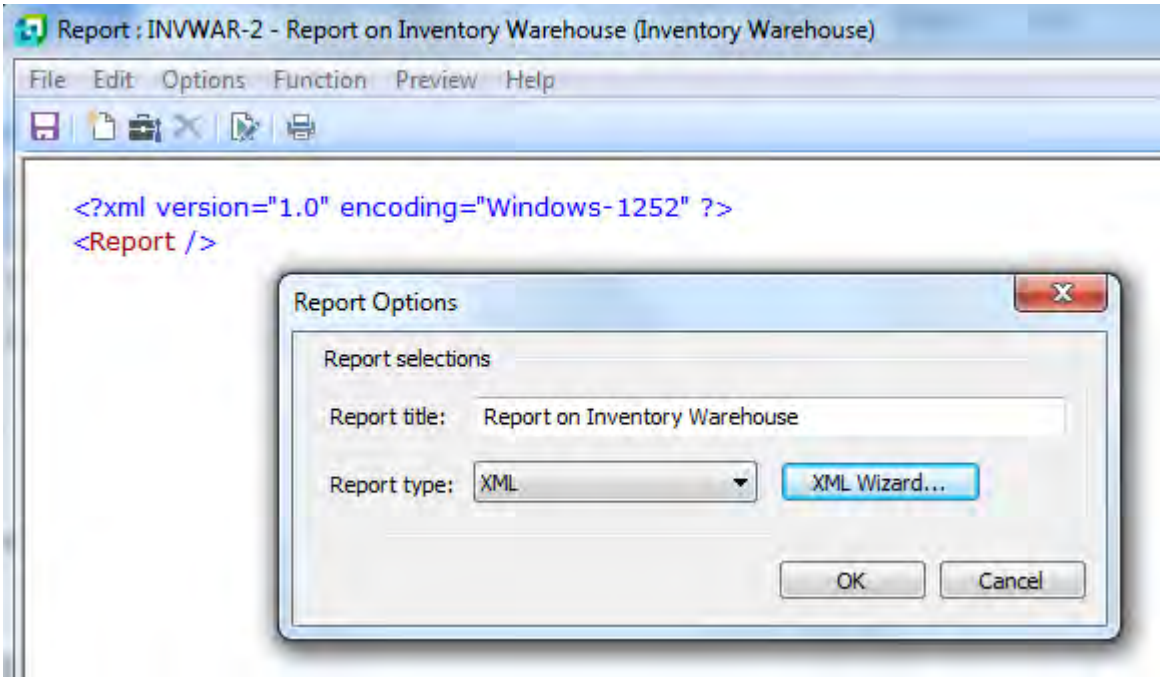


Figure 11-43: The browser screen showing the report preview

Click on the *Detail* option within the *Layout* section. The screen on the right will be a listview with the column headings of *Column name*, *Line*, *Column no*, *Update*, *Value*, and *Condition*. This is where you add the detail fields to be output. Right-click on the listview and select *Add*. The *Field* screen is displayed and this contains five tabs (*Details*, *Heading*, *Condition*, *Update* and *XML*).

The *Field* screen is used to add fields in the same way as reports with other output formats. However, there are things that you can do to simplify your XML output, and make it easier to consume in your customized pane listview.

On the *Heading* tab, select the *No heading* radio button. This will reduce the XML's size.

On the *XML* tab, check the *Define element name*. This enables you to supply your own meaningful element name for this field. In addition, if you are using a field from a linked table (i.e. from a table that is not the primary one for this report) the default element name will contain the table name and field name, which can get a little unwieldy.

As you add these fields a preview of the report will appear in the browser pane. **Figure 11-44** shows where the fields have been entered and the browser pane showing the preview. You can see that the

StockCode and *Description* element names match the column names, but the *QtyOnHand* column has had its XML element name changed to *OnHand* and so appears this way in the browser.

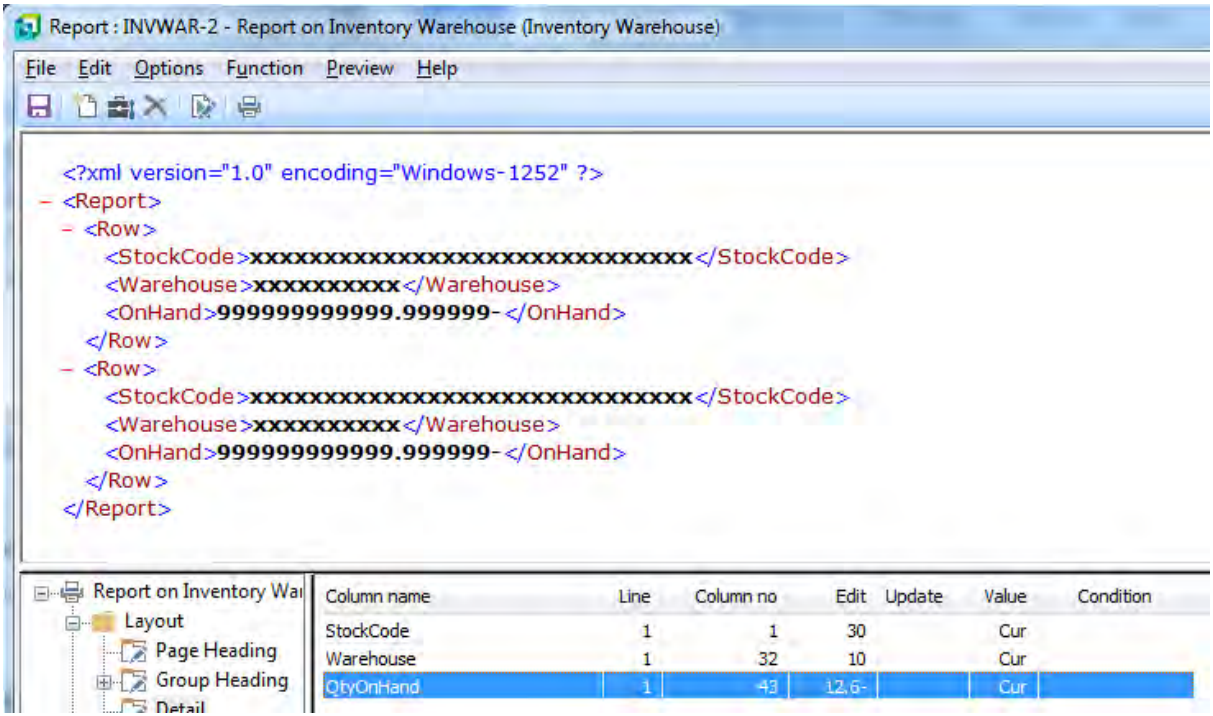


Figure 11-44: The preview pane showing a sample of the report output

This browser pane preview shows you the structure of the XML that will be output by the business object when it is run, so the listview structure in the customized pane must match this.

The code below contains the structure of the listview, and would appear within the customized pane's *OnLoad* function. It consists of seven lines, and appears below with a blank line between each line to make it easier to follow, as many of the lines wrap around on this page.

The *Name=* attribute within each of the *Column* elements must exactly match the output of the XML, or the column will not be populated.

```
dim ListXML

ListXML = ListXML & "<Columns PrimaryNode='Row' Style='DataGrid' AutoSize='true'
FreezeColumn='0' AutoInsert='false' >"

ListXML = ListXML & "<Column Name='StockCode' Description='Stock Code' Type='Alpha'
Alignment='Left' HdrAlignment='Left' />"

ListXML = ListXML & "<Column Name='Warehouse' Type='Alpha' Alignment='Left'
HdrAlignment='Left' />"

ListXML = ListXML & "<Column Name='OnHand' Description='Qty on Hand' Type='Numeric'
Decimals='3' Alignment='right' HdrAlignment='Left' />"

ListXML = ListXML & "</Columns>"

CustomizedPane.CodeObject.ListviewProperties = ListXML
```

The business object that is invoked to run the report is called ***REPQRY***, and it is called in the same way as any other business object. This can be added to the end of the *OnLoad* function, if it is only to be run once per run of the customized pane. Alternatively, if required to be refreshed on demand, or automatically after a set time interval, it should be added to the customized pane's *OnRefresh* function.

You can use the *Call Business Object* button on the toolbar of the editor to invoke the *Call a Query Business Object* wizard to build this code for you. Supply the business object name ***REPQRY*** at the *Query Business Object prompt* and *Tab* off this field. The sample XML for this business object will populate the *Parameters* tab. Replace this with the following that just specifies the name of your report.

```
<Query>
  <Key>
    <Report>INVWAR-2</Report>
  </Key>
</Query>
```

Click on the *Insert VBScript Code* button on the toolbar and the code to call the ***REPQRY*** business object will be inserted for you. All that is required is to add a line to pass the output from the business object (that is in the *XMLOut* variable) to the *CustomizedPane.CodeObject.ListviewData* variable. The completed code appears below.

```
dim XMLOut, XMLParam
XMLParam = XMLParam & " <Query>"
XMLParam = XMLParam & "   <Key>"
XMLParam = XMLParam & "     <Report>INVWAR-2</Report>"
XMLParam = XMLParam & "   </Key>"
XMLParam = XMLParam & " </Query>"
on error resume next
XMLOut = CallBO("REPQRY",XMLParam,"auto")
if err then
  msgbox err.Description, vBCritical, "Calling Business Object"
  exit function
end if
' Switch on error handling
on error goto 0

CustomizedPane.CodeObject.ListviewData = XMLOut
```

The end results can be seen in **Figure 11-45**.

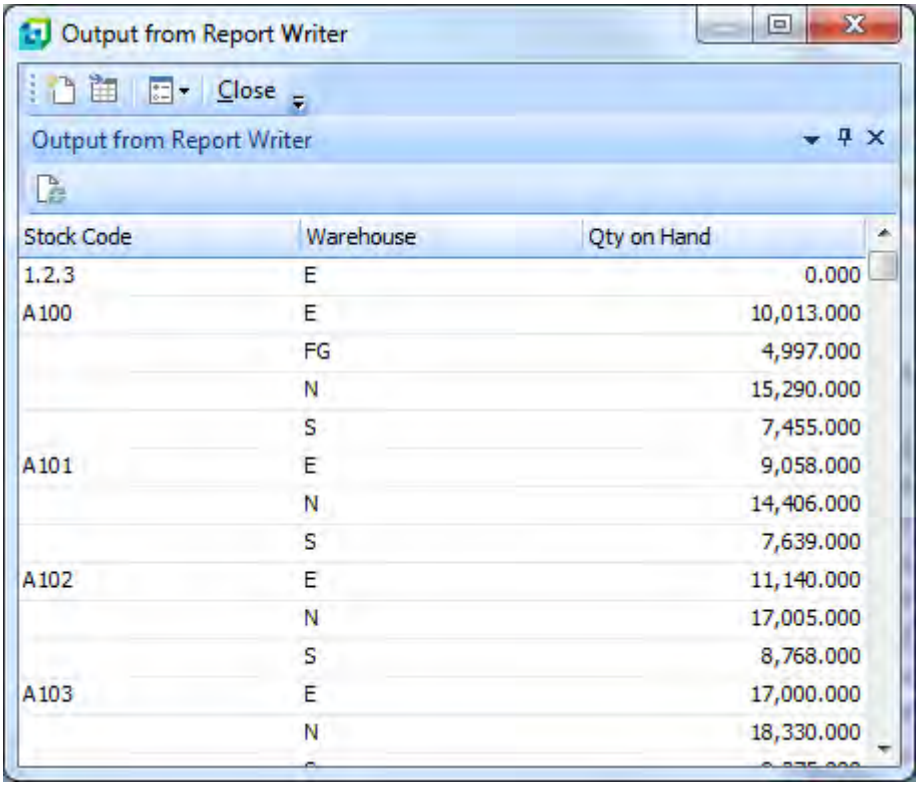


Figure 11-45: The finished result

Populating the Listview without Using XML

Using XML is the simplest way to populate a customized pane listview. When the data to be displayed is not too large, or it is already in XML, it does not take a long time to render the results. However, if there is a lot of information to be displayed, and it is not already in XML (such as being extracted from SQL Server), it can be more efficient to put the data into the listview's native format and let it handle the display. If you use XML you have to convert the existing data into XML, only to have SYSPRO unpack it so that it is rendered in the listview.

The structure of the listview is defined using XML as covered above, regardless of whether you are going to use XML to populate it.

The native format of the listview is that each cell of the listview row is separated using the ASCII code 255. The last cell on a row has the *Carriage Return* and *Line Feed* characters (there is no ASCII 255 character after the last cell of a row before the *Carriage Return* and *Line Feed* characters). This is probably best explained with an example.

Non-XML Example
This example was built using the *Application Builder* option from the *Administration* tab of the SYSPRO Ribbon bar. The *Application Builder* dropdown list contains 20 empty applications (***IMPDH1*** - ***9*** and ***IMPDHA*** - ***H***). These are empty shells to which you can insert one or more customized panes. In this example a single customized pane listview has been added.

Use the *Application Builder* dropdown list to select the application to be used. In this example *Application 2* was used. Not strictly necessary, but worth doing to make life easier later, is to change the display name of the application. This is performed from the toolbar of the application (see **Figure 11-46**).

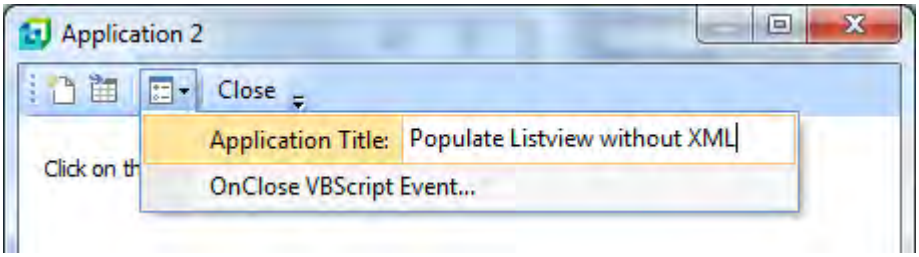


Figure 11-46: Changing the title of an *Application Builder* application

Click on the *Add a Customized Pane* button on the toolbar to start adding the customized pane listview. Set the *Object type* to listview, add a *Window title*, and click on the *Edit VBScript* button on the toolbar. The *VBScript Editor* screen will be displayed. On the *VBScript Editor* double-click on the

OnLoad event name. The screen where you edit the VBScript will be displayed, the *CustomizedPane_OnLoad* function will be created for you, and the cursor placed within it.

Double-click on the *ListviewDesigner* option within the *CustomizedPane* section of the *Variables* pane. This will load the *Listview Designer* screen, which you should use to build the listview structure. The *PrimaryNode* should be set to *MyRow*.

The listview should have three columns called *When*, *Quality*, and *Amount*. The *When* field should be configured as *alphanumeric*, with a description of *When this was sold*. The *Quality* field should be configured as *alphanumeric* with a description of *Unit quality*. The *Amount* field should be defined as numeric with two decimals, and have the description of *How many tons were sold*.

Once you have built this listview structure, use the *Insert VBScript* button on the toolbar to add the code to your *OnLoad* function.

Exit back to the *VBScript Editor* screen and double-click the *OnRefresh* event name to create this for you. If you were using XML to populate this listview the XML would be similar to the following:

```
dim ListData
ListData = "<MyRoot>"
ListData = ListData & "<MyRow>"
ListData = ListData & "<When>2014-06-25</When>"
ListData = ListData & "<Quality>Good</Quality>"
ListData = ListData & "<Amount>123.99</Amount>"
ListData = ListData & "</MyRow>"
ListData = ListData & "<MyRow>"
ListData = ListData & "<Amount>30.25</Amount>"
ListData = ListData & "<When>2014-08-25</When>"
ListData = ListData & "<Quality>Bad</Quality>"
ListData = ListData & "</MyRow>"
ListData = ListData & "</MyRoot>"

CustomizedPane.CodeObject.ListviewData = ListData
```

Note that the last line of code passes the XML to the *CustomizedPane.CodeObject.ListviewData* variable which processes it and displays it. The sequence of the elements within the *MyRow* node is not important as the data is extracted using the element name. In the XML example the sequence is different for each *MyRow* node. If there is no element for one of these columns in one of the rows it is ignored, as it is not mandatory.

However, this example does not use XML, so the values to be passed to the individual cells must be in the correct sequence and separated by the ASCII character 255. The rows must also be terminated with the *Carriage Return* and *Line Feed* characters. If a cell must be empty then the ASCII character 255 for this cell must still be present.

The following VBScript code builds up four rows containing the content for this listview, and passes it to the *CustomizedPane.CodeObject.ListviewData* variable. A blank line has been inserted between each of these lines to make it easier to read, as several of the lines wrap around on this page.

The first line defines the variable that is used to contain all of the information as it is built up. The last line takes the contents of this variable and passes it to the *CustomizedPane.CodeObject.ListviewData* variable so that it is processed/displayed.

The second line adds the values of the three cells to the variable *MyValues*, separated with the ASCII character 255. After the third value is a *String Constant* of *VbCrLf*, which is used to provide the *Carriage Return* and *Line Feed* characters.

The third, fourth and fifth lines are similar to the second line except that they append to the content of the *MyValues* variable.

The sixth line passes the content of the *MyValues* variable to the *ListviewData* variable so that it can be displayed.

```
Dim MyValues

MyValues = "2014-01-31" & Chr(255) & "Best" & Chr(255) & "100.25" & VbCrLf

MyValues = MyValues & "2014-01-31" & Chr(255) & "Not so good" & Chr(255) & "30.5" &
VbCrLf

MyValues = MyValues & "2014-02-28" & Chr(255) & "Best" & Chr(255) & "105.10" &
VbCrLf

MyValues = MyValues & "2014-02-28" & Chr(255) & "Not so good" & Chr(255) & "22.75"
& VbCrLf

CustomizedPane.CodeObject.ListviewData = MyValues
```

The results of this can be seen in **Figure 11-47**.

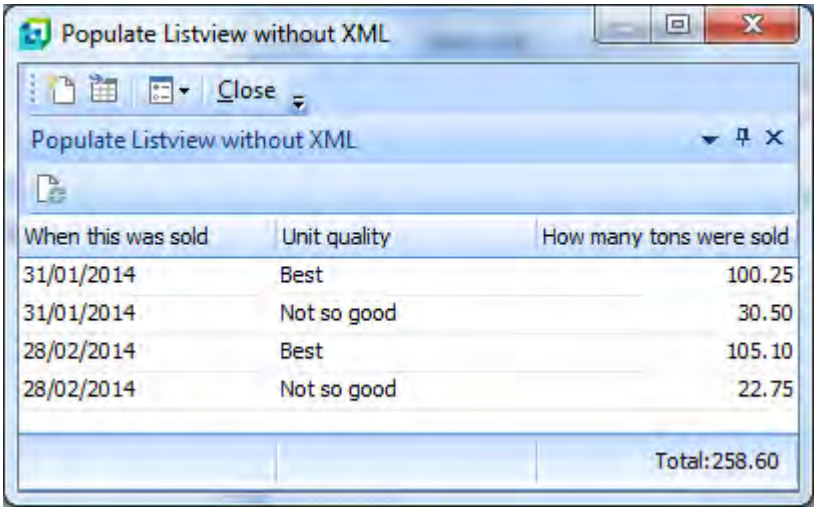


Figure 11-47: The finished results

Populating Cells with Values within the OnPopulate Function

When using the *OnPopulate* function to supply a value, the *_OUT* array must be used.

Alpha and Numeric Cells

The cell of a column defined as *Alpha* or *Numeric* can be populated with a value in one of two ways. The first is using the *Field Property* of *Value*, and the second is just providing the value as a string. When using the *Field Property* you must supply the array variable name of the cell, followed by a space, an equal sign, and another space. Expand the *Field Properties* pane, populate the *Value* property and click on the *Insert VBScript Code* button. The following code snippet was created when the *Value* property was populated with *XXX*. It will replace the current contents of Column 2, Row 1 with *XXX*.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,0) = "<Field Value='XXX'></Field>"
End Function
```

The same result could be achieved using the following code.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,0) = 'XXX'
End Function
```

Date Cells

Date cells can be populated in a similar way to cells belonging to *Alpha* and *Numeric* columns. The main difference is that there is a strict format for the date to be supplied. It must be supplied in the format *CCYY-MM-DD* or *CCYY MM DD* (the delimiters being hyphens or spaces). The following code snippet supplies the date of 20th October 2014 with hyphens, using the *Field Properties* method.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(5,0) = "<Field Value='2014-10-20'></Field>"
End Function
```

This code snippet supplies the date of 28th February 2015 with spaces, as a string.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(5,0) = "2015 02 28"
End Function
```

Checkbox Cells

There are two pieces of information that can be supplied to a cell within a column that has been defined as a *Checkbox* column. These are the status of the checkbox (checked or unchecked), and text to appear alongside the checkbox. If the value Y or 1 is passed to the cell, the checkbox is checked. If the value N or 0 is passed to the cell, the checkbox is unchecked. If any other value is passed to the cell it appears as text alongside the checkbox.

Figure 11-16 shows the results of passing the value of *XXX* to the cell in column 2, row 2, and this text appears alongside the checkbox. Either of the two code snippets below could have been used to add the value *XXX* alongside this checkbox.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='XXX'></Field>"
End Function
```

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "XXX"
End Function
```

Either of the following two code snippets could be used to check the checkbox alongside the *XXX* value in **Figure 11-16**.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='1'></Field>"
End Function
```

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "1"
End Function
```

It is possible to set the status of the checkbox, as well as supply the text that must appear alongside it, in the same line of code. This is done by supplying the checkbox status in braces/curly brackets, followed by the text. The following code snippet sets the checkbox to a checked status, and the text alongside the checkbox to *XXX*.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "<Field Value='{1}XXX'></Field>"
End Function
```

Just like with the other code snippets above, this can also be supplied as a string. The following code snippet set the checkbox status to unchecked, and puts the word *Done* alongside the checkbox.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(1,1) = "{0}Done"
End Function
```

Address Cells

Address cells can also be populated by supplying a string. The following code snippet will add an address to a cell within a column that has been defined as an *Address* column. The result can be seen in the third column or the first row in **Figure 11-19**. Each part of the address is delimited with a carriage return line feed character (using the *VBCrLf* string constant in VBScript). Note that the line populating the address has wrapped around onto two lines in the snippet below.

```
Function CustomizedPane_OnPopulate()
CustomizedPane_OUT.CodeObject.Array(2,0)= "20" & VbCrLF & "Receivables - South" &
VbCrLF & "44 Tenth Street" & VbCrLF & "Southgate" & VbCrLF & "33121"
End Function
```

Dropdown Cells

You cannot populate the dropdown list during the *OnPopulate* function, but you can prepopulate the cell with one of the values from the dropdown list, by supplying its entry number. The first entry in the list is 0, the second is 1, the third is 2, etc. When defining the structure of the listview, the available entries for the dropdown list are provided against an attribute called *List*. The following code snippet is extracted for the listview definition to create the listview in **Figure 11-18**, and creates the dropdown list against the first column (this is one line that has wrapped around).

```
ListXML = ListXML & "<Column Name='DD' Description='Dropdown' Type='DropDown'
Editable='true' List='Normal;Billing;Scheduled;Credit note;Debit note' Total='True'
/>"
```

The following code snippet would prepopulate the first row of this column with the *Billing* option from the dropdown list. *Billing* is the second item in the list, and has an entry number of 1.

```
Function CustomizedPane_OnPopulate()
  CustomizedPane_OUT.CodeObject.Array(0,0)= "<Field Value='1' > </Field>"
End Function
```

Note that unlike defining a dropdown list against a form (covered in Chapter 14) it is not possible to supply a code against each entry in square brackets (that is used to prepopulate the entry, and returned when the contents of the cell is read back). The prepopulating of the cell in a customized pane Listview is performed using the entry number, and if you read back the contents of a cell containing a value provided by a dropdown list, you will receive the entry number.

Adding Custom Form Fields to a Listview

Although this does not require any coding, this section has been added for the sake of completeness.

This functionality is only available if all of the following conditions are met:

- The SYSPRO company is configured to use SQL Server for data storage
- The option to store the custom form field information separately for each custom form has been selected (*Ribbon bar* | *Administration* tab | *Custom Forms*)
- None of the columns defined in the listview is configured to be editable (i.e. they have the *Editable='true'* attribute configured against them)

If any of the columns are configured as editable the option to *Add Custom Columns* in **Figure 11-48** will be greyed-out.

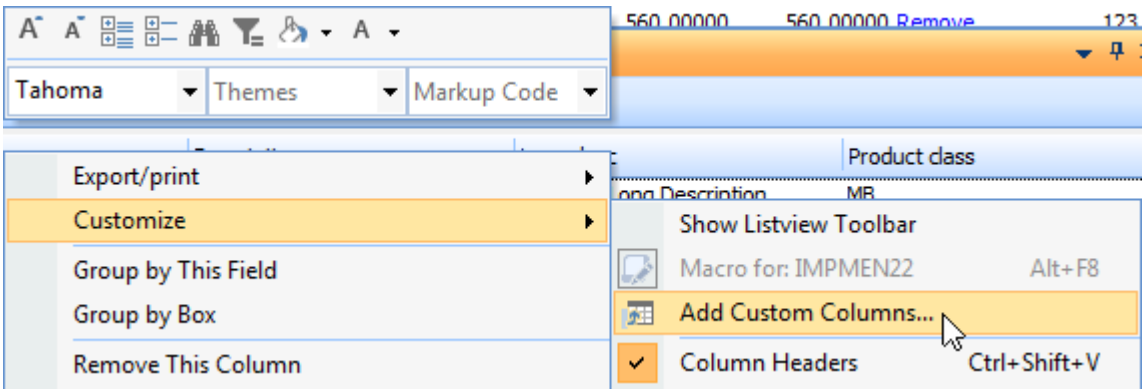


Figure 11-48: Selecting the *Add Custom Columns* option

If the customized pane listview has columns that contain key fields, it is an easy task to add custom form fields related to this key field. **Figure 11-49** shows a customized pane that lists out the stock code, description, long description and product classes of four stock codes.

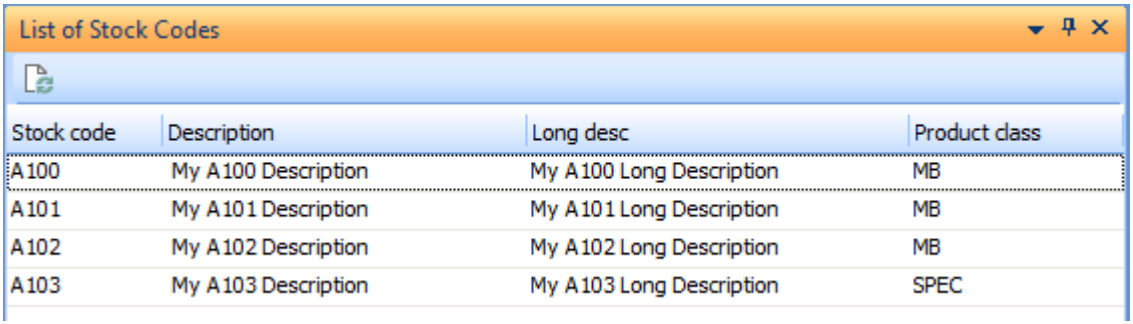


Figure 11-49: Listview showing information for four stock codes

Custom form fields can be added by right-clicking on one of the column headers and selecting *Customize* | *Add Custom Column* from the displayed menu (see **Figure 11-48**). The *Custom Columns for Listviews* pane is displayed.

Figure 11-50 shows the *Custom Columns for Listviews* screen which contains radio buttons to add a field from a custom form, to add a blank column, or to add a column from a master table. In this case the *Custom form column* option has been selected. When this is selected the dropdown list against the *Custom form type* prompt contains the custom forms associated with the columns in this listview. In this case only *stock code* is available.

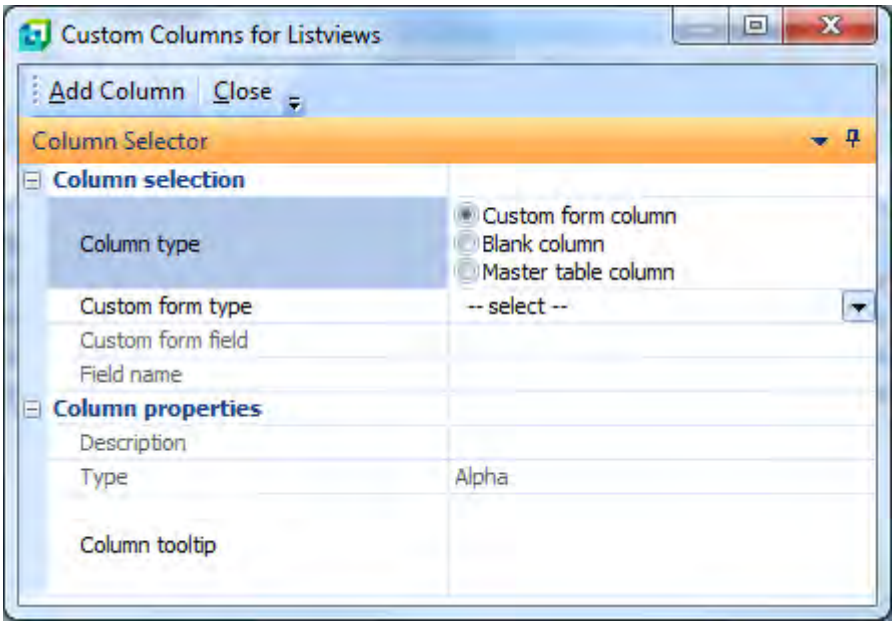


Figure 11-50: Selecting the *Custom form type* from the dropdown list

The list of column names that can be matched appears in the `IMPQVW.IMP` file in the SYSPRO program folder. The matching will be performed against the field names that start in column 1 of this file.

When the *Custom form type* is selected the dropdown list alongside the *Custom form field* prompt is populated with all the fields for this custom form (see **Figure 11-51**). The *Field name*, *Description* and *Type* fields are populated using the custom form field's information, and a tooltip can be added against the *Column tooltip* prompt, if required.

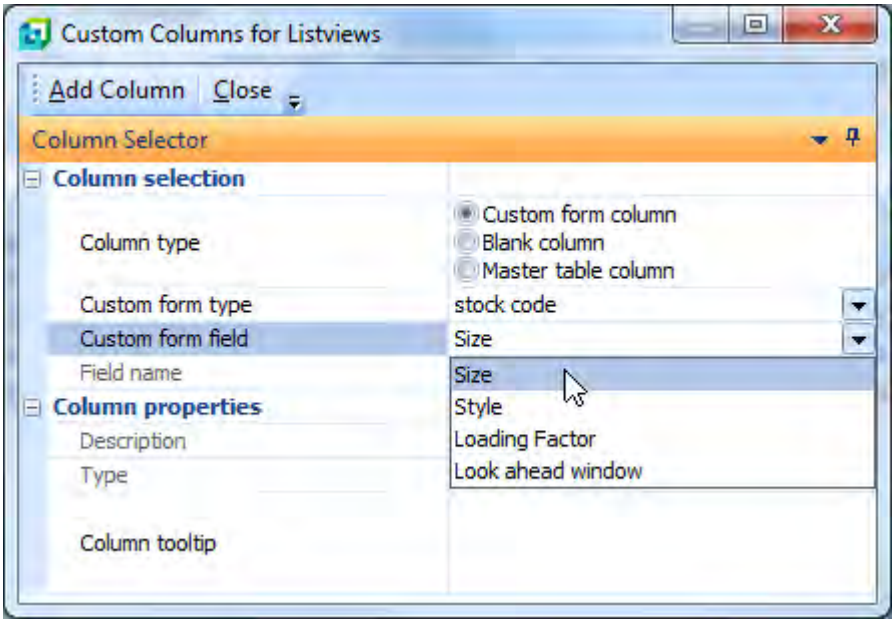


Figure 11-51: The custom form fields linked to the selected custom form

Clicking on the *Add Column* button on the toolbar will add this column to the listview, but leave you in the *Custom Columns for Listviews* screen so that you can add more columns, if required. Clicking on the *Close* button on the toolbar will close the *Custom Columns for Listviews* screen and return you to the listview.

The listview will contain the selected columns (see **Figure 11-52**), and these can be repositioned as required. By default the column is always added to the beginning of the listview, and in this case has been dragged to be the last column.

Note that the business object ***COMCOL*** is used to return the custom form column data.

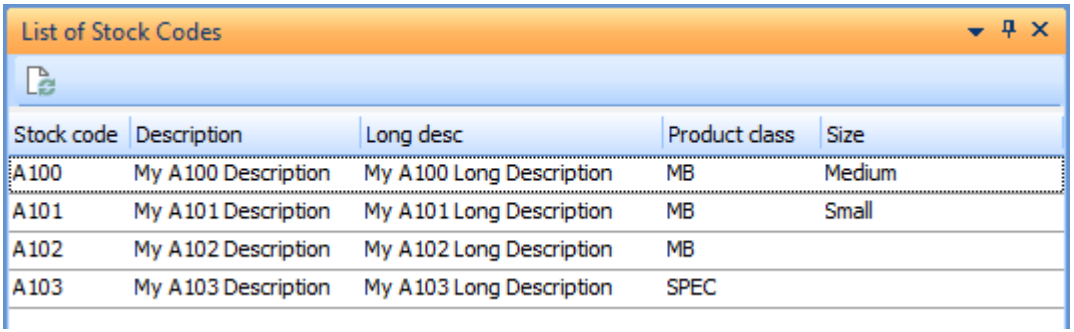


Figure 11-52: The listview containing the custom form field

The column can be removed from the listview by dragging the column header from the column header section. This column name will appear in the *Field Chooser* (right-click on the *Column Header* | *Field Chooser*) and can be dragged back to the column header if this column is required again. **Figure 11-53** shows the *Field Chooser* after this column has been dragged from the listview.

If you right-click on the column heading and select *Remove This Column* from the context-sensitive menu the column is removed completely and will not appear in the *Field Chooser*.

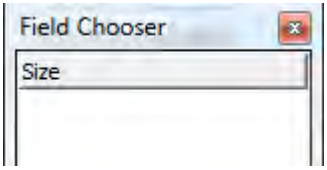


Figure 11-53: The custom form field *Size* in the *Field Chooser* after being dragged from the listview

Adding a Blank Column to a Listview

Note that this facility is only available if none of the columns in the customized pane listview is configured as editable (i.e. none of them have the *Editable='true'* attribute against them). If any of the columns are configured as editable, the option to *Add Custom Columns* in **Figure 11-49** will be greyed-out.

Blank columns can be added to a customized pane listview in a similar way to adding a custom form column, but selecting the *Blank column* radio button that appears in **Figure 11-50**. The blank column is populated using the *CustomizedPane_OnPopulate* function. In the following example a blank column called *My Blank Column* has been added to the customized pane that appears in **Figure 11-48**.

The following code, when added to the *CustomizedPane_OnPopulate* function, will check the contents of the *Product class* column and populate the *My Blank Column* cell with the text *Mountain Bike* if the product class is *MB*, and clear the cell if it is not.

The first line of code creates the variables to be used in this function.

```
Dim  HighestArrayNum, TheArray, Count, PCType
```

The second line of code populates a variable called *TheArray* with the current contents of the array (the whole listview). If you have not already read Chapter 8 on listviews and data grids, it would be worth reading this before continuing (especially the section called *Arrays*).

```
TheArray = CustomizedPane.CodeObject.Array
```

The third line of code populates the *HighestArrayNum* variable with the length of the array (the number of rows in the listview). The -1 is because arrays start at zero, and rows start at 1.

```
HighestArrayNum = UBound(TheArray,2) -1
```

The fourth line of code starts *For*/*Next* loop that uses a count to run from array row 0 to the highest array row number.

```
For Count = 0 To HighestArrayNum
```

The fifth line of code populates the *PCType* variable with the content of the *Product class* column for this row.

```
PCType = TheArray(003,Count)
```

The sixth line of code performs a test to see if the *PCType* variable contains the text *MB*.

```
If PCType = "MB" then
```

The seventh line of code populates the *My Blank Column* cell with the text *Mountain Bikes* if the product class contains *MB*.

```
CustomizedPane_OUT.CodeObject.Array(005,Count) = "<Field Value='Mountain Bikes'>
</Field>"
```

The eighth line of code specifies the code to be run if the product class is not *MB*.

```
Else
```

The ninth line of code specifies what should happen if the product class is not *MB*, which is to clear the *My Blank Column* cell.

```
CustomizedPane_OUT.CodeObject.Array(005,Count) = "<Field Value=''></Field>"
```

The tenth line of code ends the *IF* statement

```
End If
```

The final line of code specifies the end of the *For*/*Next* loop. If there are more lines to process it will increase the *Count* by one and start processing again at the line beginning with *PCType*. If this is the last line it will drop through and continue with the next line of code (in this case that would be the *End Function* statement).

```
Next
```

Below is the complete code to perform this, including comment lines to make it easier to follow. Note that the line that populates the *My Blank Column* cell has wrapped around on this page, and that comment lines start with an apostrophe and do not affect the processing of the VBScript code.

```
Function CustomizedPane_OnPopulate()
' Create the variables to be used
Dim  HighestArrayNum, TheArray, Count, PCType

' Set the variable TheArray to contain the whole first array
TheArray = CustomizedPane.CodeObject.Array

' Work out the number of rows in the array, subtract one and populate the
' HighestArrayNum variable with this value.
HighestArrayNum = UBound(TheArray,2) -1

' Loop through each line
For Count = 0 To HighestArrayNum
  ' PCType to contain contents of Product class column
  PCType = TheArray(003,Count)

  ' Test if Product class = to MB
  If PCType = "MB" then
    ' Populate My Blank Column cell with "Mountain Bikes"
    CustomizedPane_OUT.CodeObject.Array(005,Count) = "<Field Value='Mountain Bikes'
> </Field>"
  Else
    ' Clear the My Blank Column cell
    CustomizedPane_OUT.CodeObject.Array(005,Count) = "<Field Value=''></Field>"
  End If
Next

End Function
```

Adding a Master Field to a Listview

Note that this facility is only available if none of the columns in the customized pane listview is configured as editable (i.e. none have the *Editable='true'* attribute against them). If any of the columns are configured as editable, the option to *Add Custom Columns* in **Figure 11-49** will be greyed-out.

Adding a master field to a customized pane listview is similar to adding a custom form column. The steps to perform this are the same up to **Figure 11-50**. At this point, instead of selecting the *Custom form column* you select *Master table column*.

When this is selected the *Table Fields* screen is displayed. On the toolbar is the *Select table* prompt that has a dropdown list alongside it. This dropdown list will contain the names of all of the master tables that are associated with fields that appear in the listview. In the listview example used for both the custom form column and blank column above, this dropdown list would contain the *Inventory Master* entry.

When the master table is selected the *Table Fields* screen is populated with the field from this table. Fields that are already present in the listview are italicised, and do not have the *Add* hyperlink alongside them. **Figure 11-54** shows the *Table Fields* screen where the *Inventory Master* table has been selected, and the operator has highlighted the *Buyer* field.

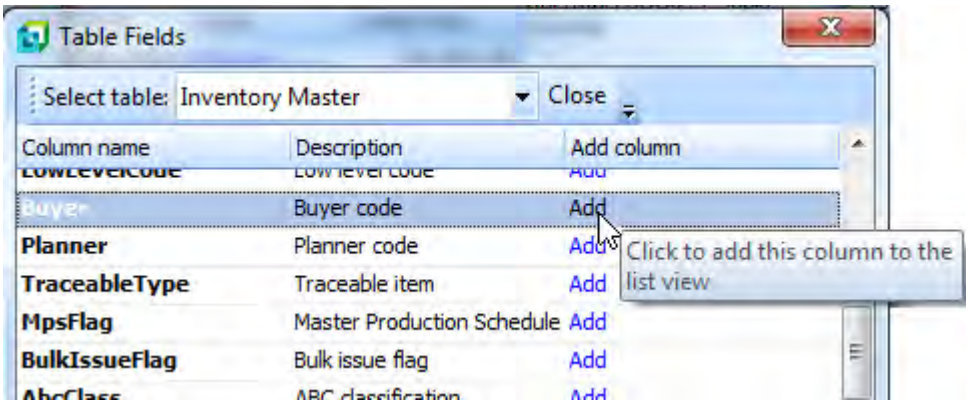
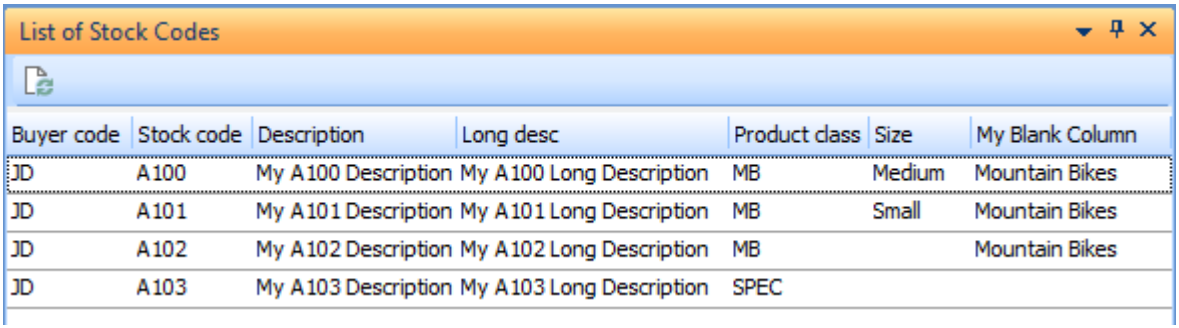


Figure 11-54: The populated *Table Fields* screen with the *Buyer* field highlighted

Clicking on the *Add* hyperlink will add this field as a column in the listview and leave you on the *Table Field* screen so that you can add more than one column. When you have finished adding columns you use the *Close* button on the toolbar to return to the *Custom Columns for Listview* screen. If there are no other columns to add, use the *Close* button on the toolbar to return to the listview.

The columns will be added to the beginning of the listview, but will only be populated the next time that the listview is loaded or refreshed (see **Figure 11-55**).



Buyer code	Stock code	Description	Long desc	Product class	Size	My Blank Column
JD	A100	My A100 Description	My A100 Long Description	MB	Medium	Mountain Bikes
JD	A101	My A101 Description	My A101 Long Description	MB	Small	Mountain Bikes
JD	A102	My A102 Description	My A102 Long Description	MB		Mountain Bikes
JD	A103	My A103 Description	My A103 Long Description	SPEC		

Figure 11-55: The listview after the *Buyer* column was added and the listview refreshed

Adding Master Table Fields that use a Compound Key

Where the column in the customized pane listview contains the key field for a table, SYSPRO will detect this automatically and add the table name to the dropdown list when you select the *Master table column* radio button**. Figure 11-56** shows the dropdown list containing the tables whose master fields can be used. The listview contains the Sales order, Order status, Branch, Salesperson, Customer po number, My balance, and Order date columns.

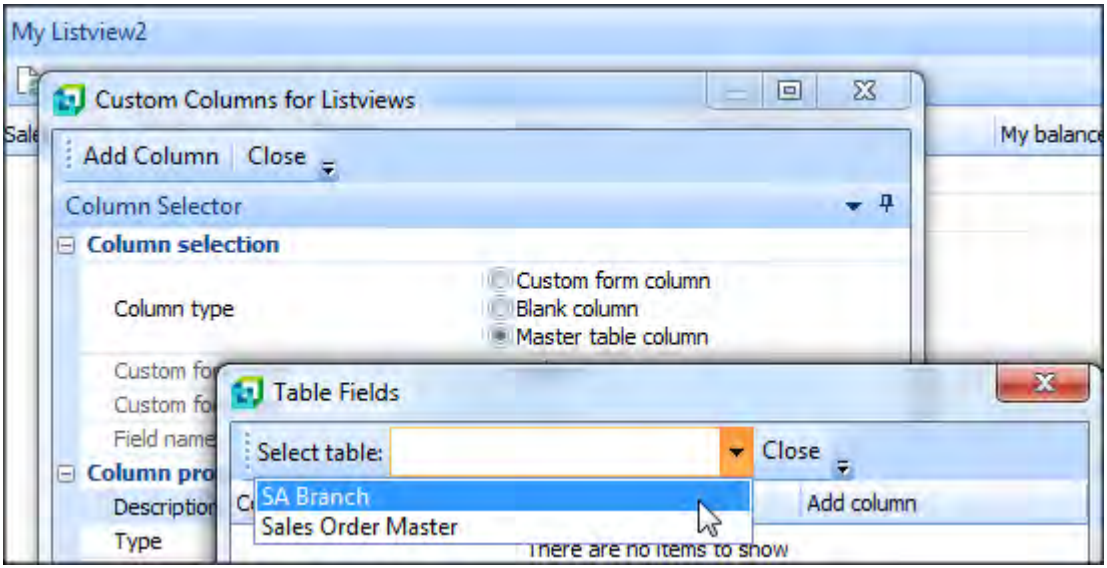


Figure 11:56: The list of tables related to the key fields in the listview

However, the keys of many SYSPRO tables contain more than one key field, called compound keys, and these will not be detected automatically. To be able to access a table with a compound key, all of the components of the key must exist as columns in the listview (they do not need to be displayed in the listview, they just need to be available in the *Field chooser* if they are not being displayed). A file called IMPQVW.IMP (that resides in your SYSPRO Programs folder) contains entries to build the compound keys for some tables, such as the *Multiple ship to address* table.

If you find that the table with the compound key that you want to use is not in the IMPQVW.IMP file, and all the parts of the key are contained in the listview, you can build your own equivalent of IMPQVW.IMP. The file must be called CUSQVW.IMP, and it needs to be located in your custom program folder. The location of your custom program folder can be found/set in your IMPACT.INI file on the SYSPRO application server. It should exist under the *[Customer Directories]* section and will start with *CUSPRG=*. The following is a sample of this entry:

```
[Custom Directories]
CUSPRG=C:\TST700\CUSTOM
```

When you use the dropdown list against the *Select table* prompt (when adding a field for a master table) the list of tables contains the master tables that SYSPRO has detected along with the relevant ones from the CUSQVW.IMP file.

The following is a worked example of creating a compound key entry using a customized pane listview that contains the Sales order, Order status, Branch, Salesperson, Customer po number, My balance, and Order date columns. The requirement is to display the salesperson's name against each row.

The salesperson's name is not available in either of the master tables that are linked to this listview (*SA Branch* or *Sales Order Master*). The *Sales Analysis Salesperson* table contains the salesperson's name. This table does not appear in the list in **Figure 11-56** because it has a compound key. This key consists of the *Branch* code and the *Salesperson* code. The *Salesperson* table does not exist in the *[Compound Keys]* section of the IMPQVW.IMP file, so a custom file needs to be built.

The location of the custom programs folder can be found in the IMPACT.INI file. In this case the folder already existed so did not need to be created. However, there was no CUSQVW.IMP file in this folder so one had to be created. This was done by copying the IMPQVW.IMP file from the SYSPRO *Program* folder to the custom program folder, and renaming it. The file was edited using a text editor (Notepad in this case) to remove the entries above the section starting with *List view Custom Column definitions using compound keys*.

Below is an extract from the IMPQVW.IMP file. The line containing the *BomStructure* entry has been truncated, else it would have wrapped around. The section that would have appeared immediately above this contains information about the layout of the file, so has been omitted here.

```
[Compound Keys]
; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

ArMultAddress      ARSSHP Customer|Address code
BomStructure       BOMMAT Parent stock code|Parent Revision|Parent Release|Route

; End of IMPQVW.IMP
```

In the `COMQVW.IMP` file these two entries must be removed, because they already exist in the `IMPQVW.IMP` file, and the entry for the *Salesperson* table must be added. The content of this file is positional, which means that the *SQL Table name* must start as the first character of the row, the information for the *Custom* column must start as the 21st character of the row, and the first column names must start as the 30th character.

The simplest way of doing this is to remove the *BomStructure* entry, leaving the *ArMultAddress* entry. Then modify the *ArMultAddress* entry to contain the information for the *Salesperson* table. In the sample below the *BomStructure* entry has been removed (and the *End of IMPQVW.IMP* text has been changed *to End of CUSQVW.IMP*):

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

ArMultAddress      ARSSHP Customer|Address code

; End of CUSQVW.IMP
```

The name of the *Salesperson* table is *SalSalesperson*. The table name can be looked up using the *Data Dictionary Viewer* utility (see **Figure 11-57**).

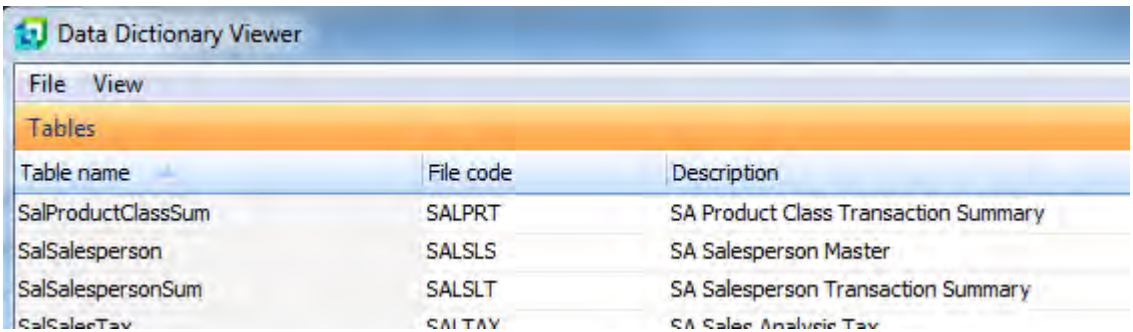


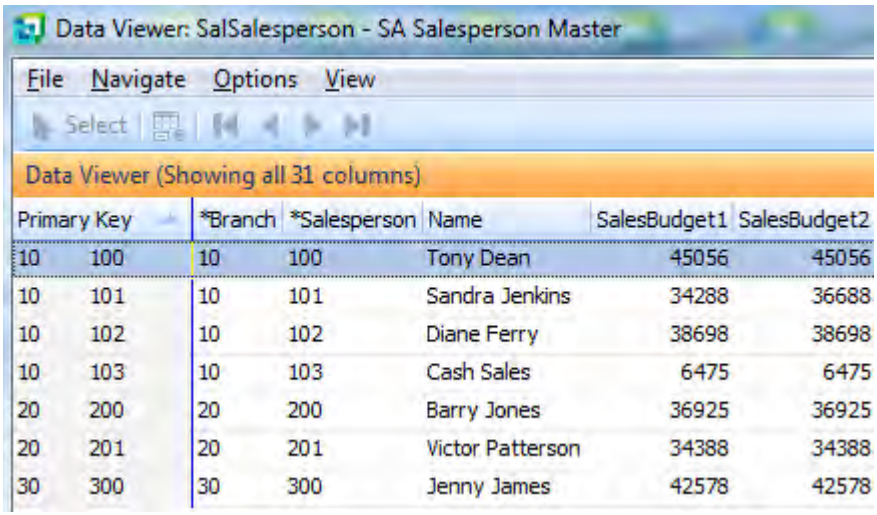
Figure 11-57: The *SalSalesperson* table name in ***DDSBFI***

The *Data Dictionary Viewer* can be loaded using *Ctrl+R* from the main menu, entering the program name ***DDSBFI*** and clicking on the *OK* button.

Now that the *SQL Table name* is known, this can be entered into the `CUSQVW.IMP` file:

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      ARSSHP Customer|Address code

; End of CUSQVW.IMP
```

Double-clicking on the *SalSalesperson* entry in ***DDSBFI*** allows you to see the structure of the table and the content. **Figure 11-58** shows the *SalSalesperson* table. The column to the left of the vertical line is the *Primary key*. The columns to the right show the columns that appear in the table. Those marked with an asterisk are part of the key, in this case *Branch* and *Salesperson*.



Primary Key		*Branch	*Salesperson	Name	SalesBudget1	SalesBudget2
10	100	10	100	Tony Dean	45056	45056
10	101	10	101	Sandra Jenkins	34288	36688
10	102	10	102	Diane Ferry	38698	38698
10	103	10	103	Cash Sales	6475	6475
20	200	20	200	Barry Jones	36925	36925
20	201	20	201	Victor Patterson	34388	34388
30	300	30	300	Jenny James	42578	42578

Figure 11-58: The structure of the *SalSalesperson* table

This information can be added to your `CUSQVW.IMP` file (note that the column names are separated by pipe signs):

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      ARSSHP Branch|Salesperson

; End of CUSQVW.IMP
```

The last piece of information that is required is the name of the custom form for this table. This can be found in the `IMPCFM.IMP` file in the SYSPRO *Program* folder. The following is an extract from this file showing the relevant section. The *Salesperson* entry is the middle one, so the custom form entry required is *SLS*.

```
SORRD  Sales Order Route Detail              SorRouteDetail     -
SLS    Salesperson                           SalSalesperson     -
SERIAL Serial                                InvSerialHead      -
```

This information can be added to the `CUSQVW.IMP` file, as below:

```
[Compound Keys]

; ------------------------------------------------------------------------------
;SQL Table name     Custom Column names, separated by |
;------------------ ------ ----------------- ------ ------ ------

SalSalesperson      SLS    Branch|Salesperson

; End of CUSQVW.IMP
```

The `CUSQVW.IMP` file is now complete, and you need to check that you have built this correctly. First, login to SYSPRO from scratch so that the new `CUSQVW.IMP` file is read in by SYSPRO. Call up the program containing the customized pane listview, right-click on one of the column headers in the listview, select *Customize*, and then *Add Custom Columns*. Select the *Master table column* radio button and the *Table Fields* screen will be displayed. The dropdown list alongside the *Select table* prompt should contain the *SA Salesperson Master* table name (see **Figure 11-59**).

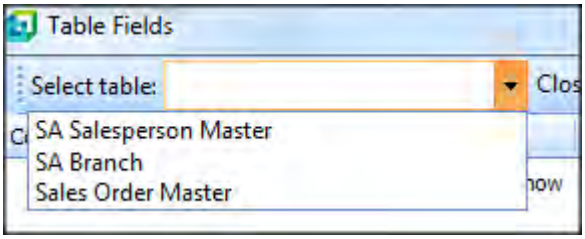


Figure 11-59: The *SA Salesperson Master* table name available in the dropdown list

Selecting this table name displays the fields from this table in the listview (see **Figure 11-60**). The fields from this table that are already in the listview will appear italicized.

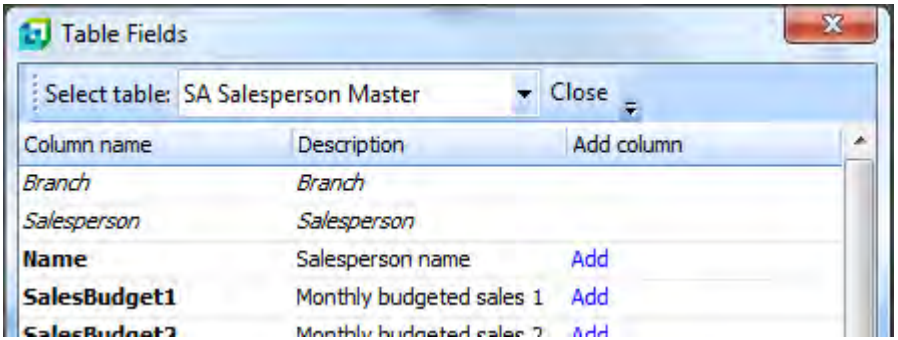


Figure 11-60: The fields from the *Salesperson* table

Select the *Add* hyperlink against the *Name* entry and the *Salesperson name* column is added to the listview. This is always added as the first column of the listview. This column header can be dragged to the required location, and the column will be populated with data the next time the listview is used.

Clearing the Content of a Listview

The content of a customized pane listview can be cleared during an *OnRefresh* event using the following line of code:

```
CustomizedPane.CodeObject.ListviewData = " "
```

For example, if your customized pane listview pulls information from an external database depending on which stock code is selected, you would probably want to clear the listview if there are no records in the database for the current stock code. If the listview is not cleared the operator may assume that the values remaining in the listview relate to the selected stock code.

Adding Rows to an Already Populated Listview

It is possible to add rows to the end of an already populated listview using VBScript. This is done by first setting the *ListviewClearData* variable to *false*, creating the XML in the normal format to populate the listview, and passing this to the *ListviewData* variable

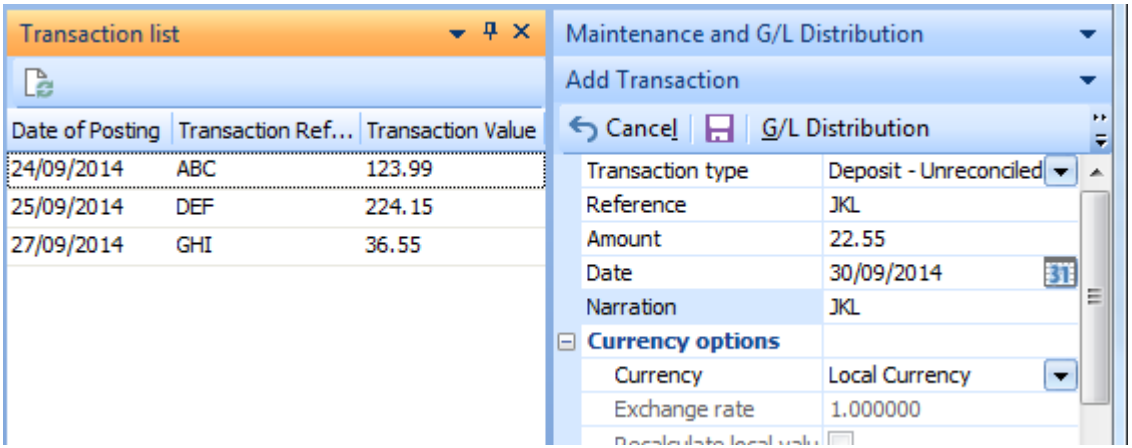


Figure 11-61: The listview before a new row is added to it

The following example is from the *Cash Book Deposits and Withdrawals* program. In **Figure 11-61** the *Transaction list* listview is a customized pane that keeps track of all the transactions that have been posted since the operator loaded the program. Each time that the operator posts a transaction using the *Add Transaction* screen the *Date*, *Reference* and *Amount* are passed from the *Add Transaction* pane to the *Transaction list* pane, and added as a new row at the end of the listview.

Figure 11-62 shows the same listview after the operator has saved the transaction in the *Add Transaction* pane, the values were passed through to the *Transaction list* pane, and the row added.

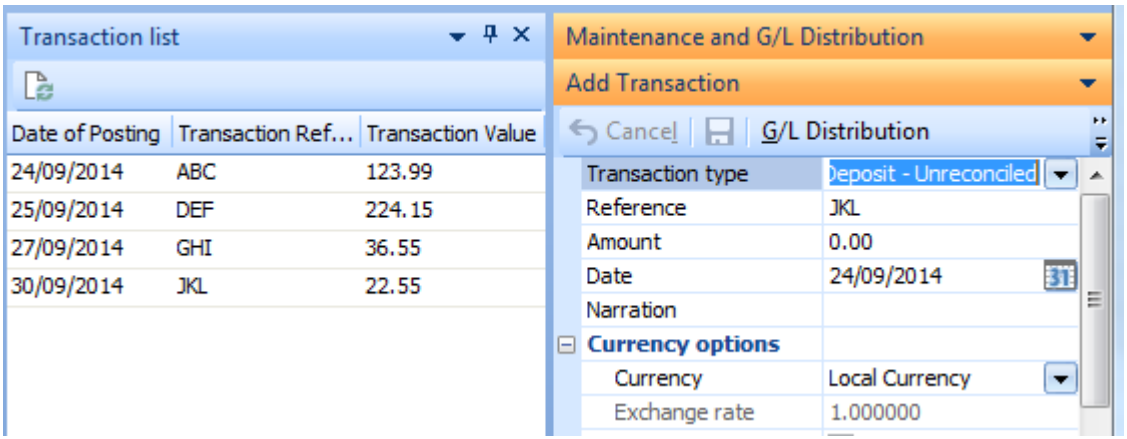


Figure 11-62: The listview after the row has been added

The following three lines of code appear within the *MaintainTransactions_OnSubmit* function of the *Add Transaction* pane. Each line has been separated by a blank line to make it easier to follow as the second line wraps around twice on the page.

```
Dim BuildValues

BuildValues = "FromAdd:" & MaintainTransactions.CodeObject.Date & ":" &
MaintainTransactions.CodeObject.Reference & ":" &
MaintainTransactions.CodeObject.Amount

CustomizedPanes.CodeObject.TransactionList = BuildValues
```

The first line of code makes the *BuildValues* variable available.

The second line of code builds up the string to be passed to the *Transaction list* customized pane. This consists of the text *FromAdd*, a comma, the date, a comma, the transaction reference, a comma, and the value. Each of the four pieces of information is separated by a comma so that when it reaches the customized pane the string can be stripped down to its constituent parts.

The *FromAdd* text is just a means of knowing which pane causes the customized pane to be refreshed, because different actions may need to happen depending on why the pane was refreshed.

The third line of code passes the contents of the *BuildValues* variable (the string) to the variable that will cause the *Transaction list* pane to be refreshed.

The following is the code that is added to the *CustomizedPane_OnRefresh* function to receive the string and add the new row:

```
Dim RefValue, BuildData
RefValue = Split(CustomizedPane.CodeObject.RefreshValue, ":")

If RefValue(0) = "FromAdd" then
  CustomizedPane.CodeObject.ListviewClearData = "false"
  BuildData = "<MyRoot>"
  BuildData = BuildData & "<MyRow>"
  BuildData = BuildData & "<Date>" & RefValue(1) & "</Date>"
  BuildData = BuildData & "<Reference>" & RefValue(2) & "</Reference>"
  BuildData = BuildData & "<TrnVal>" & RefValue(3) & "</TrnVal>"
  BuildData = BuildData & "</MyRow>"
  BuildData = BuildData & "</MyRoot>"
  CustomizedPane.CodeObject.ListviewData = BuildData
End If
```

The first line of code creates two variables that will be used within the script.

If the customized pane was refreshed by another pane, the *RefreshValue* variable will contain the string that is passed through from the other pane. The second line of code takes the content of the *RefreshValue* variable and splits it into its components using a colon as its delimiter. The *RefValue* variable is populated with the four pieces of information and becomes an array. *RefValue(0)* contains the *FromAdd* text, *RefValue(1)* contains the date, *RefValue(2)* contains the reference, and *RefValue(3)* contains the value.

The third line of code tests to see if the first item in the array is equal to *FromAdd*. If it is, the next nine lines of code are processed, and if not, they are skipped.

The fourth line of code sets the *ListviewClearData* variable to *false* so that the existing rows in the listview are not cleared out.

Lines five to eleven build up the XML to populate the listview. This must use the same element names as would be used to populate the listview from scratch. The content of array entries one, two, and three are used to populate the XML.

The twelfth line passes the content of the *BuildData* variable (the XML) to the *ListviewData* variable so that the listview can be populated.
The thirteenth line of code closes off the *IF* statement.

Browsing on Editable Key Fields

If a key field has been defined as editable (it has the *Editable='true'* attribute configured against it), when the field is selected a browse icon will automatically appear against the cell.

Using the same listview as in **Figure 11-55**, if the *Stock code* column was changed to have the *Editable='true'* attribute and the operator clicked on the stock code, a browse icon would appear against this cell (see **Figure 11-63**).

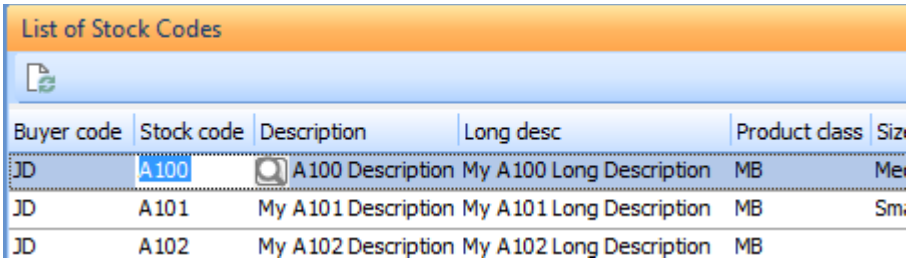


Figure 11-63: The browse appearing against a key field when the cell is selected

This browse works in the same way as elsewhere within SYSPRO in that if the operator selects a stock code from the browse, the stock code is returned and populates the cell.

Smart Links within the Listview

If a column contains a key field (such as stock code, customer, sales order, etc.) when the operator moves the mouse pointer over a cell containing a key field an icon appears against the cell to show that queries are available. This is known as a *Smart Link*. The icon can be seen against the *Stock code* column in the first row of **Figure 11-64**.

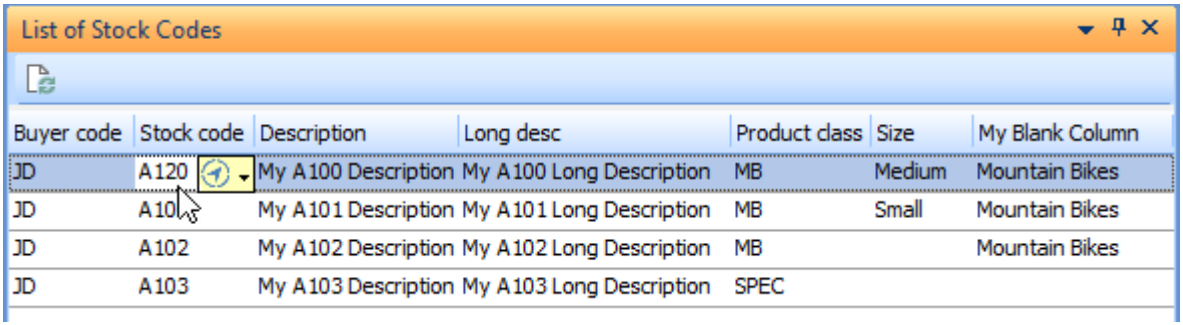


Figure 11-64: The *Smart Link* icon that shows that queries are available

If the operator clicks on the *Smart Link* icon, the list of queries is displayed in a menu, along with the option to switch off *Smart Links*. The list of queries will change depending on which key field was selected. **Figure 11-65** shows the queries when the *Stock code* field is selected. If the *Product class* field had been selected the list would contain the product class *Query* option, the option *Switch Smart Link Off* and the option *Customize Smart Links*.

If the operator switches off the *Smart Links*, they can be switched back on using the *Smart Link* option from the context-sensitive menu that is displayed when you right-click on the column heading (or by using the shortcut keystrokes *Shift+Alt+F7*).

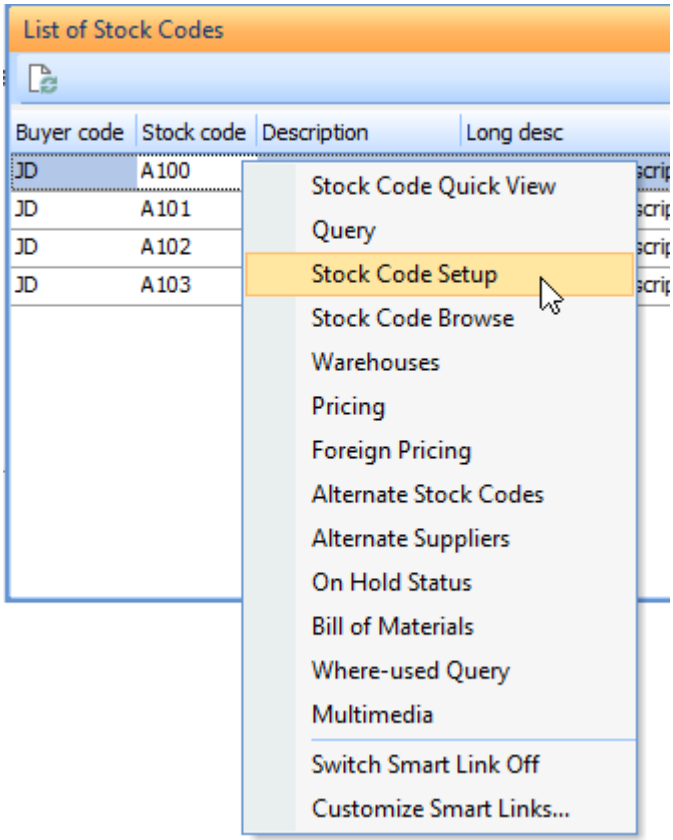


Figure 11-65: The list of available queries for the *Stock code* field using *Smart Links*

Macro Events

There are ten *Macro Events* available for a listview customized pane. These are:

- OnLoad
- OnRefresh
- OnToolbarButton1Clicked
- OnToolbarButton2Clicked
- OnPopulate
- OnRowSelected
- OnDblClick
- OnLinkClicked
- OnAfterChange
- OnChecked
- OnDELPressed

OnLoad
The *OnLoad* macro event is used to define the listview's structure using the *ListviewProperties* variable. This is covered in detail within the *Defining the Listview Structure* section above.

OnRefresh
The *OnRefresh* macro event is used to populate/re-populate the listview with information using the *ListviewData* variable. Note that the array containing all the listview's cells is not available during this function.

OnToolbarButton1Clicked and OnToolbarButton2Clicked
If the *Control type* against one or both of the *Toolbar controls* is set to *Button* within the *Pane Properties* pane of the *Customized Pane Editor* screen, when the operator clicks on the button the relevant macro event is fired (*OnToolbarButton1Clicked* or *OnToolbarButton2Clicked*).

An array containing all the listview's cells is available within the functions associated with these two events, so values can be read/extracted from them. This is accessed using the *CustomizedPane.CodeObject.Array* variable name. The matching *_OUT* variable (*CustomizedPane_OUT.CodeObject.Array*) is also available during this function, so the values and attributes associated with the cells can be set/changed.

OnPopulate
The *OnPopulat*e macro event is fired each time that the listview is populated with information, such as when the *OnRefresh* event fires. However, the *OnPopulate* event does not fire if there are no rows to be displayed in the listview.

An array containing all the listview's cells is available within this, so values can be read/extracted from them. This is accessed using the *CustomizedPane.CodeObject.Array* variable name. The matching *_OUT* variable (*CustomizedPane_OUT.CodeObject.Array*) is also available during this function, so the values and attributes associated with the cells can be set/changed. The *OnPopulate* function is covered in detail within the *Populating Cells with Values within the OnPopulate Function* section above.

OnRowSelected
The *OnRowSelected* macro event is fired when the operator selects a row within the listview. Within this function the contents of the selected row is available in an array using the *ListviewRowReturned* variable. The row number of the row that the operator selected is available using the *SelectedRow* variable. Note that this is the array's row number, not the listview's row number, so if the operator clicked on row 2 of the listview the *SelectedRow* variable will contain 1.

OnDblClick
The *OnDblClick* macro event fires when the operator double-clicks on a listview cell. Within the *OnDblClick* function:

- An array containing all the listview's cells is available, so values can be read/extracted. This is accessed using the *CustomizedPane.CodeObject.Array* variable name. The matching *_OUT* variable (*CustomizedPane_OUT.CodeObject.Array*) is also available during this function, so the values and attributes associated with the cells can be set/changed

- The content of the double-clicked row is available in an array using the *ListviewRowReturned* variable.

- The (array) row number is available using the *SelectedRow* variable.

OnChecked
The *OnChecked* macro event is fired when a checkbox within the listview customized pane is either checked or unchecked. Within the *OnChecked* function:

- An array containing all the listview's cells is available, so values can be read/extracted. This is accessed using the *CustomizedPane.CodeObject.Array* variable name. The matching *CustomizedPane_OUT.CodeObject.Array* variable is also available during this function, so the values and attributes associated with the cells can be set/changed

- The row containing this checkbox is available in an array using the *ListviewRowReturned* variable.

- The (array) row number is available using the *SelectedRow* variable.

- The (array) column number containing the checkbox is available using the *ColumnClicked* variable.

- The *BeforeChangeValue* system variable will contain the value of the checkbox before the operator checked/unchecked it.

In this example, the checkbox is in the eighth column of the listview, which is the array's column 7. The operator checks the checkbox in row two of the listview (array row 1). The *ColumnClicked* variable will contain the value 7 and the *SelectedRow* variable will contain the value 000001.

The value of the checkbox (either 0 for unchecked, or 1 for checked) can be read back in one of two ways. The first is using the contents of the *Tab* delimited *ListviewRowReturned* variable. This requires that the contents be separated into the individual columns, and array column 7 read back. The code to display the checkbox's status after it is changed appears below:

```
Dim MyArray
MyArray = Split(CustomizedPane.CodeObject.ListviewRowReturned,VBtab)
msgbox MyArray(CustomizedPane.CodeObject.ColumnClicked)
```

The second way is using the *CustomizedPane.CodeObject.Array* variable that represents the whole listview. As the array column and row numbers are already known this can be done in one line of code. This line of code appears below, but has wrapped around on the page:

```
msgbox CustomizedPane.CodeObject.Array(CustomizedPane.CodeObject.ColumnClicked,
CustomizedPane.CodeObject.SelectedRow)
```

OnLinkClicked

The *OnLinkClicked* macro event is fired when the operator clicks on a value within a column that is defined as containing hyperlinks (using the *Link='true'* attribute). Within the *OnLinkClicked* function:

- The row containing this hyperlink is available in an array using the *ListviewRowReturned* variable.
- The (array) row number is available using the *SelectedRow* variable.
- The (array) column number containing this hyperlink is available using the *ColumnClicked* variable.

The value of the cell containing the hyperlink can be determined using the contents of the *Tab* delimited *ListviewRowReturned* variable. This requires that the contents be separated into the individual columns, and required column number read back. The code to display the cell's value appears below:

```
Dim MyArray
MyArray = Split(CustomizedPane.CodeObject.ListviewRowReturned,VBtab)
msgbox MyArray(CustomizedPane.CodeObject.ColumnClicked)
```

OnAfterChange

If a listview column is defined as being editable, and the operator changes its value, the *OnAfterChange* macro event will fire. Within the *OnAfterChange* function:

- An array containing all the listview's cells is available, so values can be read/extracted. This is accessed using the *CustomizedPane.CodeObject.Array* variable name. The matching *CustomizedPane_OUT.CodeObject.Array* is also available during this function, so the values and attributes associated with the cells can be set/changed
- The row containing the cell that was changed is available in an array using the *ListviewRowReturned* variable.

- The (array) row number is available using the *SelectedRow* variable.

- The (array) column number containing the changed value is available using the *ColumnClicked* variable.

- The *BeforeChangeValue* system variable will contain the value of the cell before the operator changed it, so a comparison can be made, except for columns defined as Address or Checkbox.

OnDELPressed
If the *Columns* element contains the *AllowDEL* attribute set to *True*, operators are allowed to delete rows from the listview by highlighting the row and pressing the *DEL* key. If the *OnDELPressed* function exists the event will be fired at this time. The following three customized pane variables can be used at this time:

- *ListviewRowReturned* – returns the content of the row selected to be deleted in a one-dimensional array.

- *SelectedRow* – returns the array row number (the first row number is zero) of the row selected to be deleted.

- The Array name – the content of the whole array (before the *DEL* key was pressed) is available. You can access any cell within this array by specifying the row and column array numbers.

Within the *OnDELPressed* function you can prevent the row from being deleted by setting the *OnDELPressed* function to *False*.

The following example is from a customized pane listview containing sales orders. The second column of the listview contains the sales order status. If the operator attempts to delete a row, the content of this row is read into an array (after splitting it into its constituent parts) and a check is made to see if the second entry in the array contains the status code of 8. If this contains an 8 the line is not deleted. If it contains any other value the row is deleted from the listview.:

```
Function CustomizedPane_OnDELPressed()
  Dim MyArray, MyStatus

  MyArray = Split(CustomizedPane.CodeObject.ListviewRowReturned,VBTab)
  MyStatus = MyArray(1)
  If MyStatus = "8" then
    Msgbox "You cannot delete a Sales Order in status <8>"
    CustomizedPane_OnDELPressed = False
    Exit Function
  End If
End Function
```

Chapter 12 - Graphs

There are ten different types of graph that can be created using VBScript. These are Area, Bar, Bubble, Funnel, Line, Pie, Pyramid, Spline, Step, and Scatter. Each of the different graph types can be used to represent your data, but certain types can be used to represent the information in the best way. The *Gallery* option against the finished customized pane (*Options* button | *Gallery*) enables the style of the graph to be changed, and this includes the Torus type graph.

Although most customized panes with an *Object Type* of *Graph* are used to display a single graph, it is possible to have up to eight graphs displayed within one customized pane. An example of a graph customized pane containing three graphs appears within the templates section that is available when adding a graph customized pane. Its name is *Sample Multiple Diagrams*. Many of the graph properties within the VBScripting only take effect when you initially create the graph, and are saved in the operator preferences file when the customized pane is created. If these are subsequently changed they do not change the way that the graph is displayed, as the operator's preference is used.

Graphs work in a similar way to listviews in that you define the structure in XML, and then populate it using XML. The definition would typically be against the *OnLoad* function, and the population against the *OnRefresh* function. The *OnRefresh* event will fire immediately after the *Onload* event.

If the graph depends on a field from another form within the same program to be populated first, the refreshing will have to be done later. However, if you want the graph to populate immediately (because it isn't dependent on information from a form within the program in which it resides) it is advisable to create your own function to contain the code to perform the population. This should contain just the code to populate the graph. This function would then be called at the end of the *OnLoad* function, and could also be called within the *OnRefresh* function.

When writing your code this way, you only have the code in one place. There are several benefits to this. If the code needs to be changed for any reason, you only have to change it in this function. You won't get different values displayed between the initial display (using the *OnLoad* function) instead of when the graph is refreshed (using the *OnRefresh* function) because you mistakenly only changed the code in one of these places.

A Basic Example

The simplest way of explaining the graphs is with an example. This example creates a graph to show all the banks that a company deals with, and the current balance of each in the local currency. First create a customized pane and set the *Object type* to *Graph*. Then use the *Edit VBScript* button on the toolbar to start adding your code. Double-click on the *OnLoad* event to create the *CustomizedPane_OnLoad* function, and place the cursor within it.

Within the *CustomizedPane* section of the *Variables* pane is the *GraphProperties* option. When this is clicked the *Graph Properties* screen is displayed. **Figure 12-1** shows the *Graph Properties* screen where the graph's title, description against the X-axis, and description against the Y-axis have already been supplied. The type of graph is in the process of being selected. Once this information has been supplied, clicking on the *Insert VBScript Code* button inserts the graph's structure.

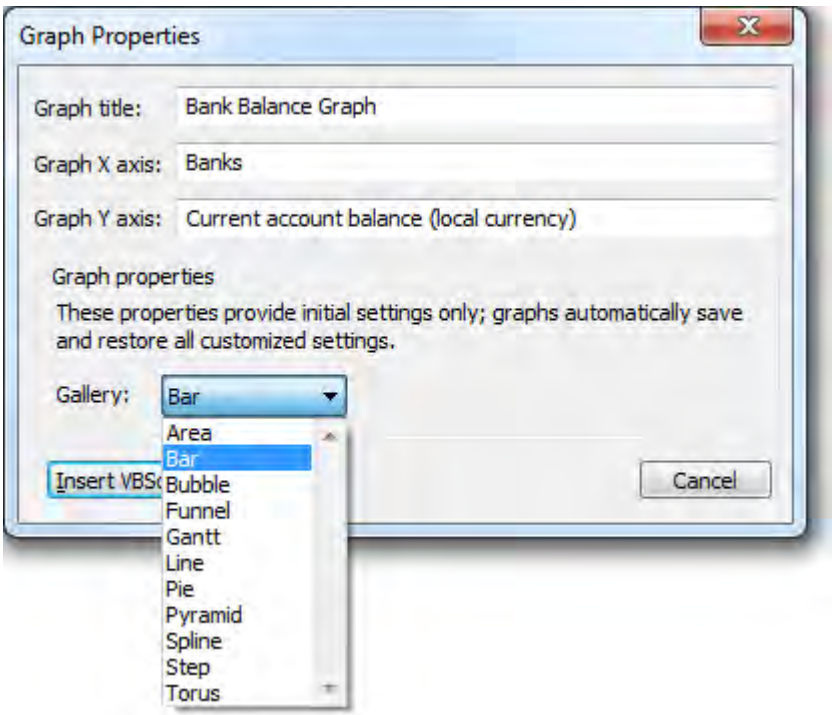


Figure 12-1: The *Graph Properties* screen where you design your graph structure

When the *Insert VBScript Code* button is clicked the VBScript code defining the graph's structure is created for you. The XML below contains the structure of the graph created using the *Graph*

Properties screen in **Figure 12-1**, but with the VBScript code that envelopes it removed to make it easier to follow.

The *Titles* section contains the overall title of the graph, and this appears at the top of the graph. The *Panels* section contains the descriptions that appear underneath the graph (on the X-axis) and to the left of the graph (on the Y-axis). The *Series* element defines that this is a bar chart.

```
<CodejockChart>
  <Titles>
    <Title Visible='1' Text='Bank Balance Graph'></Title>
  </Titles>
  <Panels>
    <Panel>
      <AxisX><Title Text='Banks' /></AxisX>
      <AxisY><Title Text='Current account balance (local currency)' /></AxisY>
    </Panel>
  </Panels>
  <Series LegendText='Current Balance' Style='Bar' />
</CodejockChart>
```

Exit the *OnLoad* function, back to the *VBScript Editor* screen. Double-click on the *OnRefresh* event to create the *CustomizedPane_OnRefresh* function.

A graph is populated by supplying XML to the *GraphData* variable. The *GraphData* variable can be found within the *CustomizedPane* section on the *Variables* pane. Double-clicking on this variable name loads the *Graph Data* screen, and this is pre-populated with a sample of the XML (see **Figure 12-2**).

Clicking on the *Insert VBScript Code* button wraps this XML in VBScript, and inserts this into your code.

The code also contains a line where this XML is passed to the *GraphData* variable so that it will populate the graph. Saving the VBScript, exiting the editor, and clicking on the *Refresh* button on the graph's toolbar, causes the graph to be populated with these values (see **Figure 12-3**).

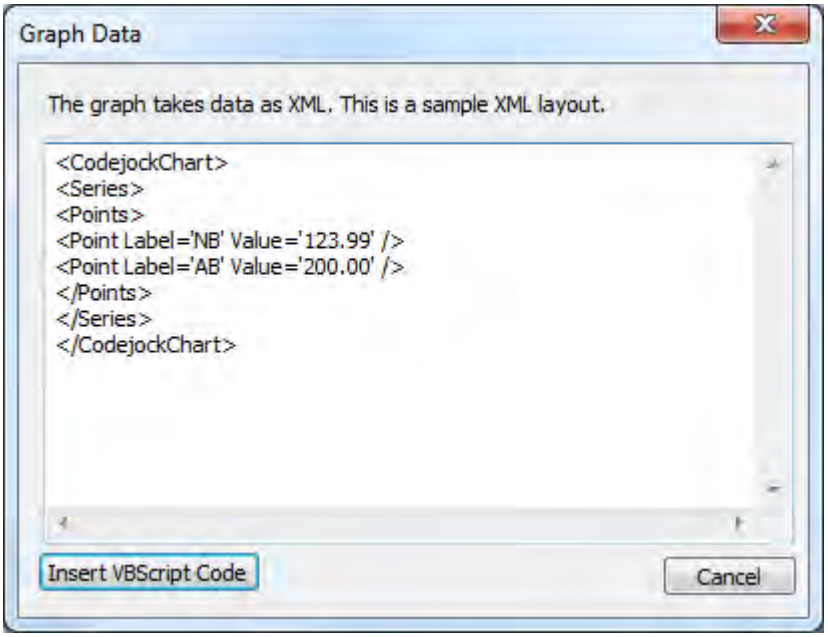


Figure 12-2: The *Graph Data* screen that comes pre-populated with an XML sample

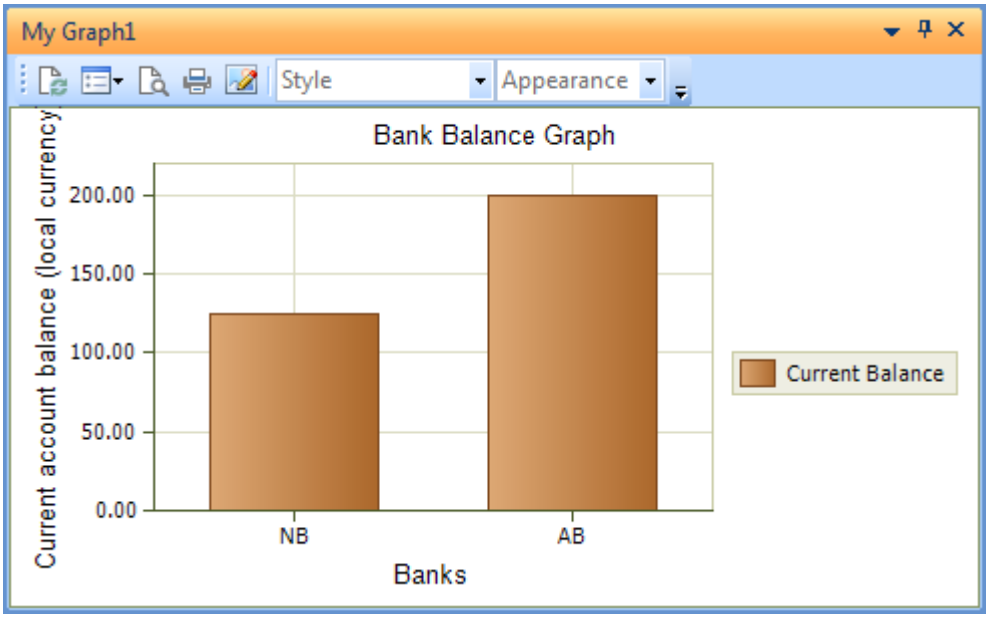


Figure 12-3: Populating the graph with the hardcoded sample values

Although this shows you how the values populate the graph, hard-coding these values is not particularly useful. You need to retrieve the real values, massage them into the correct format, and use these to populate the graph. In most cases the values will be supplied by an e.net Solutions business object, so that is the example that will be used here.

Edit the customized pane's *OnRefresh* function, and clear out the code that was added by the *Graph Data* screen. You should end up this just the following two lines of code for this function.

```
Function CustomizedPane_OnRefresh()

End Function
```

Place the cursor on the blank line within this function. Click on the *Call Business Object* button on the editor's toolbar to start the *Call a Query Business Object* wizard. Supply the business object name of ***COMQEX***, and trim the XML down to the minimum that will just return back the details for the bank (see **Figure 12-4**). Once the XML is correct, select the *Insert VBScript* button to add the code to call the business object to your function.

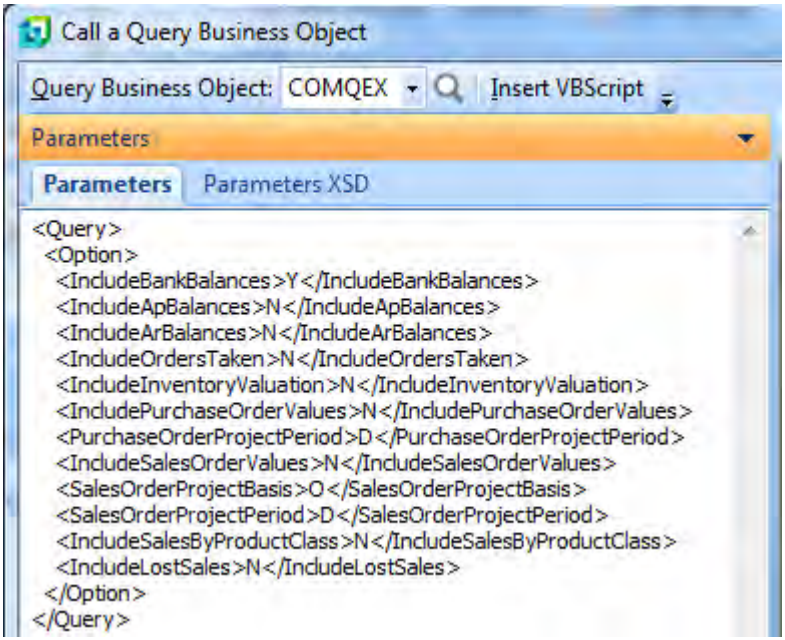


Figure 12-4: Populating the *Call a Query Business Object* wizard

Immediately after the code to call the business object, add the following code:

```
'Load the XmlOut into the DOM
Dim xmlDOM, objNodeList, DataXML, Count
Set xmlDOM = createobject("Msxml2.FreeThreadedDOMDocument.3.0")
xmlDOM.async = false
xmlDOM.LoadXML(XMLOut)
```

This will load the output from the business object (held in the *XMLOut* variable) into the *Document Object Model* (DOM) so that it becomes easier to interrogate. Remember that any line starting with an apostrophe is a comment line, and is only present to make it easier to follow the code.

The following is a simplification of the structure of the XML output by the ***COMQEX*** business object:

```
<ExecutiveViewQuery>
    <BankBalance>
        <BankItem>
            All the details for the first bank
        </BankItem>
        <BankItem>
           All the details for the second bank
        </BankItem>
    etc.
```

The next line of code is used to extract all the *BankItem* nodes and add them to a list. It uses an absolute path of *//ExecutiveQuery/BankBalances/BankItem*. This line of code appears below, and has wrapped around on this page as the line is so long.

```
Set objNodeList =
xmlDOM.DocumentElement.SelectNodes("//ExecutiveViewQuery/BankBalance/BankItem")
```

The next line of code starts to build up the XML in the same format as appears in **Figure 12-2**.

```
DataXML = DataXML & "<CodejockChart><Series><Points>"
```

This is followed by a line that starts a *For*/*Next* loop to iterate through all the nodes that were added to the *objNodeList* variable above (i.e. all of the *BankItem* nodes). Just as with the arrays, the first node in the list starts at zero, so the value of one needs to be subtracted from the node list length.

```
For Count = 0 To objNodeList.Length - 1
```

The next line is a mammoth one, and has wrapped around twice in the example below. It gets run once for each bank (as each bank has its own node). To populate the graph, each bank must have its own *<Point>* element, and this bit of code is used to create it.

```
DataXML = DataXML & "<Point Label='" +
objNodeList(Count).SelectSingleNode("Bank").childnodes(0).xml + "' Value='" +
objNodeList(Count).SelectSingleNode("CbCurBalLoc1").Text + "' />"
```

This is followed immediately by a *Next* statement.

```
Next
```

Below is a (slightly) sanitized example of the output from the ***COMQEX*** business object, with a few elements removed to make it simpler to follow. Also, only the first two banks are shown (there were three in the output from the business object).

For the first time through the *For*/*Next* loop the *Count* variable will contain 0. The first part of the line of code extracts the contents of the *Bank* element, which contains *FB* and adds it to the *Label=* attribute of the *Point* element.

The next section extracts the contents of the *CbCurBalLoc1* element, which contains the value *63329428.49*, and adds it to the *Value=* attribute.

```
<?xml version="1.0" encoding="Windows-1252"?>
<ExecutiveViewQuery Language='05' Language2='EN' Version='7.0.011' >
  <BankBalance>
    <BankItem>
      <Bank>FB</Bank>
      <Description>First United Bank</Description>
      <CbStmtBal1>64310025.92</CbStmtBal1>
      <CbStmtBalLoc1>64310025.92</CbStmtBalLoc1>
      <CbCurBal1>63329428.49</CbCurBal1>
      <CbCurBalLoc1>63329428.49</CbCurBalLoc1>
      <Currency>$</Currency>
      <BuyExchangeRate>1.000000</BuyExchangeRate>
    </BankItem>
    <BankItem>
      <Bank>GB</Bank>
      <Description>Global Bank</Description>
      <CbStmtBal1>20000.00</CbStmtBal1>
      <CbStmtBalLoc1>20000.00</CbStmtBalLoc1>
      <CbCurBal1>19960.00</CbCurBal1>
      <CbCurBalLoc1>19960.00</CbCurBalLoc1>
      <Currency>$</Currency>
      <BuyExchangeRate>1.000000</BuyExchangeRate>
    </BankItem>
```

When the first node (bank) in the *For*/*Next* loop is complete the *DataXML* variable will contain the static part of the XML, plus the values for the first line. This has been displayed indented, to make it easier to follow.

```
<CodejockChart>
  <Series>
    <Points>
      <Point Label='FB' Value='63329428.49' />
```

When the processing hits the *Next* statement of the *For*/*Next* loop, the *Count* variable is automatically increased by 1 and the line is processed again. The *Bank* element contains *GB*, and the *CbCurBalLoc1* element contains *19960.00*, so the following *Point* element is added to the existing XML.

```
<Point Label='GB' Value='19960.00' />
```

As mentioned above, the unabridged XML contained three banks, so when the processing hits the *Next* statement the count is increased by one and processing starts again. The *Bank* element contains *NB* and the *CbCurBalLoc1* element contains *701127.15*, so the following *Point* element is added to the existing XML.

```
<Point Label='NB' Value='701127.15' />
```

When processing hits the *Next* statement this time, there are no more nodes in the list, so processing drops through.

The next line of code completes the XML that will be used to populate the graph. It just closes off the open *Points*, *Series*, and *CodejockChart* elements and appends them to the *DataXML* variable.

```
DataXML = DataXML & "</Points></Series></CodejockChart>"
```

When this has completed, the *DataXML* variable in memory will contain the following XML (you should not add this to your code; it is just to explain what will be produced when the code is run).

```
<CodejockChart>
  <Series>
    <Points>
      <Point Label='FB' Value='63329428.49' />
      <Point Label='GB' Value='19960.00' />
      <Point Label='NB' Value='701127.15' />
    </Points>
  </Series>
</CodejockChart>
```

The contents of this *DataXML* variable must be passed to the *GraphData* variable, as this is what will populate the graph. The full name of the *GraphData* element is added to the VBScript code by double-clicking on its name within the *CustomizedPane* section of the *Variables* pane. Add to this an equal sign and the *DataXML* variable name, and you will have the line of code below.

```
CustomizedPane.CodeObject.GraphData = DataXML
```

The last line of code to be added is just good housekeeping, because it frees up the resources that you used.

```
set xmlDOM = nothing
```

Figure 12-5 shows the completed graph. Once the graph has been accessed by the operator their preferences are saved, which includes the type of graph. Changing the type of graph (from *Bar* to *Line*, for example) within the code after this will not change the graph.

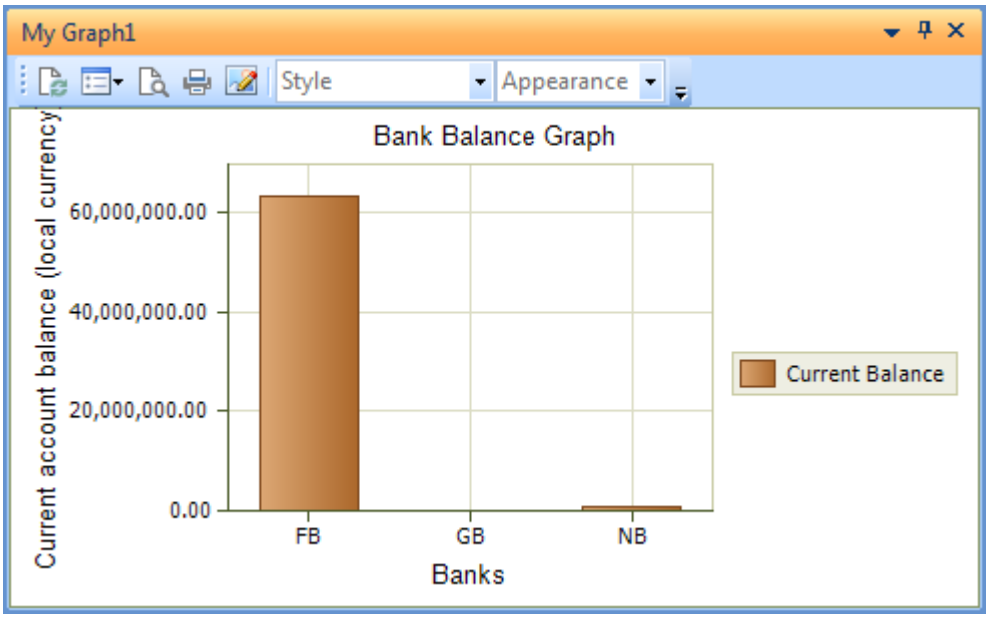


Figure 12-5: The completed graph

The operator can change the type of graph as required by using the *Options* button on the toolbar, and selecting the type of graph under the *Gallery* section.

Alternatively, if you want to change the graph type you can make the change in your code, exit the customized pane, then exit the program containing the customized pane. Call up the program again, and against the *Options* button on the customized pane's toolbar is the *Reset Chart Settings* option. Clicking on this will reset the graph to have the new settings.

Defining the Structure for Multiple Graphs within a Single Customized Pane

As mentioned at the beginning of this chapter, it is possible for the customized pane to contain up to eight graphs. An example can be seen in **Figure 12-6**. Within the XML that defines the structure the whole customized pane is known as a *Chart*, and the graphs within it are known as *Panels*. The XML below shows the rough outline of the XML required to define the structure of a customized pane that will contain three graphs.

```
<CodejockChart>
  <Labels Angled='1'/>
  <Legend Visible='1' ColumnCount='1' />
  <PanelDirection Direction='1' Rotated='0' />
  <Titles>
    <Title Visible='1' Text='My Title'></Title>
  </Titles>
  <Panels>
    <Panel>
       Information for the first panel
    </Panel>
    <Panel>
       Information for the second panel
    </Panel>
    <Panel>
       Information for the third panel
    </Panel>
  </Panels>
  <Series LegendText='Daily' Style='Bar' Panel='1' />
  <Series LegendText='Summary' Style='Line' Panel='2' />
  <Series LegendText='Summary Hours' Style='Pie' Panel='3' />
</CodejockChart>
```

The *Labels*, *Legend*, and *PanelDirection* elements relate to the whole customized pane, so appear directly under the *CodejockChart* root element. The *Title* is also for the whole *Chart* so appears within the *Titles* node under the *CodejockChart* root element.

The *Panels* node can contain one or more *Panel* nodes. Each *Panel* node is used to define information about the X and Y axes, such as the text to appear alongside it, where the values should contain decimals, and the alignment of the text. Although not defined within the node, the first *Panel* node is known as *Panel 1*, the second as *Panel 2*, etc.

At the same level as the *Panels* node are multiple *Series* elements, one for each panel. These are used to specify what type of graph is to be used (using the *Style* attribute), the descriptions to go in the *Legend* box (using the *LegendText* attribute), and to which panel these setting refer (using the *Panel* attribute).

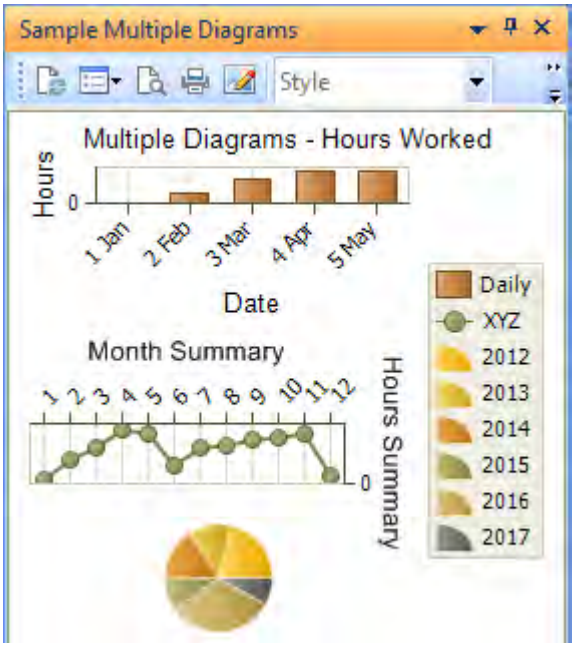


Figure 12-6: Multiple graphs within the same customized pane

The Elements of a Chart

The following are the elements of all graphs.

Title Element

The *Title* element resides within the *Titles* node and provides a title for the whole *Chart* (i.e. the one at the top of the customized pane). This element is used to specify whether the title should be displayed or not using the *Visible* attribute (where 0 is not to display, and 1 is to display). It also contains the title to be displayed against the *Text* attribute. The following XML extract would display a title of *Hours Worked*.

```
<Titles>
  <Title Visible='1' Text='Hours Worked'></Title>
</Titles>
```

If this element is changed after the customized pane is initially saved, the change takes effect the next time that the program containing the customized pane is loaded.

Each *Panel* within the *Chart* can have its own title, which will be covered in the *Panel Element* section.

Labels Element

The *Labels* element is used to define whether the labels that appear alongside the axis containing the static information should be straight or angled. If this element is changed after the customized pane is initially saved, the change is ignored during all subsequent displays of the customized pane.

If the *Labels* element is not present the labels will appear straight by default (see **Figure 12-7**). If the *Labels* element is present and the *Angled* attribute is set to 0 the labels will also appear straight.

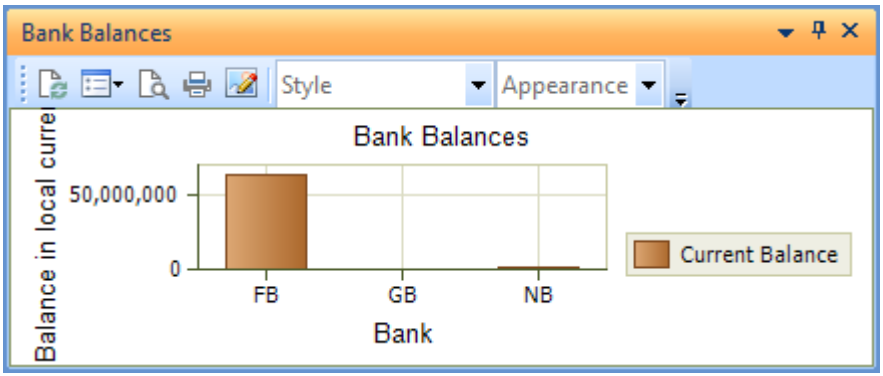


Figure 12-7: The *Labels* in the straight orientation

If the *Labels* element is present and the *Angled* attribute is set to 1, the labels will be displayed at an angle (see **Figure 12-8**).

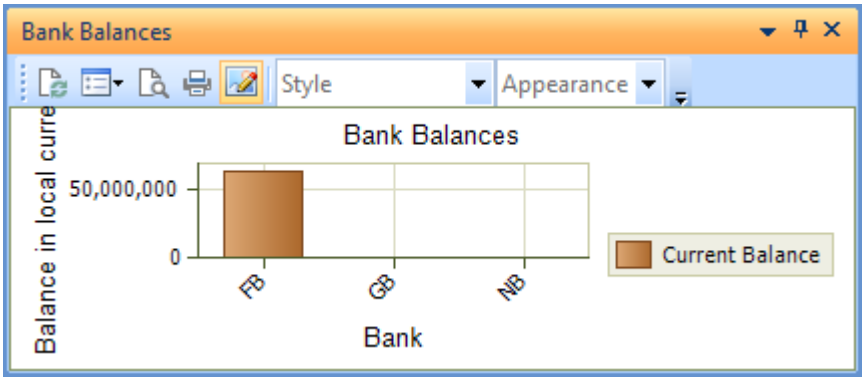


Figure 12-8: The *Labels* displayed at an angle

Legend Element

The *Legend* element is used to specify whether the box containing the legend should be displayed, and how many columns it should consume. If this element is changed after the customized pane is initially saved, the change is ignored during all subsequent displays of the customized pane.

The *Legend* is displayed if the *Visible* attribute is set to 1, and the number of columns to be displayed is defined using the *ColumnCount* attribute. **Figure 12-9** shows the *Legend* with the *ColumnCount* attribute is set to 1, and **Figure 12-10** with it set to 2.

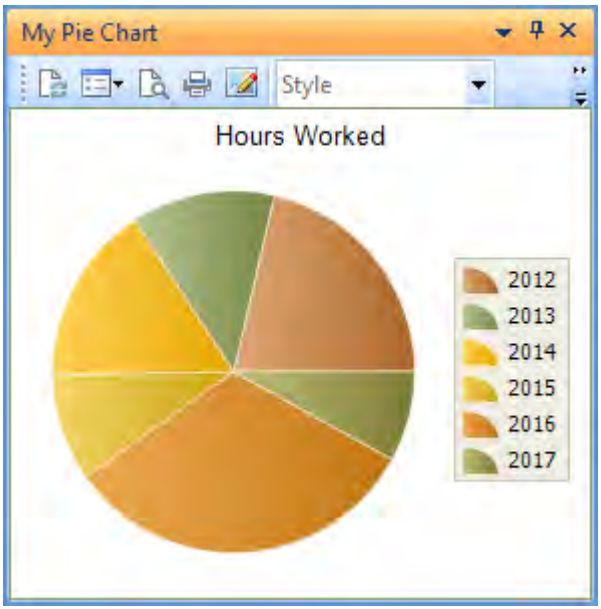


Figure 12-9: The *Legend* defined to take up one column

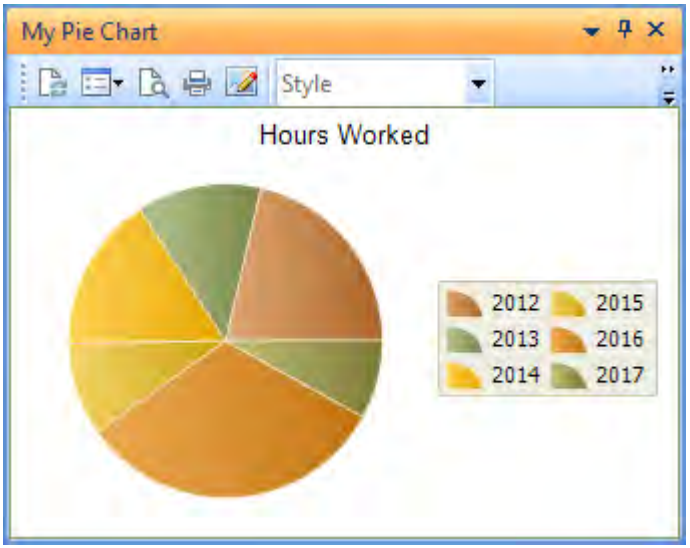


Figure 12-10: The *Legend* defined to take up two columns

PanelDirection Element

The *PanelDirection* element is used to specify whether the graph should appear in its standard orientation, or if it should be rotated through 90 degrees using the *Rotated* attribute. If this element is changed after the customized pane is initially saved, the change is ignored during all subsequent displays of the customized pane.

If the *PanelDirection* element is not present, or its *Rotated* attribute contains 0, it means that the graph must appear as designed. **Figure 12-8** is an example of a *Bank Balances* graph where the *PanelDirection* element is not present. If the element is present and its *Rotated* attribute is set to one (as per the code snippet below) the graph will be rotated by 90 degrees. This can be seen in **Figure 12-11**.

```
<PanelDirection Rotated='1' />
```

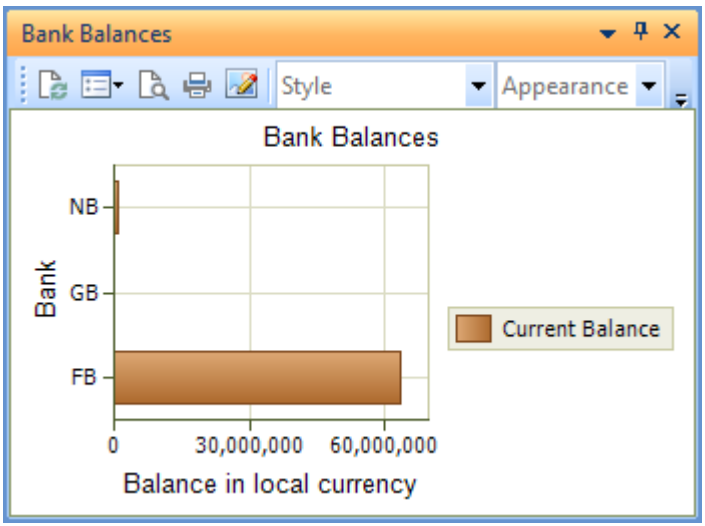


Figure 12-11: A *Rotated* graph

Panel Element

The *Panel* element is used to supply information regarding the X and Y axes of the graph. These are defined within the AxisX and AxisY elements respectively. The X-axis is the one that runs from left to right, and the Y-axis is the one that runs from bottom to top.

When there are multiple *Panel*s against a *Chart* there should be the same number of *Panel* elements in the XML. The first *Panel* element in the XML refers to the first panel, the second *Panel* element to the second panel, etc. Below are the *AxisX* and *AxisY* elements for the graph in **Figure 12-8**.

```
<AxisX><Title Text='Bank' /></AxisX>
<AxisY Decimals='0'><Title Text='Balance in local currency' /></AxisY>
```

Against the *AxisX* node is the *Title* element that contains the title within the *Text* attribute. The bank codes come from the data used to populate the graph (and there is one *Label* element for each bank in the XML used to populate it). The *Title* of the X-axis defaults to appearing below the graph. This title can be moved to above it by supplying the *Alignment* attribute to the *AxisX* element, as per the example below. **Figure 12-12** shows the results.

```
<AxisX Alignment='top'><Title Text='Bank' /></AxisX>
```

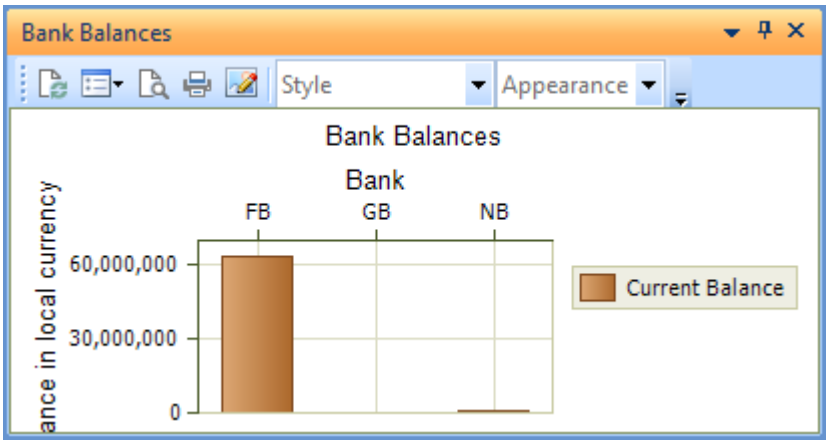


Figure 12-12: Moving the X axis *Title* to the top

As the *AxisY* element will contain values, the *Decimals* attribute enables the number of decimal places to display to be set, in this case to zero. The *AxisY* element also has a *Title* element, and this defaults to the left of the graph. This can be moved to the right by supplying the *Alignment* attribute using the code below. This is one line, but has wrapped around on this page. The effects of this can be seen in **Figure 12-13**.

```
<AxisY Decimals='0' Alignment='right'><Title Text='Balance in local currency'
/></AxisY>
```

If this element is changed after the customized pane is initially saved, the change is ignored during all subsequent displays of the customized pane.

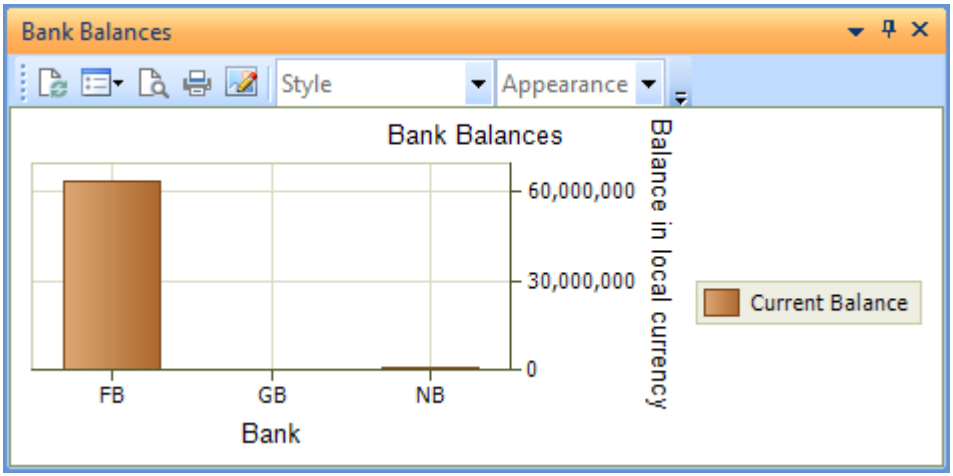


Figure 12-13: The *Title* moved to the right

Series Element

The *Series* element is used to define the type of graph (using the *Style* attribute), what will appear within the *Legend* box (using the *LegendText* attribute), and to which *Panel* this *Series* entry relates (using the *Panel* attribute). There should be one *Series* element for each *Panel* when there are multiple graphs in one customized pane. If this element is changed after the customized pane is initially saved, the change is ignored during all subsequent displays of the customized pane.

```
Series LegendText='Daily' Style='Bar' Panel='1' />
Series LegendText='Summary' Style='Line' Panel='2' />
Series LegendText='Summary Hours' Style='Pie' Panel='3' />
```

Depending on the *Style* attribute, the *Legend* box will contain either one entry for a graph, or one for each value within the graph. The graphs that will contain one entry per value are *Pie*, *Bubble*, *Pyramid*, and *Funnel*. All the others will only pass one entry to the *Legend* box. **Figure 12-13** shows a *Bar chart* and this only provides one entry to the *Legend* box. **Figure 12-10** shows a *Pie chart* and this provides one value to the *Legend* box for each segment of the pie. **Figure 12-6** shows a customized pane that contains a *Bar chart*, a *Line graph*, and a *Pie chart*. The *Legend* box contains one entry for the *Bar chart*, one for the *Line graph*, and six for the *Pie chart* (one for each segment).

Tooltips

Each graph type displays a tooltip when the operator moves their mouse pointer over certain parts of the graph. In the case of a *Pie chart*, *Bubble chart*, a *Pyramid*, a *Bar chart*, and a *Funnel chart*, moving the mouse pointer over any colored part of the chart displays the tooltip. With the other graph types the mouse pointer must be moved over the point representing the value.

The tooltip displays a combination of the text supplied against the *LegendText* attribute of the *Series* element, the text supplied against the *Label* attribute of the *Point* element, and the value.

Populating Multiple Graphs within a Single Customized Pane

When populating a graph with data, the data is provided in XML that has a root element of *CodejockChart*. Within the *CodejockChart* node is one or more *Series* elements, one for each graph in the customized pane. If there is only one graph in the customized pane the *Series* element does not need a *Panel* attribute.

The XML below was used to populate the *Pie chart* that appears in **Figure 12-10**. The *Series* element does not have a *Panel* attribute.

```
<CodejockChart>
  <Series>
    <Points>
      <Point Label='2012' Value='2000' />
      <Point Label='2013' Value='1200' />
      <Point Label='2014' Value='1500' />
      <Point Label='2015' Value='900' />
      <Point Label='2016' Value='3000' />
      <Point Label='2017' Value='750' />
    </Points>
  </Series>
<CodejockChart>
```

However, the second line of the XML could have contained a *Panel* attribute to stipulate that the contents are for the first graph, in which case the second line would match the following line.

```
<Series Panel='1'>
```

When there are multiple graphs within the customized pane it is recommended that you supply the *Panel* attribute to each *Series* element so that it is guaranteed that the correct data is used to populate each graph. The following is the outline structure of the XML that would be supplied to a customized pane that contains three graphs. Each *Series* entry has a *Panel* attribute so that the values populate the correct graph.

```
<CodejockChart>
  <Series Panel='1'>
    <Points>
      <Point Label='1 Jan' Value='0'  />
        --  etc.  --
      <Point Label='12 December' Value='32' />
    </Points>
  </Series>
  <Series Panel='2'>
    <Points>
      <Point Label='1' Value='2' />
        --  etc.  --
      <Point Label='12' Value='5' />
    </Points>
  </Series>
  <Series Panel='3'>
    <Points>
      <Point Label='2006' Value='2000' />
        --  etc.  --
      <Point Label='2017' Value='750' />
    </Points>
  </Series>
</CodejockChart>
```

Pie Chart

A *Pie chart* is a circle divided up in a similar way to cutting a pie. Each pie “slice” or segment is given a different color to differentiate them. The *Legend* (if displayed) explains what the segments represent. A tooltip containing the value and what it represents is displayed if the operator moves their mouse pointer over a segment.

Behind the scenes, the sum of all the values is divided by 360 to find the value of each degree, and the sizes are allocated accordingly. Unless the values are significantly different, it can be difficult to see which segments are larger. The pie chart is also not good at displaying information that is made up of lots of values, as it is sometimes difficult to work out which segment represents which option.

Figure 12-14 shows a *Pie chart* that contains three options. The operator has moved the mouse pointer over the *Yes* segment. As the values are not similar, and there are not many items to represent, this is a good use of a pie chart.

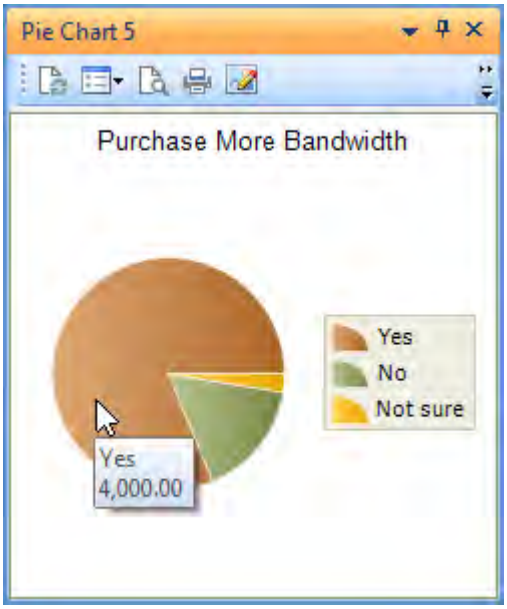


Figure 12-14: A *Pie chart* example where it is easy to distinguish between the segments

Line Chart

A *Line chart* consists of a series of points that are typically plotted on the X-axis. Each point is joined to the next with a straight line. A line chart is typically used to show a trend in the data over a specific interval of time. The line always flows from left to right, no matter what sequence the values appear in the XML.

Figure 12-15 shows a line chart consisting of the sales by month for the year. The chart has been configured to display the title and values for the X-axis at the top of the screen, and the title for the Y-axis has been configured to appear on the right.

The biggest disadvantage of a line chart is that it is not easy to see the exact value that a point represents without using the mouse pointer to call up the tooltip.

Area Chart

An *Area chart* is similar to a *Line chart*, with the addition of the area between the line and the X axis being highlighted with a color (see **Figure 12-16**)

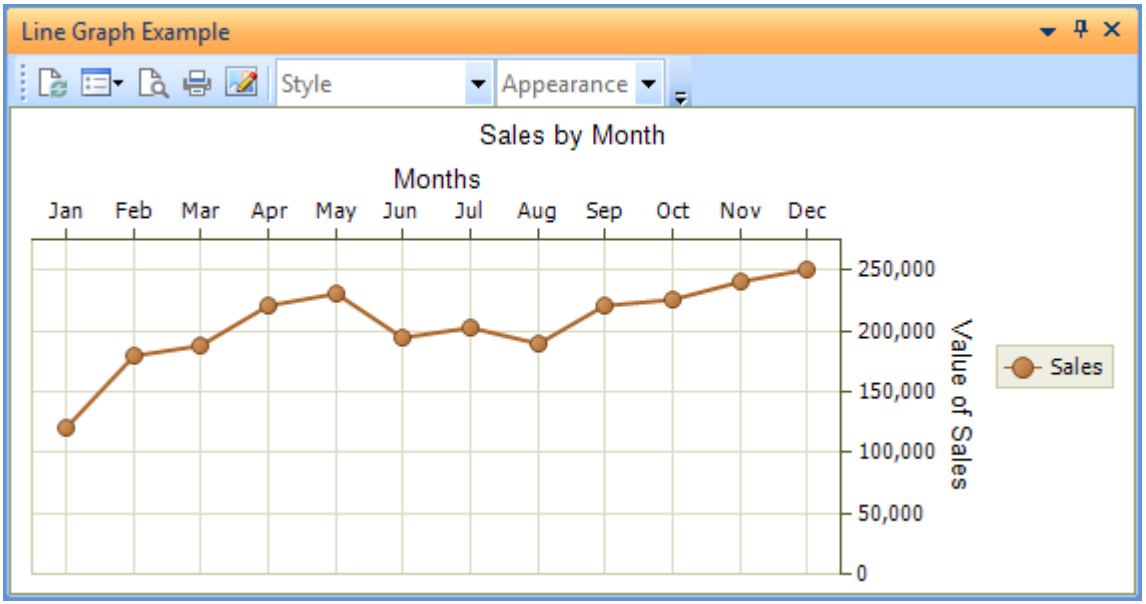


Figure 12-15: A *Line chart* showing sales for the year

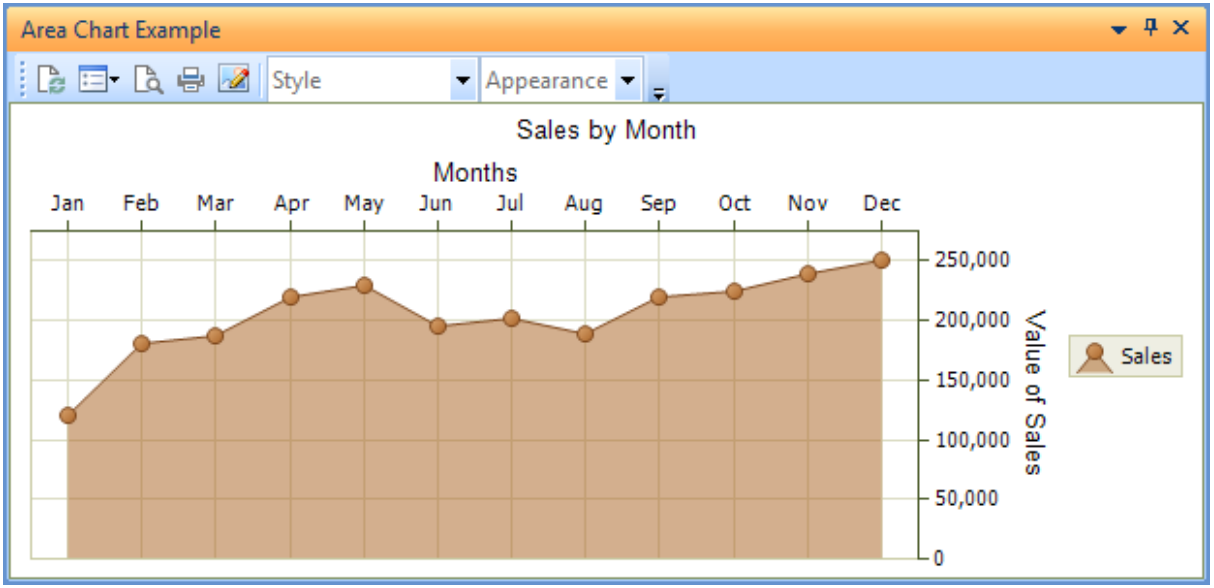


Figure 12-16: The same information drawn as an *Area chart*

Funnel Chart

A *Funnel chart* is a chart in the shape of a funnel. It is typically used to represent values that will decrease, or start at 100% and end with a lower percentage. An example is the number of prospective customers in the sales cycle. It is the depth of each segment (not the width) that represents the value for each level.

Figure 12-17 shows a funnel chart representing the different stages of the sales process. Depending on the requirements, this could be the number of prospects, or the potential revenue thought to be generated from these prospects.

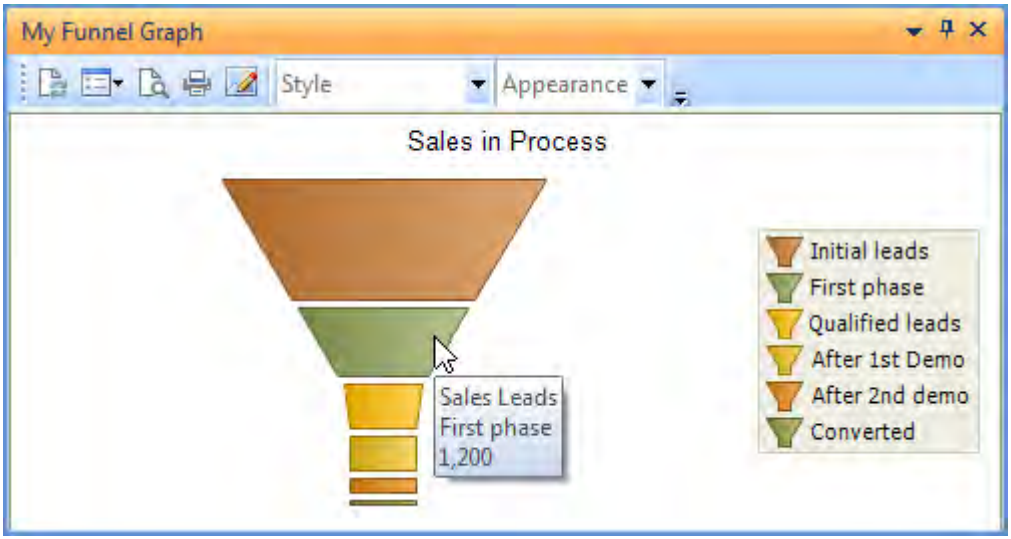


Figure 12-17: A *Funnel chart* representing the sales cycle

Bar Chart

A *Bar chart* is a series of horizontal or vertical rectangular bars that are proportional in length to the value. One of the axes shows the categories to be compared and the other represents the discrete values.

Bar charts make it easy to compare the values, although like a line chart, it is difficult to see the actual values without using the mouse pointer to display the tooltip.

Figure 12-18 shows a *Bar chart* representing the average number of hours worked per week each year.

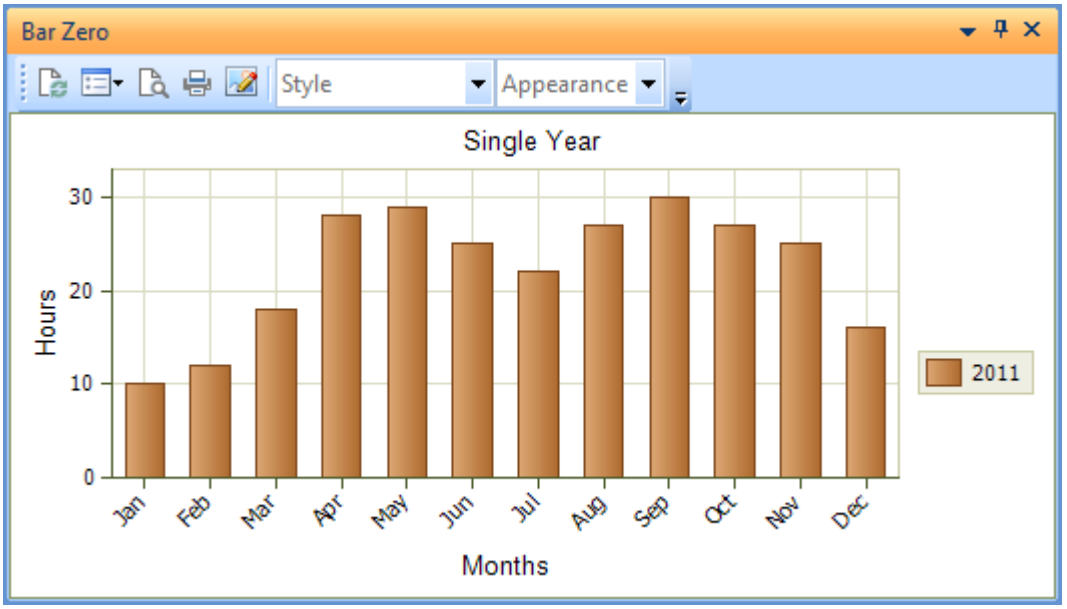


Figure 12-18: A *Bar chart* representing the average number of hours worked on a project per year

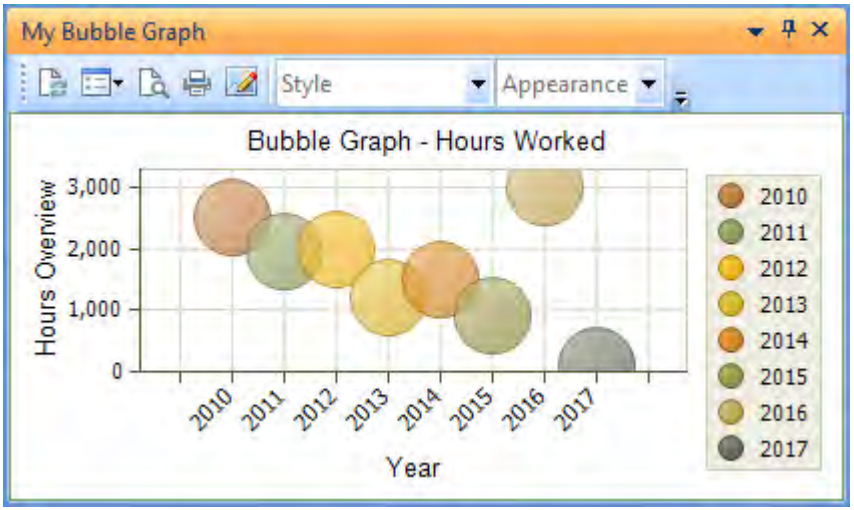


Figure 12-19: A *Bubble chart* showing the hours worked per year

Bubble Chart

A *Bubble chart* is similar to a bar chart except that instead of showing the value as a bar, a circle is displayed. All the circles are the same size, and it is the center point of the circle that represents the value. **Figure 12-19** shows a bubble chart that represent the total number of hours worked on a project per year. It is not always easy to see the difference between values, such as between 2011 and 2012.

Step Chart

A *Step chart* is similar to a line chart except that the line between the values is always horizontal. When a value changes a step appears between the two levels. This is useful when showing quantities that change intermittently, but remain constant between these changes. It highlights the changes, whereas on a line chart this might not be noticeable for small changes.

Figure 12-20 shows a *Step chart* where the value remains constant for a couple of months before changing. Because of the deliberate steps it is obvious when the value changed.

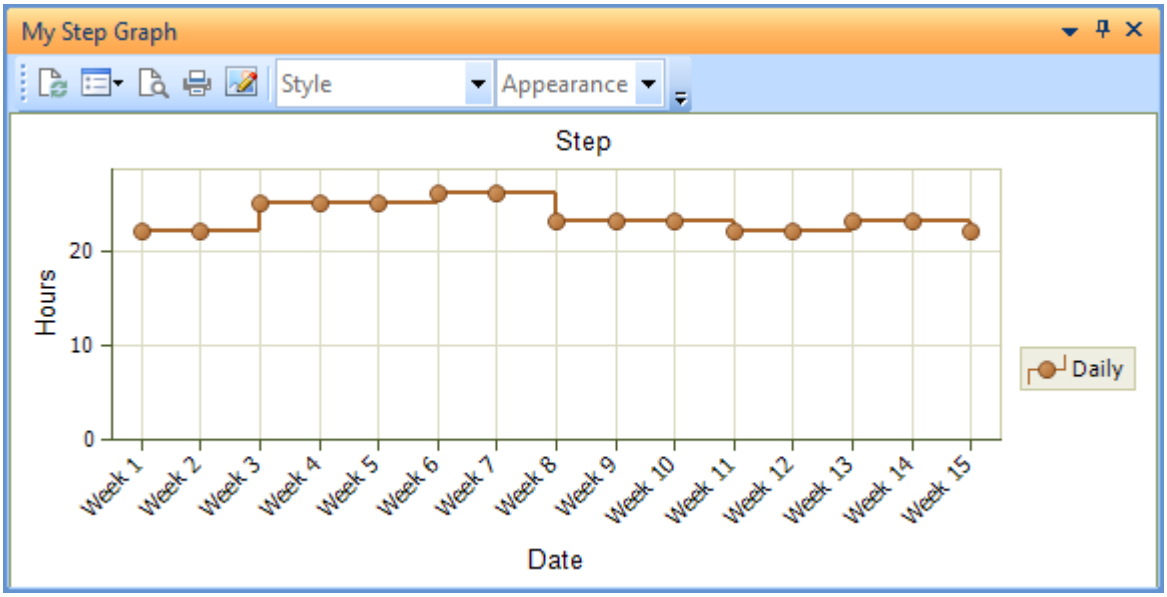


Figure 12-20: A *Step chart* showing how changes in a value can be easily seen using this chart type

Spline Chart

A *Spline chart* is similar to a line chart, the difference being that the lines between the points are curved instead of straight. Depending on your audience, a *Line chart* or *Spline chart* may be better at representing your data. A *Spline* chart can be seen in **Figure 12-21**.

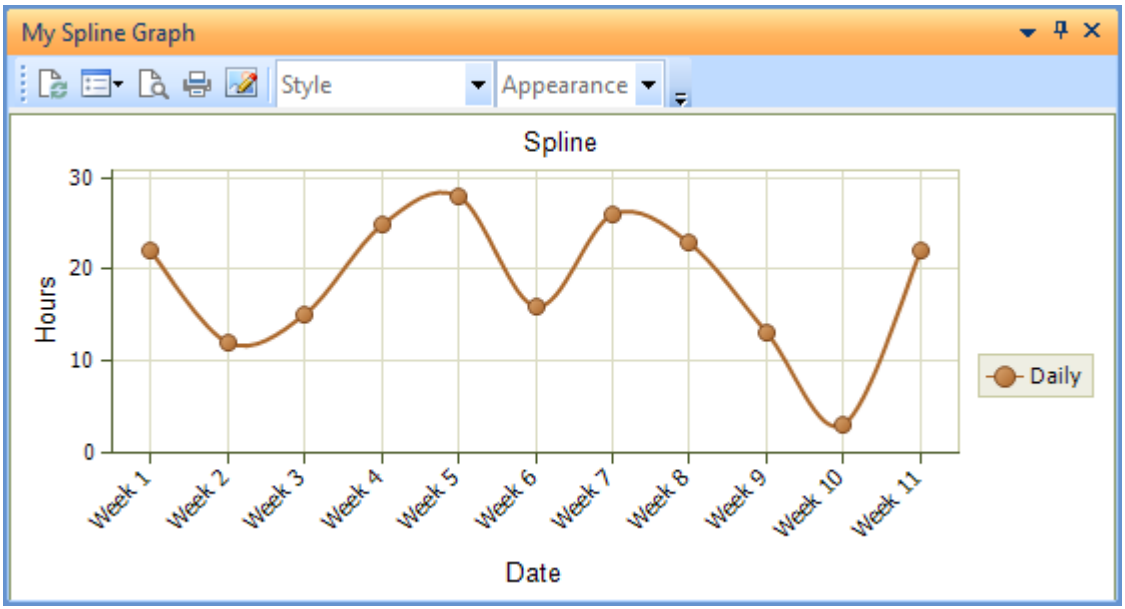


Figure 12-21: A *Spline chart* showing the curved lines between the plotted points

Pyramid Chart

A *Pyramid chart* consists of one of more segments making up a triangle. It is similar to a *Funnel chart* in that it is the height of each segment that is important, not the width, or the area. This can be a little misleading when the value being represented near the top is the same or larger than a value at the bottom. As the values are not shown it is not that easy to see what the real values are, as the mouse pointer must be moved over each one to use the tooltip to view the values.

In the example in **Figure 12-22**, it is difficult to compare the value that the uppermost segment is representing with that of the third segment down. The uppermost segment is representing the value 2,000 and the third one down is 1,800. The third one down appears much larger, but is representing a smaller value.

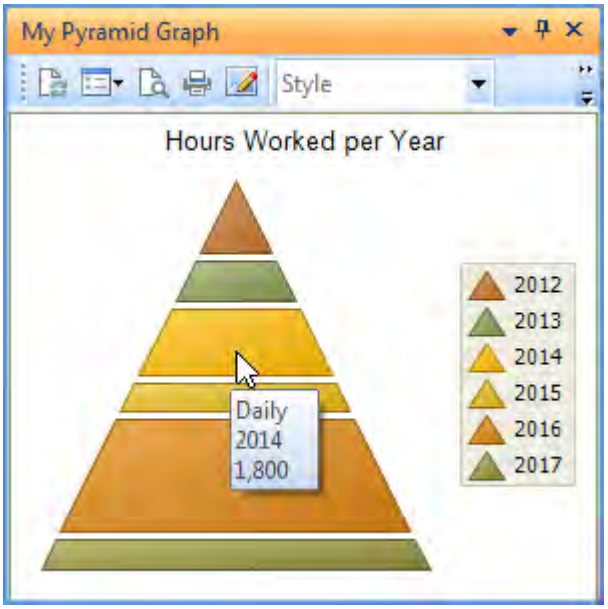


Figure 12-22: A *Pyramid chart* showing the hours worked per year

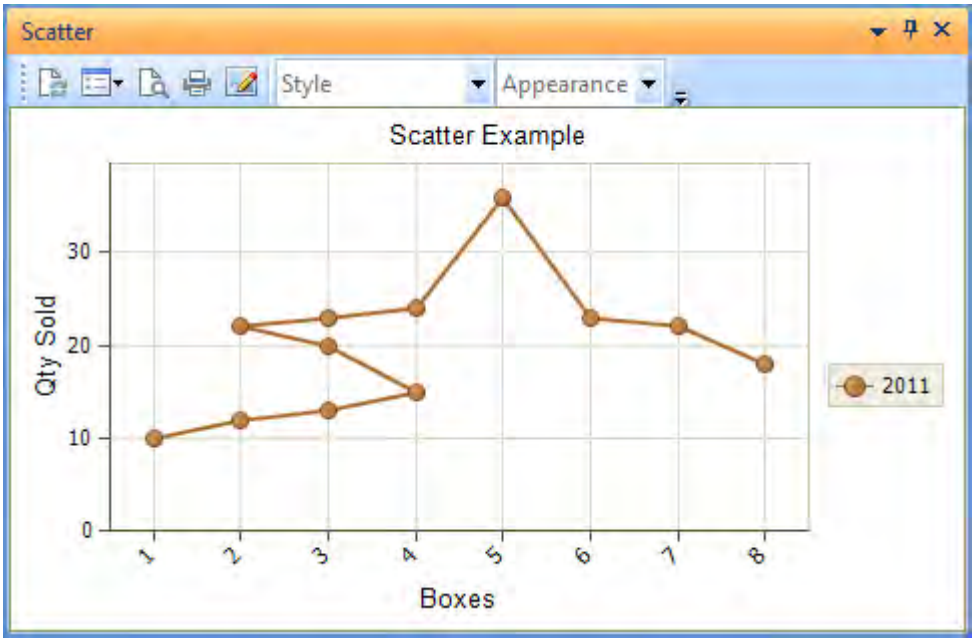


Figure 12-23: A *Scatter chart* showing the line following the sequence that the points were entered

Scatter Chart

The *Scatter chart* in SYSPRO is a scatter line chart. A scatter chart is similar to a line chart, only the scatter chart will plot the data values in the exact order that they appear in the XML, and connect the points with a line. A *Line chart* will always connect the points from left to right along the X-axis. **Figure 12-23** shows a scatter chart where the values do not flow from left to right on the X-axis.

The data used to populate this scatter chart appears below:

```
<CodejockChart>
  <Series>
    <Points>
      <Point Label='1' Value='10' />
      <Point Label='2' Value='12' />
      <Point Label='3' Value='13' />
      <Point Label='4' Value='15' />
      <Point Label='3' Value='20' />
      <Point Label='2' Value='22' />
      <Point Label='3' Value='24' />
      <Point Label='4' Value='25' />
      <Point Label='5' Value='35' />
      <Point Label='6' Value='23' />
      <Point Label='7' Value='22' />
      <Point Label='8' Value='18' />
    </Points>
  </Series>
</CodejockChart>
```

Gantt Chart

A Gantt chart (named after Henry Gantt in the 1910's) is typically used to show elements of a project or a (*Work in Progress*) job, and can also be used to show the dependencies between these. The names of the elements of the project or job run from top to bottom on the left of the chart. The number of days (or dates) appears across the top. Terminal elements appear as bars alongside the elements, and these consist of the planned values, and where appropriate, the actual values. A vertical progress line shows the current status of the project, or if dates are used, the line represents "today".

Figure 12-24 shows the *Sample Gantt Chart* template that is included with SYSPRO. This represents a project and has six elements on the left. The horizontal scale is days, and the progress line is set on day 23. The thicker of the bars against each element is the planned time taken for that task and the thinner is the actual time taken. This is just a sample showing what can be achieved, as it is unlikely that the tasks in the future will have been started already.

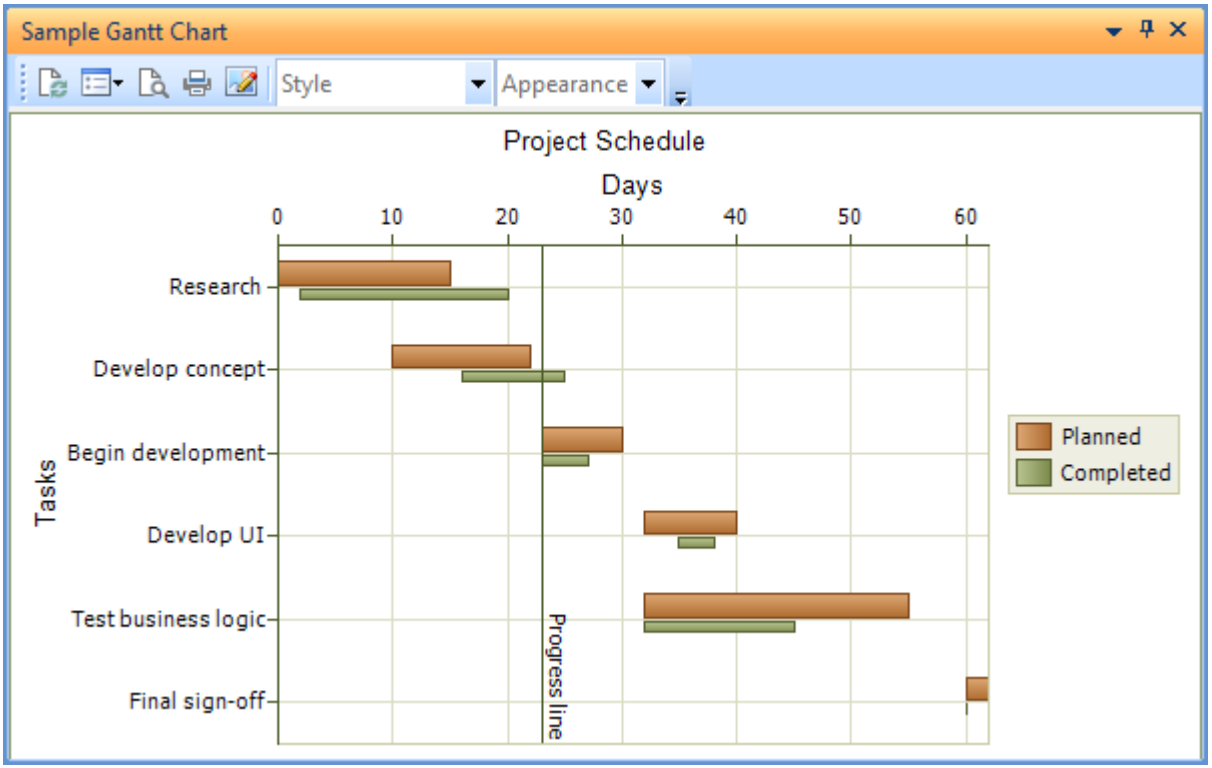


Figure 12-24: The *Sample Gantt Chart* template that ships with SYSPRO

The following XML is that used to create the structure of the Gantt chart in **Figure 12-24**. Most of the items in this XML have already been covered in the sections above, so only those that haven't, or those that include an attribute not previously been covered, will be mentioned here.

```
<CodejockChart>
  <Titles><Title Visible='1' Text='Project Schedule'></Title></Titles>
  <Panels>
    <Panel>
      <AxisX>
        <Title Text='Tasks' />
      </AxisX>
      <AxisY Decimals='0' Alignment='top'>
        <Title Text='Days' />
      </AxisY>
    </Panel>
  </Panels>
  <Series LegendText='Planned' Style='Gantt' />
  <Series LegendText='Completed' Style='Gantt' BarWidth='.3' />
</CodejockChart>
```

Within the *Series* element the *BarWidth* attribute specifies the width of the bar as a percentage. If the *BarWidth* attribute is not present the width of the bar defaults to 100 percent. The first of the two *Series* elements below does not contain a *BarWidth* attribute so the *Planned* bars in **Figure 12-24** are at 100%. The second *Series* element contains a *BarWidth* of .3 which means that the *Completed* bars in **Figure 12-24** are 30% of the standard size.

If you specify a *BarWidth* value greater than 100% the bar will appear this size but the spacing of the terminal elements will be compromised (see **Figure 12-25**).

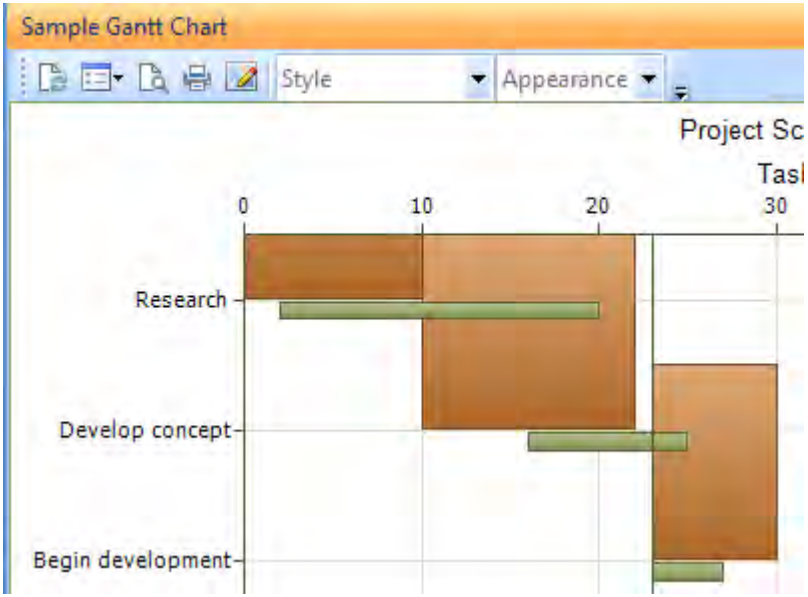


Figure 12-25: using a *BarWidth* value that is greater than 100%

The second XML is what is used to populate the Gantt chart in **Figure 12-24**. Within the *AxisY* section is the *ConstantLine* element. This is used to provide the position of the vertical progress line and the text that must appear alongside it.

For each of the *Series* elements in the structure XML there should be a *Series* section in the XML that is used to populate the chart. Each of these *Series* sections contains a *Points* section, and this contains one or more *Point* elements.

The *Point* elements contain a *Value* attribute that specifies the start point of the bar, and a *Value2* attribute that specifies the end point of the bar. The *Point* elements within the first *Series* element should contain the labels to appear on the left of the chart. Although the second series in the XML provided in this sample also contains labels, this is not being used and could be omitted.

The sequence in which the *Point* elements appear within the *Points* section is the sequence that the bars will appear in the chart.

```
<CodejockChart>
  <Panels>
    <Panel>
      <AxisY>
        <ConstantLine Text='Progress line' Value='23'/>
      </AxisY>
    </Panel>
  </Panels>
  <Series>
    <Points>
      <Point Label='Research' Value='0' Value2='15' />
      <Point Label='Develop concept' Value='10' Value2='22' />
      <Point Label='Begin development' Value='23' Value2='30' />
      <Point Label='Develop UI' Value='32' Value2='40' />
      <Point Label='Test business logic' Value='32' Value2='55' />
      <Point Label='Final sign-off' Value='60' Value2='62' />
    </Points>
  </Series>
  <Series>
    <Points>
      <Point Label='Research' Value='2' Value2='20' />
      <Point Label='Develop concept' Value='16' Value2='25' />
      <Point Label='Begin development' Value='23' Value2='27' />
      <Point Label='Develop UI' Value='35' Value2='38' />
      <Point Label='Test business logic' Value='32' Value2='45' />
      <Point Label='Final sign-off' Value='60' Value2='60' />
    </Points>
  </Series>
</CodejockChart>
```

Overlaying Multiple Sets of Data on One Chart

Several chart types lend themselves to displaying multiple sets of data on the same chart. These are the Bar chart, Line chart, Step chart, Scatter chart, Area chart, and Spline chart. Each of these charts is built in a similar way. The Bubble chart can also display multiple sets of data, but this is done in a slightly different way.

Overlaid Bar Chart Example

It is possible to overlay multiple sets of data on a bar chart. **Figure 12-18** shows the average number of hours worked per month in 2011. If you have the figures for the following years, these can be overlaid (see **Figure 12-26**). The maximum number of sets of values that can be overlaid in a bar chart is four, so in the example using years, only four years can be displayed.

There are two main differences in the code between these bar charts. The first is in the structure of the chart that typically appears within the *OnLoad* function. This contains a *Series* element for each set (in this case 'year') of data that must appear. The content of the four *Series* elements appears below:

```
<Series LegendText='2011' Style='Bar' />
<Series LegendText='2012' Style='Bar' />
<Series LegendText='2013' Style='Bar' />
<Series LegendText='2014' Style='Bar' />
```

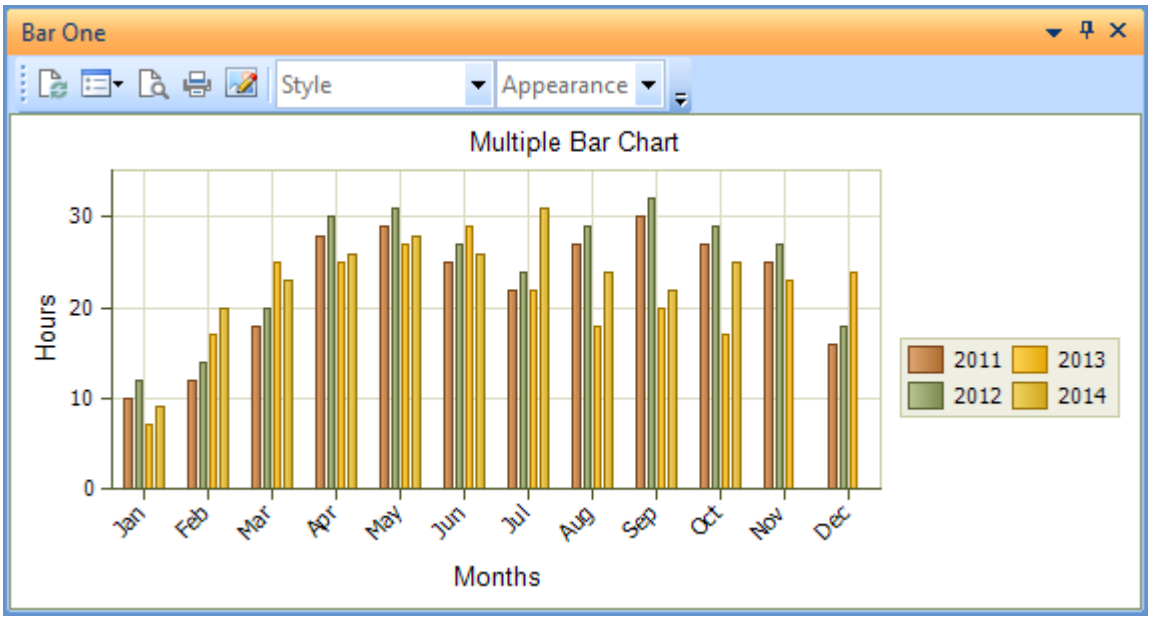


Figure 12-26: Multiple bar chart columns

The second difference is that there are multiple sets of data contained within the XML used to populate the chart within the *OnRefresh* function. In the example that appears in **Figure 12-26**, each of these sets of data reflect the values for one year. These sets of data appear within their own *Series* element.
Each of the *Series* elements has a *Panel* attribute. The value against this attribute matches the *LegendText* attribute of the *Series* element in the structure definition XML.

Within the *Series* node is a *Points* element, and within this node are the *Point* elements that contain the values.

An overview of the XML structure used to populate the chart in **Figure 12-26** appears below. Only the first (Jan) and last (Dec) entries of each data set are present to reduce the size of the code and make it easier to follow:

```
<CodejockChart>
  <Series Panel='2011'>
    <Points>
      <Point Label='Jan' Value='10' />
        --  etc.  --
      <Point Label='Dec' Value='16' />
    </Points>
  </Series>
  <Series Panel='2012'>
    <Points>
      <Point Label='Jan' Value='12' />
        --  etc.  --
      <Point Label='Dec' Value='18' />
    </Points>
  </Series>
  <Series Panel='2013'>
    <Points>
      <Point Label='Jan' Value='7' />
        --  etc.  --
      <Point Label='Dec' Value='24' />
    </Points>
  </Series>
  <Series Panel='2014'>
    <Points>
      <Point Label='Jan' Value='9' />
        --  etc.  --
      <Point Label='Dec' Value='0' />
    </Points>
  </Series>
</CodejockChart>
```

Samples of Other Overlaid Charts

The Line chart, Step chart, Scatter chart, Area chart, and Spline charts all work in the same way. When creating the chart the *Style* attribute of *Bar* in the structure definition XML would be replaced with the appropriate style.

```
<Series LegendText='2011' Style='Bar' />
<Series LegendText='2012' Style='Bar' />
<Series LegendText='2013' Style='Bar' />
<Series LegendText='2014' Style='Bar' />
```

A *Line chart* containing four years of data appears in **Figure 12-27**.

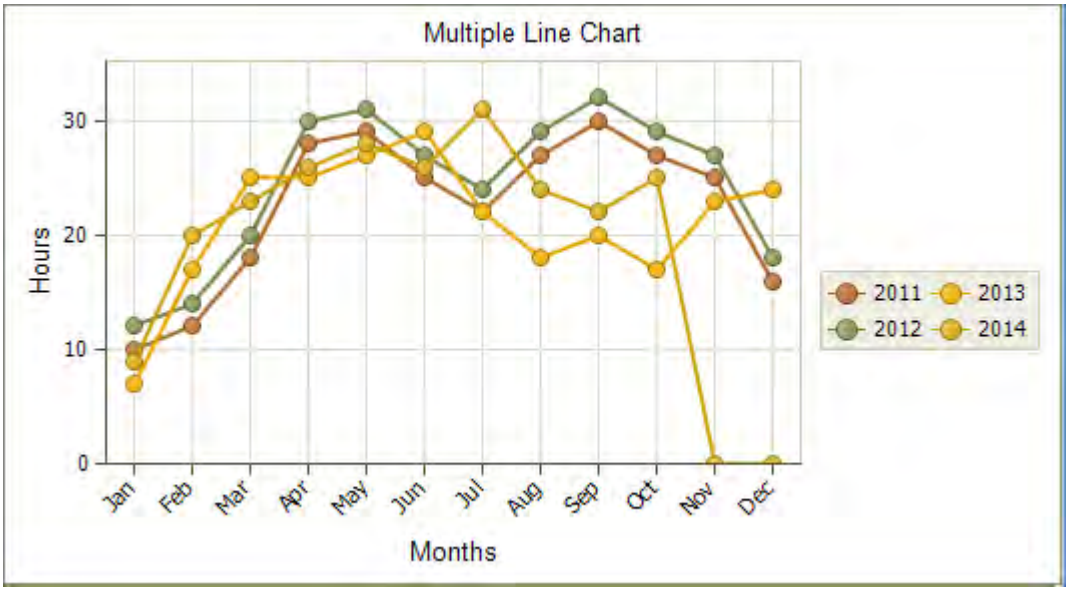


Figure 12-27: A *Line chart* containing multiple overlays

The *Area chart* only lends itself to displaying multiple sets of data when each set gets progressively smaller. In the example in **Figure 12-28**, where some years have both bigger and smaller months, it becomes difficult to read. This is the same data as used in the Line chart in **Figure 12-27**, which is easier to read.

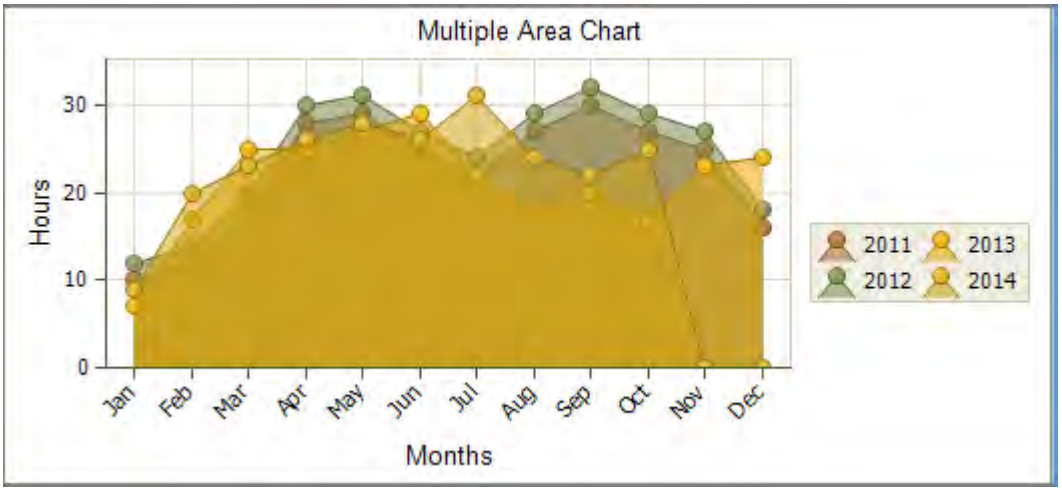


Figure 12-28: An *Area chart* containing multiple overlays

Figure 12-29 shows a *Step chart* containing just two years. The step chart is good at highlighting when a stable value changes, even by a small amount, when multiple values are overlaid they become difficult to read, especially when their values intersect each other.

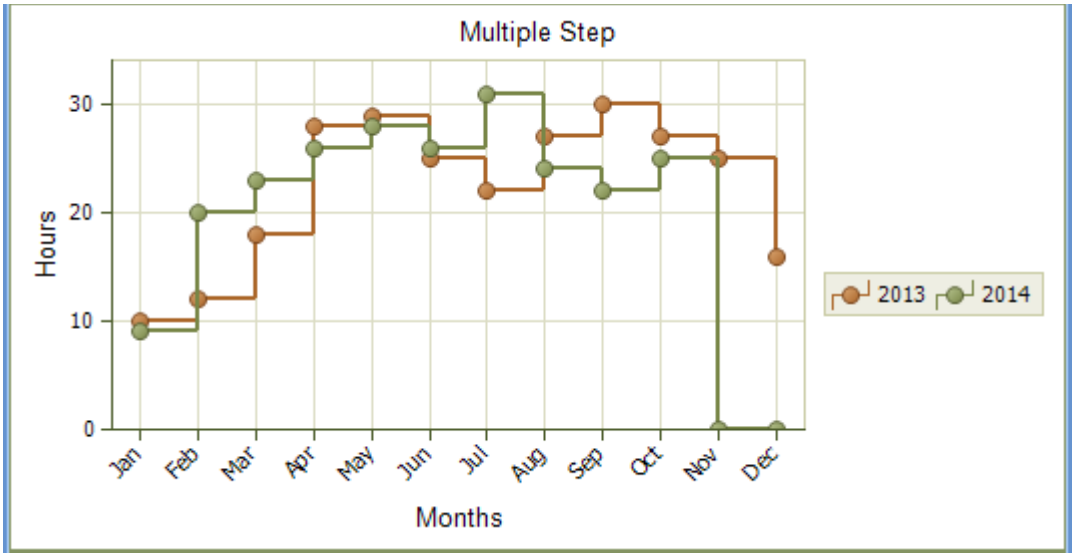


Figure 12-29: A *Step chart* showing two overlays

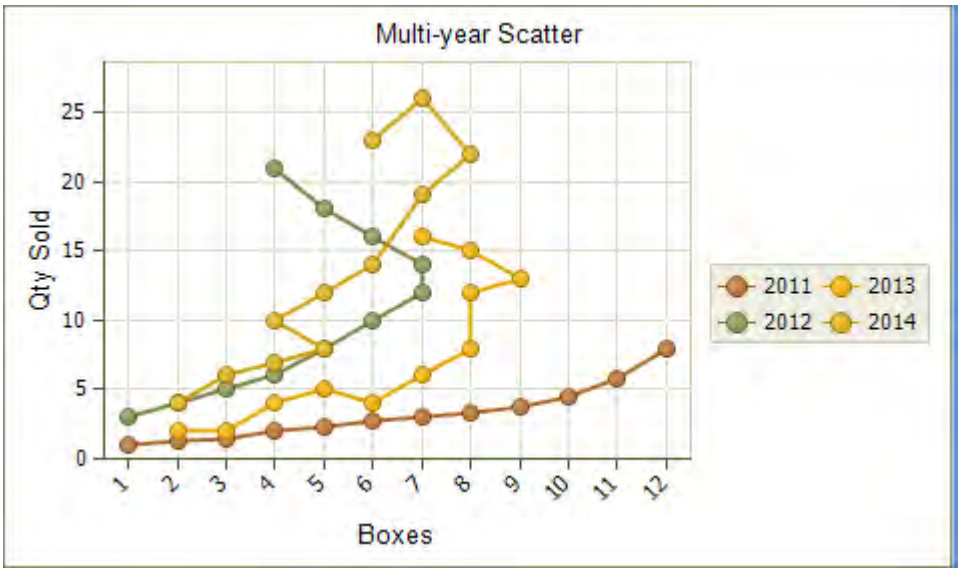


Figure 12-30: A *Scatter chart* showing multiple years of data

Figure 12-30 shows a *Scatter chart* displaying values for four years. Note that the lines follow the sequence that the values appear within the XML. **Figure 12-31** shows a *Spline chart* displaying values for four years.

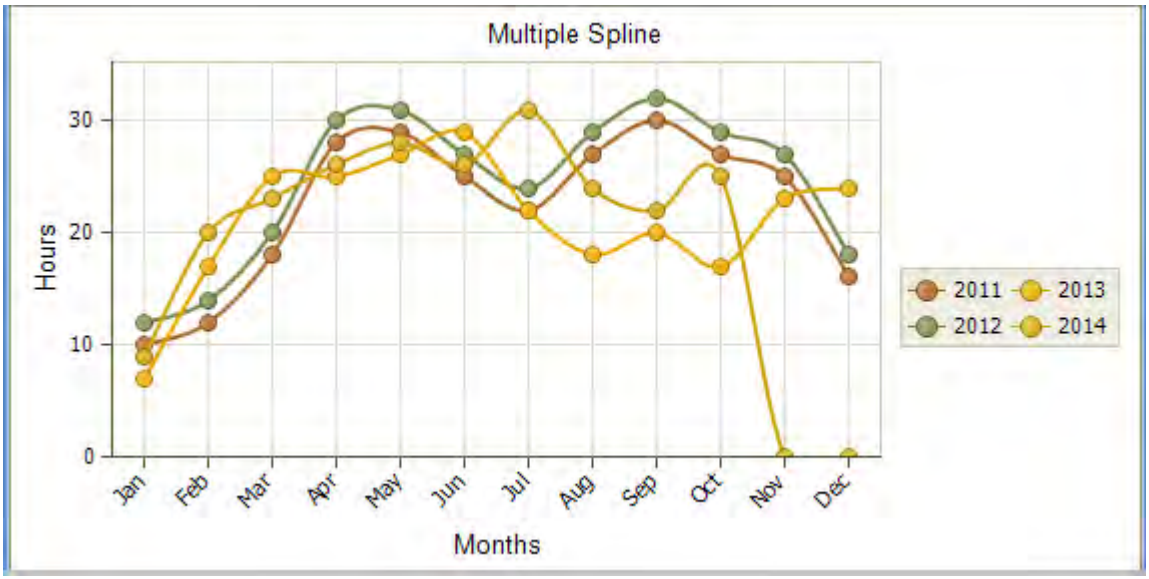


Figure 12-31: A *Spline chart* showing multiple years of data

"Overlaid" Bubble Chart

The overlaid *Bubble chart* is slightly different to the other overlaid charts, and technically it isn't overlaying the different sets of values. All the values appear within one *Series* node in the data XML, so it is just plotting multiple entries on the X-axis.

An "overlaid" bubble chart appears in **Figure 12-32**. This has four values four each month. The values appear in different colors, so this is only useful if you do not need to be able to differentiate between the sets of data.

When defining the structure of the bubble chart, the XML only needs to contain one *Series* element. When defining the data XML all the values are included in one *Series* element.

You do not need to have the same number of values for each entry on the X-axis. Whereas it is logical to have this when plotting months/years, the bubble chart could just as easily be plotting the number of drinking glasses in each box that were broken on delivery, by type of glass. If this is the case you could have multiple values for one type of glass, and none for another.

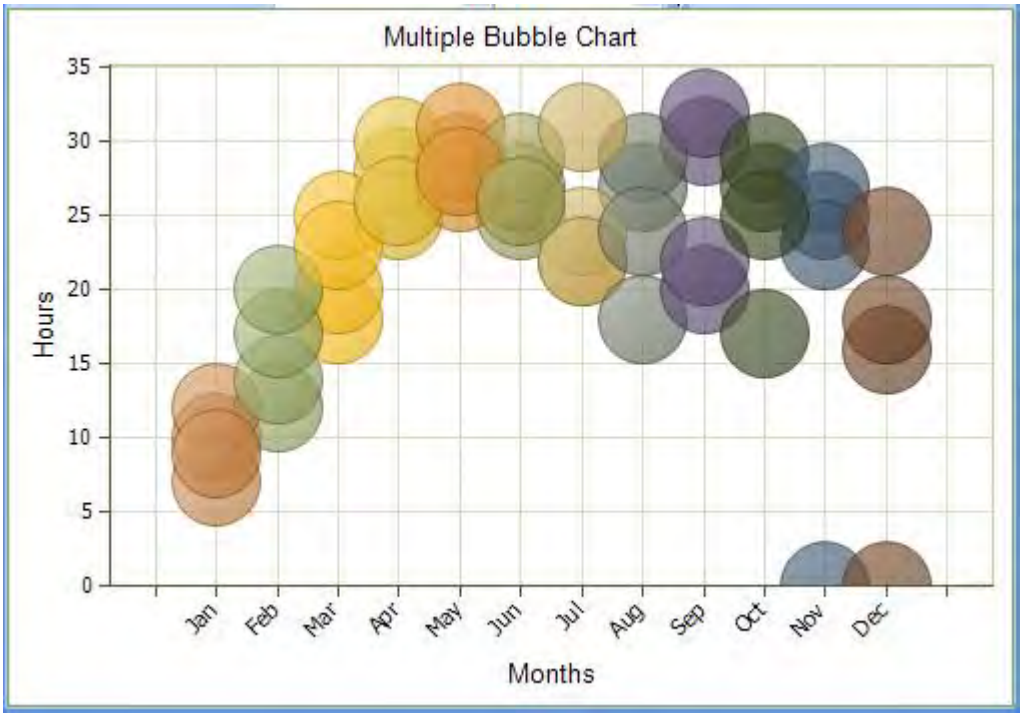


Figure 12-32: An *Bubble chart* displaying multiple values for each month

Chapter 13 - PDF Viewer, Web Browser, Rich Text Notepad, and SRS/Crystal Report

Object Type: PDF Viewer

The *PDF Viewer* object type is used to display the contents of a PDF file in a customized pane. There is an option to display a default/static PDF when the pane is loaded, and within the VBScripting there is an option to build up the name of a PDF file and display it, such as when the pane is refreshed.

When the *Object type* is *PDF Viewer*, there is a pane at the bottom of the *Customized Pane Editor* screen called *PDF Viewer*. This pane contains one option called *PDF document* (see **Figure 13-1**). If this contains the name and location of a PDF document, this document will be displayed as the customized pane is loaded. Unless VBScript coding causes a different document to be displayed, this document will remain on the screen until the program is closed.

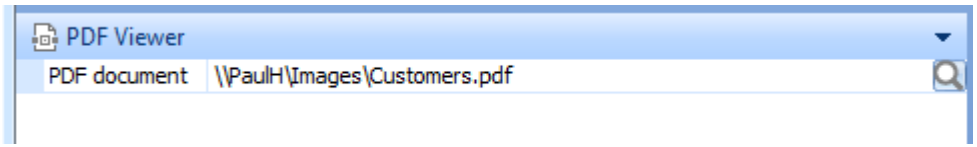


Figure 13-1: The *PDF document* option that is used to display a static or default document

This option can be used to display a static document such as in **Figure 13-2** (in which case no coding is required). It can also be used to display a default PDF document that will appear until the first proper document is displayed, such as in the *Inventory Query* program, before the first stock code has been selected. Note that the supplied address is relative to the client workstation, so this example uses a share to hold the images. This means that the same customized pane can be used for a role, or multiple operators, and the document will be found and displayed.

Within the VBScript code, the name and location of the PDF document can be passed to the *PDFDocument* variable, and the document will be displayed. The *PDFDocument* variable resides within the *CustomizedPane* section of the *Variables* pane (see **Figure 13-3**). As usual, this is best explained with an example.

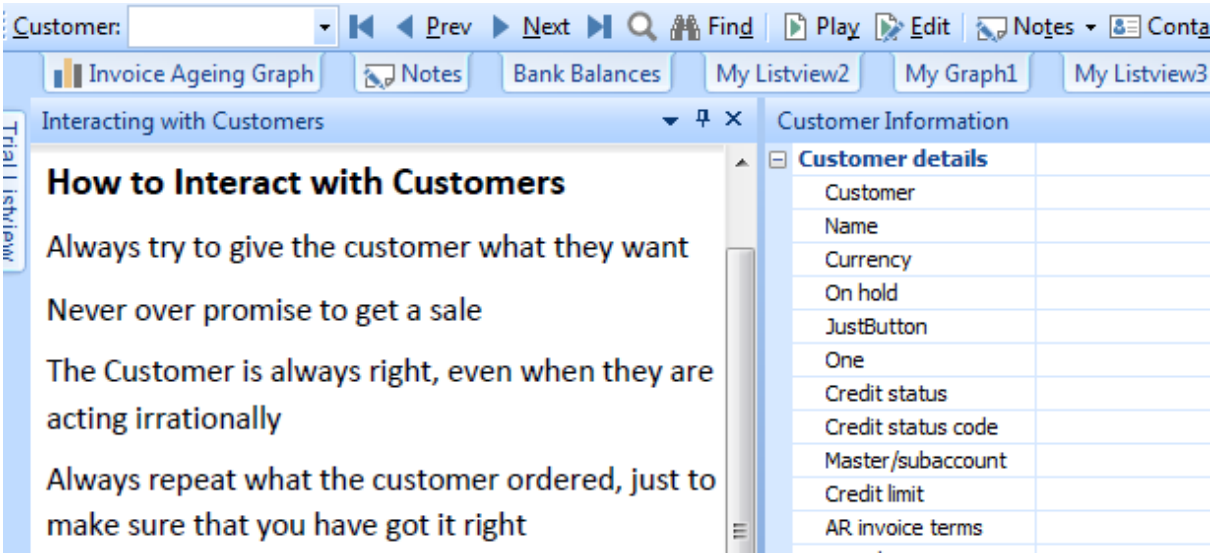


Figure 13-2: A static document displayed within the customer query

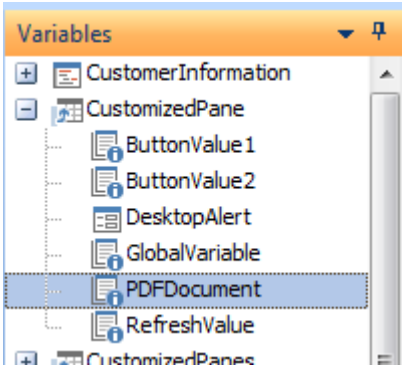


Figure 13-3: The *PDFDocument* variable that is used to display a document

The following example shows how a stock code specification can be displayed in a customized pane within the *Sales Order Entry* program. When the program is initially loaded the pane should display a blank PDF document (or it could display a PDF document containing the company logo). When a stock code is entered, the specification sheet for this stock code must be displayed, if one exists.
The first step is to create the customized pane against the *Stocked Line* pane. Within the *Customized Pane Editor* you must provide a *Window title*, set the *Object type* to *PDF Viewer*, and add the name (and path) of your default document to the *PDF document* option (see **Figure 13-4**).

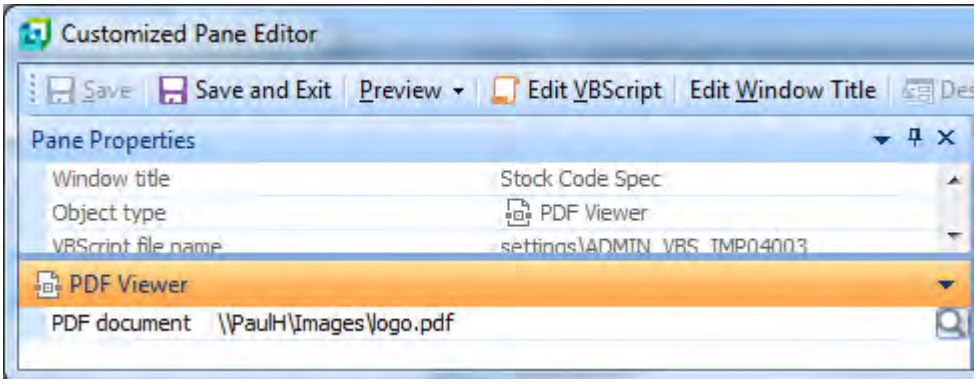


Figure 13-4: The settings for the *Customized Pane Editor*

Once this has been done, click on the *Edit VBScript* button to call up the *VBScript Editor* screen. Select the *OnRefresh* event and click on the *Edit VBScript* button to create the *CustomizedPane_OnRefresh* function.

Start the code by creating four variables called *fs*, *Location*, *FullFileName*, and *CompanyLogon*. The *fs* variable will be used to instantiate the *File System Object* (so that you can check if the file exists). The *Location* variable will hold the name of the share containing the documents. The *FullFileName* variable will contain the built up name and location of the specification document for this stock code. The *CompanyLogo* variable will contain the full name and location of the document to be displayed if there is no specification document for this stock code. The *DIM* statement to create these variables should match that below.

```
Dim fs, Location, FullFileName, CompanyLogo
```

The next line of code adds the name of the share to the *Location* variable. The address of the share in this example must be changed to match your location. Note that the location ends with a backslash, as this is required before the name of the document, and it is easier to add it here than when building up the filename.

```
Location = "\\PaulH\Images\"
```

Add a new line that populates the *CompanyLogo* variable with the contents of the *Location* variable, and the name of the document to be displayed when there is no specification document for the stock code. The code to do this appears below.

```
CompanyLogo = Location & "Logo.pdf"
```

The next line of code populates the *FullFileName* variable with the contents of the *Location* variable, the current stock code (which will be passed in the *RefreshValue* variable) and the document's suffix.

The code to do this appears below. The *RefreshValue* variable name can be added by double-clicking on its name within the *CustomizedPane* section of the *Variables* pane.

```
FullFileName = Location & CustomizedPane.CodeObject.RefreshValue & ".pdf"
```

The next line of code instantiates the *File System Object* so that the code can interrogate the file system. Whenever you need to communicate with the *File System Object* you can now use the string *fs*.

```
Set fs = CreateObject("Scripting.FileSystemObject")
```

The next line of code tests to see if a file exists with the filename and path that you have built up and used to populate the *FullFileName* variable.

```
If fs.FileExists(FullFileName) Then
```

This is followed by a line that specifies what should happen if the file does exist. In this case it is to populate the *PDFDocument* variable with the name and path of the document.

```
CustomizedPane.CodeObject.PDFDocument = FullFileName
```

This is followed by an *ELSE* statement, which is followed by a line specifying what should happen if the file does not exist.

```
Else
CustomizedPane.CodeObject.PDFDocument = CompanyLogo
```

Finally, the *IF* statement is closed using an *END IF* statement.

```
End If
```

The completed code should match the code below (but with your own location). Note that lines starting with an apostrophe are comment lines and are only present to make it easier to follow this code. They do not need to be in your code. Also note that lines that are indented do not need to be indented. The indentations are only present to make it easier to follow the logic.

Click on the *Syntax Check* button on the toolbar to find any basic typos that have been made. Then save this code, and *Save and Exit* the customized pane, back to the *Sales Order Entry* program.

There is one more task to be completed before you can use your customized pane. The *Stocked Line* pane needs to contain the code to fire the customized pane's *OnRefresh* event whenever the stock code is changed. Right-click on the *Stock code* prompt in the *Stocked Line* form, and select *Macro for: IMP040LA* from the displayed menu. Highlight the *OnAfterChange* event and click on the *Edit VBScript* button.

```
Function CustomizedPane_OnRefresh()
' Create the variables to be used in the code
Dim fs, Location, FullFileName, CompanyLogo

' Populate the Location variable with the name of the share containing
' the documents
Location = "\\PaulH\Images\"

' Populate the CompanyLogo variable with the location and name of the
' document to be displayed if no stock code specification exists
CompanyLogo = Location & "Logo.pdf"

' Populate the FullFileName variable with the location of the
' documents, followed by the RefreshValue variable and the document
' suffix
FullFileName = Location & CustomizedPane.CodeObject.RefreshValue & ".pdf"

' Instantiate the File System Object
Set fs = CreateObject("Scripting.FileSystemObject")

' Check to see if the document exists
If fs.FileExists(FullFileName) Then

  ' If the document exists, pass its name and location to the
  ' PDFDocument variable
  CustomizedPane.CodeObject.PDFDocument = FullFileName
Else

  ' If the document does not exist, pass the name and path of the
  ' logo document to the PDFDocument variable
  CustomizedPane.CodeObject.PDFDocument = CompanyLogo
End If

End Function
```

If this function does not already exist, it will be created for you and the cursor placed on a blank line within it. If the function does already exist the cursor will be placed at the beginning of it, and you need to create a blank line between the last line of the existing code, and the *END FUNCTION* statement.

The *CustomizedPanes* section of the *Variables* pane contains the names of all of the customized panes associated with this program (any space characters will have been removed from these names). Locate the one for your *PDF Viewer* customized pane. In this case the customized pane was called *Stock Code Spec*, so its name has been shortened to *StockCodeSpec* (see **Figure 13-5**).

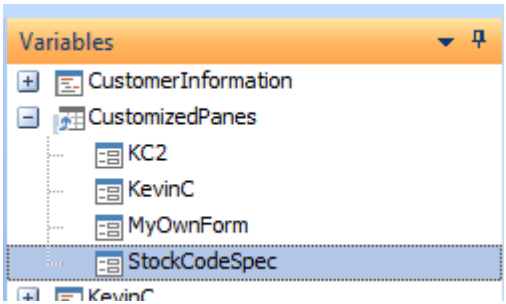


Figure 13-5: The list of customized panes associated with this program

Double-click on the customized pane name and the *Define Action* screen will be displayed (see **Figure 13-6**).

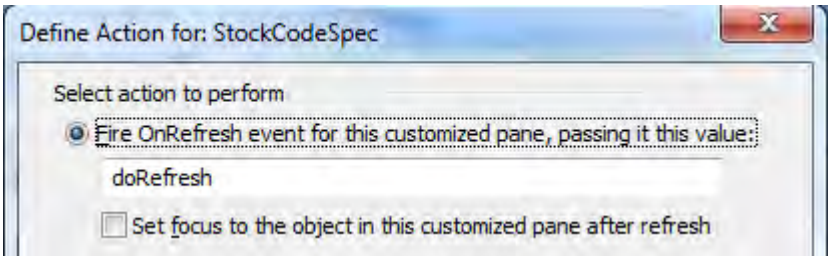


Figure 13-6: The *Define Action* screen that will cause the customized pane to refresh

The default radio button option is the correct one, so click on the *Insert VBScript Code* to insert the code to perform this action. The code that is inserted appears below. When the customized pane's refresh event is fired, this code will pass through the text *doRefresh*. This text needs to be changed to pass through the current stock code.

```
CustomizedPanes.CodeObject.StockCodeSpec = "doRefresh"
```

Highlight the "*doRefresh*" text, locate the *StockCode* variable name within the *StockedLine* section of the *Variables* pane, and double-click on it. This will replace the "*doRefresh*" text with the full name of the *StockCode* variable. The code should now match the code that appears below.

```
CustomizedPanes.CodeObject.StockCodeSpec = StockedLine.CodeObject.StockCode
```

Exit the editor and save the script. Exit the *Sales Order Entry* program. The next time that you call up this program the customized pane will contain your default document (see **Figure 13-7**).

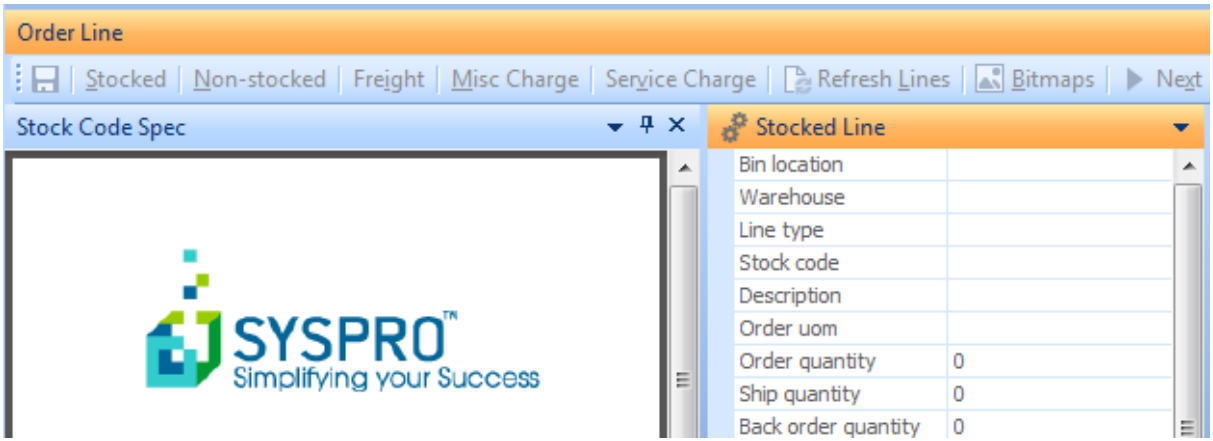


Figure 13-7: The default PDF document being displayed.

Once a stock code has been entered, a check is made to see if a specification exists for this stock code. If it does, the specification is displayed (see **Figure 13-8**). If there is no specification for the entered stock code the default PDF document will be displayed again.

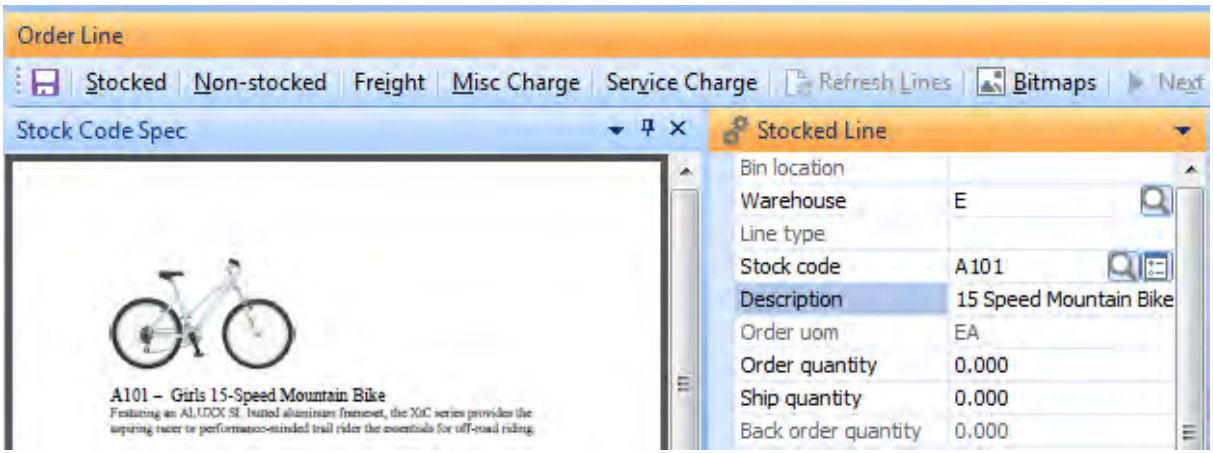


Figure 13-8: The stock code specification being displayed

Object Type: Web Browser

The *Web Browser* object type is used to display the contents of a web page in a customized pane. This can be a default/static web page that is displayed when the customized pane is loaded, in which case no coding is required. The address of the web page is supplied against the initial address field in the *Web Browser* pane of the *Customized Pane Editor* (see **Figure 13-9**).

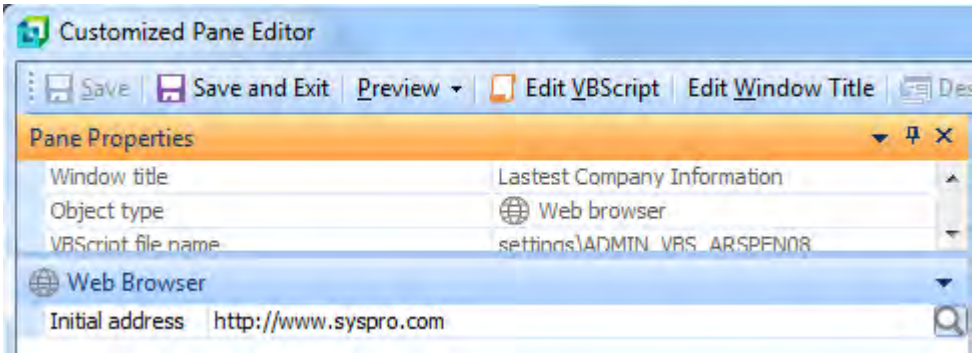


Figure 13-9: Setting the *Window title*, *Object type*, and *Initial address*

The address of the webpage is relative to the client workstation. This can be a page on a share, or point to a page on a web server. **Figure 13-9** shows the *Initial address* points to a web server, but could also have been to an HTML page on a share, such as *\\PaulH\Images\Company.html*.

Hyperlinks inside the web page will work as normal, so if the page has links the operator can navigate using them.

The name of a web page can be passed to the *BrowserAddress* variable within a VBScript function (such as when the customized pane is refreshed) and the new web page will be displayed at that time. For example, this can be used to display a supplier's web page whenever the *Supplier Query* program is used. In this example the supplier's web site is held against the *Trading partner* field (see **Figure 13-10**), which is on the *Fax Details* pane.

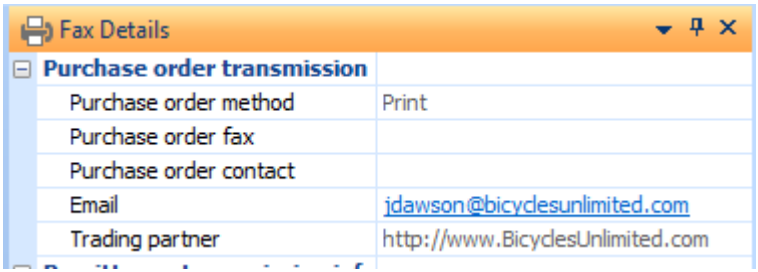


Figure 13-10: The supplier's web site held against the *Trading partner* field

The customized pane should have a blank page for its Initial address (see **Figure 13-11**).

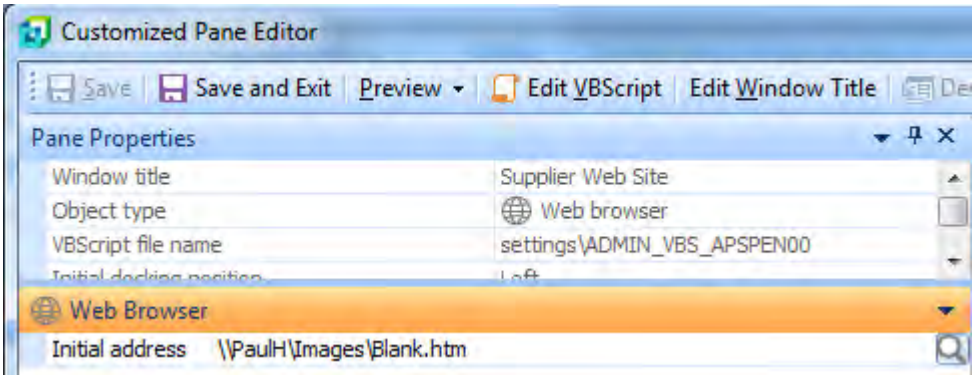


Figure 13-11: Pointing the *Initial address* to an HTML file that does not contain text

Code needs to be added to the *CustomizedPanes_OnRefresh* function to pass the web site address to the *BrowserAddress* variable (if one exists for this supplier) or to pass through a blank page (if one does not exist against this supplier). The code to achieve this appears below.

```
Function CustomizedPane_OnRefresh()
If FaxDetails.CodeObject.TradingPartner <> "" then
  CustomizedPane.CodeObject.BrowserAddress = FaxDetails.CodeObject.TradingPartner
Else
  CustomizedPane.CodeObject.BrowserAddress = "\\PaulH\Images\Blank.htm"
End If
End Function
```

After the customized pane has been saved, code also needs to be added to another pane in this program to cause the customized pane to be refreshed (so that the above code will be run). The code needs to be added to the correct pane so that the correct information is displayed. If the code to refresh the pane were placed against the *Supplier Information* pane, the customized pane would be refreshed before the *Fax Details* pane, so the content of the *Trading partner* field at the time the customized pane was refreshed would still contain the previous supplier's web site. Therefore, the ideal place to add the code to refresh the customized pane is against the *Fax Details* pane's *OnRefresh* function.

The following is the code that should be placed against the *Fax Detail* pane's *OnRefresh* function. The *Define Action* screen was used to build this code (by clicking on the *SupplierWebSite* customized pane name under the *CustomizedPanes* section of the *Variables* pane).

```
Function FaxDetails_OnRefresh()
  CustomizedPanes.CodeObject.SupplierWebSite = "doRefresh"
End Function
```

Once this is saved and a supplier selected in the query, the web site will be displayed if one exists against the *Trading partner* field (see **Figure 13-12**). A blank page will be displayed if one does not exist.

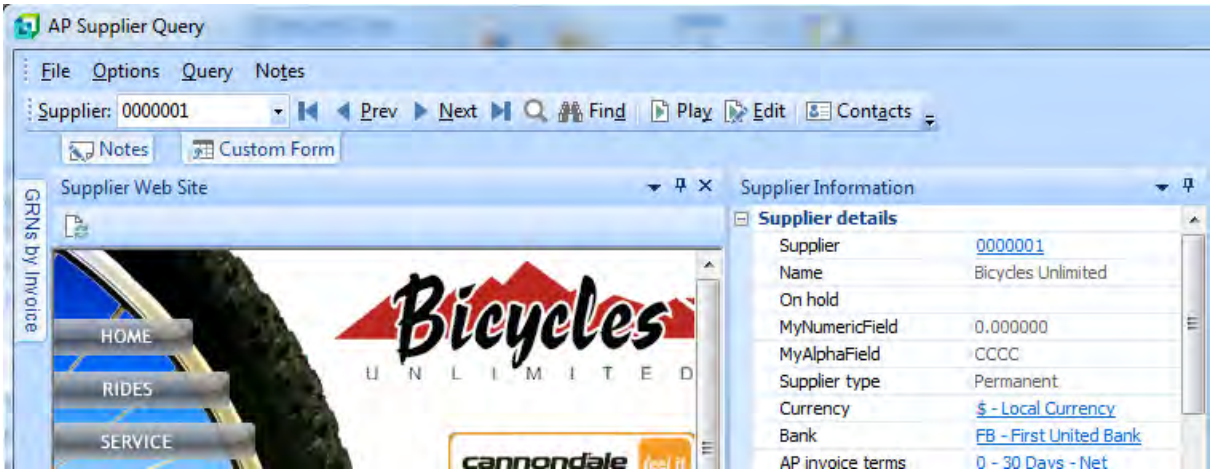


Figure 13-12: Displaying the web site associated with the supplier in a customized pane

You can also construct HTML code for a web page and pass it to the *BrowserSource* variable, and the customized pane will then render this HTML as your web page. This way you can embed information in your page from other panes within this program, or include the latest information that may change frequently.

The following is a basic example of how to build the HTML and pass it to the *BrowserSource* variable. This customized pane is created in the *Inventory Query* program, and displays information about the currently selected stock code in the browser, such as the stock code, description, and an image (see **Figure 13-13**).

The customized pane is created in the normal way, and the *Object type* of *Web browser* is selected. The pane is given a title, and the *Initial address* points to a web page that does not display anything (see **Figure 13-14**).

Using the *Edit VBScript* button, the *VBScript Editor* screen is displayed. The *OnRefresh* event is double-clicked and the *CustomizedPane_OnRefresh* function is created for you.

Figure 13-13: Using the *BrowserSource* variable to display stock code information

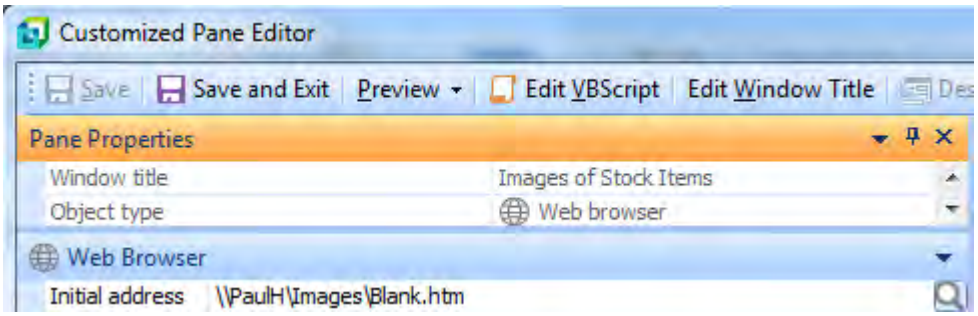


Figure 13-14: Creating a *Web browser* customized pane

Before listing the steps to build the code, you should know what your code is trying to achieve. The line of HTML code below (yes, this is just one line that has wrapped around) is what was passed to

the *BrowserSource* variable to display the information in **Figure 13-13**. The code that you will be building up will supply the stock code and stock description for the page's title, as well as using the stock code when building up the name of the image to be displayed.

```
<HTML><H3>Images of A101 - 15 Speed Mountain Bike Girls</H3><img
src='\\PaulH\Images\A101.jpg' align='left' alt='No image available for this stock
code' ></HTML>
```

The first line of code that is added to this function appears below, and it creates the six variables required to build up the information.

```
Dim Part1, Part2, Part3, SCode, SDesc, CompletedCode
```

The next three lines of code build parts of the HTML string that will be put together around the stock code and description. Note that the spaces (such as the one at the end of the first line between the word *of* and the double quote) are important.

```
Part1 = "<HTML><H3>Images of "
Part2 = "</H3><img src='\\PaulH\Images\"
Part3 = "' align='left' alt='No image available for this stock code' ></HTML>"
```

The next line of code populates the *SCode* variable with the current stock code. This was done to make the code narrower when putting it all together. The full name of the stock code variable could have been used when populating the *CompletedCode* variable, but would have caused the code to wrap around on this page, and consequently make it more difficult to read.

```
SCode = StockCodeDetails.CodeObject.StockCode
```

The next line is similar, and populates the *SDesc* variable with the stock description.

```
SDesc = StockCodeDetails.CodeObject.Description
```

The next line puts all the variables and other text together and populates the *CompletedCode* variable. The hyphen (with the spaces either side of it between the stock code and stock description variables) is only included for presentation purposes. The ".jpg " is appended to the stock code so that the image type is known.

```
CompletedCode = Part1 & SCode & " - " & SDesc & Part2 & SCode & ".jpg " & Part3
```

The final line of code populates the *BrowserSource* variable with the contents of the *CompletedCode* variable. The step of populating the *CompletedCode* variable and using this to populate the *BrowserSource* code could have been bypassed (by building the code against the *BrowserSourc*e variable), but the code was written this way so that it did not wrap around on this page.

```
CustomizedPane.CodeObject.BrowserSource = CompletedCode
```

The completed code appears below.

```
Function CustomizedPane_OnRefresh()
  Dim Part1, Part2, Part3, SCode, SDesc, CompletedCode
  Part1 = "<HTML><H3>Images of "
  Part2 = "</H3><img src='\\PaulH\Images\"
  Part3 = "' align='left' alt='No image available for this stock code' ></HTML>"
  SCode = StockCodeDetails.CodeObject.StockCode
  SDesc = StockCodeDetails.CodeObject.Description
  CompletedCode = Part1 & SCode & " - " & SDesc & Part2 & SCode & ".jpg " & Part3
  CustomizedPane.CodeObject.BrowserSource = CompletedCode
End Function
```

Calling a SYSPRO Program from the Browser

The browser can be used to launch SYSPRO programs. It could therefore be used to build a complete menu system, or to just launch one program that might change depending on current circumstances.

To call a SYSPRO program the HTML code must contain a hash sign (#) followed by the name *SYSPRO*, and the name of the program to call. The example below supplies this information to the *BrowserSource* variable, so that the page and program name can be changed on-the-fly. It displays a hyperlink with the text *Customer Query*, and when you click on the hyperlink the *Customer Query* program is launched.

```
CustomizedPane.CodeObject.BrowserSource = "<a href='#SYSPROARSPEN'>Customer Query"
```

Taking this a step further, the *Customer Query* can be launched and populated with the information for a specific customer. The customer account code is supplied immediately after the program name. The following code calls the *Customer Query* program and passes it the account number 0000001.

```
CustomizedPane.CodeObject.BrowserSource = "<a href='#SYSPROARSPEN0000001'>Customer
Query"
```

The browser can also launch SYSPRO Reporting Services reports by supplying the hash sign, the name *SYSPRO*, the name of the SRS menu program (***SRSMNU***) and the name of the report.

In the example below the *List of Invoice Terms* SRS report (***IMPL09***) is run when the hyperlink is clicked.

```
CustomizedPane.CodeObject.BrowserSource = "<a href='#SYSPROSRSMNUIMPL09'>Invoice
Terms"
```

Object Type: Rich Text Notepad

The *Rich Text Notepad* customized pane enables you to display the contents of an existing RTF file, add a new RTF file, change the content of an existing RTF file, and save the contents of the current pane to an RTF file.

When adding a *Rich Text Notepad* customized pane, the same options are available as other customized panes. However, the area that would contain *Templates* for other customized pane types will contain four options. These options are *Notepad file*, *Read only*, *Auto save on exit*, and *Add save button to toolbar* (see **Figure 13-15**). Unlike with *Templates*, these are still available when maintaining the customized pane.

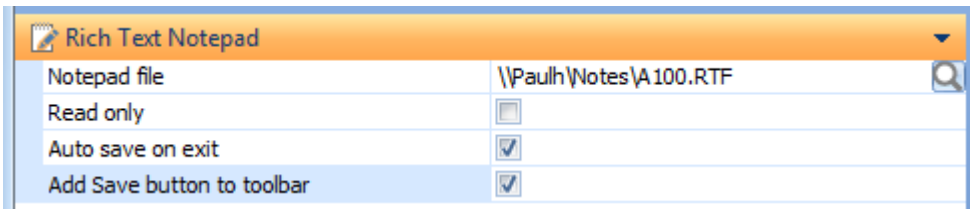


Figure 13-15: The *Rich Text Notepad* specific options

The *Notepad file* option is used to supply the name of an RTF file that is to be displayed when the pane is initially loaded. If no *Notepad file* is supplied, the pane will initially be blank. During the *OnLoad*/*OnRefresh* events, the notepad can be populated with the contents of an RTF file, or text supplied as a string. If the *Read only* option is not checked, the operator can also modify the content manually.

The *Read only* option prevents an operator from modifying the displayed contents of the customized pane. If VBScript code appears against one of the customized pane's macro events, and is configured to display different text, the new text will be displayed when the event fires, as this option only affects the operator input.

The *Auto save on exit* option is used to fire the customized pane's *OnSave* event if the currently displayed rich text has been changed, and the operator exits the program. However, this only fires the *OnSave* event if they exit the program. If, for example, this customized pane resides in the *Inventory Query* program and the operator changes the rich text, then clicks *Next* to go to another stock code, the *OnSave* event is not fired.

The *Add save button to toolbar* option adds a *Save* button to the customized pane's toolbar (see **Figure 13-16**). If the operator clicks on this *Save* button, the customized pane's *OnSave* event is fired.

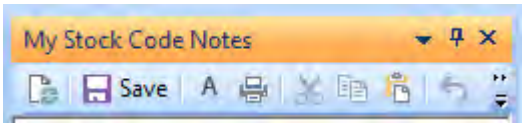


Figure 13-16: The *Save* button on the toolbar

When the *Edit VBScript* button is selected, the *VBScript Editor* screen contains five *Object Events*. These events are *OnLoad*, *OnRefresh*, *OnToolbarButton1Clicked*, *OnToolbarButton2Clicked*, and *OnSave*. The first four of these events work in the same way as all other customized panes. The *OnSave* event is fired either when the operator clicks on the *Save* button on the customized pane's toolbar, or when the operator exists the program (if the *Auto save on exit* option is checked).

There are four variables within the *Variables* pane that are specific *to Rich Text Notepad* customized panes. These are *RTFChanged*, *RTFFileName*, *RTFText*, and *RTFTextFileIn*. There are also two global variables specific to this type of customized pane. **Figure 13-17** shows all six of these variables.

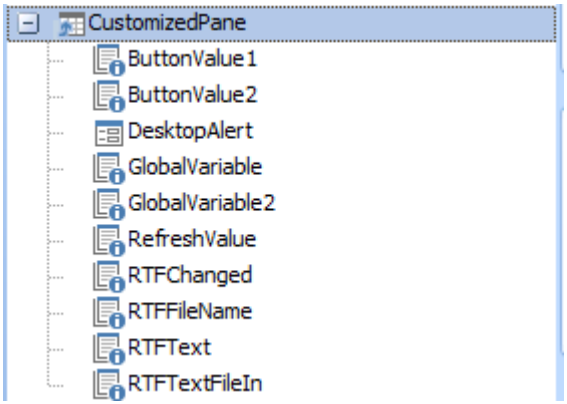


Figure 13-17: The *Rich Text Notepad* specific variables

The *RTFChanged* variable is used to determine if the text against the customized pane has been changed since it was first displayed. If the text has been changed by the operator this variable will contain a 1, and if it has not changed it will contain a 0. This can be used within the *OnRefresh* and *OnSave* functions to check if the information that was previously present had been changed, and then write it away if it had. An example of how to do this appears later in this chapter.

When the *OnRefresh* or *OnSave* events fire, a temporary rich text file becomes available that contains the text as it was in the pane before the event was fired. Within these functions the variable *RTFFileTextIn* variable contains the name of this (local) temporary file. This means that during the *OnRefresh* (or *OnSave*) function you can either read this information back from this temporary file and

write it away to the correct file, or copy this file to the correct filename. Note: this variable contains the temporary filename, not the rich text itself. The default temporary filename is *RTFTextFileIn*, and the default location is the SYSPRO `Base\Settings` folder on the client workstation. But rather interrogate this variable to find the correct name and location. An example of how to use this variable to save away the text appears later in this chapter.

The content of an existing RTF file can be displayed in the customized pane by passing its name and location to the *RTFFileName* variable. Typically this would be passed to the variable in the customized pane's *OnRefresh* function. For example, if you have an RTF file for each stock code, and your rich text notepad customized pane resides in the *Inventory Query*, the pane's *OnRefresh* function could contain a line of code that passes the stock code's RTF file to the variable as per the code below. This code builds up the name and the location of RTF file, and passes it through to the *RTFFileName* variable.

```
Function CustomizedPane_OnRefresh()
  Dim MyFileLocation, MyNewFileName
  MyFileLocation = "\\Paulh\Notes\"
  MyNewFileName = MyFileLocation & CustomizedPane.CodeObject.RefreshValue & ".RTF"
  CustomizedPane.CodeObject.RTFFileName = MyNewFileName
End Function
```

You can also populate the rich text notepad with text programmatically by passing it to the *RTFText* variable. This can be just a string of text, or could be text retrieved from a SYSPRO Notepad RTF file using the ***COMNOT*** business object, and passed to the variable. An example of this appears later in this chapter.

Examples of Using the Rich Text Notepad Variables

The following are examples of using the *Rich Text Notepad* specific variables in customized panes. Example 1 uses the *RTFChanged*, *GlobalVariable*, *RTFTextFileIn*, and *RTFFilename* variables to retrieve and update RTF files associated with stock codes. This uses a Windows share as a hard-coded location.

Example 2 uses the *RTFChanged*, *GlobalVariable*, *RTFTextFileIn*, and *RTFText* variables in conjunction with the ***COMNOT*** and ***COMSNO*** business objects to do the same. This uses SYSPRO's architecture and configuration to handle the storage location.

Rich Text Notepad Example 1

The first example is very basic, and is just to show how the *RTFChanged*, *GlobalVariable*, *RTFTextFileIn*, and *RTFFilename* variables work. This customized pane was designed to sit within the *Inventory Query* program. It expects to receive a stock code in the *RefreshValue* variable whenever it is refreshed, and it displays the contents of an RTF document based on the supplied stock code. As it does not directly communicate with any other form within the *Inventory Query*, it can reside within any program, provided that it receives the stock code when it is refreshed.

The customized pane uses a Windows share to hold RTF files for each stock code. When the operator selects a stock code, another pane within the program forces the customized pane to refresh, and passes it the current stock code in the *RefreshValue* variable. The customized pane builds up the name of the RTF file, and causes it to be displayed using the *RTFFileName* variable.

When the operator changes the stock code (by manually typing it in, or using the *Next* or *Previous* buttons) a check is made using the *RTFChanged* variable to see if the operator has modified the displayed content. If they have, this content is stored in a temporary file, the name of which is kept in the *RTFTextFileIn* variable. This temporary file is copied over the existing RTF file for this stock code before the content of the new stock code's RTF file is displayed.

When the operator exits the program the *OnSave* event is fired. The same logic exists in the *OnSave* function to detect if the operator has modified the displayed content. If they have, the content is written away to the RTF file before the program closes.

At the time the operator selects the new stock code and the customized pane is refreshed, the old stock code is no longer available. So right at the end of the *OnRefresh* function the stock code is written to the *GlobalVariable* variable. Unlike other variables that only exist for the time that the function is being executed, the global variables associated with this customized pane (*GlobalVariable* and *GlobalVariable2*) exist until the program is closed. By adding the stock code to this global variable at the end of the *OnRefresh* function, the stock code is still available to be used even though the customized pane is being refreshed with the new stock code.

The following is the code that appears against the *OnRefresh* function:

```
Dim MyFileLocation, MyOldFileName, MyNewFileName, fs
MyFileLocation = "\\Paulh\Notes\"
MyNewFileName = MyFileLocation & CustomizedPane.CodeObject.RefreshValue & ".RTF"
MyOldFileName = MyFileLocation & CustomizedPane.CodeObject.GlobalVariable & ".RTF"

If CustomizedPane.CodeObject.RTFChanged = 1 then
  Set fs = CreateObject("Scripting.FileSystemObject")
  fs.CopyFile CustomizedPane.CodeObject.RTFTextFileIn,MyOldFileName
End If
CustomizedPane.CodeObject.GlobalVariable = CustomizedPane.CodeObject.RefreshValue
CustomizedPane.CodeObject.RTFFileName = MyNewFileName
```

The first line creates the four variables that will be used in this function.

The second line populates the *MyFileLocation* variable with the name of the share that contains the RTF files.

The third line builds up the name of the document for the new stock code using the *MyFileLocation* variable and the *RefreshValue* that contains the stock code. This is passed to the *MyNewFileName* variable.

The fourth line builds up the name of the document for the previous stock code using the *MyFileLocation* variable and the *GlobalVariable* that contains the old stock code. This is passed to the *MyOldFileName* variable.

The fifth line checks the contents of the *RTFChanged* variable. This will contain 0 if the displayed RTF file has not been changed and 1 if it has been changed.

Lines six and seven are only run if the content has changed. Line six instantiates the *FileSystemObject*. Line seven uses the *RTFTextFileIn* variable to find the temporary file containing the updated text, then copies this over the existing RTF file for this stock code.

Line eight closes off the *IF* statement

Line nine populates the variable *GlobalVariable* with the current stock code. The global variables against a customized pane remain available for the time that the program is open. Storing the stock code in this variable means that it will still be available when the *OnRefresh* function is used again, and the previous stock code is still available for use in line four above.

Line ten takes the built up filename and location of the RTF for the new stock code and passes it to the *RTFFileName* variable, which causes its contents to be displayed in the customized pane.

The following is the code that appears against the *OnSave* function. The *OnSave* event fires when the operator exits the program:

```
Dim MyFileLocation, MyOldFileName, MyNewFileName, fs
MyFileLocation = "\\Paulh\Notes\"
MyOldFileName = MyFileLocation & CustomizedPane.CodeObject.GlobalVariable & ".RTF"

If CustomizedPane.CodeObject.RTFChanged = 1 then
  Set fs = CreateObject("Scripting.FileSystemObject")
  fs.CopyFile CustomizedPane.CodeObject.RTFTextFileIn,MyOldFileName
End If
```

If you compare the contents of the *OnSav*e function to the *OnRefresh* function it can be seen that the *OnSave* function is a subset of the *OnRefresh* function. The differences are that the *OnSave* function does not need to store the new stock code, and it does not need to build up the new stock code filename and location and cause it to be displayed.

For completeness, the following is the code that was added to the *Stock Code Details* pane's *OnRefresh* function in the *Inventory Query* program. This assumes that the customized pane is called *Tech Specs*.

```
CustomizedPanes.CodeObject.TechSpecs = StockCodeDetails.CodeObject.StockCode
```

Rich Text Notepad Example 2

Example 2 is a much more sophisticated version of Example 1. It uses the ***COMNOT*** business object to retrieve the text from the RTF file associated with this stock code, and the ***COMSNO*** business object to write out the text to the RTF file. Rather than describe how it works here, the explanation will appear alongside the individual sections of code. Remember that any line of code that starts with an apostrophe is a comment line and does not need to be there. It is only present to make the code easier to follow. Also, any indenting at the beginning of the line is only present to make the code easier to follow.

This customized pane consists of three functions, *OnLoad*, *OnRefresh*, and *OnSave*. The *OnLoad* function contains one statement to force the customized pane's *GlobalVariable2* to be empty when the pane is loaded. The code for this appears below:

```
CustomizedPane.CodeObject.GlobalVariable2 = ""
```

The complete *OnRefresh* function appears below. Note that within the first section that builds XML (to be passed to the ***COMSNO*** business object) there are three lines that have wrapped around on the page. And in the second section that builds XML (to be passed to the ***COMNOT*** business object) one line has wrapped around. As each section of code is explained, the lines that have wrapped around will be highlighted.

```
' Check if the contents of the notepad has changed since
' it was displayed, to work out if it must be saved.
If CustomizedPane.CodeObject.RTFChanged = "1" then
  dim XMLOut, XMLParam, XMLOut2, XMLParam2, XMLDoc2
  dim s2, fn2, MyContents2, MyContents3, Mylength2

  ' Customized Pane Global Variable at this point contains
  ' the previous stock code. Check if it is empty. If it is
  ' not empty the currently displayed notes must be written
  ' away before displaying those for the new stock code.
  If CustomizedPane.CodeObject.GlobalVariable <> "" then

    ' The current contents of the notepad are written to a
    ' temporary file on the client. The name of this temporary
    ' file is held in the varaiable RTFTextFileIn. Open this
    ' file and read in the contents to a variable called
    ' MyContents. Remove any Null characters that may appear
    ' in the temporary file. Remove the trailing carriage return
    ' and line feed characters.
```

```
    Set fs2 = CreateObject("Scripting.FileSystemObject")
    Set fn2 = fs2.OpenTextFile(CustomizedPane.CodeObject.RTFTextFileIn)
    MyContents2 = Replace(fn2.ReadAll, vbNullChar, "")
    fn2.close
    Set fn2 = nothing
    Set fs2 = nothing
    MyLength2 = Len(MyContents2)
    MyContents3 = Left(MyContents2, Mylength2 -2)

    ' Build up the XML required to write the contents of the
    ' notepad away. At this point the Customized Pane Global
    ' Variable still contains the previous stock code.
    XMLParam2 = XMLParam2 & " <SetupNotepad/>"

    XMLDoc2 = XMLDoc2 & "<SetupNotepad>"
    XMLDoc2 = XMLDoc2 & " <Item>"
    XMLDoc2 = XMLDoc2 & "  <Key>"
    XMLDoc2 = XMLDoc2 & "   <Company><![CDATA[" &
SystemVariables.CodeObject.Company & "]]></Company>"
    XMLDoc2 = XMLDoc2 & "   <NoteType><![CDATA[STK]]></NoteType>"
    XMLDoc2 = XMLDoc2 & "   <KeyField><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyField>"
    XMLDoc2 = XMLDoc2 & "  </Key>"
    XMLDoc2 = XMLDoc2 & "  <NotepadText><![CDATA[" & MyContents3 &
"]]></NotepadText>"
    XMLDoc2 = XMLDoc2 & " </Item>"
    XMLDoc2 = XMLDoc2 & "</SetupNotepad>"

    on error resume next

    ' Check to see if the notes were read in from COMNOT, or
    ' created from scratch.
    If CustomizedPane.CodeObject.GlobalVariable2 = "Y" then

      ' If the notes were read in from COMNOT, call the COMSNO
      ' business object and use the UPDATE method
      XMLOut2 = CallSetup("COMSNO",XMLParam2,XMLDoc2,"Update","auto")
    Else
      ' If the notes were not read in from COMNOT, call the
      ' COMSNO business object and use the ADD method
      XMLOut2 = CallSetup("COMSNO",XMLParam2,XMLDoc2,"Add","auto")
    End If

    If err then
      exit function
    End if
    ' Switch on error handling
    on error goto 0

  End If
End If
```

```
' This is the end of the section to save the notes related for
' the old stock code.

' This is the start of the section to read in the notes for the
' new stock code. Populate the Customized Pane Global Variable
' with the refresh value (the new stock code).
CustomizedPane.CodeObject.GlobalVariable = CustomizedPane.CodeObject.RefreshValue

' Check to see if the Customized Pane Global Variable is empty,
' and exit the function if it is.
If CustomizedPane.CodeObject.GlobalVariable = "" then
  exit function
End if

' Build up the XML required to invoke the COMNOT business object
' to see if notes exist for this stock code. Note that the
' Globalvariable is used as the stock code has already been
' updated.
XMLParam = XMLParam & "<Query>"
XMLParam = XMLParam & " <Key>"
XMLParam = XMLParam & "  <NoteType><![CDATA[STK]]></NoteType>"
XMLParam = XMLParam & "  <KeyFieldValue><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyFieldValue>"
XMLParam = XMLParam & " </Key>"
XMLParam = XMLParam & " <ReturnRtf>Y</ReturnRtf>"
XMLParam = XMLParam & "</Query>"
on error resume next

' Call the COMNOT business object, and put the results into XMLOut
XMLOut = CallBO("COMNOT",XMLParam,"auto")

' Switch on error handling
on error goto 0

' Check to see if XMLOut is empty.
If XMLOut = "" then
  ' If XMLOut is empty, set the Customized Pane GlobalVariable2
  ' to N. This is required when the notes are saved as you need
  ' to ADD them. The RTFText variable is cleared out so that the
  ' previous stock code notes do not appear.
  CustomizedPane.CodeObject.GlobalVariable2 = "N"
  CustomizedPane.CodeObject.RTFText = " "
Else
  ' If XMLOut is not empty, set the Customized Pane
  ' GlobalVariable 2 to Y. This is required when the notes are
  ' saved as you need to UPDATE them. The RTFText variable is
  ' populated with the contents of XMLOut so that the existing
  ' notes appear.
  CustomizedPane.CodeObject.GlobalVariable2 = "Y"
  CustomizedPane.CodeObject.RTFText = XMLOut
End If
```

The first section of code against the *OnRefresh* section appears below. It performs a check to see if the *RTFChanged* variable contains a 1 (this variable will contain a 1 if the content has changed, and a 0 if it has not). It then creates variables.

```
If CustomizedPane.CodeObject.RTFChanged = "1" then
  dim XMLOut, XMLParam, XMLOut2, XMLParam2, XMLDoc2
  dim s2, fn2, MyContents2, MyContents3, Mylength2
```

The next line of code checks to see if the *GlobalVariable* is empty. If the *GlobalVariable* contains a value it means that notes have already been displayed for another stock code during this run of the program, and these notes need to be saved away.

```
If CustomizedPane.CodeObject.GlobalVariable <> "" then
```

This is followed by eight lines of code that read in the changed text, and strip out the unwanted characters. The first two lines instantiate the *FileSystemObject* and read in the contents of the temporary file to a variable called *fn2*. This is followed by a line that removes any *NULL* characters and populates the *MyContents2* variable. The next three lines of code close the file that was opened to read in the text, and performs a little housekeeping.

The last two lines of code work out the length of the text and strip of the last two characters. When all of this is complete the *MyContents3* variable contains just the text.

```
Set fs2 = CreateObject("Scripting.FileSystemObject")
Set fn2 = fs2.OpenTextFile(CustomizedPane.CodeObject.RTFTextFileIn)
MyContents2 = Replace(fn2.ReadAll, vbNullChar, "")
fn2.close
Set fn2 = nothing
Set fs2 = nothing
MyLength2 = Len(MyContents2)
MyContents3 = Left(MyContents2, Mylength2 -2)
```

Business objects belonging to the *Setup* class required two XML strings/documents when they are invoked. One is referred to as the *Document* string (which contains the XML that must be processed), and the other is referred to as the *Parameter* string (and contains instruction on how the business object should process the *Document* string).

Shortly you will be calling the ***COMSNO*** business object to write out the text to the RTF file. The ***COMSNO*** business object is a member of the *Setup* class, so you need to build up both the *Parameter* and *Document* XML strings. The next line of code builds up the *Parameter* string in a variable called *XMLParam2*.

```
XMLParam2 = XMLParam2 & " <SetupNotepad/>"
```

This is followed by ten lines of code that build up the *Document* XML string, and all of these lines start with *XMLDoc2*. In the code below, two of these lines have wrapped around. Each of these lines adds to the previous one so at the end of this section you will have one XML string in the *XMLDoc2* variable.

When ***COMSNO*** is invoked, it needs to contain four pieces of information inside the XML. The first is the ID of the current company, which is available from a system variable called *Company*. There are several different RTF note types available to be used in SYSPRO, and ***COMSNO*** needs to know which one to use. Each of these has a standard three or six character code associated with it. The note type associated with stock codes is *STK*, and this has been hard-coded in this XML. The third piece of information is the stock code to which these notes belong. As the stock code has already changed to the new one, the old one stored in the *GlobalVariable* variable must be used. The fourth piece of information is the text that must be used. This has already been placed in the *MyContents3* variable.

These four pieces of information in the XML have been wrapped in *CDATA* sections. This means that the content must be used as it is, and is not processed by the parser.

```
XMLDoc2 = XMLDoc2 & "<SetupNotepad>"
XMLDoc2 = XMLDoc2 & " <Item>"
XMLDoc2 = XMLDoc2 & "  <Key>"
XMLDoc2 = XMLDoc2 & "   <Company><![CDATA[" & SystemVariables.CodeObject.Company &
"]]></Company>"
XMLDoc2 = XMLDoc2 & "   <NoteType><![CDATA[STK]]></NoteType>"
XMLDoc2 = XMLDoc2 & "   <KeyField><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyField>"
XMLDoc2 = XMLDoc2 & "  </Key>"
XMLDoc2 = XMLDoc2 & "  <NotepadText><![CDATA[" & MyContents3 & "]]></NotepadText>"
XMLDoc2 = XMLDoc2 & " </Item>"
XMLDoc2 = XMLDoc2 & "</SetupNotepad>"
```

This is followed by a line that tells the script to keep processing, even if an error is detected.

```
on error resume next
```

The next five lines determine if the displayed notes were read in using the ***COMNOT*** business object, or supplied by the operator. The ***COMSNO*** business object is then called using different methods. If the notes were read in using the ***COMNOT*** business object, the variable *GlobalVariable2* will contain the letter Y. If not, the variable will contain the letter N. The code to set this variable appears later in the *OnRefresh* function.

The *Setup* class business object ***COMSNO*** can be called using one of three methods, *Add*, *Update*, and *Delete*. The correct method must be used when calling the business object otherwise an error will be returned. If the notes came from ***COMNOT*** they already exist, so ***COMSNO*** must be called with the *Update* method. If they did not come from ***COMNOT***, then ***COMSNO*** must be called with the *Add*

method. In both cases the business object is called and the *Parameter* and *Document* XML is passed to the business object and the results are put into the *XMLOut2* variable.

```
If CustomizedPane.CodeObject.GlobalVariable2 = "Y" then
  XMLOut2 = CallSetup("COMSNO",XMLParam2,XMLDoc2,"Update","auto")
Else
  XMLOut2 = CallSetup("COMSNO",XMLParam2,XMLDoc2,"Add","auto")
End If
```

The next lines of code perform a check to see if an error was returned when the business object was called, and if it was, the script should exit the *OnRefresh* function.

```
If err then
  exit function
End if
On error goto 0
```

This is followed by two lines containing *END IF* statements. The first ends the *IF* statement that checked to see if *GlobalVariable* was blank. The second ends the *IF* statement from right at the beginning of the *OnRefresh* function that checked to see if the *RTFChanged* variable contained the number 1.

```
End If
End If
```

The next line of code takes the new stock code that was passed to the customized pane when its *OnRefresh* event was fired, and stores it in *GlobalVariable*, so that there is a record of the previous stock code when *OnRefresh* is fired next time.

```
CustomizedPane.CodeObject.GlobalVariable = CustomizedPane.CodeObject.RefreshValue
```

A check is made to see if a value was passed through when the *OnRefresh* event fired. If this is blank it means that the *OnRefresh* event wasn't fired by another pane and there is no stock code, so there is no point in continuing with this function (as this may cause an error when the ***COMNOT*** business object is called).

```
If CustomizedPane.CodeObject.GlobalVariable = "" then
  Exit function
End if
```

The ***COMNOT*** business object belongs to the *Query* class, and is invoked using the *Query* method. It is used to retrieve the content of an RTF file associated with this key for this note type. *Query* class business objects only accept one XML string/document, (referred to as the *Document* string). The next seven lines of code (one has wrapped around on the page) build up the XML to be passed to the ***COMNOT*** business object. The three pieces of information that are passed to this business object are

the note type of *STK* (because this customized pane is for stock codes), the contents of *GlobalVariable* (which contains the previous stock code), and whether the results must be rich text, or plain text.

```
XMLParam = XMLParam & " <Query>"
XMLParam = XMLParam & "   <Key>"
XMLParam = XMLParam & "     <NoteType><![CDATA[STK]]></NoteType>"
XMLParam = XMLParam & "     <KeyFieldValue><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyFieldValue>"
XMLParam = XMLParam & "   </Key>"
XMLParam = XMLParam & "   <ReturnRtf>Y</ReturnRtf>"
XMLParam = XMLParam & " </Query>"
```

This is followed by the same *On Error Resume Next* statement, and the statement that calls the business object and passes it the built up XML. The results are returned to the *XMLOut* variable

```
On error resume next
XMLOut = CallBO("COMNOT",XMLParam,"auto")
```

The final piece of code within the *OnRefresh* function performs a check to see if anything was returned by the ***COMNOT*** business object. If nothing was returned, *GlobalVariable2* is set to N, and the *RTFText* variable is populated with a space. Setting *GlobalVariable2* to N specifies that when the *OnRefresh* event is fired again, the code knows that the displayed text did not come from ***COMNOT***, and if there is text to be saved, ***COMSNO*** must be called using the *Add* method.

If the ***COMNOT*** business object did return a result, *GlobalVariable2* is set to Y, and if text needs to be saved the code knows to call the ***COMSNO*** business object using the *Update* method. The *RTFText* variable is populated with the output from the business object, and this is what gets displayed in the customized pane.

```
On error goto 0
If XMLOut = "" then
  CustomizedPane.CodeObject.GlobalVariable2 = "N"
  CustomizedPane.CodeObject.RTFText = " "
Else
  CustomizedPane.CodeObject.GlobalVariable2 = "Y"
  CustomizedPane.CodeObject.RTFText = XMLOut
End If
```

The complete *OnSave* function appears below. Note that within the section that builds XML (to be passed to the **COMSNO** business object) there are two lines that have wrapped around on the page. As each section of code is explained, the lines that have wrapped around will be highlighted :

```
' Check if the contents of the notepad has changed since it was
' displayed, to work out if it must be saved.
If CustomizedPane.CodeObject.RTFChanged = "1" then

  ' Create variables for use in saving away the existing notes
  dim XMLOut, XMLParam, XMLDoc, fs, fn, MyContents, MyContents4, MyLength

  ' The current contents of the notepad are written to a temporary
  ' file on the client. The name of this temporary file is held in
  ' the variable RTFTextFileIn. Open this file and read in the
  ' contents to a variable called MyContents. Strip out any NULL
  ' characters, and the the trailing carriage return and line feed
  ' characters.
  Set fs = CreateObject("Scripting.FileSystemObject")
  Set fn = fs.OpenTextFile(CustomizedPane.CodeObject.RTFTextFileIn)
  MyContents = Replace(fn.ReadAll, vbNullChar, "")
  fn.close
  Set fn = nothing
  Set fs = nothing
  MyLength = Len(MyContents)
  MyContents4 = Left(MyContents, Mylength -2)

  ' Build up the XML required to write the contents of the notepad away.
  XMLParam = XMLParam & " <SetupNotepad/>"

  XMLDoc = XMLDoc & "<SetupNotepad>"
  XMLDoc = XMLDoc & " <Item>"
  XMLDoc = XMLDoc & "  <Key>"
  XMLDoc = XMLDoc & "   <Company><![CDATA[" & SystemVariables.CodeObject.Company &
"]]></Company>"
  XMLDoc = XMLDoc & "   <NoteType><![CDATA[STK]]></NoteType>"
  XMLDoc = XMLDoc & "   <KeyField><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyField>"
  XMLDoc = XMLDoc & "  </Key>"
  XMLDoc = XMLDoc & "  <NotepadText><![CDATA[" & MyContents4 & "]]></NotepadText>"
  XMLDoc = XMLDoc & " </Item>"
  XMLDoc = XMLDoc & "</SetupNotepad>"
  on error resume next

  ' Check to see if the notes were read in from COMNOT, or created
  ' from scratch.
  If CustomizedPane.CodeObject.GlobalVariable2 = "Y" then
    ' If the notes were read in from COMNOT, call the COMSNO business
    '  object and use the UPDATE method.
    XMLOut = CallSetup("COMSNO",XMLParam,XMLDoc,"Update","auto")
  Else
    ' If the notes were not read in from COMNOT, call the COMSNO
    ' business object and use the ADD method
    XMLOut = CallSetup("COMSNO",XMLParam,XMLDoc,"Add","auto")
  End If
```

```
  If err then
    Exit function
  End if

  ' Switch on error handling
  on error goto 0
 End If
```

The first section of code against the *OnSave* function appears below. It performs a check to see if the *RTFChanged* variable contains a 1 (this variable will contain a 1 if the content has changed, and a 0 if it has not). It then creates eight variables.

```
If CustomizedPane.CodeObject.RTFChanged = "1" then
dim XMLOut, XMLParam, XMLDoc, fs, fn, MyContents, MyContents4, MyLength
```

This is followed by eight lines of code to read in the changed text and strip out unwanted characters. The first two lines instantiate the *FileSystemObject* and read in the contents of the temporary file to a variable called *fn*. This is followed by a line that removes any *NULL* characters and populates the *MyContents* variable. The next three lines of code close the file that was opened to read the text, and performs a little housekeeping.

The last two lines of code work out the length of the text and strip of the last two special characters. When all of this is complete the *MyContents4* variable contains just the text.

```
Set fs = CreateObject("Scripting.FileSystemObject")
Set fn = fs.OpenTextFile(CustomizedPane.CodeObject.RTFTextFileIn)
MyContents = Replace(fn.ReadAll, vbNullChar, "")
fn.close
Set fn = nothing
Set fs = nothing
MyLength = Len(MyContents)
MyContents4 = Left(MyContents, Mylength -2)
```

Business objects belonging to the *Setup* class require two XML strings/documents when they are invoked. One is referred to as the *Document* string (which contains the XML that must be processed), and the other is referred to as the *Parameter* string (and contains instruction on how the business object should process the *Document* string).

Shortly you will be calling the ***COMSNO*** business object to write out the text to the RTF file. The ***COMSNO*** business object is a member of the *Setup* class, so you need to build up both the *Parameter* and *Document* XML strings. The next line of code builds up the *Parameter* string in a variable called *XMLParam*. In this case the parameter XML only contains the root element.

```
XMLParam = XMLParam & " <SetupNotepad/>"
```

This is followed by ten lines of code that build up the *Document* XML string, and all of these lines start with *XMLDoc* (in the code below, two of these lines have wrapped around). Each of these lines adds to the previous one so at the end of this section you will have one XML string in the *XMLDoc* variable.

When ***COMSNO*** is invoked it needs to contain four pieces of information inside the XML. The first is the ID of the current company (which is available from a *SystemVariable* called *Company*). There are several different RTF note types available to be used in SYSPRO, and ***COMSNO*** needs to be told which one to use. Each of these has a standard three or six character code associated with it. The note type associated with stock codes is *STK*, and this has been hard-coded in this XML. The third piece of information is the stock code to which the text belongs. As the stock code has already changed to the new one, the old one stored in the *GlobalVariabl*e variable must be used. The fourth piece of information is the text that must be used. This has already been places in the *MyContents4* variable.

These four pieces of information in the XML have been wrapped in *CDATA* sections. This means that the content must be used “as is”, and is not processed by the parser.

```
XMLDoc = XMLDoc & "<SetupNotepad>"
XMLDoc = XMLDoc & " <Item>"
XMLDoc = XMLDoc & "  <Key>"
XMLDoc = XMLDoc & "   <Company><![CDATA[" & SystemVariables.CodeObject.Company &
"]]></Company>"
XMLDoc = XMLDoc & "   <NoteType><![CDATA[STK]]></NoteType>"
XMLDoc = XMLDoc & "   <KeyField><![CDATA[" &
CustomizedPane.CodeObject.GlobalVariable & "]]></KeyField>"
XMLDoc = XMLDoc & "  </Key>"
XMLDoc = XMLDoc & "  <NotepadText><![CDATA[" & MyContents4 & "]]></NotepadText>"
XMLDoc = XMLDoc & " </Item>"
XMLDoc = XMLDoc & "</SetupNotepad>"
```

The next five lines determine if the displayed notes were read in using the ***COMNOT*** business object, or supplied by the operator. The ***COMSNO*** business object is then called using different methods. If the notes were read in using the ***COMNOT*** business object the variable *GlobalVariable2* will contain the letter Y. If not, the variable will contain the letter N. The code that sets this variable appears in the *OnRefresh* function.

The *Setup* class business object ***COMSNO*** can be called using one of three methods, *Add*, *Update*, or *Delete*. The correct method must be used when calling the business object otherwise an error will be returned. If the notes came from ***COMNOT*** they already exist, so ***COMSNO*** must be called using the *Update* method. If they did not come from ***COMNOT***, then ***COMSNO*** must be called using the *Add* method. In both cases the business object is called, and the *Parameter* and *Document* XML is passed to the business object and the results are put into the *XMLOut* variable.

```
If CustomizedPane.CodeObject.GlobalVariable2 = "Y" then
  XMLOut = CallSetup("COMSNO",XMLParam,XMLDoc,"Update","auto")
Else
  XMLOut = CallSetup("COMSNO",XMLParam,XMLDoc,"Add","auto")
End If
```

The next lines of code perform a check to see if an error was return when the business object was called, and if it was, the script should exit the *OnSave* function.

```
If err then
  exit function
End if
On error goto 0
```

The last line of code within the *OnSave* function is *END IF*, and this ends the *IF* statement from right at the beginning of the *OnSave* function that checked to see if the *RTFChanged* variable contained the number 1.

```
End If
```

Although all the code used in the customized pane has been covered above, there still needs to be an entry against the *Stock Code Details* form's *OnRefresh* function that causes the customized pane's *OnRefresh* function to fire. This must also pass through the stock code as the refresh value. The following is the code to perform this, assuming that the customized pane is called *My Stock Code Notes*.

```
Function StockCodeDetails_OnRefresh()
CustomizedPanes.CodeObject.MyStockCodeNotes = StockCodeDetails.CodeObject.StockCode
End Function
```

Object Type: SRS/Crystal Report

The *SRS*/*Crystal Report* object type is used to display the contents of an SRS report, or a Crystal Report, in a customized pane. This report must be the output of a report (a `.rpt` file), not a report that requires executing.

When adding the customized pane and selecting the *Object type* of *SRS*/*Crystal Report* the area at the bottom of the screen (that contains templates for some object types) will contain the *Report file name* prompt. This must contain the name and path of the report to be displayed when the pane initially loads (see **Figure 13-18**). It is mandatory to populate this prompt, as you cannot save the customized pane if this is not present.

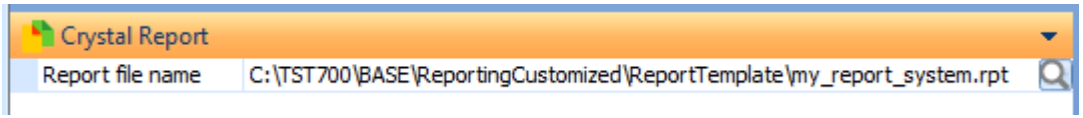


Figure 13-18: The *Report file name* prompt contains the name of the initial report to be displayed

If this is the only report to be displayed, you just need to supply a *Title* for the pane and save it.

Note that both the SRS customized pane and the report viewer have their own toolbars, so the normal customized pane toolbar is not displayed. This means that the *Toolbar control 1* and *Toolbar control 2* controls are not available (so neither are their macro events). The customized pane toolbar consists of a *Print* button and an *Email* button (see **Figure 13-19**).

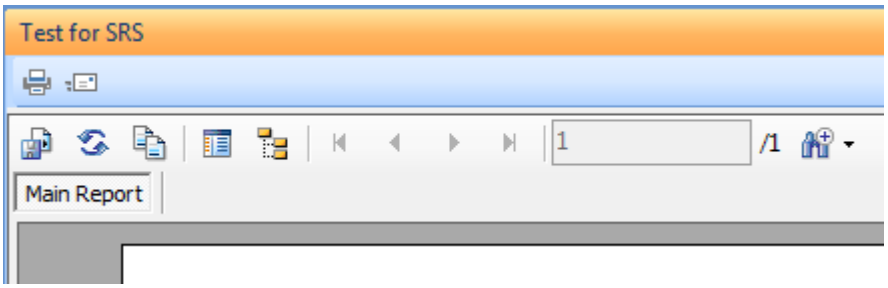


Figure 13-19: The customized pane toolbar and the report viewer toolbar

The *Email* button displays a screen where you can enter the recipient's email address, subject line, text, and select from five different formats for the report (see **Figure 13-20**). The email is sent with an attachment of the relevant format.

The report viewer toolbar contains the *Export Report*, *Refresh*, *Copy*, *Toggle Parameter Panel*, *Toggle Group Tree*, and *Zoom* buttons. The *Export Report* button saves a copy of the `.rpt` file to the specified location. The *Refresh* button re-reads the currently displayed `.rpt` file and redisplays it. The *Copy* button copies the currently selected report text to the clipboard so that it can be pasted into another application. The *Toggle Parameter Panel* and *Toggle Group Tree* options add/remove the relevant panels to the left of the report. The *Zoom* button enables you to zoom in/out by selecting a percentage.

If the report is longer than one page the *Go to First Page*, *Go to Previous Page*, *Go to Next Page*, and *Go to Last Page* buttons will also be enabled so that you can navigate around the report.

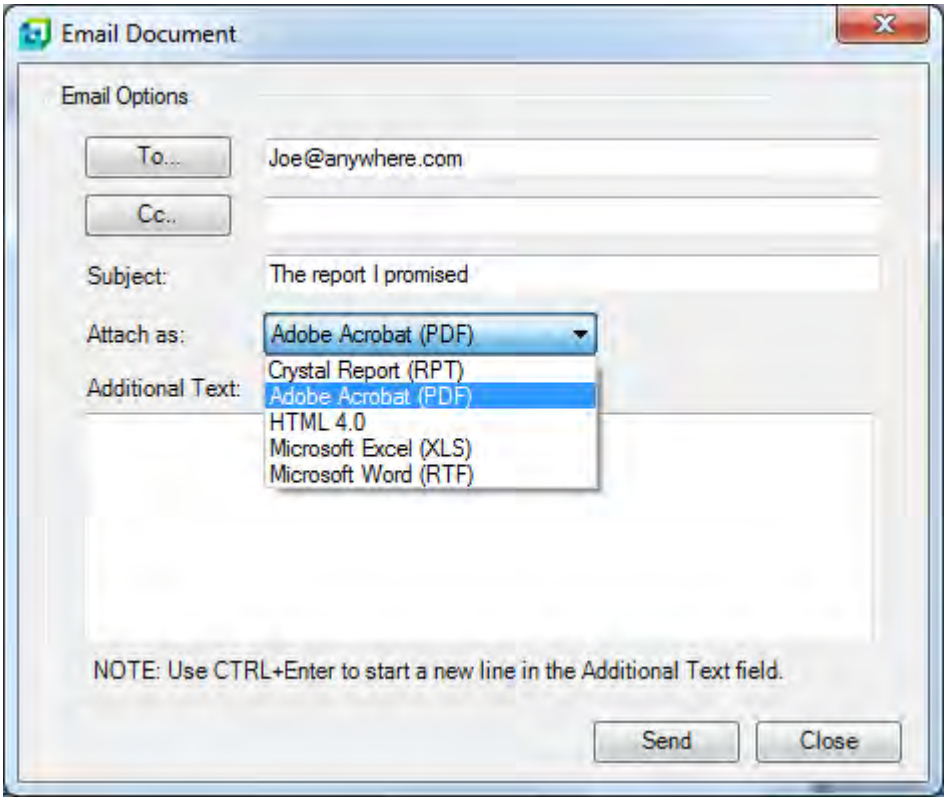


Figure 13-20: Selecting the format of the email attachment

VBScript variable

There is one VBScript variable that is specific to *SRS*/*Crystal Report* customized panes called *CrystalReport*. When this is populated with the full name and pathname of a `.rpt` file the report is displayed. If the following code appears against the customized pane's *OnRefresh* function, and the pane is refreshed, the report `MyReport.rpt` is displayed (this is one line of code even though it has wrapped around on this page):

```
CustomizedPane.CodeObject.CrystalReport =
"C:\TST700\BASE\ReportingCustomized\ReportTemplate\MyReport.rpt"
```

It is unlikely that you would want to change the customized pane to display a different fixed report.

If the following line of code (that has wrapped around on this page) is placed against the customized pane's *OnRefresh* function, it will accept the refresh value passed by another pane and use it as the report name, and display the report.

```
CustomizedPane.CodeObject.CrystalReport =
"C:\TST700\BASE\ReportingCustomized\ReportTemplate\" &
CustomizedPane.CodeObject.RefreshValue & ".rpt"
```

The *SRS*/*Crystal Report* customized pane can be refreshed from another standard pane (or customized pane) that resides within the same program. This is configured within the other pane's VBScripting environment. Once in the VBScripting environment, locate the name of the required customized pane within the *Customized Panes* section of the *Variables* pane, and double-click it. The *Define Action for* screen is displayed (see **Figure 13-21**). Against the *Fire OnRefresh event for this customized pane, passing it this value* prompt, replace the *doRefresh* text with the name of your report (without the `.rpt` suffix).

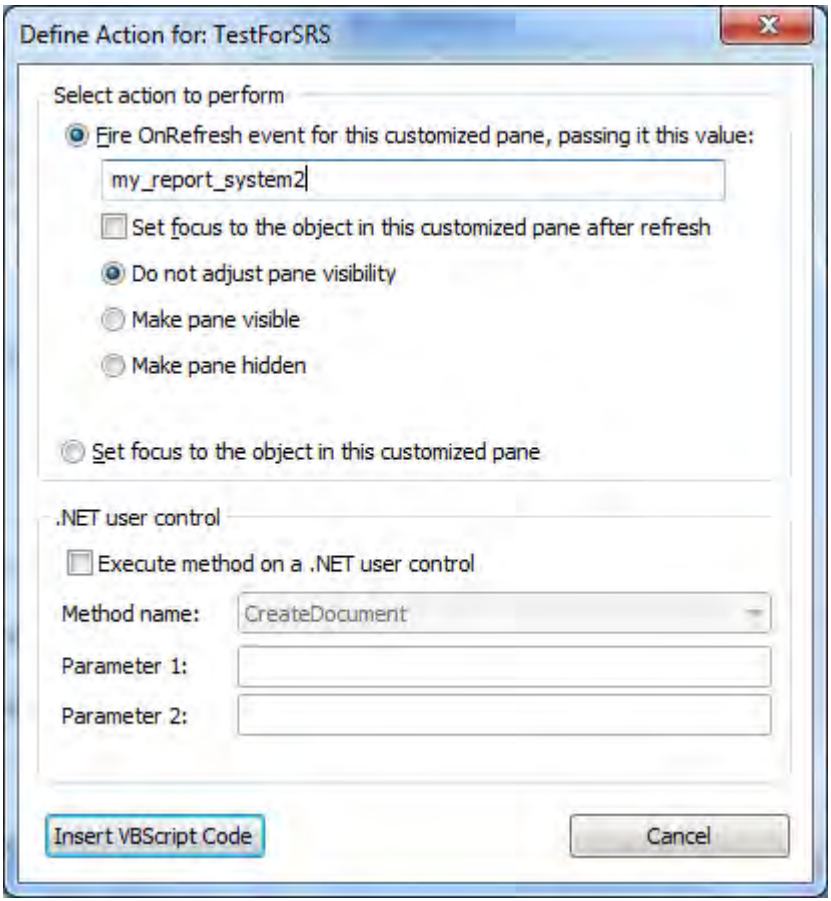


Figure 13-21: The *Define Action for* screen

Once this has been done, click on the *Insert VBScript Code* button at the bottom of the screen and code similar to the following will be added to your code:

```
CustomizedPanes.CodeObject.TestForSRS = "my_report_system2"
```

When this code is invoked the *SRS*/*Crystal Report* customized pane will be passed this value and its *OnRefresh* function invoked. This will display the report.

Chapter 14 - Forms

The *Form* object type displays a customized pane that resembles a standard SYSPRO form. The structure of the form can be built in one of two ways. The first is using the *Design Form* screen that can be accessed from the *Customized Pane Editor* (see **Figure 14-1**). The second is by supplying XML within the customized pane's *OnLoad* function.

Many field types can be added to the form, and placed within groups. Each *Field type* has many properties that can be set against the individual fields to make them appear or perform differently.

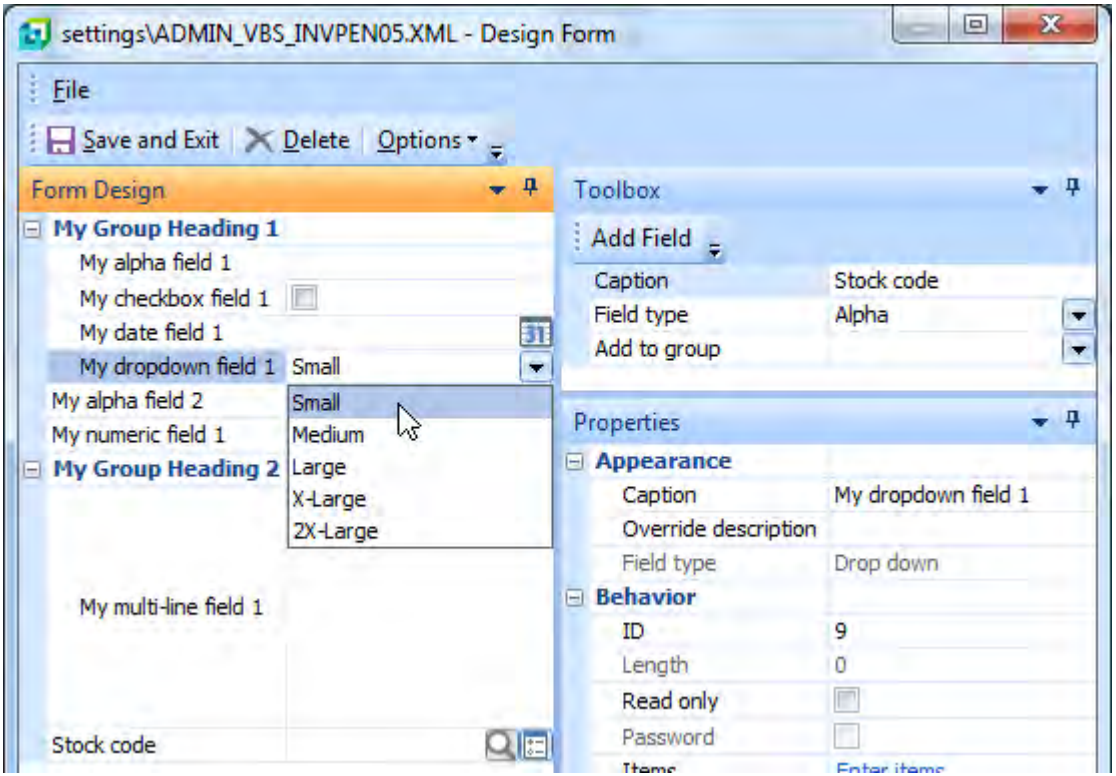


Figure 14-1: The *Design Form* screen

Design Form Screen

The *Design Form* screen makes it easy to build your form. This screen is available by clicking on the *Design Form* button that becomes available on the *Customized Pane Editor* screen's toolbar when the *Object Type* is set as *Form*. It consists of three panes, *Form Design*, *Toolbox*, and *Properties*. The *Toolbox* pane is where you create fields and groups. The *Properties* pane is where you configure their appearance and behaviour, and the *Form Design* pane gives you a representation of how your form will appear when it is complete.

Design Form Screen - Toolbox Pane

The *Toolbox* pane contains three fields; *Caption*, *Field type*, and *Add to group* (see **Figure 14-2**). The *Caption* field is where you supply the field's caption. The *Field type* field has a dropdown list containing the fifteen different field types that are available, including a *Group heading* type (an example of a *Group heading* is the *My Group Heading 1* field in the *Form Design* pane of **Figure 14-1**).

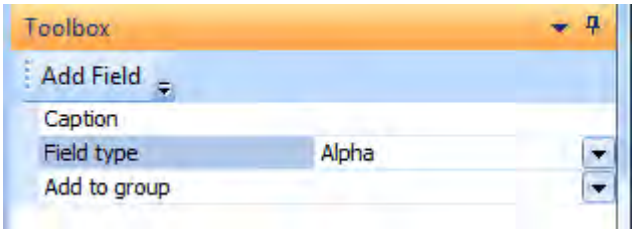


Figure 14-2: The *Toolbox* pane of the *Design Form* program

The *Add to group* dropdown list enables you to add the field that you are creating to a group. The dropdown list contains a list of the groups that have already been created against this form (the fields that have been created with a field type of *Group heading*) along with a blank entry. You can add the field to one of these group headings, or select the blank entry which means that this field is not part of a group. In **Figure 14-1** the field *My alpha field 2* was added using this blank entry against the *Add to group* field, whereas all of the fields preceding this one were added to the *My Group Heading 1* group.

Once you have the correct *Caption*, *Field type* and group, clicking on the *Add Field* button on the toolbar will add the field to the *Form Design* pane. The available field types are listed below, with their basic functions.

Alpha Field
An *Alpha* field can contain up to 3,000 characters, although you can restrict its maximum length to less using the *Length* field property. The displayed value can be set programmatically.

Checkbox Field

A *Checkbox* field displays a square outline (see **Figure 14-3**) which can be checked, or unchecked. When it is checked the value is 1, and when it is unchecked the value is zero. The checkbox can also be checked/unchecked programmatically.

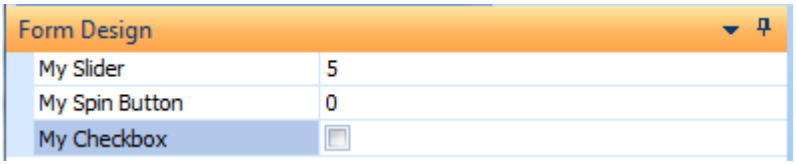


Figure 14-3: A *Checkbox* field that has just been added and appears under the *Form Design* pane

Color Picker

The *Color picker* is used to specify a color. The color can be chosen from the picker that is displayed when you click on the button alongside the field (or from the *More Colors* option beneath the displayed color blocks).

Alternatively, the RGB (Red/Green/Blue) value can be manually entered, delimited by semi-colons.

The displayed RGB value can also be set programmatically, although this value must be supplied as a decimal value. The following is an example of calculating the decimal value from the RGB value.

If the desired RGB color is 204;153;255 you would need to supply the decimal value of 016751052. This is calculated by:

1) Multiplying the green value by 256 (153 x 256 = 39168)
2) Multiplying the blue value by 65536 (255 x 65536 = 16711680)
3) Adding these two results together with the red value (16711680 + 39168 + 204 = 16751052)

Note: the 65536 that you use to multiply the blue value is 256 multiplied by 256.

The end result of this can be seen in **Figure 14-4**. The method of passing a value to the form is covered in the section *Writing Values to a Form* later in this chapter.

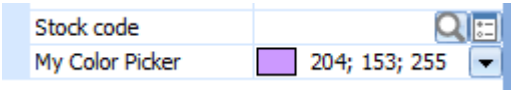


Figure 14-4: The color picker after updating with the decimal value 16751052

Date Field

The *Date* field uses your SYSPRO *Short Date Format* (*Ribbon bar* | *Company Setup* | *General* tab) to specify the format for the display/entry of the date. A date can be entered manually by typing it in, or by clicking on the button and selecting a date from the displayed chooser. Alternatively a date can be

provided programmatically, in which case it must be supplied in the format CCYY-MM-DD. An example using the date of 30th September 2014 would be 2014-09-30.

Drop Down Field

The *Drop down* field type is used to create a dropdown list where the operator can only select from the items in the list; they cannot manually enter a value. The values are supplied by clicking on the *Enter items* hyperlink against the *Items* option on the *Properties* pane (see **Figure 14-5**). The sequence in which the values are entered here is the sequence that they will appear in the dropdown list. In addition, standard SYSPRO icons can be added so that they appear alongside the items. The maximum number of entries in a *Drop down* list, if created using the *Design Form* screen, is 20.

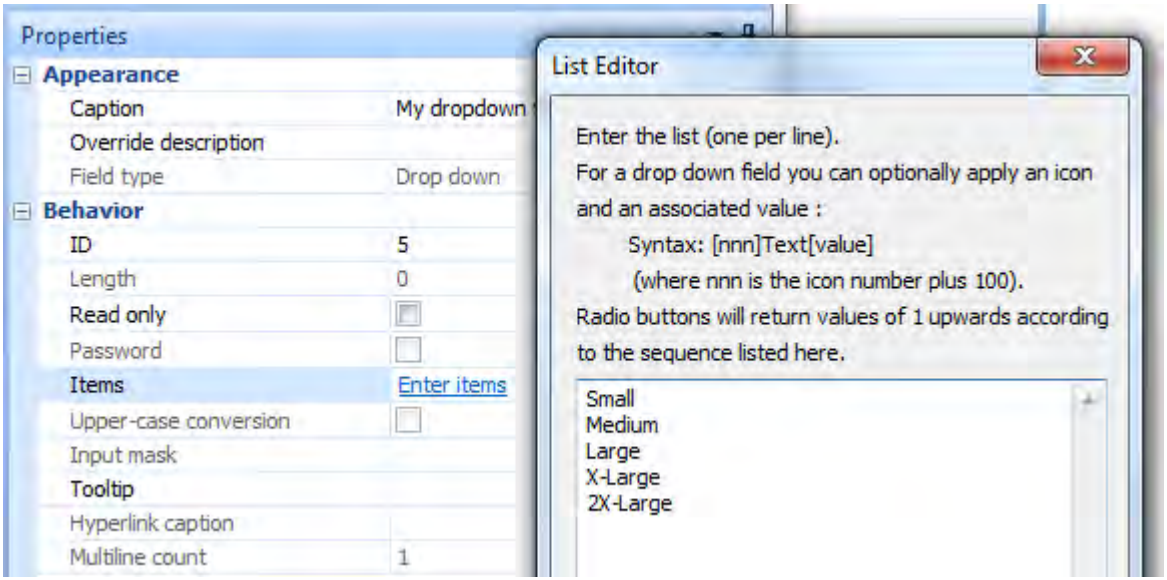


Figure 14-5: Adding the entries to the *List Editor* for use in a dropdown list

If you were to programmatically read back the value of the selected item from the *Drop down* field, the value would match that displayed for the selected item. Using **Figure 14-1** as an example, the value returned for *My dropdown field 1* would be the text: *Small*. You can also programmatically set this by supplying the text: *Small* in your VBScript.

There are times when you don't want to use the text against the selected item to programmatically select the option, or read it back, such as when you are in a multilingual environment. In this environment the list may be language-dependant (which is more likely if the form is being generated using VBScript rather than by adding the fields through the *Design Form* screen). A lot of extra coding would be required to compare the sizes in both Spanish and English, to detect whether the operator selected the *Small*/*Pequeña* size shirt or the *Medium*/*Mediano* one.

To cater for these environments you can provide a parameter within square brackets in the list after the item, as per the examples below. Note that only the real value is displayed in the listview when the operator is using the form.

```
Small[A]
Medium[B]
Large[C]
X-Large[D]
2X-Large[E]
3X-Large[F]
```

If you programmatically read back the value after it has been selected from the list, it is the value within the square brackets that is returned, which can be made to be the same in all languages. If the operator had selected the *Medium* option, the value returned to you would be the letter *B*. You can also prepopulate the field by programmatically selecting the value within the square brackets.

SYSPRO standard icons can also be added to the items in a dropdown list so that they appear in front of each item (see **Figure 14-6** where the icons happen to be colored squares).

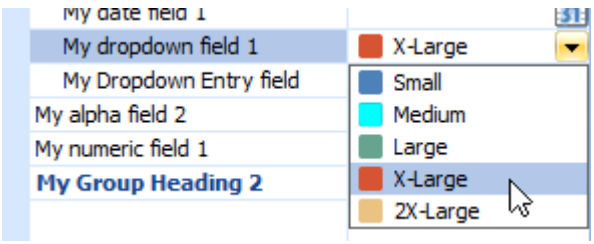


Figure 14-6: A *Drop down* list with icons added

This is done by supplying the icon numbers in square brackets in front of the option when defining the list. The code to achieve the dropdown list in **Figure 14-6** appears below:

```
[266]Small[A]
[267]Medium[B]
[268]Large[C]
[269]X-Large[D]
[270]2X-Large[E]
```

The list of available icons can be seen by clicking on the *View Icons* button at the bottom of the *List Editor* (where you enter the items to appear in the dropdown list). The *Change Icon* screen appears is **Figure 14-7**. Scroll through the list to find the icons that you are going to use. When one of the icons is selected its icon number is displayed at the bottom left of the screen. In the case of **Figure 14-7** the *Diamond Blue* icon is selected, and the icon number is 166.

When adding the icon number to the item in the list, you must first increase the icon number by 100. So, to use icon number 166 you must add 100 and use the icon number 266 (as indicated in the code sample above).

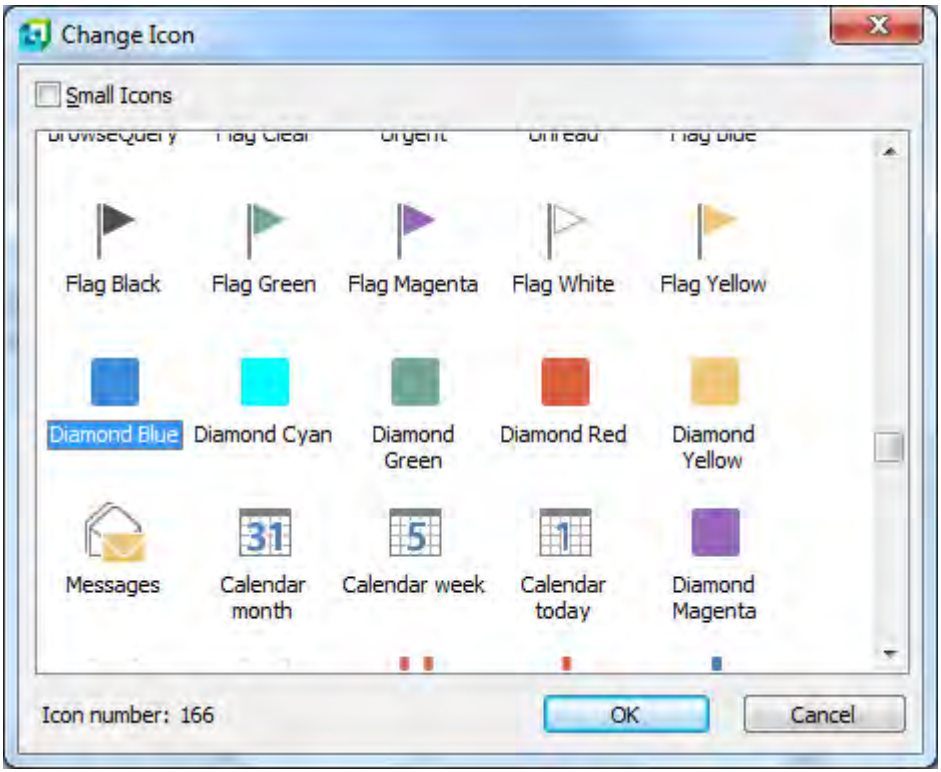


Figure 14-7: The list of icons, with icon number 166 selected

If (when designing the form) you select an item from this *Drop down* field within the *Form Design* pane, then click on the *Save and Exit* button, each time that the customized pane is loaded this dropdown field will be prepopulated with this default value (unless your code overwrites this setting each time the pane is displayed).

Drop Down Entry Field

A *Drop down entry* field is very similar to a *drop down* field. The difference being that the operator can also manually enter a value that doesn't appear in the list. **Figure 14-8** shows that the dropdown list against the *My Dropdown Entry field* field only contains the options *Small*, *Medium* and *Large*, but the operator had previously typed in the value *Enormous*.

The maximum number of entries in a *Drop down entry* list (if created using the *Design Form* screen) is 20.

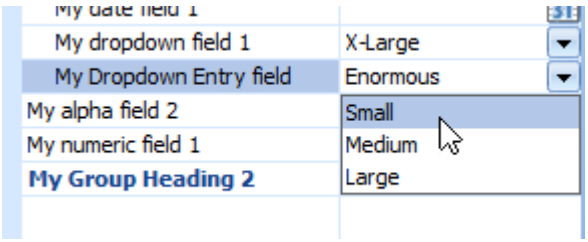


Figure 14-8: A *Drop down entry* field can have values entered that do not appear in the list

Font Field

The *Font* field is used to create a field on the form where a font type/size/style can be selected by clicking on a button. When the button is selected the *Font* screen is displayed (see **Figure 14-9**)

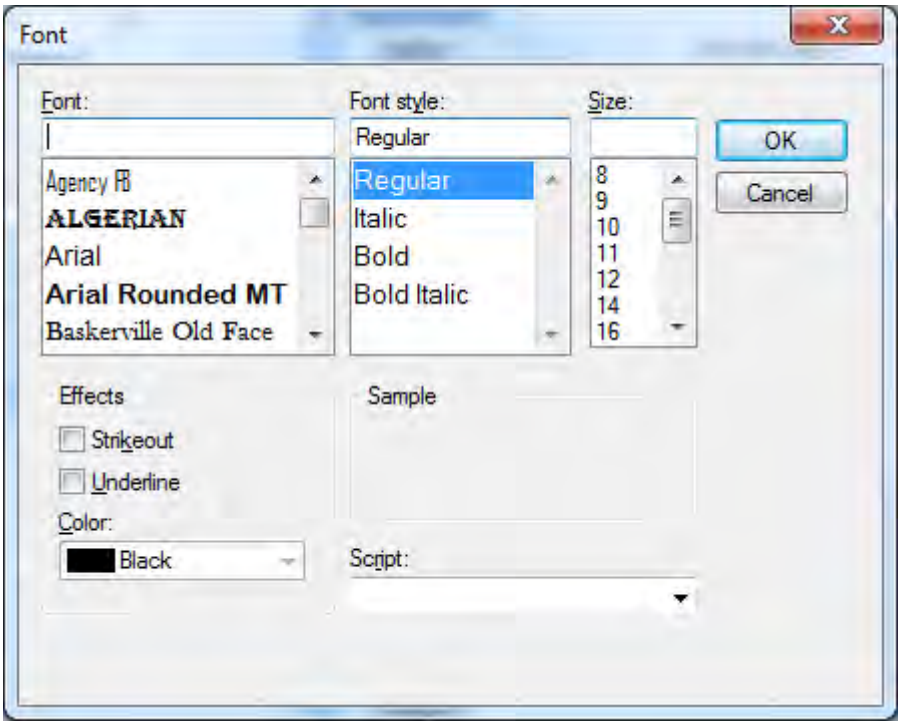


Figure 14-9: The *Font* screen where you select the font type/size/style

Reading back the value from a *Font* field returns the text as it is displayed in the field (see **Figure 14-10**). You cannot programmatically write to the *Font* field.

Figure 14-10: Displaying the contents of a *Font* field.

Hyperlink Field

The *Hyperlink* field is used to display a hyperlink, and call a VBScript function if the hyperlink is clicked by the operator. The text of the hyperlink can be defined, as can the tooltip. **Figure 14-11** shows where the hyperlink text has been changed to *Validate Shipping Cost*, and the tooltip changed to give a meaningful description of what will happen if the hyperlink is clicked.

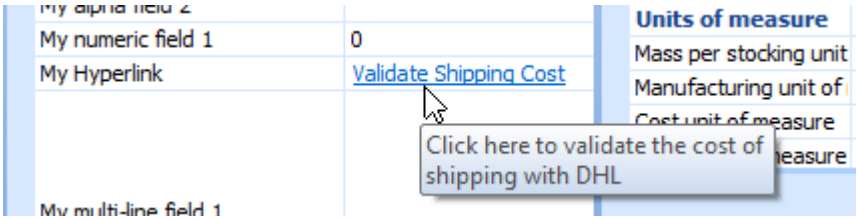


Figure 14-11: The text and tooltip changed against the *Hyperlink*

Because the hyperlinked field could have any name, you will need to manually create the function in your VBScript. The name required for the function is the name of the *Hyperlink* field (with any spaces and special characters removed) followed by *_OnLinkClicked*. An example function appears below that matches the *My Hyperlink* field in **Figure 14-11**. You add your VBScript code between the *Function* and *End Function* statements.

```
Function MyHyperlink_OnLinkClicked()
   --- Insert the code required ---
End Function
```

Group Heading Field

A *Group heading* field is used to house fields that fit together logically. **Figure 14-12** shows the *Group heading* of *Units of Measure* which is used to group these related fields.

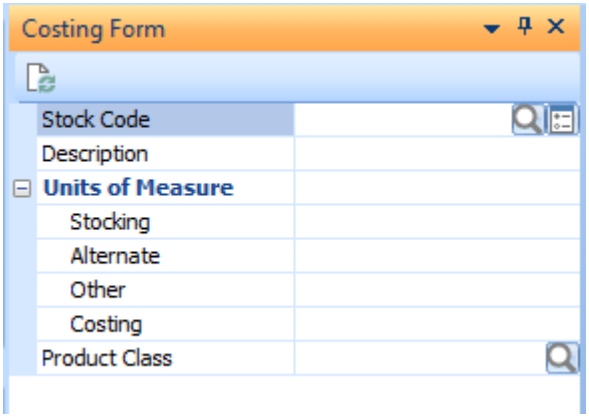


Figure 14-12: Using a *Group heading* to group fields logically

A group can be collapsed by clicking on the "–" sign that appears alongside the *Group heading* field. If the group is in a collapsed state, the group can be expanded by clicking on the + sign that appears against the *Group heading*. When the operator right-clicks on the form and selects the *Collapse all* option from the context-sensitive menu (see **Figure 14-13**), all of the groups are collapsed. The *Expand all* button will expand all of these groups.

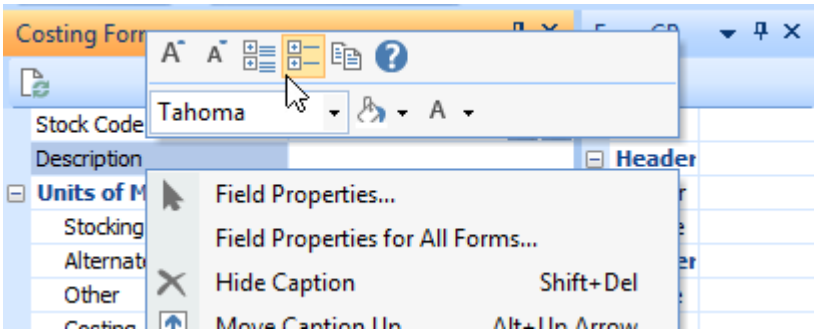


Figure 14-13: Using the *Collapse all* option against the context-sensitive menu

Fields within a group can be dragged from the group and placed elsewhere on the form. Fields that were not part of a group (or which were in another group) can be dragged into this group. If the *Group*

heading is dragged to a new location, all the fields within the group move with the header. A group cannot be dragged into another group, as only one level of grouping is allowed.

If the *Group heading* is hidden using the *Hide caption* option (from the context-sensitive menu that appears when you right-click on the form), all of the fields within the group will be hidden. If in the *Design Form* screen the *Group heading* field is deleted, all the fields within the group are deleted. If you drag the last field from a group, the "–" sign no longer appears against the group heading. **Figure 14-14** shows where the last entry has been dragged from the group called *Group* and the "–" sign has been removed.

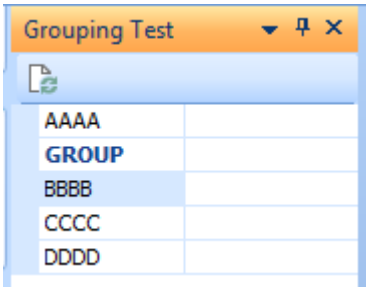


Figure 14-14: The *Group Heading* no longer contains any entries

As the group heading no longer acts like a group, you cannot drag a field back into the group. However, a field can be added to this group by dragging it so that it is directly below the group heading, and using the *Alt+UpArrow* keyboard shortcut (that is normally used to move a field up one position on a form). **Figure 14-15** shows what happens when the field *BBBB* is positioned directly below the group heading of *GROUP*, and the *Alt+UpArrow* keyboard shortcut is used. The field is added to the group, the "–" sign reappears, and the group can be accessed as before.

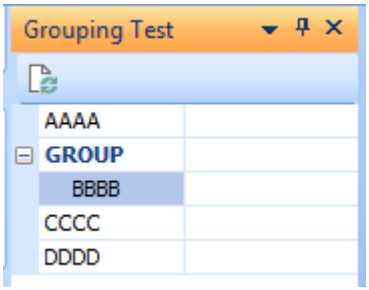


Figure 14-15: Using the *Alt+UpArrow* keys to add an item to a group

Within the *Properties* pane when designing the form there is a property called *Show group collapsed* within the *On first time load* section. If this checkbox is checked against a *Group heading* field when

doing the initial creation of the customized pane, the group will be collapsed. However, if the operator expands this group and exits the program, this will be saved within the operator's preferences and will appear expanded the next time that the operator uses this program.

Multiline Field

A *Multiline* field can contain up to 3,000 characters, which is the same as an *Alpha* field. However, an *Alpha* field can only appear on one line, and if the content is larger than the available space you can only see the first few characters, depending on the width of the field. Alternatively, moving your mouse pointer over the field will display the full content in the tooltip, providing that a tooltip has not previously been defined against this field. This can be very limiting if the customized pane is not very wide. An example can be seen in **Figure 14-16**.

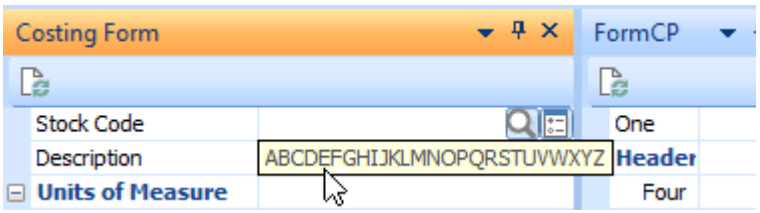


Figure 14-16: The full content of an *Alpha* field being displayed if no tooltip is configured

A *Multiline* field will wrap the text across multiple lines. If the text is greater than the available display space, a slider bar will appear alongside the text. **Figure 14-17** shows a *Multiline* field that contains more text than can be displayed, and a slider bar appears at the right of the field.

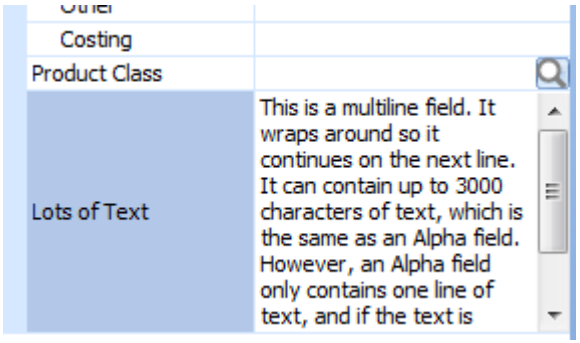


Figure 14-17: Using a *Multiline* field

The depth of the *Multiline* field can be set using the *Multiline count* property on the *Properties* pane of the *Design Form* screen. The number against this field is the number of lines that the field covers. The default size is 9 lines. Text can be programmatically posted to a *Multiline* field.

Numeric Field

A *Numeric* field is used to hold numeric values; a non-numeric value cannot be input to this field. The maximum size of a numeric field on a form is 12.6 (999,999,999,999.999999), but the field can be configured to have a smaller size using the *Length* and *Decimals* properties on the *Properties* pane. On this pane the *Length* field sets the total size of the field, and defaults to 0, which means using the field's maximum size. Setting the *Length* property to be 17, and the *Decimals* property to be 6, means that the operator can enter an 11.6 value (99,999,999,999.999999).

Other properties that affect numeric fields are *Allow negative values*, and *Edit without commas*. The *Allow negative values* property specifies whether a negative value can be entered into the field. The negative sign can be supplied before or after the value. The *Edit without commas* property causes the value to be displayed with/without the comma separators that make a large number easier to read. If the property is checked the 11.6 value above would be displayed as 99999999999.999999 instead of 99,999,999,999.999999. The number can be entered by the operator with or without the comma separator present, regardless of whether this property is checked. The value will be displayed as per the property setting.

Picture Field

A *Picture* field is used to display an image on the form. When a picture field is added within the *Toolbox* pane of the form designer, it appears in the *Form Design* pane (see **Figure 14-18**). At this point there is no image associated with the picture field. A browse button appears alongside the picture field, and this can be used to locate an image to be used for this field. **Figure 14-19** shows the *Form Design* pane once the browse has been used to locate the image and retrieve it.

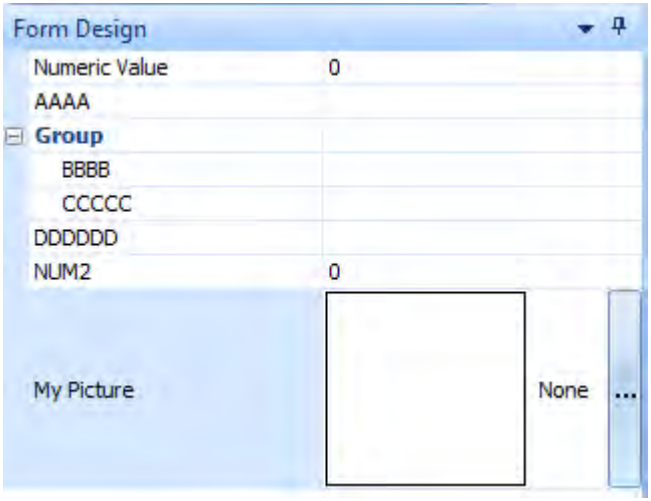


Figure 14-18: A *Picture* field added to the *Form Design* pane

When the customized pane is saved, this image will always appear against this picture field. **Figure 14-20** shows the picture field when the customized pane is in use. If you double-click on the image it will be displayed in whatever image viewer is associated with this image type on the operator's machine.

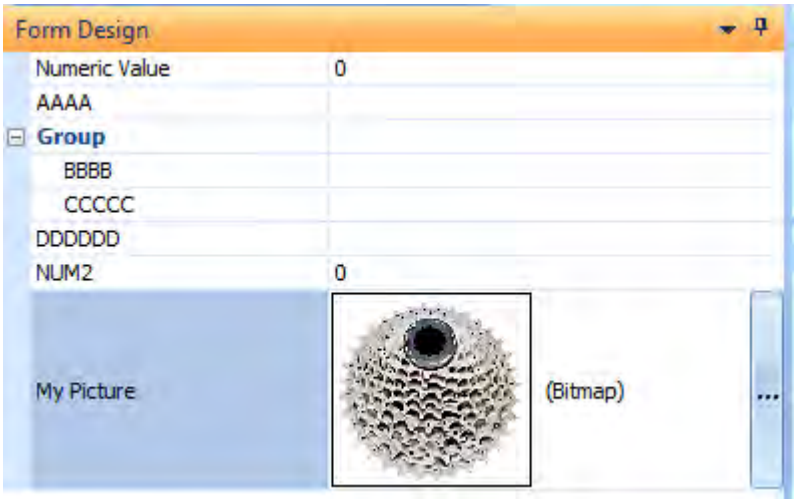


Figure 14-19: The *Form Design* pane after the browse has been used to locate and retrieve an image

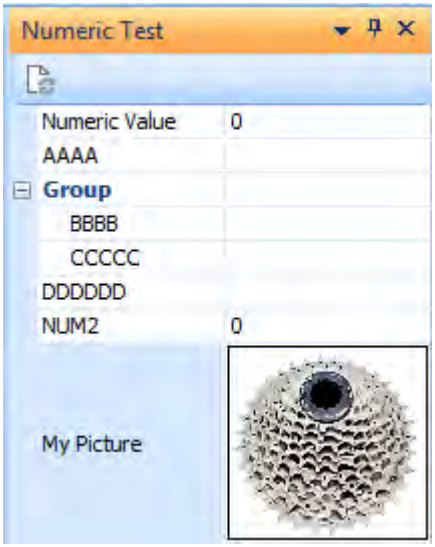


Figure 14-20: The *Picture* field in the customized pane with the image associated with it

Note that there is no browse button associated with this field in the customized pane. It is possible to create a browse button against this picture field that will allow the operator to select an image. However, as this requires some VBScript code, this is covered in the section *Creating a Picture Browse* that appears later in this chapter.

Radio Button Field

A *Radio button* field is used where you want to give the operator a choice between a fixed number of options, and they can only select one. Because of the amount of space that it can take up, you would usually only use a radio button when there are a limited number of items from which to choose. When the list becomes large you would typically use a dropdown list instead. When a *Radio button* field is added using the form designer, the field appears similar to an *Alpha* field in the *Form Design* pane.

Figure 14-21 shows the newly-added radio button field before it has been configured with the *Items*. The *Items* are added by highlighting the *Radio button* field in the *Form Design* pane and clicking on the *Enter items* hyperlink against the *Items* property in the *Properties* pane. This displays the *List Editor* screen where you enter your items. This is the same screen that is used to enter items for a dropdown list.

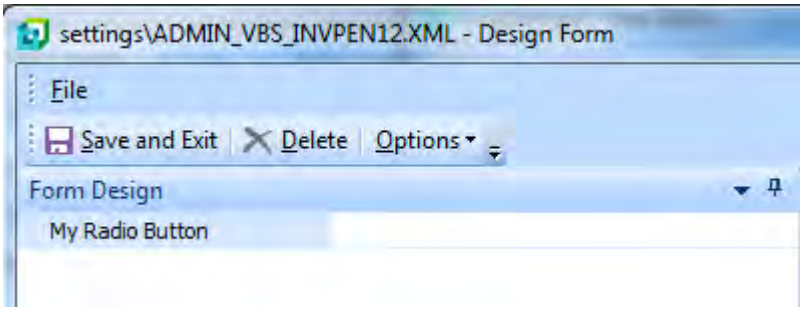


Figure 14-21: A newly-added *Radio button* field before it is configured.

Figure 14-22 shows the *List Editor* with 5 options. Unlike dropdown fields, the *List Editor* cannot be used to add icons against the items for radio buttons; nor can it specify the value that must be returned when an item is selected.

Once the list has been entered in the *List Editor* and the *OK* button is selected, the *Form Design* pane is updated to include the radio buttons for this field (see **Figure 14-23**).

Within the screen where the VBScript is edited, each *Form* customized pane has its own section (within the *Variables* pane). The name of the section is the customized pane name with any spaces and special characters removed. The name of the customized pane being used in **Figure 14-23** is *My Radio Button Test*, so this section is called *MyRadioButtonTest*.

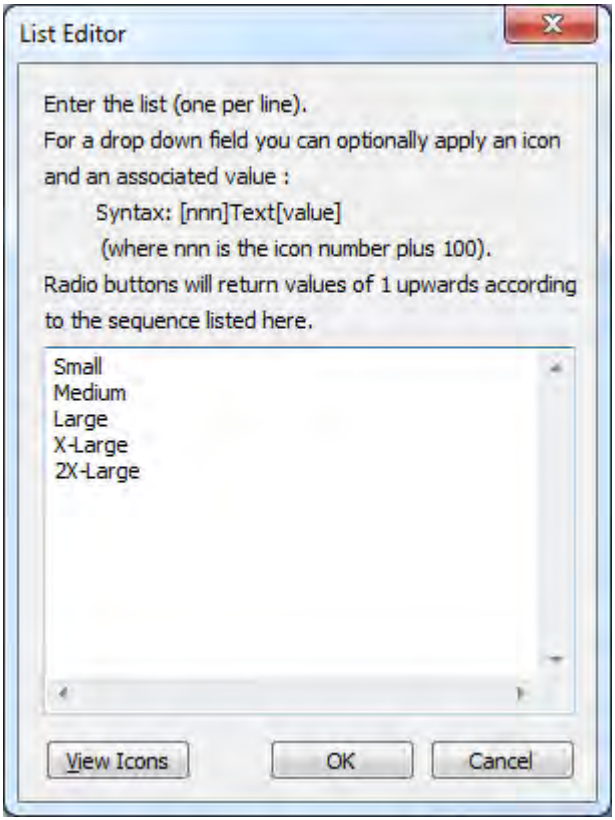


Figure 14-22: Entering the list items in the *List Editor*

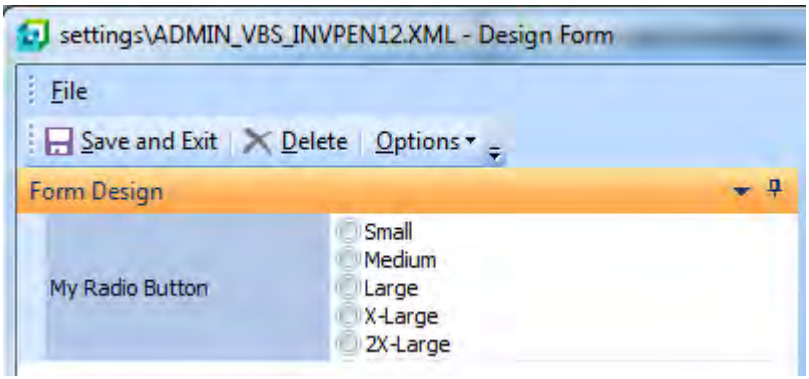


Figure 14-23: The radio buttons added to the *Radio button* field

Each field within this *Form* customized pane will be listed under this section (see **Figure 14-24**). The name is the name of the field with any special characters and spaces removed, so *My Radio Button* is *MyRadioButton*.

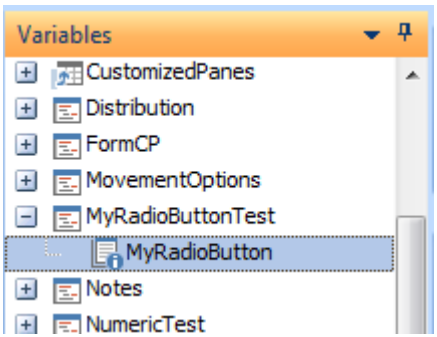


Figure 14-24: Selecting the *MyRadioButton* variable name

The value of the radio button that has been selected can be retrieved programmatically using this variable. If no radio button has been selected the value will be zero (or 000000000). If the first item in the list has been selected (*Small* in **Figure 14-23**) the returned value will be one (or 000000001). The second item will be two (or 000000002) etc.

The radio button value can be set programmatically by specifying the value. An example using a radio button on a *Form* customized pane appears below. This selects the second item of the radio button if the product class is *MB – Mountain Bicycles*, or the first one if the product class is anything else.

```
Function CustomizedPane_OnRefresh()
  If StockCodeDetails.CodeObject.ProductClass = "MB - Mountain Bicycles" then
    MyRadioButtonTest.CodeObject.MyRadioButton = 2
  Else
    MyRadioButtonTest.CodeObject.MyRadioButton = 1
  End If
End Function
```

Slider Field

A *Slider* field provides a means for an operator to enter a value, but being restricted within a range. When a *Slider* field is added it appears in the *Form Design* pane. **Figure 14-25** shows a slider that has just been added to the *Form* customized pane. When the slider is moved, the value representing this position appears to the left of the slider.

When the *Slider* is selected in the *Form Design* pane, the *Min value* and *Max value* properties become available. These enable you to set the lowest and highest values for the slider. If *Min value* is set to 5 and *Max value* is set to 25, the operator can only select from 5 to 25 (inclusive) using the slider.

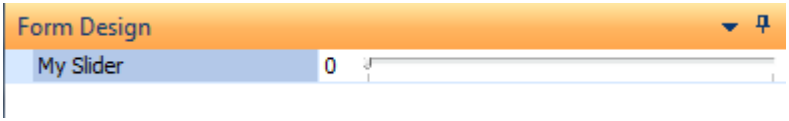


Figure 14-25: A *Slider* field that has just been added to the *Form Design* pane

When focus is not set on the *Slider* field in the customized pane, the slider disappears and just the selected value remains against the field. If you set focus back on the slider field the slider reappears.

Spin Button Field

A *Spin button* field provides the same functionality as a *Slider*. The difference is that a *Spin button* has up and down buttons in place of a slider. **Figure 14-26** shows a *Spin button* that has been added to a customized pane. The buttons to change the value up and down appear on the right. The *Min value* and *Max value* properties can also be used to set limits.

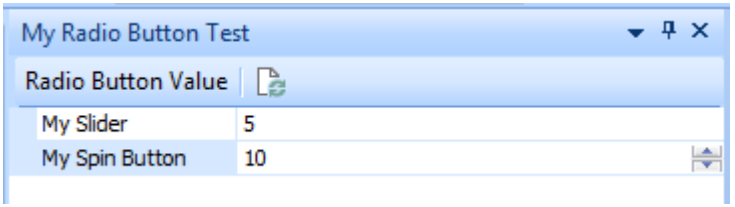


Figure 14-26: A *Spin button* showing up and down arrows, and the selected value of 10

Design Form Screen - Form Design Pane

The *Form Design* pane shows you how the form will appear when you save the customized pane. As new fields are created they are added to the bottom of this pane. If a group is specified against the field when it is created, the field will be added to the end of that group. If no group is specified, the field is added to the end of the form.

Fields can be dragged to a new location within the form. If a group is dragged to a new location, the group heading and all of the items within the group are moved to the new position. Fields can be extracted from a group by dragging the field to a new location outside of the group. Fields can be added to a group by dragging them to a position within a group. Note that only one level of grouping exists, so you can't drag a group into another group.

The keyboard shortcuts *Alt+UpArrow* and *Alt+DownArrow* can be used to move the currently selected field up or down one position. If the last field has been dragged from a group, the group loses its grouping functionality. The field immediately below the *Group heading* can be added back to the group by highlighting it and using the *Alt+UpArrow* keyboard shortcut, at which point the full group functionality will be restored.

When you select one of the fields within the *Form Design* pane, the functionality associated with this field type becomes available, so that you can test it. For example, in **Figure 14-1** the dropdown button against the *My dropdown field 1* field has been selected, and the *Small* option highlighted.

As with other forms, when a key field is added to a form (or a field with the same name as a key field) a browse is automatically added alongside it. In **Figure 14-27** a field has been added called *Stock code*. Because this is the same name as a key field, a browse button has been added automatically. Clicking on the browse button, even in this form design environment, causes the browse to be run.

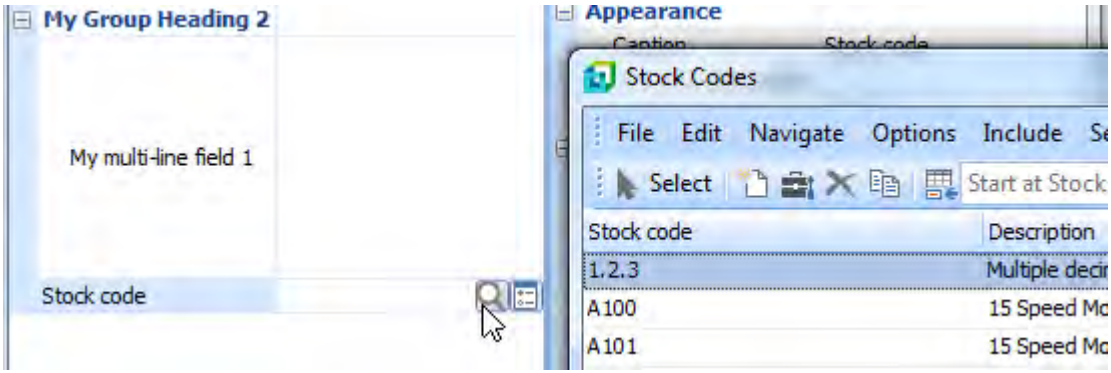


Figure 14-27: Testing the browse associated with the *Stock code* field

As this key field has alternate keys associated with it the *Alternate keys* button is also automatically added. This is used to select whether the standard key, or one of the available alternate keys is to be used for the browse.

Design Form Screen - Properties Pane

The *Properties* pane is used to set/amend the properties against a field. However, the *Field type* that was specified when the field was created cannot be changed. If the field was created with the incorrect *Field type*, the field must be deleted and re-added with the correct *Field type*.

There are fifteen field types that can be selected when adding a field, and these have been covered within the *Toolbox* pane section above. There are 27 properties that can be set, and these appear

within four groups, *Appearance*, *Behavior*, *Numeric behaviour*, and *On first time load*. Some properties are available for all field types, whereas others are only relevant for certain types of fields.

When a field is highlighted within the *Form Design* pane, the *Properties* pane is populated with any properties that have been selected for this field (or the default values if none have already been selected). The properties that are not relevant to this field type are greyed-out.

The following properties appear within the *Appearance* group:

Caption Property (all field types)
The *Caption* is the label that will appear against this field on the form. The caption can be changed after the form has been created.

Override Description Property (all field types)
The *Override description* property is used to cater for forms that must contain multiple fields with the same captions. **Figure 14-28** shows where two fields needed to appear on the same form with the caption of *Value*. The highlighted second one has an *Override description* of *Branch Value*.

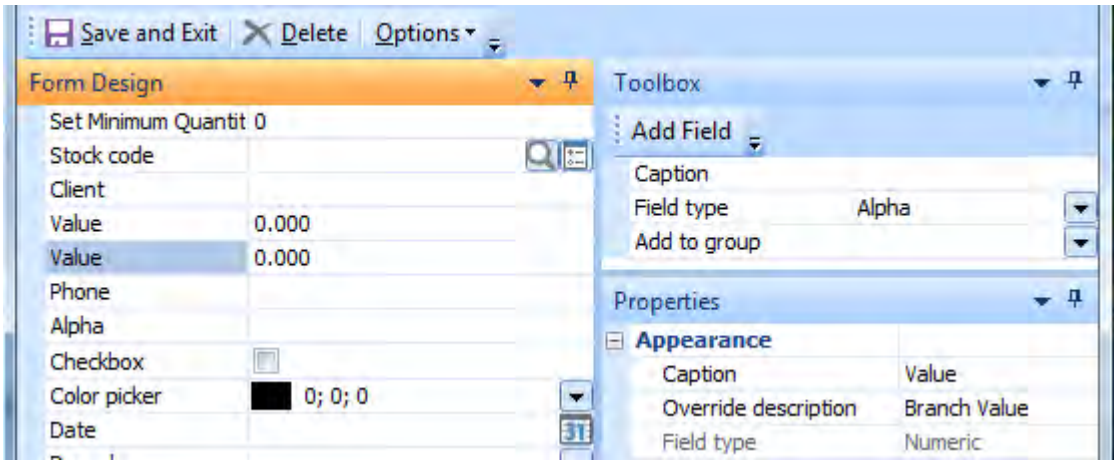


Figure 14-28: Using the *Override description* property

Another use of the *Override description* property is where you want to use the functionality that automatically becomes available when you add a field to the form that has the same name as a standard SYSPRO field. If the standard SYSPRO field would have a browse, and you create a field on your form with the same name, this field automatically has a browse against it. For example, adding a field to your form with the name *Customer* will mean that this *Customer* field will have a browse button against it. If the operator clicks on this button the *Customer Browse* program is invoked.

If you want to add a field to your form that uses a standard browse, but you don't want to use the standard field name, you can use the *Override description* option to enable the browse. An example is that you want the caption of *Client* to appear on the form, but you want to have the functionality of the *Customer Browse*, as if this field's caption was *Customer*. By making the caption *Client*, and the *Override description* of *Customer*, the *Client* field will automatically have a *Customer Browse* against it.

Figure **14-29** shows that the field *Client* has an *Override description* of *Customer*. The *Form Design* pane (that previews how the form will display) shows the *Client* field but without the browse against it.

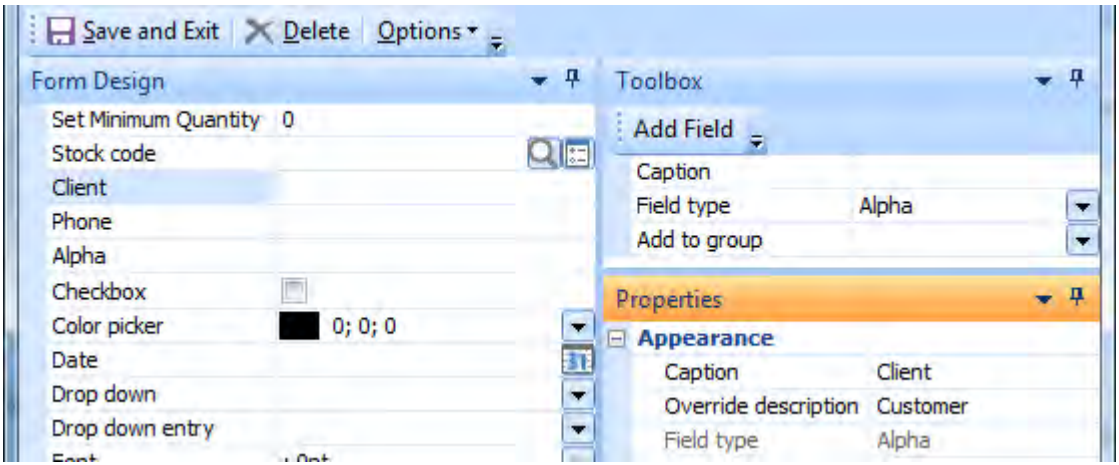


Figure 14-29: Using the *Override description* property to put a browse against a field

Figure 14-30: Shows the customized pane form after the form has been saved. The *Client* field has a browse button against it. When this button is clicked by the operator, the *Customer Browse* is invoked. If the operator selects one of these customers, the customer account code is returned to the *Client* field on the form.

Field Type Property (all field types)
The *Field type* property displays the field type that was selected when the field was created. This is a read-only field for all fields.

If the field type is incorrect against a field, the field will need to be deleted and added again, as the field type cannot be changed.

The displayed value against this field is the same as the *Field type* selected when adding the field, except for the fields *Color picker* which is displayed as *Color*, *Radio button* which is displayed as *Option*, and *Spin button* which is displayed as *Numeric*.

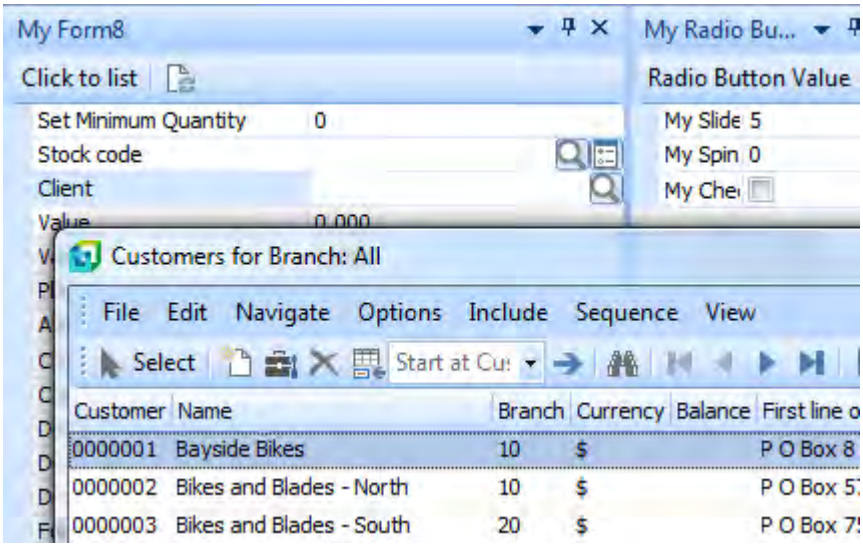


Figure 14-30: The *Customer* browse against the *Client* field

The following properties appear within the *Behavior* group:

ID Property (all field types)
A unique (to this form) *ID* is allocated to each field as it is added to the form. The ID number can be changed. The ID number can be accessed programmatically in VBScript, which means that you can iterate through the fields looking for the one associated with this ID number, even if you do not know the name of the field.

Length Property (Alpha, Numeric, Multiline, Drop down entry, Spin button)
The *Length* property is used to define the maximum length of a field. For *Alpha* and *Multiline* fields the maximum value for this property is 3,000. For *Numeric* fields this property works in conjunction with the *Decimals* property that sets the number of digits after the decimal point. The maximum size of a numeric field is 12.6 (twelve integers with six digits after the decimal point). If you want to restrict this field to a maximum value of 999,999.999 (6.3) you would need to set the *Length* to 9, and the *Decimals* to 3.

The *Length* property can be set against a *Drop down entry* field. This only affects the length of the field when the operator manually enters a value, instead of selecting a value from the dropdown list.

The *Length* property can also be set against a *Spin button* field. This only affects the length of the field if the operator manually enters a value. If the operator uses the *Up* button to increase the value, it can increase until it is equal to the *Max Value* property.

Read Only Property (all field types)
When checked, the *Read Only* property is used to prevent operators from changing the value of a field. For fields that would have a button alongside them (browse, dropdown list, color picker, etc.) the button will not appear when the field is set to *read-only*. For *Hyperlink* fields the hyperlink will not appear if the field is set to *read-only*.

The default behavior for *read-only* fields on a form is for the caption to be greyed-out. In **Figure 14-31** the fields *Alpha*, and *Drop down* have their *Read only* property checked and have been greyed-out. In addition, the dropdown button alongside the field *Drop down* has been removed.

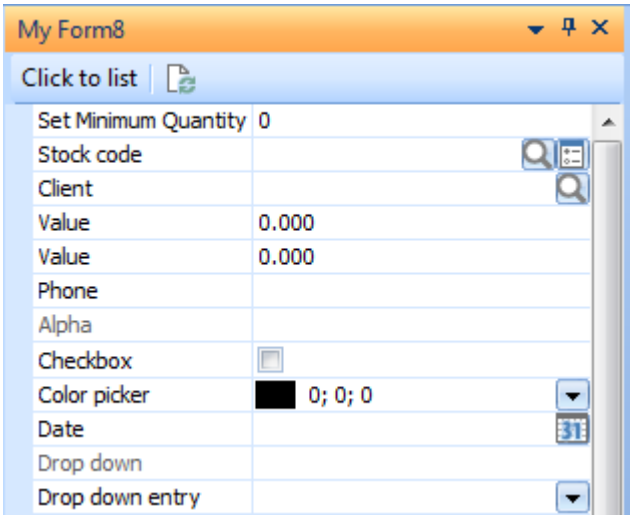


Figure 14-31: The captions of the read-only fields *Alpha* and *Drop down*, greyed-out

An option exists on the *Forms* tab of the *Customize* screen (*Ribbon bar* | *Home* tab | *Customize*) to specify the color of the caption for *read-only* fields. The default color is a medium grey. This can be changed using the color picker against the *Read-only color* field. Note that if you later decide to set the *Read-only color* back to its default you should select the *Automatic Color* option from the color picker.

Figure 14-32 shows the *Read-only color* field being set to *Plum*. The *Automatic Color* option can be seen at the top of the color picker. Selecting this will set the *Read-only color* back to its default.

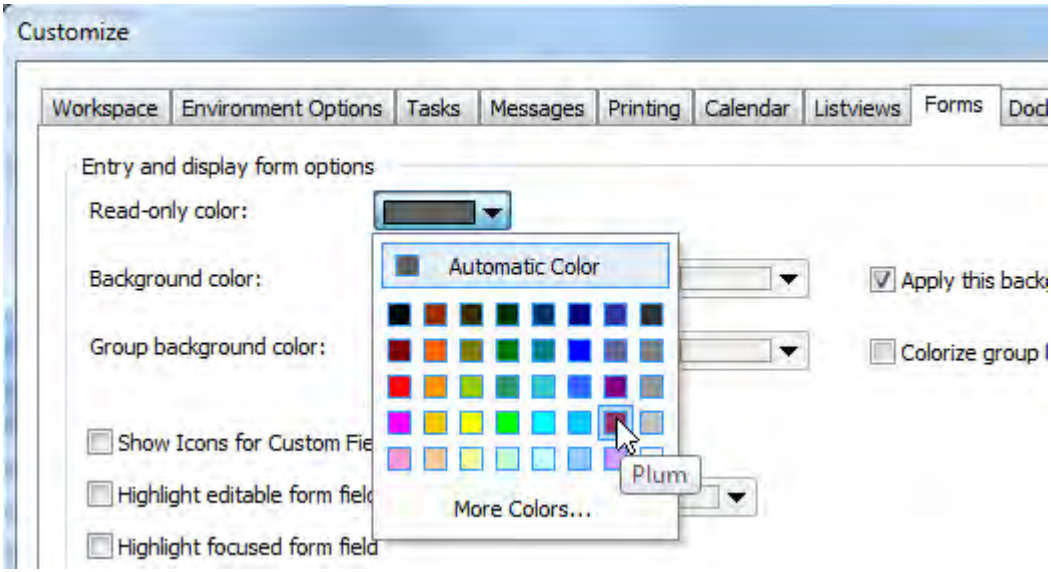


Figure 14-32: Changing the color of the captions of read-only fields.

Password Property (Alpha)
Only *Alpha* fields can have the *Password* property set. This property replaces each character with an asterisk as it is displayed, so that the true value is hidden. Only the display is affected. The value for this field remains as is. **Figure 14-33** shows the field called *Alpha* after it has been configured with the *Password* property. The displayed value against this field is replaced with asterisks, but the entered value remains the same. The real content of this field can be seen as ABCDE as it is being displayed in a message box.

Items Property (Drop down, Drop down entry, Radio button)
For *Drop down*, *Drop down entry*, and *Radio button* fields, the *Items* property will have an *Enter items* hyperlink against it. When this hyperlink is selected the *List Editor* screen is displayed. **Figure 14-5** shows the *Enter items* hyperlink against the *Properties* pane, and the *List Editor* screen.

The *List Editor* screen is used to supply the options for dropdown and radio button fields. The options will appear in the dropdown list (or against the radio button field) in the same sequence as they appear in the *List Editor*.

Drop down and *Drop down entry* fields can have icons displayed alongside the options in the dropdown list. They can also be configured to return a value other than that selected by the operator from the list. This was covered in detail within the *Drop Down* sub-section of the *Design Form Screen - Toolbox Pane* section above.

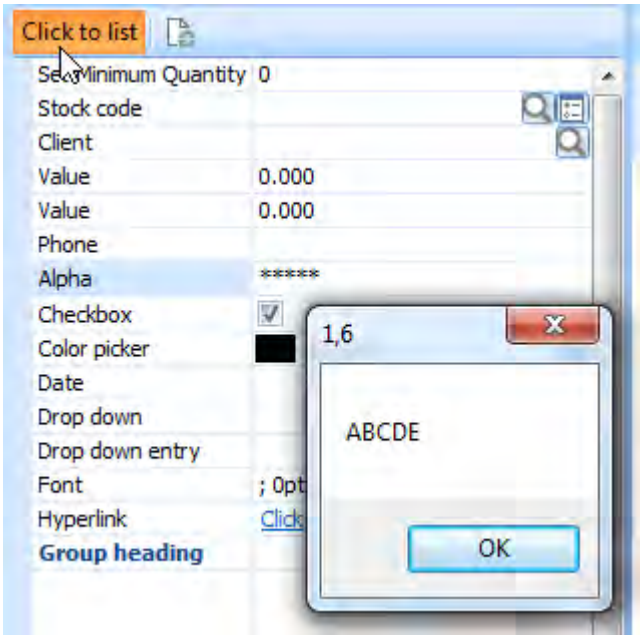


Figure 14-33: The *Password* property set against the *Alpha* field

Uppercase Conversion Property (Alpha)
The *Uppercase conversion* property is only available for *Alpha* fields. It converts all lowercase letters to uppercase as the operator enters them. This does not just affect the display of the value, it converts the value to uppercase. If the form/value is saved, the uppercase letters are written away.

Figure 14-34 shows where the operator has entered the value *AbCdE* against the field *Alpha*, which has its *Uppercase conversion* property checked. The value has been converted to *ABCDE* on the form. The value of this field has also been displayed using a message box statement, and it can be seen that the value has been converted to *ABCDE*; it is not just the display of the value on the form that is affected.

Input Mask Property (Alpha)
The *Input mask* property is only available for *Alpha* fields. This property is used to define the structure of the field, and which characters can appear in each position within the field. An example of how the *Input mask* property works can be found within the *Input Mask Sample* section in Chapter 7, which covers using the input mask with standard forms, but the way it works is exactly the same as with customized panes.

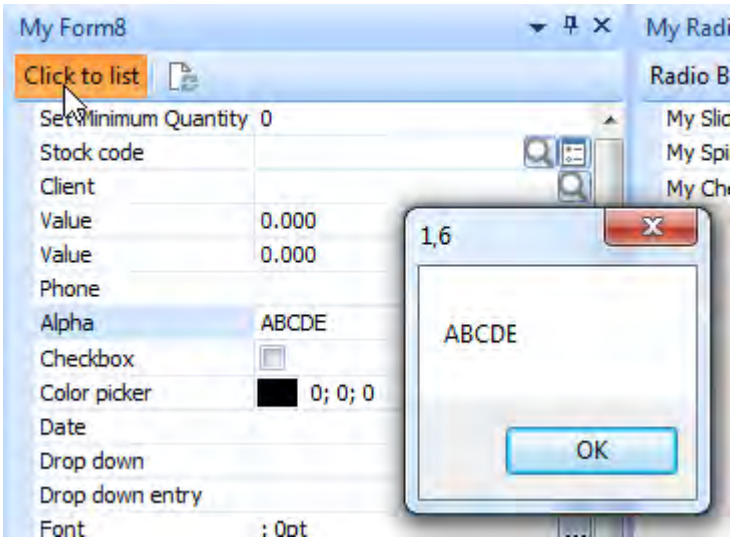


Figure 14-34: The *Uppercase conversion* of the *Alpha* field

Tooltip Property (all field types except Group heading)
The *Tooltip* property allows you to enter text that will appear when an operator moves the mouse pointer over a field. The exception is the *Group heading* field where although a tooltip can be entered against a *Group heading* field, the tooltip will not be displayed.

If a tooltip has been entered against a *Date* field, it will appear when the operator moves the mouse pointer over this field. However, if no tooltip has been entered against a date field's *Tooltip* property, the property will default to the text *No date*. If the field does not contain a date, the tooltip will display *No date* when the operator moves their mouse pointer over the field. If a date has been entered against this field, the tooltip will display this date using SYSPRO's long date format.

When a *Hyperlink* field is added to a form, a default tooltip of *Click to query* will be present. This tooltip can be changed.

Hyperlink Caption Property (Hyperlink)
The *Hyperlink caption* property is the text that will become the hyperlink against a *Hyperlink* field. When a *Hyperlink* field is added within the *Design Form* screen, the default *Hyperlink caption* is *Click*. This property can be changed to any text, up to 40 characters long.

Multiline Count Property (Multiline)
The *Multiline count* property is used to specify how many lines the *Multiline* field should take up on the form. The *Multiline count* property itself is a *Slider*, so the number of lines can be manually

entered, or the slider bar used. The minimum number of lines is 1, and the maximum is 19. The *Multiline* field that appears in **Figure 14-17** is set to the default of 9 lines.

Allow Zero Date Property (Date)
The *Allow zero date* property specifies whether 00/00/0000 is considered to be a valid date or not. If not, and the operator enters a date of 00/00/0000, the field will be cleared when the operator moves to another field on the form. This enables you to make the date field a mandatory field, even if there is a remote chance that the date will be unknown at the time of data entry. A check can also be made when saving the contents of the form, to see if the date field is empty without having to check if it contains zeros.

Disallow Null Field Property (Alpha, Numeric)
The *Disallow null field* property is only valid for *Alpha* and *Numeric* fields. It checks to see if the current field is blank (for Alpha fields) or zeros (for *Numeric* fields) as the operator leaves the field using the *Tab* key. **Figure 14-35** shows the error message that is displayed if one of these conditions is met.

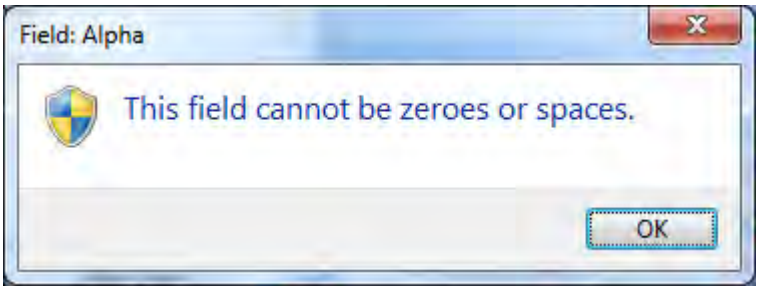


Figure 14-35: The error message displayed if tabbing off a field that disallows nulls

Cannot Remove Field Property (all field types)
The *Cannot remove field* property prevents an operator from removing a field from a form. The two ways that an operator can normally remove a field are by highlighting the field, right-clicking on the form and selecting *Hide Caption* from the context-sensitive menu, or by highlighting the field and using the *Shift+Del* shortcut keystrokes. When a field has the *Cannot remove field* property set against it, both of these options are disabled (see **Figure 14-36**).

Icon Property (all field types except Group heading)
The *Icon* property is used to place one of SYSPRO's standard icons to the left of the field's caption. Within the *Design Form* screen, when one of the fields (other than a *Group heading* field) is selected the hyperlink *Edit icon* appears against the *Icon* property. When this is selected, the *Change Icon* screen is displayed and you can select one of the icons from the list by double-clicking on it (or highlighting it and clicking on the *OK* button). The icon will appear alongside the field's caption on the

Form Design pane, and when this change is saved, alongside the field on the customized pane. **Figure 14-37** shows icons added to both the *Set Minimum Quantity* and *Stock code* fields.

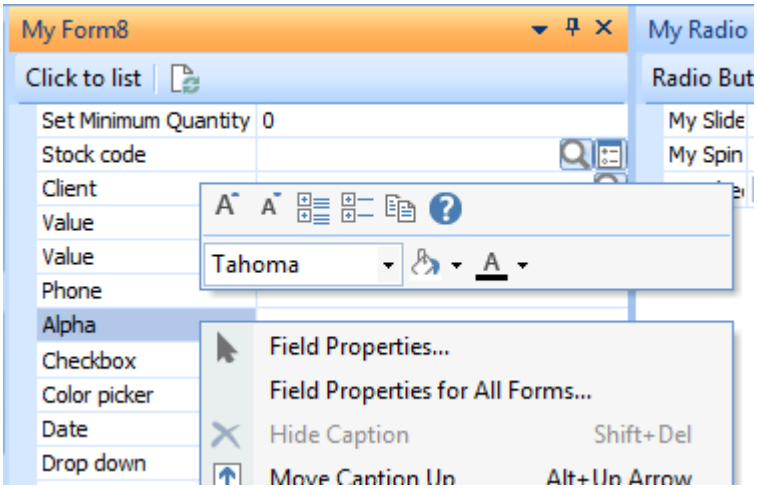


Figure 14-36: The *Hide Caption* option disabled when the *Cannot remove field* property is set

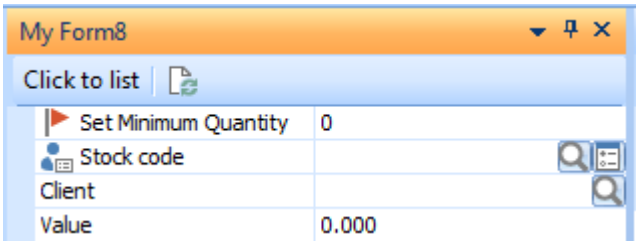


Figure 14-37: Icons against both the *Set Minimum Quantity* and *Stock code* fields

When an icon has already been added to a field, and the field is highlighted within the *Form Design* pane of the *Design Form* screen, the icon appears alongside the *Edit icon* hyperlink (see **Figure 14-38**).

Icons can be removed from fields by highlighting the field, selecting the *Edit icon* hyperlink, and choosing the *Cancel* button on the *Change Icon* screen (or just exiting the *Change Icon* screen without selecting an icon).

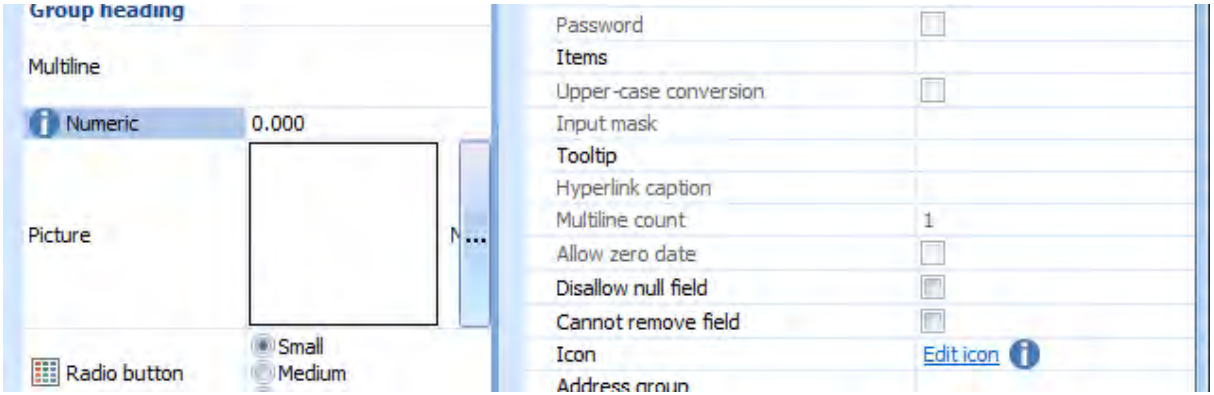


Figure 14-38: The icon appearing alongside the *Edit icon* hyperlink in the right-hand pane

Address Group Property (Alpha)
The *Address group* property is a one character field that is used to group the components of an address together, primarily so that the *Geolocation* function knows which fields to match up. If a form has more than one set of address fields, such as a delivery address and a billing address, when the *Resolve Geolocation* button is selected it needs to know which set of address fields it must use. It will use those that have the same *Address group* as it has.

Category Property (Alpha)
The *Category* property enables the different components of an address to be categorised, as well as adding client or server-side browses on fields or folders. The name of an address field may not accurately describe what it contains, or it may use a term that SYSPRO does not use. By setting one of the *Address category* property's options against a field, when the operator clicks on the *Resolve Geolocation* button, SYSPRO knows which field refers to the street, which field refers to the state/province, which refers to the country, etc.

Figure 14-39 shows a customized pane form that contains seven address fields. With their names being *Address1* to *Address7*, this doesn't make it easy to work out which contains the state/province.

Figure 14-40 shows the *Properties* pane of the *Design Form* screen. On the *Form Design* pane the *Address1* field has been selected. The *Address group* has been set to *S*, and the *Category* of *Street* is being selected. The same process is followed for all of the address fields (adding the *Address group* of *S*, and selecting the relevant *Category*). Field *Address7* is given a *Category* of *Geolocation*.

When the form is saved the fields appear with their captions matching their category (see **Figure 14-41**).

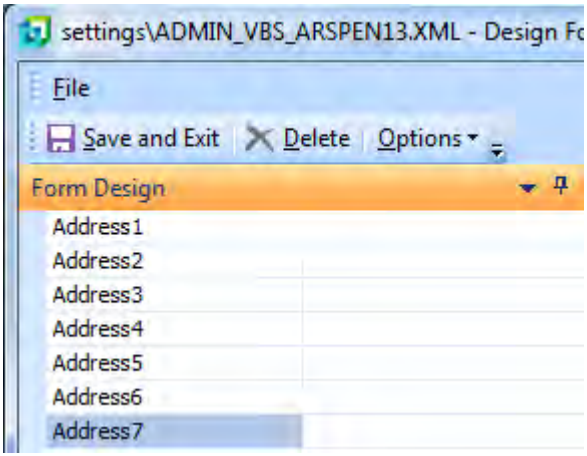


Figure 14-39: The seven address fields on the form

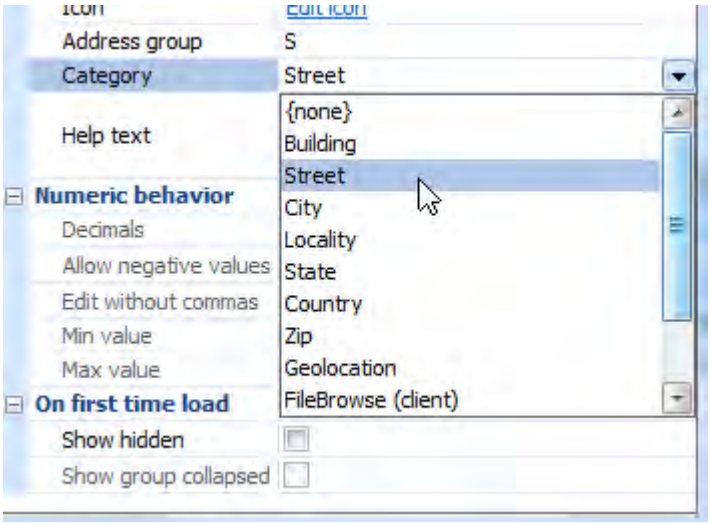


Figure 14-40: Setting the *Address group* to *S* and the *Category* to *Street* for the *Address1* field

When an address is entered, the operator can click on the *Resolve Geolocation* button (the one to the far right of the *Geolocation* field that contains the two down arrows). This can be seen in **Figure 14-42**.

When the *Resolve Geolocation* button is selected, SYSPRO attempts to resolve the address using fields against the form that match the *Address group* of the *Geolocation* field. A popup window will appear showing addresses that match the supplied information (see **Figure 14-43**).

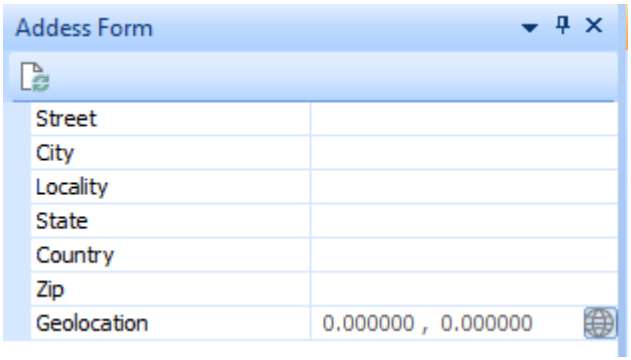


Figure 14-41: The *Address Form* with the field captions matching their categories

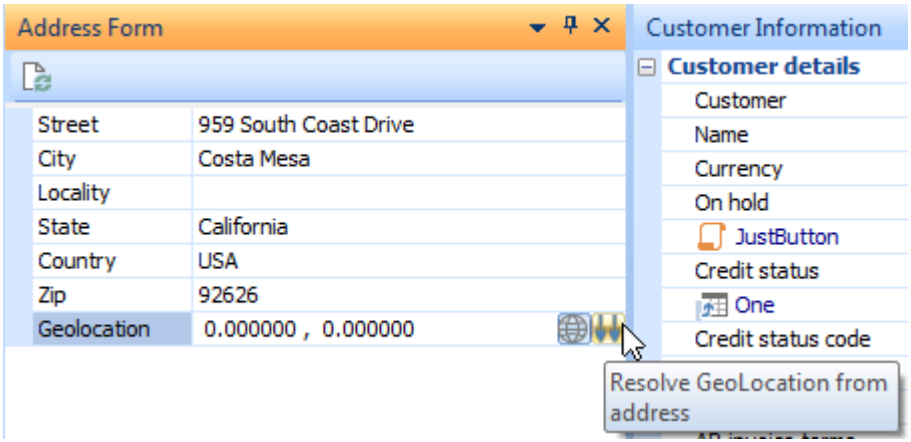


Figure 14-42: The completed address entry, with the *Resolve Geolocation* button

If the address entered is complete, only one entry will be included in the list. If the address entered is incomplete there may be many entries in the list and the operator must select the correct one.

When the operator clicks on the correct address from the list, the geolocation information of this address will be added to the *Geolocation* field and the address updated (see **Figure 14-44**).

At the bottom of the *Select the correct address* popup window that appears in **Figure 14-43** is the text *Short names in use (click to use long names)*. SYSPRO has a *System-wide Personalization* option that specifies whether the short name address format should be used for the address when the *Resolve Geolocation* button is selected, or the long name address (see **Figure 14-45**).

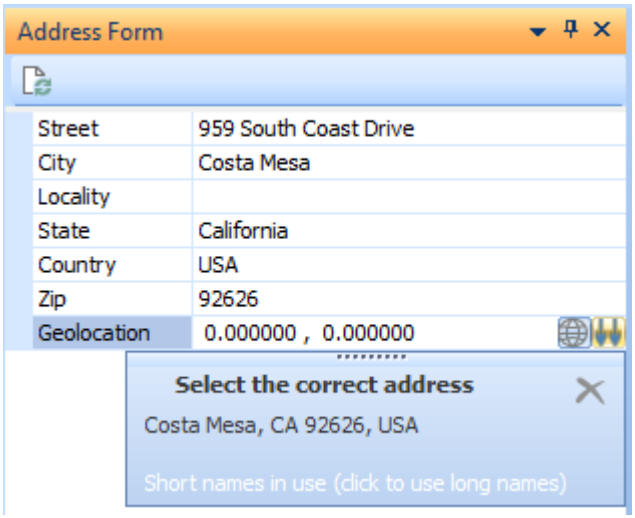


Figure 14-43: The address that matches the available information, being displayed in a popup

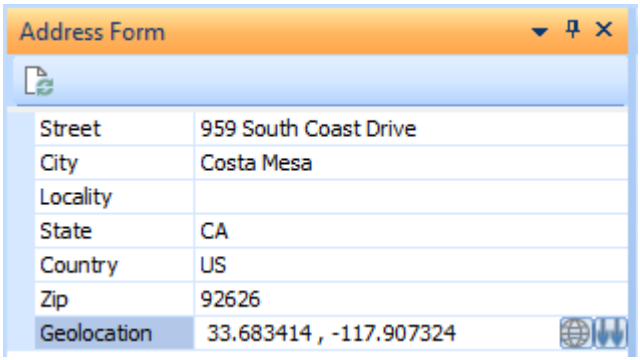


Figure 14-44: The geolocation information added to the *Geolocation* field and the address updated

The resolved address that appears in **Figure 14-44** is in the short address format. If the checkbox in **Figure 14-45** was not checked, when the operator clicked on the button to resolve the address, the text in the popup window would contain *Short names in use (click to use long names)*. If the operator clicked on the address in the popup window the long name address format would be used for the resolved address (see **Figure 14-46**).

Note the differences between the address in **Figure 14-44** and that in **Figure 14-46**. The *State* is either *CA* or *California*, and the *Country* is either *US* or *United States*.

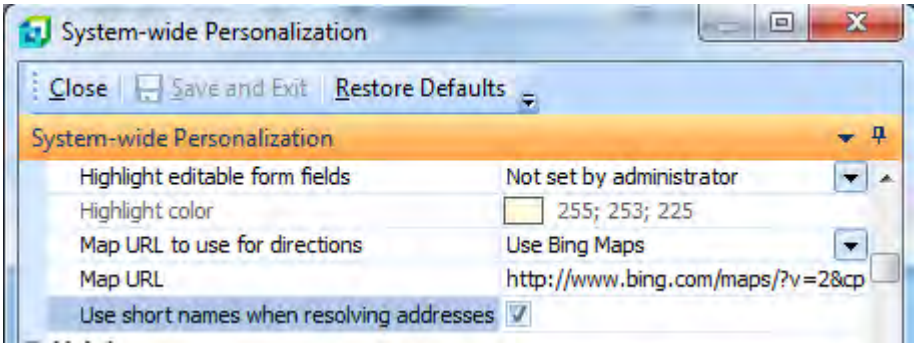


Figure 14-45: The *Use short names when resolving addresses* option

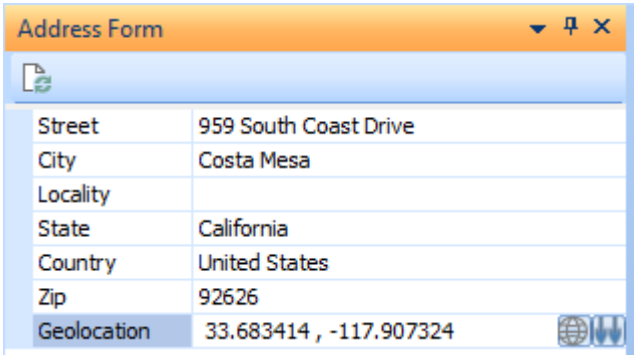


Figure 14-46: The resolved address in the long address format

Also against the *Geolocation* field is the *Map directions* button (see **Figure 14-47**).

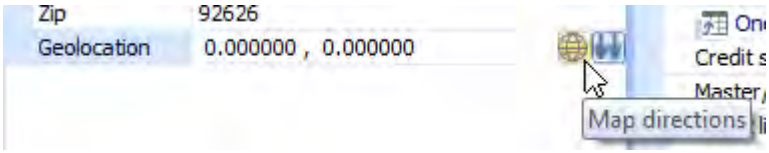


Figure 14-47: The Map directions option

When the operator clicks on the *Map directions* button, a map is displayed, with a location pointer highlighting the location provided by the geolocation values (see **Figure 14-48**).

You have a choice of which mapping software to use. This is configured within the *System-wide Personalization* program (*Ribbon bar* | *Administration* tab | *System-wide Personalization* | *Messages*

section). Against the *Map URL to use for directions* caption is a dropdown list containing the options *Use Google Maps*, *Use Bing Maps*, and *Use other*. **Figure 14-49** shows the dropdown list against the *Map URL to use for directions* field, and the default *Use Bing Maps* string.

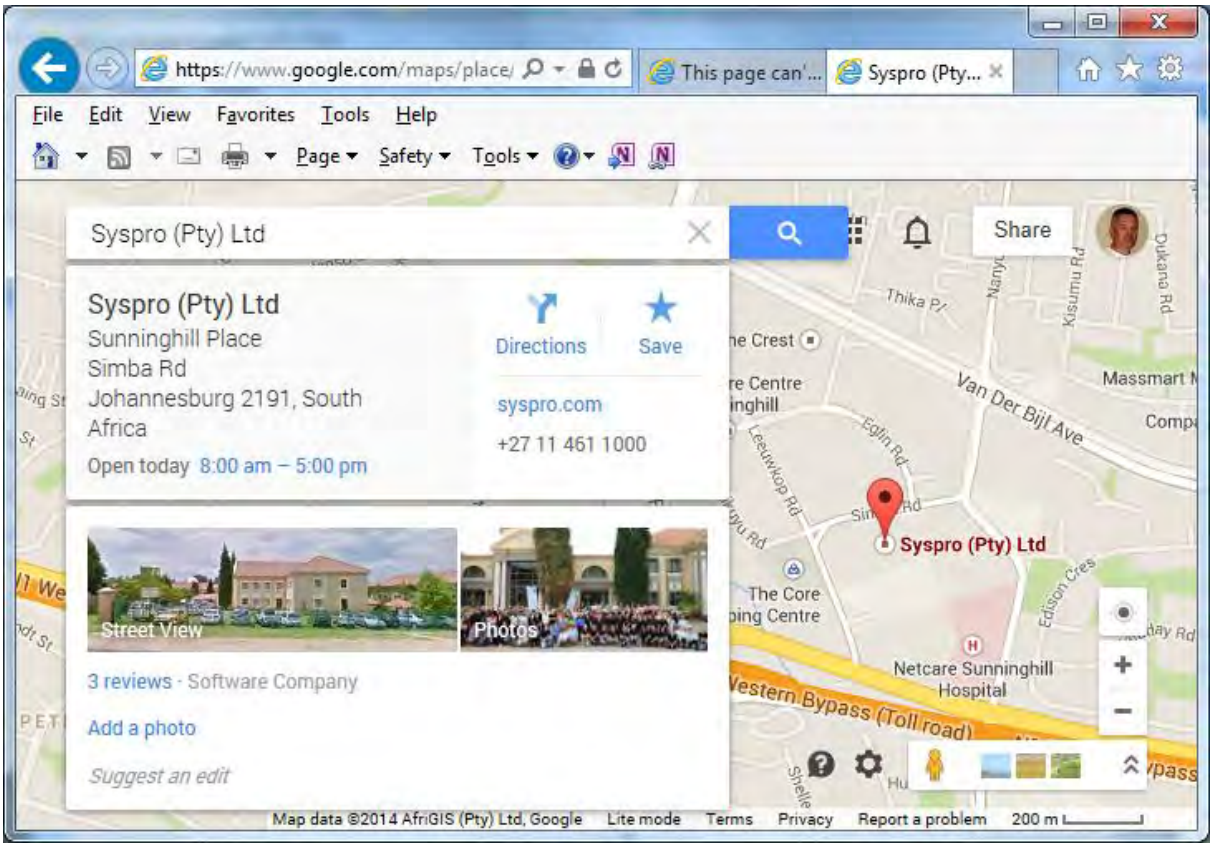


Figure 14-48: The map showing the location from the contents of the *Geolocation* field

If you select the *Use Google Maps* option the *Map URL* field will be populated with a string to call Google Maps. This string contains the variables *%latitute%* and *%longitude%*, which are replaced by the latitude and longitude values from the geolocation field at the time that the button is clicked.

Selecting the *Use Bing Maps* option will populate the *Map URL* field with the string to call Bing Maps, and this contains the same two variables.

Selecting the *Use other* option will leave the current string against the field, and you can change it to call the mapping software of your choice. The *%latitude%* and *%longitude%* variable can be used, and will be replaced with the latitude and longitude at run time.

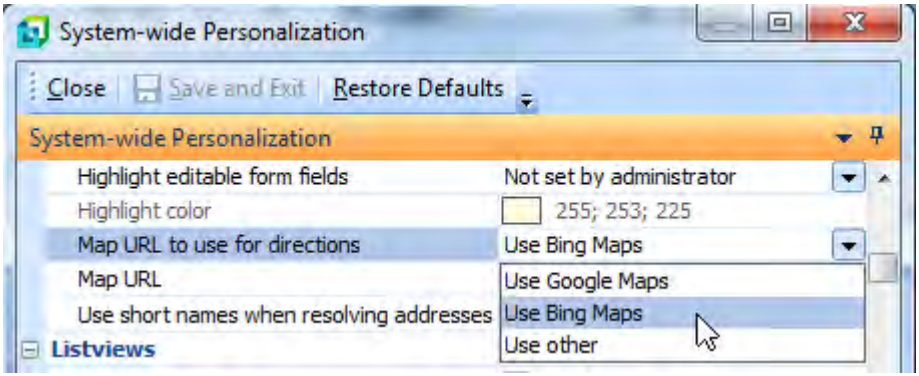


Figure 14-49: The *Map URL* and *Map URL to use for directions* options

Within the *SystemVariables* section of the variables pane is a *MapURL* variable. This variable contains the string against the *Map URL* field from the *System-wide Personalization* screen.

The *FileBrowse (client)*, *FileBrowse (server)*, *FolderBrowse (client)*, and *FolderBrowse (server)* category options add browses to the *Alpha* field. The *FileBrowse (client)* option creates a browse on the client machine that enables you to select a file. The *FileBrowse (server)* option creates a browse on the server machine that enables you to select a file. The *FolderBrowse (client)* option creates a browse on the client machine that enables you to select a folder. The *FolderBrowse (server)* option creates a browse on the server machine that enables you to select a folder.

Help Text Property (all field types except Group heading)
Text can be attached to each field (except for a *Group heading* field) to assist the operator in making the correct choice, or supplying the correct information. The *Help text* appears at the bottom of the pane if the *Help* option was selected for this pane. The help section is enabled for a pane by right-clicking on one of the captions on the pane, and clicking on the *Help* button (a white question mark on a blue circle). **Figure 14-50** shows how to enable the help section for the pane.

When the help section is enabled it appears at the bottom of the pane. **Figure 14-51** shows the help section enabled, but no text is present for the *Street* field.

Text is added to the *Help text* property within the *Properties* pane of the *Design Form* screen. Once this text is present for a field and help text is enabled for this pane, when the field is selected within the customized pane the help text will appear within the help section. **Figure 14-52** shows the help displayed for the selected field.

The default help that is displayed for a *Color picker* field is 0; 0; 0. However, if help text has been added against the *Help text* property for this field, the entered help text will be displayed.

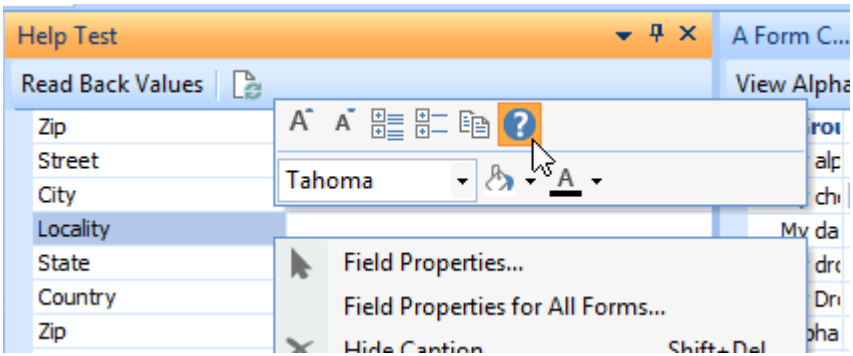


Figure 14-50: Enabling the pane help for this pane

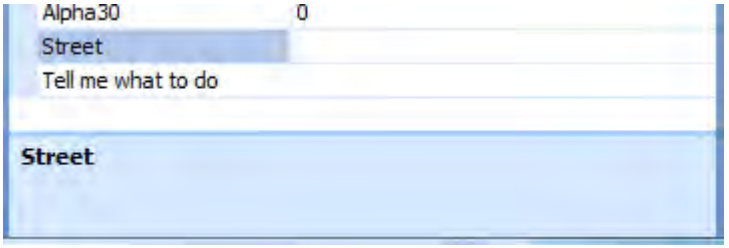


Figure 14-51: The help section enabled, but no content for the *Street* field

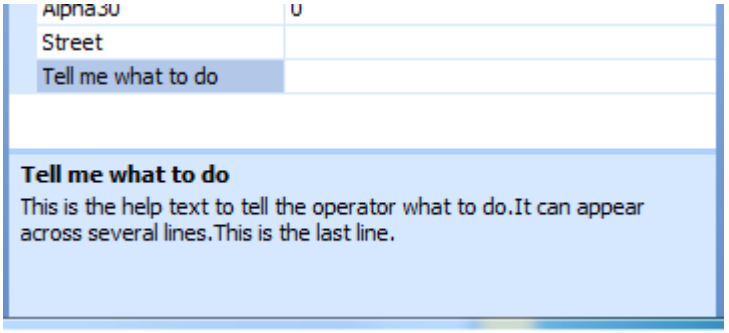


Figure 14-52: The help section displaying the help for the *Tell me what to do* field

The following properties appear within the *Numeric behaviour* group:

Decimals Property (Numeric)
The *Decimals* property is used in conjunction with the *Length* property for *Numeric* fields to specify the length of the field. The *Decimals* property specifies how many digits are to appear to the right of the decimal point. This is covered in detail within the *Numeric* section above.

Allow Negative Values Property (Numeric)
The *Allow negative values* property is used with *Numeric* fields to specify whether a negative value can be entered or not.

Edit Without Commas Property (Numeric)
The *Edit without commas* property specifies whether the displayed *Numeric* value must contain comma separators for the thousands or not. This not only affects the display of the value in the field on the form, it also affects how the value is displayed in a message box, and how the value is passed through to another customized pane if you pass it within the *RefreshValue* variable.

Min Value Property (Slider, Spin button)
The *Min value* property is only available with the *Slider* and *Spin button* field types. It is used to set the lowest value for the field. The lowest value that can be entered against the *Min value* property is zero.

Max Value Property (Slider, Spin button)
The *Max value* property is only available with the *Slider* and *Spin button* field types. It is used to set the highest value for the field. The highest value that can be entered is 99999.

The following properties appear within the *On first time load* group:

Show Hidden Property (all field types)
The *Show hidden* property is used to set the status of a field as the form is created for the first time. If checked against a field, this property hides the field when the form is created. After the form has been created this property is ignored, as the field's status will be controlled by the operator preferences (being set to the same status as this field had when the pane was last closed by this operator).

Show Group Collapsed Property (Group heading)
This property is only available for the *Group heading* field type. It defines the initial status of the group (expanded/collapsed) when the form is created. After initial creation, the group's status will be stored against the operator preferences file. The next time that they use this form the group's status will match the status when the operator last closed the form.

Using Form Properties within the Design Form Screen

Although the *Properties* pane of the *Design Form* screen enables you to set many properties against a field, there are some that are not available through this pane (such as setting the color of the font, and the background color of the field). However, these can be set using the *Field Properties* screen.

When using a form (as opposed to when designing one) the *Form Properties* pane can be launched by right-clicking on the required field and selecting *Field Properties* from the context-sensitive menu. The *Field Properties* screen is covered in detail on page 83 of the first Power Tailoring book, so will not be covered here.

When designing a form the *Field Properties* screen can be loaded by right-clicking on the field in the *Form Design* pane and selecting *Field Properties* from the context-sensitive menu. Note that many of the properties within the *Field Properties* screen are the same as those in the *Properties* pane.

Figure 14-53 shows the *Design Form* screen where the operator has selected the field *ABC* from the *Form Design* pane, right-clicked on it, and selected *Field Properties*. It can be seen that several of the properties in the *Field Properties* screen (such as *Background color*, *Foreground color*, and *Font*) do not appear within the *Properties* pane of the *Design Form* screen.

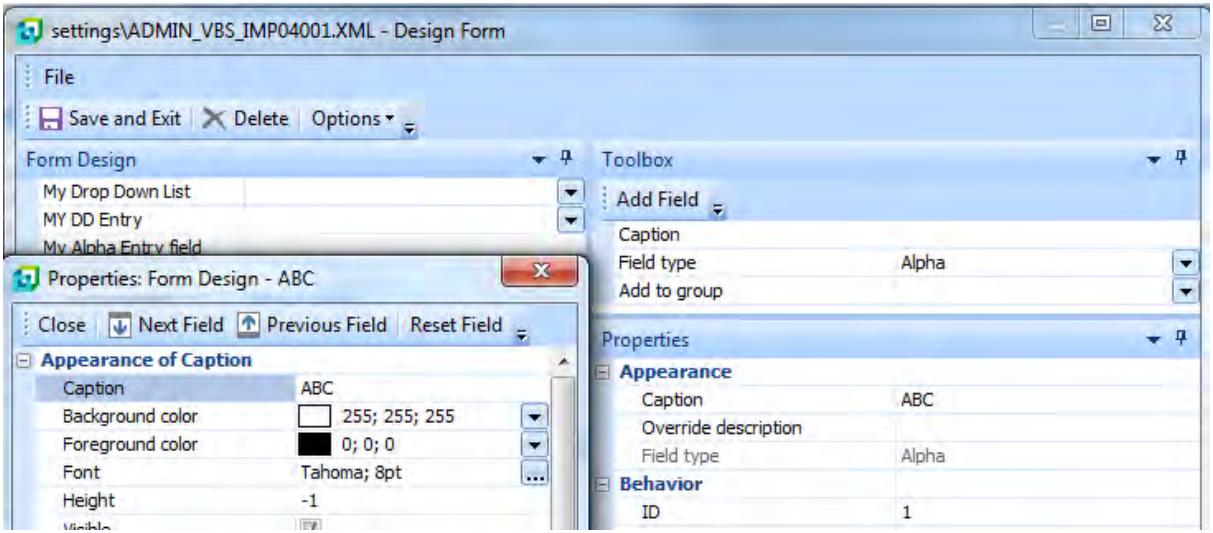


Figure 14-53: The *Design Form* pane and *Field Properties* screen

Building a Form Programmatically

A basic form can be built programmatically using XML, which enables you to build it dynamically, based on other criteria. A form can only be built once per run of the program. The form is typically built within the customized pane's *OnLoad* function, although it could be built within the *OnRefresh* function, the first time that it fires.

When a form is built, the XML has a root element of *Form*, a parent element of *Fields*, and one or more *Field* elements that contain the individual fields. This will appear similar to that below.

Once the XML has been built it is passed to the *CustomizedPane.CodeObject.FormProperties* variable. If this is a very small form it would be possible to perform this in one line of code. However, it is more likely that the XML would be built up in a variable, and the contents of this variable passed to this *FormProperties* variable.

```
<Form>
  <Fields>
    <Field Name='Customer' Type='A' />"
              ......
    <Field Name='Invoice Value' Type='N' Decimals='2' />"
  <Fields>
<Form>
```

The following line of code (that has wrapped around on this page) is used to create two fields on the form, an alpha one called *Customer*, and a numeric one with two decimals call *Value*. Even though this is a very simple form, adding it as one line directly to the *FormProperties* variable makes it difficult to read, and even more so to modify.

```
CustomizedPane.CodeObject.FormProperties = "<Form><Fields><Field Name='Customer'
Type='A' /><Field Name='Value' Type='N' Decimals='2' /></Fields></Form>"
```

The following example creates that same form, but builds the XML using a variable called *FormProps*, then passes the contents of this variable to the *FormProperties* variable. As each *Field* element exists on its own line it becomes much easier to follow what is going on, as well as making it easier to add/remove lines. Adding indentations to the XML makes it even easier to follow the structure.

```
Dim FormProps
FormProps =             "<Form>"
Formprops = FormProps & "  <Fields>"
Formprops = FormProps & "    <Field Name='Customer' Type='A' />"
Formprops = FormProps & "    <Field Name='Value' Type='N' Decimals='2' />"
Formprops = FormProps & "  </Fields>"
Formprops = FormProps & "</Form>"
CustomizedPane.CodeObject.FormProperties = FormProps
```

The field types that can be programmatically added to a form are:

- Alpha
- Checkbox
- Color picker
- Date
- Drop down
- Drop down entry
- Font
- Group heading
- Numeric
- Picture

Alpha Field
To add an Alpha field to a form, the *Type* attribute is set to A. The value of the field can also be prepopulated with text using the *Value* attribute.

```
<Field Name='Customer' Type='A' Value='000000000000001'/>
```

Checkbox Field
To add a *Checkbox* field to a form, the *Type* attribute is set to B. The value of the checkbox can be set to checked or unchecked using the *Value* attribute. Setting the value to 1 causes the checkbox to be checked, and setting it to 0 causes it to be unchecked.

```
<Field Name='Tax required' Type='B' Value='1'/>
```

Color Picker Field
To add a *Color picker* field to a form, the *Type* attribute is set to E. The color can be set by passing the decimal value of the color in the *Value* attribute. The way to calculate the decimal value of a color from its RGB color was covered in the *Color Picker* sub-section of the *Design Form Screen - Toolbox Pane* section above.

```
<Field Name='Background Color' Type='E' Value='16751052'/>
```

Date Field
To add a *Date* field to a form, the *Type* attribute is set to D. The field can be prepopulated a date using the *Value* attribute. The format of the date to be supplied is CCYY-MM-DD.

```
<Field Name='Date of joining' Type='D' Value='2014-10-30'/>
```

Drop Down Field
To add a *Drop down* field to a form, the *Type* attribute is set to L. The items to appear in the dropdown list appear again the *List* attribute, delimited by semi-colons. The field can also be prepopulated with one of these items by supplying it in the *Value* attribute.

```
<Field Name='MyList' Type='L' List='Small;Medium;Large' Value='Large' />
```

If a value is supplied that does not appear within the list attribute, it will appear as the value against the field. However, as soon as an item is selected from the dropdown list this value will no longer be available.

Within the *Drop Down* subsection of the *Design Form Screen - Toolbox Pane* section above, the topic of adding icons against the items in the dropdown list was covered. This facility is also available when building the form using XML. Also covered in the same subsection was using strings other than the dropdown value as the returned value. This is also available when building a form using XML.

The following line of code (that has wrapped around on this page) builds up a dropdown list that contains the options of *Small*, *Medium*, and *Large*. These have colored block icons against them (represented by the numbers 266, 267, and 268 in square brackets immediately in front of each item). The values in square brackets immediately behind each item are the values that will be returned if the item is selected. These values can also be supplied to prepopulate the field with the corresponding item from the dropdown list. In this code the value of l (a lowercase L) is supplied so the field is prepopulated with the *Large* option. **Figure 14-54** shows the *Drop down* built using this XML.

```
<Field Name='MyList' Type='L' List='[266]Small[s];[267]Medium[m];[268]Large[l]'
Value='l' />
```

The restriction of having only 20 items in the *Drop down* list when adding entries via the *Design Form* screen does not apply when the list is built programmatically (see section *Adding more than 20 Items to a Dropdown List against a Form Field* in Chapter 21 for more information).

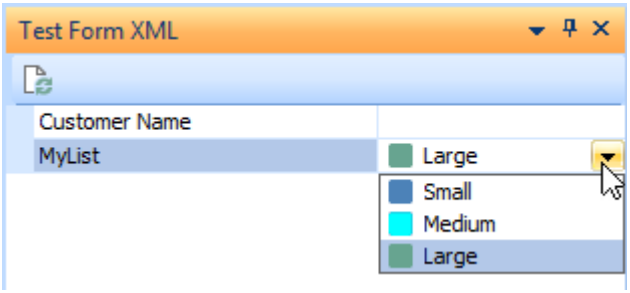


Figure 14-54: A dropdown list built using XML

Drop Down Entry Field

A *Drop down entry* field is added to a form using the *Type* of K. All the other options are the same as a *Drop Down* field. The restriction of having only 20 items in the *Drop down entry* list when adding entries via the *Design Form* screen does not apply when the list is built programmatically.

Font Field
To add a *Font* field to a form, the *Type* attribute is set to F. The field can also be prepopulated with font information using the *Value* attribute.

```
<Field Name='Customer' Type='F' Value='Arial;12'/>
```

Group Heading Field
A *Group heading* field is added to a form using the *Type* of H. This creates the group heading, but does not specify which other fields belong within it. If the next item in the XML contains the attribute *Group='1'* it will be included in the group. Each successive item with the *Group* attribute set to 1 will be included, until the first one that does not have this set. All items from this one onwards will not be included in the group.

In the following example a group called *Customer information* contains an alpha field called *Customer*, and a checkbox called *Tax required*. The date field called *Date of joining* falls outside of the group because it does not have the attribute of *Group='1'*.

```
<Field Name='Customer information' Type='H'/>
<Field Name='Customer' Type='A' Value='000000000000001' Group='1' />"
<Field Name='Tax required' Type='B' Value='1' Group='1' />
<Field Name='Date of joining' Type='D' Value='2014-10-30'/>
```

Numeric Field
To add a *Numeric* field to a form the *Type* attribute is set to N. The number of decimals can be supplied using the *Decimals* attribute. The field can also be prepopulated with a value using the *Value* attribute.

```
<Field Name='Invoice value' Type='N' Decimals='2' Value='123.45' />
```

Picture Field
To add a *Picture* field to a form the *Type* attribute is set to I (a capital "i"), and the pictures name and location is supplied in the *Value* attribute.

```
<Field Name='The vista' Type='I' Value='\\PaulH\Images\A100.jpg' />"
```

Importing a Form Programmatically

As well as being able to build the XML programmatically and pass it to the *FormProperties* variable, it is possible to use an existing file containing XML to create your form. In the following example, an XML file called *MyForm*`.txt` exists in a share. The filename/location of this file is passed to the *FormProperties* variable, in this case within the pane's *OnLoad* function.

```
Function CustomizedPane_OnLoad()
  CustomizedPane.CodeObject.FormProperties = "\\PaulH\Notes\MyForm.txt"
End Function
```

The `MyForm.txt` file contains the following XML:

```
<Form>
 <Fields>
  <Field Name='Customer Name' Type='A' />
  <Field Name='Invoice value' Type='N' Decimals='2' Value='123.45' />
  <Field Name='MyList' Type='L' List='[266]Small;[267]Medium;[268]Large' />
  <Field Name='CC' Type='B' />
  <Field Name='Contact details' Type='H' />
  <Field Name='Contact' Group='1' Type='A' />
  <Field Name='Telephone number' Group='1' Type='A' />
  <Field Name='Telephone extension' Group='1' Type='A' />
  <Field Name='Additional telephone number' Group='1' Type='A' />
  <Field Name='Email address' Group='1' Type='A' />
  <Field Name='Sold to address' Type='H' />
  <Field Name='Sold to address line 1' Group='1' Type='A' />
  <Field Name='Sold to address line 2' Group='1' Type='A' />
  <Field Name='Sold to address line 3' Group='1' Type='A' />
  <Field Name='Sold to address line 4' Group='1' Type='A' />
  <Field Name='Sold to address line 5' Group='1' Type='A' />
  <Field Name='Sold to address postal code' Group='1' Type='A' />
 </Fields>
</Form>
```

When the customized pane's *OnLoad* function is invoked the content of this file is passed to the variable and the form is rendered.

Accessing the Content of a Form

Each field on a form displays two parts: a caption, and a value. You can programmatically access these as well as the field's ID (that was added when the field was created) using VBScript. The current content and structure of the form is available within a two-dimensional array called *FormValues*. Accessing the content of arrays using VBScript has been covered in numerous places within this book, so I will just reiterate that both the row and columns start at zero.

Figure 14-55 shows a customized pane form called *Basic Form Access*. This contains three fields, and each field has a caption, and each has a value against it.

The array is addressed by supplying the *Column* and *Row* number of the cell that you want to access. The captions are in column zero, and the values are in column one. In addition to the two columns that you can see, there is a third one containing the ID that was added to the field when it was created. The ID is in column two.

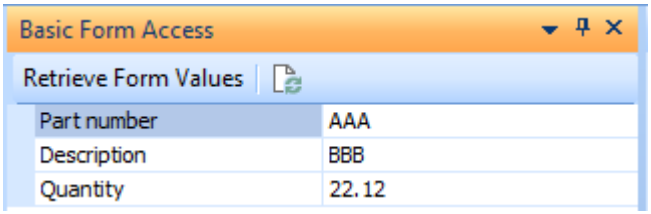


Figure 14-55: The *Basic Form Access* form

The cell containing the value AAA (which is column 1, row 0) can be accessed using the following line of code, which will display its value in a message box:

```
Msgbox CustomizedPane.CodeObject.FormValues(1,0)
```

If you do not know the row of the field containing the item that you want to access, but you do know the field's ID, you can iterate through the form looking for this ID, then extract the information that you need.

In the example in **Figure 14-56**, the *Quantity* caption has been renamed to *Price* at some point. Any code that used the caption of *Quantity* to retrieve the information from this field would fail to locate it, as there is no field with a caption of *Quantity*.

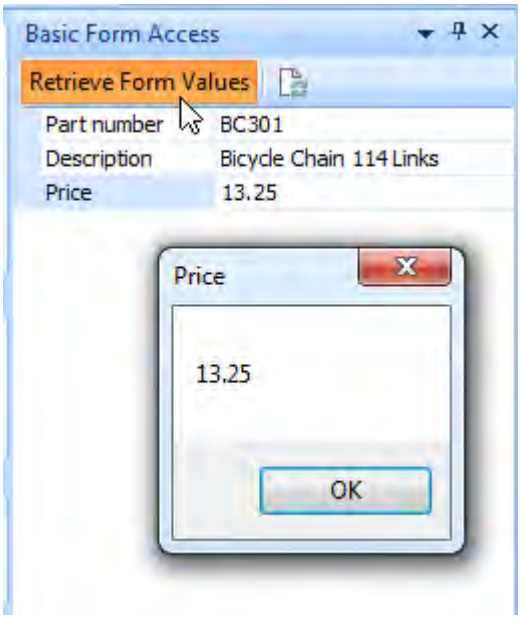


Figure 14-56: Using the field's ID to retrieve the value

However, changing the field's caption does not change its ID, so this can still be used. If the value against the *Price* field needs to be retrieved so that it can be used in another calculation, VBScript code can be used to iterate through all the lines of the form and check to see if its ID matches, and then extract the value. This was done in **Figure 14-56** and the value displayed in a message box.

Below is the code that was added to the customized pane's *OnToolbarButton1Clicked* function. At the top of the customized pane are two toolbar controls. In this case *Toolbar control 1* has been set as a button that contains the text *Retrieve Form Values*. When this button is clicked the *OnToolbarButton1Clicked* event will be fired and any code against the corresponding function will be actioned.

The code against this function appears below. It performs the following steps:

- Calculates how many rows are on the form
- Runs through each row extracting the *Caption*, *Value* and ID number
- Performs a check to see if the ID matches the number three
- If this check fails it goes to the next row
- If this check succeeds it displays the *Value*, and the *Caption* as the message box title

```
Dim RowCount, NumberOfRows, myCaption, myValue, myID
NumberOfRows = uBound(CustomizedPane.CodeObject.FormValues,2) -1

For RowCount = 0 to NumberOfRows
  myCaption = CustomizedPane.CodeObject.FormValues(0,RowCount)
  myValue = CustomizedPane.CodeObject.FormValues(1,RowCount)
  myID = CustomizedPane.CodeObject.FormValues(2,RowCount)

  If myID = 3 then
    msgbox myValue,,myCaption
  End If
Next
```

The first line of code creates the five variables that will be used in the script.

```
Dim RowCount, NumberOfRows, myCaption, myValue, myID
```

The second line of code uses the VBScript *UBound* function to locate the upper boundary of the row dimension of the *FormValues* array (dimension 2). It then subtracts one from this as the number of rows starts at one, whereas the rows in an array start at zero. It stores this in a variable called *NumberOfRows*. In this example the upper boundary is three, so subtracting one from this gives the highest array entry as two.

```
NumberOfRows = uBound(CustomizedPane.CodeObject.FormValues,2) -1
```

The next line of code starts a *For*/*Next* loop using a count called *RowCount*. This runs from zero to the number stored in the *NumberOfRows* variable (in this example, two). The *For*/*Next* loop processes everything between the *FOR* and *NEXT* statements for each row of the form.

```
For RowCount = 0 to NumberOfRows
```

The next three lines of code extract the *Caption*, *Value*, and ID for the current row and store them in variables.

```
myCaption = CustomizedPane.CodeObject.FormValues(0,RowCount)
myValue = CustomizedPane.CodeObject.FormValues(1,RowCount)
myID = CustomizedPane.CodeObject.FormValues(2,RowCount)
```

The next three lines of code test the contents of the *myID* variable to see if it contains the number three. If it does contain the number three, it displays a message box with the contents of the *myValue* variable, with a message box title using the contents of *myCaption*.

```
If myID = 3 then
  msgbox myValue,,myCaption
End If
```

The last line of code causes processing to start at the *FOR* statement again.

```
Next
```

Writing Values to a Form

Values can be written back to the form using two different methods. The first is by supplying values to individual variables, and the second is by building up XML containing all the values, and passing them to the *UpdateFormValues* variable.

Each customized pane has a section in the *Variables* pane, and this section contains a variable for each field on the form. **Figure 14-57** shows the section within the *Variables* pane for the *Basic Form Access* form (spaces and special characters are removed from these section names).This is the same form that appears in **Figure 14-56**.

Double-clicking on one of these variables will add its full name to the VBScript. In the case of the *Description* variable, the full name would be *BasicFormAccess.CodeObject.Description*. The following line of code will update the Description field with the text *Bicycle Chain 114 Links*.

```
BasicFormAccess.CodeObject.Description = "Bicycle Chain 114 Links"
```

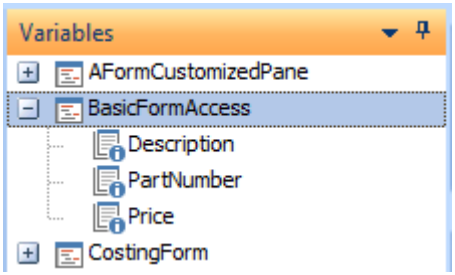


Figure 14-57: The variables associated with the *Basic Form Access* form

Each field that requires updating must be processed in the same way.

The form's values can be updated using XML, which only needs to contain the fields that must be updated. The field can be specified using either the field's caption, or its ID. Once the XML has been built it is passed to the *UpdateFormValues* variable. The following lines of code change the value against the *Price* field using the field's ID, and the *Part number* field using the field's caption.

```
Dim MyValues

MyValues = "<Form><Fields>"
MyValues = MyValues & "<Field ID='3' Value='123.45' />"
MyValues = MyValues & "<Field Name='Part number' Value='BC305' />"
MyValues = MyValues & "</Fields></Form>"

CustomizedPane.CodeObject.UpdateFormValues = MyValues
```

Field Properties

The field properties available within the *Field Properties* pane work with customized pane form fields in the same way as they work with the fields of an ordinary form. These include setting fields to read-only/read write, hiding/showing fields, etc.

Variables Specific to Form Customized Panes

There are four variables under the *CustomizedPane* section of the *Variables* pane that are specific to *Form* customized panes. These are *FormDesignName*, *FormProperties*, *FormValues*, and *UpdateFormValues*. Three of these variables have been covered to some degree earlier in this chapter, so will not be covered in great detail here.

FormValues (read-only)
Each field on a form displays two parts: a caption and a value. You can programmatically access these, as well as the ID that was added when the field was created, using VBScript. The current content and structure of the form is available to view within a two-dimensional array called *FormValues*. Accessing the content of arrays using VBScript has been covered in numerous places within this book, so I will just mention that both the row and columns start at zero.

The array is addressed by supplying the *Column* and *Row* number of the cell that you want to access. The captions are in column zero, and the values are in column one. In addition to the two columns that you can see on a form, there is a third one containing the ID that was added to the field when it was created. The ID is in column two.

Double-clicking on the *FormValues* variable name within the *CustomizedPane* section of the *Variables* pane will insert the full name of the *FormValues* variable into your script. To be able to access a specific entry within the array you need to provide the column number (less one) and the row number (less one) within parentheses, with a comma separating the values. The following example would enable you to view the second column (which will contain the values) of the seventh row of the form.

```
Msgbox CustomizedPane.CodeObject.FormValues(1,6)
```

UpdateFormValues (write-only)
A form's values can be updated using XML, which only needs to contain the fields that must be updated. The field can be specified using either the field's caption, or its ID. Once the XML has been built it is passed to the *UpdateFormValues* variable. The following lines of code change the value against the *Price* field using the field's ID, and the *Part number* field using the field's caption.

```
Dim MyValues

MyValues = "<Form><Fields>"
MyValues = MyValues & "<Field ID='3' Value='123.45' />"
MyValues = MyValues & "<Field Name='Part number' Value='BC305' />"
MyValues = MyValues & "</Fields></Form>"

CustomizedPane.CodeObject.UpdateFormValues = MyValues
```

FormProperties (write-only)
A form does not need to be created using the *Design Form* screen. The form can be built using XML within the script (see the section *Building a Form Programmatically* above). Alternatively, an XML file containing the form's structure can be read in as the customized pane is loaded, each time that it is loaded (see the section *Importing a Form Programmatically* above).

FormDesignName (read-only)
When a customized pane is created the VBScript code is stored in a file. If the customized pane is for a role, the file will be stored in a folder under the SYSPRO `Base\Settings` folder that contains the role's three digit number. For example, if the role's number is 13, the location will be

`Base\Settings\Role_013`. The name of the file will be the program's name followed by the next available two digit number for this program, with no suffix. If this is the first customized pane for the *Customer Query* program the name would be *ARSPEN00* (see **Figure 14-58**).

Where the customized pane is for an operator (instead of a role) the location is in the `Base\Settings` folder. The name of the file consists of the operator code, followed by underscore, VBS, underscore, the program's name and the next available two digit number for this program, with no suffix. If there are already 13 customized panes for the *Customer Query* program (for the *ADMIN* operator) the file will be called `ADMIN_VBS_ARSPEN14` (see **Figure 14-59**).

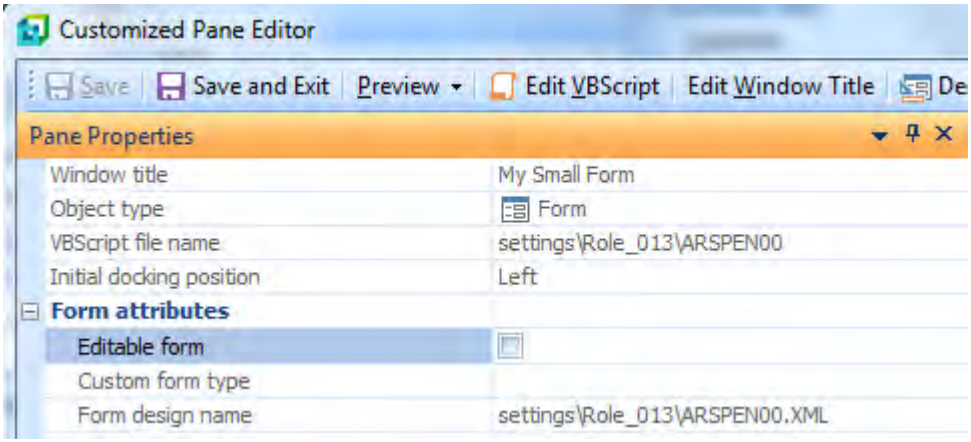


Figure 14-58: *Pane Properties* for a role showing both *VBScript file name* and *Form design name*

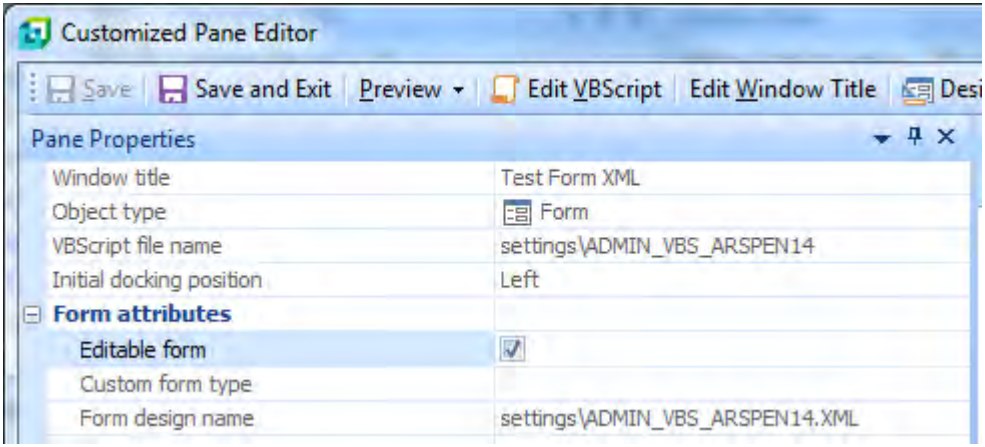


Figure 14-59: *Pane Properties* for an operator showing *VBScript file name and Form design name*

The name of the file and its location appear on the customized pane's *Pane Properties* screen against the *VBScript file name* caption. When a *Form* customized pane is created, a file with the same name (with a suffix of XML) appears against the *Form design name* caption. If you use the *Design Form* screen to build your form, this is the name of the file that will hold the form's structure. An example of the *Form design name* for a role can be seen in **Figure 14-58**, and the *Form design name* for an operator can be seen in **Figure 14-59**.

The *FormDesignName* variable makes available the value against the *Form design name* field. **Figure 14-60** shows the contents of this variable being displayed using a message box statement.
Note that the name of the script can be accessed using the *System Variable* called *ScriptName*, although this is just the name of the script without a location.

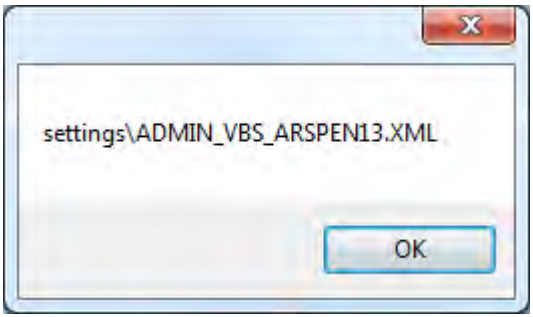


Figure 14-60: The contents of the *FormDesignName* variable

Using Design Form and Using XML to Build a Form

As has been covered above, the form can be built in one of three ways:

- Designing it in the *Design Form* screen
- Building it using XML and supplying it to the *FormProperties* variable
- Importing it from an XML file and supplying it to the *FormProperties* variable.

However, there may be times where the form has been built up using XML within the pane's *OnLoad* function (and supplied to the *FormProperties* variable) and at a later date someone attempts to modify the form using the *Design Form* screen. If they save their form, when the form is next loaded it will use the form created using the *Design Form* screen. It is not able to combine the two.

The same is true if your form is being imported from an XML file and someone attempts to modify the form using the *Design Form* screen. If they save their form, the next time that the form is loaded, only the form created using the *Design Form* screen will be displayed.

Resetting the Contents of a Form

The content of a form can be reset (or cleared out) using VBScript code. This is done by passing a space character to the *UpdateFormValues* variable. This variable is found in the *CustomizedPane* section within the *Variables* pane. In the sample code below, a checkbox has been added to the customized pane toolbar using *Toolbar control 1*. When the customized pane is refreshed, a check is made to see the status of this checkbox and the form cleared if it is checked.

```
Function CustomizedPane_OnRefresh()
  If CustomizedPane.CodeObject.ButtonValue1 = 1 then
    CustomizedPane.CodeObject.UpdateFormValues = " "
  Else
    <-- Add the code here to populate the form -->
  End If
End Function
```

Creating a Picture Browse

When a *Picture* field is added to a customized pane *Form*, a browse is present so that you can locate the image that will permanently appear against this field. However, you may need to allow an operator to select a picture for this field using a browse at run time. This can be done using the *Button wording* field property.

First of all, add the *Picture* field to the form, but do not populate it with a picture. Exit and save your changes back to the program containing the customized pane, then edit the customized pane again. If you do not exit the customized pane the new field will not appear in the list of fields against this customized pane.

Edit the customized pane's *OnLoad* function in the VBScript, and place the cursor on a blank line immediately before the *End Function* statement. Locate the section within the Variables pane related to this customized pane, and then locate the picture field.

In **Figure 14-61** the customized pane is called *Test of XML*, so the name of the section is *TestofXML*, (spaces and special characters are removed). The picture field is called *MyPicture*. Once this field has been located, double-click it to insert the full name of this variable into your code. Follow this with a space, and equals sign, and another space.

Next locate the *Button wording* field property within the *Field Properties* pane, and add the wording that must appear on your button (see **Figure 14-62**).

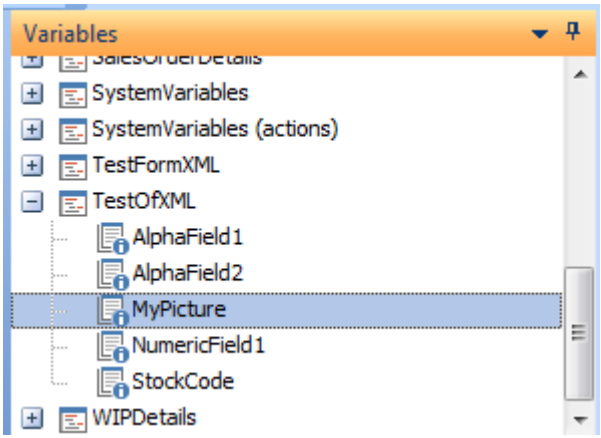


Figure 14-61: Locating the picture field within the customized pane section

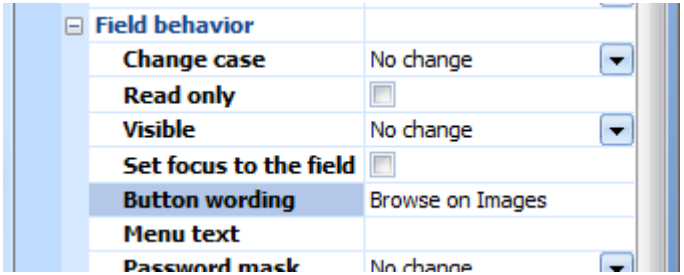


Figure 14-62: Adding the wording to appear on the button

Click on the *Insert VBScript Code* button, and the code to create the button will be added for you. It should look similar to the line of code below, which has wrapped around on this page.

```
TestOfXML.CodeObject.MyPicture = "<Field><Button>Browse on Images</Button>
</Field>"
```

Close the customized pane editor, and close the SYSPRO program that contains the customized pane. When you next run the program, the picture field will have a browse against it that will enable you to select an image to be used. **Figure 14-63** shows the browse button alongside the empty image containing. The browse button will always be the same height as the image holder.

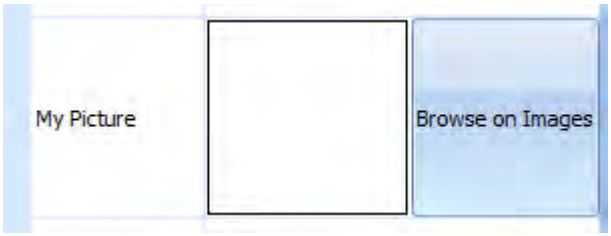


Figure 14-63: The *Browse for Images* button alongside the unpopulated image container

Create Editable Form

The *Create Editable Form* option is available from the *Menu* button of a pane (*Menu* | *Customized Pane* | *Create Editable Form*). This can be seen in **Figure 14-64**.

This creates an empty customized pane *Form* with the *Editable form* flag checked (see **Figure 14-65**). You can edit the customized pane later to add the fields to the form using the *Design Form* option and/or edit the VBScript, and add the required code to the VBScript functions.

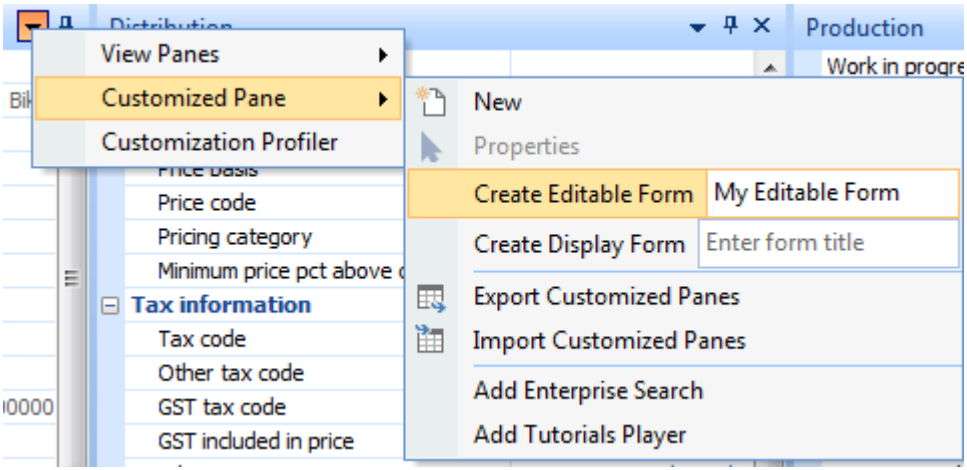


Figure 14-64: Adding an empty editable customized pane *Form*

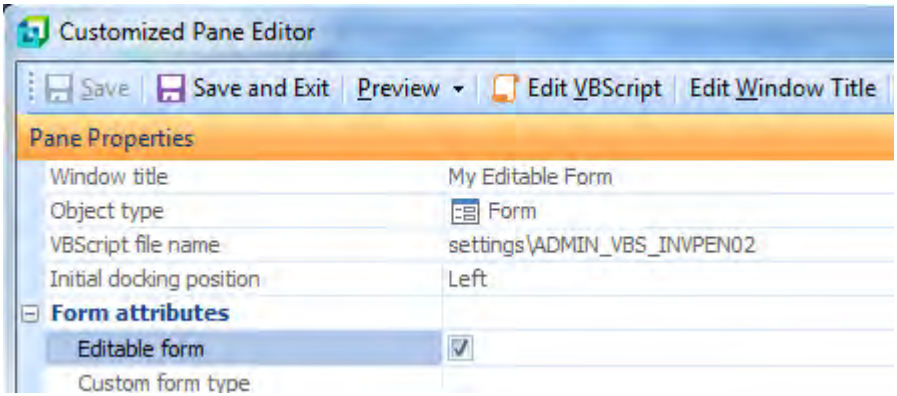


Figure 14-65: The empty customized pane *Form* with the *Editable form* option checked

Create Display Form

The *Create Display Form* option works in a similar way to the *Create Editable Form* except that the *Editable form* checkbox is unchecked. If the *Create Display Form* was used to create the form, when it is edited the *Editable form* checkbox can be checked making it an editable form.

Warehouse Address Customized Pane from Chapter 8

Within Chapter 8 (that covers *Listviews* and *Data grids*) there is an example that uses a customized pane *Form* called *Warehouse Address*. This mentions that the creation of the customized pane is covered in Chapter 14. Most of what you need to know to create the customized pane has been covered early in this chapter, or in Chapter 10. The additional information appears below.

The customized pane's form was designed using the *Design Form* option, and consists of seven fields (*Warehouse name*, *Address Lines 1 – 5*, and *Postal Code*). This can be seen in **Figure 14-66**.

As covered in Chapter 8, when the operator clicks on one of the rows in the *Warehouse Values* listview its *OnClicked* event is fired. The VBScript code against this passes the first column of value return (which contains the warehouse code) to the *WarehouseAddress* customized pane. This causes the *Warehouse Address* customized pane's *OnRefresh* event to fire and the warehouse code is available within the customized pane's *RefreshValue* variable.

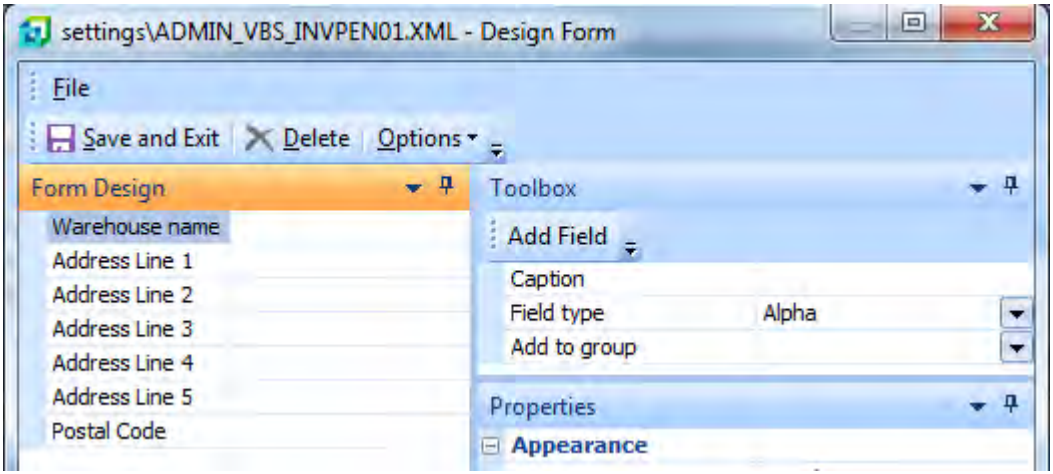


Figure 14-66: The *Design Form* screen showing the seven fields

The single line of code to perform this appears below, although it has wrapped around on this page. Note that it is worth adding the same code to the listview's *OnDblClick* macro event in case the operator double-clicks on the row by mistake.

```
CustomizedPanes.CodeObject.WarehouseAddress =
WhValuesForStockCode.CodeObject.Array(000,0)
```

The only macro event that is used in the customized pane is *OnRefresh*. The completed VBScript code that appears against the customized pane's *OnRefresh* function appears below. Each section of code has been commented, so no further explanation should be required.

```
' Create the variables to be used in the script.
Dim XMLParam, XMLOut, XMLDOMDocument, WH, WHDesc, FormData
Dim Addr1, Addr2, Addr3, Addr4, Addr5, PostCode

' Populate the WH variable with the contents of the RefreshValue variable.
' that was passed through by the listview
WH = CustomizedPane.CodeObject.RefreshValue

' Perform a check to make sure that the warehouse code received is not the value
' Combined, which means all of the warehouses added together, which wouldn't
' have an address. If it is Combined, exit the function as no processing is
' required.

If WH = "Combined" then
  exit function
End If

' Build the XML to be passed to the COMFCH business object. This specifies to use
```

```
' the Inventory Warehouse Control table (InvWhControl) and the warehouse code
' passed through from the listview must be used as the key.
XMLParam = XMLParam & " <Fetch>"
XMLParam = XMLParam & "   <TableName>InvWhControl</TableName>"
XMLParam = XMLParam & "   <Key>" & WH & "</Key>"
XMLParam = XMLParam & " </Fetch>"
on error resume next

' Call the COMFCH business object using the above XML, and place the result in
' the XMLOut variable
XMLOut = CallBO("COMFCH",XMLParam,"auto")

' Exit the function if the business object failed.
if err then
  msgbox err.Description, vBCritical, "Calling Business Object"
  exit function
end if
' Switch on error handling
on error goto 0

' Load the output from the business object into the Document Object Model to
' make it easier to process.
Set XMLDOMDocument = createobject("MSXML2.DOMDocument")
XMLDOMDocument.async = false
XMLDOMDocument.LoadXML(XMLOut)

' Extract the contents of the various address nodes and place these values
' into the relevant variables.
WHDesc = XMLDOMDocument.SelectSingleNode("//InvWhControl/Description").Text
Addr1 = XMLDOMDocument.SelectSingleNode("//InvWhControl/DeliveryAddr1").Text
Addr2 = XMLDOMDocument.SelectSingleNode("//InvWhControl/DeliveryAddr2").Text
Addr3 = XMLDOMDocument.SelectSingleNode("//InvWhControl/DeliveryAddr3").Text
Addr4 = XMLDOMDocument.SelectSingleNode("//InvWhControl/DeliveryAddr4").Text
Addr5 = XMLDOMDocument.SelectSingleNode("//InvWhControl/DeliveryAddr5").Text
PostCode = XMLDOMDocument.SelectSingleNode("//InvWhControl/PostalCode").Text

' Build up the XML to populate the form fields
FormData = "<Form><Fields>"
FormData = FormData & "<Field ID='1' Value='" & WHDesc & "' />"
FormData = FormData & "<Field ID='2' Value='" & Addr1 & "' />"
FormData = FormData & "<Field ID='3' Value='" & Addr2 & "' />"
FormData = FormData & "<Field ID='4' Value='" & Addr3 & "' />"
FormData = FormData & "<Field ID='5' Value='" & Addr4 & "' />"
FormData = FormData & "<Field ID='6' Value='" & Addr5 & "' />"
FormData = FormData & "<Field ID='7' Value='" & PostCode & "' />"
FormData = FormData & "</Fields></Form>"

' Pass the XML to the UpdateFormValues variable so that SYSPRO can populate
' the form with the values.
CustomizedPane.CodeObject.UpdateFormValues = FormData
```

Figure 14-67 shows the completed form. The operator has selected warehouse N in the listview and the address is populated in the customized pane

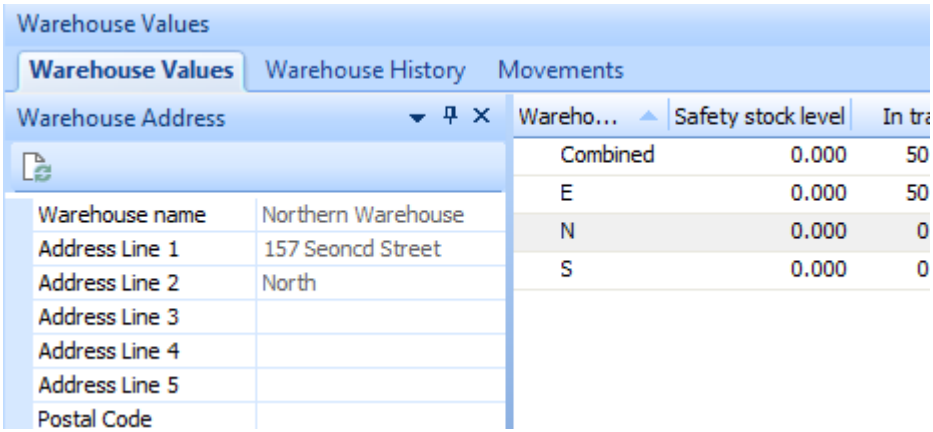


Figure 14-67: The completed form being populated with the address for warehouse N

Chapter 15 - Search Windows

The *Search Window* object type creates a customized pane consisting of two panes. The upper pane is where you define the search criteria, and the lower pane is where the results of the search are displayed. The *Search Windows* were originally designed to complement/replace the standard SYSPRO *Browses*.

There are sixteen standard search windows that ship with SYSPRO, and these have been written as customized panes so they can also be modified. **Figure 15-1** shows the screen containing the list of search windows, which consists of five columns.

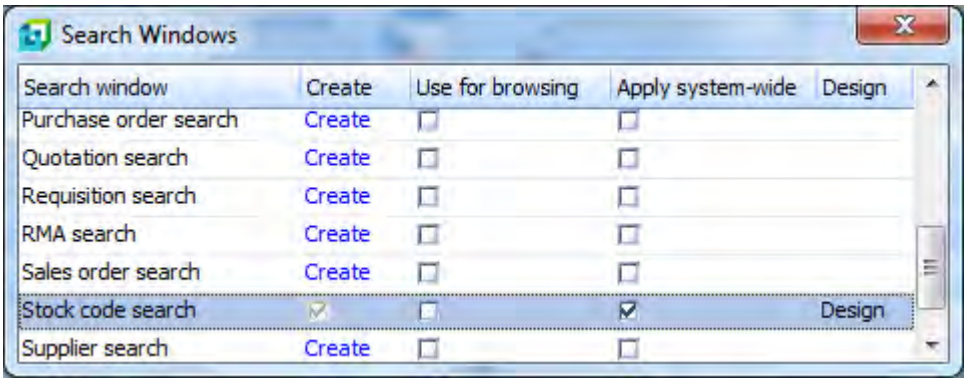


Figure 15-1: The *Search Windows* screen where you specify which search windows are in use

The first column contains the name of the search window.

The second column contains a *Create* hyperlink that creates the standard search windows, or contains a checked checkbox if the search window has already been created.

Some search windows also have a checkbox in the *Use for browsing* column. If this is checked and the search window created, when a browse is performed for this key field type the *Search Window* is displayed in place of the standard browse. It is possible to create your own search window customized panes (see the section *Building Your Own Search Window* below) and adding it to the *Search Windows* pane (see the section *Adding Your Search Window to the Search Windows List* below).

The fourth column contains the *Apply system-wide* checkbox. This makes the search window the preferred browse for all users (see the *System-wide Search Windows* section below).

The fifth column will contain the *Design* hyperlink if the *Apply system-wide* checkbox is checked. This allows you to specify which columns must be included in the output listview for all users. When this option is selected a screen is displayed containing a *Save Layout* and a *Cancel Changes* button. You manipulate the *Search Results* columns (as covered in the *Adding Columns to the Search Results Listview* section of the *Search Results* section below) and then click on the *Save layout* or *Cancel Changes* button.

System-wide Search Windows

Checking the checkbox in the *Apply system-wide* column against one of the searches replaces the standard browse with this search for all operators. When this is checked for any search, the *Design* hyperlink appears.

The *Apply system-wide* column will be empty if the operator does not have the *Main Menu – Allow to create system-wide search windows* activity checked (*Ribbon bar* | *Setup* tab | *Operators* | select operator | *Edit* | *Change* | *Security* tab | *Activities* section) (see **Figure 15-2**).

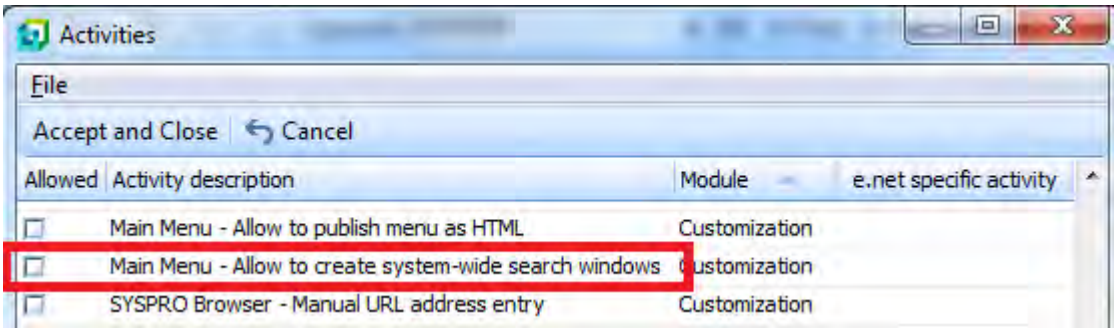


Figure 15-2: The *Main Menu – Allow to create system-wide search windows* option

Any custom search windows that you have created and added to the `CUSSCH.IMP` file will appear in this list. If the operator has permission to check the *Apply system-wide* checkbox, this will also be available for your custom search windows.

If you check the *Apply system-wide* checkbox against a search window without clicking on the *Create* hyperlink and *Use for browsing* checkbox, the next time that you (or any other operator) logs in it will be as if these had been done. Opening the *Search Windows* pane will show that these have been selected.

Removing a System-wide Search Window

You cannot delete a system-wide search window, even if you are the operator that made it system-wide. If you attempt to delete a system-wide search window you will receive the error message that appears in **Figure 15-3**.

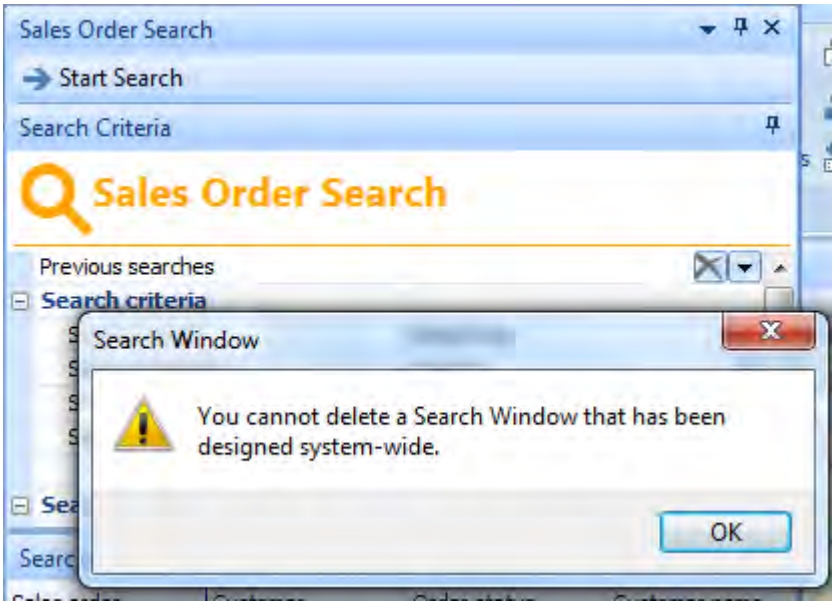


Figure 15-3: The error message when attempting to delete a system-wide search window

When a search window is configured as system-wide the search window becomes configured against each operator. If you uncheck the *Apply system-wide* option it is no longer flagged as system-wide, but still exists against each operator. Operators can now remove it from their environment if they no longer want it.

Designing System-wide Search Windows

The *Design* hyperlink enables you to design the output listview of the search window. If this is the first time that you have designed the layout for this search window, the *Designing Search Window* screen will be displayed and the *Save Layout* and *Cancel Changes* buttons will be enabled (see **Figure 15-4**).

Existing columns in the *Search Results* listview can be moved to other positions within the listview by dragging their column heading to the new location. They can also be removed by dragging their column heading from the heading section.

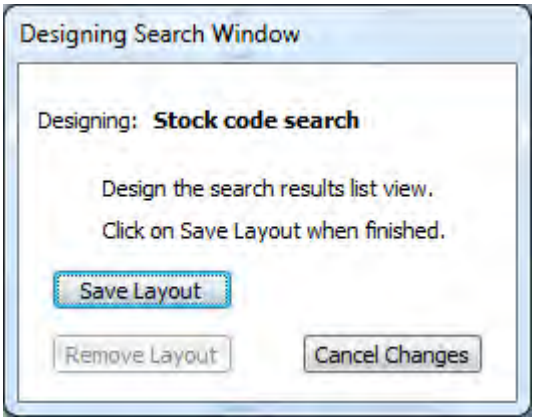


Figure 15-4: The initial *Designing Search Windows* screen

The *Field Chooser* is displayed by right-clicking on the column header and selecting *Field Chooser* from the displayed menu (see **Figure 15-5**). Any columns that have previously been removed will also appear in the *Field Chooser*.

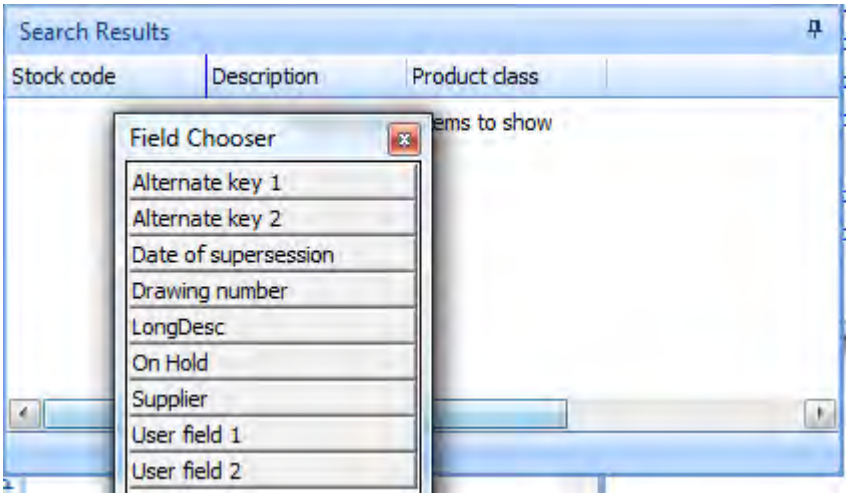


Figure 15-5: The *Field Chooser* screen

New columns can be added by dragging them from the *Field Chooser* to the listview's column heading (see **Figure 15-6**). When the column name is released it will be added to the listview and removed from the *Field Chooser*.

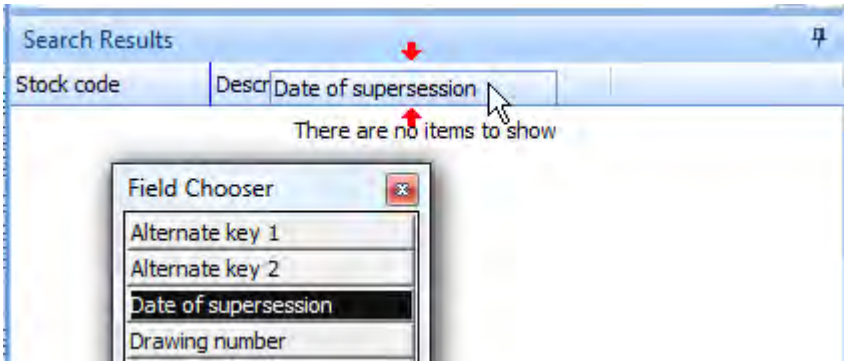


Figure 15-6: Adding the *Date of supersession* column to the listview

Both custom form fields and master table fields can also be added to the listview. This is covered later in this chapter, as well as in the following sections of chapter 8 on listviews and data grids:

- Adding Custom Form Fields to a Listview
- Adding Master Table Fields to a Listview

When you have completed your changes you can save them using the *Save Layout* button on the *Designing Search Windows* screen, or discard them using the *Cancel Changes* button. In both cases the *Designing Search Window* screen will be closed and the *Search Window* will remain.

Using the *Design* hyperlink for a second (or subsequent time) for the same search window will display the *Designing Search Window* screen and all three buttons will be enabled (previously the *Remove Layout* button was disabled because there was no layout to remove). The *Remove Layout* button is used to remove the layout for this search window type (see **Figure 15-7**).

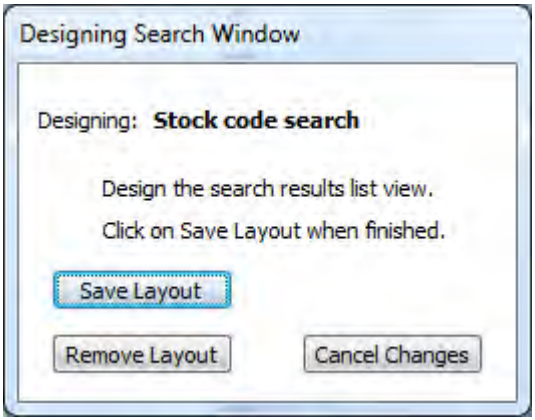


Figure 15-7: The *Designing Search Window* screen when a layout is already present

Even though the system-wide search window has had a layout designed for it, each operator can adjust his or her personal view of the listview. However, if the administrator uses the *Design* option to make a change, the new design will replace any changes made by individual operators.

Using Search Windows

In **Figure 15-8** the *Stock code search* has already been created. If its *Use for browsing* checkbox is checked, when a stock code browse is performed, the *Stock Code Search* window will appear in place of the standard stock code browse program.

It does not matter if the stock code browse is a standard one, or because a field with the same name has been added to a customized pane form; if the *Use for browsing* checkbox is checked and the *Stock Code Search* window has been created, the *Search Window* will be called.

Figure 15-8 shows the standard *Stock Code Search* search window. The upper pane of the search window is titled *Search Criteria*, and the lower pane is titled *Search Results*. Within the *Search Criteria* pane you build up the *Search statement* to be used using the fields above it. All of the fields in the *Search Criteria* section of the *Search Criteria* pane have dropdown lists associated with them.

In the case of the *Stock Code Search* the *Search column* dropdown list contains the following fields:

- StockCode
- Description
- LongDesc
- ProductClass
- Supplier
- StockOnHold
- AlternateKey1
- AlternateKey2
- UserField1
- UserField2
- SupercessionDate
- DrawOfficeNum

The *Search operator* dropdown list contains the following options:

- Equal to
- Is not equal to
- Greater than
- Greater than or equal to
- Less than
- Less than or equal to
- Contains
- Starts with

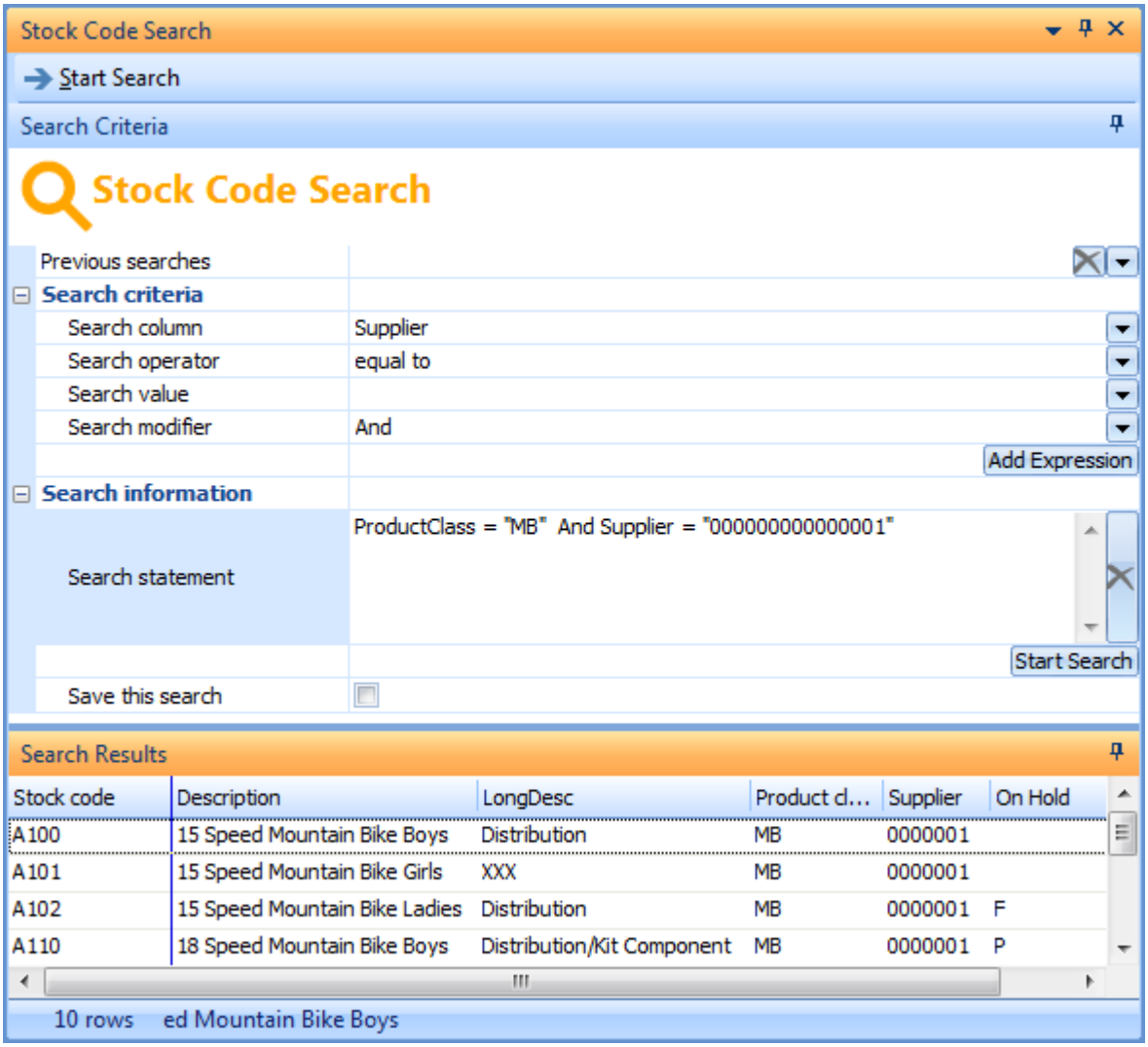


Figure 15-8: The standard *Stock Code Search*

The *Search value* dropdown list just contains the option *{today}*, so you can manually enter a value to be compared with the field added to the *Search column* field, or you can compare it to today's date.

If this is the first *Expression* of a long *Search statement* that you are building (or the only *Expression* in the *Search statement*) you click on the *Add Expression* button to add this expression to the *Search statement* section of the pane. If this is the second or subsequent *Expression* you must choose either

the *And* or *Or* options from the dropdown list against the *Search modifier* field before clicking the *Add Expression* button to add this expression to the existing *Search statement*. The example in **Figure 15-8** has two expressions in the *Search statement*, and the *Search modifier* of *And* was used to join them together.

If you know what must appear as the *Search statement* you can manually enter it in the *Search statement* field. If you have already entered a string against the *Search statement*, you can also modify it manually.

If checked, the *Save this search* checkbox will add this expression to a list of previously built *Search statements* when the search is run. The previous searches can be seen using the dropdown list against the *Previous searches* option. Clicking on one of these will replace the current *Search statement* with the selected one. If you no longer need one of the previous searches it can be removed from this list. This is performed by selecting it from the *Previous sear*ches dropdown and clicking on the X alongside this field; at which point it will be removed from the list.

The *Start search* button is used to start the search using your *Search statement*. Only items that match the criteria in the statement will be returned, and these populate the *Search Results* listview.

Search Operators

As mentioned above, there are eight *Search operators* that are used to perform comparisons.

Equal To
The *Equal to* search operator is used to perform exact comparisons, such as the product class must be equal to the string *MB*. In this case only the stock codes that have a product class of *MB* will be included in the result. If there is a product class of *MBS*, the items with this product class against them will not be included.

The *Search value* must be provided in double quotes, as per the example below. This is for both alphanumeric and numeric values.

```
ProductClass = "MB"
```

Is Not Equal To
The *Is not equal to* search operator works in a similar way to the *Equal to* search operator; it just excludes the items that are matched from the returned results. The *Search value* must be provided in double quotes, as per the example below.

```
ProductClass != "MB"
```

Greater Than

The *Greater than* search operator is used to return items that have a value greater than the supplied *Search value*. The comparison can be performed for numeric and alphanumeric values. For example, the following *Search statement* is from the *Job Search* and will return all the jobs where the *Quantity to make* is greater than 1.

```
QtyToMake > "1"
```

An example of the results from this search can be seen in **Figure 15-9**. The last row in the listview contains a job where the *Quantity to make* is 1.001. This line is included because 1.001 is greater than 1. Another example using an alphanumeric field (in this case the stock code key is configured to be alphanumeric) appears below. Here the comparison is that the stock code must be greater than A112, so the results would contain stock code A113 and above.

```
StockCode > "A112"
```

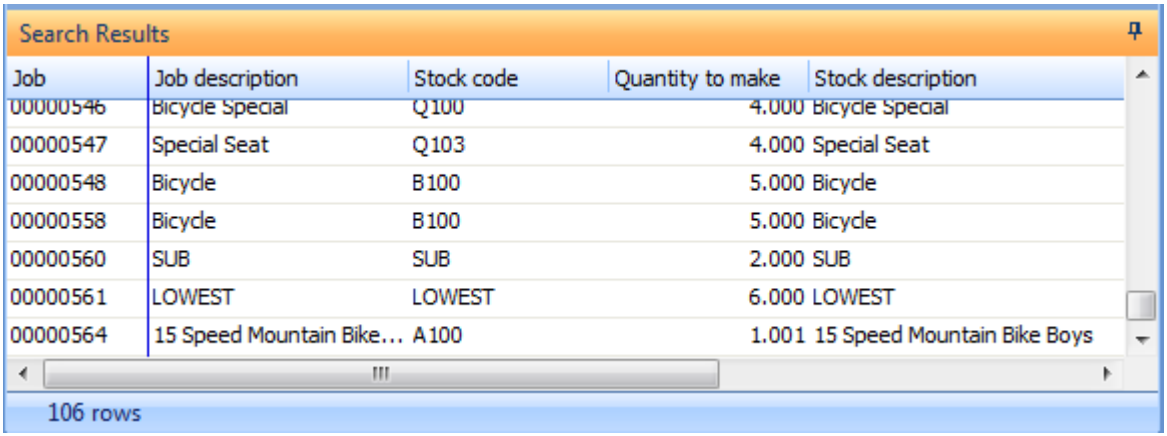


Figure 15-9: The *Search Results* including a row with a *QtyToMake* of 1.001

Greater Than Or Equal To

The *Greater than or equal to* search operator is used to find values that are either the same as the supplied *Search value*, or greater than it. Using similar examples to those used in the *Greater than* section above, the *Quantity to make* search would return all jobs that had a 1.000 or above in the *QtyToMake* field. The *Stock Code* search would return stock code A112 (if it exists) and all the stock codes above this.

```
QtyToMake >= "1"

StockCode >= "A112"
```

Less Than
The *Less than* search operator is used to return items that have a value less than the supplied *Search value*. The comparison can be performed for numeric and alphanumeric values. For example, the following *Search statement* is from the *Job Search* and will return all the jobs where the *Quantity to make* is less than ten.

```
QtyToMake < "10"
```

Less Than or Equal To
The *Less than or equal to* search operator is used to find values that are either the same as the supplied *Search value*, or less than it. The *Quantity to make* example below would return jobs with a *Quantity to make* of 10.000 or less.

```
QtyToMake <= "10"
```

Contains
The *Contains* search operator is used to find rows that contain the supplied *Search value* anywhere within the search column. The following is a *Search statement* from the *Stock Code Search* that will return all stock codes that contain the string *cycle* anywhere within the stock description, in any combination of case.

```
Description LIKE "*cycle*"
```

This is similar to the *Like* operator in Transact SQL used by SQL Server, using a percent sign either side of the value, as per the Transact SQL example below:

```
Where Description Like '%cycle%'
```

There are a few differences to this Transact SQL example. The first is that SYSPRO uses asterisks instead of percent signs as its wildcard symbol. The second is that SYSPRO uses double quotes instead of single quotes. The third is that the SYSPRO query is not case-sensitive, so *Cycle*, *cycle*, and *CYCLE* will all return the same results.

Starts With
The *Starts with* search operator is used to find rows that contain the supplied *Search value* at the beginning of the search column. This is similar to the *Contains* search operator, except that the string must appear at the very beginning of the field. Also like the *Contains* search operator, the query is case-insensitive. The following example will find all rows where the stock description starts with the string bicycle, in any combination of case.

```
Description LIKE "biCYCLE*"
```

Search Results

The *Search Results* pane is a listview like any other, so the normal listview functionality is available, such as sequencing the result by a column, filtering, grouping, etc. When each of the standard *Search Window* customized panes was developed, the developer picked the fields that were most likely to be required and specified these to be returned by the business object, and made these available as default columns in the listview.

A second group of fields were selected that may be required, and these were specified to be returned by the business object, but do not appear as listview columns by default. This is because of the width available within the listview. These columns are available within the *Field Chooser* screen (from the context-sensitive menu that appears when you right-click on one of the column headers), and can be dragged onto the listview. They will then populate like any other column. **Figure 15-10** shows the *Alternate key 1* field being dragged from the *Field Chooser* screen to the listview. When the operator drops the field it will be added to this position within the listview, and removed from the *Field Chooser*.

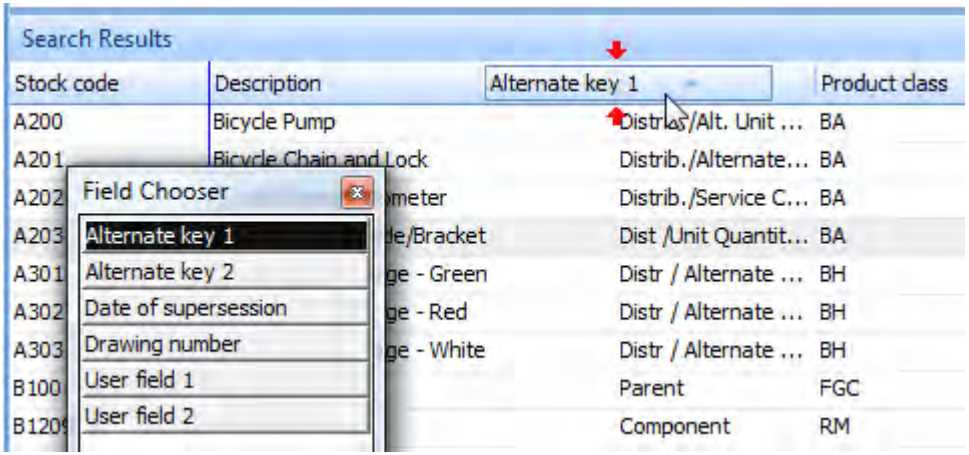


Figure 15-10: Dragging the *Alternate key 1* field from the *Field Chooser* to the listview

Adding Columns to the Search Results Listview

The standard *Search Window* customized panes use the ***COMFND*** business object. This business object is different to most of the others in that you specify which columns must be returned. There is a slight performance overhead when more columns must be returned by this business object, which is why only the specified fields are returned. The time taken is insignificant if the number of rows returned by the search is less than about 500. However, when returning a large number of rows this can add up to many seconds.

The business object returns XML, and once this XML has been received by the customized pane it must be unpacked and the listview populated. This is an additional overhead that can become significant when dealing with a large number of rows/columns.

Restricting the number of columns returned and displayed was one way of improving the performance of the *Search Windows*. Another was adding an element to the input XML for this business object to change its output to that of the listview's native data format. This significantly reduces the time that the listview takes to display when there are many thousands of rows, and lots of columns. The element name is *NoXML*, and it can be seen in the following extract of the XML supplied to ***COMFND*** for the *Stock Code Search*. Note that this element is specific to the ***COMFND*** business object. It will not change the output format of any other business object if supplied.

```
<Query>
  <TableName>InvMaster</TableName>
  <NoXml>1</NoXml>
  <ReturnRows>999999</ReturnRows>
  <Columns>
    <Column>StockCode</Column>
    <Column>Description</Column>
```

Because the results from the business object the native format for the listview, the columns specified to be returned by the business object must exactly match the columns specified for the listview. Not only must there be the same number of columns in both, but the sequence of the columns must be the same. If this is not done the information displayed in a column might not match the column's heading.

Within the VBScript code, many additional columns have been added to the listview structure and the XML input to the business object. These additional columns have been commented out with an apostrophe so that they don't get actioned. To add one of these columns it is a simple a case of removing the apostrophe against both the listview configuration, and the XML input to the business object. The next time that the customized pane is loaded the column will appear and be populated.

It is also possible to apply apostrophes to the beginning of both the listview columns, and XML input, for fields that are displayed by default but you will not use. However, this will only make a difference if the search returns a significant number of results.

Figure 15-11 shows an extract from the *Stock Code Search*. There are six columns against the listview structure that have been commented out.

Figure 15-12 shows the same additional columns against the input XML for the ***COMFND*** business object. These commented out fields appear in the VBScript to make it easier to add more fields to the listview. Simply removing the apostrophe from the beginning of the line of code for both the structure and XML input lines will add the column and display the value the next time that the search window is loaded.

```
ListXML = ListXML & "<Column Name='DrawOfficeNum' Description='Drawing number' Hidden='tru
'  ListXML = ListXML & "<Column Name='UserField3' Description='User field 3' />"
'  ListXML = ListXML & "<Column Name='UserField4' Description='User field 4' />"
'  ListXML = ListXML & "<Column Name='UserField5' Description='User field 5' />"
'  ListXML = ListXML & "<Column Name='ClearingFlag' Description='Clearing stock flag' />"
'  ListXML = ListXML & "<Column Name='TraceableType' Description='Traceable' />"
'  ListXML = ListXML & "<Column Name='WarehouseToUse' Description='Warehouse to use' />"
ListXML = ListXML & "</Columns>"
CustomizedPane.CodeObject.ListviewProperties = ListXML
End Function
```

Figure 15-11: The commented out columns against the listview structure

```
XMLParam = XMLParam & "     <Column>DrawOfficeNum</Column>"
'  XMLParam = XMLParam & "     <Column>UserField3</Column>"
'  XMLParam = XMLParam & "     <Column>UserField4</Column>"
'  XMLParam = XMLParam & "     <Column>UserField5</Column>"
'  XMLParam = XMLParam & "     <Column>ClearingFlag</Column>"
'  XMLParam = XMLParam & "     <Column>TraceableType</Column>"
'  XMLParam = XMLParam & "     <Column>WarehouseToUse</Column>"
XMLParam = XMLParam & "   </Columns>"
XMLParam = XMLParam & "   <Where>"

XMLParam = XMLParam & CustomizedPane.CodeObject.Statement

XMLParam = XMLParam & "   </Where>"
XMLParam = XMLParam & "  </Query>"
```

Figure 15-12: The additional fields against the input XML for the ***COMFND*** business object

If you need to add other fields from the same table that don't appear in this list, they can be added within the VBScript manually. You must make sure that the same field is added to both the listview configuration and the XML input, and that all fields appear in the same sequence in both places.

In addition, custom form field columns and columns from master tables related to the columns that are already present can be added to the listview as the search window is being added. You can also add blank columns, although for a search window there is little point adding a blank column because it cannot be populated as there is no *OnPopulate* event. This is performed by right-clicking on the column heading of the listview and selecting *Customize* | *Add Custom Columns* from the displayed menu. The *Customize Columns for Listviews* screen is displayed. At the top of this screen, within the *Column Selection* section, is the *Column type* prompt. Against this prompt are three radio buttons, one to add columns from custom forms, one to add blank columns, and one to add columns from master tables linked to columns that already exist in this listview.

If the *Custom form column* radio button is selected, the custom forms linked to the tables of fields already present in the listview appear in the *Custom form type* dropdown list. **Figure 15-13** shows the custom form tables available when the *Customer* search window is in use.

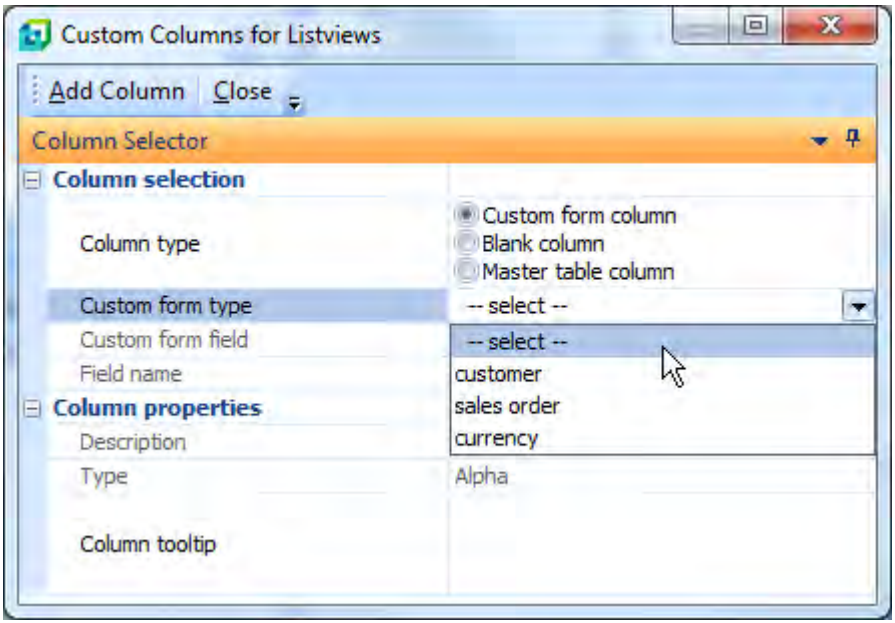


Figure 15-13: Adding a *Custom form column*

If the *Master table column* radio button is selected, the *Table Fields* screen is displayed (see **Figure 15-14**). At the top of this screen is the *Select table* prompt which has a dropdown list against it containing a list of tables that have columns already in the search window results listview.

Selecting one of these table names populates the *Table Fields* screen with the fields from this table. Those fields that already appear in the search window results listview will appear italicised. Those that do not will appear in normal font, and will have the hyperlinked text *Add* in the *Add column* column.

For more information on adding custom form columns, blank columns, and master table columns to the search window results listview, see the following sections in chapter 8 on listviews and data grids:

- Adding Custom Form Fields to a Listview
- Adding Blank Columns to a Listview and Populating Them
- Adding Master Table Fields to a Listview

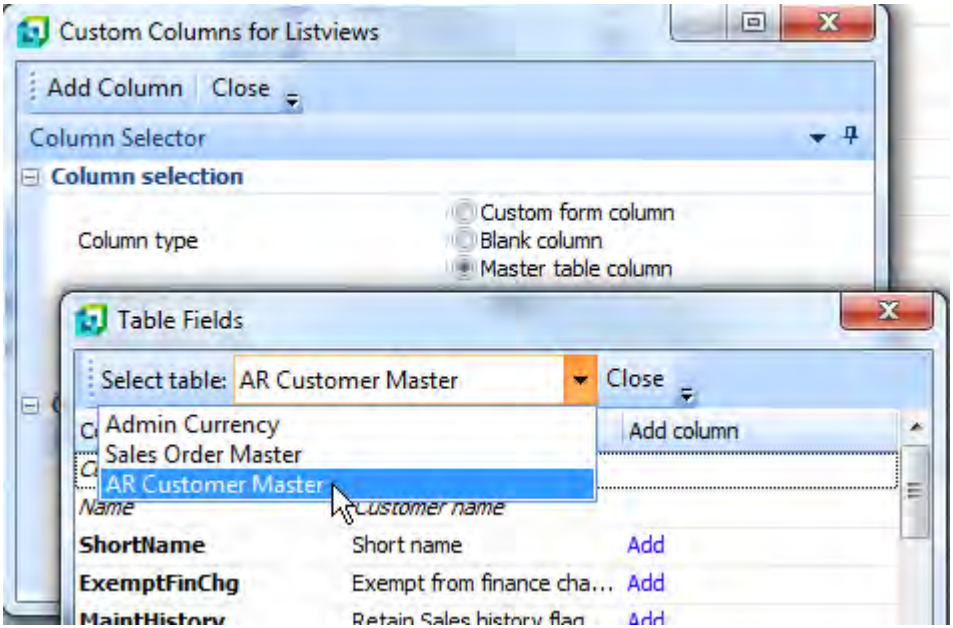


Figure 15-14: Adding master table columns to the search window results listview

Closing a Search Window

If you use the *Create* hyperlink to create a search window, this will remain open while SYSPRO is running. When you exit SYSPRO it will close, but re-open again when SYSPRO is next run. When it re-opens, it will not contain the data from the previous run, but any changes to the columns/VBScript will still be present.

If you close a search window using the X at the top right of the screen (see highlighted item in **Figure 15-15**), any changes made to the columns or VBScript will be lost. If you use the *Create* hyperlink to create it again, it will be the standard one that is created, which will not contain your changes.

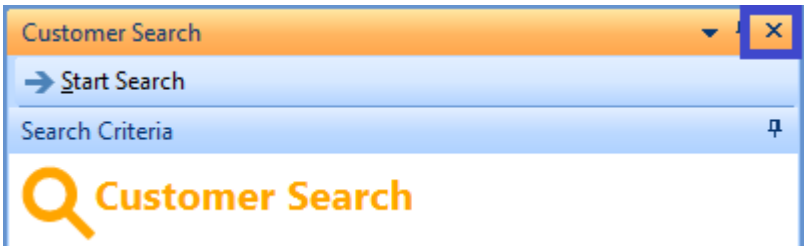


Figure 15-15: Closing a *Search Window* using this X will lose any settings or changes that you made

Using the Results of the Search

If the search is linked to a browse, double-clicking on any part of the row of the search results that contains the key field that you require will pass the key field back to the program with the browse button. For example, if your *Stock Code Search* is linked to the stock code browse button in the *Inventory Query* program, the *Stock Code Search* will be launched when the browse button is selected. After building up your *Search Statement* and clicking on the *Start Search* button, the results will be displayed in the listview at the bottom of the screen. Double-clicking on any part of a row will return the stock code for that row to the *Inventory Query*, and invoke the query for that stock code.

Although not specific to this listview, it is possible to copy the contents of the listview to the clipboard to be used elsewhere. This is covered in Chapter 5 of the first *Power Tailoring* book.

If you need to select more than one record you can select the *Multiple Row Selection* option (right-click on the listview column header | *Export/print* | *Multiple Row Selection*) or use the *Ctrl+Alt+F7* shortcut keystrokes. Select the first row, then hold down the *Ctrl* key and click all of the other rows required. On this same menu you can use the *Copy Selected Records* to the clipboard option (or use the *Ctrl+Alt+F*8 shortcut keystrokes), *Export to Excel* option (or use the *Ctrl+F8* shortcut keystrokes) or the *Export HTML* option (or use the *Ctrl+F7* shortcut keystrokes). You can also copy to the clipboard and paste to any application (such as Word, Excel, or Outlook).

Pane Properties

There are several *Pane Properties* that can be set against a customized pane with the *Object type* of *SearchWindow*.

Pane Caption Section
Within the *Pane Caption* section, the *Show icon* checkbox enables you to place an icon against the titlebar of the customized pane. If this option is checked, the *Edit icon* hyperlink is enabled and this can be used to select the icon to be used.

The *Show toolbar* checkbox is used to display/hide the toolbar. If it is displayed, and no other toolbar controls added, the toolbar will show a *Start Search* button. The *Show caption using XAML theme* dropdown list enables you to select from multiple XAML themes to be used as the top section of the pane. The default theme is *SearchWindow*, which displays a stylized magnifying glass followed by the title of the pane, in an orange color.

Figure 15-16 shows a search window customized pane showing the toolbar (containing the *Start Search* button), the *Wizard* icon against the titlebar, and the top of the customized pane using the *SearchWindow* XAML theme.

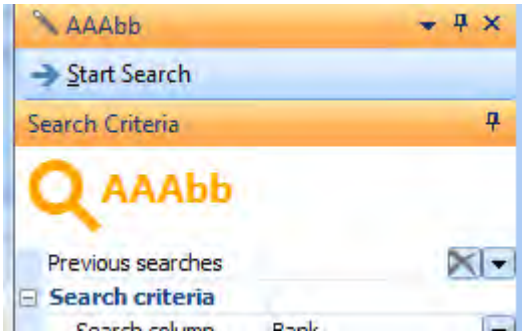


Figure 15-16: A *Search Window* customized pane using the *SearchWindow* XAML

Refresh Details
The *Refresh details* section contains four options, *Ignore OnLoad VBScript*, *Automatic Refresh*, *Refresh time*, and *Refresh period*. The *Ignore OnLoad VBScript* checkbox is used to bypass the *OnLoad* event when a customized pane loads. This can be used to reduce the initial load time of a customized pane. However, this option does not affect a *Search Window* customized pane.

The *Automatic refresh* checkbox specifies whether the customized pane should automatically fire its *OnRefresh* event after a certain time period. If this is checked the *Refresh time* and *Refresh period* options are enabled. The *Refresh time* is used to specify the time interval, and the *Refresh period* is used to specify whether this time interval is in minutes or seconds.

Toolbar Controls
Toolbar control1 and *toolbar control 2* work in the same way as the other customized panes. They were initially covered within Chapter 10, and an example of using them appears within the *Accessing the Content of a Form* section of Chapter 14.

Building Your Own Search Window

A *Search Window* customized pane can be built using the *Object type* of *SearchWindow* (see the section *Pane Properties* section below for the properties that can be set for *Search Windows*). There are several differences between a search window customized pane and a standard search window. The first is that if this customized pane is within a SYSPRO program (instead of the SYSPRO *Main Menu*) it is only available for use within this program. It becomes available when the program is loaded, and shuts down when the program is closed. If this customized pane was created against the SYSPRO *Main Menu* and dragged from this location to float, it will be become available when SYSPRO is loaded, and will still be accessible when a SYSPRO program is loaded, provided that its floating location is outside the boundary of the loaded program, or the operator moves the program to get to the customized pane.

The second difference is that the customized pane search window will not be linked to a browse. So clicking on the browse button within the program will not launch the search window. The third difference is that the customized pane search window will not return an item selected from the search results back to the browse. Other than these differences, a search window customized pane will perform in the same way as a standard *Search Window*.

The simplest way of explaining how a search window customized pane can be created is by showing an example. Within the *Bank Query* program in the *Cash Book* module, click on the *Menu* button at the top of *Bank Information* pane, and select *Customized Pane* | *New*. Against the *Object type* field, select the *SearchWindow* option. Add a title to the customized pane using the *Window title* field, in this example *Cash Book Search*. Click on the *Edit VBScript* button.

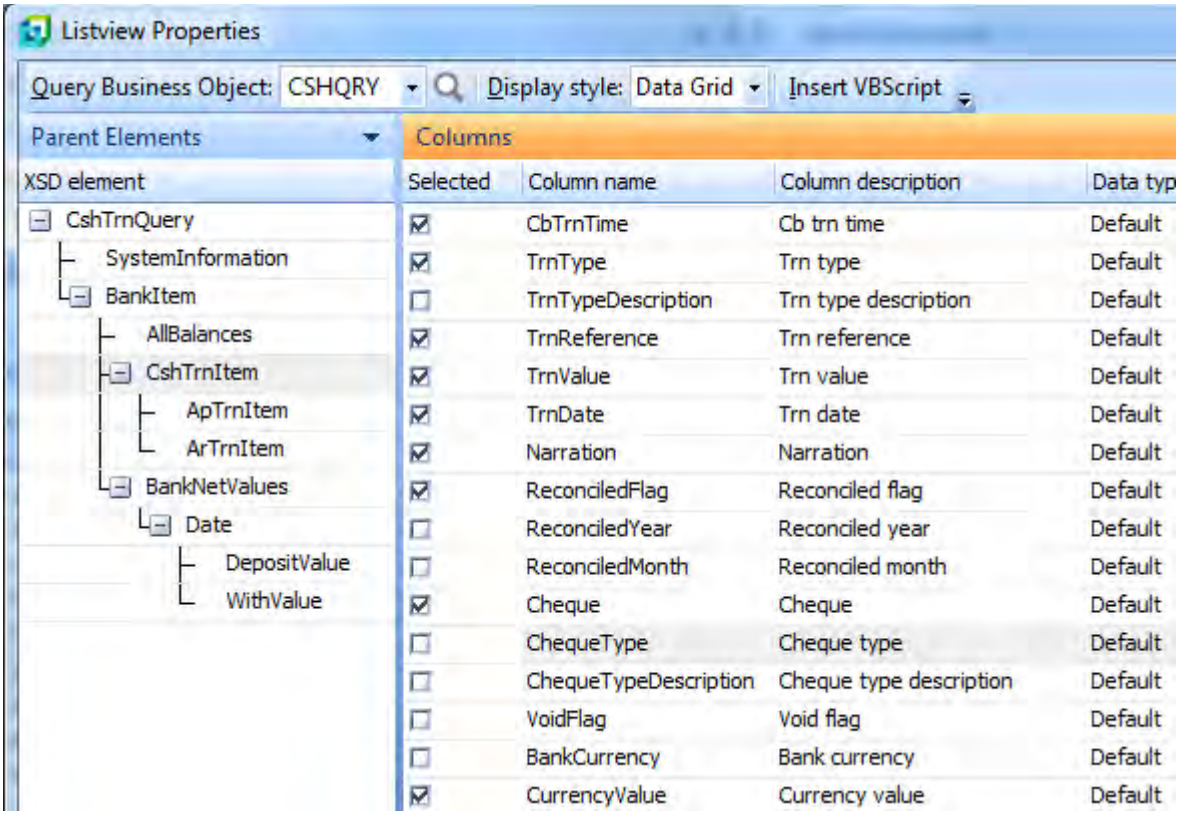


Figure 15-17: The *Listview Properties* screen

Within the *VBScript Editor* screen, double-click on the *OnLoad* event name, and the *CustomizedPane_OnLoad* function will be created. Place the cursor within this function. Double-click on the *ListviewProperties* variable within the *CustomizedPane* section of the *Variables* pane. The

Listview Properties screen will be displayed. The *Listview Properties* screen can be seen in **Figure 15-17**.

Enter the business object name ***CSHQRY*** against the *Query Business Object* prompt, and press the *Tab* key. The contents of the ***CSHQRY*** schema will populate the *Parent Elements* pane. Select the *CshTrnItem* node name in this pane and the list of elements within this node will populate the *Columns* pane on the right.

Within the *Columns* pane, check the checkbox in the *Selected* column against the *CbTrnTime*, *TrnType*, *TrnReference*, *TrnValue*, *TrnDate*, *Narration*, *ReconciledFlag*, *Cheque*, and *CurrencyValue* rows. Although the ***CSHQRY*** business object is not the one that will be used to retrieve the values, it contains most the fields that are required, so can be used to build the listview structure. The ***COMFND*** business object that will be used to retrieve the values is so flexible that its schema can't be used to build this structure. Some of these column names are not the ones that must be used, and you will be changing these in your code later.

After checking the relevant checkboxes against the *Selected* column, click on the *Insert VBScript* button. The code to build the listview structure will be inserted for you. A little tweaking of this structure is required before it can be used. The line of code containing the *Columns* element will match the following line (that has wrapped around on this page).

```
ListXML = ListXML & "<Columns PrimaryNode='CshTrnItem' Style='DataGrid'
AutoSize='true' FreezeColumn='0' AutoInsert='true' >"
```

The *AutoSize* attribute must be changed from *true* to *false*, and the *FreezeColumn* attribute changed from *0* to *1*. These changes need to be made when creating the customized pane. If they are not made now, and you attempt to make this change later, they will be ignored because of the operator preferences that will have been saved when you exit the program the first time. When the changes have been completed, the *Columns* element should match the following (which has wrapped around on this page):

```
ListXML = ListXML & "<Columns PrimaryNode='CshTrnItem' Style='DataGrid'
AutoSize='false' FreezeColumn='1' AutoInsert='true' >"
```

The following lines are those created for the individual columns. The part of each line that adds it to the *ListXML* variable (including the double quote at the end of each line) has been removed to make it easier to follow on this page, as it does not wrap around).

```
<Column Name='CbTrnTime' Description='Cb trn time' Type='alpha' />
<Column Name='TrnType' Description='Trn type' Type='alpha' />
<Column Name='TrnReference' Description='Trn reference' Type='alpha' />
<Column Name='TrnValue' Description='Trn value' Type='alpha' />
<Column Name='TrnDate' Description='Trn date' Type='alpha' />
<Column Name='Narration' Description='Narration' Type='alpha' />
<Column Name='ReconciledFlag' Description='Reconciled flag' Type='alpha' />
<Column Name='Cheque' Description='Cheque' Type='alpha' />
<Column Name='CurrencyValue' Description='Currency value' Type='alpha' />
```

The first line specifies that the element name used to extract the values is *CbTrnTime*, and that the column header should be *Cb trn time*. The element name needs to be changed to *Ban*k, and the column header needs to be changed *to Bank*. The *Type* of *alpha* is correct, so does not need to be changed. The second and third lines require no changes.

The fourth line specifies that the element name is *TrnValue* and the *Type* is specified as *alpha*. The *Type* needs to be changed to be *numeric*, and a new attribute of *Decimals='2'* must be added between the *Type* attribute and the slash at the end of the element.

The ninth line also needs to have the *Type* changed from *alpha* to *numeric*, and the adding of the *Decimals='2'* attribute. When you have finished, the XML containing the individual column entries should match the following (note that line nine has wrapped around).

```
<Column Name='Bank' Description='Bank' Type='alpha' />
<Column Name='TrnType' Description='Trn type' Type='alpha' />
<Column Name='TrnReference' Description='Trn reference' Type='alpha' />
<Column Name='TrnValue' Description='Trn value' Type='numeric' Decimals='2' />
<Column Name='TrnDate' Description='Trn date' Type='alpha' />
<Column Name='Narration' Description='Narration' Type='alpha' />
<Column Name='ReconciledFlag' Description='Reconciled flag' Type='alpha' />
<Column Name='Cheque' Description='Cheque' Type='alpha' />
<Column Name='CurrencyValue' Description='Currency value' Type='numeric'
Decimals='2'/>
```

This completes the code for the *OnLoad* function. Exit back to the *VBScript Editor* screen and double-click on the *OnRefresh* event name. The *CustomizedPane_OnRefresh* function will be created for you.

Make sure that the cursor is positioned within this function and click on the *Call Business Object* button on the toolbar. This will start the *Call a Query Business Object* wizard. Against the *Query Business Object* prompt insert the business object name ***COMFND*** and use the *Tab* key. The *Parameters* tab of the *Parameters* section will be populated with the sample XML for this business object. Replace this sample XML with the XML below. Note that the indentations at the beginning are not relevant; they are present to make the XML easier to read. It does not affect the business object.

```
<Query>
  <TableName>CshTransactions</TableName>
  <NoXml>1</NoXml>
  <ReturnRows>999999</ReturnRows>
  <Columns>
    <Column>Bank</Column>
    <Column>TrnType</Column>
    <Column>TrnReference</Column>
    <Column>TrnValue</Column>
    <Column>TrnDate</Column>
    <Column>Narration</Column>
    <Column>ReconciledFlag</Column>
    <Column>Cheque</Column>
    <Column>CurrencyValue</Column>
  </Columns>
  <Where>
  </Where>
</Query>
```

Click on the *Insert VBScript* button on the toolbar and the code will be added for you. The line of code containing the third to last line of the XML has an opening XML element of *<Where>*. The line immediately after this has the matching closing XML element of *</Where>*. A new line of code must be inserted between these two lines. This will contain the string against the *Search Statement* prompt when the search is run.

Place the cursor on this new blank line and type in the following: (Note that there is a space after the ampersand).

```
XMLParam = XMLParam &
```

Locate the *Statement* variable name within the *CustomizedPane* section of the *Variables* pane, and double-click it. This will append the full name of the *Statement* variable to this line of code. The line should then match the following:

```
XMLParam = XMLParam & CustomizedPane.CodeObject.Statement
```

Between the last line of the code added by the *Call a Query Business Object* wizard, and the *End Function* statement, add a new line and place the cursor on it. Then locate the *ListviewData* variable name within the *CustomizedPane* section of the *Variables* pane and double-click it to add its full name to your code.

```
CustomizedPane.CodeObject.ListviewData
```

Immediately after this add a space, an equal sign, another space and the variable name *XMLOut*. This is the variable containing the XML output from the ***COMFND*** business object, and this statement passes it to the listview.

```
CustomizedPane.CodeObject.ListviewData = XMLOut
```

When completed the *OnRefresh* function should match the code that appears below.

```
Function CustomizedPane_OnRefresh()
dim XMLOut, XMLParam

XMLParam = XMLParam & "<Query>"
XMLParam = XMLParam & "  <TableName>CshTransactions</TableName>"
XMLParam = XMLParam & "  <NoXml>1</NoXml>"
XMLParam = XMLParam & "  <ReturnRows>999999</ReturnRows>"
XMLParam = XMLParam & "  <Columns>"
XMLParam = XMLParam & "    <Column>Bank</Column>"
XMLParam = XMLParam & "    <Column>TrnType</Column>"
XMLParam = XMLParam & "    <Column>TrnReference</Column>"
XMLParam = XMLParam & "    <Column>TrnValue</Column>"
XMLParam = XMLParam & "    <Column>TrnDate</Column>"
XMLParam = XMLParam & "    <Column>Narration</Column>"
XMLParam = XMLParam & "    <Column>ReconciledFlag</Column>"
XMLParam = XMLParam & "    <Column>Cheque</Column>"
XMLParam = XMLParam & "    <Column>CurrencyValue</Column>"
XMLParam = XMLParam & "  </Columns>"
XMLParam = XMLParam & "  <Where>"

XMLParam = XMLParam & CustomizedPane.CodeObject.Statement

XMLParam = XMLParam & "  </Where>"
XMLParam = XMLParam & "</Query>"
on error resume next
XMLOut = CallBO("COMFND",XMLParam,"auto")
if err then
  msgbox err.Description, vBCritical, "Calling Business Object"
  exit function
end if
' Switch on error handling
on error goto 0

CustomizedPane.CodeObject.ListviewData = XMLOut
End Function
```

Exit back to the *VBScript Editor* screen and click on the *Save* button. On the *Customized Pane Editor* screen click on the *Save* and *Exit* button. The *Cash Book Search* customized pane should appear and you can test that it is working.

Figure 15-18 shows the *Cash Book Search* being used to find all transactions for the bank FB that have a transaction value greater than 100.00, and which has not yet been reconciled.

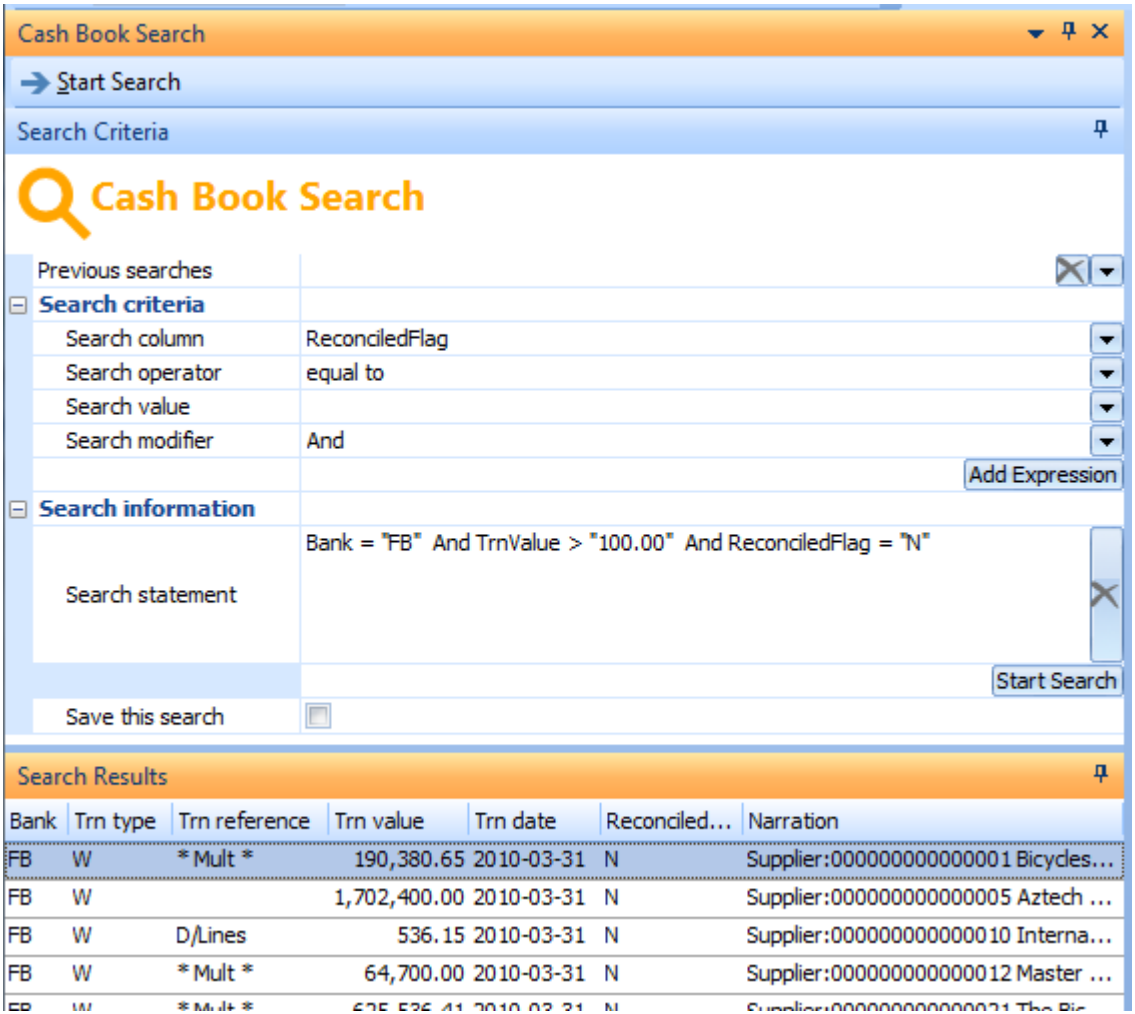


Figure 15-18: The completed *Cash Book Search* search window customized pane

Adding Your Search Window to the Search Windows List

Your search window can be added to the list of searches that appear in the *Search Windows* screen (SYSPRO *Ribbon bar* | *Administration* tab | *Search Windows*). This means that everyone has access to it, although it will not be associated with a browse. **Figure 15-19** shows the *Search Windows* screen with the *Bank* customized pane *Search Window* added.

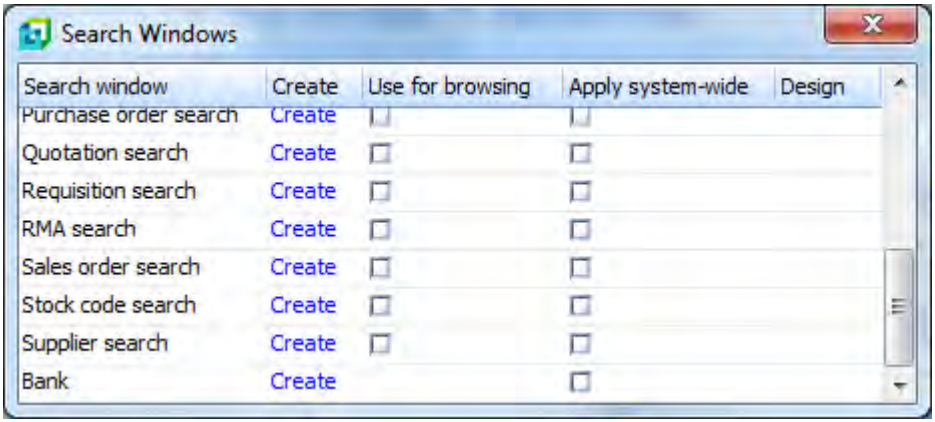


Figure 15-19: The *Bank* customized pane added to the list of *Search Windows*.

The list of available standard *Search Windows* is stored in a file called `IMPSCH.IMP`, which resides in your SYSPRO `Programs` folder. You should not add an entry for your *Search Window* in this file, as your entry will be lost if SYSPRO adds a new one and a later version of this file is downloaded. SYSPRO also reads in the contents of a file called `CUSSCH.IMP`, and this is where you should add your entry. SYSPRO does not provide this file, so it cannot be overwritten by a later version.

The location for your `CUSSCH.IMP` file is dependent on an entry in your `IMPACT.INI` file (the `IMPACT.INI` file resides in your SYSPRO `Work` folder). Within the `IMPACT.INI` file there should be a section called *[Custom Directories]*. Within this section there should be an entry called *CUSPRG=* which specifies where custom programs are to be placed. This may or may not have an entry against it. If it does, this is the folder that should contain your `CUSSCH.IMP` file. If it doesn't, you can add a location for your custom programs. Below is an example that specifies that the custom program folder is a folder called *CUSTOM* directly under the SYSPRO parent folder:

```
[Custom Directories]
CUSPRG=C:\SYSPRO7\CUSTOM
CUSGUI=
```

Make sure that this folder exists, and create it if it doesn't. If it already exists, check to see that there isn't already a `CUSSCH.IMP` file present within the folder (because you don't want to overwrite an existing one that someone else has created). If there is no `CUSSCH.IMP` file, copy the `IMPSCH.IMP` file from your SYSPRO `Base\Samples` folder to be `CUSSCH.IMP` in your custom program folder. This file will then need to be modified.

If there was already a `CUSSCH.IMP` file you just need to add a new entry to the end of this file. If you copied the `IMPSCH.IMP` file it will contain all of the existing search window entries that will need to be removed.

All lines beginning with a semi-colon are comment lines. At the beginning of the file there is a line that specifies the filename of `IMPSCH.IMP`. You must change this to `CUSSCH.IMP` so that there is no confusion if you send this file to someone.

```
; CUSSCH.IMP - Search Windows Templates
```

At the very end of the file is a similar line that must also be changed to `CUSSCH.IMP`.

```
;End of CUSSCH.IMP
```

In between these two lines are two sections, one contains comment lines describing the layout of this file, the other is the list of search windows (the lines that don't start with a semi-colon). The position of the characters in the list of search windows is critical in them appearing in the list and working. Remove all of the standard search window lines except for one, which you will use as a template. In this example the first line (the one starting with *account*) was left and used as a template.

The first column is the caption on a SYSPRO form to which this search window can be associated. The second column contains the description to appear in the *Search Windows* screen. The third column is the name of the file containing the *Search Window* script. The fourth column is a unique identifier, and the fifth column specifies whether this can to be used as the browse for key fields with this caption. The sixth column is called *Dependency*, and will contain the name of an additional field that must be passed to the browse to make up the key to the table. An example of this section of the file appears below. The spacing has been changed, and the *Dependenc*y column removed, so that each line fits on a single line of this page.

```
; ------------------------------------------------------------------------------
;Caption              Description       VBScript import file     ID  Browse
;------------------   -----------       --------------------     --  ------
account               Accounts          Form_SearchAccount.txt   AC    N
```

Replace the caption *accounts* with the caption *bank*. Add spaces to make sure that the line's description lines up with the first dash/hyphen under the *Description* column. Replace the description *Accounts* with the description *Bank*. Add spaces to make sure that the line's VBScript import filename lines up with the first dash/hyphen under the *VBScript import file* column. Replace the filename `Form_SearchAccount.txt` with `Form_SearchBank.txt` and add spaces to make sure that the text *AC* still lines up under the *ID* column. Replace the *ID* of *AC* with *UA* and make sure that the *N* appears below the "*o*" of *Browse*. When you have finished, this section of the file should look similar to the example below. Save the file and exit.

```
; ------------------------------------------------------------------------------
;Caption              Description       VBScript import file     ID  Browse
;------------------   -----------       --------------------     --  ------
Bank                  Bank              Form_SearchBank.txt      UA    N
```

Within SYSPRO you need to call up the *Bank Query* program (that contains the *Cash Book Search* customized pane that you have already created). Click on the *Menu* button on the top right of the *Cash Book Search* customized pane and select *Customized Pane* | *Export Customized Panes* from the menu (see **Figure 15-20**).

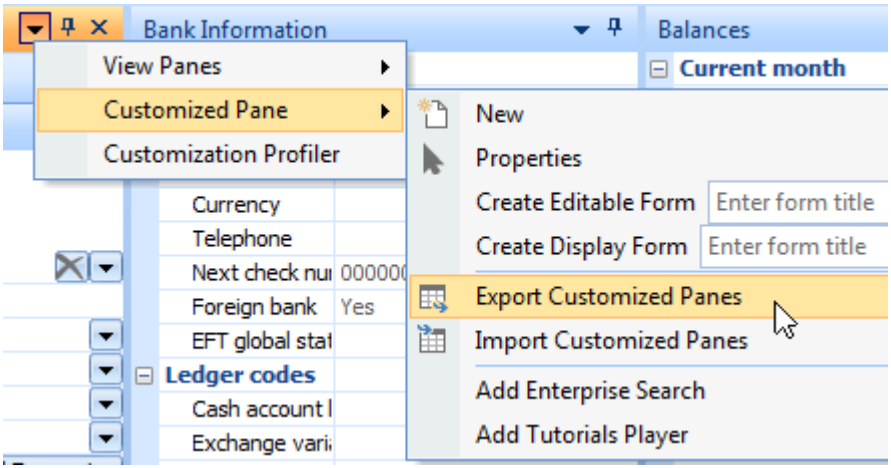


Figure 15-20: Exporting the *Search Window* customized pane

The *Export Customized Panes* screen will be displayed. If more than one entry exists, uncheck all except the *Cash Book Search* entry, and then click on the *Export* button on the toolbar (see **Figure 15-21**).

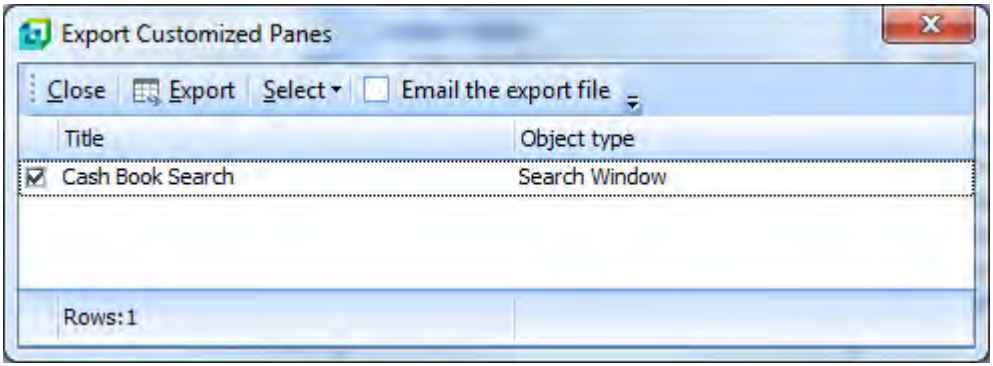


Figure 15-21: Exporting the *Cash Book* search customized pane

You will be prompted to supply the location (via a browse) and a filename. Locate the SYSPRO `Base\Samples` folder, and save the export file as `Form_SearchBank.txt` (or whatever name you

used in the `CUSSCH.IMP` file). Once you are certain that the export file was saved, delete the *Cash Book Search* customized pane from the *Bank Query*. Exit SYSPRO, and call up SYSPRO again. When the *Search Window* screen is loaded (*Ribbon Bar* | *Administration* tab | *Search Windows* option) it will include the *Bank* search window at the end of the listview, as can be seen in **Figure 15-19**). Clicking on the *Create* hyperlink alongside it will create the *Cash Book Search* pane as a customized pane at the same level as the SYSPRO *Main Menu*. This can be dragged from this location and allowed to float free, so can be used while in any program.

Chapter 16 - .NET User Controls

The *.NET User Control* object type is designed to host *Managed Code* inside a SYSPRO customized pane. This enables you to create your own application and host it within SYSPRO. This means that you are no longer reliant on the SYSPRO user interface, or just using VBScript to perform customization.

The *.NET User Control* is not part of SYSPRO, although it appears to the operator as if it is. The customized pane in which it is hosted provides a mechanism of two way communication between the control and SYSPRO, so when something changes in the SYSPRO environment your user control can react accordingly.

From the Microsoft Developer Network (MSDN) website :

> What is Managed Code?
>
> Managed code is code written in one of over twenty high-level programming languages that are available for use with the Microsoft .NET Framework, including C#, J#, Microsoft Visual Basic .NET, Microsoft JScript .NET, and C++. All of these languages share a unified set of class libraries and can be encoded into an Intermediate Language (IL). A runtime-aware compiler compiles the IL into native executable code within a managed execution environment that ensures type safety, array bound and index checking, exception handling, and garbage collection.
>
> By using managed code and compiling in this managed execution environment, you can avoid many typical programming mistakes that lead to security holes and unstable applications. Also, many unproductive programming tasks are automatically taken care of, such as type safety checking, memory management, and destruction of unneeded objects. You can therefore focus on the business logic of your applications and write them using fewer lines of code. The result is shorter development time and more secure and stable applications.

Adding the Customized Pane

When adding a customized pane with an *Object type* of *.NET User Control*, the section at the bottom of the screen (that with some object types would contain templates) is titled *.NET User Control* (see **Figure 16-1**). This is where the user control's name, namespace, methods, and the version of the .NET Framework with which it will work are defined.

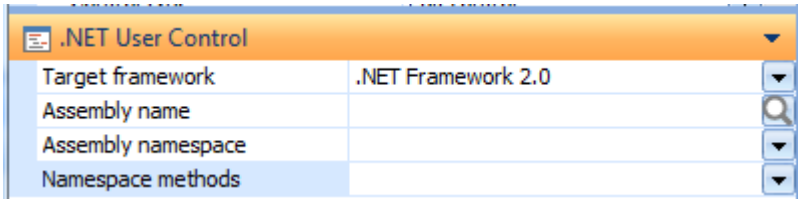


Figure 16-1: The *.NET User Control* pane where the control information is defined

The *Target framework* option is used to define the version of.NET Framework with which the .NET User Control is designed to work. The .NET User Controls can be created using Microsoft Visual Studio .NET 2005 upwards. The control can use *Winforms* (which look similar to SYSPRO) or *WPF*.

The *Target framework* choices are *.NET Framework 2.0* or *.NET Framework 4.0*. Products compiled for other versions will not work.

The *Assembly name* prompt is where you supply the name of the DLL containing the .NET User Control. The DLL can contain .NET User Controls and forms, and both can be present in the DLL. An assembly can contain multiple DLLs. SYSPRO requires that the selected DLL exists in the `ManagedAssemblies` folder under the SYSPRO `Base` folder. The browse against this field performs a lookup on the local machine.

When supplying your DLL it must be placed into the `ManagedAssemblies` folder under SYSPRO's `Base` folder on the SYSPRO application server. The DLL will be copied to the same folder on the SYSPRO client the next time someone logs in on that workstation. This is known as *self-healing*.

Note that in a Terminal Server environment the client software is shared amongst many users. If the DLL is being added to the `ManagedAssemblies` folder for the first time it will be copied down in the same way as a normal SYSPRO client. However, if this is a newer version of a DLL that already exists in the `ManagedAssemblies` folder, the DLL may already be in use by another client. In this case the file will be locked and cannot be overwritten. To ensure that the latest version of the DLL is *self-healed* to the Terminal Server, all operators using SYSPRO from the Terminal Server must exit SYSPRO. The next time that an operator logs in from this Terminal Server, the DLL will be self-healed to the Terminal Server's SYSPRO `Base` folder.

If the .NET User Control has dependencies, these must also be placed in the `ManagedAssemblies` folder on the server. The dependency is a separate assembly (DLL), and accessing it will need to be resolved at run time. This is typically done by using a relative path name of `..\ManagedAssemblies` in your code, as SYSPRO is run from its `Base` folder.

The *Assembly namespace* prompt has a dropdown list against it. This is populated with the names of all the *Controls*, *Modules, Classes* and *Forms* that are contained within the DLL.

The *Namespace methods* prompt contains a dropdown list with the functions within the namespace selected above.

Figure 16-2 shows the *.NET User Control* settings for a .NET User Control that ships with SYSPRO called *SYSPROMA_NETTestHarness.dll*. This test harness can be used to retrieve system information, as well as information from other forms within the same program. It can also be used to change the contents of those forms.

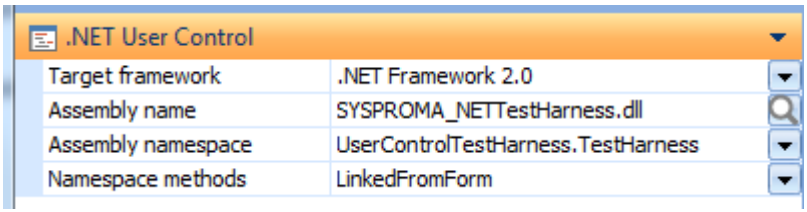


Figure 16-2: The *.NET User Control* settings for the test harness that ships with SYSPRO

Interacting with a .NET User Control Customized Pane

There are several ways to interact with a .NET User Control customized pane. The first is where a field on another form is linked to the customized pane. When the value against the linked field changes the *LinkedFromForm* method in the linked .NET User Control is invoked and passed two values. The first is the name of the form and field containing the value that changed, and the second is the value of the field after it changed. This is covered in the section *Passing Information from a Form Field to the Customized Pane* below.

The second way is where the .NET User Control customized pane can retrieve information about the SYSPRO environment (see *Retrieving Information about the SYSPRO Environment* section below). It can also update the content of the *System Variables* (see *Updating a System Variable Value*).

The .NET User Control can retrieve information from other forms within the same program as the customized pane (see *Retrieving Information from Other Forms* section below).

The values against other forms can be changed from the .NET User Control as if they had been changed by the operator (see *Setting a Value against a Field on Another Form* section below).

The .NET User Control can call e.net Solutions business objects to retrieve information, or post information. Providing that this is done the correct way, these business objects do not need to be licensed for use by this operator (see *Invoking a Customized Pane's VBScript Function from the User Control* and *Interacting with an Embedded VBScript* below).

Passing Information from a Form Field to the Customized Pane

When a .NET User Control customized pane resides within a SYSPRO program (as opposed to within the SYSPRO *Main Menu*), the context-sensitive menu that is displayed when you right-click on a field on another form within the same program will include the *Link to .NET control* option. If this is highlighted with the mouse pointer the names of the available .NET User Control customized panes are displayed. If the field which was highlighted when you right-clicked on the form is already linked to one of the customized panes it will have a checkmark against it.

Figure 16-3 shows the context-sensitive menu where there are two .NET User Control customized panes within this program, *Test Harness*, and *Gate Calculator*. The field that was highlighted when the form was right-clicked is already linked to the *Test Harness* customized pane. Each field can be linked to more than one .NET User Control.

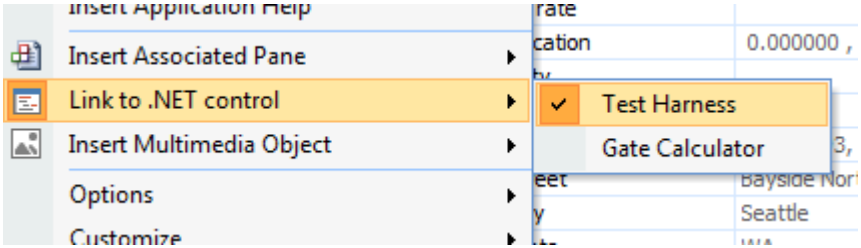


Figure 16-3: Using the *Link to .NET control* option to view the customized panes

Once this link is set up, whenever the value against this field changes the *LinkedFromForm* method is called in the relevant .NET User Control (note that the method name *LinkedFromForm* is fixed and cannot be changed). The method is passed two parameters. The first parameter consists of the form and field names of the value that changed, separated with an underscore. The second parameter contains the value to which this field changed.

An example is where the *Customer* field on the *Customer Information* form within the *Sales Order Entry* program was linked to the .NET user control. If its value changed to *0000002* the *LinkedFromForm* method would be called with the first parameter containing *CustomerInformation_Customer*, and the second parameter containing *0000002*.

Because the incorrect field could have been linked to the customized pane it would be worth building a check into the *LinkedFromForm* method to make sure that the field name passed through within the first parameter matches the name of the expected field. This would highlight that the incorrect field had been linked using the *Link to .NET control* option. It may also be worth performing a similar check against the form name.

A schematic showing this interaction called *Schematic 3 - Passing Information from a Form Field to the Customized Pane* appears at the end of this chapter.

Retrieving Information about the SYSPRO Environment

The user control can retrieve information about the SYSPRO environment, and this is returned as XML. This is performed by the user control calling the *GetSYSPROInfo* method. An example in C# appears below:

```
String MethodName = "GetSysproInfo";
String Output = "";
MyMethodName(ref MethodName, ref Output);
```

The environment XML contains two main sections. The first is *SystemVariables*, which contains the information that is available within both the *SystemVariables* and *SystemVariables (Actions)* sections of the *Variables* pane within the VBScript editing environment. The second is *Personalization*, which contains personalization information regarding this operator code. The following is an abbreviated extract of the XML returned to give an idea of what is available.

```
<Variables>
  <SystemVariables>
    <Operator Value="ADMIN" />
    <OperatorName Value="SYSPRO Administrator" />
    <OperatorEmail Value="" />
    <OperatorGroup Value="ADMIN" />
    <SMTPIPAddress Value="" />
    <SYSPROVersion Value="7.0" />
    <IMPINIParameters Value="C:\TST700\WORK\IMPACT.INI" />
  </SystemVariables>
  <Personalization>
    <OverallTheme Value="03" />
    <UserDefinedSkin Value="" />
    <ListviewDefaultFont Value="Tahoma,0008" />
    <ListviewCaptionFont Value="Tahoma,0008" />
    <DefaultDateFormat Value="{short}" />
    <FormReadOnlyColor Value="095095095" />
    <FormBackgroundColor Value="255255255" />
    <DockingPaneHighlightActive Value="1" />
    <DockingPaneShowMaximize Value="0" />
  </Personalization>
</Variables>
```

Retrieving information from Other Forms

The user control can retrieve information about fields on other forms in the program within which it resides. The information is returned as XML. This is performed in a similar way to the *GetSYSPROInfo* method above, but using the *GetApplicationInfo* method. Below is an example in C#:

```
String MethodName = "GetApplicationInfo";
String Output = "";
MyMethodName(ref MethodName, ref Output);
```

By default, the XML returned contains all of the fields for all of the forms in this program, grouped by form. This includes customized panes with an *Object type* of *Form*, associated panes that were defined as a form, and the header section of customized panes with an *Object type* of *SearchWindow*. The *Form* customized panes and associated panes include those that have had their forms built using the *Design Form* facility, or built dynamically using VBScript. The returned XML will reflect the status of the form at the time that the method is called, as these forms can be manipulated dynamically.

Each form has its own section within the returned XML. The *Form* element contains the *Name* and *Description* attributes. For each field on a form there is a *Variable* element. Each *Variable* element has a *Name* attribute containing the field's caption, a *Value* attribute containing the field's current value, and a *Type* attribute that specifies if the field is alphanumeric, numeric (allowing negatives), numeric (not allowing negatives), or a date. Other attributes can also appear such as *IsReadOnly*='*true*' when a field is defined as read-only. Below is an extract from the results returned by the *GetApplicationInfo* method when the .NET User Control customized pane resides within the *Sales Order Entry and Maintenance* program.

```
<Variables>
  <FormVariables>
    <Form Name="IMP040L3" Description="OrderHeader">
      <Variable Name="A/R invoice terms" Value="" Type="A" />
      <Variable Name="Address" Value="" Type="A" />
      <Variable Name="AlternateKey" Value="" Type="A" />
      <Variable Name="Area" Value="" Type="A" />
      <Variable Name="Cancelled order" Value="" Type="A" IsReadOnly="true" />
      <Variable Name="Customer purchase order" Value="" Type="A" />
      <Variable Name="Customer tax regn" Value="" Type="A" />
      <Variable Name="Delivery route" Value="" Type="A" />
      <Variable Name="Discount % 1" Value="0.000000" Type="N" />
      <Variable Name="Discount % 2" Value="0.000000" Type="M" />
      <Variable Name="Discount % 3" Value="0.000000" Type="M" />
      <Variable Name="Discount percentage" Value="1" Type="A" />
      <Variable Name="Document type" Value="" Type="A" IsReadOnly="true" />
      <Variable Name="Email address" Value="" Type="A" />
      <Variable Name="Exchange rate" Value="0.000000" Type="N" IsReadOnly="true" />
```

It is possible to reduce the XML returned to contain just one form by supplying the name of the form (as opposed to its description) appended to the method name with an underscore. The following example is from the *Sales Order Entry and Maintenance* program, and would return the form and field information for only the *Customer Information* form (whose name is ***IMP040L4***).

```
String MethodName = "GetApplicationInfo_IMP040L4";
String Output = "";
MyMethodName(ref MethodName, ref Output);
```

Updating a System Variable Value from the User Control

One or more system variable values can be updated from the user control using the *SetSYSPROInfo* method. The values to be updated must be supplied as parameters in XML. The XML that is given to this method must be the same format as is returned by the *GetSYSPROInfo* method. The root element must be *Variables*, with the two possible elements within this *SystemVariables,* and *Personalization*. Within the *SystemVariables* section the variable name becomes the element name, with a *value* attribute containing the new value.

The sample XML below could be used within a .NET User Control embedded in a customized pane within the *Sales Order Entry and Maintenance* program. The XML updates the system variables that would cause the *Entered Order Lines* listview to be refreshed, and *Global Variable 4* to contain the text *Updated*.

```
<Variables>
  <SystemVariables>
    <ActionToInvoke value=' DoRefreshLines,40004'/>
    <GlobalVariable4 value='Updated'/>
  </SystemVariables>
</Variables>
```

If the above XML is added to a variable called *UpdateValues* in the C# code below, the code would cause the *Entered Order Lines* listview to be refreshed and *GlobalVariable4* to be updated with the value *Updated*.

```
String MethodName = "SetApplicationInfo";
MyMethodName(ref MethodName, ref UpdateValues);
```

Setting a Value against a Field on another Form

The user control may need to update the value of fields against one or more forms in the current SYSPRO application. To do this the user control calls the *SetSYSPROInfo* method passing an XML

string containing the form, fields, and new values. The XML string must be in the same format as would be returned by the *GetSYSPROInfo* method.

Within the *Form* element, the *Name* attribute must contain the eight character form name, not the description. Within the *Variable* element the field's name is the standard English caption for this field. Also within the *Variable* element, focus can be set on a field using the *Focus*='*true*' attribute, and the field can be set to read-only using the *IsReadOnly*='*true*' attribute.

The following XML (if used within a .NET User Control hosted in a customized pane residing in the *Sales Order Entry and Maintenance* program) would set the *Customer Purchase Order* field to contain the text *BIG CUS*, the *Special Instruc* field to contain the value *Ship 3*, and focus would be set on this field.

```
<Variables>
  <FormVariables>
    <Form Name="IMP040L3" Description="Order Header">
      <Variable Name='Customer purchase order' Value='BIG CUS'></Variable>
      <Variable Name='Special Instruc' Value='Ship 3' Focus='true'></Variable>
    </Form>
  </FormVariables>
</Variables>
```

If the above XML is added to a variable called *UpdateValues* in the C# code below, it would cause the above mentioned actions to occur.

```
String MethodName = "SetSYSPROInfo";
MyMethodName(ref MethodName, ref UpdateValues);
```

Interacting with an Embedded VBScript

The .NET User Control can contain its own VBScript code that can be invoked by the user control. This VBScript can be used to interact with business objects without requiring them to be licensed.

Functions within the VBScript can be called by raising the *ExecuteScript* event followed by a colon, and the name of a function within the script that should be called. Optionally, up to four parameters can be passed to the script, each of which can be up to 255 characters long. Any result from the VBScript (such as the XML output from a business object, results from a SQL Select statement, etc.) can be passed back to the user control by assigning the value to the name of the function.

There is no limit to the length of the VBScript contained within the user control. The VBScript can also contain references to SYSPRO variables that will be available within the environment in which will be running, such as variables for fields on forms within the program that it will reside, and any SYSPRO system variables.

An example of where this could be useful is where a user control is used to build up a list of parts required by a customer that need to be used in a sales order. A function within the VBScript is used to call the ***SORTOI*** business object using the XML built up by the user control to add these lines to the order. Because the business object is called from within this script there is no requirement to license the business object. The result returned by the business object is assigned to the function name, and the user control can detect if the posting was successful. If it was successful it would have XML containing the sales order number that was created. If it was unsuccessful it would have XML containing the error message(s).

The following is a sample of C# code showing how the *ExecuteScript* event is called. The VBScript is assigned to the *vbScriptInput* string. The name of the function to call within the VBScript is *PostSO*, and any result to be returned by the function would be available using this function name.

```
String MethodName = "ExecuteScript:PostSO(param1,param2)";
String vbScriptInput = "VBScript code";
MyGetSetSYSPRO(ref MethodName, ref vbScriptInput);
```

From a developer's point of view there are a couple of benefits to using this method of interacting with the business objects. The first is that as the VBScript is held within the DLL you know that nobody can make changes to it, causing errors, or changing the way that it works. The second is that if you are distributing the user control to other departments within your enterprise, or even via the *SYSPRO App Store*, you know that the script will ship with the user control because it is embedded within it. The third is that it protects your intellectual property, as the human-readable script is not visible to any humans.

A schematic showing this interaction called *Schematic 4 - Interacting with an Embedded VBScript* appears at the end of this chapter.

Invoking a Customized Pane's VBScript Function from the User Control

The user control can talk to the customized pane's VBScript to get it to perform tasks on its behalf, such as retrieving information, or calling a business object without requiring it to be licensed. This is performed by firing the customized pane VBScript's *OnRefresh* function, and optionally passing it one or more messages in a parameter.

When the user control needs to call the VBScript it does so by calling the *doRefresh* method of the *MyGetSetSYSPRO* event and optionally passing through a parameter. The VBScript's *OnRefresh* event is fired and the code within the *OnRefresh* function is run. The information that was passed through within the parameter is available to the script in a system variable called *RefreshValue*.

The following is a sample of C# code where the *doRefresh* method of the *MyGetSetSYSPRO* event is executed and passed a parameter containing *retrieveinvoices:000000000000016*. This could be to

notify the VBScript that it should call the ***ARSQRY*** business object for customer 16, and return the invoices for this customer.

```
String MethodName = "doRefresh";
String Output = "retrieveinvoices:000000000000016";
MyGetSetSYSPRO(ref MethodName, ref Output);
```

The *OnRefresh* function of the VBScript would split up the text *retrieveinvoices:000000000000016* that appears in the *RefreshValue* variable using the colon as a delimiter, so that it contains *retrieveinvoices* and *000000000000016*. It would then know to call the business object, and for which customer.

A schematic showing this interaction called *Schematic 1 - Invoking a Customized Pane's VBScript function from the User Control* appears at the end of this chapter.

Invoking methods in the User Control from a SYSPRO Form

When a .NET User Control is hosted within in a customized pane, its methods can be executed from within the VBScripts of other forms within the same program. When the user control's method is executed, two parameters can be passed to the method.

When editing a VBScript for a form, the *Variables* pane contains a section called *CustomizedPanes*, which has a list of all of the customized panes associated with this program (see **Figure 16-4**).

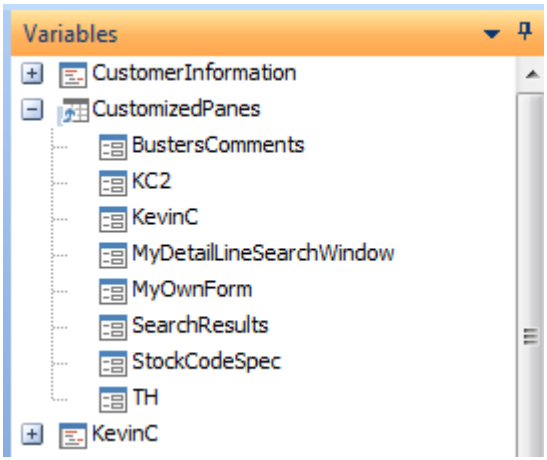


Figure 16-4: The list of customized panes associated with the program containing this form

Double-clicking on one of these names displays the *Define Action for:* screen. This has been covered in Chapter 10 where it was used to force the customized pane to fire its *OnRefresh* event. Whatever

text appears against the *Fire OnRefresh event for this customized pane, passing it this value* prompt will be available within the customized pane's *RefreshValue* variable when it refreshes. This enables you determine which action to take based on why the *OnRefresh* event occurred.

In addition, for customized panes that contain a .NET User Control, the lower half of this screen that is normally greyed-out becomes enabled.

Figure 16-5 shows the *Define Action for:* screen where the selected customized pane contains a .NET User Control. The *Execute method on a .NET user control* checkbox has been checked within the *.NET user control* section of the screen, so the *Method name* dropdown is enabled. This is populated with the method names associated with the *Assembly namespace* selected when the customized pane was created/last updated.

The customized pane *TH* that appears in **Figure 16-5** contains the test harness .NET User Control that ships with SYSPRO. The list of methods that is available in the dropdown list against the *Method name* prompt are those from the user control's *UserControlTestharness.TestHarness* namespace.

Figure 16-6 shows the lower section of the *Customized Pane Editor* screen where the details of the *Test Harness* user control were entered. It can be seen that the methods available here are the same as the methods available in **Figure 16-5**.

There are two parameters that can be passed through, and each can contain up to 70 characters. These can be seen under the *Method name* dropdown list in **Figure 16-5**.

The completed *Define Action for:* screen appears in **Figure 16-7**. Against the *Fire OnRefresh event for this customized pane, passing it this value* prompt is the text *BeyondJustdoRefresh*. When the script is run and the customized pane is refreshed, the *RefreshValue* variable will contain this value. The *Execute method on a .NET user control* checkbox has been checked and the *LinkedFromForm* method has been selected from the dropdown list. The first parameter to be passed through contains text, and the second parameter contains XML.

When the *Insert VBScript Code* button is selected the following three lines of code are added to your script. They appear below with a blank line between each of them to make it easier to follow, because the first line wraps around and covers three lines on the page.

```
CustomizedPanes.CodeObject.TH =
"BeyondJustdoRefresh;ExecuteMethod=LinkedFromForm;Parameter1=MyParameter1;Parameter
2=<MyXML><Detail1>ABC</Detail1><Detail2>123</Detail2></MyXML>"

CustomizedPanes.CodeObject.ExecuteMethodParameter1=""

CustomizedPanes.CodeObject.ExecuteMethodParameter2=""
```

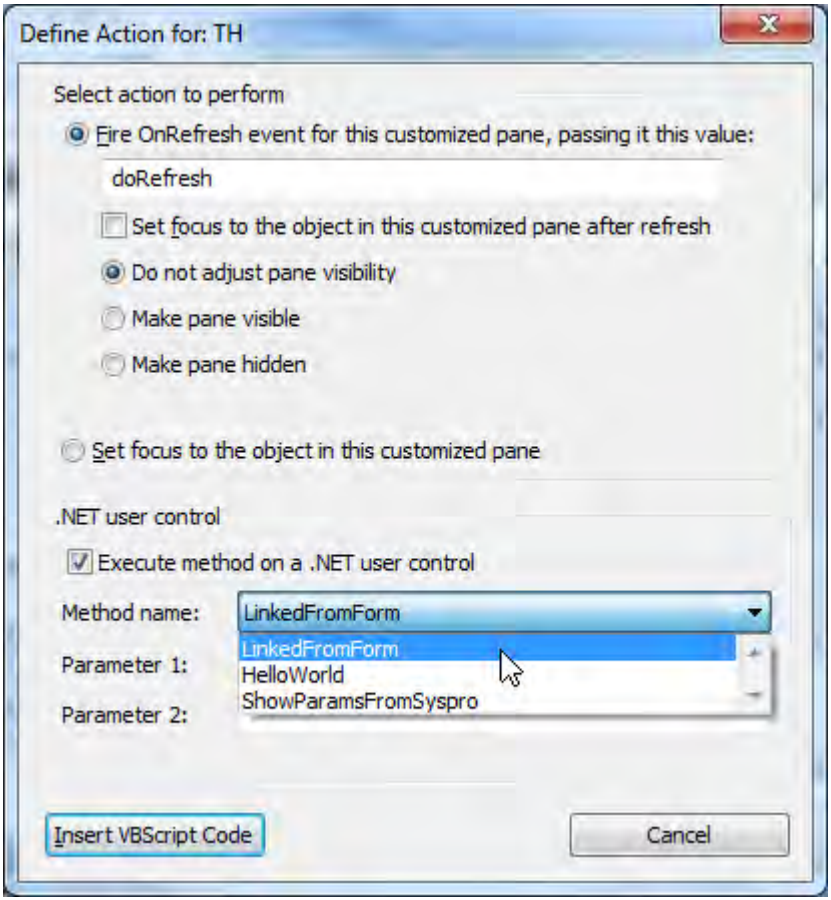


Figure 16-5: The dropdown list of *methods* associated with the user control's *namespace*

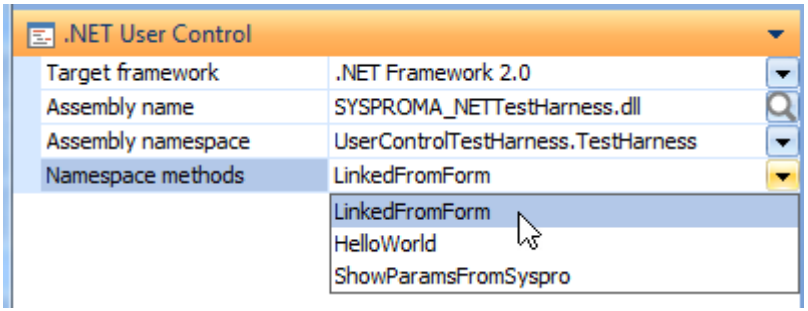


Figure 16-6: The *namespace*/*methods* when configuring the customized pane

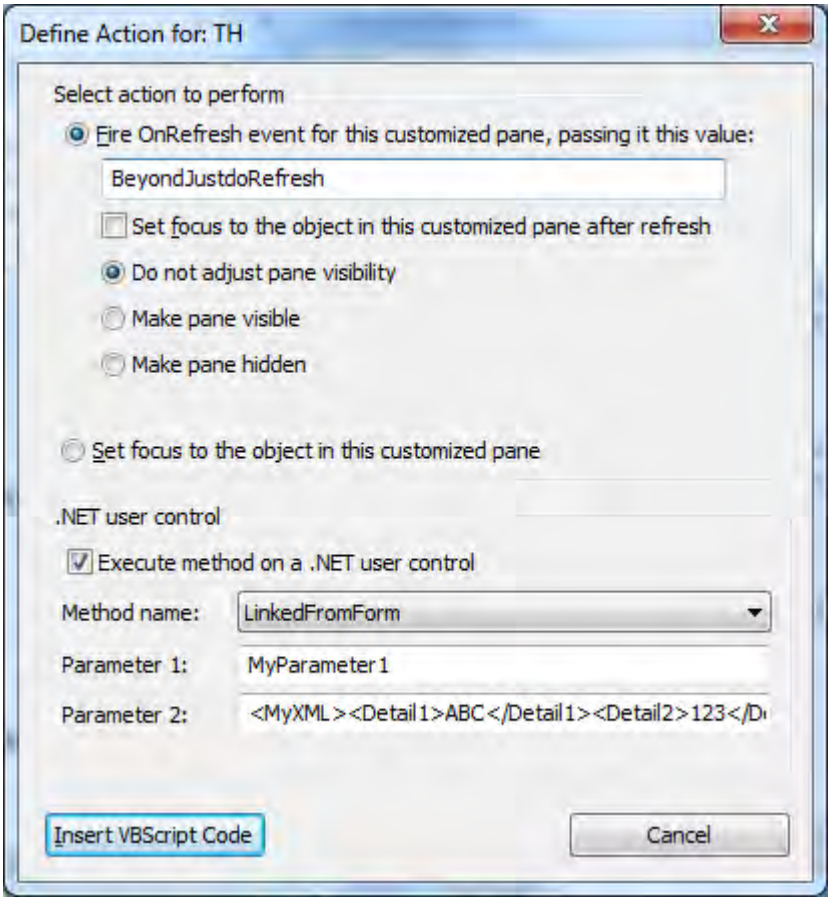


Figure 16-7: The completed *Define Action for:* screen

The first line contains all of the information from the *Define Action for:* screen. The second and third lines are an alternate means of passing the parameters, and these strings can be of unlimited length.

If either the *ExecuteMethodParameter1* or *ExecuteMethodParameter2* variables are not null, the value against them is the value passed to the method for this parameter number in place of anything that appears against this parameter in line 1.

The following three lines of code (separated by blank lines) show the same information being passed to the customized pane/.NET User Control method as above, but using the *ExecuteMethodParameter* variables.

```
CustomizedPanes.CodeObject.TH = "
BeyondJustdoRefresh;ExecuteMethod=LinkedFromForm;Parameter1=;Parameter2="

CustomizedPanes.CodeObject.ExecuteMethodParameter1="MyParameter1"

CustomizedPanes.CodeObject.ExecuteMethodParameter2="<MyXML><Detail1>ABC</Detail1><D
etail2>123</Detail2></MyXML>"
```

When the specified method of the user control has finished executing, the *OnAfterExecuteMethod* event is fired in the customized pane's VBScript. The name of the user control method that was executed is available within the *ExecuteMethodName* variable, and any value returned by the user control's method will be available within the *ExecuteMethodReturnValue* variable. Both of these variables are available within the *CustomizedPane* section of the *Variables* pane (see **Figure 16-8**). This enables you to determine within your script which method in the user control was executed, and handle the returned value accordingly.

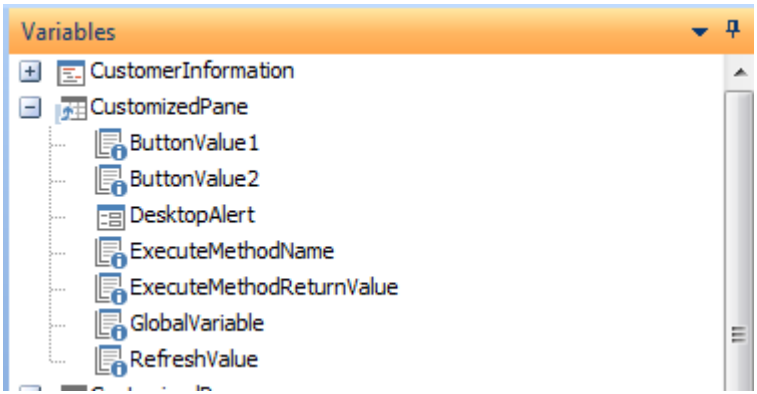


Figure 16-8: The *ExecuteMethodName* and *ExecuteMethodReturnValue* variables

When this section of script is run, the sequence of executions is:

1) The *OnRefresh* event fires in the customized pane's VBScript, where the *RefreshValue* variable will contain the value passed against the *Fire OnRefresh event for this customized pane, passing it this value* prompt.

2) The specified method in the user control will be executed, passing it the two parameters.

3) The *OnAfterExecuteMethod* event is fired in the customized pane's VBScript, the *ExecuteMethodName* variable contains the name of the method that was executed, and the *ExecuteMethodReturnValue* variable contains any results returned by the method.

A schematic showing this interaction called *Schematic 2 - Invoking methods in the User Control from a SYSPRO Form* appears at the end of this chapter.

Additional Information

Toolbar Controls

When creating a .NET User Control customized pane (although the *Toolbar control 1* and *Toolbar control 2* options are available and can be populated on the *Customized Pane Editor* screen) the toolbar will not appear against the customized pane, so cannot be used.

UserControlDestroyed Method

When a customized pane containing a .NET User Control is closed, the customized pane automatically calls a method in the user control called *UserControlDestroyed*. This enables the developer of the user control to perform any final processing required, or clean up any temporary files etc. When this method is called, no parameters are passed to it. If the user control does not contain this method, no error message is returned by SYSPRO.

Smart Link Context Menu

A .NET User Control customized pane can contain a listview. If the listview columns include key fields from SYSPRO, there is a facility to apply the SYSPRO *Smart Link* functionality to the listview. **Figure 16-9** shows where this *Smart Link* functionality has been applied to a column called *Field*. The operator has right-clicked on one of the cells in this column and the context-sensitive menu matching this key is displayed. Clicking on one of the options in this menu will call the appropriate SYSPRO program, and pass it the value. The highlighted cell is *Customer*, and the highlighted menu option is *Query*. When the operator clicks on this, the *Customer Query* program ***ARSPEN*** will be called, and the customer account code of 0000001 will be passed to the program. The query program will open and display the details for customer 0000001.

If the operator had right-clicked on row two of the *Key* column, the context-sensitive menu would contain the *Supplier Quick View*, *Supplier Query*, etc. The list of key fields that are *Smart Link* aware appear in a file called `IMPLNK.IMP` in the SYSPRO `Programs` folder, along with the menu items that will appear, and the programs that will be called when the menu item is selected.

The *Smart Link* functionality is added to the user control's listview using the *ShowSmartTag* event and passing through the key field name, a semi-colon, and the value required. In the example that appears in **Figure 16-9**, the following would be passed:

```
Customer;0000001
```

If the operator had right-clicked on row two, the following would have been passed:

```
Supplier;0000001
```

And for row three this would have been:

```
Sales order;000821
```

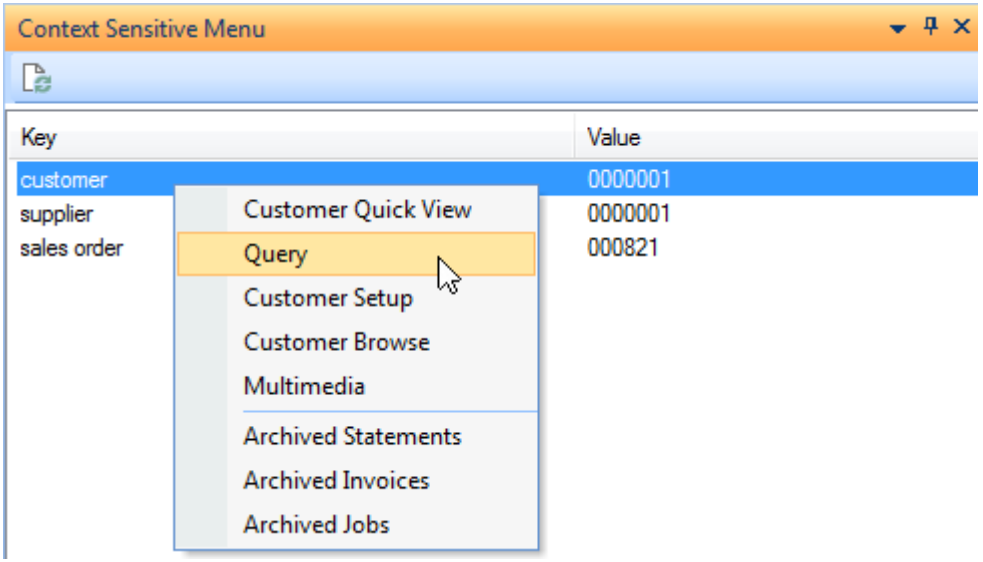


Figure 16-9: The *SmartTag* functionality embedded in a .NET User Control

Below is the code that was used with the listview in the user control that appears in **Figure 16-9**. Each line of code is followed by a blank line to make it easier to follow, as some lines wrap around on the page. The variable *strSysproInfo* contains the text from column one, a semi-colon, and the text from column two when the *ShowSmartTag* event is raised.

```
Public Class UserControl1

  Public Event SysproInteraction(ByRef EventName As String, ByRef XmlString As
String)

  Private Sub ListView1_MouseClick(sender As Object, e As MouseEventArgs) Handles
ListView1.MouseClick

    If ListView1.SelectedItems.Count > 0 Then 'Check to see if item is selected

      Dim objRow As ListViewItem

      objRow = ListView1.SelectedItems(0)

      If e.Button = Windows.Forms.MouseButtons.Right Then

        Dim strSysproInfo As String = objRow.Text & ";" & objRow.SubItems(1).Text

        RaiseEvent SysproInteraction("ShowSmartTag", strSysproInfo)

      End If

    End If

  End Sub

End Class
```

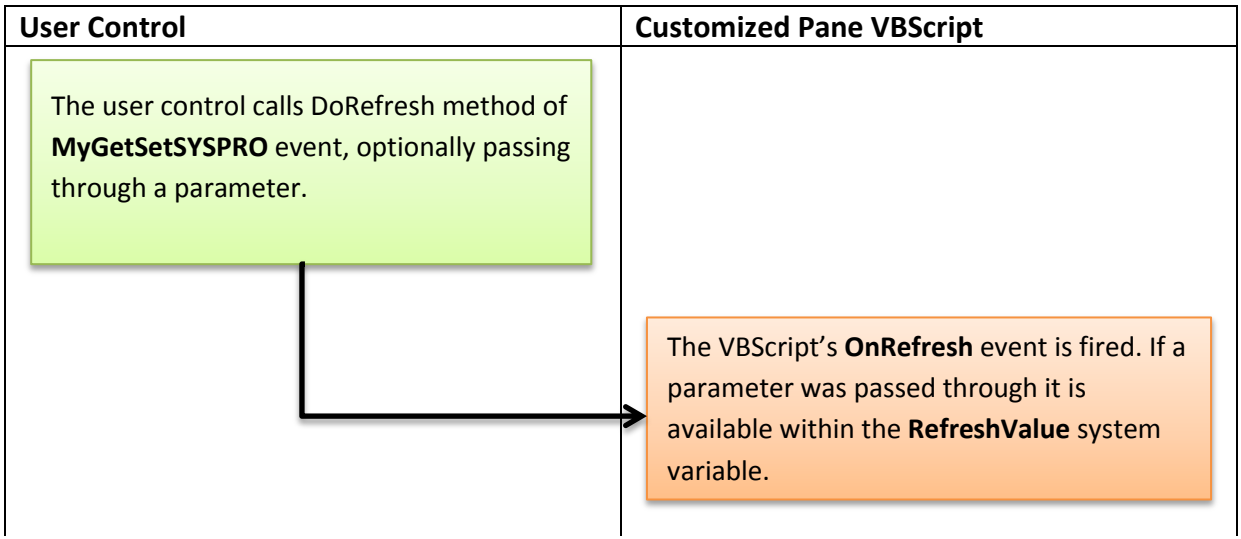
Schematics

Below are four schematics that show the interactions between the different components for:

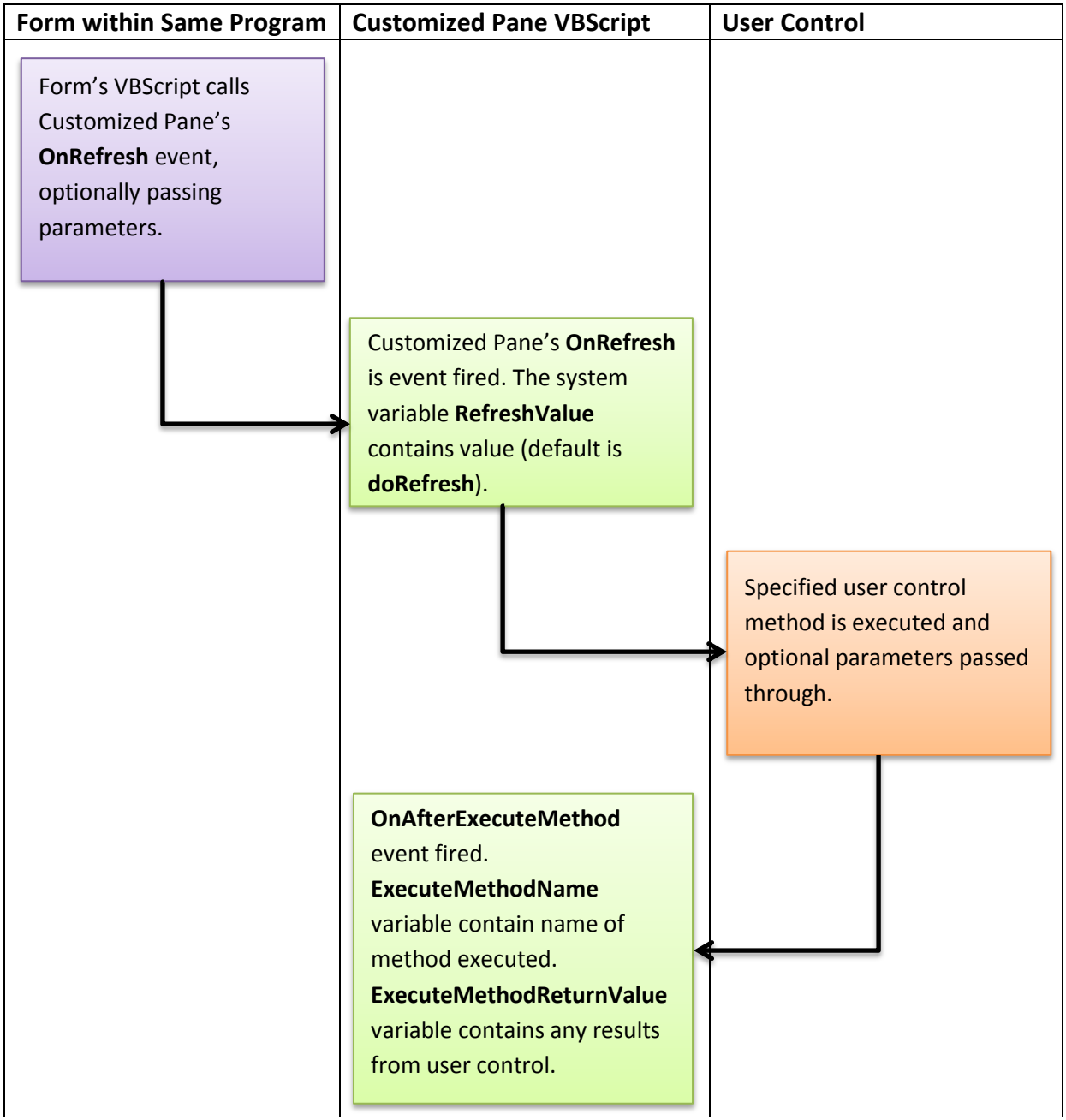
- Invoking a Customized Pane's VBScript function from the User Control
- Invoking methods in the User Control from a SYSPRO Form
- Passing Information from a Form Field to the Customized Pane
- Interacting with an Embedded VBScript

These are intended to clarify the steps, and can also be used as a quick reference.

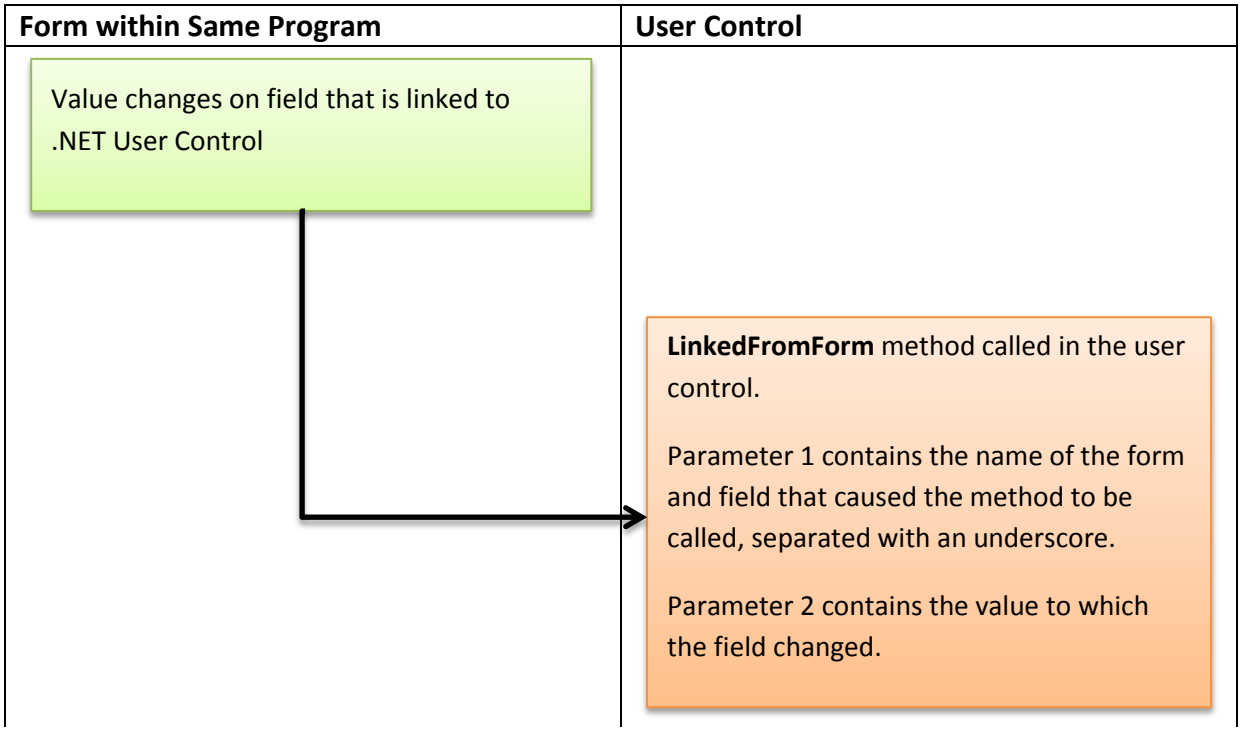
Schematic 1 - Invoking a Customized Pane's VBScript function from the User Control



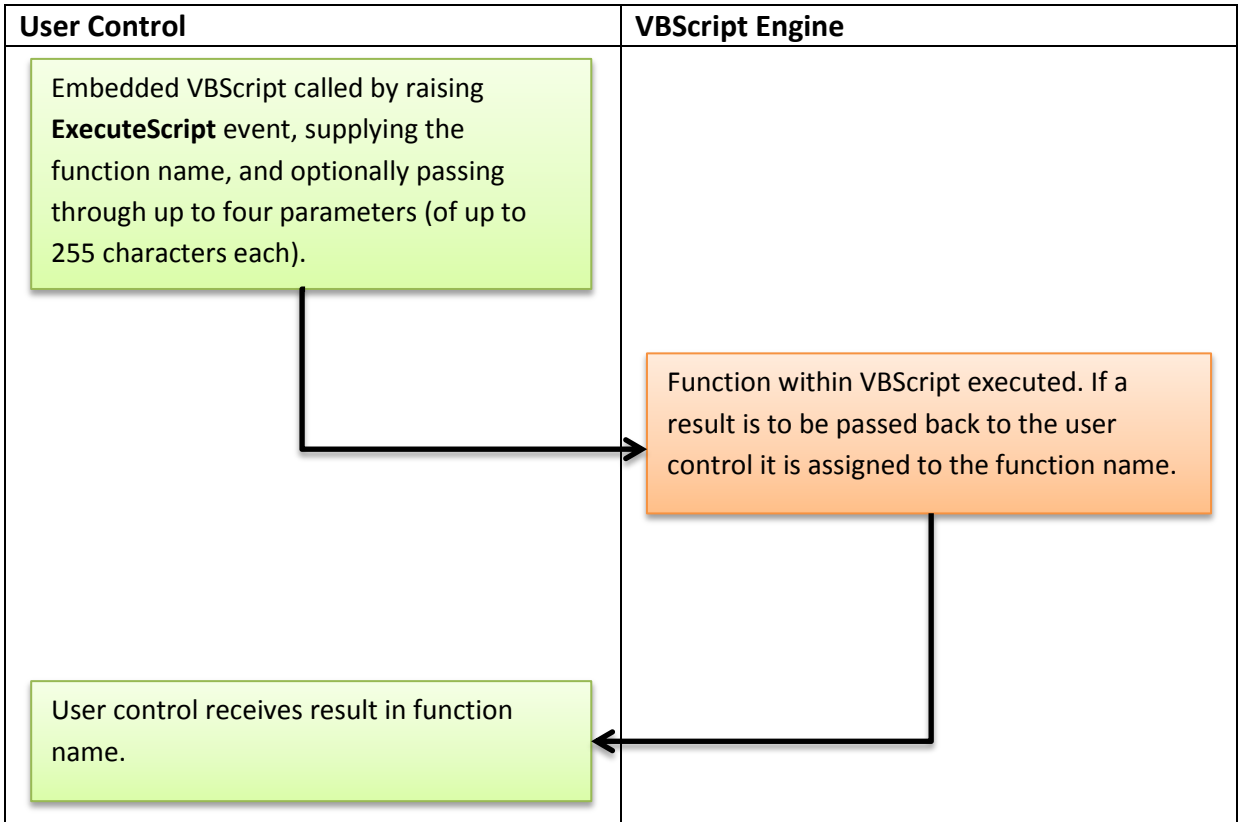
Schematic 2 - Invoking methods in the User Control from a SYSPRO Form



Schematic 3 - Passing Information from a Form Field to the Customized Pane



Schematic 4 - Interacting with an Embedded VBScript



Chapter 17 - Executive Dashboards

SYSPRO *Executive Dashboards* are a means of visually representing data (usually from SYSPRO) so that it becomes meaningful information. The dashboards can be in the form of gauges, flowcharts, indicators, trend analysis, comparisons, and there are even some that enable what-if analysis. The dashboards are also used to highlight areas of the business that may need attention and prompt further investigation. **Figure 17-1** shows the standard gauge *Executive Dashboard called GL – Gauges – Period & budget comparison for Current Assets*.

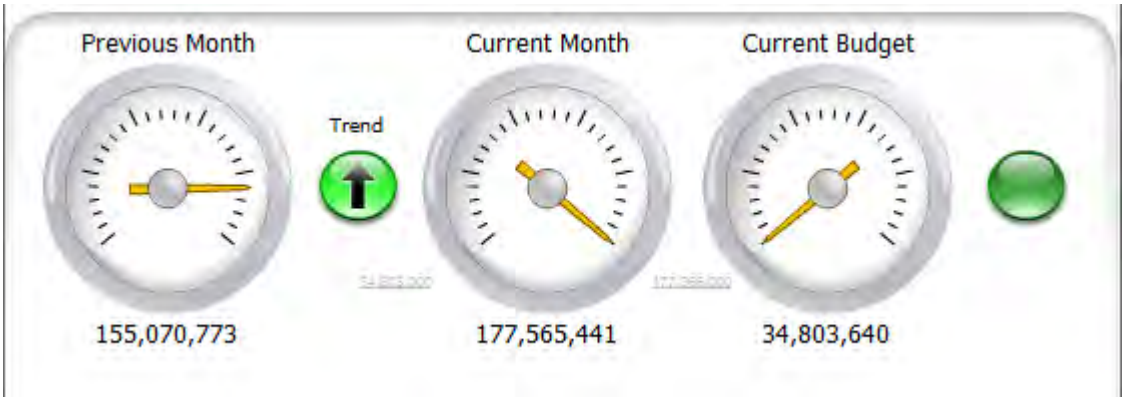


Figure 17-1: *The GL – Gauges – Period & budget comparison for Current Assets dashboard*

Within the *Customized Pane Editor* program, the *Object type* of *Executive Dashboard* can only be selected if *Executive Dashboards* has been licensed.

There are over 300 standard executive dashboards that ship with SYSPRO. If these do not suit your requirements you can tweak one of the existing ones, or build your own from scratch. To build your own you will require access to a product called *Crystal Dashboard Designer* (previously known as *Xcelsius*), as well as access to a version of Microsoft Excel that matches the version of *Crystal Dashboard Designer*.

An *Executive Dashboard* consists of two parts, a *Shockwave* (`.swf`) file created using a product called *Crystal Dashboard Designer*, and VBScript. The VBScript retrieves the latest copy of the `.swf` file from the server, then calls the business object and stores the results in a predetermined filename. The

VBScript passes the `.swf` file to the customized pane and *Shockwave* uses the output from the business object to render the dashboard.

Using a Standard Executive Dashboard

When adding a customized pane and the *Object type* of *Executive Dashboard* is selected, several of the items on the *Pane Properties* pane are available for use. These include the *Show icon*, *Icon*, and *Show toolbar* options under the *Pane Caption* section. The *Toolbar control 1* and *Toolbar control 2* sections are also available.

Within the *Refresh details* section the *Ignore OnLoad VBScript* checkbox is present so that you can prevent the *Executive Dashboard* from firing during the initial load of the program within which it resides. If this option is checked, the dashboard's *OnLoad* event will only fire the first time that the customized pane's *OnRefresh* event is fired (either by the operator clicking on the *Refresh* button, another pane causing it to refresh, or the timer causing it to automatically refresh). Some dashboards can take a significant amount of time to retrieve the data and render the dashboard, and control is not returned to the program until the dashboard has completed loading. The *Automatic refresh*, *Refresh time*, and *Refresh period* options work in the same way as with other customized pane types.

While adding the customized pane, the area below the *Pane Properties* pane will contain several hundred pre-built dashboards. Moving the mouse pointer over the name of each of these will display a tooltip containing a detailed description of what it does, and which values are included (see **Figure 17-2**).

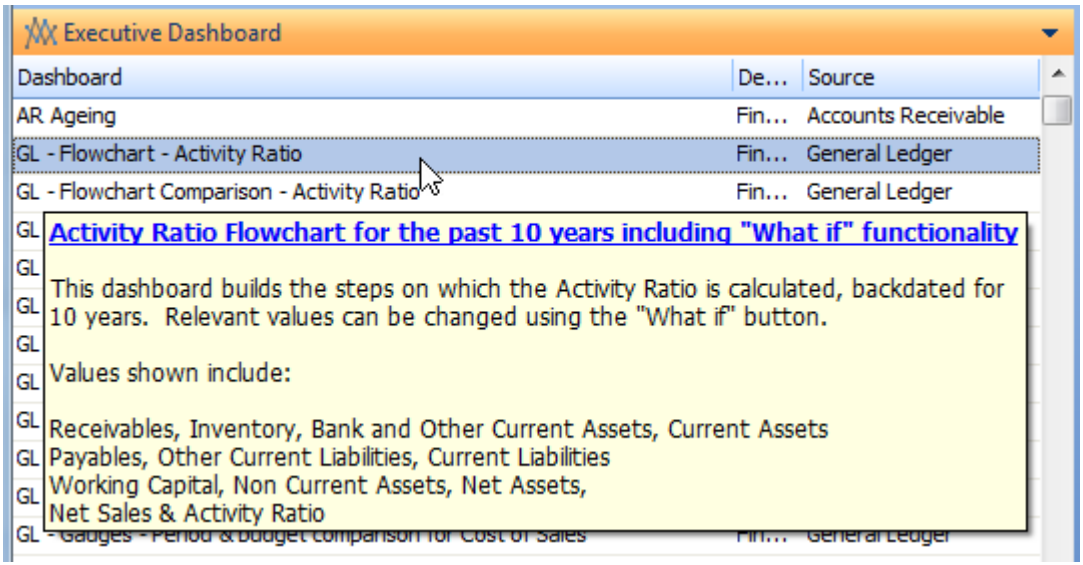


Figure 17-2: The list of dashboard templates displaying the description for *Activity Ratio*

Double-clicking on the name of any of these templates causes a preview of the dashboard to appear in the *Preview* pane on the right of the screen, and the *Window title* field to contain the name of the template. If this template contains the information that you want, in the correct format, just click on the *Save* or *Save and Exit* buttons. Clicking on the *Save* button creates the customized pane, leaves you in the *Customized Pane Editor* and clears the settings ready for adding another one. The *Save and Exit* button saves the customized pane and exits the *Customized Pane Editor*.

Once an *Executive Dashboard*, customized pane has been saved, if you call up its *Customized Pane Properties* screen you will not be able to change to another template, but you can edit the VBScript.

Executive Dashboard Architecture

A schematic appears at the end of this chapter (*Schematic 1 – The Architecture of a Standard Executive Dashboard*) showing the steps below.

1. When the customized pane is launched its *OnLoad* event fires, and any code against its *OnLoad* function is actioned.

2. The *OnLoad* function builds the XML required to call the ***COMGET*** business object. This is used to retrieve the latest version of the .swf file from the server and store a temporary copy on the client. The file is stored in a folder called *Dashboards* under SYSPRO's `Base\Settings` folder. The temporary name of the file is built using `exec` followed by an underscore, the content of the *SYSPROUserNumber* variable, an underscore, the content of the *dashname* variable and the `.swf` suffix. An example using the *dashname* of *gl_activity_ratio* and the SYSPRO user number of *0918* appears below:

   ```
   ..\Base\Settings\Dashboards\exec_0918_ gl_activity_ratio.swf
   ```

3. The ***COMGET*** business object retrieves the `.swf` file and stores it in the specified location with its temporary name.

4. The *OnLoad* function calls the *OnRefresh* function.

5. The *OnRefresh* function builds up the XML required to call the business object that is used by the customized pane to retrieve the data, and then calls the business object. The results from the business object are written to the same *Dashboards* folder as the `.swf` file. The filename is typically built using `exec`, and underscore, the contents of the *dashname* variable, and underscore, the text output, with the `.xml` suffix.

6. As the *OnRefresh* function finishes it passes control back to the *OnLoad* function. This builds up the name and path of the `.swf` file and passes it to the *BrowserAddress* variable where the `.swf` file is displayed in a browser control.

Building your own Executive Dashboard

It is possible to build your own *Shockwave* file, but this requires a product called *Crystal Dashboard Designer*, and Microsoft Excel. The basic steps appear below, but a detailed description of using *Crystal Dashboard Designer* is beyond the scope of this book.

A folder called *ExecDashboards* under the SYSPRO `Base` folder contains the standard *Executive Dashboard* components. If it does not already exist, create a folder called *ExecCustomized* at the same level. When the customized pane is looking for components, this folder is always searched first. If components are found in this folder they are used instead of those in the standard folder. Putting your components in the customized folder guarantees that they will be the ones used, and that your components will not be overwritten by standard SYSPRO ones if updated with the same name.

1. Establish which business object will return the information that is required for the *Executive Dashboard*.

2. Using the schema that ships with the business object, establish which elements will return just the information that is required. Note that some elements default to returning values, and others don't. The description against the element in the schema will specify the defaults for each element.

3. Use the *e.net Diagnostics* utility to call the business object and return the data. Save the input XML and output XML to text files in the *ExecCustomized* folder.

4. Load Microsoft Excel and import the output XML (*Data* tab | *From Other Sources* dropdown | *From XML Data Import* option | locate the XML output file | select to create a schema). Save the `.xslx` file to the *ExecCustomized* folder.

5. Load *Crystal Dashboard Designer*. Use the *Import* option (*Data* | *Import*) to import the `.xslx` from the *ExecCustomized* folder. The content of the Excel spreadsheet populates the *Crystal Dashboard Designer* spreadsheet. This can be seen in **Figure 17-3**.

6. Add the gauges/charts/other components to the workspace, and link them to the relevant cells of the displayed spreadsheet. Save the dashboard (`.xlf` file) to the *ExecCustomized* folder, but remain in the *Crystal Dashboard Designer* product.

7. Call up the *Data Manager* screen (*Data* | *Connections*) and use the *Add* button to call up the *Excel XML Maps* option. Change the *XML Data URL* prompt to point at the output from the business object on the *Definitions* tab. Optionally, set a refresh interval on the *Usage* tab. Save the changes that have been made but remain in the program.

8. Export the design to a *Shockwave Flash* file (*File* | *Export* | *Flash (SWF)*) and save the file to the *ExecCustomized* folder.

9. Create a customized pane with the *Object type* of *Executive Dashboard*. Supply a title and use the *Edit VBScript* option to call up the *VBScript Editor*. Probably the best way to build the VBScript is to copy one of the existing ones and tweak it to use the names of the `.swf` file, and call the correct business object using the input XML you created earlier.

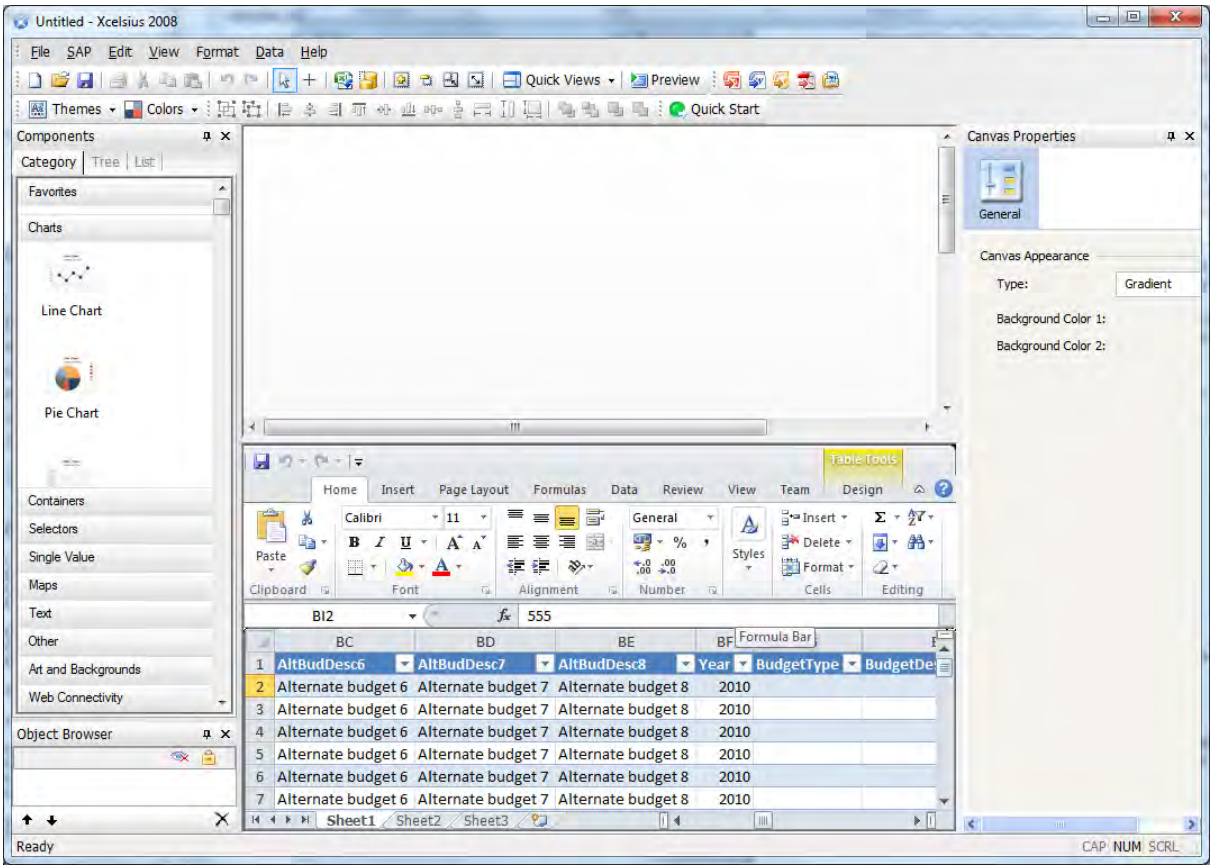
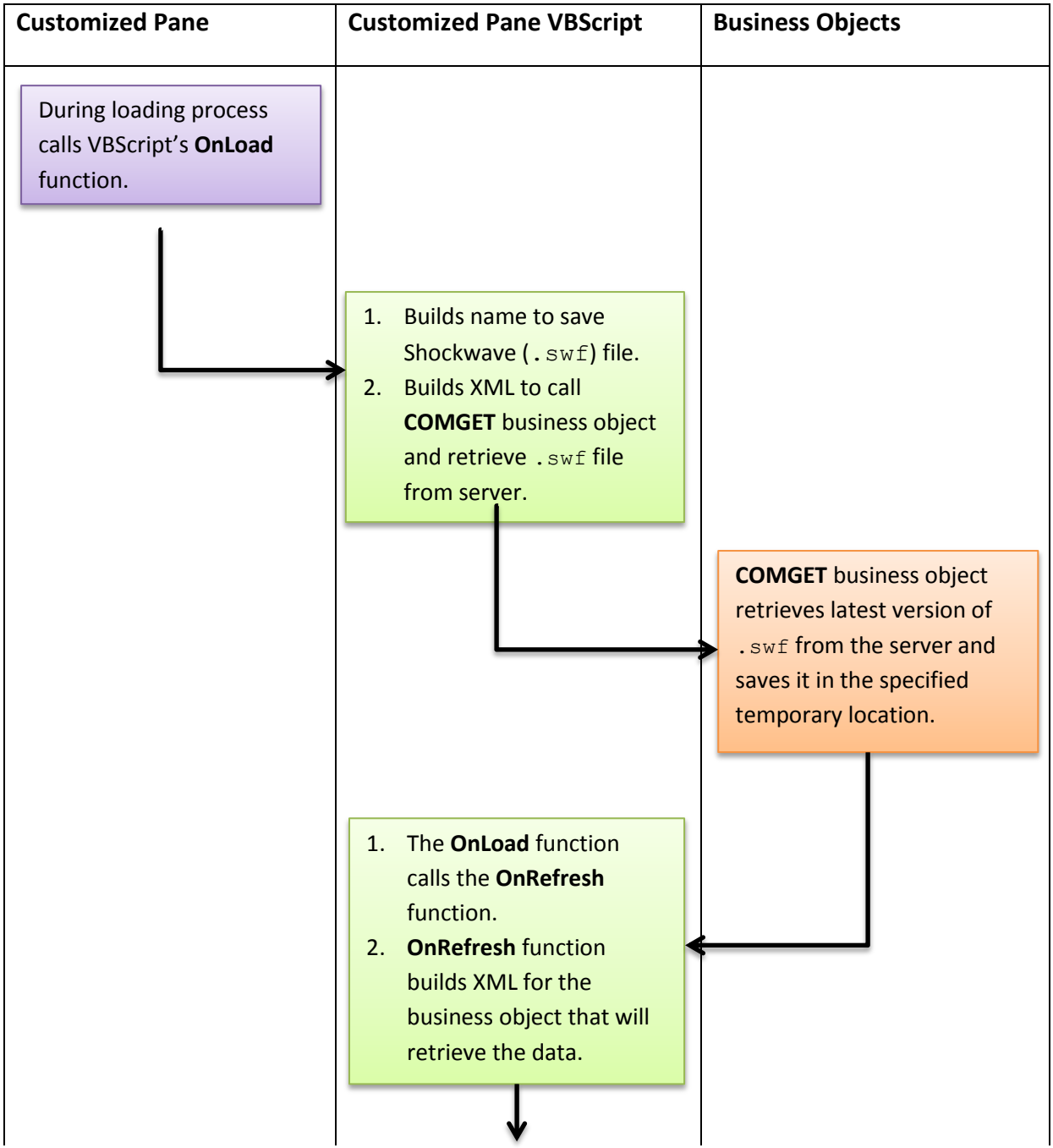
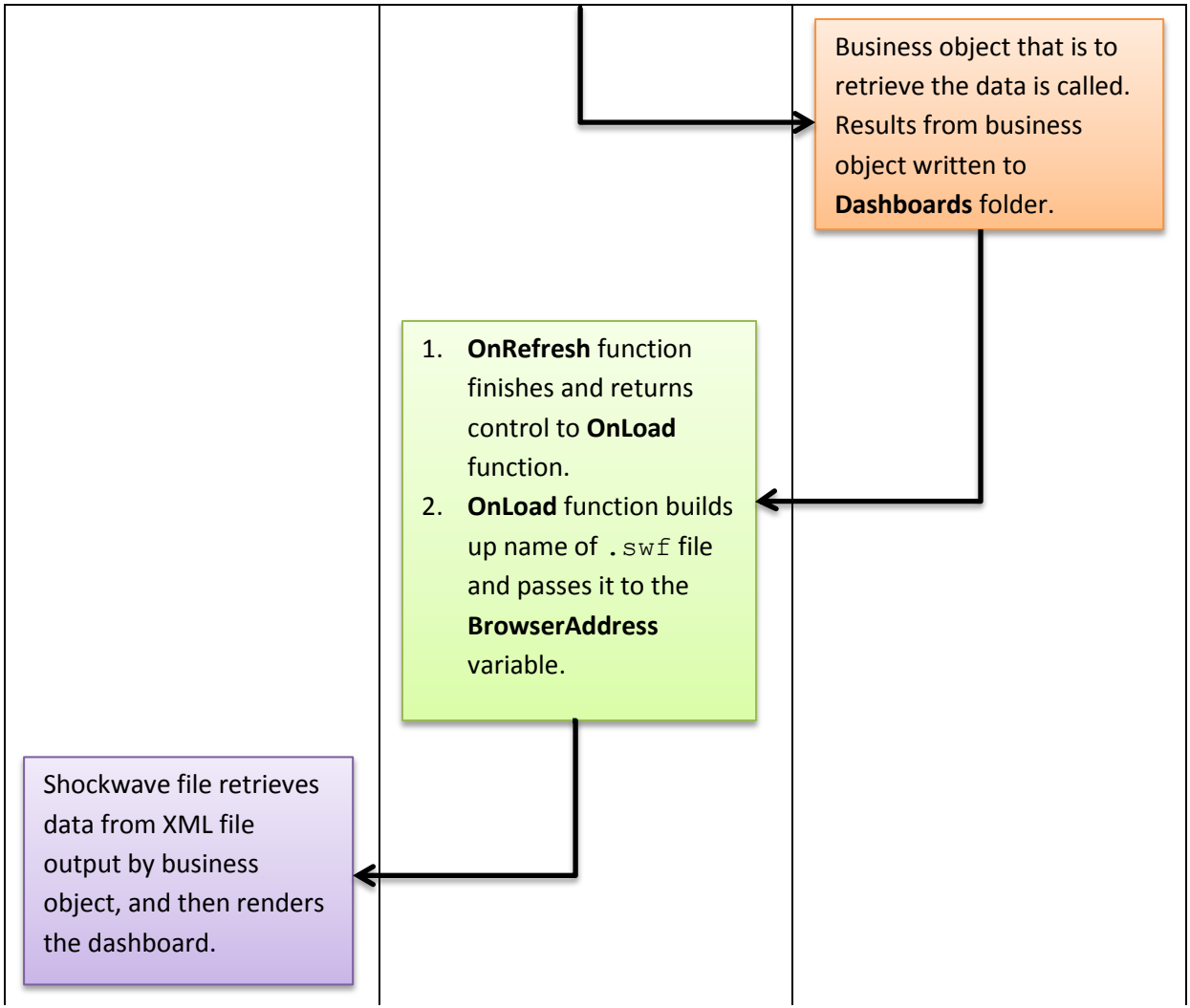


Figure 17-3: The Excel spreadsheet populating the *Crystal Dashboard Designer* spreadsheet

Schematic 1 – The Architecture of a Standard Executive Dashboard




Business object that is to retrieve the data is called. Results from business object written to **Dashboards** folder.
1. **OnRefresh** function finishes and returns control to **OnLoad** function.
2. **OnLoad** function builds up name of `.swf` file and passes it to the **BrowserAddress** variable.
Shockwave file retrieves data from XML file output by business object, and then renders the dashboard.

Chapter 18 - Tiles and Favorites

The SYSPRO *Main Menu* has been significantly enhanced for SYSPRO 7. One of the biggest enhancements is the way that the *Favorites* and *Navigation Pane* work with tiles. This chapter covers the *Favorites* pane and tiles.

Favorites

The *Favorites* pane is where you can create and store shortcuts to SYSPRO programs, reports, and other applications. These shortcuts are associated with *Tiles*. The *Favorites* pane can be divided up into sections called Categories that enable you to logically group your tiles. These categories can be collapsed or expanded as required.

The tile can be used as a placeholder to display images, information, as well as information displayed within its tooltip. A refresh time can be set against individual tiles so that the displayed values are updated on a regular basis.

An example of the *Favorites* pane can be seen in **Figure 18-1**. Three categories have been created called *Favorites*, *Values*, and *Old Style Favorites*, and the tiles/shortcuts added to them. The tiles within the *Favorites* category have had images applied to them. The *Welcome to SYSPRO* tile has a tutorial video associated with it that will play when clicked. The *Critical Warehouses* tile has a tooltip added to it that displays the warehouse values for the critical warehouses, and this is updated every 30 seconds.

Prior Versions
Prior to SYSPRO 7 the *Favorites* pane was a simple folder containing shortcuts to SYSPRO programs, customized reports, SRS reports, and custom applications. Creating new shortcuts was accomplished through a wizard after right-clicking on the *Favorites* pane. SYSPRO programs and reports could be created through the wizard, or the item could be dragged from the *Main Menu* to the *Favorites* folder. The programs and reports that were dragged to this pane would keep their existing icon, and icons for shortcuts that were created were selected during the wizard. Each of the shortcuts could have their description and icon changed. The name of the application, report, or SYSPRO program to be run could also be changed, but the type of shortcut would remain as it was created.

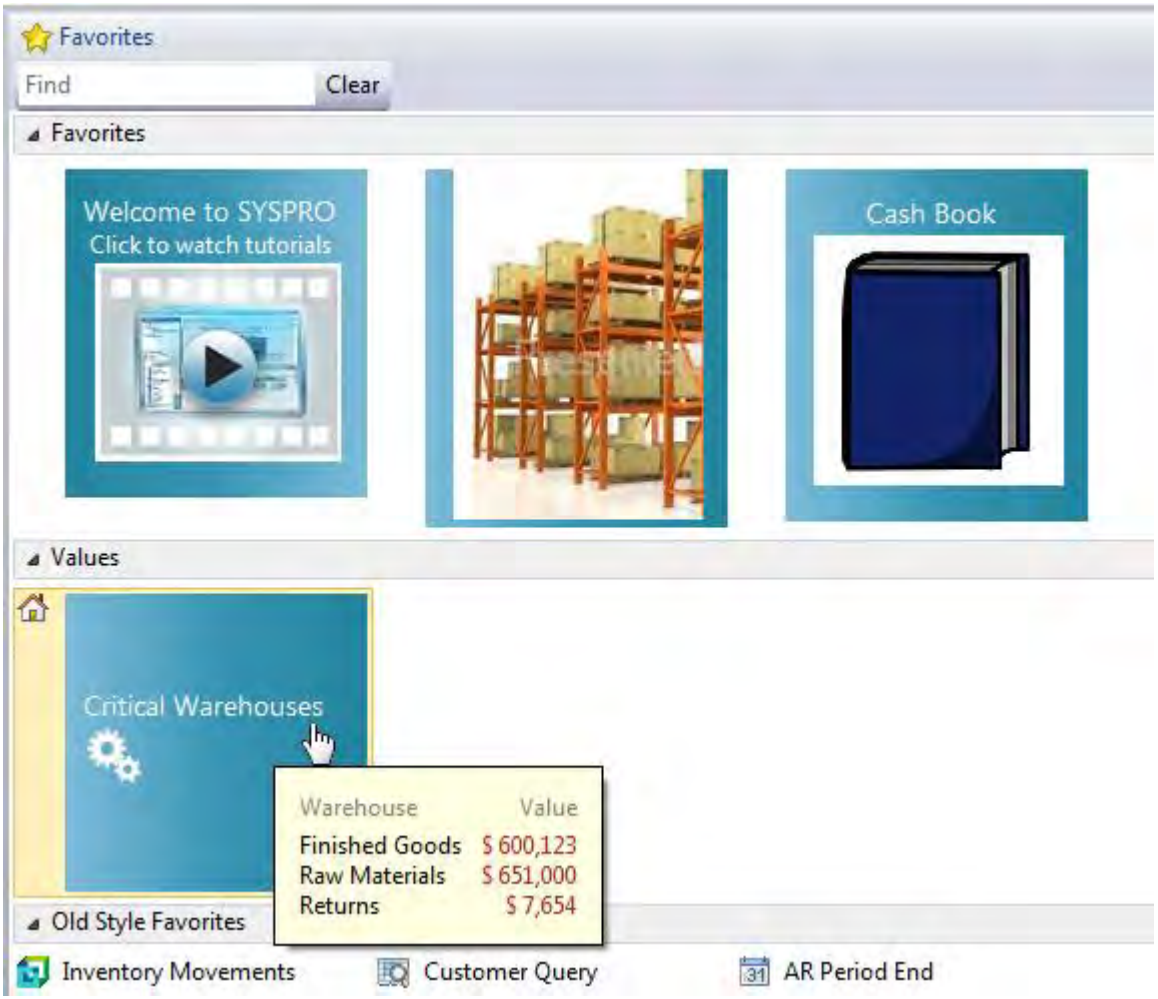


Figure 18-1: Some examples of what can be done with the *Favorites* pane

The new *Favorites* pane can be configured to look and work in exactly the same way as versions prior to SYSPRO 7, if so desired. The *Add New Shortcut Wizard* also works in exactly the same way as prior versions. However, the wizard creates a tile instead of a menu option. See the section below called *Changing a Tile to Resemble a Prior Version's Shortcut* if you want the shortcut to look exactly the same as prior versions.

In **Figure 18-1**, the tiles within the *Old Style Favorites* category have had their appearance changed to look the same as the prior versions. Note that any existing *Favorites* that are converted from SYSPRO 6.1 to SYSPRO 7 will appear this way by default, and can be changed later.

Categories

Categories are used for grouping tiles in any way that suits your requirements. The *Favorites* pane must contain at least one category. If you delete the last remaining category in the *Favorites* pane another one is immediately created for you. Categories are added by right-clicking anywhere on the *Favorites* pane and selecting *New*, *Category* from the context-sensitive menu. The new category will be added to the pane below the existing categories and tiles, and the cursor will be placed at the beginning so that you can enter the category's name (see **Figure 18-2**). When you have finished naming the category, press the *Enter* key and the category name will be saved.

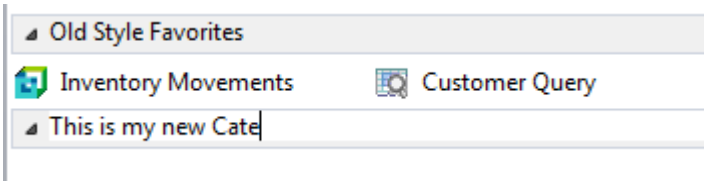


Figure 18-2: Adding a *Category*

An existing category can be renamed by right-clicking on it and selecting *Properties*, *Title* from the context-sensitive menu that is displayed, then replacing the text with the new name (see **Figure 18-3**).

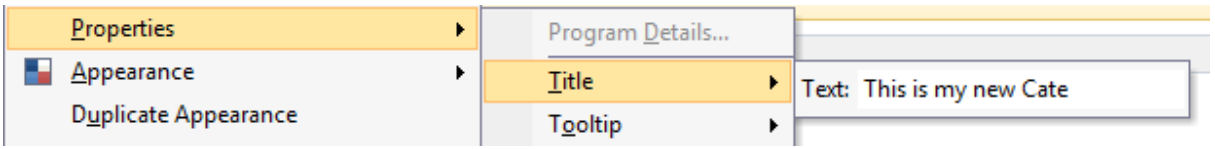


Figure 18-3: Changing the name of a category

The sequence of the categories can be changed by clicking and dragging them to a new location. The category and its contents will be moved to the new location.

A category can be expanded to show its contents, or collapsed so that it only shows the category header. **Figure 18-4** shows the same pane as in **Figure 18-1**, only with the *Values* category collapsed.

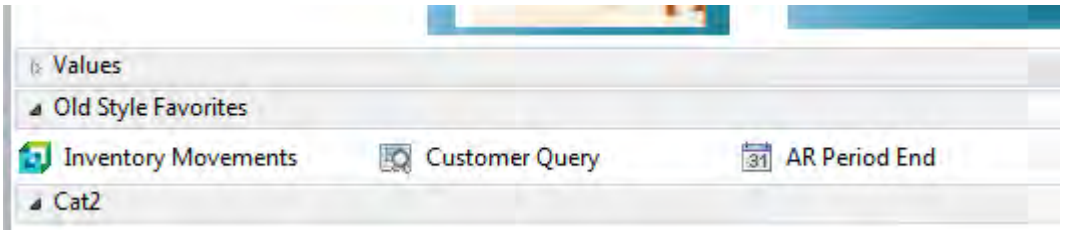


Figure 18-4: The *Favorites* pane with the *Values* category collapsed

A category can have its own icon, which is added by right-clicking on the category name and selecting *Appearance* | *Icon* from the context-sensitive menu, and then selecting the icon from the displayed list. A XAML theme can also be applied to a category by selecting *Appearance* | *Style* from the same menu, and if a style is chosen the *Color* option becomes available against *Appearance*, so a color can be chosen from the displayed list. Note that the height of a category item cannot be adjusted so some XAML themes may cause the caption not to be displayed correctly; the XAML theme *Category* is designed specifically for *Category* items.

Categories can be removed from the *Favorites* pane using the *Delete* option from the context-sensitive menu that is displayed when you right-click on the category name. You will be prompted as to whether you want to continue. Deleting a category will delete all the tiles/shortcuts associated with this category.

The *Undo Delete* option is also available from this menu, and it will reinstate deleted categories (and their contents) while you are still within SYSPRO. Once you have exited SYSPRO the *Undo Delete* option will not allow you recover the categories deleted in the previous session.

Changing the Appearance of the Favorites Pane

The default appearance of the *Favorites* pane is called *VS Studio 2010* which gives a white background with medium gray category headers. This can be changed by right-clicking on the pane and selecting the *Layout* option from the context-sensitive menu. The *Layout* option has an *Appearance* dropdown list containing themes. Selecting one of these will change the appearance of the whole pane. However, if a category's header has previously been changed manually, that category header will not be affected by the selected theme. The *Favorites* panes appearance can be set back to its default by selecting *VS Studio 2010*.

Against the *Layout* option is the ability to change the width of the columns containing the tiles, and the margins around the tiles. There is also an *Auto Fit For All Items* option which will select the best size for the spacing of the tiles, and the size of the categories. The last option against *Layout* is *Reset Layout* which will put everything back to its default settings, but not change your selected theme.

Adding to the Favorites Pane by Dragging a Menu Item

Standard SYSPRO programs and reports can be added to a category within the *Favorites* pane by dragging them from the *Program List* or the *Recent Programs* list to the desired location. When the mouse pointer hovers over the category, the vertical location where the tile will be created is highlighted by a solid line. The position of the mouse pointer will reflect the horizontal position. All items that appear after this location (both vertically and horizontally) will be shuffled along to make room for this to be inserted.

The created tile will contain the same description as the menu item from which it was dragged, and the tooltip will contain the same long description as appeared in the tooltip against the original menu item. The color will reflect the type of menu item (program or report).

Tiles can be deleted from the category by right-clicking on them and selecting *Delete* from the context-sensitive menu. If you delete a tile by mistake, the *Undo Delete* option can be selected from the same menu and the tile will be replaced from where it was deleted (whilst you are still logged into SYSPRO).

Menu items can be dragged either from the *Program List* or *Recent Programs* list; parent folder items cannot be dragged and dropped onto the *Favorites* pane.

Adding a Tile Using the Wizard

New tiles are added to the *Favorites* pane by right-clicking on the pane's background, a category header, or another tile, and selecting *New*, *Program Tile* from the context-sensitive menu. The *Add New Shortcut Wizard* screen is displayed where you can add a SYSPRO program, a SYSPRO customized report, a Net Express program, any other application, or an SRS report (see **Figure 18-5**).

Figure 18-5: The *Add New Shortcut Wizard*

If you select to add a SYSPRO program you are prompted for the program name (which has a browse) and description (which defaults to SYSPRO's description for this program). Then the tile is added. The tooltip is automatically added that contains the full description that would appear against this program in the standard SYSPRO menu. If you select to add a SYSPRO customized report you are prompted to enter the name of a report writer report (which also has a browse) and the description (which defaults to the report's description). Then the tile is added. If you select to add a Net Express

program you must enter the program's name and path (which also has a browse) and the description which defaults to the program name. The tile is then added. The default tile color for all the above is *Teal*.

If you select to add *Any other application* you are prompted for the application name (and optional path) along with a description. Then the tile is added for you. The default color of an *Any other application* tile is *Black*. If you select to add a *SYSPRO Reporting Services report* you are prompted to choose between standard SRS reports and user-defined ones. A list of the relevant reports is displayed and you select from this list. You are prompted to supply a description, which defaults to the report's description. The tile is then added for you and will appear in *Dark Green*.

Changing a Tile to Resemble a Prior Version's Shortcut

It is a simple process to change a tile to resemble a shortcut on a prior version's *Favorites* pane. Right-click on the tile and select *Appearance* | *Icon* from the context-sensitive menu, and choose the required icon from the list. Right-click on the same tile and select *Appearance* | *Style* from the same menu. Highlight any of the displayed styles, then click on the *No XAML Theme* option. The items that have changed will resemble those in **Figure 18-6**.

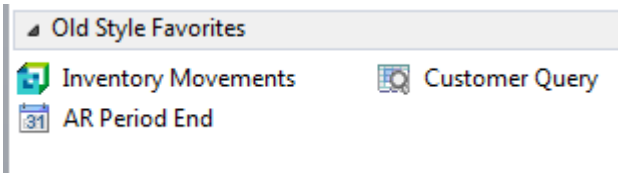


Figure 18-6: Reproducing the old style *Favorites* pane

Adding a Blank Tile

Blank tiles are tiles that are not associated with one of the shortcut types created by the *Add New Shortcut Wizard*. Blank tiles are added by right-clicking on the background of the *Favorites* pane, one of the category headers, or on one of the tiles, and selecting *New* | *Blank Tile* from the menu. Supply the description for the tile and press the *Enter* key, and the tile will be created for you. A blank tile can also be added with no description if required (although if you go to change the tile's description using the *Properties* | *Title* option you will see that *Text* will contain *<!--*).

If you create a blank tile and later decide that you want to associate a shortcut with it you can right-click on the tile and select *Properties* | *Program Details* from the displayed menu. This invokes the *Add New Shortcut Wizard*. Follow the steps that appear in the section *Add a Tile Using the Wizard* above.

Changing the Appearance of a Tile

At any time the tile's appearance can be changed using the *Properties* section of the context-sensitive menu that appears when you right-click on a tile.

The screen displayed by the *Program Details* option will depend on the current settings of the tile. If it is a blank tile (or one that only has VBScript against it) the *Add New Shortcut Wizard* will be invoked. If it is already associated with a SYSPRO program name (or Net Express program) the screen will allow you to change the program name, description, and icon. If it is associated with another application the screen will allow you to change the application, location, description, and icon. If the tile is already associated with an SRS report you can changed the description and icon. If it is already associated with a Report Writer report the screen will enable you to change the report to be run, its description, and the icon.

The tile's description can be changed using the *Title* option and replacing the value that is already present against its *Text* prompt.

The *Tooltip* text that is displayed when the mouse pointer hovers over the tile can be changed, removed, or replaced with an image.

An image can be placed on the tile using the *Attach Picture* option, and removed if one is present.

The tile height can be adjusted using the slider bar against the Item *Height* option.

The *Auto Fit* option will make the tile just tall enough for what must be displayed on it (description, image, etc.). If there is another row of tiles beneath this row they will all start below the longest one on this row.

The *Macro* option on the *Properties* menu will be covered in detail within the *VBScripting for Favorites* section below.

A tile's appearance can also be changed using the *Appearance* option from the context-sensitive menu. By default the *Icon* and *Style* dropdown lists are enabled, and the *Color* dropdown list is only enabled if the *Style* is set to anything other than *No XAML theme*. The *Icon* dropdown list enables you to choose an icon that will be displayed alongside the tile. The *Style* dropdown enables you to choose a XAML theme for your tile. The XAML theme will display the description, along with the image associated with this style (although some themes do not have an image associated with them, such as *Rounded* that gives the tile rounded corners).

The *Duplicate Appearance* option will take the currently selected tile's *Style* and *Color* and apply it to all the other tiles in this *Category*.

The *Duplicate Height* option will take the currently selected tile's height setting and apply it to all the other tiles in this Category.

The *Large Icons* option toggles between small icons against the tiles, and large ones.

The final item to affect the appearance of a tile is the *Show Program Names in Tooltips*. This adds the name of the program to be called to the existing tooltip. In **Figure 18-7** the program name of ***CSHP01*** is displayed in the tooltip when the mouse pointer hovers over the *Cash Book Period End* tile.

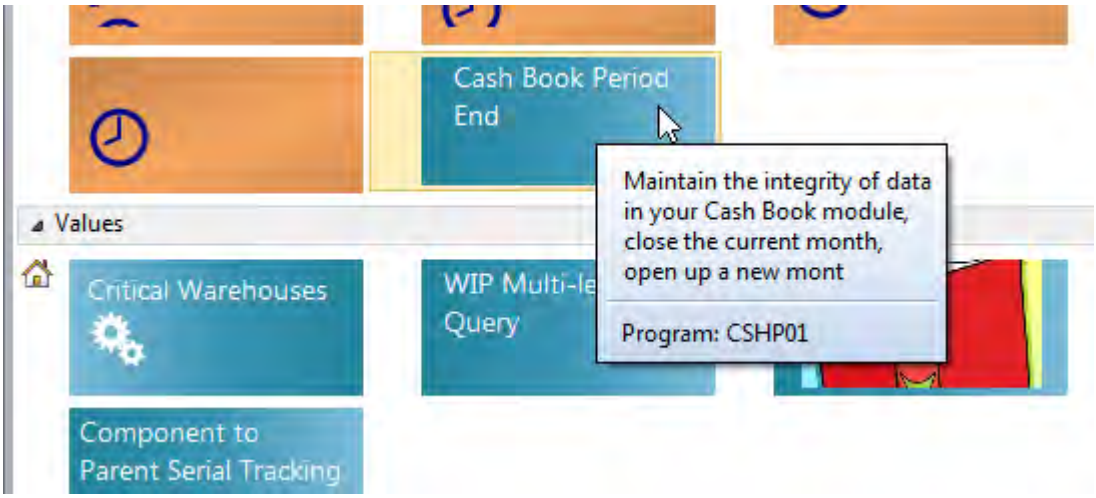


Figure 18-7: The *Program Name* and program's long description in the tooltip

Changing the Appearance of the Favorites Pane

As with the individual tiles, the *Favorites* pane's appearance can also be changed. This is done using the *Layout* option from the context-sensitive menu, which has five options. These are: *Appearance*, *Item Width*, *Item Margins*, *Auto Fit for all Items*, and *Reset Layout*. The *Appearance* option has a dropdown list of themes, with the default of *VS Studio 2010*. Choosing one of these themes will change the look of the *Favorites* pane (background, category headers, etc.) except for those that have already had changes made to them. These updated items will retain their existing changes.

The *Item Width* option has a slider bar that is used to change the width of the columns containing the tiles. If the *Item Width* is set to be less than the width of a tile, the tile's width will be reduced. If the *Item Width* is set to be greater than the width of a tile, the desktop of the *Favorites* pane will appear between the end of this tile and the start of the next one. The *Reset* option directly below the slider will set the *Item Width* back to its default position.

The *Item Margins* option also has a slider, which is used to increase/decrease the margin between this tile and the next (both to the right, and below). When the margin is increased the columns containing the tiles remain in the same location, so the increased margin is created by subtracting from the tile.

The *Auto Fit for all Items* option attempts to organize the tiles to fit as best it can, given the constraints imposed by the *Item Width*/*Item Margins* options.

The *Reset Layout* sets the tile widths and margins to their default settings. This can be used in conjunction with the *Auto Fit for all Items* option to return to a "sane" setting.

The *Expand only Selected Category* option changes the *Favorites* pane so that only the currently selected category appears expanded, and the others are automatically collapsed. The expanded category takes up all of the space in the *Favorites* pane that is not used by the collapsed category headers. If you click on another category header, this will expand and the previously expanded one will collapse. **Figure 18-8** shows the *Expand only Selected Category* option checked.

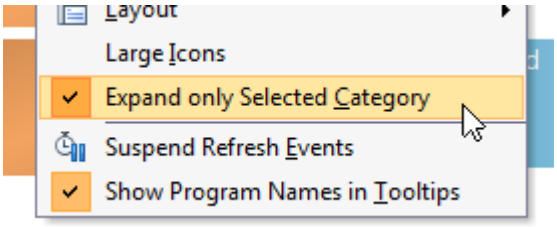


Figure 18-8: The *Expand only Selected Category* option on the context-sensitive menu

Refresh Events

Tiles that have VBScript code configured against them can be configured to fire their *OnRefresh* event at a set time interval. This is set in minutes, and can contain up to three decimals. As an example, 0.5 would be every 30 seconds, and 1440 would be once every 24 hours.

If you right-click on a tile that has an *OnRefresh* event configured against it, the displayed menu will include a *Refresh* option that will cause the tile's *OnRefresh* event to be fired immediately.

On the same menu, but available when right-clicking anywhere within the *Favorites* pane, is the option to *Suspend Refresh Events*. If this option is selected it stops the refresh events from firing until it is unselected, or until you exit SYSPRO. The next time that you call up SYSPRO this option will have been unselected, and events will fire again.

If you attempt to add/edit a VBScript associated with a tile, you will be prompted that the refresh events will be suspended (see **Figure 18-9**). This sets the *Suspend Refresh Events* option mentioned in the previous paragraph. Once you have completed the changes to your script you can re-enable the refresh events by deselecting this option.

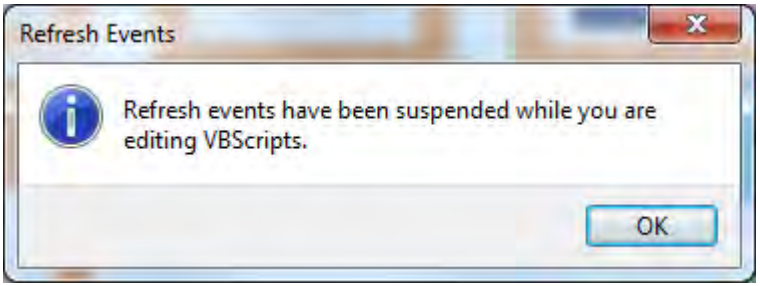


Figure 18-9: Notification that the refresh events will be suspended when editing a VBScript

Export/Import

Against the *Export/Import* option are options to export and import tiles, as well as to export and import menus. These can be seen in **Figure 18-10**.

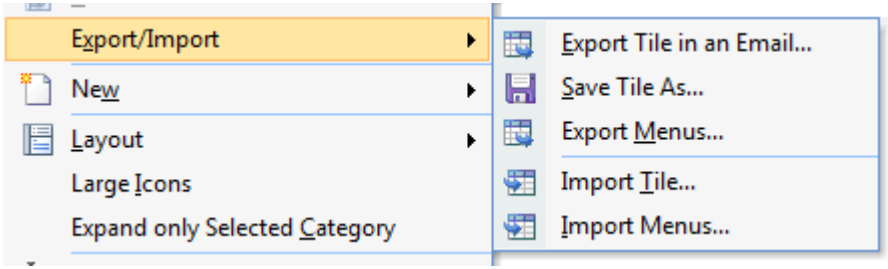


Figure 18-10: The options to *Export/Import* tiles and menus

The *Export Tile in an Email* option can only be used if the operator has the option *Fax/mail integration required* checked (*Operator maintenance* | *Options* tab). If this is not checked a message stating that this cannot be done will be displayed (see **Figure 18-11**).

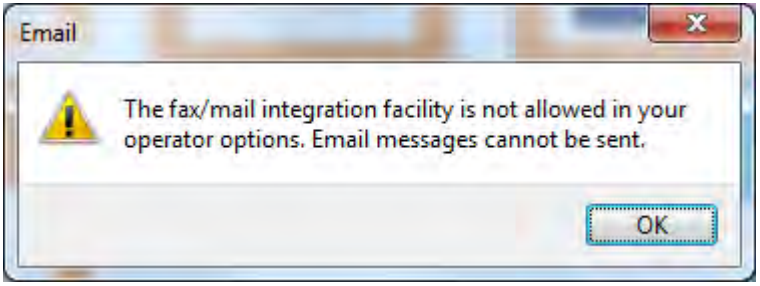


Figure 18-11: The message displayed if fax/mail integration is not configured for this operator

When a file is exported using this option it appears against the email as an attachment. If this is being sent to another system, you should make sure that any Report Writer reports, applications, or SRS reports that are being used by the tile exist on the other machine, as they will not be included in the export file. Note that any VBScript code that exists against this tile (such as against the *OnLoad*, *OnRefresh*, *OnClicked* events) is included in the exported file.

The *Save Tile As* option prompts for a location and filename, then saves the tile as a `.txt` file.

When a tile is imported using the *Import Tile* option, it is added to the end of the currently focused *Category*, although it can then be moved to the required location.

Exporting a menu is done using the *Export Menus* option. When this is selected a screen is displayed that lists the menus that can be exported (see **Figure 18-12**). The listview contains a list of menus (in this case *Favorites*, and *My New Menu*) which can be selected/deselected from being exported. The *Notes* column against the *Favorites* row shows that there are seven images linked to this menu. If this menu is exported and imported on another system, these images will need to be taken across manually (unless images are linked to the `…\Base\Samples` folder). See the section *Ensuring Image Path Names are Resolved across Different Machines* below for more information. The toolbar also contains a checkbox option to *Email the export file*.

When the file is exported, the selected menus will be written to a file. If the *Email the export file* option is checked on the toolbar, an email will be created that includes the exported file as an attachment.

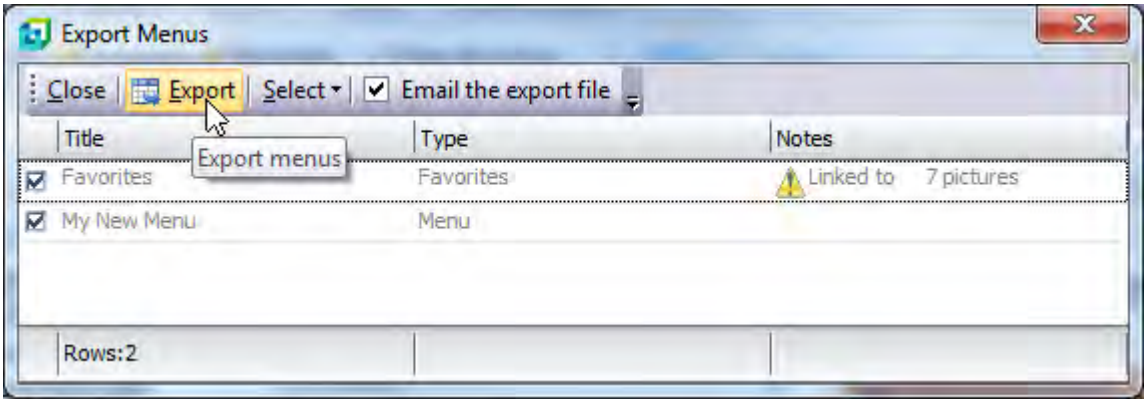


Figure 18-12: The *Export Menus* screen

The *Import Menus* screen is similar to the *Export Menus* screen except that the *Email the export file* option on the toolbar is replaced with the name of the import file (after first being prompted to select the import file). The value against the *Title* column can also be changed. For example, after exporting the menus in **Figure 18-12** the import would contain the entries of *Favorites* (with a *Type* of *Favorites*)

and *My New Menu* (with a *Type* of *Menu*). The *Title* of *My New Menu* can be changed in the listview. Changing the *Title* against the *Favorites* option is ignored during the import.

If you attempt to import a menu where this menu name already exists you will be prompted to rename the menu to be imported. If you rename it the menu will be created with the new name.

When you use the *Import Menus* option and the import file contains a *Favorites* entry and you already have a *Favorites* menu, then your existing *Favorites* menu will be replaced by the imported one.

VBScripting for Favorites

Each tile in the *Favorites* pane can be configured to use one or more of the macro events associated with it. The available events are *OnLoad*, *OnRefresh*, and *OnClicked*. The *OnLoad* event fires as the *Favorites* pane is loaded, and fires only once per run of SYSPRO.

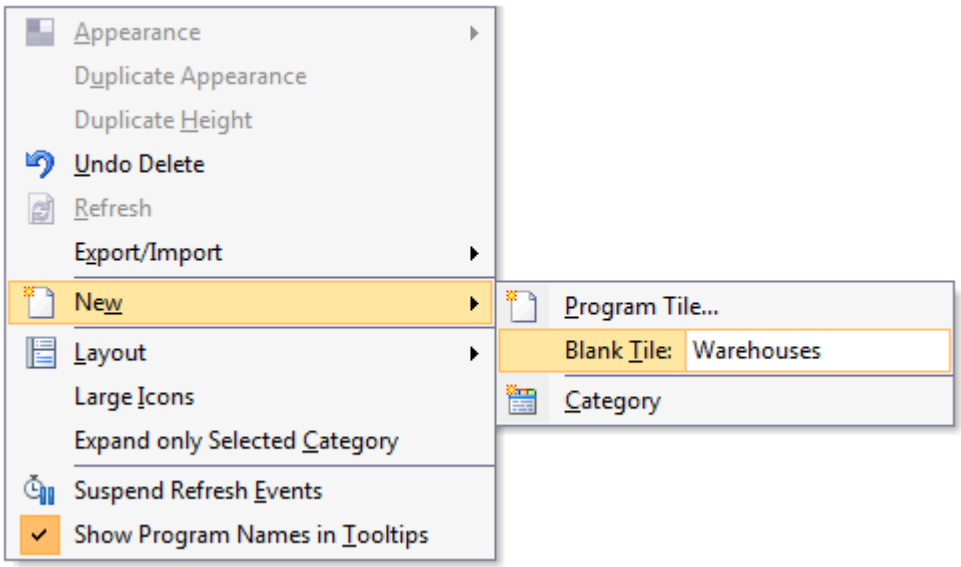


Figure 18-13: Adding a new *Blank Tile*

Unlike a form, a tile's *OnRefresh* event does not fire immediately after its *OnLoad* event; it has to be told when to fire. Each tile has its own timer and a variable can be set to say how frequently this event should fire. The *OnRefresh* event can also be confirmed to fire within the *OnLoad* event's code. There is also an option on the context-sensitive menu against each tile to manually execute the *OnRefresh* event.

The *OnClicked* event fires when you click on the tile. Note that if this is a *Program Tile* (because it has a program, report, or application linked to it) and also has VBScript against its *OnClicked* function, the script will execute when the tile is clicked followed by launching the program.

Figure 18-13 shows a new blank tile being added to the *Favorites* pane by right-clicking on the pane and selecting *New* | *Blank Tile* from the menu. The name of the tile is *Warehouses*, and this tile will be used for all the examples within this section.

Adding Macro Events

Macro events are added to tiles by right-clicking on a tile and selecting *Properties* | *Macro* from the context-sensitive menu (see **Figure 18-14**).

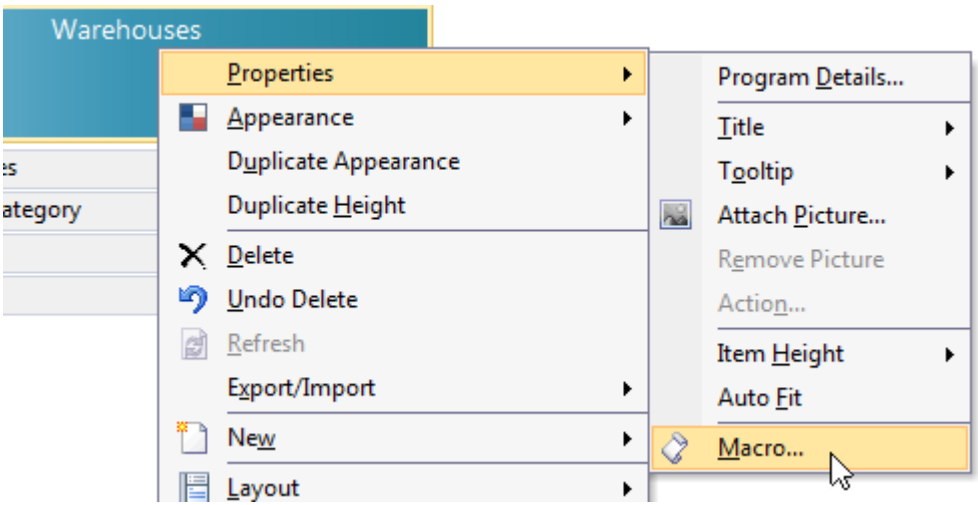


Figure 18-14: Adding a *Macro* event to the *Warehouses* tile

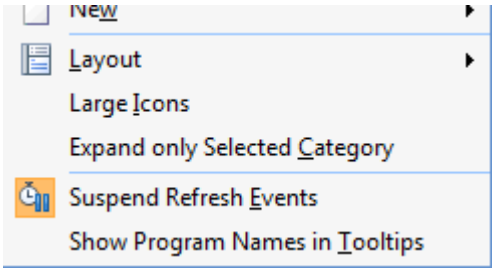


Figure 18-15: The *Suspend Refresh Events* option checked

All tiles can be set to automatically refresh after a specified amount of time, and this is configured using the *RefreshInterval* variable against the tile. As the VBScript against the tiles can be set to perform almost any task, when you click on the *Macro* option the refresh events are suspended for all tiles. A message to this affect is displayed (see **Figure 18-9**), and the *Suspend Refresh Events* option of the context-sensitive menu becomes checked (see **Figure 18-15**).

In the *VBScript Editor* screen, double-click the *OnLoad* event. The *Tile_OnLoad* function will be created for you and the cursor placed within it (see **Figure 18-16**). Using the *VBScript Editor* was covered in great detail in *Chapter 6*. If you have not read this you should read it before continuing.

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function Tile_OnLoad()
6
7  End Function
```

Figure 18-16: The *Tile_OnLoad* function

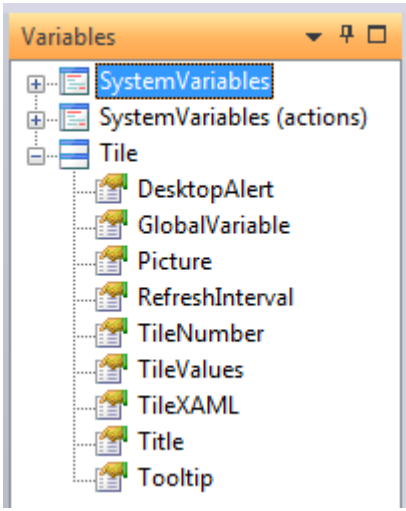


Figure 18-17: The variables under the *Tile* section of the *Variables* pane

The area of interest for tiles is the *Tile* section of the *Variables* pane. The *Variables* pane can be seen in **Figure 18-17**, where the *Tiles* section has been expanded. The *SystemVariables* and *SystemVariables (actions)* sections are covered in *Chapter 6*.

Desktop Alert

The *Desktop Alert* variable is used to pop up a message on the operator's screen when a certain condition has been met. Double-clicking on the *Desktop Alert* variable name displays the *Desktop Alert Settings* screen where you can configure what the alert will contain. **Figure 18-18** shows the *Desktop Alert Settings* screen being populated. The *Duration* is how long the alert will stay on the screen until you interact with it. The *Animation style* specifies how the alert will appear/disappear.

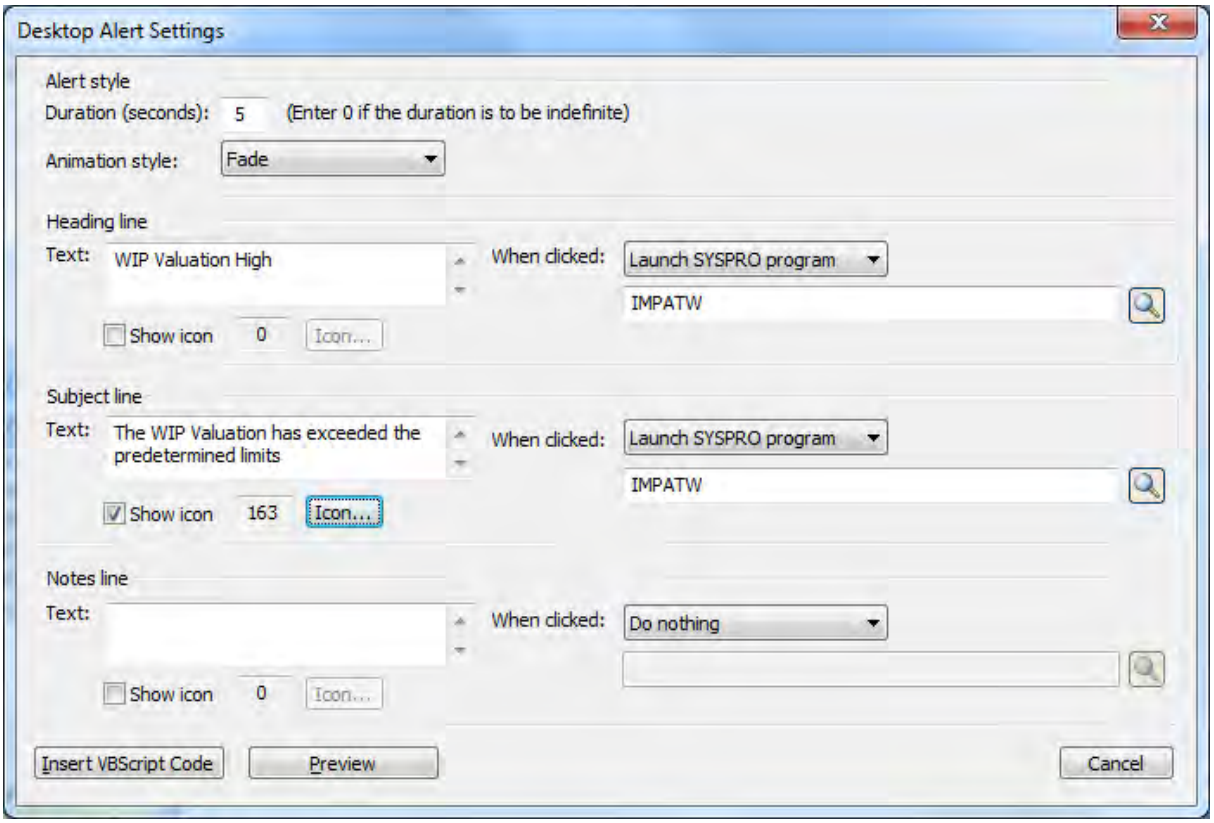


Figure 18-18: Configuring the *Desktop Alert Settings* screen

Against the *Heading line* section you can add a title for the alert against the *Text* prompt, and an icon can be chosen from a list to appear against this title. Against the *When clicked* prompt is a dropdown where you specify what will happen if the operator clicks on the alert's heading. The choices are *Do*

nothing, *Launch SYSPRO program*, or *Run executable*. If either of the last two options are selected a browse enables you to locate the SYSPRO program or executable.

The *Subject line* section has similar options to specify the content of the main body of the alert, and the *Notes line* section has the same for the notes section at the bottom of the alert.

Figure 18-18 shows the *Desktop Alert Settings* screen where it has been configured to state that the WIP valuation is getting too high. If the operator clicks on either the heading or subject line, the *WIP at a Glance* query (***IMPATW***) will be run to show the detail values.

When you have finished configuring the alert, click the *Insert VBScript* button to add the code (see **Figure 18-19**). You would still need to add some code to check the current WIP values and compare these to predetermined values before firing the alert. **Figure 18-20** shows the *Desktop Alert* being displayed. **Figure 18-21** shows the *WIP at a Glance* query that is displayed if the operator clicks on either the alert's title or subject line.

Tile

```
1   ' This script contains functions for form and field events.
2   ' You must not modify the name of the functions.
3   Option Explicit
4
5   Function Tile_OnRefresh()
6     dim Popup
7     Popup = Popup & "<Popup Duration='05' Animation='Fade'>"
8     Popup = Popup & "<Heading Text='WIP Costing High' Icon='163'  />"
9     Popup = Popup & "<Subject Text='The WIP Valuation has exceeded th
10    Popup = Popup & "</Popup>"
11    Tile.CodeObject.DesktopAlert = Popup
12  End Function
```

Figure 18-19: An extract of the code added to perform the *Desktop Alert*

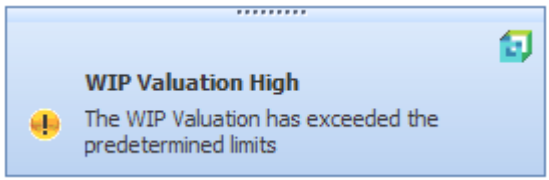


Figure 18-20: The *Desktop Alert*

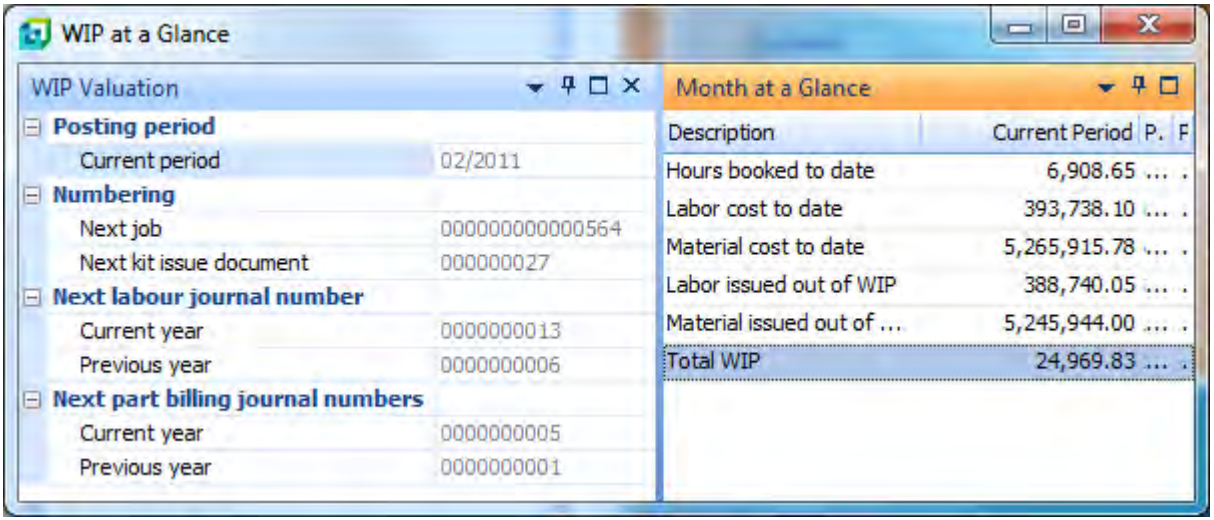


Figure 18-21: The *WIP at a Glance* query that appears if the operator clicks on the alert

GlobalVariable

Tile functions are similar to functions against forms and listviews in that they exist in isolation. They have no knowledge of other functions in either the same script, or other scripts. If you need to pass information between functions for the same tile, the tile's *GlobalVariable* can be populated with a value, and this becomes available to other functions against the same tile. This is in addition to the four global variables within the *SystemVariables* section.

For example, if you generated a value within a tile's *OnLoad* function, and you needed this value later in the *OnRefresh* function, you could put it in the tile's *GlobalVariable* during the *OnLoad*, and then read it back from the *OnRefresh* function.

If the *OnRefresh* event is set to fire at regular intervals you may need to keep state information regarding the last time that the event fired. Providing that the operator has not exited SYSPRO in the meantime, this information can be held within the *GlobalVariable* and read back the next time that the tile's *OnRefresh* event fires.

Picture

The *Picture* variable enables you to either change or add a picture to the tile. If you had a picture configured against a tile by default, you can use this variable to specify a different picture to be used if the operator clicks on the tile. For example, you could create a menu specific to this operator that consists of tiles with images relating to the tasks they perform. In addition to configuring these tiles to run the relevant programs you can also replace the images with a checkmark when the program has been run. If the tile is configured to call the SYSPRO program the code would be as simple as the following:

```
Function Tile_OnLoad()
  Tile.CodeObject.Picture = "C:\TST700\MMEDIA\Book - ClipArt.png"
End Function

Function Tile_OnClicked()
  Tile.CodeObject.Picture = "C:\TST700\MMEDIA\Check.png"
End Function
```

Figure 18-22: The picture against the tile denoting what the operator must do

Figure 18-22 shows the tile before the operator clicks on it. When they click on it the SYSPRO program associated with the tile is run, and when the program exits the picture against the tile is changed to show that the program has been run (see **Figure 18-23**).

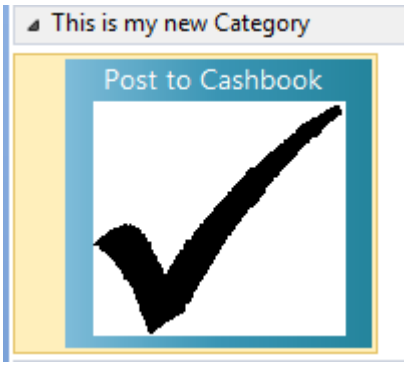


Figure 18-23: The picture against the tile reflects that the task has been performed

For information on making the picture available across multiple workstations if the tile or task panel is copied to another machine, see the section entitled *Ensuring Image Path Names are Resolved across Different Machines* below.

RefreshInterval
Each tile can be configured to automatically fire its *OnRefresh* event at a set interval, and this interval can be set differently for each tile. This interval is set using the *RefreshInterval* variable within the VBScript, and is set in minutes (and it can contain decimals). A refresh interval of 0.25 would be a quarter or a minute, or every 15 seconds. Typically you would not set a refresh interval this low as (depending on what tasks it has to perform) it may only finish processing just as it is time to fire again. If this variable is not set the tile will not automatically refresh.

As mentioned previously, a tile's *OnRefresh* event does not fire without being told to do so (unlike on a form where the *OnRefresh* event typically fires immediately after its matching *OnLoad* event). If you require the tile to automatically refresh the *RefreshInterval* variable should be set against the tile's *OnLoad* function. Note that if this variable is set against the tile's *OnRefresh* function it will only get triggered if something causes the tile's *OnRefresh* event to fire.

The following code shows the *RefreshInterval* being set to half a minute (30 seconds) against the tiles *OnLoad* function.

```
Function Tile_OnLoad()
  Tile.CodeObject.RefreshInterval = .5
End Function
```

If you want to force the tile to refresh when it first loads (for example, because it displays values on the tile that have to be retrieved/calculated) you would call the *OnRefresh* function within the tile's *OnLoad* function. This way it would be possible to add the *RefreshInterval* to the *OnRefresh* event, although there is no benefit to this over adding it to the *OnLoad* function.

```
Function Tile_OnLoad()
  Call (Tile_OnRefresh)
  Tile.CodeObject.RefreshInterval = .5
End Function
```

As mentioned in the *Adding Macro Events* section above, as you start editing a macro the automatic refreshing is suspended. You can check to see if the refresh events are currently suspended by right-clicking on a tile or the *Favorites* pane. If the *Suspend Refresh Events* option icon is highlighted (see **Figure 18-15**) the automatic refreshing is currently suspended. Click on this option to uncheck it, and start the events refreshing again. Refresh events are automatically enabled when you load SYSPRO. So even if they were suspended when you exited SYSPRO, they will automatically start working the next time you use SYSPRO.

TileNumber
The *TileNumber* variable is a means of displaying an identifying number on the tile, and is set per tile. The number can be an integer between 0 and 9999999999, and is injected into the XAML theme associated with this tile. The exact location of the number is specific to this XAML theme.

The *TileNumber* variable is a write-only variable, meaning that you can write to it to supply the number, but you cannot read this number back. However, within the same function that you used to set it, after the point that you have set this variable, you will still be able to access it. The following is an example of setting a tile number against the tile's *OnLoad* function.

```
Function Tile_OnLoad()
  Tile.CodeObject.TileNumber = 32
End Function
```

Note that if the tile has an image associated with it, and this image is changed using VBScript, the tile number will disappear unless the code that changes the image also sets this variable again.

TileValues

The *TileValues* variable enables you to supply values delimited by pipe signs, and these will be displayed on the tile. These values are injected into the XAML, providing a theme is associated with the tile. Most of the XAML themes will display the contents of the *TileValues* variable, the exceptions being *Category* and *Image Only*. The *Trendline* XAML theme will only display the *TileValues* if they are numbers, as this theme is using them to build a graph.

The following code passes four values to the *TileValues* variable, separated by pipe signs. These are treated as completely separate variables when they are passed to XAML. An example of how the tile will appear can be seen in **Figure 18-24**, where this tile is associated with the *Diagram* XAML theme.

```
Function Tile_OnLoad()
  Tile.CodeObject.TileValues = "LineValue entry 1|and 2|Three|and four"
End Function
```

Figure 18-24: Using the *TileValues* variable to display information on the tile

Title

The *Title* variable is used to assign a tile a title, or replace an existing one. A title is injected into the XAML of a tile. In **Figure 18-24** the title of *ssss* can be seen at the top of the tile. The code to replace this with *This is the title* when the tile loads, appears below.

```
Function Tile_OnLoad()
  Tile.CodeObject.Title = "This is the title"
End Function
```

Tooltip

A tooltip can be associated with a tile by right-clicking on the tile and selecting *Properties* | *Tooltip* from the context-sensitive menu. This can be either a text-based tooltip, or an image. A tooltip can also be added programmatically with VBScript, using the *Tooltip* variable.

If a tooltip (text or image) has been associated with a tile, and there is a script to set the tooltip against one of the tile's functions, when this function runs the existing tooltip will be replaced. For example, a tile might have a tooltip against it that lists what will happen when you click on it. As part of the tile's *OnClicked* function you could use this variable to change the tooltip to state that this tile has already been clicked.

Apart from just changing the tooltip to a fixed string of text, you can also dynamically build the content to include the results from a business object, or some other lookup. **Figure 18-25** shows an example of this where the current balance is displayed for all bank accounts.

Figure 18-25: A tooltip containing the current balances for each bank

TileXAML

The final variable under the *Tile* section of the *Variables* pane is *TileXAML*. This enables you to not only supply values, but also supply the XAML (the same logic was used to populate the tooltip in **Figure 18-25**).

Figure 18-26 shows a tile that refreshes every 30 minutes and displays the valuation of the three warehouses that this operator considers to be critical. This is performed using the *TileXAML* variable, the ***COMQEX*** business object, and a few lines of VBScript.

Figure 18-26: Using the TileXAML variable to keep track of your critical warehouses

Against the tile's *OnLoad* function are two lines. The first calls the tile's *OnRefresh* function so that it is populated immediately, and the second sets the tile to refresh every 30 minutes. If the *OnLoad* function did not call the *OnRefresh* function, the tile would remain blank for the first 30 minutes after loading.

```
Function Tile_OnLoad()
  call (Tile_OnRefresh)
  Tile.CodeObject.RefreshInterval = 30
End Function
```

The first three lines of the *OnRefresh* function create variables that are used later in the function.

```
Function Tile_OnRefresh()
  Dim XMLOut, XMLParam, WarehouseCode, FGDesc, FGValue
  Dim RMDesc, RMValue, RDesc, RValue, xList, Counter, GridValues
  Dim XMLDoc, FGFormatValue, RMFormatValue, RFormatValue
```

The next section contains the code to call the ***COMQEX*** business object and return the warehouse valuations. This is one of the examples where supplying a little more XML will greatly reduce the amount of XML that is returned. The default against the *IncludeBankBalances*, *IncludeApBalances*, and *IncludeArBalances* elements is *Y*, which can return extensive XML. By adding these three elements and setting them to *N*, this is removed from the returned XML.

```
XMLParam = XMLParam & "<Query>"
XMLParam = XMLParam & " <Option>"
XMLParam = XMLParam & "  <IncludeBankBalances>N</IncludeBankBalances>"
XMLParam = XMLParam & "  <IncludeApBalances>N</IncludeApBalances>"
XMLParam = XMLParam & "  <IncludeArBalances>N</IncludeArBalances>"
XMLParam = XMLParam & "  <IncludeInventoryValuation>Y</IncludeInventoryValuation>"
XMLParam = XMLParam & "  </Option>"
XMLParam = XMLParam & "</Query>"
on error resume next
XMLOut = CallBO("COMQEX",XMLParam,"auto")
if err then
    exit function
end if
' Switch on error handling
on error goto 0
```

The above code was built up using the *Call Query* business object wizard that is accessed by clicking on the *Query* option from the *Call Business Object* button on the toolbar (see **Figure 18-27**).

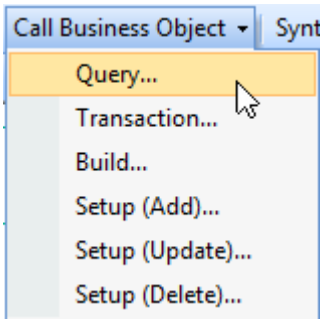


Figure 18-27: Starting the *Call a Query Business Object* wizard

At the *Query Business Object* prompt on the toolbar, add the business object name of ***COMQEX*** and tab off the field. The sample XML for ***COMQEX*** will be populated in the *Parameters* pane. Replace the XML with the following, and click on the *Insert VBScript* button.

```
<Query>
  <Option>
   <IncludeBankBalances>N</IncludeBankBalances>
    <IncludeApBalances>N</IncludeApBalances>
    <IncludeArBalances>N</IncludeArBalances>
    <IncludeInventoryValuation>Y</IncludeInventoryValuation>
  </Option>
</Query>
```

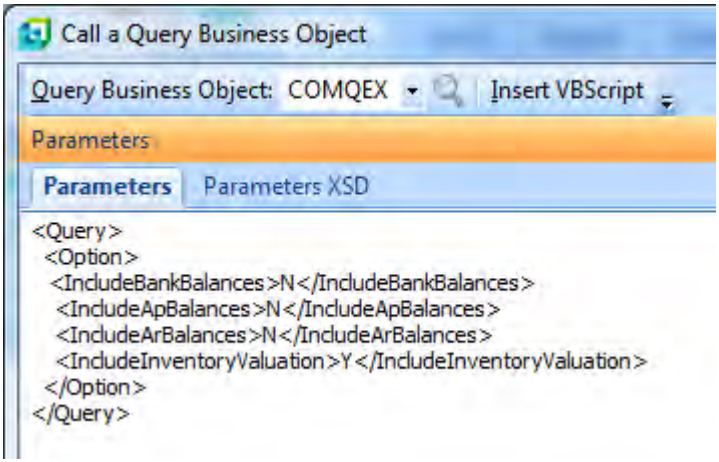


Figure 18-28: The *Call a Query Business Object* wizard with the updated XML

Figure 18-28 shows the screen just before the button is pressed. The next section of code takes the output from the business object (that is in the *XMLOut* variable) and loads it into the *Document Object Model* so that it can be manipulated more easily.

```
' Load the output from the business object into the DOM.
Set XMLDoc = createobject("MSXML2.DOMDocument")
XMLDoc.async = false
XMLDoc.LoadXML(XMLOut)
```

The next section of code looks through the XML and finds the nodes called *WarehouseItem*, of which there is one for each warehouse.

```
' Look for all LocalCurrency element
Set xList = XMLDoc.SelectNodes("//WarehouseItem")
```

The next line starts a *For*/*Next* loop that will cycle through the XML, once for each warehouse entry. As the counter starts at 0, one must be subtracted from the length of the list. For example, if there were five warehouses, because the counter starts at 0 it must continue to entry 4 (5 minus 1).

```
For Counter = 0 To xList.length - 1
```

The next section is within this *For*/*Next* loop so happens once for each *WarehouseItem* node. It populates the *WarehouseCode* variable with the warehouse code for this node.

```
' Loop through each of the LocalCurrency elements
WarehouseCode = xList(Counter).SelectSingleNode("Warehouse").Text
```

After that there is a check to see if the content of the *WarehouseCode* variable matches one of the three warehouses that you are interested in (*FG*, *RM*, and *R*). If it matches *FG* it extracts the warehouse description and puts it in a variable called *FGDesc*, and extracts the warehouse's current valuation and puts it in the variable *FGValue*. If the warehouse matches *RM* it puts the description and value in the *RMDesc* and *RMValue* variables, and if it matches *R* it puts the description and value in the *RDesc* and *RValue* variables.

```
If WarehouseCode = "FG" Then
  FGDesc = xList(Counter).SelectSingleNode("Description").Text
  FGValue = xList(Counter).SelectSingleNode("WarehouseValueCur").Text
ElseIf WarehouseCode = "RM" Then
  RMDesc = xList(Counter).SelectSingleNode("Description").Text
  RMValue = xList(Counter).SelectSingleNode("WarehouseValueCur").Text
ElseIf WarehouseCode = "R" Then
  RDesc = xList(Counter).SelectSingleNode("Description").Text
  RValue = xList(Counter).SelectSingleNode("WarehouseValueCur").Text
End If
```

Immediately after this is a line containing the *NEXT* statement, which tells the script that if there are any more *WarehouseItem* nodes to process it must go back to the *FOR* statement and process the next one. If there are no more to process it must drop through.

```
Next
```

The following three lines of code format the three value variables (*FGValue*, *RMValue*, and *RValue*) so that they are easier to read using the *FormatNumber* function of VBScript. The formatted values are stored in the *FGFormatValue*, *RMFormatValue*, and *RFormatValue* variables respectively.

```
FGFormatValue = FormatNumber(FGValue,2,-2,-2,-2)
RMFormatValue = FormatNumber(RMValue,2,-2,-2,-2)
RFormatValue = FormatNumber(RValue,2,-2,-2,-2)
```

After formatting the values, the next section is where a variable called *GridValues* is populated with the XAML, and the variables containing the values. It is not intended to explain how to create your own XAML, but rather show it so that you can tweak it to suit your requirements.

It creates the equivalent of a table of three columns (*ColumnDefinition*) and four rows (*RowDefinition*). The first two columns are defined as *Auto* sizing so that they become just big enough for the values within them, and the third column is defined as * which means it must consume whatever is left.

The grid is then populated with the *Warehouse* and *Value* titles that are hardcoded. This is followed by the descriptions, then the values.

Within the code below, each line (except the last one) ends with an ampersand, a space and an underscore. This tells VBScript that the line continues on the next line. This has been done here to prevent the lines from wrapping on the page. So you have really created one long line, and if you were to write out the *GridValues* variable to a file, you would end up with one long line.

The lines have also been indented to help follow the logic.

```
GridValues = "<StackPanel Grid.Row = '1' Grid.Column='2' " & _
  " Background='#24849C' Margin='5, 5, 5, 5'>" & _
  "<StackPanel  Margin='5' >" & _
    "<Grid >" & _
      "<Grid.ColumnDefinitions>" & _
        "<ColumnDefinition Width='Auto'/>" & _
        "<ColumnDefinition Width='Auto'/>" & _
        "<ColumnDefinition Width='*'/>" & _
      "</Grid.ColumnDefinitions>" & _
      "<Grid.RowDefinitions>" & _
        "<RowDefinition/>" & _
        "<RowDefinition/>" & _
        "<RowDefinition/>" & _
        "<RowDefinition/>" & _
      "</Grid.RowDefinitions>" & _
      "<TextBlock Foreground='White' Grid.Column='0' TextAlignment='Left' " & _
        " Text='Warehouse' Margin='0, 0, 0, 4'/>" & _
      "<TextBlock Foreground='White' Grid.Column='1'  TextAlignment='Right' " & _
       " Text='Value' Margin='0, 0, 0, 4'/>" & _
      "<TextBlock Grid.Row='1' Margin='0, 0, 10, 0'>" & _
        "<Run Text='" & FGDesc & "' Foreground='White'/>" & _
      "</TextBlock>" & _
      "<TextBlock Grid.Row='2' Margin='0, 0, 10, 0'>" & _
        "<Run Text='" & RMDesc & "' Foreground='White'/>" & _
      "</TextBlock>" & _
      "<TextBlock Grid.Row='3' Margin='0, 0, 10, 0'>" & _
        "<Run Text='" & RDesc & "' Foreground='White'/>" & _
      "</TextBlock>" & _
      "<TextBlock Grid.Column='1' Grid.Row='1' TextAlignment='Right' " & _
         "Foreground='White'  Text='$ " & FGFormatValue & "'/>" & _
      "<TextBlock Grid.Column='1' Grid.Row='2' TextAlignment='Right' " & _
         "Foreground='White'  Text='$ " & RMFormatValue & "'/>" & _
      "<TextBlock Grid.Column='1' Grid.Row='3' TextAlignment='Right' " & _
         "Foreground='White'  Text='$ " & RFormatValue & "'/>" & _
    "</Grid>" & _
  "</StackPanel>" & _
"</StackPanel>"
```

The code is finished off by populating the *TileXAML* variable with the contents of the *GridValues* variable, and the ending the function.

```
  Tile.CodeObject.TileXAML = GridValues
End Function
```

Once complete, after saving the code, exiting SYSPRO, and calling SYSPRO again, you will have a tile similar to that in **Figure 18-26**.

Ensuring Image Path Names are Resolved across Different Machines
When a picture is attached to a tile, a link to that image is inserted into the tile's properties. This image can reside in almost any location, and it will be displayed. If a tile is to be copied for any reason (such as being exported and imported to a new operator, or the task panel XML is copied) the pictures may not be displayed unless they were selected from a shared network folder, because the supplied address may not be valid from the new location.

If the image is copied to the SYSPRO client's `Base\Samples` folder it can be referenced using *{samples}* followed by the image name, rather than specifying the full path name. This means that SYSPRO will always look in the client's `Base\Samples` folder, no matter what folder structure was used for this client. This does mean that the image will need to be available in the other workstations `Base\Samples` folder if a tile is copied to another workstation. The following is an example where the image *Bike1.png* is being accessed from the `Base\Samples` folder.

```
Tile.CodeObject.Picture = "{samples}\Bike1.png"
```

Example Using the TrendLine XAML Style on a Tile
This example uses the sales quantity history output from the ***INVQRY*** business object to populate a tile that is using the *TrendLine* XAML style (see **Figure 18-29**).

A list of four stock codes are provided and each time that the tile refreshes it cycles through these stock codes, updating the title with the stock description, and the *TrendLine* with the sales values for a specified warehouse.

Figure 18-29: A *TrendLine* style tile populated with the values from the *A100* stock code

The first step to producing this is to create a blank tile, which is done by right-clicking on the *Favorites* pane and selecting *New* | *Blank Tile*, and supplying a name for the tile. The tile should be created for you. **Figure 18-30** shows the adding of a blank tile called *Sales Qty*.

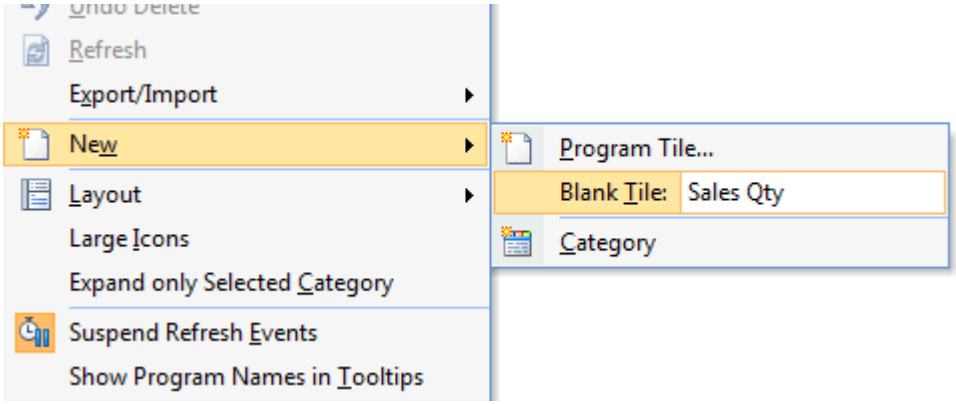


Figure 18-30: Adding a *Blank Tile*

Right-click on the tile and select *Properties* | *Macro* from the context-sensitive menu. If this is the first time that you have edited the script against a tile since you called up SYSPRO, a message will be displayed to inform you that the automatic refreshing of all tiles will be suspended.

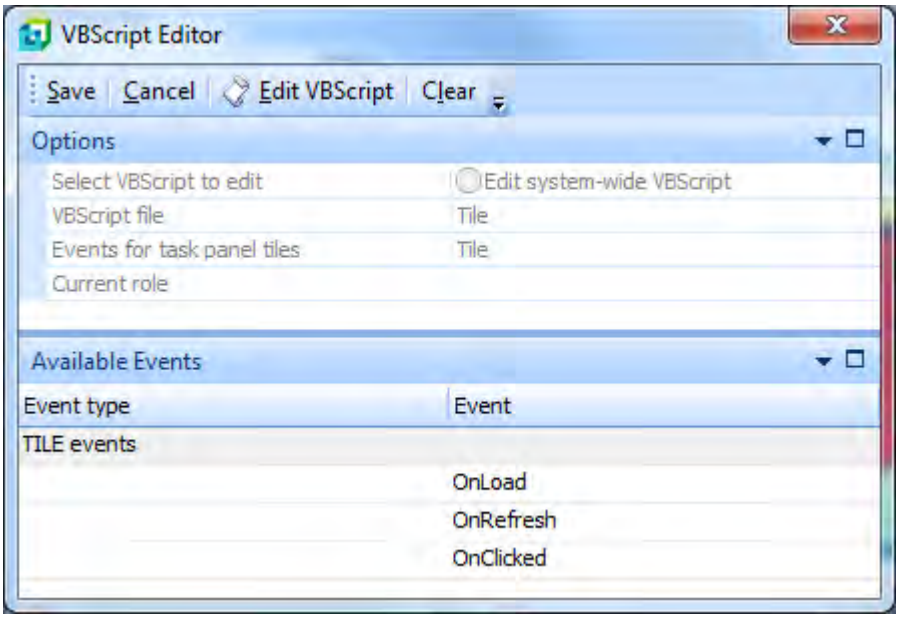


Figure 18-31: The *VBScript Editor* screen

The VBScript Editor screen is displayed (see **Figure 18-31**). Highlight the *OnLoad* event and click on the *Edit VBScript* button on the toolbar. The *VBScript for: Tile* screen is loaded, the *Tile_OnLoad* function is created for you, and the cursor placed within it.

You need to add the following code within this function. Note that the lines beginning with an apostrophe are comment lines to make the code easier to follow, and do not need to be present.

The first of the lines of code sets the refresh interval to one minute so that the *OnRefresh* event fires every minute. The second line adds the stock code *A100* to the tile's *GlobalVariable*.

```
' Set the refresh interval to every minute
Tile.CodeObject.RefreshInterval = 1

' Set the first stock code to A100
Tile.CodeObject.GlobalVariable = "A100"
```

Exit the *VBScript for: Tile* screen back to the *VBScript Editor* screen. You will see that the *OnLoad* event name has a script icon against it to show that there is already code against it. Highlight the *OnRefresh* event and click on the *Edit VBScript* button. The *VBScript for: Tile* screen will be loaded, the *Tile_OnRefresh* function will be created, and the cursor will be placed within it.

Within the *OnRefresh* function, add the following lines of code that create variables that will be used in the script.

```
' Create the variables to be used in the script
Dim StockDesc, MyGV, XMLDoc, numHighValue, numHighValuePerc
Dim xList, xList2, Counter, Counter2, Counter3, Counter4
Dim SalesArray(12), VectorPoint(12), VectorPointA(12)
Dim VectorPointB(12), VectorPointC(12), GraphXAML
```

Immediately after these *DIM* statements, add the following lines of code. The first of these two lines puts the current content of the tile's *GlobalVariable* into the *MyGV* variable. The second line sets the *numHighValue* variable to zero. This variable will be used to find the highest value amongst the twelve months sales figures.

```
' Populate the MyGV variable with the contents of the GlobalVariable
MyGV = Tile.CodeObject.GlobalVariable

' Set the highest sales value to 0
numHighValue = 0
```

Once this code has been added you will use the *Call Query Business Object* wizard to add the code to call the ***INVQRY*** business object. Start by clicking on the *Call Business Object* button on the toolbar (the default option is to call a *Query* class business object). The *Call a Query Business Object* screen is

displayed. At the *Query Business Object* prompt add the business object name ***INVQRY*** and tab off the field. The *Parameters* tab will be populated with the sample XML for this business object.
Replace the sample XML with the following and click on the *Insert VBScript* button on the toolbar to insert the code to call this business object with the XML. Note that the six Xs will be replaced with a variable name once the code has been inserted.

```
<Query>
  <Key>
    <StockCode><![CDATA[XXXXXX]]></StockCode>
  </Key>
  <Option>
    <IncludeHistory>Y</IncludeHistory>
  </Option>
  <Filter>
    <Warehouse FilterType="L" FilterValue="E"/>
  </Filter>
</Query>
```

The following code will have been added for you.

```
Dim XMLOut, XMLParam

XMLParam = XMLParam & " <Query>"
XMLParam = XMLParam & "   <Key>"
XMLParam = XMLParam & "     <StockCode><![CDATA[XXXXXX]]></StockCode>"
XMLParam = XMLParam & "   </Key>"
XMLParam = XMLParam & "   <Option>"
XMLParam = XMLParam & "     <IncludeHistory>Y</IncludeHistory>"
XMLParam = XMLParam & "   </Option>"
XMLParam = XMLParam & "   <Filter>"
XMLParam = XMLParam & "     <Warehouse FilterType='L' FilterValue='E'/>"
XMLParam = XMLParam & "   </Filter>"
XMLParam = XMLParam & " </Query>"
on error resume next
XMLOut = CallBO("INVQRY",XMLParam,"auto")
if err then
    exit function
end if
' Switch on error handling
on error goto 0
```

Locate the following section within the code that was added.

```
<StockCode><![CDATA[XXXXXX]]></StockCode>
```

Change this line to match the following. This replaces the text *XXXXXX* with the contents of the *MyGV* variable at run time.

```
<StockCode><![CDATA[" & MyGV & "]]></StockCode>
```

At the end of the code that was added by the wizard (immediately before the *End Function* statement), add the following lines of code. These lines instantiate the *Document Object Model* (*DOM*) and load the XML output from the business object into it. The *DOM* makes it easier to manipulate the XML, or extract values from it.

```
' Load the output from the business object into the DOM.
Set XMLDoc = createobject("MSXML2.DOMDocument")
XMLDoc.async = false
XMLDoc.LoadXML(XMLOut)
```

The next line of code to add uses the *DOM* to extract the contents of the *Description* element and place it in the *StockDesc* variable that you created earlier.

```
' Populate the StockDesc variable with the stock description from the XML
StockDesc = XMLDoc.SelectSingleNode("//InvQuery/StockItem/Description").Text
```

This is followed by a line of code that locates all the *SalesQtyByMonth* nodes. There should only be one of these because of the *Warehouse* filter in the XML supplied to the business object. Within the *DOM* the number of the nodes starts at 0, so the first one is numbered 0, the second numbered 1, etc.

```
' Look for all SalesQtyByMonth elements, there should only be one
Set xList = XMLDoc.SelectNodes("//SalesQtyByMonth")
```

This is followed by a line setting up a *For*/*Next* loop to cycle through all the *SalesQtyByMonth* nodes so that they can be processed. The count has to start at 0 and carry on until it reaches the number of nodes in the list (less 1, because it starts at 0). The count could have started at 1, but as the elements are being accessed using this count it was easier to start the count at 0 and subtract 1 from the highest node to know when to finish.

```
' Loop through each SalesQtyByMonth section
For Counter = 0 To xList.length - 1
```

The next twelve lines of code extract the contents of each of the twelve *SalesQty* elements and add them into an array. When the variables were defined some had *(12)* at the end of their names. This tells VBScript that these variables are arrays, which can hold multiple pieces of information. In this case they can hold 12 pieces. Locations within an array start at zero, so array entry 0 contains sales quantity 1, etc.

```
' Populate array with values from the XML
SalesArray(0) = CDbl(xList(Counter).SelectSingleNode("SalesQty1").Text)
SalesArray(1) = CDbl(xList(Counter).SelectSingleNode("SalesQty2").Text)
SalesArray(2) = CDbl(xList(Counter).SelectSingleNode("SalesQty3").Text)
SalesArray(3) = CDbl(xList(Counter).SelectSingleNode("SalesQty4").Text)
SalesArray(4) = CDbl(xList(Counter).SelectSingleNode("SalesQty5").Text)
SalesArray(5) = CDbl(xList(Counter).SelectSingleNode("SalesQty6").Text)
SalesArray(6) = CDbl(xList(Counter).SelectSingleNode("SalesQty7").Text)
SalesArray(7) = CDbl(xList(Counter).SelectSingleNode("SalesQty8").Text)
SalesArray(8) = CDbl(xList(Counter).SelectSingleNode("SalesQty9").Text)
SalesArray(9) = CDbl(xList(Counter).SelectSingleNode("SalesQty10").Text)
SalesArray(10) = CDbl(xList(Counter).SelectSingleNode("SalesQty11").Text)
SalesArray(11) = CDbl(xList(Counter).SelectSingleNode("SalesQty12").Text)
```

Once the array contains all of the sales values there needs to be another *For*/*Next* loop to find the highest value within the array. The code to do this appears below. On completion, the highest of these values is stored in the *numHighValue* variable.

```
' Loop through array looking for the highest value, then write the
' highest value to the variable numHighValue
For Counter3 = LBound(SalesArray) To UBound(SalesArray)
  If SalesArray(Counter3) > numHighValue Then
    numHighValue = SalesArray(Counter3)
  End If
Next
```

Once the highest value is known, a test is performed to see if it is a positive value using an *IF* statement.

```
' Test to make sure that the highest values is greater than zero
If numHighValue > 0 then
```

If the highest value is positive, the value is divided by 40, as this is the maximum value that can appear in the *trendline*. The *numHighValuePerc* variable is populated with the result.

```
' Calculate the number to divide all the sales values by, so that
' they all fit within the scale (the scale of the graph is 0 to 40)
numHighValuePerc = numHighValue / 40
```

As mentioned above, the highest value within the *TrendLine* XAML is 40. The *numHighValuePerc* variable contains the highest sales value divided by 40. All the sales values must now be divided by this figure so that they all appear in proportion.

These calculated sales values are stored in an array called *VectorPointA*. As the XAML displays the values inversely (the highest point on the *trendline* is 1 and the lowest is 40) the inverse of the values stored in the *VectorPointA* array is required. These inverse values are stored in an array called *VectorPoint*.

All of the above is done in the following *For*/*Next* loop.

```
' Loop through the array and calculate the vector position for each period,
' which is done by calculating the value as a percentage of the highest
' value. The full scale of the trendline is 40. However, the lowest point
' is 40 and the highest point is 1 so the values must be inverted (a value of
' 1 must be 40, and a value of 40 must be 1).
For Counter4 = LBound(SalesArray) To UBound(SalesArray)
  VectorPointA(Counter4) = SalesArray(Counter4) / numHighValuePerc
  VectorPoint(Counter4) = 40 - CInt(VectorPointA(Counter4))
Next
```

Now that all the values required for the *trendLine* are known, each entry is added to a variable called *GraphXAML*, and delimited with pipe signs. Remember that the array containing the values is called *VectorPoint*, and that the array starts with 0.

```
' Populate the variable GraphXML with these twelve values, delimited
' with pipe signs
GraphXAML = VectorPoint(0) & "|" & VectorPoint(1) & "|" & VectorPoint(2) & "|"
GraphXAML = GraphXAML & VectorPoint(3) & "|" & VectorPoint(4) & "|"
GraphXAML = GraphXAML & VectorPoint(5) & "|" & VectorPoint(6) & "|"
GraphXAML = GraphXAML & VectorPoint(7) & "|" & VectorPoint(8) & "|"
GraphXAML = GraphXAML & VectorPoint(9) & "|" & VectorPoint(10) & "|"
GraphXAML = GraphXAML & VectorPoint(11)
```

As all of this is happening within the *IF* statement, the *ELSE* statement states what should happen if the highest sales value is not a positive value. In this case it sets the *GraphXAML* variable to contain twelve values of 40 delimited by pipe signs. This means that a straight *trendline* will appear at the bottom of the tile. The *IF* statement is finished with an *END IF* statement.

```
Else
  ' If there are no values, display the lowest possible line
  GraphXAML="40|40|40|40|40|40|40|40|40|40|40|40"
End If
```

The stock description held in the *StockDesc* variable is used to populate the tile's *Title* variable, and the contents of the *GraphXAML* variable is passed to the *TileValues* variable. Both these variables appear under the *Tile* section of the *Variables* tab (see **Figure 18-32**).

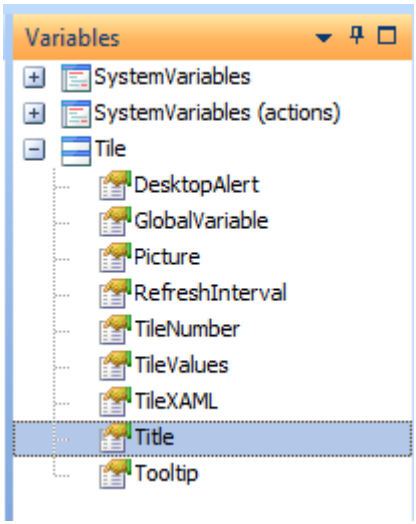


Figure 18-32: The *Title* and *TileValues* variables

```
' Pass the stock description through to the tile
Tile.CodeObject.Title = StockDesc
' Pass the values for the graph through to the tile
Tile.CodeObject.TileValues = GraphXAML
```

This is followed by a line containing a *NEXT* statement. This matches the *FOR* statement that created the loop through the *SalesQtyByMonth* nodes in the *DOM*.

```
Next
```

The last piece of code is used to keep track of the stock codes. It takes the content of the *MyGV* variable (that contained the same value as the tile's *GlobalVariable*) and evaluates it. If it contains *A100* the *GlobalVariable* is updated with the *A101* stock code. If it contains *A101* it will be updated with the *A102* stock code. If it contains *A102* it will be updated with the *A103* stock code. If it contains any other value it will be updated with the *A100* stock code.

```
' Update the GlobalVariable with the next stock code
Select Case MyGV

  Case "A100"
    Tile.CodeObject.GlobalVariable = "A101"

  Case "A101"
    Tile.CodeObject.GlobalVariable = "A102"

  Case "A102"
    Tile.CodeObject.GlobalVariable = "A103"

  Case Else
    Tile.CodeObject.GlobalVariable = "A100"

End Select
```

You have completed the coding for the *OnRefresh* function, but there is still a change that is required for the *OnLoad* function. It currently contains the following:

```
Function Tile_OnLoad()
  ' Set the refresh interval to every minute
  Tile.CodeObject.RefreshInterval = 1

  ' Set the first stock code to A100
  Tile.CodeObject.GlobalVariable = "A100"

End Function
```

This must have one line of code added to the end that calls the *OnRefresh* event when it runs. If this line is not present the tile will only be populated when the refresh runs after one minute. The update code appears below.

```
Function Tile_OnLoad()
  ' Set the refresh interval to every minute
  Tile.CodeObject.RefreshInterval = 1

  ' Set the first stock code to A100
  Tile.CodeObject.GlobalVariable = "A100"

  ' Refresh the values against the tile
  Call (Tile_OnRefresh)
End Function
```

Before exiting the *VBScript for:Tile* screen it is advisable to click on the *Syntax Check* button on the toolbar. This performs a check to see if the basic syntax has been followed, and will highlight these issues. However, it won't find logic problems. If any items are highlighted these should be addressed before exiting the editor.

If no syntax issues were found, or you have corrected them, exit the *VBScript for: Tile* screen back to the *VBScript Editor* screen. Click on the *Save* button and you will be taken back to the *Favorites* screen.

Log out of SYSPRO and back in. The tile should display the *trendline* for the first stock code, and a minute later it should display the *trendline* for the second stock code.

Below is the complete *Tile_OnLoad* function:

```
Function Tile_OnLoad()
  ' Set the refresh interval to every minute
  Tile.CodeObject.RefreshInterval = 1

  ' Set the first stock code to A100
  Tile.CodeObject.GlobalVariable = "A100"

  ' Refresh the values against the tile
  Call (Tile_OnRefresh)
End Function
```

Below is the complete *Tile_OnRefresh* function:

```
Function Tile_OnRefresh()
  ' Create the variables to be used in the script
  Dim StockDesc, MyGV, XMLDoc, numHighValue, numHighValuePerc
  Dim xList, xList2, Counter, Counter2, Counter3, Counter4
  Dim SalesArray(12), VectorPoint(12), VectorPointA(12)
  Dim VectorPointB(12), VectorPointC(12), GraphXAML

  ' Populate the MyGV variable with the contents of the GlobalVariable
  MyGV = Tile.CodeObject.GlobalVariable

  ' Set the highest sales value to 0
  numHighValue = 0

  Dim XMLOut, XMLParam

  XMLParam = XMLParam & " <Query>"
  XMLParam = XMLParam & "   <Key>"
  XMLParam = XMLParam & "     <StockCode><![CDATA[" & MyGV & "]]></StockCode>"
  XMLParam = XMLParam & "   </Key>"
  XMLParam = XMLParam & "   <Option>"
  XMLParam = XMLParam & "     <IncludeHistory>Y</IncludeHistory>"
  XMLParam = XMLParam & "   </Option>"
  XMLParam = XMLParam & "   <Filter>"
  XMLParam = XMLParam & "     <Warehouse FilterType='L' FilterValue='E'/>"
  XMLParam = XMLParam & "   </Filter>"
  XMLParam = XMLParam & " </Query>"
  on error resume next
```

```
XMLOut = CallBO("INVQRY",XMLParam,"auto")
if err then
  exit function
end if
' Switch on error handling
on error goto 0

' Load the output from the business object into the DOM.
Set XMLDoc = createobject("MSXML2.DOMDocument")
XMLDoc.async = false
XMLDoc.LoadXML(XMLOut)

' Populate the StockDesc variable with the stock description from the XML
StockDesc = XMLDoc.SelectSingleNode("//InvQuery/StockItem/Description").Text

' Look for all WarehouseItem elements, there should only be one
Set xList = XMLDoc.SelectNodes("//SalesQtyByMonth")

' Loop through each SalesQtyByMonth section
For Counter = 0 To xList.length - 1

  ' Populate array with values from the XML
  SalesArray(0) = CDbl(xList(Counter).SelectSingleNode("SalesQty1").Text)
  SalesArray(1) = CDbl(xList(Counter).SelectSingleNode("SalesQty2").Text)
  SalesArray(2) = CDbl(xList(Counter).SelectSingleNode("SalesQty3").Text)
  SalesArray(3) = CDbl(xList(Counter).SelectSingleNode("SalesQty4").Text)
  SalesArray(4) = CDbl(xList(Counter).SelectSingleNode("SalesQty5").Text)
  SalesArray(5) = CDbl(xList(Counter).SelectSingleNode("SalesQty6").Text)
  SalesArray(6) = CDbl(xList(Counter).SelectSingleNode("SalesQty7").Text)
  SalesArray(7) = CDbl(xList(Counter).SelectSingleNode("SalesQty8").Text)
  SalesArray(8) = CDbl(xList(Counter).SelectSingleNode("SalesQty9").Text)
  SalesArray(9) = CDbl(xList(Counter).SelectSingleNode("SalesQty10").Text)
  SalesArray(10) = CDbl(xList(Counter).SelectSingleNode("SalesQty11").Text)
  SalesArray(11) = CDbl(xList(Counter).SelectSingleNode("SalesQty12").Text)

  ' Loop through array looking for the highest value, then write the
  ' highest value to the variable numHighValue
  For Counter3 = LBound(SalesArray) To UBound(SalesArray)
    If SalesArray(Counter3) > numHighValue Then
      numHighValue = SalesArray(Counter3)
    End If
  Next

  ' Test to make sure that the highest values is greater than zero
  If numHighValue > 0 then

    ' Calculate the number to divide all the sales values by, so that
    ' they all fit within the scale (the scale of the graph is 0 to 40)
    numHighValuePerc = numHighValue / 40

    ' Loop through the array and calculate the vector position for each period,
```

```
      ' which is done by calculating the value as a percentage of the highest
      ' value. The full scale of the trendline is 40. However, the lowest point
      ' is 40 and the highest point is 1 so the values must be inverted (a value
      ' of 1 must be 40, and a value of 40 must be 1).
      For Counter4 = LBound(SalesArray) To UBound(SalesArray)
        VectorPointA(Counter4) = SalesArray(Counter4) / numHighValuePerc
        VectorPoint(Counter4) = 40 - CInt(VectorPointA(Counter4))
      Next

      ' Populate the variable GraphXML with these twelve values, delimited
      ' with pipe signs
     GraphXAML = VectorPoint(0) & "|" & VectorPoint(1) & "|" & VectorPoint(2) & "|"
     GraphXAML = GraphXAML & VectorPoint(3) & "|" & VectorPoint(4) & "|"
     GraphXAML = GraphXAML & VectorPoint(5) & "|" & VectorPoint(6) & "|"
     GraphXAML = GraphXAML & VectorPoint(7) & "|" & VectorPoint(8) & "|"
     GraphXAML = GraphXAML & VectorPoint(9) & "|" & VectorPoint(10) & "|"
     GraphXAML = GraphXAML & VectorPoint(11)

    Else
      ' If there are no values, display the lowest possible line
      GraphXAML="40|40|40|40|40|40|40|40|40|40|40|40"

    End If

    ' Pass the stock description through to the tile
    Tile.CodeObject.Title = StockDesc
    ' Pass the values for the graph through to the tile
    Tile.CodeObject.TileValues = GraphXAML

  Next

  ' Update the GlobalVariable with the next stock code
  Select Case MyGV

    Case "A100"
      Tile.CodeObject.GlobalVariable = "A101"

    Case "A101"
      Tile.CodeObject.GlobalVariable = "A102"

    Case "A102"
      Tile.CodeObject.GlobalVariable = "A103"

    Case Else
      Tile.CodeObject.GlobalVariable = "A100"

  End Select

End Function
```

Chapter 19 - Flow Graphs

Flow Graphs enable you to build up a graphical menu, a series of tasks, a representation of a business process, or simply a reminder list. For new operators loading SYSPRO for the first time, the *Flow Graph* pane is hidden and is available by clicking on the *Flow Graph* icon on the *Home* tab of the *Ribbon Bar* (see **Figure 19-1**).

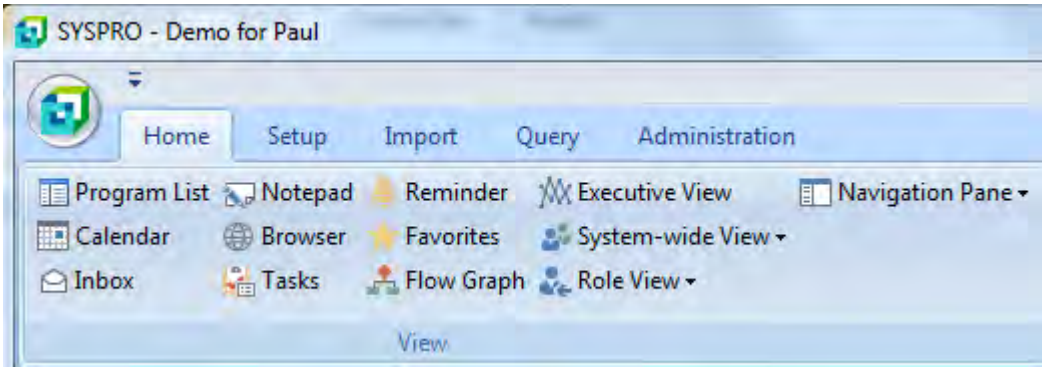


Figure 19-1: The *Flow Graph* icon on the SYSPRO Ribbon Bar

As the *Flow Graph* pane is a docking pane, it can be expanded, minimized, or docked in another location on the screen, just like most other panes. **Figure 19-2** shows the initial state of the flow graph pane before any information has been added to it.

When flow graphs are in use each operator has a default flow graph (*Home Flow Graph*), and these can be linked to other flow graphs. You can transition back and forward between flow graphs, but only one flow graph can be displayed at a time. Each flow graph must have at least one *Page*, and may contain multiple *Pages* within it. If this flow graph contains multiple pages, you can transition between pages, but only one can appear at a time.

Within a flow graph page you can add *Shapes*, and link these shapes using *Connectors*. A shape can call a SYSPRO program, Report Writer report, Net Express program, third party application, or SRS report. Shapes can also use VBScript to perform tasks such as calling a business object, changing the status of a shape, or writing an entry to a database.

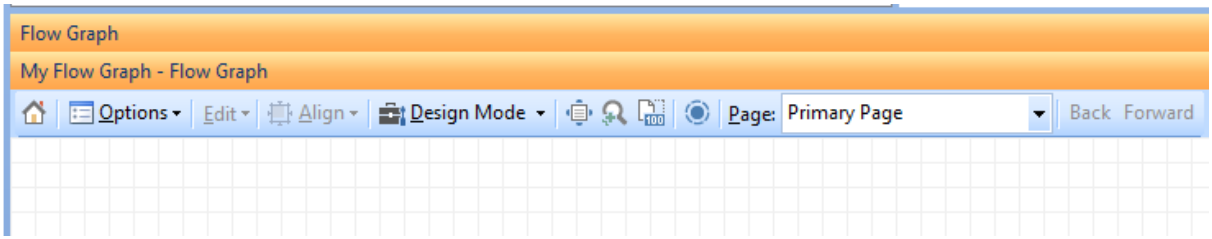


Figure 19-2: The initial state of a new *Flow Graph* pane

Home Flow Graph

When a flow graph is created for the first time it is automatically saved in the `Base\Settings` folder with the default flow graph name for this operator code. This is known as the *Home Flow Graph*.

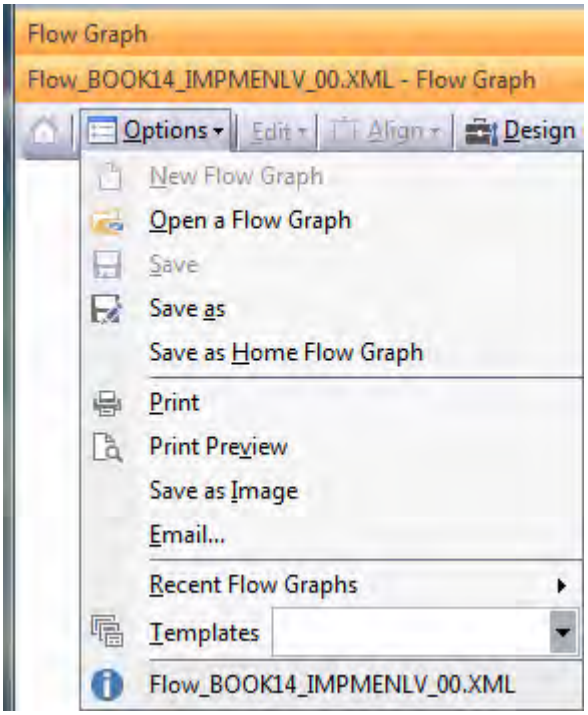


Figure 19-3: Flow Graph *Options* menu showing the name of the flow graph

The default name appears below, where *OperatorCode* is replaced by the operator's code.

Flow_*OperatorCode*_IMPMENLV_00.XML

When the operator accesses the flow graph pane for the first time for this run of SYSPRO, they are always taken to their default flow graph. The name of the current flow graph can be seen on the *Options* menu, which is accessible from the toolbar of the *Flow Graph* pane. Unless a *Title* has been added to this flow graph, the title in the titlebar will also contain the flow graph's name. **Figure 19-3** shows the flow graph's name appearing in both the titlebar and the *Options* menu. In this case the operator code is *BOOK14*, so the default flow graph name is *Flow_BOOK14_IMPMENLV_00.XML*.

If the operator clicks on a shape that has an *Action* to transition to another flow graph, the name of the new flow graph will appear at the bottom of the *Options* menu when the new flow graph is displayed. The titlebar will contain either the title of the new flow graph (or the flow graph name, if a title has not been added).

Initial Flow Graph

When the *Flow Graph* pane is loaded for the first time (before a flow graph has been added for this operator) the screen appears as if it is a blank sheet of graph paper. A toolbar is present at the top of the pane, and the *Page* dropdown list on the toolbar contains the entry *Primary Page* (as can be seen in **Figure 19-2**). No changes can be made to this flow graph unless you first enter *Design Mode* by clicking on the *Design Mode* button on the toolbar.

Security

By default, all operators are prevented from modifying and saving flow graphs. There are two options that prevent these tasks from being performed, and they both exist against the *Customization* section of the operator's *Activities* settings. The first activity is *Flow Graph – Allowed to design flow graphs* and this prevents operators from modifying any flow graph, and from saving the current flow graph as their *Home Flow Graph* (this option is disabled under the *Options* menu).

The second operator activity is *Main Menu – Allow to design flow graphs by role*, and this prevents the operator from designing flow graphs for a role (the *Design Flow Graphs* button on the *Administration* tab of the *Ribbon Bar* is disabled).

If the *Main Menu – Allow to design flow graphs by role* option is disabled, but the *Flow Graph – Allowed to design flow graphs* option is enabled, the operator will be able to change their own flow graph, but not change the ones for a role.

If the option *Flow Graph – Allowed to design flow graphs* option is disabled, the operator will not be able to change their own, or role flow graphs, regardless of how the *Main Menu – Allow to design flow graphs by role* option is configured, as the *Design Flow Graphs* button on the *Administration* tab of the *Ribbon Bar* will be disabled.

For more information on using flow graphs with roles, see both the *Flow Graphs and Roles* and *Design Flow Graphs for Roles* sections below.

Flow Graph Toolbar

The flow graph toolbar consists of twelve buttons, including the name of the current page.

Page Name and Prompt
The name of the currently selected page appears in the toolbar. If more than one page is present for this flow graph, you can navigate between pages using the dropdown list against the *Page* prompt. The default name for the initial page in each flow graph is *Primary page*. This name can be changed when you are in *Design Mode*, but this page will always remain the primary one, and cannot be deleted from the flow graph.

Home Button (Open Your Home Flow Graph)
The *Home* button on the toolbar becomes enabled if you have moved away from your home flow graph to another flow graph during this run of SYSPRO. This could be by selecting the *Open a Flow Graph* option from the *Options* menu, or by clicking on one of the *Shapes* that has been configured to transition to another flow graph. Clicking on the *Home Flow Graph* button will navigate you from whichever flow graph you are on back to your home flow graph.

Back and Forward Buttons
The *Back* button becomes enabled if you have clicked on a *Shape* that transitions you to another page in the same flow graph. If you click the *Back* button, the *Forward* button becomes enabled so that you can return to the previous page. These buttons do not become enabled if you navigate between pages using the dropdown list on the toolbar, only if you navigate between pages by clicking on a *Shape* that has been configured to perform a transition.

Edit Button
The *Edit* button is only enabled when you are in *Design Mode*. It has a dropdown list containing *Undo*, *Redo*, *Cut*, *Copy*, *Paste*, and *Select All*. The enabled options depend on which item in the flow graph is highlighted, and performs these tasks as you would expect, so will not be covered in detail here.

Fits the Flow Graph to the Window
The *Fits the Flow Graph to the Window* button will resize the contents of the flow graph so that it fits the current size of the flow graph pane. If the existing content is too large for the pane, the content is proportionally reduced in size until it just fits. If the existing content easily fits within the pane, all of the content is proportionally increased in size until it just fits within the pane.

Zooms to the Currently Selected Shapes
When selected, the *Zooms to the Currently Selected Shapes* button increases or decreases the size of the currently selected shape so that it just fits within the flow graph pane. If multiple shapes have been added to a *Group*, the size is changed so that the group just fits within the flow graph pane.

Set Zoom at 100%
The *Set Zoom at 100%* button returns the flow graph page to its default size.

Align
The *Align* button is only available in *Design Mode*. When it is clicked a dropdown list of the possible permutations is displayed (see **Figure 19-4**).

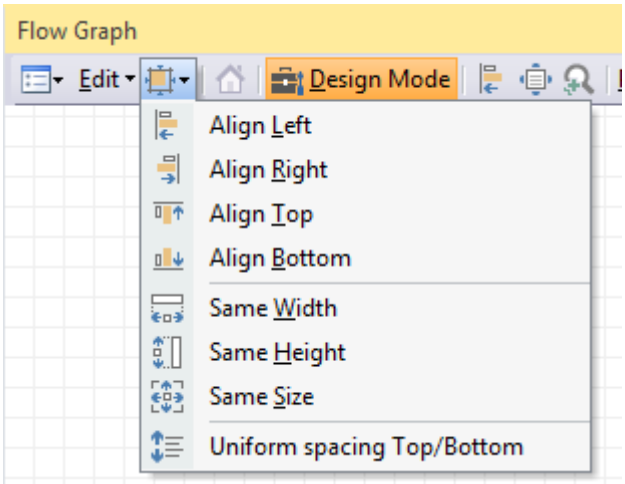


Figure 19-4: The options available when selecting the *Align the Edges/Sizes of Multiple Shapes* option

All of the options in this dropdown list need more than one item to be selected on the flow graph desktop before they perform a task. Items can be selected individually while pressing the *Ctrl* key and using the mouse pointer to click on them. Alternatively they can be grouped by lassoing them with the mouse pointer.

When one of the first four options in the dropdown list (those starting with *Align*) is selected the alignment happens in relation to the furthest item in that direction. For example, if the *Align Left* option is selected the alignment is in line with the item that is furthest to the left of those in the group. When one of the next three options is selected (those starting with the word *Same*) the change performed is in relation to the one last highlighted as they were grouped together.

The *Uniform spacing Top/Bottom* option manipulates the position of some of the grouped items so that the vertical gap is the same between all of the grouped items.

Enables Double-Click to Launch Applications
When not in *Design Mode*, clicking on a shape will launch whichever application, VBScript or action is associated with it. When in *Design Mode*, the same functionality is available, but only when you double-click on the shape (so that you can check the functionality without having to exit *Design Mode*).

The *Enables Double-Click to Launch Application* button configures the flow graph when not in *Design Mode* to require a double-click to invoke the action, instead of a single-click. Clicking on the *Enables Double-Click to Launch Application* button toggles the option on/off.

Options Button
When clicked, the *Options* button expands to show a menu containing many options (see **Figure 19-3**). As mentioned previously, the name of the current flow graph appears at the bottom of this menu.

The *New Flow Graph* option is only available when you are in *Design Mode*. It is used to create a new flow graph and causes the *Enter Name of New Flow Graph* screen to be displayed, where you supply the name for the new flow graph. If you supply the name of a file that does not already exist, the name of the current flow graph in the *Options* menu will change to the new name, but the file is not created at the point. If you had made changes to the flow graph that was previously open before starting the new one, those changes are automatically saved to the old file first.

If the name that you supplied against the *Enter Name of New Flow Graph* screen already exists, you are prompted to replace this file. If you answer *Yes* the contents of this file is not immediately replaced. However, when you select to create another new flow graph, open an existing flow graph, open a template, or exit SYSPRO, this file will automatically be updated with the contents of your new flow graph.

The *Open Flow Graph* option displays the *Open* screen and enables you to select which flow graph should be opened from a list. As this flow graph is opened, if any changes were made to the previous flow graph these changes are saved automatically before this one is opened.

The *Save* option is only available in *Design Mode*. If clicked, it will save the information in the current flow graph to a file with the current flow graph's name. The *Save As* option will prompt for the name of the file to be used, and will warn you if that filename already exists.

The *Email* option will create an email message, and attach the current flow graph's XML file. If changes have been made to the current flow graph, the file is automatically saved first. When a flow graph is emailed it will contain all the pages within this flow graph. You can add the recipient, subject, and any message to the email before sending it. The *Email* option requires the operator to have the *Fax/mail integration required* option checked on the *Options* tab of the *Operator Maintenance* program (see **Figure 19-5**).

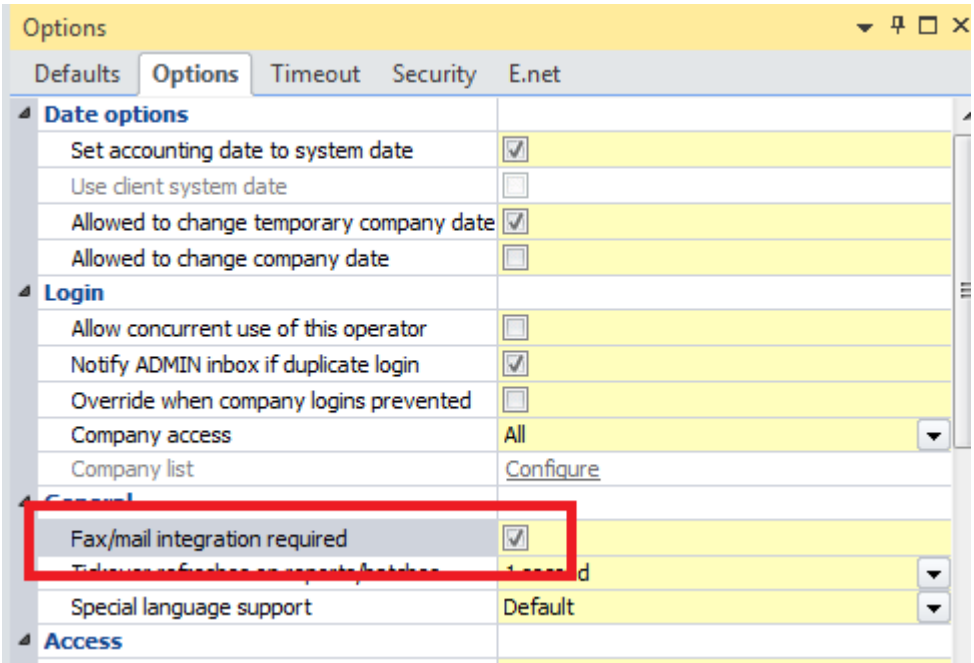


Figure 19-5: The *Fax/mail integration required* option within the *Operator Maintenance* program

If the operator does not have this checkbox checked they will receive an error message stating that the email will not be sent (see **Figure 19-6**).

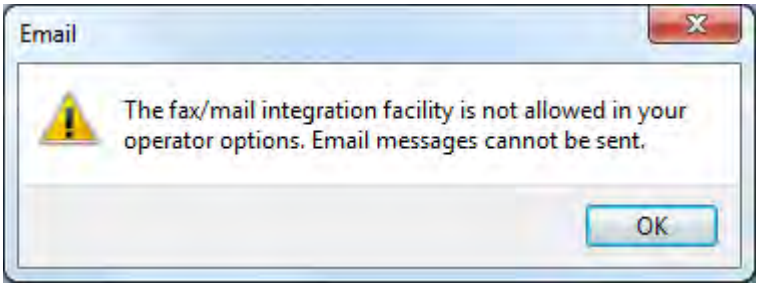


Figure 19-6: The error message if the operator is not allowed to email

The *Recent Flow Graphs* option keeps a list of the last 10 flow graphs that were accessed by this operator. Clicking on one of these will load it. If any changes were made to the currently open flow graph it will automatically be saved before the new one is loaded. The list of recent flow graphs can be seen in **Figure 19-7**.

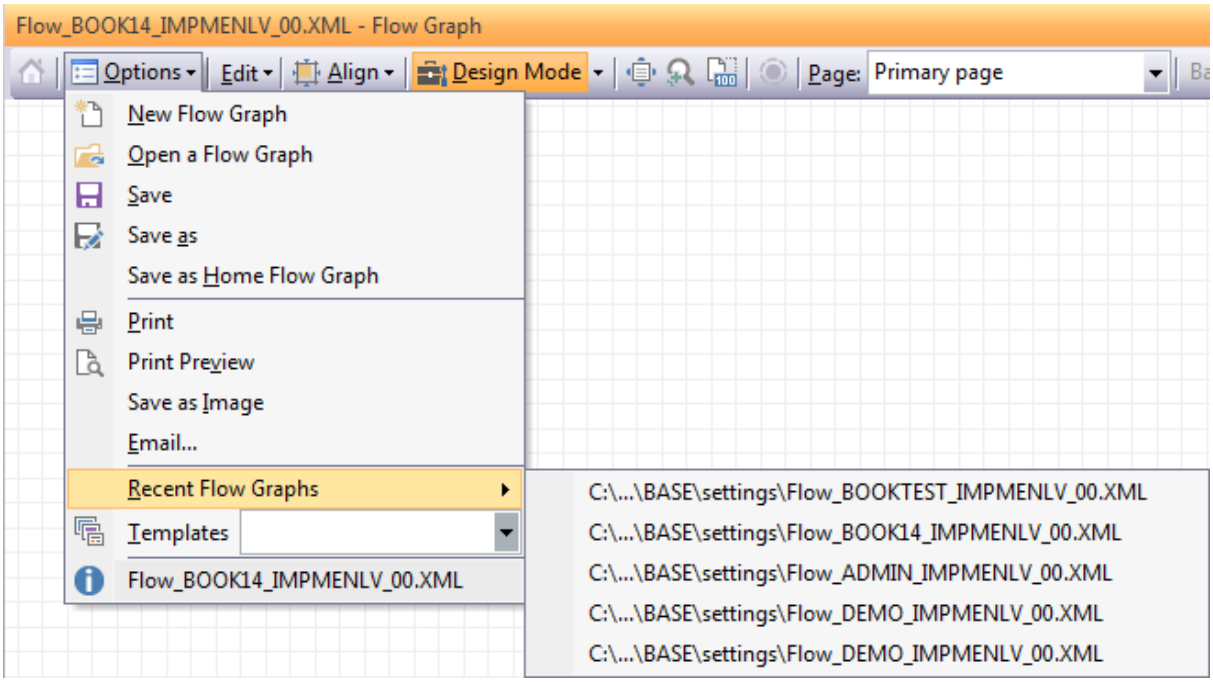


Figure 19-7: The list of most recently accessed flow graphs

The *Templates* option (see **Figure 19-8**) contains a dropdown list of standard flow graph templates provided with SYSPRO, which document some of the standard business processes. These reside in the SYSPRO `Base\Samples` folder with the name *FlowTemplate_TemplateName.xml*, where *TemplateName* is replaced by the name of the template.

If one of the templates is selected from this list, the current flow graph is replaced with the contents of the template (and any changes made to the current flow graph are saved before the template is loaded). Templates are different to other flow graphs in that they cannot be edited. **Figure 19-9** shows a section of the flow graph where the *GL Integration* template has been selected. You can see that directly below the toolbar is a message stating that this is a protected view, and this template cannot be edited. The *Design Mode* button will also be disabled.

If you want to use a flow graph template as it is, or modify it in some way, you must first save it as your home flow graph name using the *Save As Home Flow Graph* option from the *Options* menu. After doing this you are free to edit this flow graph as the *Design Mode* button will be enabled.

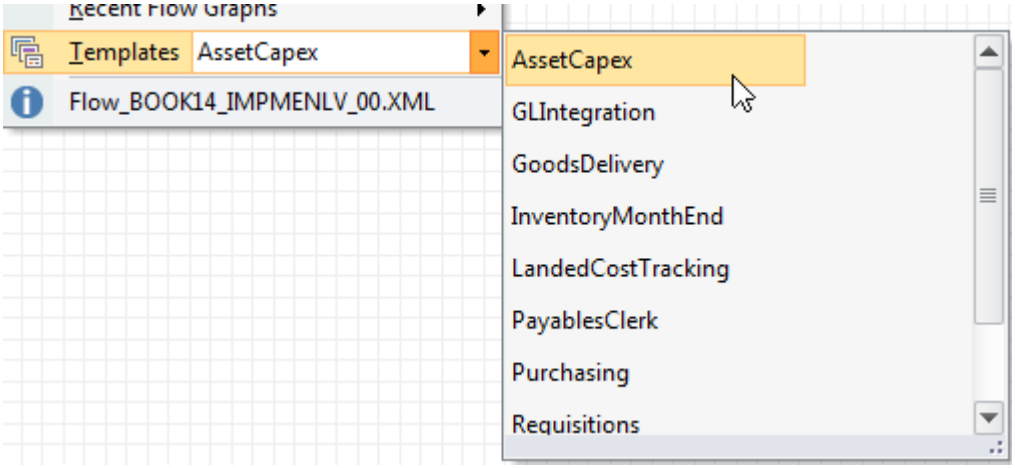


Figure 19-8: Displaying the list of available flow graph *Templates*

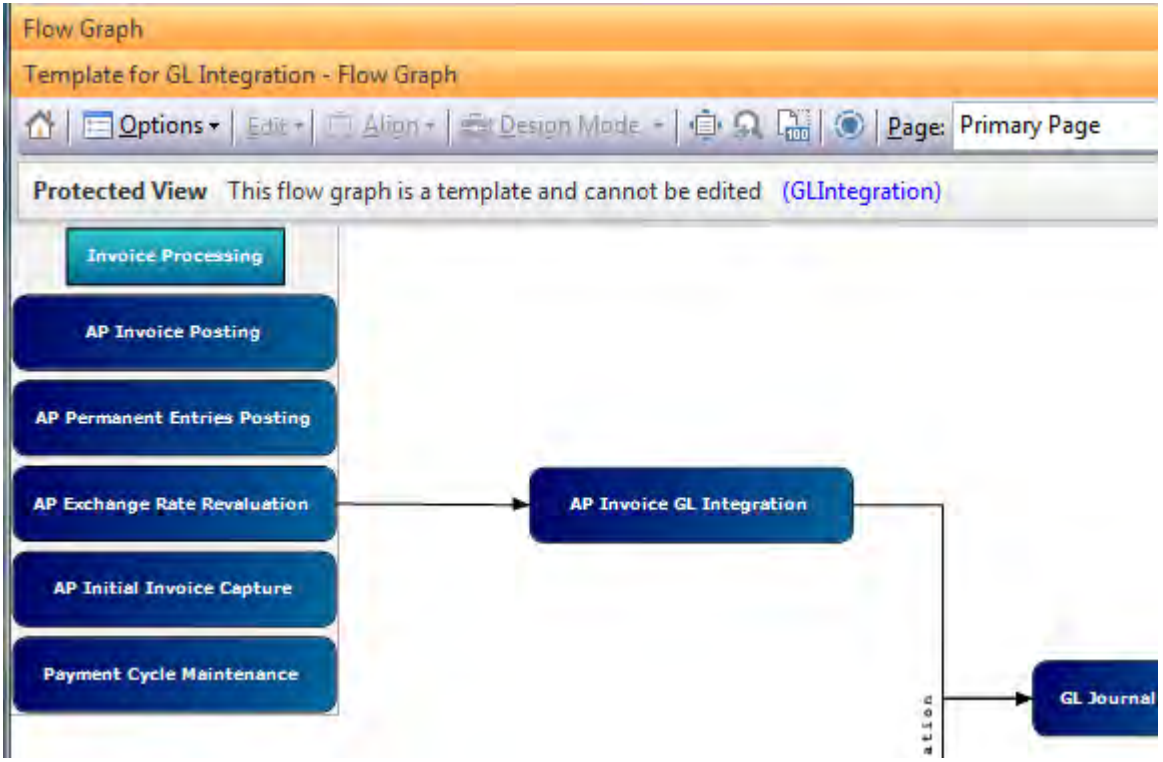


Figure 19-9: A flow graph showing the *GL Integration* template

The *Print* and *Print Preview* options both work as you would expect. The *Save as Image* option creates a .PNG file in your `Base\Settings` folder showing the currently displayed page of the flow graph. It then displays this .PNG file in your default image viewer, such and *Window Photo Viewer* (see **Figure 19-10**). The name of the .PNG file is the *OperatorCode_FlowGraph.PNG*, where *OperatorCode* is replaced with the operator code.

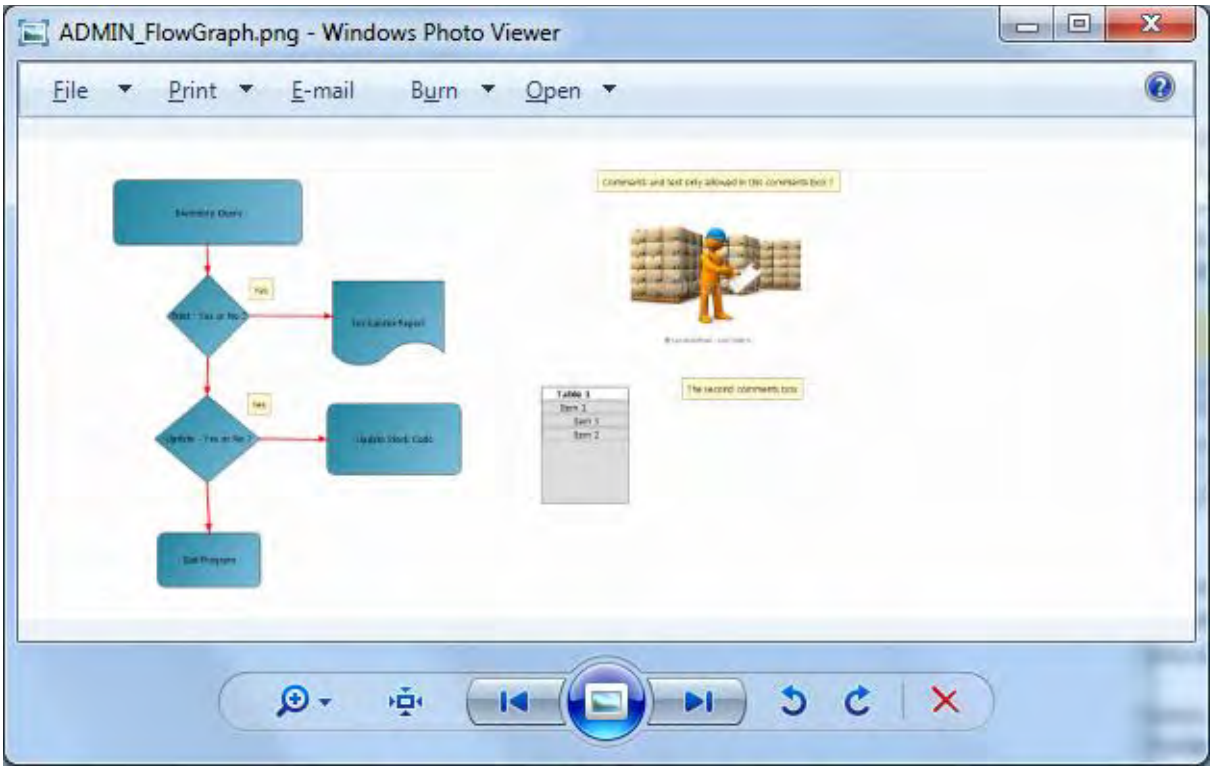


Figure 19-10: The *Save as Image* option displaying the .PNG image

Design Mode

Clicking on the *Design Mode* button puts the flow graph into design mode, which is where you add and modify flow graphs and pages. If this is the first time that you have entered design mode, a pane called *Shapes* will appear on the left. On the right will be a pane containing two tabs, *Flow Graph Properties*, and *Shape Properties*. The *Flow Graph* pane will appear in the middle. Some of the buttons on the toolbar that are only available in *Design Mode* will also become enabled.

Clicking a second time on the *Design Mode* button on the toolbar exits design mode, and displays the *Primary Page* of the currently selected flow graph. As you exit design mode the flow graph is saved. If you change to another flow graph while in design mode (because you want to work on a different

one), as you change to the next flow graph the previous one is saved for you automatically if any changes had been made.

Alongside the text on the *Design Mode* button is a downward facing triangle that contains a dropdown list. If you are in design mode when you click on this triangle, the options *Save and Exit Design Mode* and *Exit Design Mode without Saving* are displayed (see **Figure 19-11**). One exits without saving the changes. The other saves the changes, overwriting the existing flow graph.

Design Mode is covered in more detail within the *Flow Graph Design* section below.

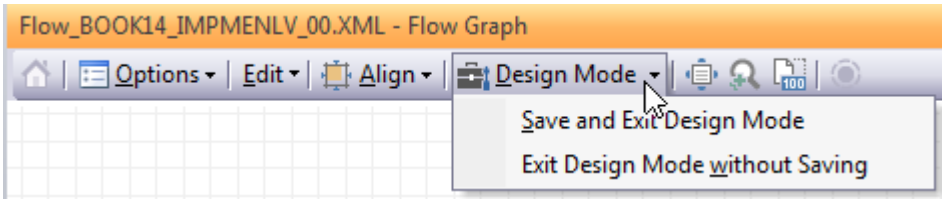


Figure 19-11: Using the different save options when exiting *Design Mode*

Flow Graph Design

Changes and additions to a flow graph, or pages within a flow graph, are performed in *Design Mode*. When you enter design mode for the first time, a pane called *Shapes* will appear on the left, and a pane will appear on the right that contains two tabs, *Flow Graph Properties*, and *Shape Properties*. The *Flow Graph* pane will appear in the middle. Some of the buttons on the toolbar that are only available in design mode will also become enabled.

Pages

Each flow graph has one or more *Pages*. Each page is like its own flow graph. All pages for a flow graph are held within the flow graph XML file, which makes it easier to deploy to another system than having multiple flow graphs. This also means that using the email option to send the flow graph will include all the pages.

Changing a Page's Name

When a new flow graph is created, a default page called *Primary Page* is created within it. This page name can be changed when in design mode by right-clicking on the page background and selecting *Page*, *Title*, and replacing the name of the page (see **Figure 19-12**). The *Primary Page* will always remain as the primary page, even when its name is changed, and cannot be deleted from the flow graph.

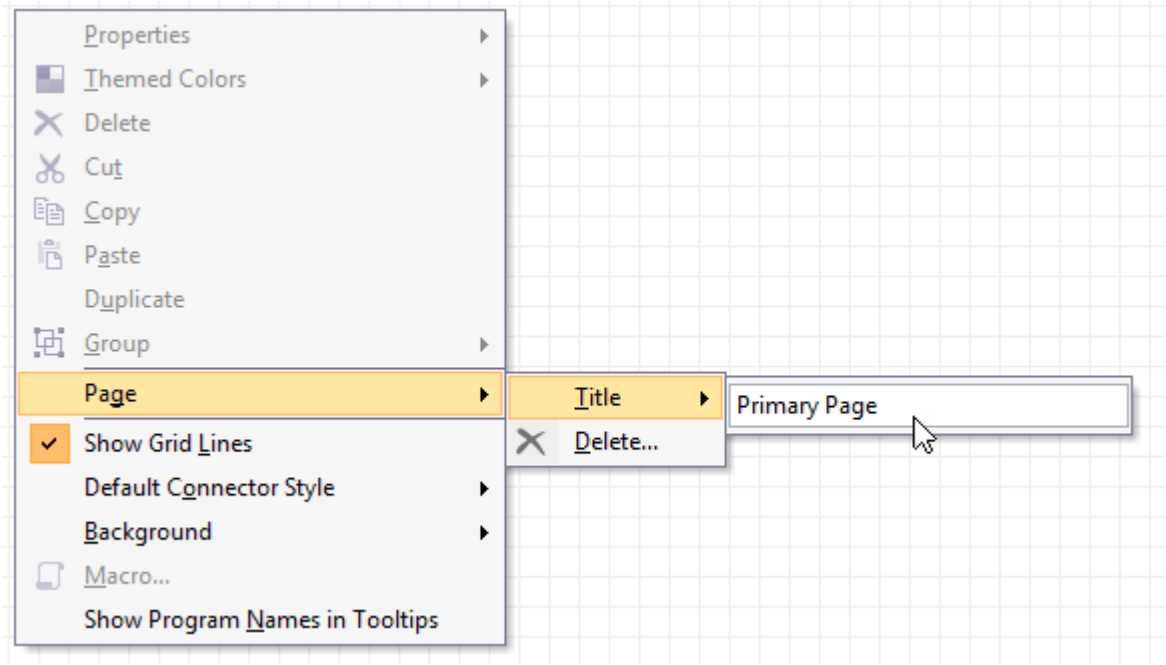


Figure 19-12: Changing the existing name of a page

Adding New Pages

New pages are added to a flow graph by typing their name against the *Page* prompt on the toolbar. When the operator presses the *Enter* key the existing information on the page is cleared, and they are working on the new page. **Figure 19-13** shows the new page name being entered.

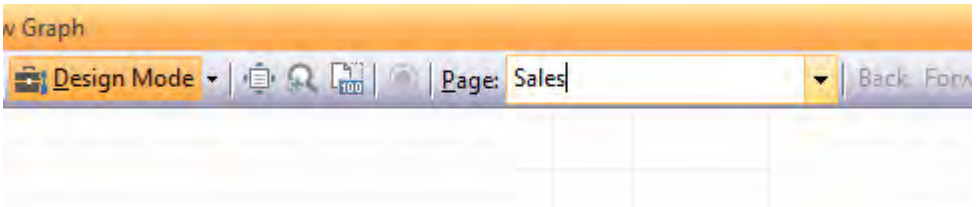


Figure 19-13: Adding a new page

Deleting Pages

Pages are deleted by right-clicking on the page's background and selecting *Page*, *Delete* from the context-sensitive menu (see **Figure 19-14**). The primary page against a flow graph cannot be deleted, even if its name has been changed from *Primary page* to something else. If you attempt to delete a flow graph's primary page you will receive the message "You cannot delete the first or primary page".

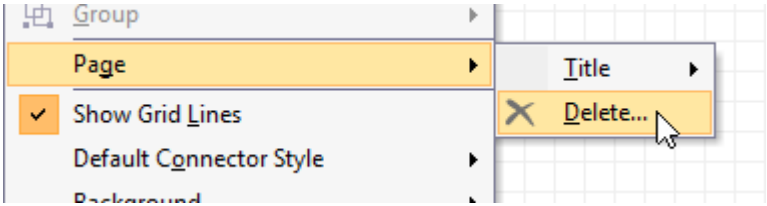


Figure 19-14: Deleting the current page

Flow Graph Background

The background grid can be removed by right-clicking on it and unchecking the *Show Grid Lines* option, which leaves a plain background. It can be redisplayed by checking this option.

The background color can be changed by right-clicking on it and selecting *Background* from the menu, and choosing a color. A background color can be removed by selecting the *Automatic* option (see **Figure 19-15**).

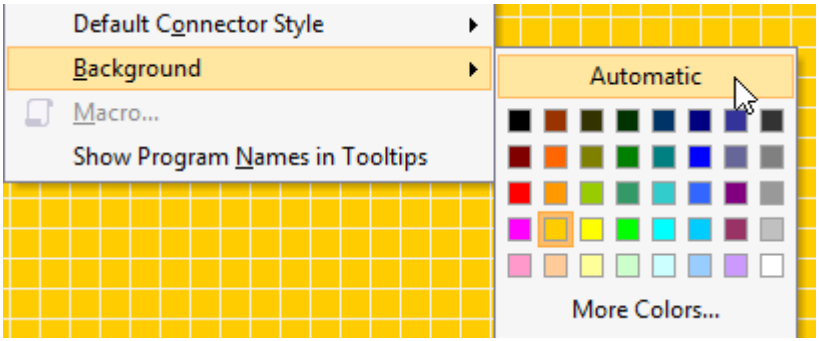


Figure 19-15: Setting the background color back to default using the *Automatic* option

Show Program Names in Tooltips

Also available from the menu that is displayed when you right-click on the *Flow Graph* pane's background is the option to include the program/report/application name in the tooltip, if the program/report/application was added using the *Add New Shortcut Wizard* (see **Figure 19-17**).

Macro

Each flow graph can have VBScript associated with it by right-clicking on the desktop when in *Design Mode*, and selecting *Macro* from the displayed menu. There are two *FlowGraph* events, *OnLoad* and *OnClicked*. The *OnLoad* event fires whenever the flow graph is loaded. If the script associated with the operator's *Home Flow Graph* contains an *OnLoad* function, the *OnLoad* event will fire as SYSPRO is loaded, and the code within this function will be executed. This will happen regardless of whether the

Flow Graph pane is visible to the operator or not. The *OnLoad* event also fires when you enter and exit *Design Mode*.

The *OnClicked* event is configured against the whole flow graph. The event is fired when the operator clicks on any shape on any page within the flow graph, and there is VBScript code against that flow graph's *OnClicked* function. Each shape has its own *ID*, so it is possible to detect which shape was clicked and process accordingly.

The section *VBScripting the Flow Graphs* covers adding VBScript to the flow graphs, and what can be done, in more detail.

Shapes Pane

The *Shapes* pane contains various shapes and geometric figures that can be dragged onto the *Flow Graph* pane. It is divided into four sections, *Main Flow Graph Shapes*, *Geometric Figures*, *Other Shapes*, and *Arrow Shapes*. Each of these sections can be contracted and expanded as screen real estate requirements dictate.

When a shape from one of the *Geometric Figures*, *Other Shapes*, or *Arrow Shapes* sections is dragged onto the *Flow Graph* pane it will appear with its default size/appearance (see the *Default Appearance* section later in the chapter). It will also have predefined *Connection points*, a default *Caption*, and a square around it made up of a dotted line with eight square points (see **Figure 19-16**) that is used to change its size and proportions.

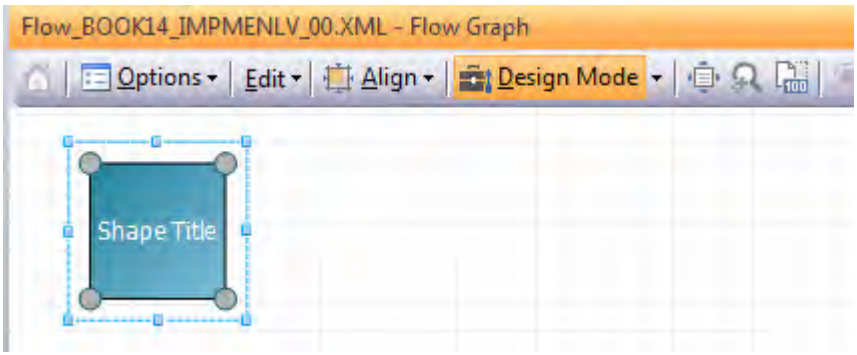


Figure 19-16: Adding a *Rectangle* shape to the *Flow Graph* pane

Dragging one of the corner points resizes the shapes proportionally. Dragging one of the central points on the horizontal plane allows the shape to be made taller or shorter, and dragging one of the central points on the vertical plane allows the shape to be made wider or narrower. Clicking on the main section of the shape enables you to drag the shape to a new location.

The properties of the shape can be configured by highlighting the shape on the desktop and setting the properties on the *Shape Properties* pane.

Each of the shapes within the *Main Flow Graph Shapes* section of the *Shapes* pane performs differently when they are dragged onto the *Flow Graph* pane. The *SYSPRO Program* shape creates a *Rounded rectangle* shape and automatically starts the *Add New Shortcut Wizard* (see **Figure 19-17**).

Figure 19-17: The *Add New Shortcut Wizard* that is started when adding the *SYSPRO Program* shape

This wizard enables the launching of a SYSPRO program, Report Writer report, a SYSPRO Reporting Services report, a third party Net Express application, or any other application. If the wizard is cancelled before it is completed the shape will become a normal *Rounded rectangle*, as if one had been dragged onto the *Flow Graph* pane from the *Geometric Figures* section.

The *Table* shape is a container for *Table item* shapes. These work in conjunction to represent a menu, or columns within a database table. **Figure 19-18** shows two tables that have been dragged onto the *Flow Graph* pane and their *Caption* properties set to *Inventory Master Table* and *Inventory Warehouse Table*. On each of the *Tables* you can see five *Table items*.

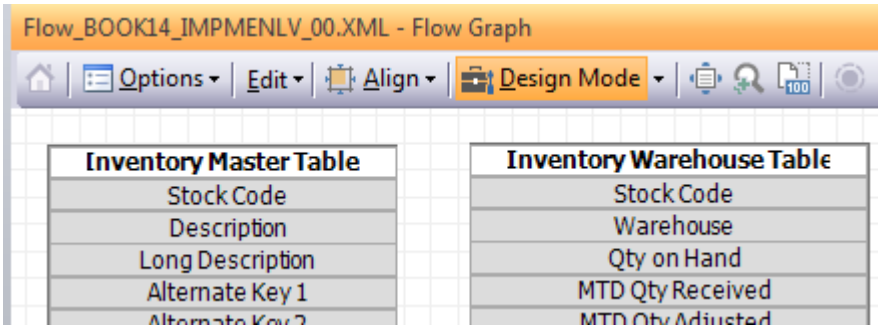


Figure 19-18: Two *Tables* each with five *Table items*

The *Picture* shape is used to add a picture or image to the *Flow Graph* pane's desktop. When the *Picture* shape is dragged onto the desktop a browse is opened enabling you to choose the picture to be displayed. Using the dotted lines around the *Picture* shape to reduce its size does not reduce it proportionally. Instead the image is cropped. Expanding the size of the image beyond its original size will not change the image at all.

The *Comment* shape is a means of adding text directly onto the flow graph's desktop. Once the *Comment* shape has been added to the flow graph's desktop the text is added using the *Caption* field on the *Shape Properties* pane. **Figure 19-19** shows a *Comment* shape where the *Caption* has been populated with text.

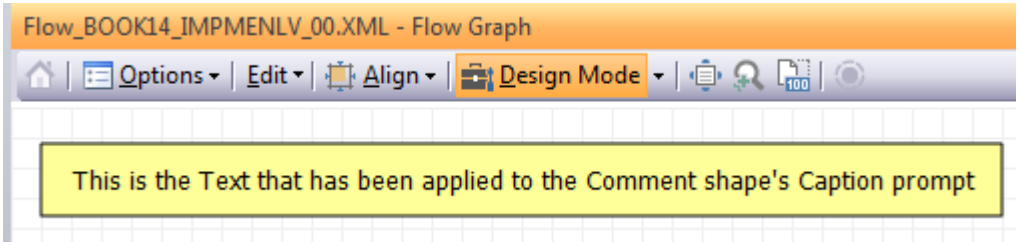


Figure 19-19: The *Comment* shape applied to the *Flow Graph* pane

Default Appearance

All of the shapes except *Table* and *Table item* can have a *Default Appearance* configured against them, and these defaults are set system-wide. The default appearance is configured by right-clicking on the shape within the *Shapes* pane, and selecting the *Default Appearance* option. A screen is displayed that contains the properties that can be set. These properties are the font, foreground color, background color, second background color, and whether this is solid color or gradient. The *Preview* displays the changes as they are made.

The toolbar contains options to save the settings for this shape, cancel out without saving these settings, restore this shape to its original settings, and restore all shapes back to their original settings. **Figure 19-20** shows the *Flow Graph Default Appearance* screen for the *Pentagon* shape.

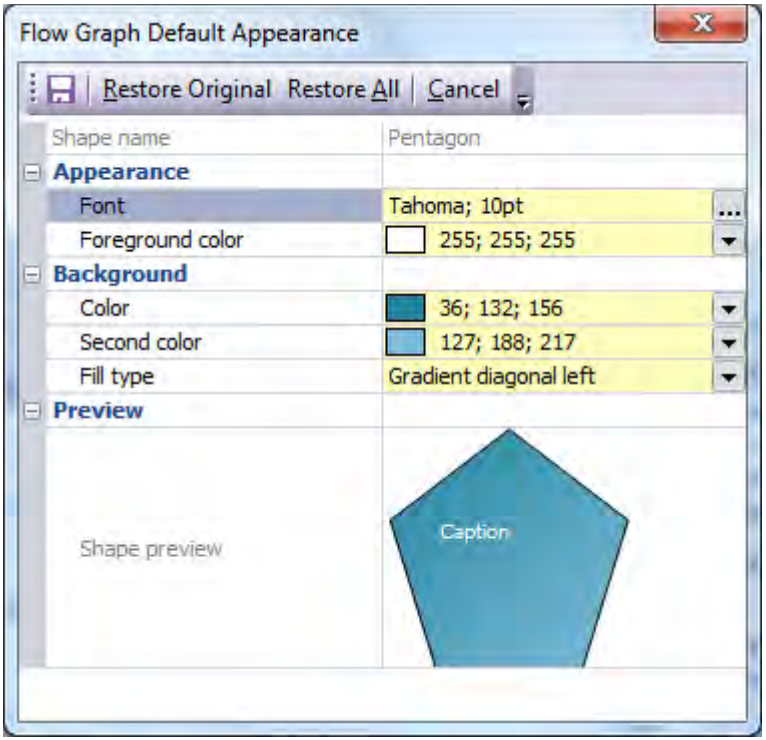


Figure 19-20: The *Flow Graph Default Appearance* screen for the *Pentagon* shape

Shape Properties Pane

When a shape is selected on the *Flow Graph* pane, such as when it has just been added, the *Shape Properties* pane is populated with the properties associated with this shape. **Figure 19-21** shows the *Shape Properties* pane for the *Rectangle* shape that appeared in **Figure 19-16**. Within the *Appearance* section are the *Caption*, *Tooltip*, *Caption position*, *Font*, *and Foreground color* fields.

For all the shapes from the *Geometric Figures*, *Other Shapes*, and *Arrow Shapes* sections, the *Shape type* field will contain the name *Shape*. This is also true of the *Comment* shape and *SYSPRO Program* shape from the *Main Flow Graph Shapes* section. The content of the *Shape type* field is read-only, and cannot be changed

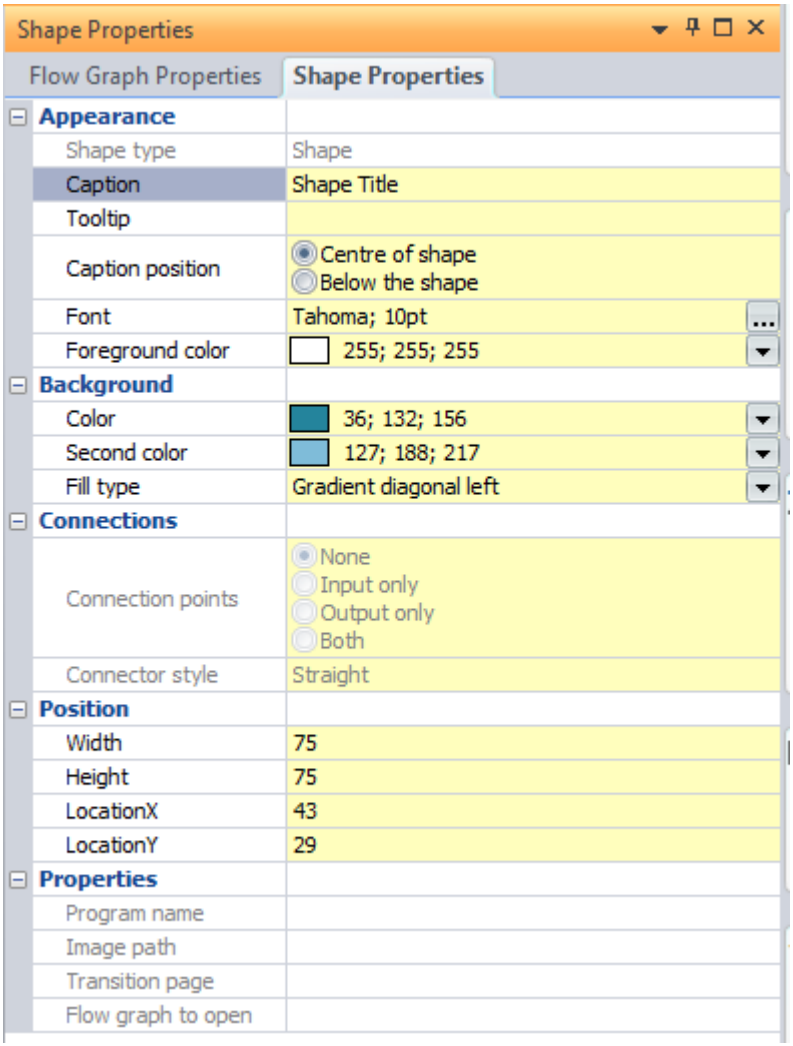


Figure 19-21: The *Shape Properties* pane for the newly-added shape that appears in **Figure 19-16**

The *Caption* field contains the caption that appears on/against the shape, which can be changed.

The *Tooltip* field can contain text that is displayed when the mouse pointer hovers over the shape. **Figure 19-22** shows where the *Caption* text has been changed to *Register the item*, and the *Tooltip* field also contains additional text.

The *Caption position* radio button works in conjunction with the *Caption* field. The choices are that the caption text appears in the centre of the shape, or appears below the shape. This can be set per

shape. **Figure 19-23** shows two *Rectangle* shapes, one configured with the *Caption position* set to *Centre of shape*, and the other to *Below the shape*.

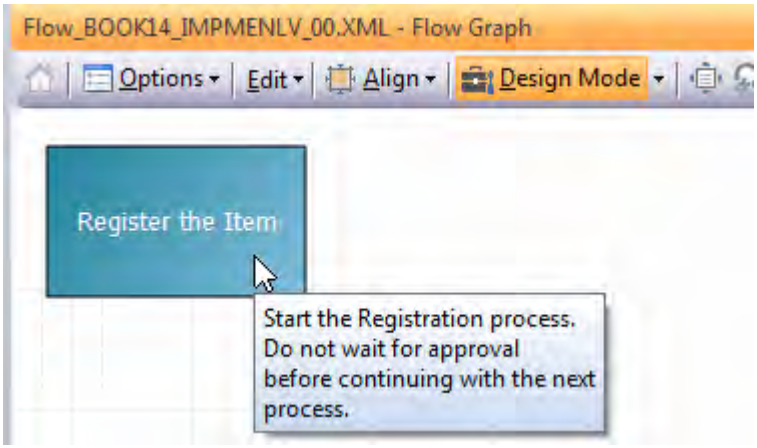


Figure 19-22: The *Caption* text and *Tooltip* text have been added

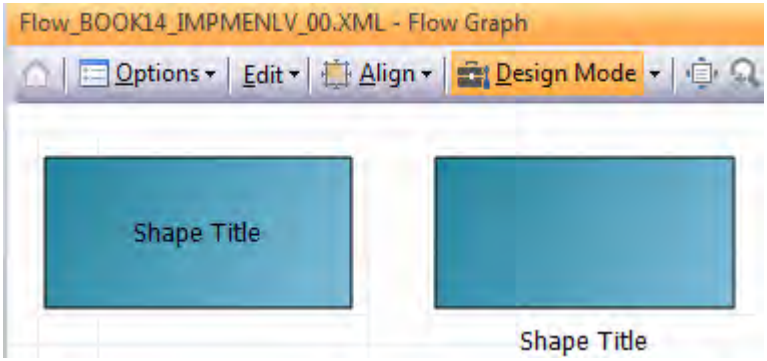


Figure 19-23: The *Caption* position in the *Centre of shape* and *Below the shape*

The *Font* specifies which font should be used for the caption, and the *Foreground color* specifies the color of the text used for the caption.

Within the *Background* section of the *Shape Properties* pane are the *Color*, *Second color*, and *Fill type* options. These work in conjunction to specify the color of the shape. If the *Fill type* field is set as *Solid*, the shape will appear in the color specified against the *Color* field. If the *Fill type* field is anything other than *Solid*, the shape will have a gradient that changes from the color specified against the *Color* field

to the color specified against the *Second color* field. The direction of the gradient depends on the option selected against the *Fill type* field.

The *Connections* section is used to define the connection points for a *Table item* shape, and the style of the connectors for those connecting other shapes. The *Connection points* option is only available if the currently selected shape is a *Table item*. It specifies which types of connections can be made to/from this table item. The options are *None*, *Input only*, *Output only*, and *Both*. **Figure 19-24** shows the *Inventory Master* table where the *Stock Code* table item has been configured with the *Connection points* option of *Input only*. This means that this table item can only have input connectors, which always connect to a point on the left.

Figure 19-24: Using the *Input only* connection point

The *Connector style* option is only available if a connector has been selected. It has a dropdown list against it. There are three styles of line (*Straight*, *Curved*, and *Elbow*) and each of these has the option of no arrows on the end, a single arrow (i.e. unidirectional flow), and double arrows (i.e. bidirectional flow).

The *Position* section of the *Shape Properties* pane shows the current *Width*, *Height*, location on the X-axis (*LocationX*), and on the Y-axis (*LocationY*). Any change made to the shape by dragging the whole shape, or one of the sides/corners is reflected here. This section can also be used to change the dimensions and position of the shape. The new values can be typed over the existing values, and when focus is lost from the field (by tabbing to the next field or using the mouse pointer to click somewhere else) the shape will change to reflect these new values.

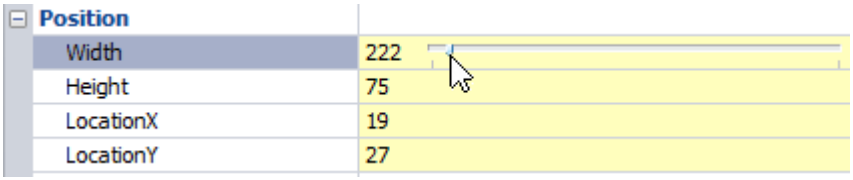


Figure 19-25: Modifying the shape's height using the *Height* property slider bar

If one of these fields is selected using the mouse pointer (see **Figure 19-25**), a slider bar will appear alongside it. The value of this field can be changed by clicking on a position along the slider bar.

The *Properties* section of the *Shape Properties* pane contains four read-only fields, *Program name*, *Image path*, *Transition page*, and *Flow graph to open* (see **Figure 19-26**). The *Program name* field will contain the name of a *SYSPRO program*, *SYSPRO customized report* (report writer report), *Any Net Express program*, *SYSPRO Reporting Services report*, or *Any other application* that was configured using the *Add New Shortcut Wizard* (see **Figure 19-17**).

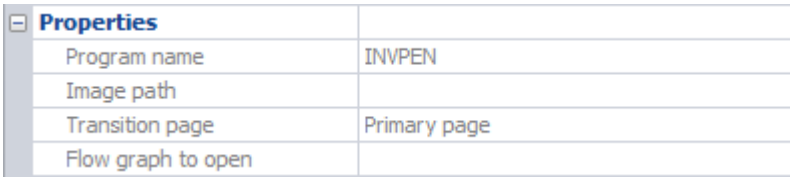


Figure 19-26: The *Properties Section* of the *Shape Properties* pane

Note that if the program to be run was added using the *SYSPROProgramToRun* or *SYSPROBrowseToRun* variables within the VBScript associated with this flow graph (see **Figure 19-27**) the name of the program will not appear against the *Program name* property.

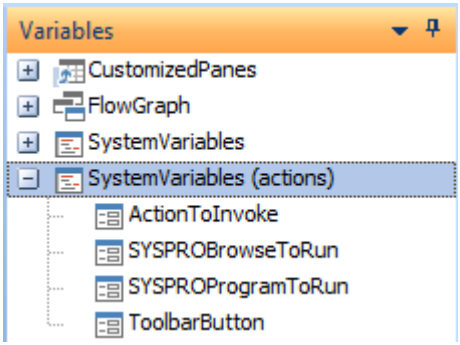


Figure 19-27: The *SYSPROProgramToRun* and *SYSPROBrowseToRun* variables

The *Image path* field will only be populated if the shape currently selected is a *Picture* shape, in which case the path and image name will be displayed.

The *Transition page* field will only contain a value if the *Action* against the currently selected shape has been configured to transition to another page, in which case it will contain the name of the other page within this flow graph.

The *Flow graph to open* field will only contain a value if the *Action* against the currently selected shape has been configured to transition to another flow graph, in which case it will contain the name of the other flow graph. **Figure 19-28** shows the *Action Settings for Shape* screen where the transitions are configured or removed. *Actions* are covered in more detail later in this chapter.

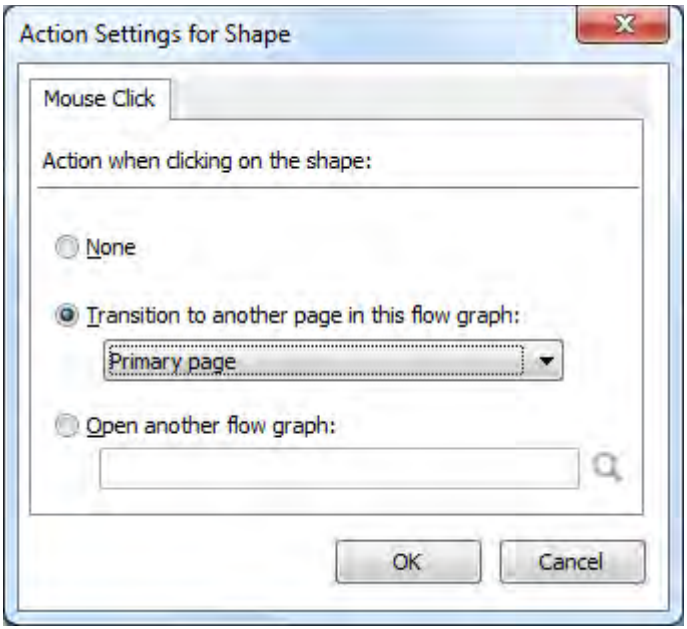


Figure 19-28: Configuring the shape to transition to another *Flow Graph* or *Page*

Flow Graph Properties Pane

The *Flow Graph Properties* pane by default will contain the date that this flow graph was created, along with the date that it was last modified. However, it can be configured to contain other information that can make the administration of the flow graphs easier.

Figure 19-29 shows a *Flow Graph Properties* pane that has been fully-populated. Populating the *Author* field is useful if multiple people can add/maintain flow graphs. It will enable you to contact the author to request changes, or notify them of changes that you want to make, or have already made.

The *Title* field enables you to supply a title for your flow graph, and the *Status* field enables you to enter a free format status description. The *Comments* field is also a free format field.

The *Auto fit to window* option specifies that when a flow graph is opened (that is not the operator's home flow graph) the content of the flow graph must be expanded/contracted to fit the size of the flow

graph window at the time that it is opened. If the flow graph window size is subsequently changed by the operator it will not cause the flow graph to resize again during the time that it is open. If you transition to another page on either the home flow graph, or another flow graph, the content of the flow graph also expands/contracts to the size of the window.

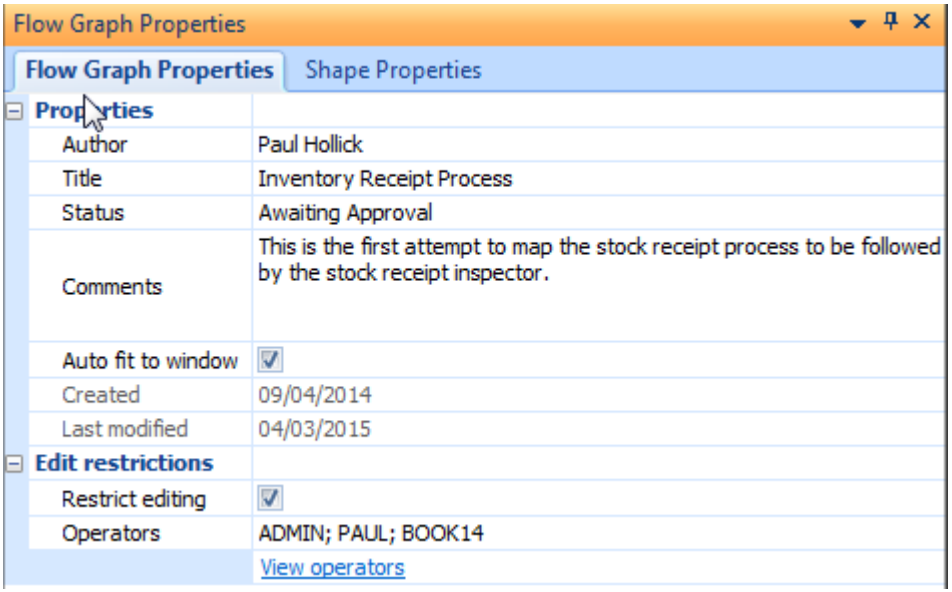


Figure 19-29: A fully-populated *Flow Graph Properties* pane

The *Restrict editing* checkbox, *Operators* field, and *View operators* hyperlink are all interlinked and enable you to restrict which operator codes can edit this flow graph. The *View operators* hyperlink calls up the *Operators* browse so that you can find the correct operator code. The operator code(s) are added manually against the *Operators* field. If more than one operator is allowed to edit the flow graph the operator codes are delimited with semi-colons. If checked, the *Restrict editing* checkbox restricts the editing to the operator list. If not checked, no restrictions are in place.

Note that this restriction does not apply to an operator's *Home Flow Graph*. If their *Flow Graph – Allowed to design flow graph* operator *Activity* is set so that they are allowed to design flow graphs, they will be allowed to enter *Design Mode* for their own *Home Flow Graph*, even if the restriction against the flow graph states that they should not be allowed to do so.

Adding Shapes

When in *Design Mode*, *Shapes* are added to the *Flow Graph* pane by dragging them from the *Shapes* pane, and they remain at the point where the mouse button is released. The shapes within the

Geometric Figures, *Other Shapes*, and *Arrow Shapes* sections of the *Shapes* pane all behave in the same manner. When they are released they remain selected, their connection points are visible, they have a default color/graduation, and a default caption. They also have a square around them made up of a dotted line with eight square points that is used to change its size and proportions (see **Figure 19-30**). The exception is the *Line* shape that only uses the *Color* property and does not apply the graduation to the *Second color*.

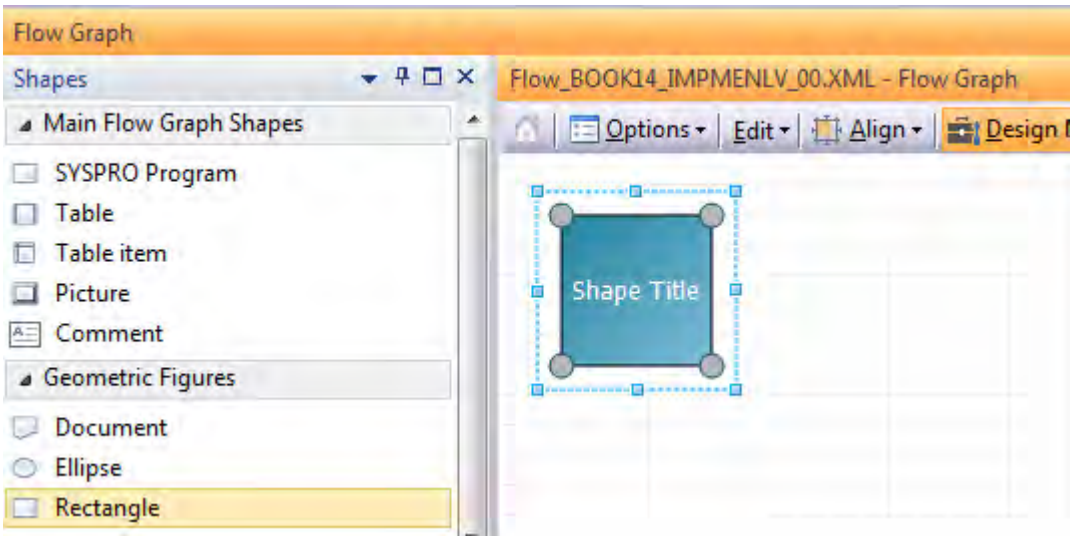


Figure 19-30: The status of a *Shape* after being added to the *Flow Graph* pane

Dragging one of the corner points resizes the shape proportionally. Dragging one of the central points on the horizontal plane allows the shape to be made taller or shorter, and dragging one of the central points on the vertical plane allows the shape to be made wider or narrower. Clicking on the main section of the shape enables you to drag the shape to a new location.

The dimensions, location, color, caption, font, font color, tooltip, and location of the caption can all be set using the *Shape Properties* pane (covered in detail above). As other shapes are added to the flow graph, connectors can be used to join connection points on one shape to connection points on another. Some shapes may have many connection points available around their perimeter (see **Figure 19-31**).

The *Connectors* are created by clicking on one of the connection points of the first shape, and dragging the mouse pointer to one of the connection points of the second shape. As the mouse pointer moves over the second shape the connection points on this shape become visible. The connection point that will be used if you release the mouse button will appear in a different shade to the other connection points, and as you move around the highlighted connection point will change (see **Figure 19-32**). See the *Connectors* section below for more details.

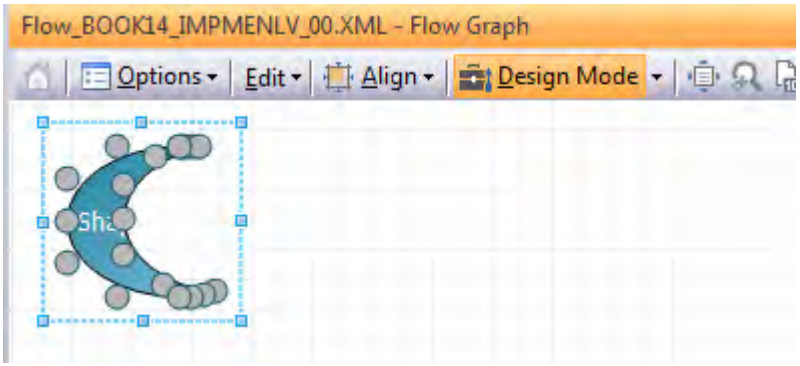


Figure 19-31: The *Moon* shape showing multiple connection points around its perimeter

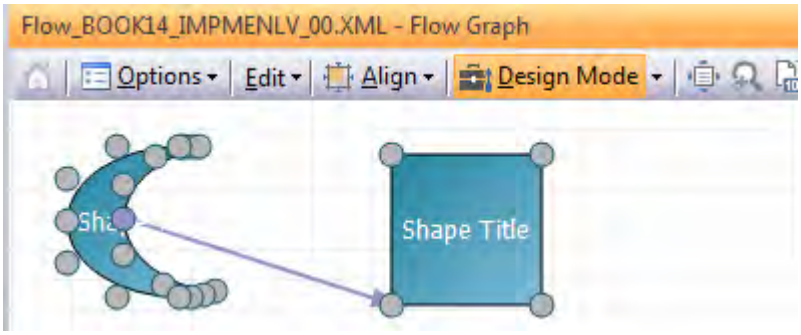


Figure 19-32: Using *Connectors* to connect two shapes

Each shape can have a SYSPRO program, report writer report, SRS report, Net Express program, or any other application associated with it. These are added using the *Add New Shortcut Wizard* which is launched by right-clicking on the shape and selecting *Properties* | *Program Details* from the displayed menu (see **Figure 19-33**). The *Add New Shortcut Wizard* first screen can be seen in **Figure 19-17**.

If you select to add a *SYSPRO Program* shape you are prompted for the program name (which has a browse) and a name for the shortcut (which defaults to SYSPRO's description for this program), then the shape is updated. The shape's *Caption* is replaced with the shortcut name supplied during the wizard. A tooltip is automatically added that contains the full description that would appear against this program in the standard SYSPRO menu. The *Caption* and *Tooltip* can be changed in the *Shape Properties* pane for this shape.

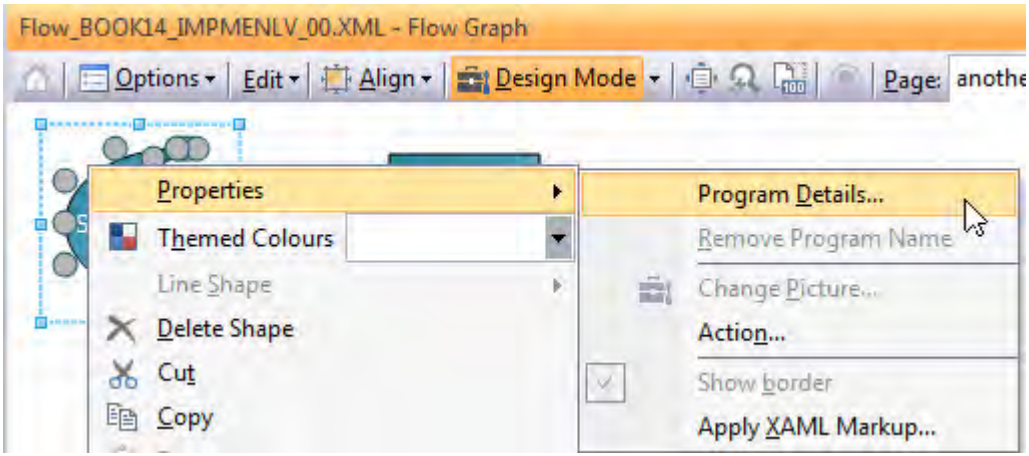


Figure 19-33: Launching the *Add New Shortcut Wizard* for a shape

If you select to add a *SYSPRO customized report*, you are prompted to enter the name of a report writer report (which also has a browse) and the description (which defaults to the report's description) before the shape is updated.

If you select to add *Any NetExpress program* you must enter the program's name and path (which also has a browse) and the description which defaults to the program name. The shape is then updated.

If you select to add *Any other application*, you are prompted for the application name (and optional start in folder) along with a description, before the shape is updated.

If you select to add a *SYSPRO Reporting Services report* you are prompted to choose between standard SRS reports and user-defined ones. A list of the relevant reports is displayed and you select from this list. You are prompted to supply a description, which defaults to the report's description. The shape is then updated.

Note that any *Action* configured against the shape will be performed before a program, customized report, or third party application is run, so if you have both configured against a shape, only the *Action's* transition will occur because you will have moved away from the shape before the shortcut is run.

If the option *Show Program Names in Tooltips* is checked, the tooltip will include the program, report, or application name at the bottom of the tooltip. The *Show Program Names in Tooltips* option is available in *Design Mode* by right-clicking on the *Flow Graph* pane's desktop (see **Figure 19-34**).

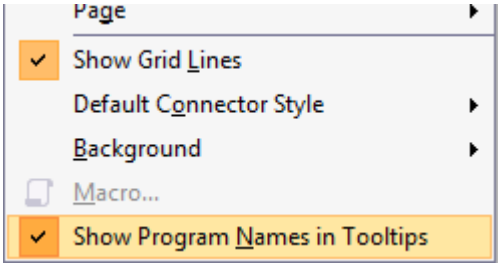


Figure 19-34: Setting the *Show Program Names in Tooltips* option

Figure 19-35 shows the tooltip against the *Moon* shape that has been configured to call the *Inventory Query* program. The tooltip description defaults to the program's description, and as the *Show Program Names in Tooltips* option is checked, the program name of ***INVPEN*** appears below it.

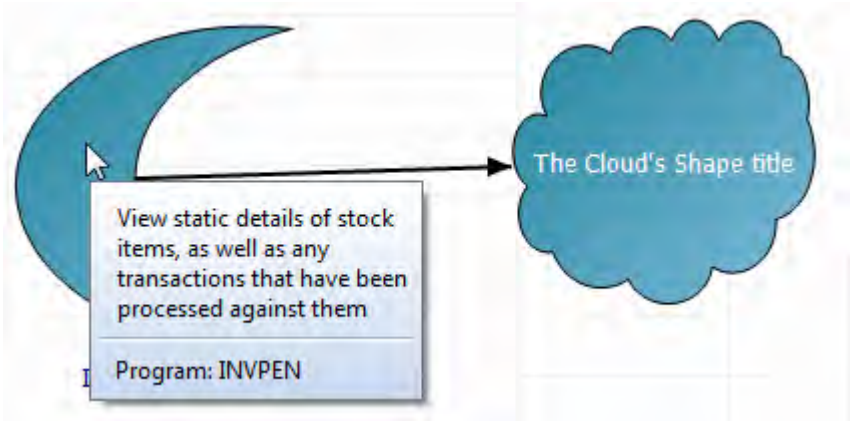


Figure 19-35: The tooltip with the program name and description displayed

If this shape was configured to call a *SYSPRO customized report*, the name of the *Report Writer* report would appear alongside the tooltip's *Program* caption. If this shape was configured to call *Any NetExpress program*, or *Any other application*, the path and program name would appear against the tooltip's *Program* caption. For *SYSPRO Reporting Services reports* the name that appears against the tooltip's *Program* caption is the name of the program that is launched (that calls the SRS report).

If, after using the *Add New Shortcut Wizard* to configure a program against a shape, you need to change the program name, this can be done by right-clicking on the shape and selecting *Properties* | *Program Details* from the displayed menu (see **Figure 19-36**).

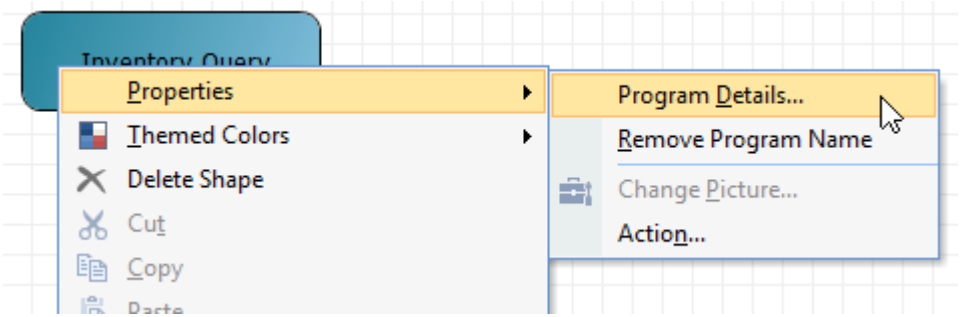


Figure 19-36: Changing the *Program Details* against a shape

The *Properties* screen is displayed where you can change the program name (against the *Program* prompt) as well as the caption for the shape (against the *Name* prompt) (see **Figure 19-37**).

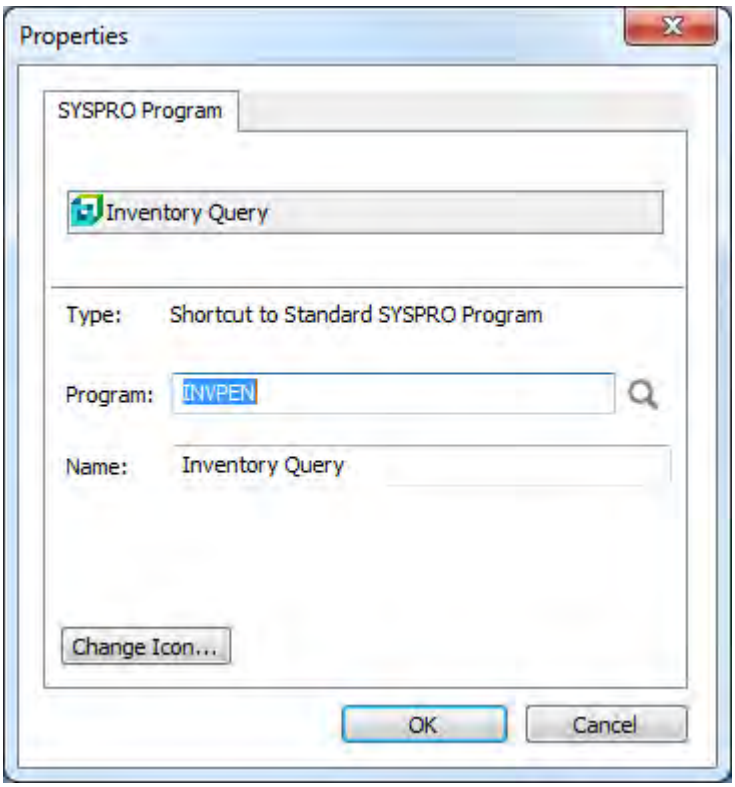


Figure 19-37: Changing the shape's *Program* and *Name*

Note that you cannot change this from one shortcut type (such as *SYSPRO program*) to any of the other available types within the *Add New Shortcut Wizard* or the *Properties* pane. To do this you must first remove the *SYSPRO program*, then run the *Add New Shortcut Wizard* again and select the required option.

If you no longer want any SYSPRO program to be associated with this shape you can use the *Remove Program Name* option that can be seen underneath the *Program Details* option in **Figure 19-36**).

The other program/application/report types work in a similar way where you can change the program/application/report to be run, but you cannot change between types.

For the shapes that have been dragged to the *Flow Graph* pane from the *Geometric Figures*, *Other Shapes*, or *Arrow Shapes* sections of the *Shapes* pane, the *Themed Colors* option is available when right-clicking on shape and selecting *Themed Colors* from the menu (see **Figure 19-38**). As the mouse pointer moves over each theme name, the shape's color will change to match. When you click on the theme that you want to use, the shape's color will remain this color. If you click on the shape, the *Shape Properties* pane will reflect the new colors/graduation.

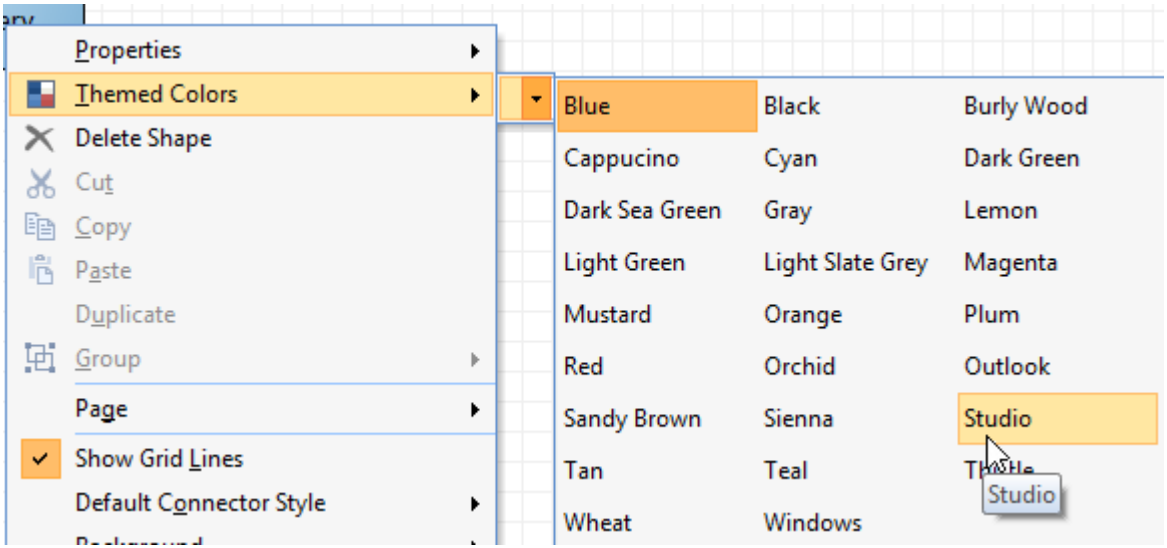


Figure 19-38: The *Themed Colors* option

Table and Table Item Shapes

A *Table* shape is just a placeholder for *Table Item* shapes. These are typically used to represent menus, or a SQL Server table. **Figure 19-39** shows where a *Table* shape has been dragged onto the *Flow Graph* pane. Once positioned, you can change its caption, tooltip, font, color, location and size using the *Shape Properties* pane.

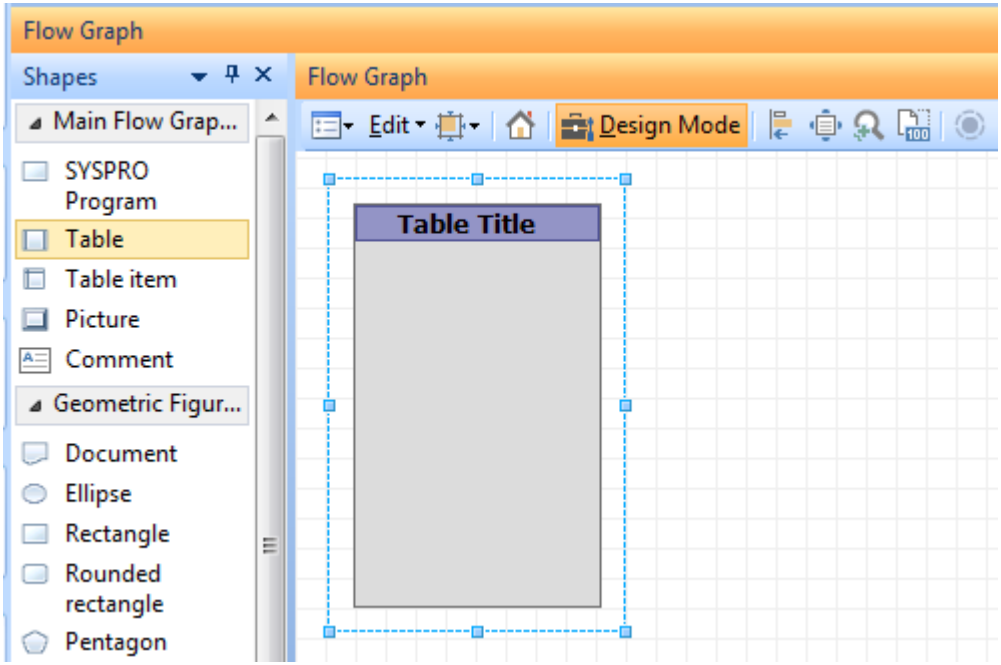


Figure 19-39: Adding a *Table* shape

Table Item shapes can then be dragged onto the *Table* shape, and their caption, tooltip, text color, and connection points can be changed using the *Shape Properties* pane. Note that the only place that a *Table Item* shape can exist is on a *Table* shape, and its size is related to the size of the *Table* shape on which it sits. **Figure 19-40** shows a combination of a *Table* and *Table* item shapes representing the *AR Customer Master table*.

Figure 19-41 shows where two tables have been joined using a connector. The *Customer* table item in the *Customer Master Table* has been configured with the *Output only* radio button against the *Connection points* option in the *Shape Properties* pane. The *Customer* table item on the *Private* table has been configured with the *Both* radio button against the *Connection points* option.

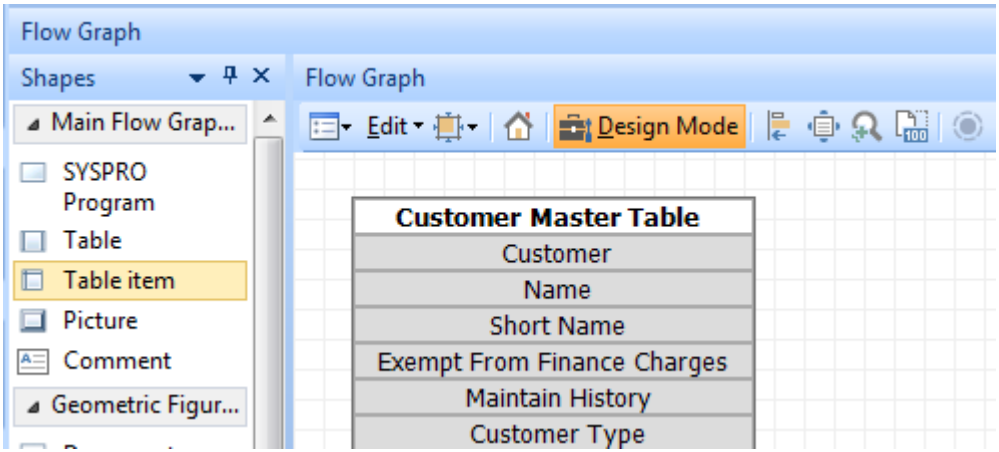


Figure 19-40: A *Table* populated with six *Table Items*

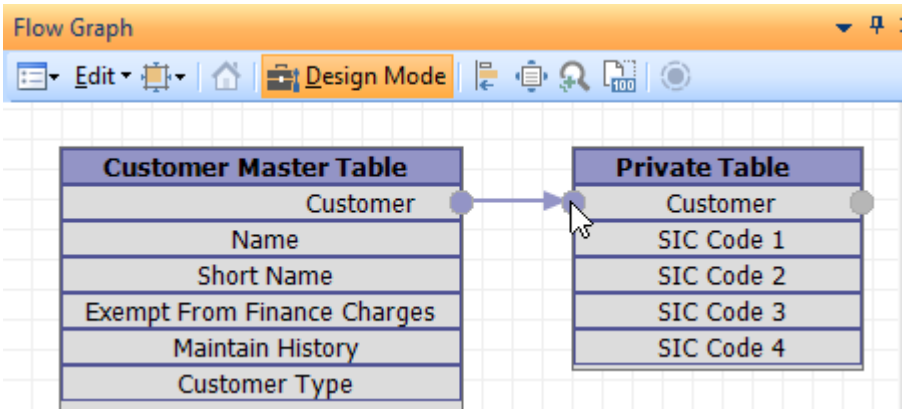


Figure 19-41: Using *Connection Points* to add a *Connector* between two *Table Items*

Unlike other shapes, you cannot assign a SYSPRO program, customized report, Net Express program, application, or SRS report to a *Table* shape or *Table Item* shapes. Although the VBScripting can detect when a *Table* or *Table Item* shape has been clicked, it cannot detect which was clicked, so it is unlikely that you will be able to use this in your VBScripting.

Picture Shape

The *Picture* shape is used to add an image to the *Flow Graph* pane's desktop. When a *Picture* shape is dragged onto the desktop, a browse is opened enabling you to choose the image to be displayed. You can choose from .jpg, .bmp, and .gif images (selecting an image with another format will mean

that the image is not displayed on the desktop). The *Picture* shape will be created with the minimum size needed to completely display the image. Using the dotted lines around the *Picture* shape to reduce its size does not shrink the image; it removes the portion of the picture that would be outside of the dotted line. Expanding the size of the image beyond its original size will not change the image at all.

The image associated with this *Picture* shape can be changed by right-clicking on it and selecting *Properties* | *Change Picture*, from the displayed menu. See the section *Deploying the Flow Graph* regarding the location of the image, and how this is stored against the shape.

A caption can be added to this *Picture* shape, which can be configured to appear across the image or below. A tooltip can also be applied. There are four connector points against a Picture shape, and each can be an input and an output. The *Add New Shortcut Wizard* can be invoked by right-clicking on the image and selecting *Properties* | *Program Details* from the menu, so you can assign a program/application/report to be run when the image is clicked.

Within the VBScripting environment you can detect that this image was clicked using the *ID*, so you can make use of this in your VBScripting.

If the image to be used is located in the SYSPRO `Base\Samples` folder, a relative address is used instead of an absolute path. The relative address will contain *{samples}* followed by the name of the image. For example, if you add a picture shape using the FlowTemplate_Assets.jpg image it would be stored as:

```
{samples}FlowTemplate_Assets.jpg
```

Using the browse to select the image from the `Base\Samples` folder causes the relative pathname to be inserted automatically.

Comment Shape

The *Comment* shape is a means of adding text directly onto the *Flow Graph* desktop. Once the *Comment* shape has been added to the flow graph's desktop, the text is added using the *Caption* field on the *Shape Properties* pane. If the text entered against the *Caption* field is longer than the current length of the *Comment* shape, the text is centred in the shape and the extra text is truncated on both ends. If you change the *Comment* shape's size in any way the *Comment* shape will automatically expand to the minimum size that is required to include all of the entered text. The text always appears on one line, and the shape cannot be made shorter than the size of the text by dragging any of the squares that appear around the shape when it is selected. **Figure 19-42** shows the *Comment* shape where text was added that was greater than the size of the shape. As soon as one of the squares around the edge was dragged, the shape jumped to the minimum size to contain the text.

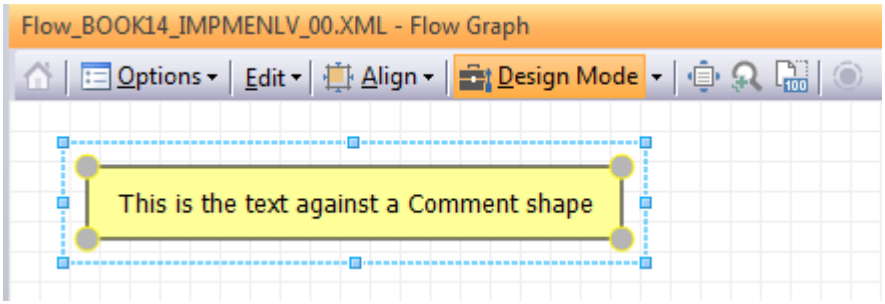


Figure 19-42: The *Comment* shape expanded to the size of the text

The rectangular shape around the text can have a solid color applied to it, or have two colors with a gradient (horizontally, vertically, or diagonally). The border around the comment shape can be removed unchecking the *Properties* | *Show border* option (see **Figure 19-43**).

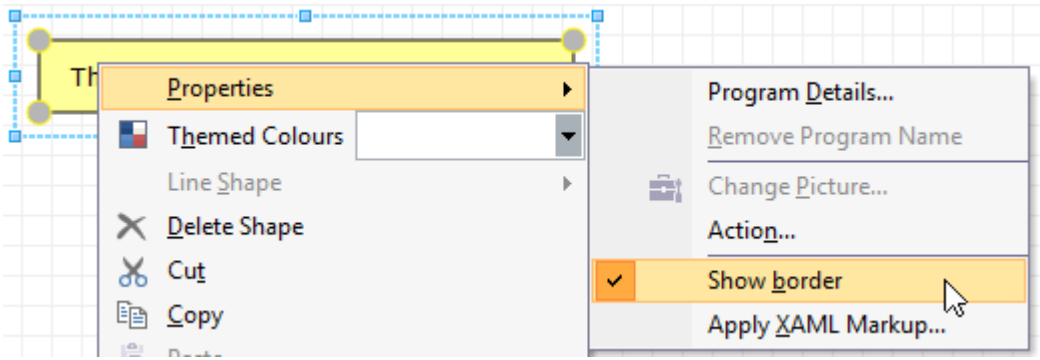


Figure 19-43: Unchecking the *Show border* option

If the desktop grid lines are removed, the border is removed from the comment shape, and its background is changed to match the color of the desktop. The text will appear to be directly on the desktop (see **Figure 19-44**).

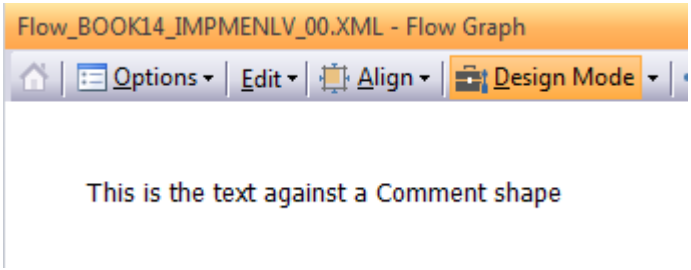


Figure 19-44: Desktop grid lines and the shape's border removed, and background color set to white

A tooltip can be added, and the shape around the text has connection points.

Within the VBScripting environment you can detect that this *Comment* shape was clicked using its *ID*, so you can also make use of this in your VBScripting.

Actions

An *Action* can be applied to all shapes (excluding *Table* and *Table Item* shapes) and enables you to transition to either another page on this flow graph, or another flow graph. *Actions* are added in *Design Mode* by right-clicking on the shape and selecting *Properties* | *Action* from the menu. The *Action Settings for Shape* screen is displayed where you define which type of action is to be performed (see **Figure 19-45**).

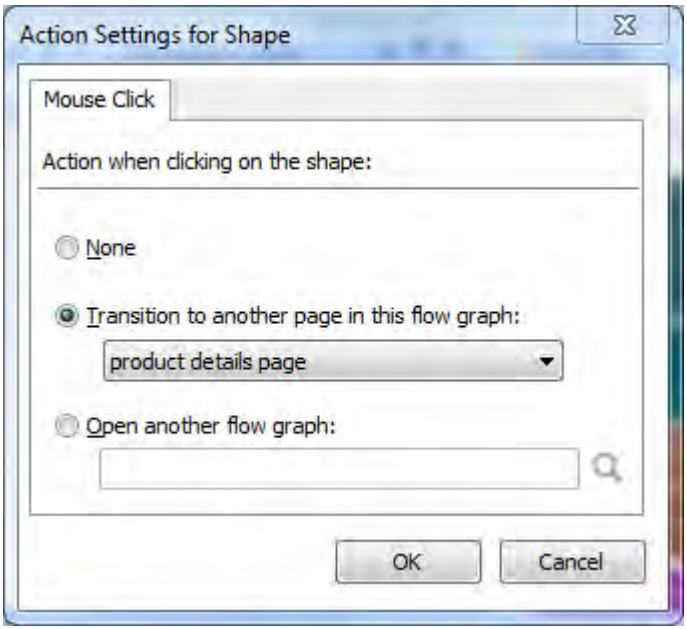


Figure 19-45: The *Action Settings for Shape* screen where *Actions* are configured

The *None* option enables you change settings from transitioning to another page/flow graph back to not performing a transition.

The *Transition to another page in the flow graph* option has a dropdown link next to it containing all the pages in this flow graph. The *Open another flow graph* option has a browse button next to it. If this is selected, a browse window will be displayed and the *Files of type* option will specify

FlowGraphs(.xml)*. The default location for flow graphs is SYSPRO's `Base\Settings` folder. Flow graph names default to starting with Flow_*OperatorName* (where *OperatorName* is replaced with the operator code).

If an *Action* is configured against a shape to transition to another page within the same flow graph and there is no *Action* to get back, the operator can use the dropdown list against the *Page* caption on the flow graph toolbar to select the required page name. Alternatively, the *Back* button on the toolbar will have become enabled, which when clicked will return to the previous page in the same flow graph.
If an *Action* is configured against a shape to transition to another flow graph, and there is no *Action* to get back, the operator can either use the *Recent Flow Graphs* option or *Open a Flow Graph* option against the *Options* menu on the toolbar to select the correct flow graph name.

Connectors

The *Connectors* are created by clicking on one of the connection points of the first shape, and dragging the mouse pointer to one of the connection points of the second shape. As the mouse pointer moves over the second shape the connection points on this shape become visible. The connection point that will be used if you release the mouse button will appear in a different color to the other connection points, and as you move around the highlighted connection point will change. This can be seen in **Figure 19-32**.

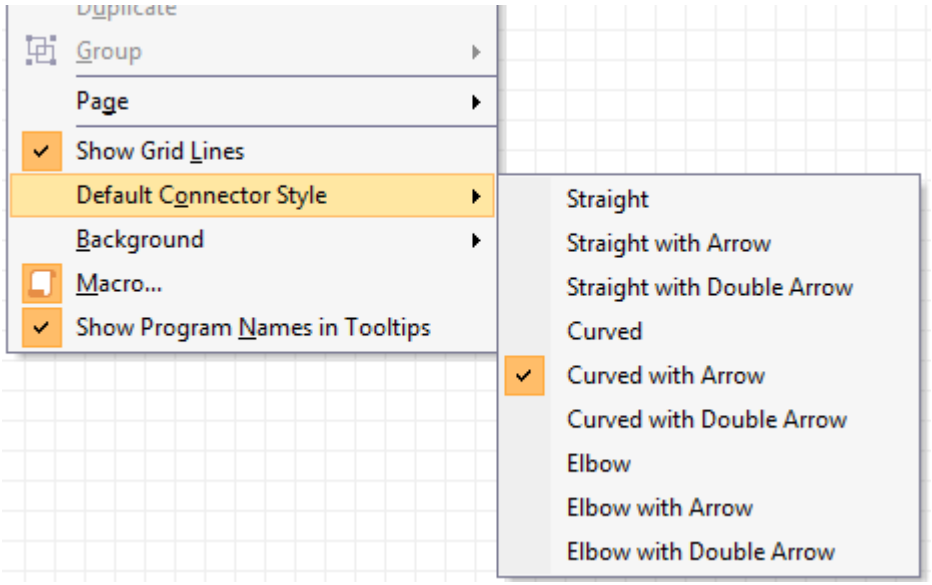


Figure 19-46: Setting the *Default Connector Style* of a *Connector*

There are nine different connector types, three straight, three curved, and three with elbows. Each of these can be configured to have no arrows, a single arrow, or a double arrow. The *Default Connector Style* is selected by right-clicking on the flow graph desktop when in *Design Mode* . Selecting the *Default Connector Style* option and choosing from the displayed list. **Figure 19-46** shows the list of available styles.

Choosing one of these options sets the *Default Connector Style* for this operator. If it is changed while in *Design Mode*, when the operator exits and calls up a flow graph that contained connectors in another style, these will appear in the newly-selected format.

An angle can be added to a straight line while in *Design Mode* by clicking on the shaft of the arrow and dragging it (see **Figure 19-47**).

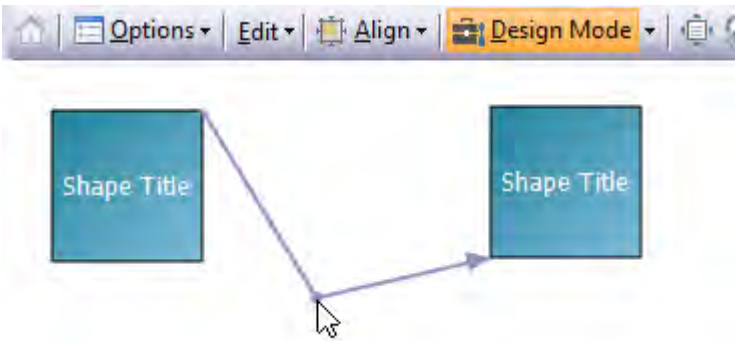


Figure 19-47: Dragging a straight arrow to create an angle

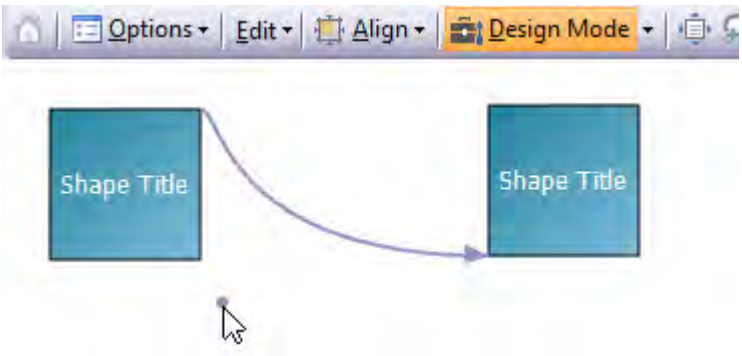


Figure 19-48: Adjusting the bend of a curved arrow

A similar technique is used to change the bend in a curved arrow, except that the point moves at a different rate to the curve. **Figure 19-48** shows where the point has been dragged from the arrow's shaft to create a curve that is steeper at the beginning and becomes flatter nearer the pointed end.

Connectors defined as having an elbow cannot be manipulated in this way.

Connectors can be deleted when in *Design Mode* by selecting them, then right-clicking and selecting the *Cut* option. Deleting a shape that has a connector linked to it will delete the connector too.

Line Shape

The *Line* shape is a line that can be configured to be horizontal or vertical (but not diagonal), and have different properties set against it. When dragged onto the flow graph desktop it will be a horizontal line (see **Figure 19-49**). Its length can be changed by dragging the squares on the border nearest to the end of the line. Dragging one of the squares in the middle of the border along its length will move the vertical position of the line. Dragging one of the squares in the corner will change its length and its vertical position.

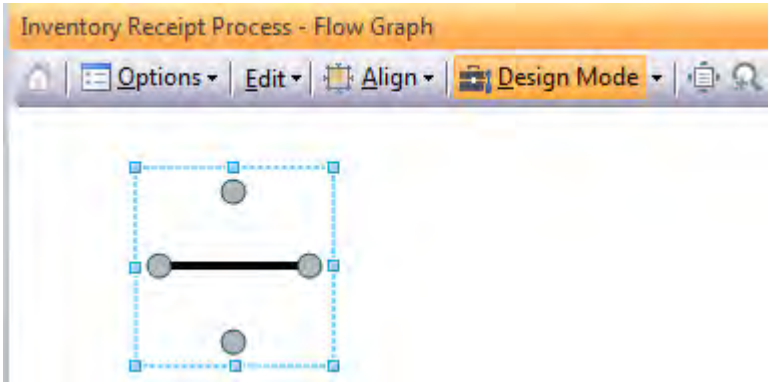


Figure 19-49: The initial state of a Line shape when dragged onto the flow graph

The line can be changed to a vertical line by right-clicking on the shape and selecting *Line Shape* | *Orientation* | *Vertical* from the displayed menu (see **Figure 19-50**)

The shape of the beginning and end of a line can be set using the *Begin Arrow Type* and *End Arrow Type* options (which can be seen in **Figure 19-50**). The available types for each are *Square*, *Round*, and *Triangle*. **Figure 19-51** shows three lines. The top one has both the *Begin* and *End Arrow Types* set to *Square*. The middle one has them set to *Round*, and the bottom one has them set to *Triangle*.

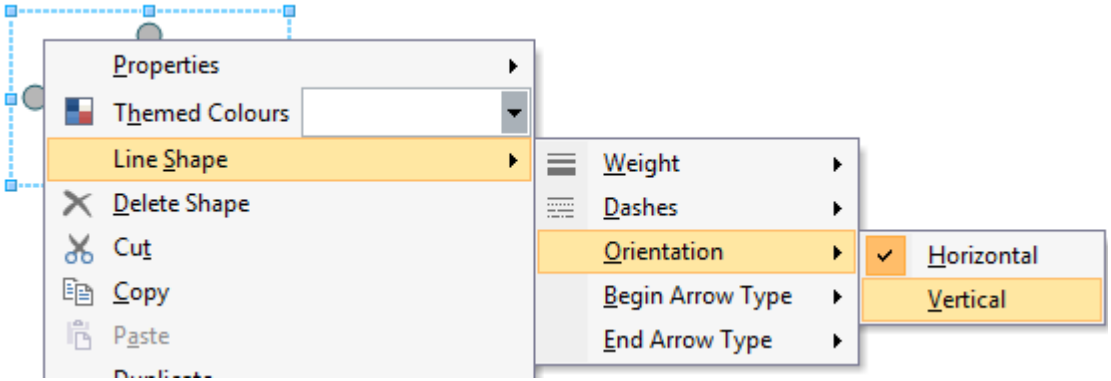


Figure 19-50: Changing the line's orientation to *Vertical*

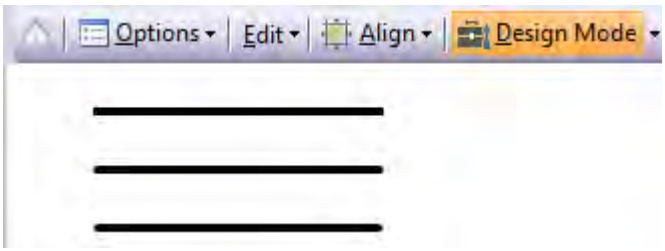


Figure 19-51: The three different begin and end arrow types

The type of line can be set using the *Dashes* option (that can be seen in **Figure 19-50**). **Figure 19-52** shows five lines, with each set to a different dash type. From the top these are set to *Solid*, *Dash*, *Square Dots*, *Dash Dot* and *Long Dash*.

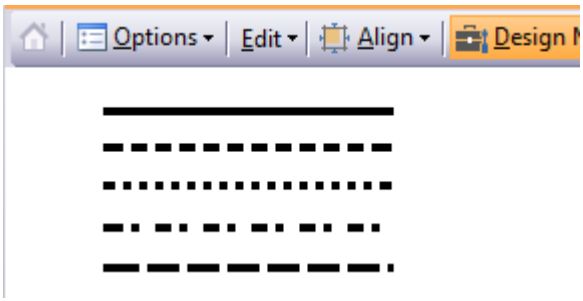


Figure 19-52: the five different types of *Dashes*

The thickness of a line is set using the *Weight* option, which uses a slider. The slider can be seen in **Figure 19-53**. **Figure 19-54** shows two lines, the first is the thinnest line that can be drawn, and the second is the thickest. If the slider is moved completely to the left the line is still present, it just can't be seen.

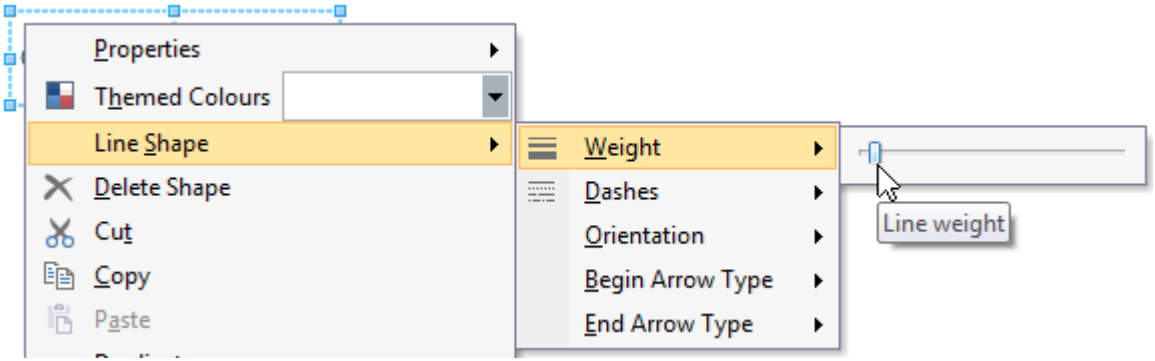


Figure 19-53: Using the slider bar to change the weight/thickness of the line

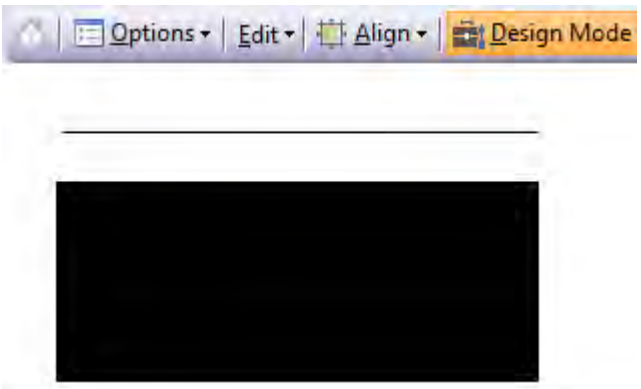


Figure 19-54: One line set to a small weight, and one set to the largest

Other than these special properties the line shape behaves in the same way as any other shape (you can set its color via the *Shape Properties* pane or via the *Themed Colors*, you can detect in VBScript when it has been clicked, etc.

Other Ways of Adding SYSPRO Programs to a Flow Graph

While in *Design Mode*, SYSPRO programs can be added to the flow graph by dragging the program from the SYSPRO menu under the *Program List* pane and dropping them on the flow graph desktop.

They can also be dragged from the *Recent Programs* section under the *Navigation Pane*. When these are added this way the result is the same as if you had dragged the *SYSPRO Program* shape onto the flow graph, but without needing to go through the *Add New Shortcut Wizard*, as the type and program name is already known.

Grouping

When in *Design Mode*, shapes can be grouped together by lassoing them (see **Figure 19-55**). The shapes within the selected area have their connection points highlighted.

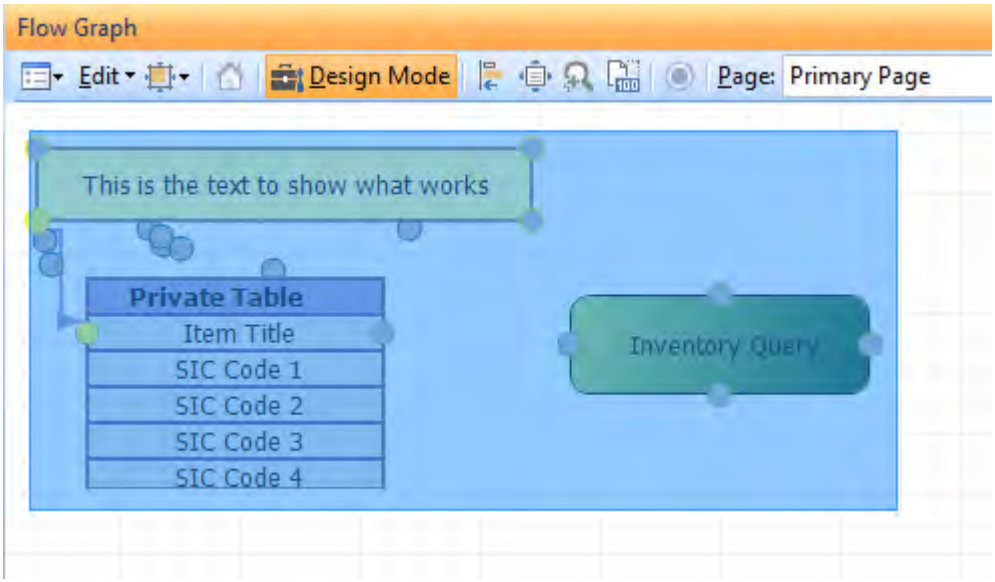


Figure 19-55: Lassoing three shapes so that they can be grouped

When you release the mouse button the highlighting color will disappear but the connection points of the selected shapes will remain highlighted. An alternate means of selecting all the items for grouping is to click on the first item, then hold down the *Ctrl* key and click on all the other shapes to be grouped.

Dragging one of the highlighted shapes will move all the shapes together. This grouping is temporary until you click on another shape, or on the flow graph pane's desktop. You can make this grouping permanent by right-clicking within the group and selecting *Group* | *Group Selected Nodes* from the menu (see **Figure 19-56**).

Once these shapes have been grouped using the *Group Selected Nodes* option, when you exit *Design Mode* you will still be able to see the grouping. **Figure 19-57** shows how three shapes appear after exiting *Design Mode* when they have been grouped this way.

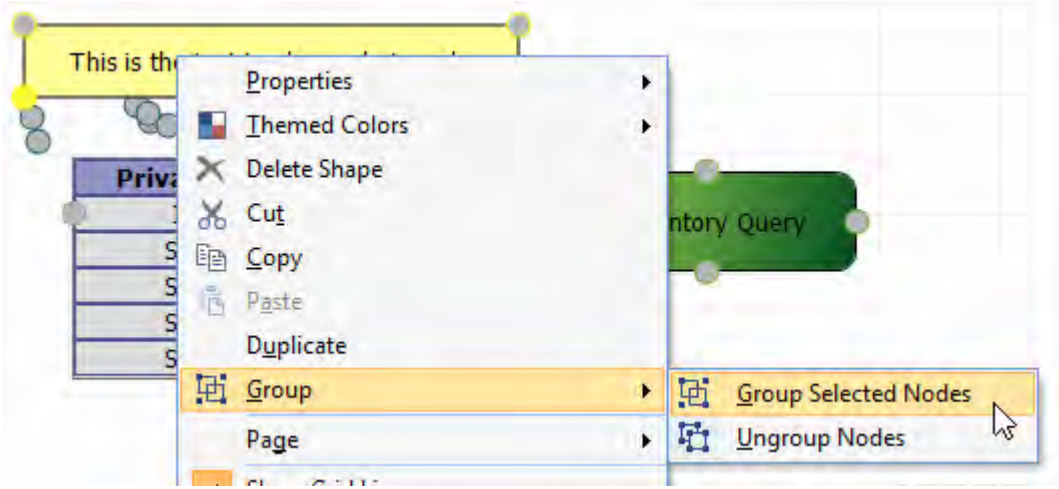


Figure 19-56: Using the *Group Selected Nodes* to group three shapes

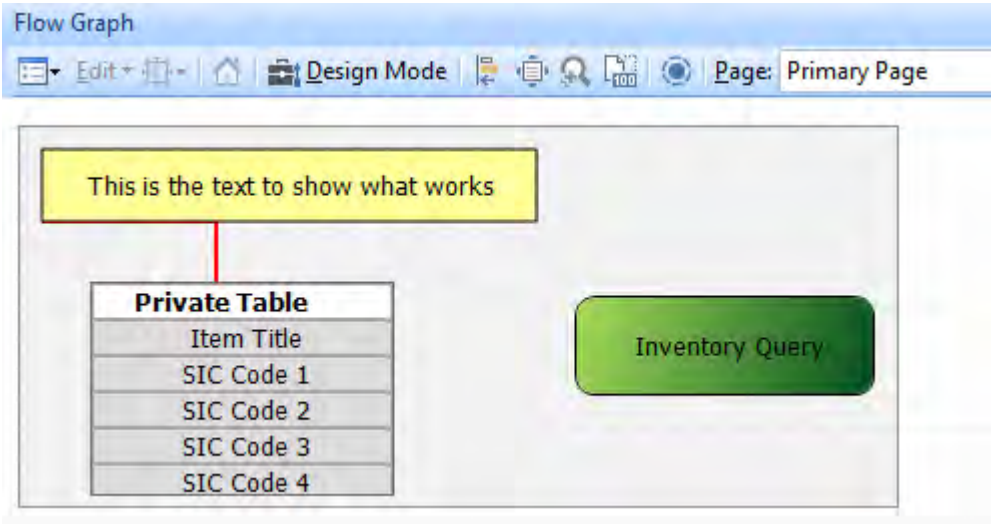


Figure 19-57: How grouped shapes appear when not in *Design Mode*

If you no longer want these shapes to be grouped, in *Design Mode*, click on one of the grouped items and the grouping will be highlighted. Right-click on any shape within this grouping and select *Group* | *Ungroup Nodes* from the menu.

Duplicate / Cut / Copy / Paste / Delete a Shape

In *Design Mode*, shapes can be duplicated by highlighting the shape, then right-clicking on it and selecting the *Duplicate* option. This assists you in having consistency with regard to the styling of the shapes within a page.

If a *Table* is duplicated, any *Table Items* associated with the *Table* will also be duplicated. With all shapes other than a *Table*/*Table Item*, the new shape is identical to the existing one except that the shape's *Node ID* is different (if this ID was not different you would not be able to detect which of the two items were clicked with the *OnClicked* event in the VBScripting). Using the *Copy* and *Paste* options give exactly the same results. The *Cut*, *Copy*, *Paste*, and *Delete* options work as you would expect.

VBScripting the Flow Graphs

Macro events are available for flow graphs in a similar way to the events against a form. However, unlike a form (where events can also be associated with individual fields) the events against a flow graph are only associated for the whole flow graph.

The *VBScript Editor* screen is invoked by right-clicking on a shape, or any part of the flow graph's desktop while in *Design Mode*, and selecting the *Macro* option.

There are two events that can be associated with individual flow graphs, *OnLoad*, and *OnClicked*. The *OnLoad* event is fired each time that the flow graph is loaded. The *OnClicked* event fires whenever the operator clicks on any of the shapes.

There are four variables under the *FlowGraph* section of the *Variables* pane, *Desktop Alert*, *FlowGraphNodes*, *NodeClickedCaption*, and *NodeClickedID*. These can be seen in **Figure 19-58**, and will be covered later in the chapter.

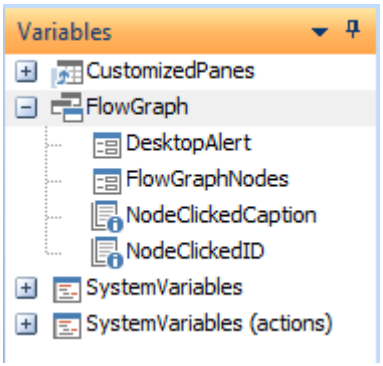


Figure 19-58: The *FlowGraph* section of the *Variables* pane

OnLoad
The *OnLoad* event fires whenever the flow graph is loaded. If the script associated with the operator's default flow graph contains an *OnLoad* function, the *OnLoad* event will fire as SYSPRO is loaded, and the code within this function will be executed. This will happen regardless of whether the *Flow Graph* pane is visible to the operator.

When an operator uses the *Open a Flow Graph* option from the *Options* button of the flow graph toolbar, if the flow graph to which they are changing has code against its *OnLoad* function, the event will fire and the code will be executed.

If an *Action* has been configured against a shape to transition to another flow graph and it is clicked, the event will fire and the code will be executed if the flow graph to which you are transitioning has code against its *OnLoad* function. Entering and exiting *Design Mode* will also cause the flow graph to be loaded, so if it has script against its *OnLoad* function this will be executed at these times.

OnClicked
The *OnClicked* event is configured against the whole flow graph. The event is fired when the operator clicks on any shape on any page within the flow graph, and there is VBScript code against that flow graph's *OnClicked* function. The *OnClicked* function can be stopped by setting the function to *false* within the function, as per the example below.

```
FlowGraph_OnClicked = false
```

The *FlowGraphNodes* variable (covered in detail later in this chapter) can be used to detect which node item (shape) was clicked.

All shapes (apart from *Tables* and *Table Items*) can be configured to run a program/application/report, can be configured to perform tasks within VBScript when it is clicked, and can be configured to transition to another page or flow graph.

If present, the VBScript is run first, followed by any transition, and then the running of a program/application/report that was applied to a shape using the *Add New Shortcut Wizard* (instead of those configured to run within the VBScript using the *FlowGraphNodes* variable, *SYSPROProgramToRun*, or *SYSPROBrowseToRun* system variables). If a transition occurs during this process, the program/application/report will be prevented from running as the transition occurs first.

Desktop Alert
The *Desktop Alert* variable is used to display a message on the operator's screen when a certain condition has been met. Double-clicking on the *Desktop Alert* variable name displays the *Desktop Alert Settings* screen where you can configure what the alert will contain. **Figure 19-59** shows the *Desktop Alert Settings* screen being populated. The *Duration* defines how long (in seconds) the alert

will stay on the screen unless you interact with it. The *Animation style* specifies how the alert will appear/disappear.

Within the *Heading line* section you can add a title for the alert against the *Text* prompt, and an icon can be chosen from a list to appear against this title. Against the *When clicked* prompt is a dropdown list where you specify what will happen if the operator clicks on the alert's heading. The choices are *Do nothing*, *Launch SYSPRO program*, or *Run executable*. If either of the last two options are selected a browse enables you to locate the SYSPRO program or executable.

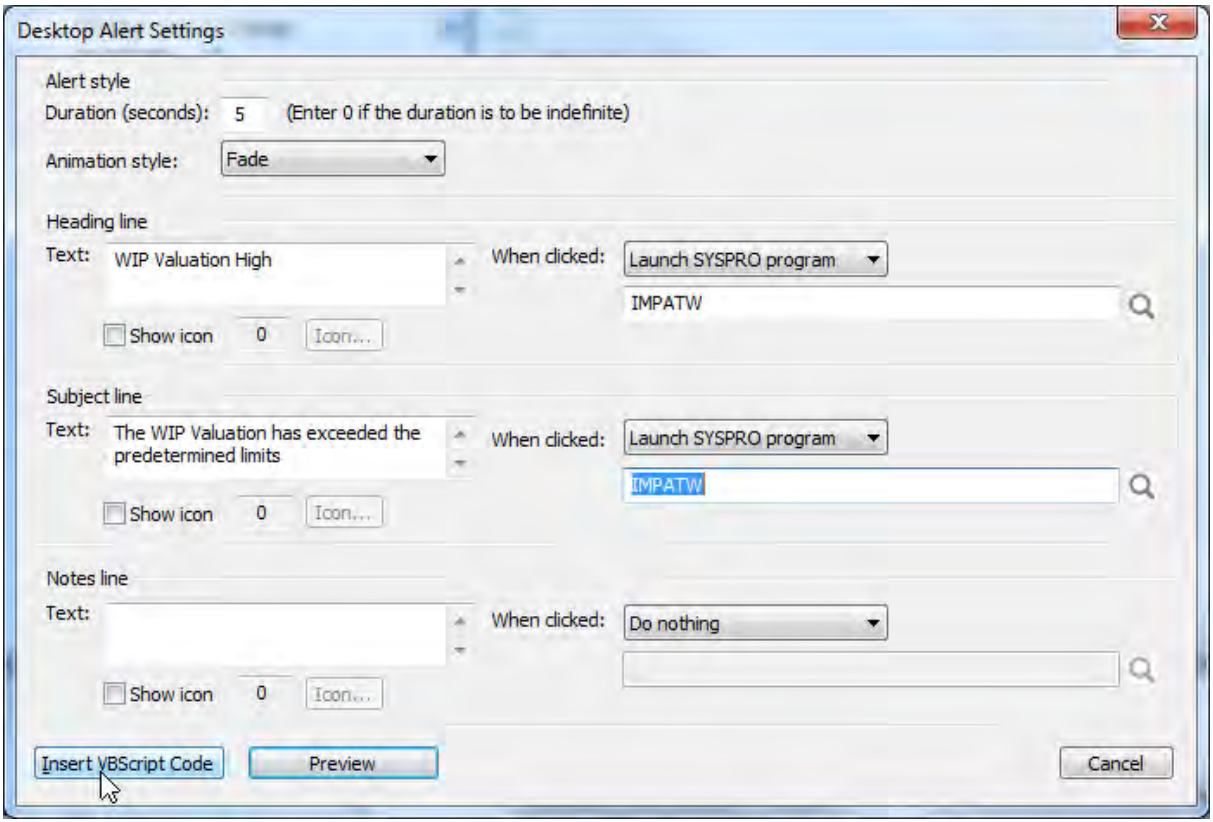


Figure 19-59: The *Desktop Alert Settings* screen

The *Subject line* section has similar options to specify the content of the main body of the alert, and the *Notes line* section has the same for the notes section at the bottom of the alert.

Figure 19-59 shows the *Desktop Alert Settings* screen where it has been configured to state that the *Work in Progress* valuation is getting too high. If the operator clicks on either the heading or subject line, the *WIP at a Glance* query (***IMPATW***) will be run to show the detail values.

Flow Graph

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function FlowGraph_OnClicked()
6  dim Popup
7  Popup = Popup & "<Popup Duration='05' Animation='Fade'>"
8  Popup = Popup & "<Heading Text='WIP Valuation High' Icon='000'  SYSPROProgram='IMPATW'
9  Popup = Popup & "<Subject Text='The WIP Valuation has exceeded the predetermined limit
10 Popup = Popup & "</Popup>"
11 Flowgraph.CodeObject.DesktopAlert = Popup
```

Figure 19-60: VBScript code to call the *Desktop Alert* added to the *FlowGraph_OnClicked* function

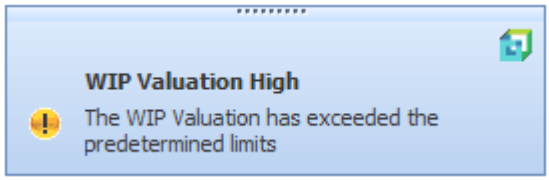


Figure 19-61: The *Desktop Alert* being displayed

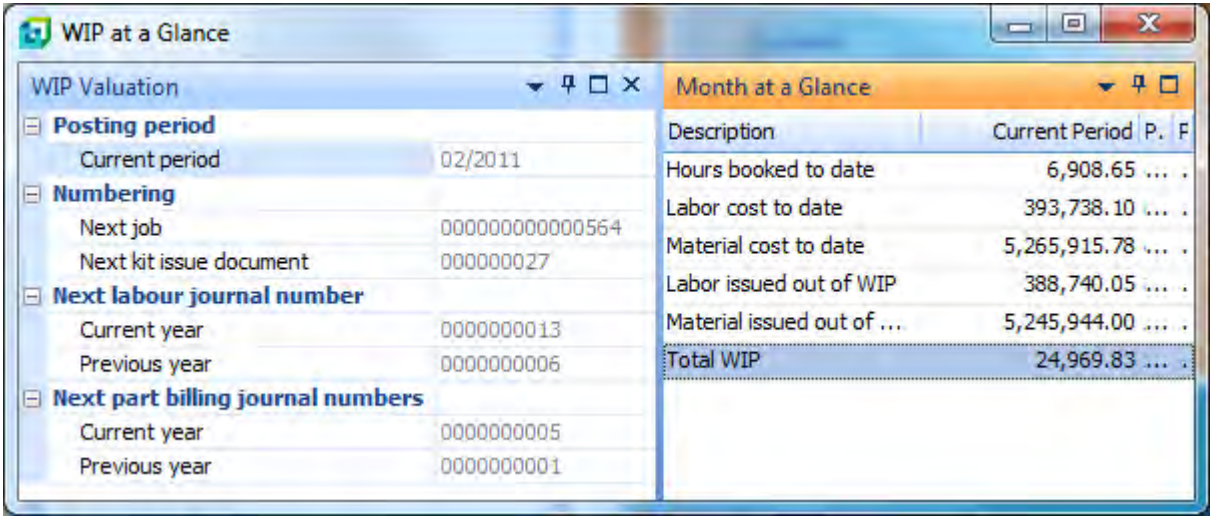


Figure 19-62: The *WIP at a Glance* query called when the operator clicked on the *Desktop Alert*

When you have finished configuring the alert, click the *Insert VBScript Code* button to add the code (see **Figure 19-60**). Note that you would still need to add some code to check the current *Work in Progress* values and compare these to predetermined values before firing the alert.

Figure 19-61 shows the *Desktop Alert* being displayed, and **Figure 19-62** shows the *WIP at a Glance* query that is displayed if the operator clicks on either the alert's title, or subject line.

FlowGraphNodes

As mentioned above, when a user clicks on a shape within a flow graph the *OnClicked* event is fired. But unlike customized panes where the event fired is specific to the item clicked, the flow graph's *OnClicked* event is for the whole flow graph.

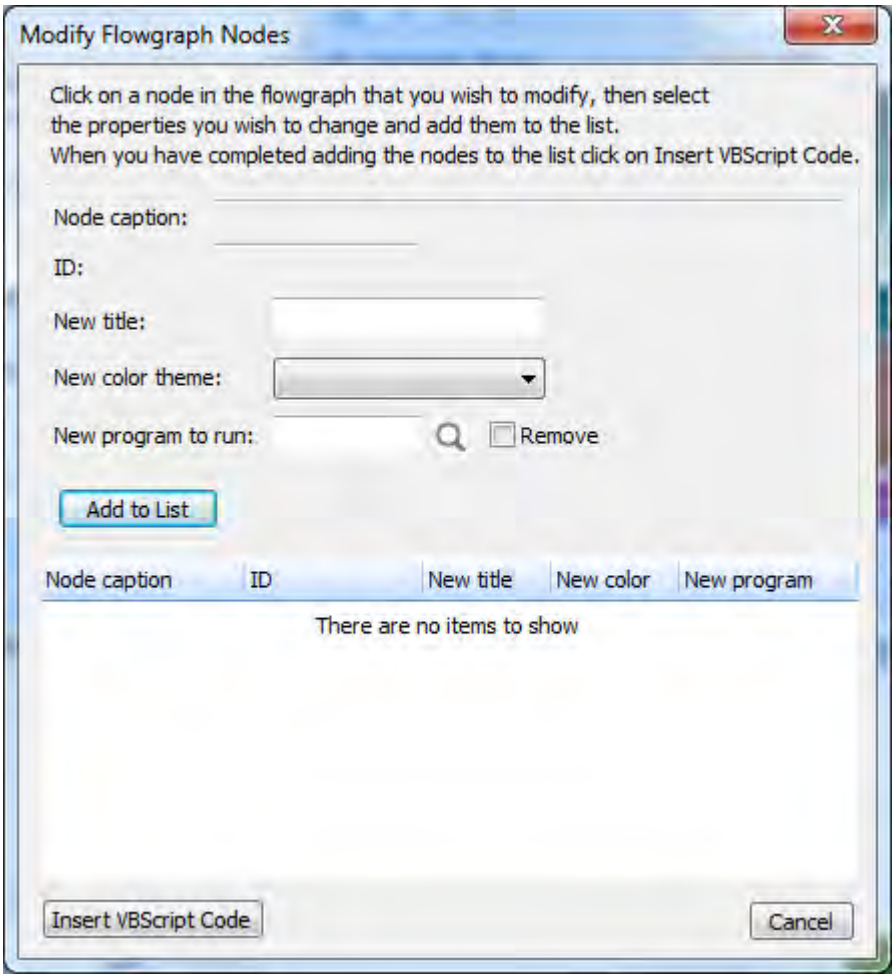


Figure 19-63: A blank *Modify Flowgraph Nodes* screen

To cater for this, each flow graph shape has a unique identifier called an *ID*. The screen that is displayed when you double-click on the *FlowGraphNodes* variable (see **Figure 19-63**) can be used to determine the *ID* of a shape, and code used to perform tasks dependent on which shape was clicked. This screen can also be used to easily build up and insert the code to change attributes of this or another shape (such as the title, color, and a program to be run).

This code can be placed against the flow graph's *OnLoad* function, in which case it will be executed as the flow graph is loaded, or against its *OnClicked* function, in which case it will be executed when the operator clicks on the selected shape on this flow graph.

Within the VBScript editing screen, when you double-click on the *FlowGraphNodes* variable name the *Modify Flowgraph Nodes* screen is displayed. The VBScript editing screen disappears from behind it, leaving the main SYSPRO screen that was open before the editor was launched, which includes the *Flow Graph* pane. **Figure 19-63** shows the *Modify Flowgraph Nodes* screen as it is initially displayed.

To determine its *ID* you must click on the shape. The shape's caption will appear against the *Node caption* prompt of the *Modify Flowgraph Nodes* screen, and its *ID* will appear against the *ID* prompt.

Figure 19-64 shows the top portion of the *Modify Flowgraph Nodes* screen where the operator has clicked on the *Inventory Query* shape. The *Node caption* has been populated with the *Inventory Query* caption, and the *ID* has been populated with the *ID* of *200198696*.

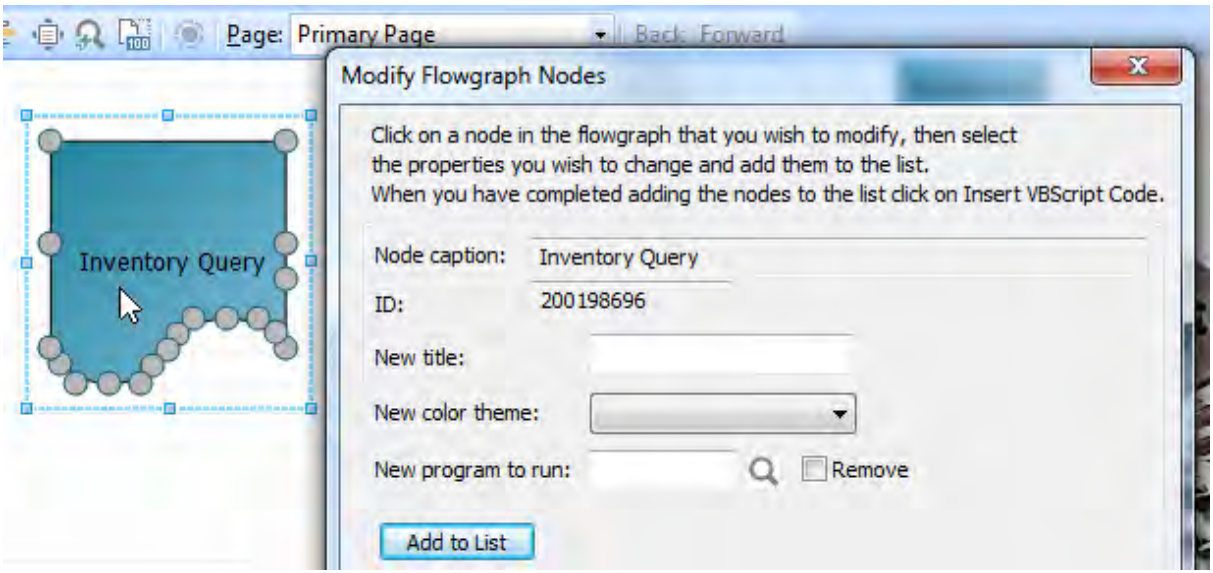


Figure 19-64: Detecting the *ID* of a shape

When a shape's caption and ID are populated on this form you can select to change its title, or color theme. This is done by supplying a caption against the *New title* prompt, and/or selecting a *New color theme* from the dropdown list. It can also be configured to call a SYSPRO program by adding the program name against the *New program to run* prompt (or selecting it using the browse).

Once the required prompts have been updated, use the *Add to List* button to add this change to the listview at the bottom of the *Modify Flowgraph Nodes* screen (see **Figure 19-65**).

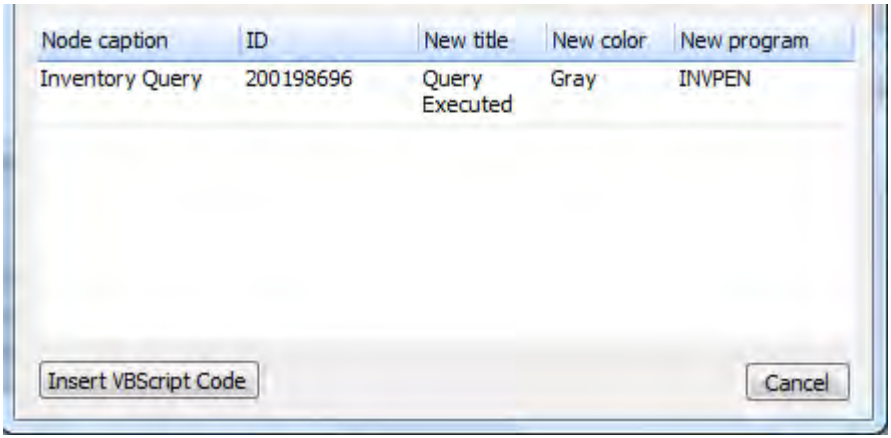


Figure 19-65: Changing a *Node Item's* attributes

Entries for multiple shapes can be made to this list. Once all the changes have been supplied and added to this listview, click on the *Insert VBScript Code* button. The *Modify Flowgraph Nodes* screen is closed, the VBScript editing screen is redisplayed, and the code to perform all of these changes is built and added to the VBScript editing screen at the point where the cursor resided before the *FlowGraphNodes* variable was selected.

Figure 19-66 shows code that was added to make changes to the attributes of two shapes. In this case the code appears over four lines. The *ampersand*, *space* and *underscore* characters at the end of the first three lines tell the VBScript engine that the line of code continues on the next line. The code is structured so that the information about each node item appears on its own line, which makes it easier to change manually at a later time if required. The *Caption* attribute is not used by SYSPRO; it is displayed so that anyone reading the script knows which shape the line refers to, without having to look up the *ID*. The new caption appears against a *Title* attribute.

As mentioned above, the VBScript is associated with the whole flow graph, as are the two available events (*OnLoad*, and *OnClicked*). If the above changes are made against the *OnLoad* function they will occur as the flow graph loads. For this operator's default flow graph this would be as SYSPRO loads. For other flow graphs this would be when the operator clicked on a shape that is configured to

transition to this flow graph, or when the operator manually selects this flow graph using the *Options* button on the toolbar.

Flow Graph

```
1  ' This script contains functions for form and field events.
2  ' You must not modify the name of the functions.
3  Option Explicit
4
5  Function FlowGraph_OnClicked()
6  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
7  "<Node Caption='Inventory Query' Id='200198696' Title='Query Executed' Color='Gray' Program='INVPEN'/>" & _
8  "<Node Caption='Updated' Id='172770018' Title='Transactions Posted' Program='IMP010'/>" & _
9  "</FlowgraphNodes>"
10 End Function
```

Figure 19-66: The code added to change the attributes against two *Shapes*

If the operator moves to another flow graph, and then back to this one, this flow graph's *OnLoad* event will fire again. Note that if a program name has been configured against one of the shapes using the *FlowGraphNodes* variable, and the code to perform this appears against the *OnLoad* function, the program will not be run, although any attribute changes for the same shape item will occur.

If the above changes are made against the flow graph's *OnClicked* function, the attribute changes will occur when the operator clicks on *any* shape (unless code has been added to check the specific *ID* of the shape that was clicked, and restrict the change to when a specific shape was clicked).

If more than one task appears against a shape, the sequence of processing is any normal VBScript code is processed first, followed by any changes to attributes, a transition to another page or flow graph, and finally calling the specified program/application/report.

If the caption or color of a shape has been changed during this run of SYSPRO because an operator clicked on a shape, when they exit SYSPRO and call it up again, the original settings will return. An exception to this is if the operator enters *Design Mode*. Double-clicking on any of the shapes in *Design Mode* will perform the same task as would be performed by a single-click on this shape outside of *Design Mode*. So if the operator double-clicked on a shape the same task would be performed, and if this included changing the caption or color of a shape, when they exited *Design Mode* it would be as if they had deliberately made this change. So, when they exit *Design Mode* this will be the new default setting.

If you have a program associated with a shape, you may decide that you don't want this program to run when this shape is clicked under a specific circumstance, such as when a global variable contains a specific value. The *FlowGraphNodes* wizard can be used to assist with this. The *New program to run* prompt must be set to six dashes/hyphens (------) to represent that no program should be run. This can be entered manually, or you can check the *Remove* checkbox and this will be done for you. **Figure 19-67** shows the *Remove* option being checked, which puts six hyphens against *New Program to run.*

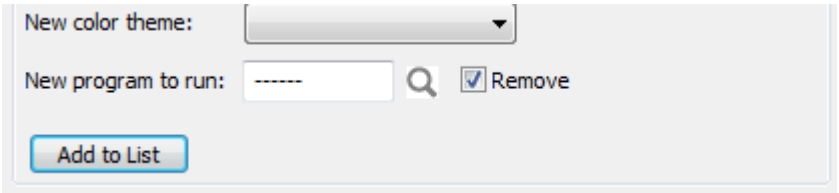


Figure 19-67: Checking *Remove* uses six hyphen characters for the program name

Alternatively, if you already have code that contains a *Program* attribute, this can be replaced with six hyphens, as appears in the code sample below.

```
If SystemVariables.CodeObject.GlobalVariable1 = "Y" then
  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
  "<Node Caption='Query Executed' Id='200198696' Program='------'/>" & _
  "</FlowgraphNodes>"
End If
```

If multiple *FlowGraphNodes* entries exist within this function, only the contents of the last one to be processed will be executed. Each new entry that is read as the VBScript progresses will overwrite the previous one in memory.

If you want to have multiple *FlowGraphNodes* entries in the same function you must ensure that only the one required *FlowGraphNodes* entry is executed, and that all the others are bypassed. Typically this is done using the *NodeClickedID* variable and an *IF* statement. See the *NodeClickedID* section below for more information and an example.

NodeClickedCaption

The *NodeClickedCaption* variable is used to return the current caption of the shape that was clicked. This could be used within a message back to the operator to confirm that a specific task must be performed, used to update a global variable, or even used to write out to a custom log file.

In theory, it could be used within an *IF* statement to perform a task when a specific shape is clicked, but as the shape's caption can be changed, this would not be 100% reliable. For example, in the code below the check is made to see if the caption of the clicked node item is *Item 1*, and if it is, the color of the node item with the *ID* of *087605296* is changed to *Cappuccino*. If the node item's caption changed from *Item 1* to anything else, this code would never be executed. This check is also case-sensitive, so *Item 1* is different from *item 1*.

```
If FlowGraph.CodeObject.NodeClickedCaption = "Item 1" then
  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
  "<Node Caption='Item 1' Id='087605296' Color='Cappucino'/>" & _
  "</FlowgraphNodes>"
End If
```

NodeClickedID

The *NodeClickedID* variable is used to return the *ID* of the shape that was clicked. Typically this is used to determine which shape was clicked, or to ensure that a task is only performed when a specific shape was clicked.

The code in the example below appears against the flow graph's *OnClicked* function. This code is executed whenever the operator clicks on any of the shapes. To know which of the three shapes was selected (and needs to have its color changed) there are three *IF* statements that check the shape *ID*s against the *NodeClickedID*. This is a much more reliable method than using the *NodeClickedCaption* variable, as a node item's *ID* cannot be changed programmatically, or by the operator. Other lines of code can be inserted within the *IF* statement to perform other tasks, such as calling an external program, calling a web service, or interrogating a database. Note that the *Add New Shortcut Wizard* can be used to associate a SYSPRO program, customized report, Net Express program, etc., with this node item. If the launching of the program/report/application were configured this way you would not need to perform this check, as this shortcut would be associated with the specific node item, not with the flow graph's *OnClicked* function.

```
If FlowGraph.CodeObject.NodeClickedID = 087605296 then
  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
  "<Node Caption='Item 1' Id='087605296' Color='Cappucino'/>" & _
  "</FlowgraphNodes>"
End If

If FlowGraph.CodeObject.NodeClickedID = 190135624 then
  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
  "<Node Caption='Not so new' Id='190135624' Color='Cappucino'/>" & _
  "</FlowgraphNodes>"
End If

If FlowGraph.CodeObject.NodeClickedID = 182773928 then
  FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
  "<Node Caption='New Item 4' Id='182773928' Color='Cappucino'/>" & _
  "</FlowgraphNodes>"
End If
```

When there are many shapes it may be more efficient to use a *Case Select* statement instead of multiple *IF* statements. Below is a *Case Select* example that performs the same logic as code above.

```
Select Case FlowGraph.CodeObject.NodeClickedID
  Case 087605296
    FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
    "<Node Caption='Item 1' Id='087605296' Color='Cappucino'/>" & _
    "</FlowgraphNodes>"

  Case 190135624
    FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
    "<Node Caption='Item 1' Id='190135624' Color='Cappucino'/>" & _
    "</FlowgraphNodes>"

  Case 182773928
    FlowGraph.CodeObject.FlowgraphNodes = "<FlowgraphNodes>" & _
    "<Node Caption='New Item 4' Id='182773928' Color='Cappucino'/>" & _
    "</FlowgraphNodes>"

  Case Else
End Select
```

Apply XAML Markup Option

The shapes that appear within the different sections of the *Shapes* pane when in *Design Mode* are rendered using XAML. These can be changed. The *Apply XAML Markup* option (in *Design Mode*, right-click on a shape on the flow graph desktop | *Properties* | *Apply XAML Markup*) loads the properties of the currently selected shape into a XAML editor. This enables you to manipulate the shape beyond the capabilities of the *Shape Properties* pane.

The XAML editor contains two panes, the *Preview* pane and the *Markup* pane. When the editor is first loaded, the XAML code to create the shape is populated in the *Markup* pane and this is rendered as a shape within the *Preview* pane. Changes made to the markup code are reflected immediately in the preview.

Figure 19-68 shows the *XAML Markup Editor*. The standard *Rectangle* shape had been dragged onto the flow graph desktop, highlighted, and the *Apply XAML Markup* option selected. Within the markup code the *nodeCanvas width* was increased to 200, the *nodePath width* was increased to 200, and the *nodePath StrokeThickness* was increased to 10. These combined changes are reflected in the *Preview* pane.

Clicking on the *Save* button applies these changes to the currently selected shape. The results can be seen in **Figure 10-69**.

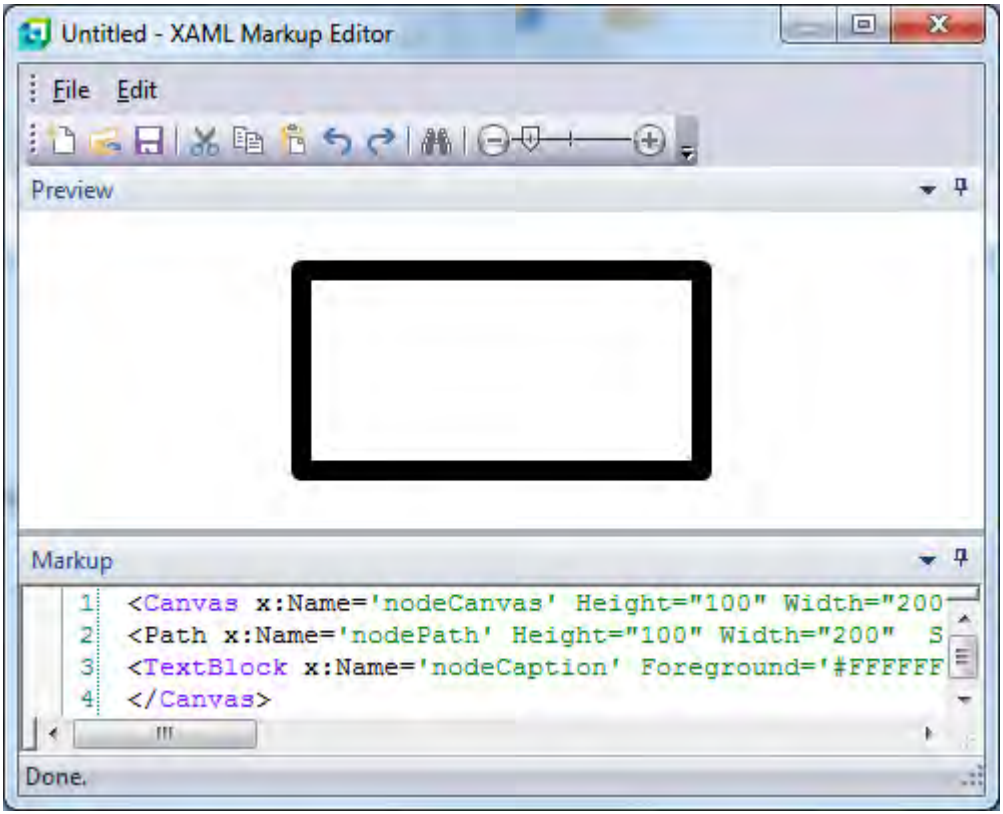


Figure 19-68: The *XAML Markup Editor* showing the markup code and rendered shape

Figure 19-69: The newly-created XAML shape

While modifying the XAML in the *XAML Markup Editor*, the bottom pane contains the markup code and the top pane renders it. If the entered markup is not valid (so can't be rendered) the top pane will change to display the markup code instead. The footer section of the screen (directly below the *Markup* pane) will display an error message highlighting the problem, and its location.

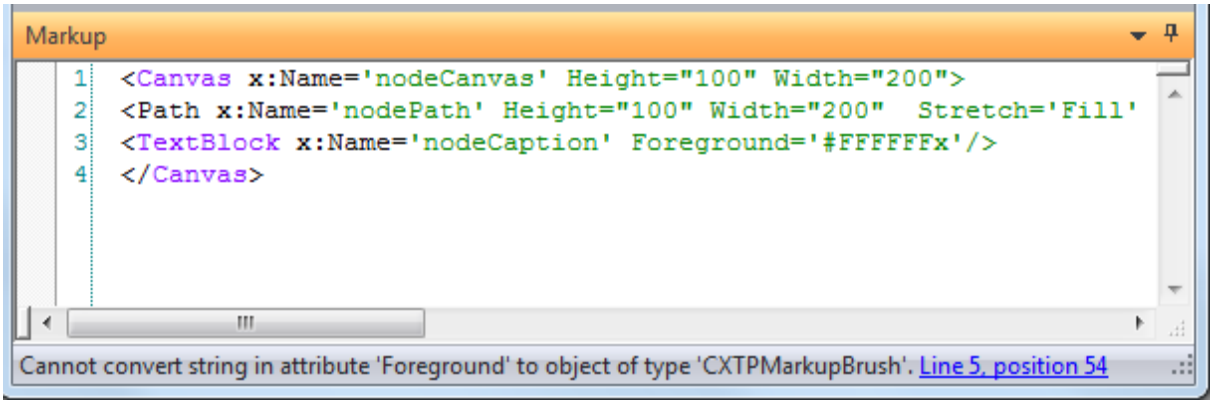


Figure 19-70: Invalid markup code causing an error message to appear in the footer

Figure 19-70 shows a section of markup code containing an error. The footer section of the screen contains the information about the error.

The intricacies of using XAML are beyond the scope of this book. However, using a search engine and the string *learn XAML* returned plenty of useful results to get started.

Flow Graphs and Roles

When logging in as a role-based operator, and flow graphs are selected to be displayed, the *Flow Graph* section consists of two panes. The first is the pane displaying the flow graph, and to its left the *Flow Graphs for Role* pane. The *Flow Graphs for Role* pane contains a list of flow graphs that have been created for this role.

Figure 19-71 shows the *Flow Graph* section with the *Flow Graphs for Role* pane on the left. This contains a list of three flow graphs that have been created for this role. The *Reconcile Cash Book* flow graph name has been selected from these, and its flow graph is displayed in the pane on the right. If no flow graphs had been created for this role, the *Flow graphs for role (018)* header would be the only entry within this pane.

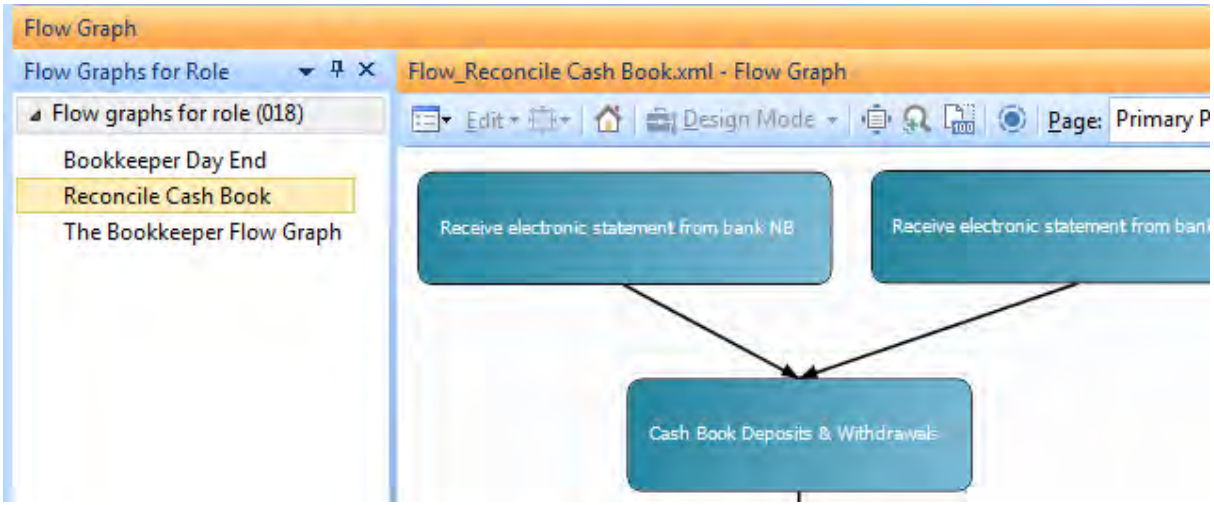


Figure 19-71: *Flow Graphs for Roles* pane

When SYSPRO is initially loaded for an operator their personal flow graph (*Home Flow Graph*) will be displayed. If they do not have a *Home Flow Graph* configured, the flow graph desktop will be blank. If they are a member of a role, the *Flow Graphs for Role* pane will contain a list of flow graphs configured for the currently selected role. Clicking on one of the flow graph names in this list will display the flow graph in the right hand pane.

If this operator is a member of multiple roles, the operator can change roles by clicking on the role name in the bottom right hand corner of the main SYSPRO screen, within the footer of the main SYSPRO screen. The *Switch Role* screen will be displayed which will contain a dropdown list of roles associated with their operator code. If the operator selects a role from this list the content of the *Flow Graphs for Role* pane will change to reflect the flow graphs associated with the newly-selected role.

Design Flow Graphs for Roles

The *Design Flow Graphs* button appears on the *Administration* tab of the SYSPRO *Ribbon Bar*. It is a means of creating, updating, copying, and deleting flow graphs against roles. The button is disabled unless both the operator activities *Flow Graph – Allowed to design flow graphs* and *Main Menu – Allow to design flow graphs by role* are checked (*Ribbon bar* | *Setup* tab | *Operators* | select operator from list | *Change* button | *Security* tab | *Activities*). If either of these two options is unchecked the *Design Flow Graphs* button will be disabled.

When the *Design Flow Graph* button is clicked the *Design Flow Graphs for Roles* screen is loaded, consisting of a toolbar and two panes. The *Flow Graph List* pane contains a list of roles that have a

flow graph associated with them, and the descriptions of these flow graphs appears alongside them. The *Preview* pane will contain a preview of whichever flow graph is highlighted in the *Flow Graph List* pane (see **Figure 19-72**).

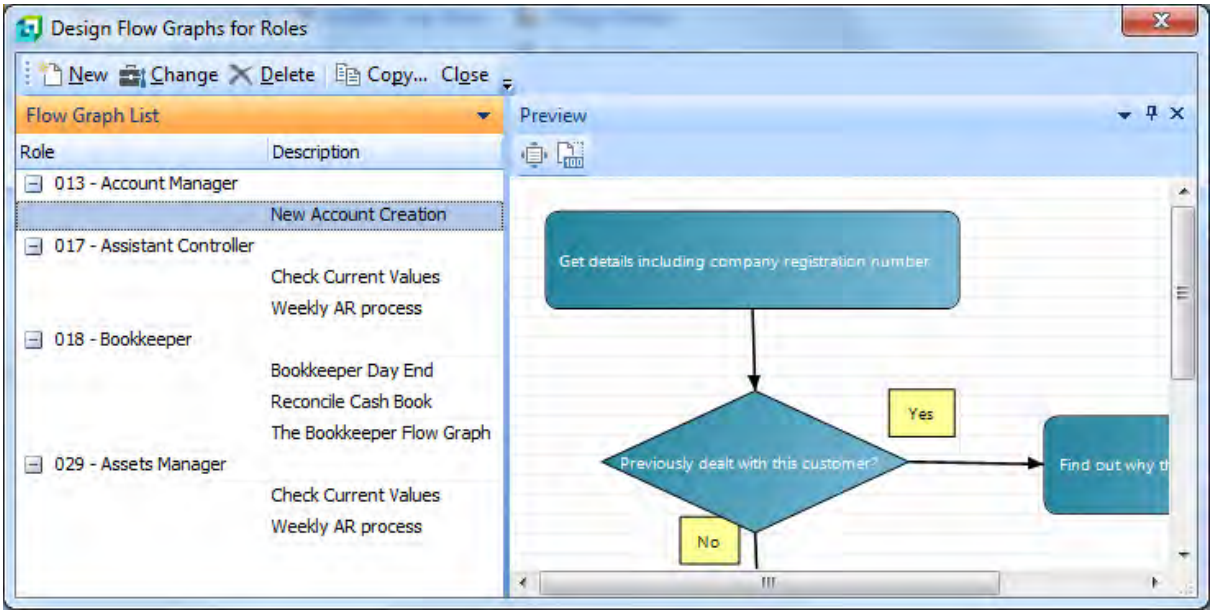


Figure 19-72: The *Design Flow Graphs for Roles* screen

New Flow Graph for Role

The *New* button on the toolbar is used to add flow graphs to roles. The *Flow Graph Maintenance* screen is displayed and this contains a dropdown list of all roles that have been configured. Select the role required and add a description for this flow graph (see **Figure 19-73**).

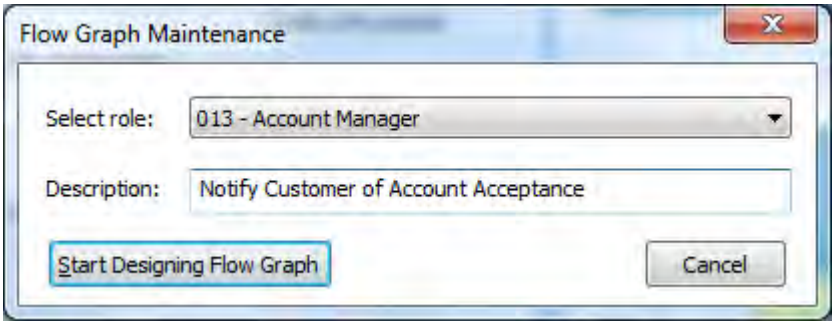


Figure 19-73: The *Flow Graph Maintenance* screen just before designing the flow graph

Clicking on the *Start Designing Flow Graph* button causes the *Flow Graph Maintenance* screen to be removed and the flow graph screen is put into *Design Mode*. Design the flow graph as normal, and when you exit *Design Mode* the displayed flow graph remains as the current flow graph, and this entry is added to the *Flow Graph List* pane of the *Design Flow Graphs for Roles* screen.

Change Flow Graph for Role

If one of the flow graph descriptions is highlighted in the *Flow Graph List* pane of the *Design Flow Graphs for Roles* screen, the *Change* button becomes enabled. Clicking on this button displays the *Flow Graph Maintenance* screen. The role cannot be changed but the description can. Clicking on the *Start Designing Flow Graph* button displays the flow graph in the *Flow Graph* section and puts it into *Design Mode*. After making the required changes to the flow graph and exiting *Design Mode*, the flow graph is saved, and if the description was changed it will be updated in the *Flow Graph List* pane.

Delete Flow Graph for Role

If one of the flow graph descriptions is highlighted in the *Flow Graph List* pane of the *Design Flow Graphs for Roles* screen, the *Delete* button becomes enabled. Clicking on this button will prompt you first, and then delete the flow graph from the list.

Copy Flow Graph for Role

If one of the flow graph descriptions is highlighted in the *Flow Graph List pane* of the *Design Flow Graphs for Roles* screen, the *Copy* button becomes enabled. Clicking on the *Copy* button displays the *Copy Flow Graphs* screen that contains a list of all the roles (see **Figure 19-74**).

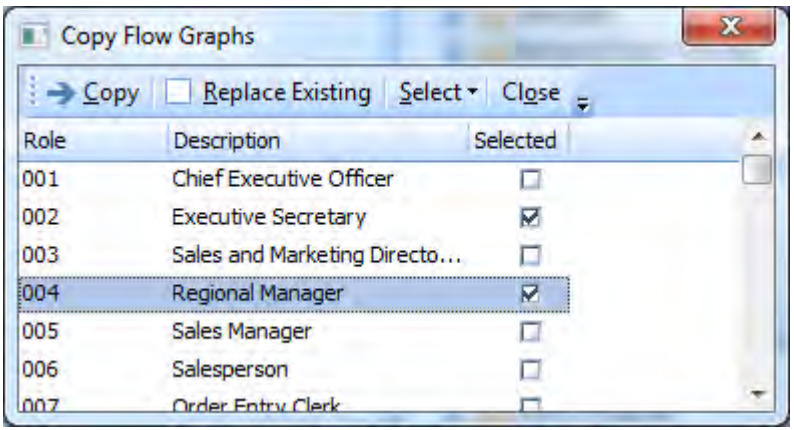


Figure 19-74: The *Copy Flow Graphs* screen where the roles are selected

Check the roles to which this flow graph should be copied and click the *Copy* button. The flow graphs will be copied to all of the roles that were checked that do not already have a flow graph with this description. If this flow graph needs to be copied to a role that already contains a flow graph with the

same description, the *Replace Existing* checkbox must be checked on the toolbar before clicking on the *Copy* button of the *Copy Flow Graphs* screen. The flow graph will be copied to the selected roles regardless of whether they already have a flow graph with this name.

The *Select* button's dropdown on the toolbar enables you to select all, or unselect all roles.

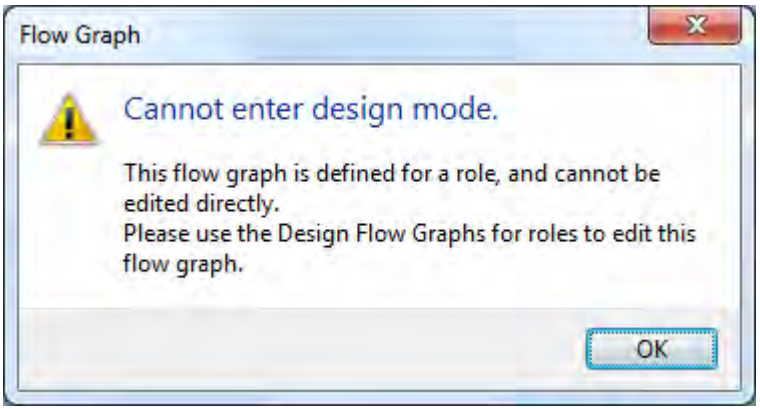


Figure 19-75: Attempting to enter normal *Design Mode* for a role-based flow graph

Editing Role-based Flow Graphs

Role-based flow graphs have to be edited using the *Design Flow Graphs for Roles* screen. If you attempt to edit a role-based flow graph as if it were a normal flow graph, you will receive an error message as you attempt to enter *Design Mode* (see **Figure 19-75**).

Deploying the Flow Graph

Unlike most other ways of using VBScript within SYSPRO (forms, listviews, etc.), the VBScript that is created for a flow graph is saved in the flow graph XML file itself. This means that once you have created your flow graph and added your VBScript, you can email the flow graph XML file to another person or organization and it should work correctly; there are no extra files that need to be deployed.

When you add a *Picture* shape to a flow graph that may need to be distributed to another system it is worth considering where the image is located. The absolute location of the image is stored against the shape in the flow graph (see **Figure 19-76**). This means that when you export a flow graph from one system to another, if the image location is different (maybe because the SYSPRO parent folder is different) the image will not found and will not appear on the flow graph.

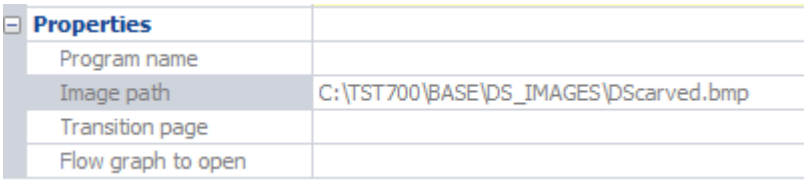


Figure 19-76: The absolute address of the image associated with a *Picture* shape

If the image to be used is located in the SYSPRO `Base\Samples` folder, a relative address is used instead of an absolute path. The relative address will contain *{samples}* followed by the name of the image (see **Figure 19-77**). Using the browse to select the image from the `Base\Samples` folder causes the relative pathname to be inserted automatically.

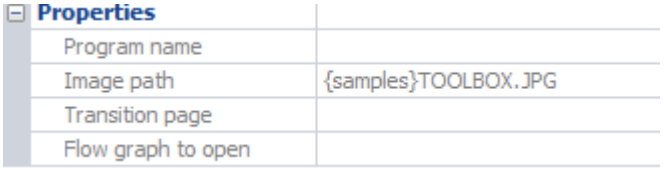


Figure 19-77: The relative path name for the image associated with the *Picture* shape

Calling a Flow Graph from a Navigation Pane Tile

Within the *Navigation Pane* it is possible to add a tile by clicking on the *Create a new personal menu in the Navigation Pane* button on the toolbar. When this button is selected the *Menu Information* screen is displayed (see **Figure 19-78**).

Supply a caption and select the *Task panel* radio button. When you click the *OK* button a *Task panel* is created for you. This is like a desktop to contain tiles within the *Navigation Pane*.

Right-click on the *Task panel*, and select *New* | *Blank Tile* from the displayed menu. Supply a title for the tile and press the *Enter* key (see **Figure 19-79**). The tile is added to the *Task panel* (see **Figure 19-80**).

Right-click on the tile and select *Properties* | *Action* from the menu, the *Action Settings for Tile* screen is displayed. Check the *Open Flow Graph file* option and use the browse to locate the required flow graph (see **Figure 19-81**). Click on the OK button to update the tile.

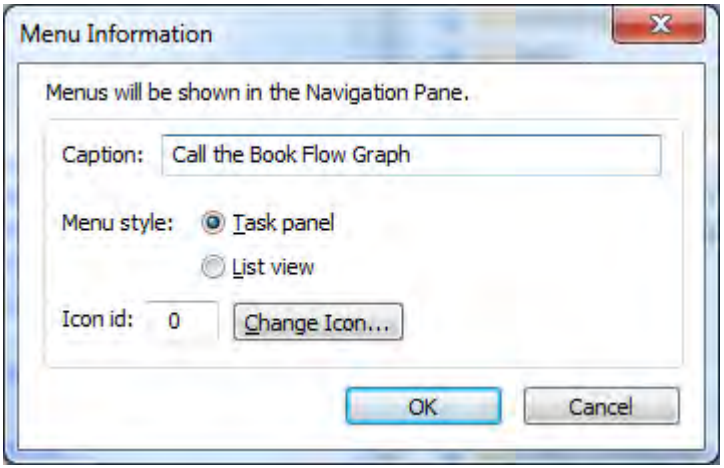


Figure 19-78: Creating the *Task panel*

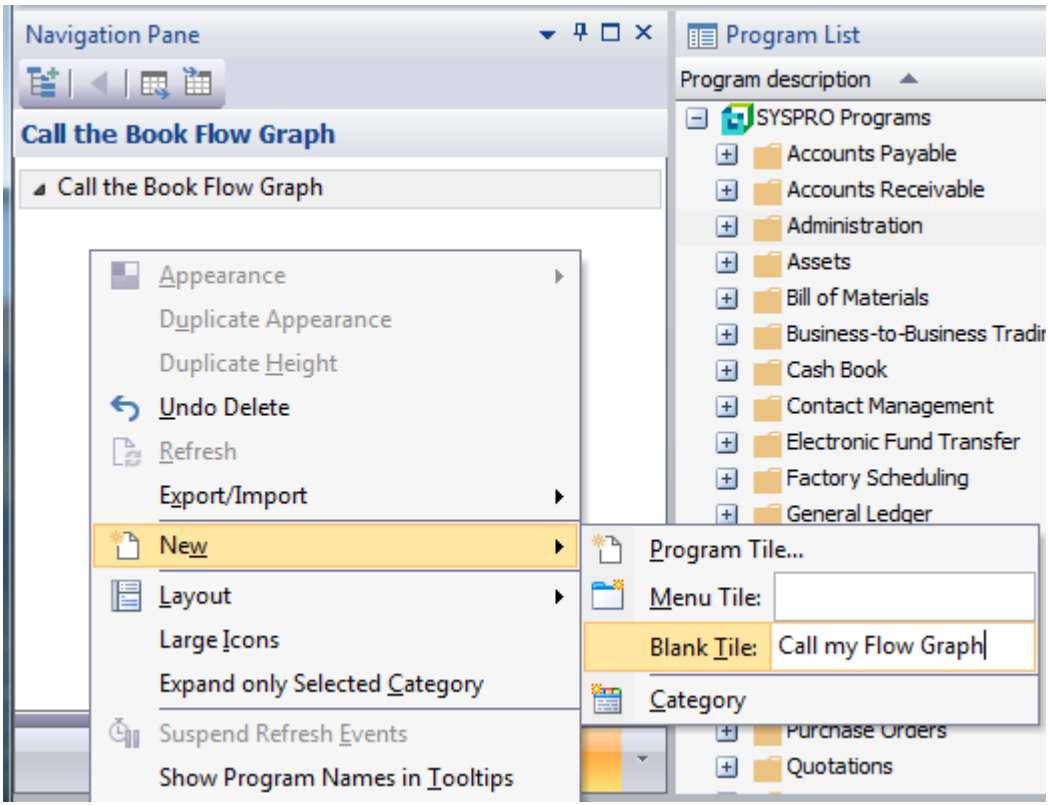


Figure 19-79: Adding a *Tile* to the *Task panel*

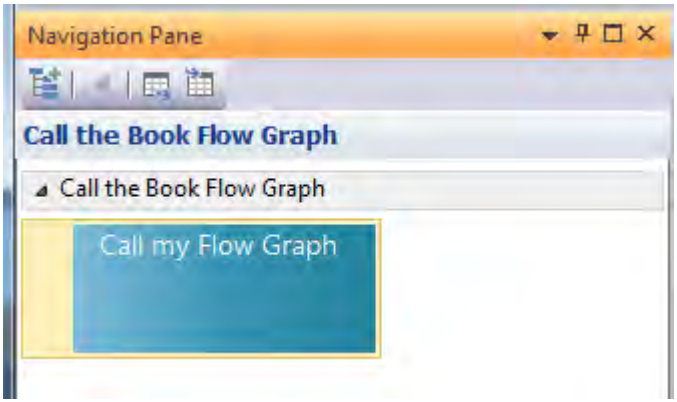


Figure 19-80: The *Tile* on the *Task panel*

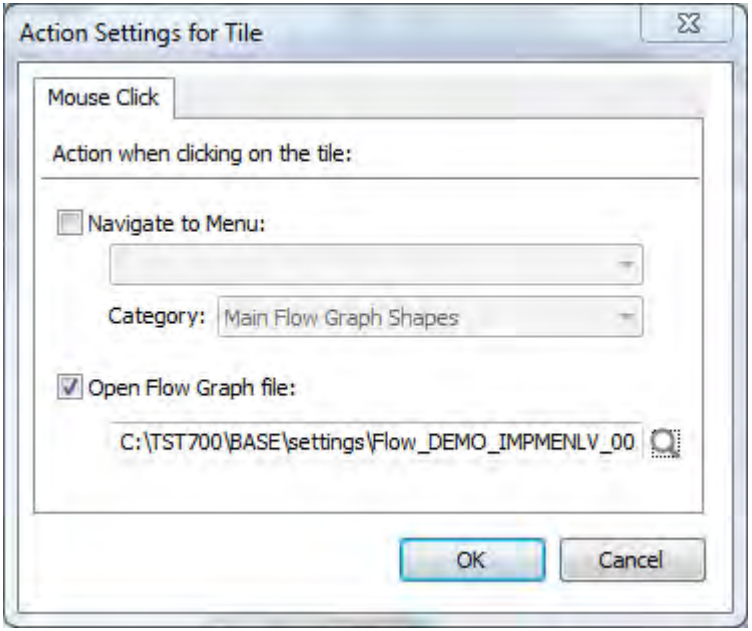


Figure 19-81: The *Action Settings for Tile* screen

When the tile is selected the flow graph associated with this tile replaces the currently displayed flow graph. However, this flow graph is opened in a *Protected View*. The flow graph will work as designed, but the *Protected View* prevents the operator from editing the flow graph accidentally. They must click on the *Enable Editing* button before that can enter *Design Mode* (see **Figure 19-82**).

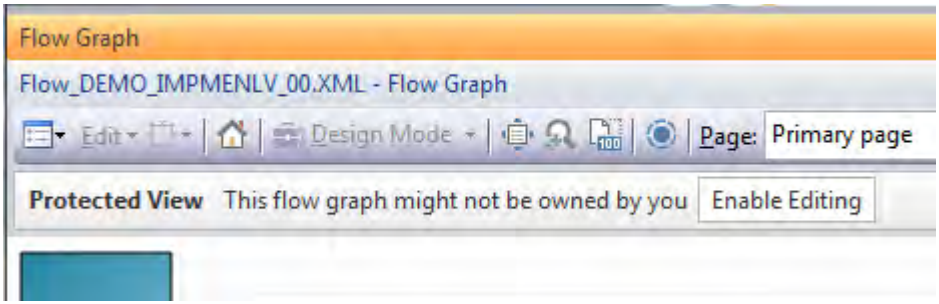


Figure 19-82: The *Flow Graph* that was called from the *Navigation Pane* tile

If the *Flow Graph* pane is not currently visible when the tile associated with a flow graph is clicked, the *Flow Graph* pane will become visible.

Chapter 20 - Associated Panes

An associated pane is a pre-written customized pane template that is specifically associated with a SYSPRO key field such as stock code, customer, requisition, etc. The associated pane can be a listview, a form, a graph, or a notepad. Some key fields have a small number of associated panes linked to them, whereas others (such as *Stock Code*) have many.

An associated pane differs from a customized pane in that when the value of the field with which it is linked changes, the associated pane automatically refreshes to match; you don't need to add any code to make this happen. A customized pane requires code to be added (to another pane, field, etc.) to cause its OnRefresh event to be fired.

Adding Associated Panes

Associated panes are added by right-clicking on a key field within a form, selecting *Insert Associated Pane*, and choosing from the available list. **Figure 20-1** shows the associated panes that are linked to the *Sales Order* key field. The icon that appears in front of the name of the associated pane indicates what type of pane it is. The icon in front of the *Dispatch Notes* associated pane shows that this is based on a *Listview* customized pane.

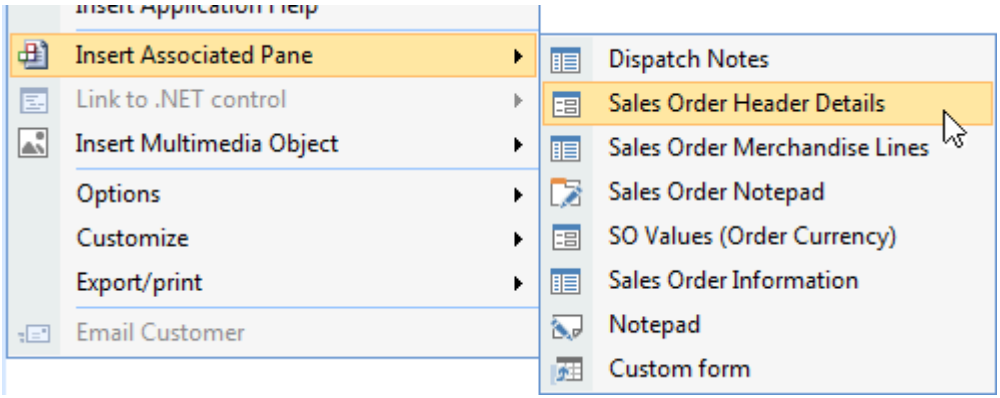


Figure 20-1: The associated panes linked to the *Sales Order* key field

The icon in front of the *Sales Order Header Details* associated pane shows that this is based on a *Form* customized pane. The one in front of the *Sales Order Notepad* shows that this is a *Rich text notepad* customized pane. The one in front of the *Custom form* associated pane shows that this is a special *Form* customized pane that will display the contents of custom form fields associated with this key field. In **Figure 20-2** the icon in front of the *Sales History Quantities* associated pane shows that this is a graph. The icon in front of the *Sales History Qty (Trendline)* associated pane shows that this is a listview that contains a *Trendline* (a *Trendline* is a mini line graph that appears in one column of the listview and is drawn using XAML).

When an associated pane is added to a program, if the field to which it is linked is already populated, the associated pane will automatically be populated on creation. If the key field is not populated the associated pane will be created, but not populated.

If an associated pane has already been linked to the selected key field on this form the associated pane name is greyed-out, as can be seen with the *Requisitions* associated pane in **Figure 20-2**.

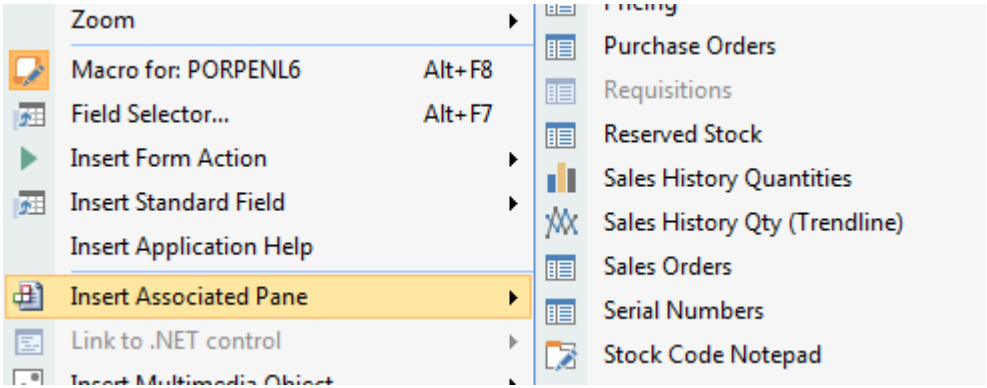


Figure 20-2: Greyed-out associated pane name

If the key field appears on more than one form within the program, the same associated pane name can be linked to each of these instances, in which case there will be a separate associated pane for each of these key fields. Typically a key field name will appear on only one form within the program, but if a customized pane form contains a field with the same name as a SYSPRO key field, the ability to add an associated pane to this field will be available.

An associated pane can only be added if the operator has the operator activity *Forms – Customization by operator* checked (*SYSPRO Ribbon Bar* | *Setup* tab | *Security* section | *Operators* | highlight the operator | *Change* button | *Security* tab). **Figure 20-3** shows the *Forms – Customization by operator* option checked.

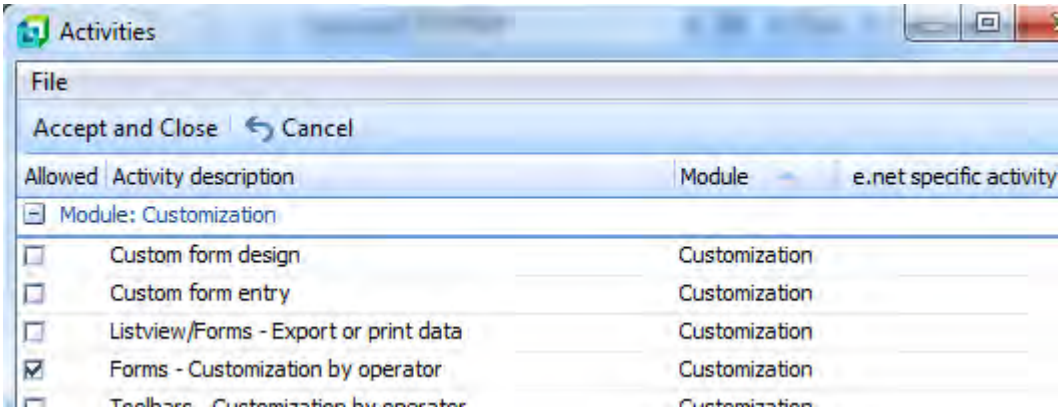


Figure 20-3: The operator activity *Forms – Customization by operator*

Deleting an Associated Pane

Associated panes that have been added to a program can be removed by clicking on the *Close* button in the top right corner of the pane (see **Figure 20-4**). The pane is removed from the program, and the associated pane name will no longer be greyed-out when you right-click on the key field and select *Insert Associated Pane*.

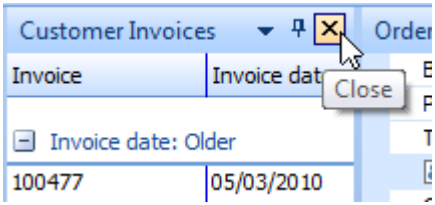


Figure 20-4: Deleting an associated pane

When does an Associated Pane get Refreshed

To reduce the amount of processing required, an associated pane will only refresh/re-populate when the value of the key field with which it is associated is changed. If the other values on the form change, but the key field's value does not, the associated pane will not refresh. This feature can save a significant amount of time, in cases where the associated pane contains a significant amount of information.

An example of this appears in **Figure 20-5**, which comes from *the Purchase Order Query* program. Within the *Detail Line* section there is a listview, and a form called *P/O Line Information*. An associated pane called *List of Requisitions* has been added by right-clicking on the *Stock code* field

within the *P/O Line Information* form, selecting *Insert Associated Pane* and choosing the *Requisitions* associated pane. As the operator highlights a row within the listview, the content of the *P/O Line Information* form is changed. If the *Stock code* value changes within this form, the *List of Requisitions* listview is refreshed to reflect the new stock code.

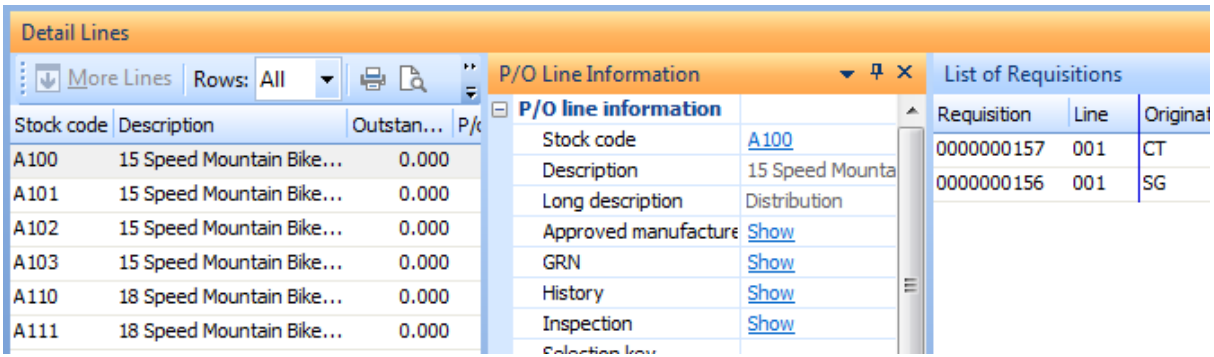


Figure 20-5: *List of Requisitions* associated pane is refreshed when the *Stock code* value changes

In **Figure 20-6** the three lines in the *Detail Lines* listview all contain the stock code *A200*. If the operator selects a different line the *P/O Line Information* form is refreshed with the new values for this line. However, as the value against the *Stock code* field on this form does not change, the *List of Requisitions* associated pane does not refresh.

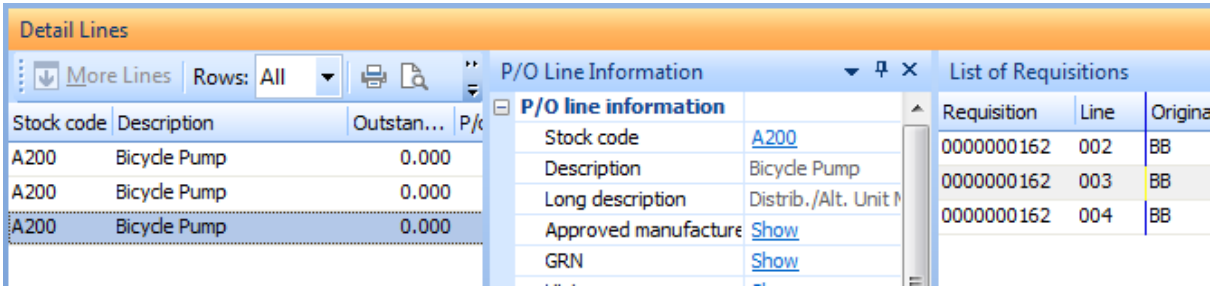


Figure 20-6: The associated pane will not be refreshed if the key field value does not change

As an associated pane is just a customized pane it can be repositioned elsewhere within the same program, and even float above the program. If the associated pane is minimized (so that it becomes a tab somewhere around the outside of the program) it will still be refreshed/populated when the value with which it is associated changes. So if the pane is expanded, it will contain the latest information.

When the value of the key field changes, the value is passed to the associated pane using the *RefreshValue* variable. The following is an extract from the *OnRefresh* function of the *List of*

Requisitions associated pane. Note that the key field value is passed in the *RefreshValue* variable, and stored in a variable called *Stockcode*.

```
Function CustomizedPane_OnRefresh()
  ' Specify the variables to be used when calling the business object
  dim XMLOut, XMLParam, Stockcode
  CustomizedPane.CodeObject.ListviewData = " "

  ' Populate the Stockcode variable with the refresh value
  Stockcode = CustomizedPane.CodeObject.RefreshValue

  if Stockcode = "" then
    exit function
  end if
```

Editing an Associated Pane

As an associated pane is a customized pane, once it has been added to the program it can be modified in the same way as any other customized pane (*Menu* button | *Customized Pane* | *Properties*). If you modify an associated pane this way, and then delete it, the changes will be lost. Adding the same associated pane again will create a new one containing the standard code.

IMPVBC.IMP and CUSVBC.IMP Files

A list of available associated panes, and the key fields to which they are associated, is held in a file called `IMPVBC.IMP` that resides in the SYSPRO `Programs` folder. This is a text file and contains the key field name, description, name of the script containing the template, the number of the icon to be displayed, and a unique three digit number. The rows are grouped by key field, and there are two special sections at the end (*notepad* and *customform*). Do not change this text file, as it will be overwritten when a later version of SYSPRO is installed, or if it is included in a patch.

If you wish to add your own associated panes (which is usually performed by taking an existing one and editing it) you can add an entry for this to a file called `CUSVBC.IMP` which is not shipped with SYSPRO, so will not be overwritten. If the file does not already exist (because you have not added custom associated panes before), the structure of the `CUSVBC.IMP` file is identical to the `IMPVBC.IMP` file, so it is easiest to make a copy of this file and edit it.

To be used by SYSPRO, the `CUSVBC.IMP` file must exist in your custom program folder. The location for this is specified against the *CUSPRG=* entry in the `IMPACT.INI` in your SYSPRO `Work` folder. This entry can also be seen/modified from within SYSPRO using the *System Setup* program (*SYSPRO Ribbon Bar* | *Setup* tab | *General Setup* dropdown list | *System Setup* | *Folders* tab)

The following is an extract from an IMPACT.INI file showing the location of its custom program folder.

```
[Custom Directories]
CUSPRG=C:\TST700\CUSTOM
CUSGUI=
```

When a custom associated pane is added using the CUSVBC.IMP file, the associated pane appears at the end of the list of normal associated panes, but before the special associated panes if they are relevant for this key field. **Figure 20-7** shows the list of associated panes for the *Stock Code* key field. The CUSVBC.IMP file contains an entry called *Food Grow Job Review*, and this appears at the end of the list of normal associated panes, but before both the *Notepad* and *Custom form* special associated panes.

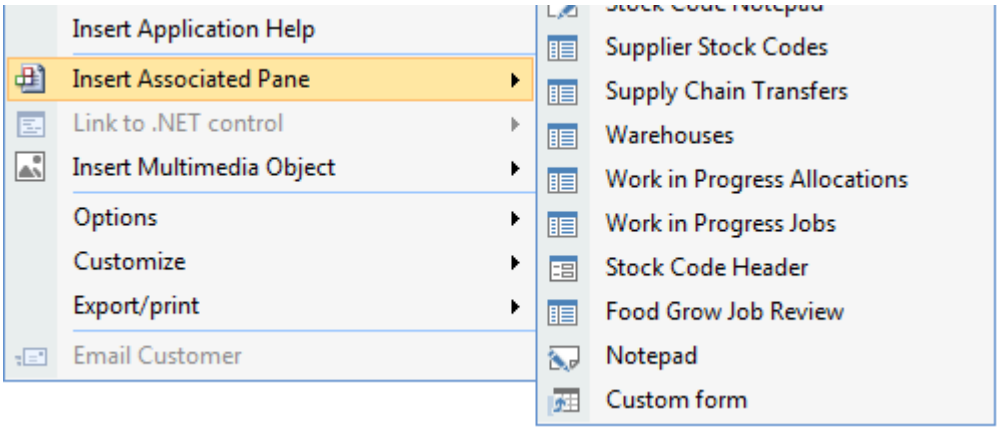


Figure 20-7: The *Food Grow Job Review* custom associated pane

The custom associated panes are added/removed in the same way as normal associated panes (right-click on the key field name | *Insert Associated Pane* | select the associated pane from the list). If the custom associated pane has already been added, the custom associated pane name will appear greyed-out in the same way as other associated panes.

Associated Pane Location and Naming Convention

The standard associated panes that ship with SYSPRO are held within the SYSPRO Base\Samples folder. These should never be modified as they may be overwritten at any time that a later version of SYSPRO (or a patch) is installed. If you want to use a modified version of one of these you should make a copy and modify the copy. An entry should be added to the CUSVBC.IMP file so that the new file appears in the list of associated panes when the operator selects to insert one (see the

IMPVBC.IMP *and* CUSVBC.IMP *Files* section above). The custom associated panes reside in the same location as the standard ones.

The naming convention for the standard associated panes is the text *Form*, followed by the key field name, followed by a descriptive name for the script, followed by the suffix of .txt. In total, the whole name including the suffix must be 30 characters or less. For example, the name of the script for the *Alternate Suppliers* linked to the *Stock Code* key field is Form_StockCode_AltSupplier.txt, which is exactly 30 characters. It is recommended that you use the same naming convention for your custom associated panes.

Custom Form Associated Pane

One of the two special associated panes appears in the list as *Custom form* and is used to dynamically build and populate a form containing the custom form fields associated with this key field. The entries for the custom form associated panes appear at the end of the IMPVBC.IMP file, under the *[customform]* section name. The entries within this section contain three fields, *Caption*, *Custom code*, and the *ID* of 901. The description and icon information are not required because the description is always *Custom form*, and the icon is always the same.

The *Caption* is the name of the key field with which the pane will be associated. The *Custom code* is the internal name that SYSPRO uses for the custom form fields for this key field, and is passed to the associated pane in the *CustomFormType* variable. The list of custom form fields used by SYSPRO appears in the IMPCFM.IMP file, also in the SYSPRO *Program* folder.

The template script used by the custom form associated panes is called Form_CustomForm.txt and exists in SYSPRO's Base\Samples folder, along with the other associated pane templates. The *OnLoad* function of this script calls the ***COMQFM*** business object passing it the custom form type. The business object returns the list of custom form fields, their types, their size, and if numeric the number of decimals. The script then builds a form.

The *OnRefresh* function of the script is passed the custom form type in the *CustomFormType* variable, and the value of the key field in the *RefreshValue* variable. The script calls the ***COMQFM*** business object using both these values and it returns the fields and values for this key field. The script then populates the form. **Figure 20-8** shows a custom form associated pane that is associated with the *Stock code* field. Note that the associated pane's title automatically includes the name of the key field with which it is associated. This prevents confusion if custom form associated panes have been created for multiple key fields.

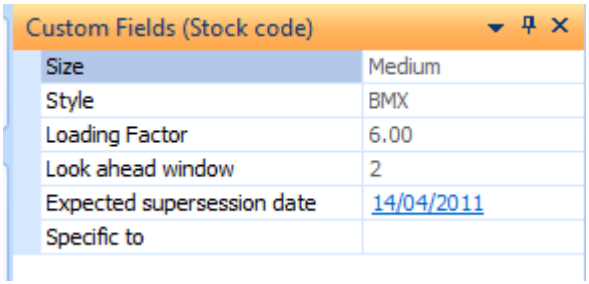


Figure 20-8: The *Custom Form* associated pane populated with values

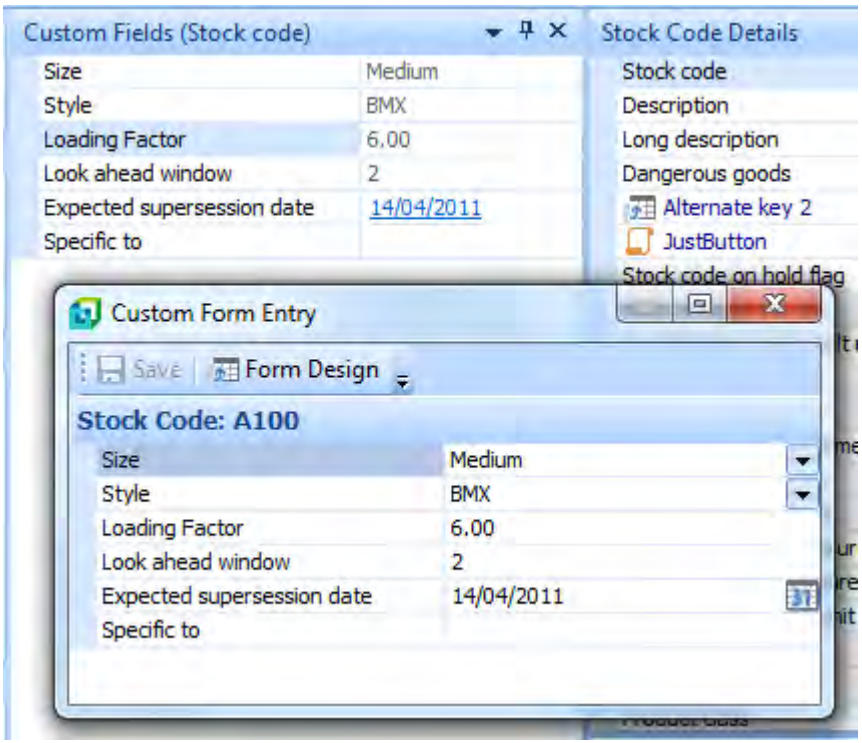


Figure 20-9: Modifying the values of a custom form

If the operator clicks on one of the values within the custom form associated pane, the *Custom Form Entry* screen is displayed. At this point the operator can use the *Custom Form Entry* screen to maintain the custom form values for this key field (see **Figure 20-9**). This is subject to the operator having the *Custom form entry* operator activity checked (*SYSPRO Ribbon Bar* | *Setup* tab | *Security*

section | *Operators* | highlight the operator | *Change* button | *Security* tab). The *Custom form entry* operator activity can be seen in **Figure 20-3**.

If the operator does not have the *Custom form entry* operator activity checked, and clicks on one of the values within this associated pane, a security access message stating that they do not have access will be displayed (see **Figure 20-10**).

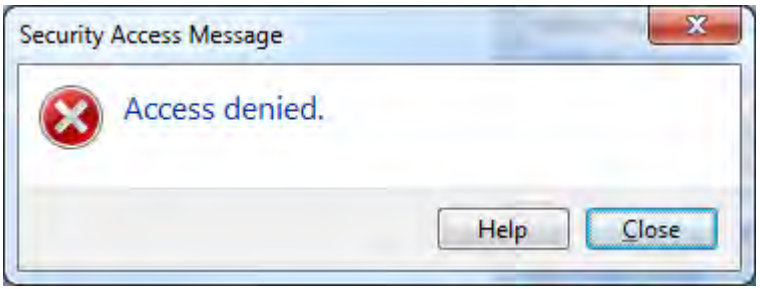


Figure 20-10: Message displayed if operator attempts to change values but does not have permission

The *Form Design* button on the toolbar of the *Custom Form Entry* screen enables you to maintain the configuration of the custom form fields as if you were doing so from the SYSPRO Ribbon Bar (*SYSPRO Ribbon Bar* | *Administration* tab | *Customization* section | *Custom Forms*). This option is only available if the *Custom form design* operator activity has been checked that can be seen in **Figure 20-3** (*SYSPRO Ribbon Bar* | *Setup* tab | *Security* section | *Operators* | highlight the operator | *Change* button | *Security* tab).

Note that if you make changes to the structure of the custom form from within the associated pane you should exit the program containing the associated pane and call up the program again. This is because the associated pane builds the form during its *OnLoad* function, so any fields added/removed will only be reflected the next time that the associated pane's *OnLoad* function is invoked.

If this *Custom form* associated pane has been inserted into a program, it can be edited in the same way as any other customized pane.

Notepad Associated Pane

The second type of special associated pane is the generic *Notepad* associated pane. SYSPRO has several places where operators can save notes against key fields, such as on the *Customer Query* toolbar. These notes are saved to Rich Text Format (`.rtf`) files so that they can contain formatting information such as different fonts. Until the *Notepad* associated panes were available, these notes were only viewable where the SYSPRO developer had made them available, or where someone had

specifically created a customized pane to view them. The *Notepad* associated pane enables you to make these notes available wherever the key field exists.

The entries for the *Notepad* associated pane appear near the end of the `IMPVBC.IMP` file, under the *[notepad]* section name. The entries within this section contain three fields, *Caption*, *Notepad code*, and the *ID* of 900. The description and icon information are not required because the description is always *Notepad*, and the icon is always the same.

The template script used by the Notepad associated panes is called `Form_Notepad.txt` and resides in SYSPRO's `Base\Samples` folder, along with the other associated pane templates. As with other associated panes, the key field is passed to the associated pane in the *RefreshValue* variable. However, in the case of the *Notepad* associated pane the *RefreshValue* variable also contains the type of note (the list of types can be found in the `IMPVBC.IMP` file under the *[notepad]* section).

These two pieces of information are both in the *RefreshValue* variable separated by a comma. The *Split* command is used to place them into an array so that they can be accessed as separate items. The ***COMNOT*** business object is called and passed these two values, and the returned information passed to the *RTFText* variable so that it can be displayed in the notepad.

The *Notepad* associated pane appears near the bottom of the list of associated panes for a key field (see **Figure 20-11**).

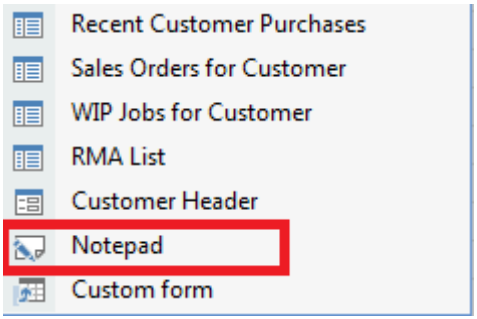


Figure 20-11: The *Notepad* associated pane

The notepad associated pane is a read-only pane, so it only displays the contents of the `.rtf` file and does not allow you to change the content. Its toolbar has a button to refresh the data, and a control to change the size of the text (see **Figure 20-12**).

Some key fields have a *Notepad* associated pane specifically written for them; such as the *Customer Notepad* associated with the *Customer* field (see **Figure 20-13**). Most of these allow the operator to modify the content of the notepad, and then save it.

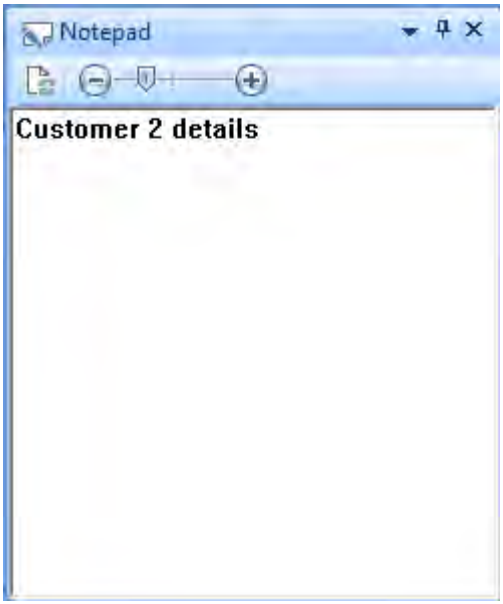


Figure 20-12: The generic *Notepad*

In the case of the *Customer Notepad*, before the pane is refreshed a check is made to see if the information that is already displayed in the notepad has been changed by the operator. If it has, this is saved before the refresh occurs. The ***COMNOT*** business object is used to retrieve the rich text from the `.rtf` file, and the ***COMSNO*** business object is used to write the text to the `.rtf` file.

The *Customer Notepad* associated pane also has a toolbar containing commands such as *Save*, *Cut*, *Copy*, *Paste*, along with the ability to change the font settings (see **Figure 20-14**).

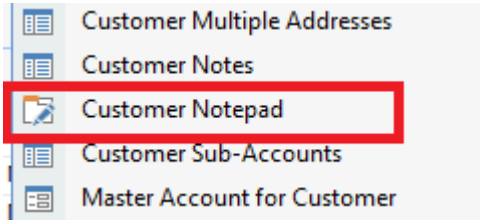


Figure 20-13: The *Customer Notepad*

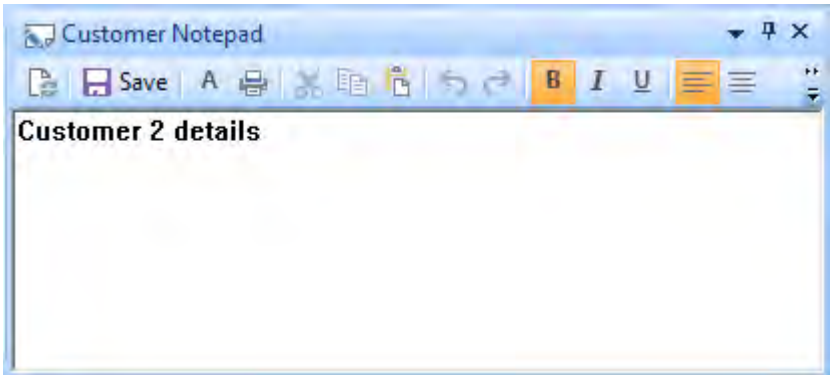


Figure 20-14: The *Customer Notepad* with the toolbar

Chapter 21 - Tips, Tricks and Gotchas

As with everything in life, once you have worked out how to do something you invariably find out that there was an easier way of doing it. This chapter can never teach you everything that you need to know, but does contain some pointers that will hopefully reduce the learning curve, or prevent you going down the wrong path.

Increased Field Sizes in SYSPRO 7

When SYSPRO 7 was released in April 2014, many of the field sizes increased in length, allowing for larger values and more transactions. If you had hard-coded checks for values in a VBScript in previous versions, in some cases these checks will fail or no longer give the desired results.

For example, in *Accounts Receivable* the customer account number increased in length from 7 characters to 15. If you use numeric numbering and your code specifically checks for a customer code of 0000002 the check will fail in SYSPRO 7 because the account number is now 000000000000002.

There is a facility to set the presentation length of key fields so that existing screens and reports do not need to be adjusted to cater for the increased field sizes. This facility is in the *Set Key Information* screen which can be seen in **Figure 21-1** (*SYSPRO Ribbon Bar* | *Setup* tab | *Setup Options* section | *General Setup* button | *Set Key Information* menu item).

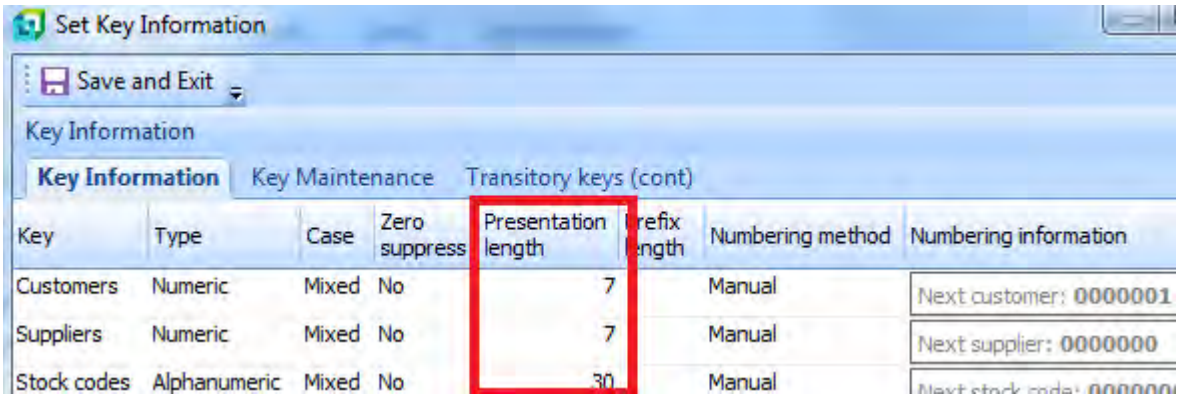


Figure 21-1: Setting/viewing the *Presentation length* of a key field

However, setting the *Presentation length* only changes the way that value is displayed; the customer value still contains 15 characters. If the following three lines of script appeared against the *OnRefresh* function of the *Customer Information* pane in the customer query, even though the value displayed against the customer prompt contains 0000002 the *IF* statement will fail because the value is 000000000000002.

```
If CustomerInformation.CodeObject.Customer = "0000002" then
  msgbox "This is customer 0000002"
End If

The script needs to be changed to the following:

If CustomerInformation.CodeObject.Customer = "000000000000002" then
  msgbox "This is customer 000000000000002"
End If
```

In **Figure 21-2** you can see the seven digit customer code is displayed on the form, but the real value is 15 digits long.

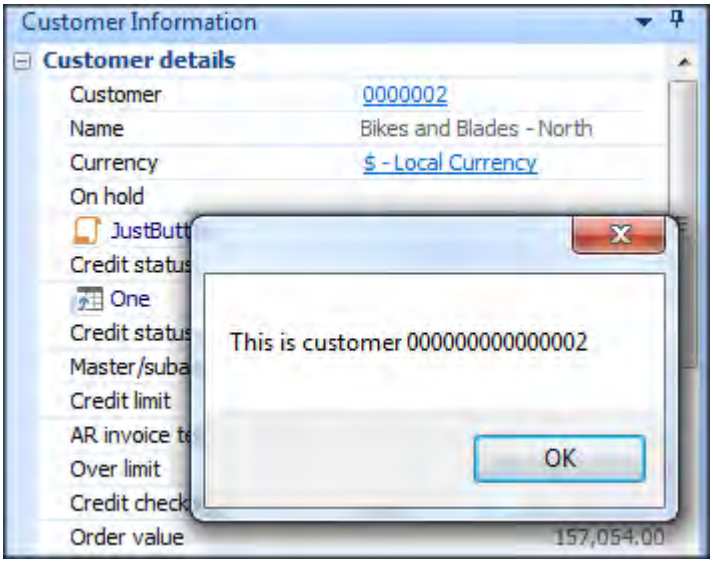


Figure 21-2: Displaying the true value of a field

Preventing Issues with Custom Form and Scripted Field names in VBScripting

It is possible to create a custom form field or scripted field where the caption is a reserved word in VBScripting. If the field containing the reserved word is used in the VBScripting, the script engine will report an error. Rather than prevent the operator from adding this field, SYSPRO adds the capital

letter “A” to the variable name used in the VBScripting. Caption names beginning with a numeric have a lowercase letter “a” added to the beginning of the variable name.

The list of reserved words in VBScripting can be seen in **Table 21-1**.

And	EndIf	LSet	RSet
As	Enum	Me	Select
Boolean	Eqv	Mod	Set
ByRef	Event	New	Shared
Byte	Exit	Next	Single
ByVal	False	Not	Static
Call	For	Nothing	Stop
Case	Function	Null	Sub
Class	Get	On	Then
Const	GoTo	Option	To
Currency	If	Optional	True
Debug	Imp	Or	Type
Dim	Implements	ParamArray	TypeOf
Do	In	Preserve	Until
Double	Integer	Private	Variant
Each	Is	Public	Wend
Else	Let	RaiseEvent	While
ElseIf	Like	ReDim	With
Empty	Long	Rem	Xor
End	Loop	Resume	

Table 21-1: The VBScript reserved words

If, for example, the operator creates a custom form field against the *Inventory Master* table called *Empty*, this can be dragged onto the *Stock Code Details* form in the *Inventory Query* program (see **Figure 21-3**).

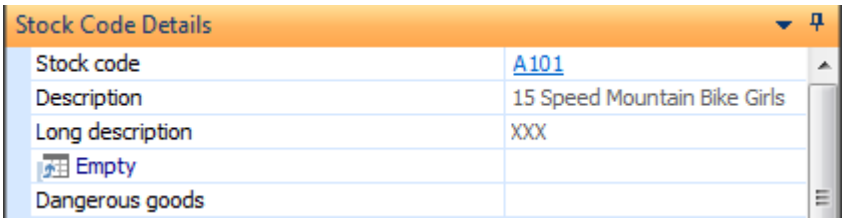


Figure 21-3: A custom form field with the name *Empty* added to the *Stock Code Details* form

Providing that this field is not used in any VBScript code this will not cause a problem. SYSPRO automatically adds the letter *A* to the end of the variable name making it *EmptyA* (see **Figure 21-4**) so that this field can be accessed using VBScript.

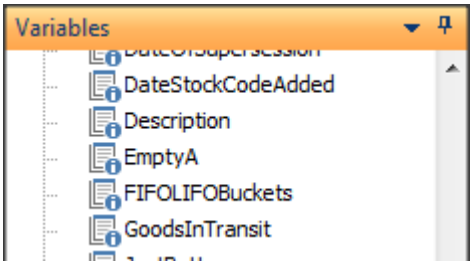


Figure 21-4: The *Empty* field appears in the list of variables as *EmptyA*

Unsetting Field Properties

When a *Field Property* is set against a field on a form, or against a cell in a listview, it typically remains set until either this field property is set to another value, or the operator exits the program. Under some circumstances you may need to set a field property against a field/cell back to its default setting.

For example, in a listview you may want to highlight the background color of a cell if the value is outside of a particular range, and remove the highlighting when the listview is refreshed and the value of this cell now falls within this range.

Within the VBScripting environment, if you clear out the value against the required field property in the *Field Properties* pane, and then click on the *Insert VBScript Code* button, the XML that is added will be blank, as per the example below:

```
CustomerInformation.CodeObject.Customer = "<Field> </Field>"
```

The code above will not change any of the currently set field properties. The *Field* element needs to include the attribute with the field property name for it to be set to nothing. However, if the setting against the field property on the *Field Property* pane is empty, clicking on the *Insert VBScript Code* button will not include this property because it is empty.

Unsetting a Field Property that Contains a Value
The simplest way to set the field property back to its default setting is by setting it to another value on the *Field Properties* pane, and clicking on the *Insert VBScript Code* button. Once this XML has been added to your code, remove the value associated with the field property's attribute. This is probably best explained with an example.

If you had previously set the background color of the *Salesperson* field to *Gold* using the code below, you may not know the value of the default background color setting.

```
OrderHeader.CodeObject.Salesperson = "<Field Background='255204000' > </Field>"
```

At the point where you need to set this field property back to its default value, you need to supply the *Salesperson* variable name, and the *Background color* field property, but without a value. This can be done by adding the variable name to your code, and adding a value to the *Background color* field property and clicking on the *Insert VBScript Code* button. You will end up with something similar to this:

```
OrderHeader.CodeObject.Salesperson = "<Field Background='255204000' > </Field>"
```

Once the line of code has been added, remove the value that appears within the two single quotes, so you end up with the following, and the field property will be set back to its default value.

```
OrderHeader.CodeObject.Salesperson = "<Field Background='' > </Field>"
```

Unsetting a Field Property defined using a checkbox
There are some exceptions to this rule. Where a field property has a checkbox against it the field property can only be *True* or *False* (see the *Bold* field property in **Figure 21-5**).

If this property is set to *True*, and later is blank, the current status of this field property will remain in effect. For example, checking the *Bold* field property and applying it to the *Salesperson* field will display the contents of the *Salesperson* field in bold. The code for this appears below:

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold='true' > </Field>"
```

If you were to change this to remove the word *true* (as per the example below) the field's content would remain in bold.

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold=' ' > </Field>"
```

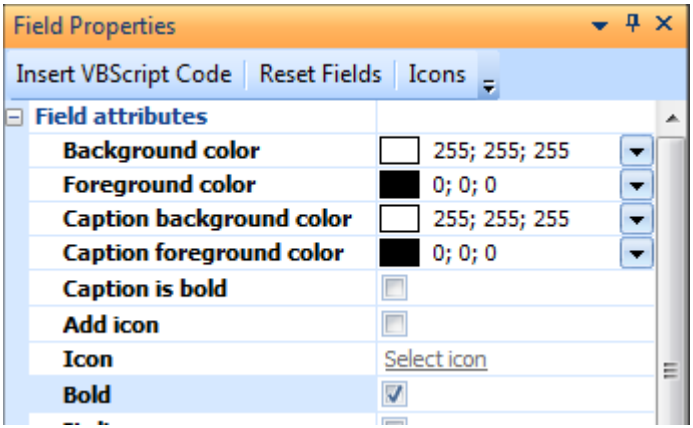


Figure 21-5: The *Bold* field property is a checkbox

In this case you must set the *IsBold* field property to *False*.

```
OrderHeader.CodeObject.Salesperson = "<Field IsBold='false' > </Field>"
```

Unsetting the XAMLCode Field Property
Another exception is where you have used the *XAML code* field property. To remove this field property it must be set to the word *none*, as per the example below.

```
OrderHeader.CodeObject.Salesperson = "<Field XAMLCode='none' > </Field>"
```

Unsetting the Field Height Field Property
If you have manipulated the height of a field using the *Field height* field property, you set this back to its default value by setting it to *-1*.

```
OrderHeader.CodeObject.Salesperson = "<Field Height='-1' > </Field>"
```

Resetting the Contents of a Customized Pane Form

The content of a customized pane form can be reset (or “cleared out”) using VBScript code. This is done by passing a space character to the *UpdateFormValues* variable. This variable is the one that you would normally use to populate the form, and is found in the *CustomizedPane* section within the *Variables* pane. In the sample code below, a checkbox has been added to the customized pane toolbar using *Toolbar control 1*. When the customized pane is refreshed, a check is made to see the status of this checkbox, and the form cleared if it is unchecked.

```
Function CustomizedPane_OnRefresh()
  If CustomizedPane.CodeObject.ButtonValue1 = 1 then
    <-- The code to populate the form would appear here -->
  Else
    CustomizedPane.CodeObject.UpdateFormValues = " "
  End If
End Function
```

Passing Multiple Values to a Customized Pane from another Pane

Another pane within the same program can force a customized pane to fire its *OnRefresh* macro event, and at the same time it can pass through a string of text. This is done from within the VBScript of the pane forcing the refresh by locating the *CustomizedPanes* section within the *Variables* pane, and double-clicking on the variable representing the customized pane to be refreshed.

In **Figure 21-6** the operator has located the customized pane to be refreshed, before double-clicking on it.

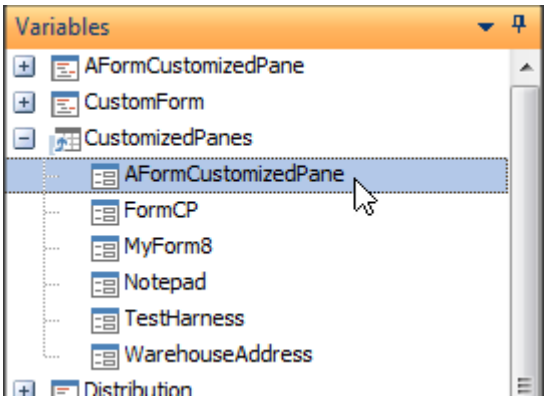


Figure 21-6: Forcing a refresh of the customized pane called *A Form Customized Pane*

When the operator double-clicks on the name of the customized pane, the *Define Action for* screen is displayed that enables the operator to specify how the customized pane is to be refreshed. The default is to pass through the text value *doRefresh* (see **Figure 21-7**).

The maximum size of the text that can be passed through to the customized pane is 80 characters. **Figure 21-8** shows where the operator has changed this value to include text that it was the *Stock Code Information* form that caused this refresh (as they want to perform different tasks depending on which pane caused the refresh).

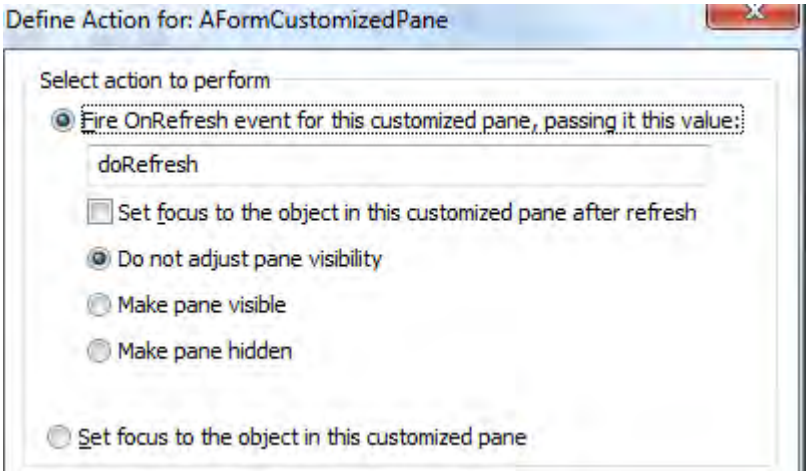


Figure 21-7: The default text of *doRefresh*

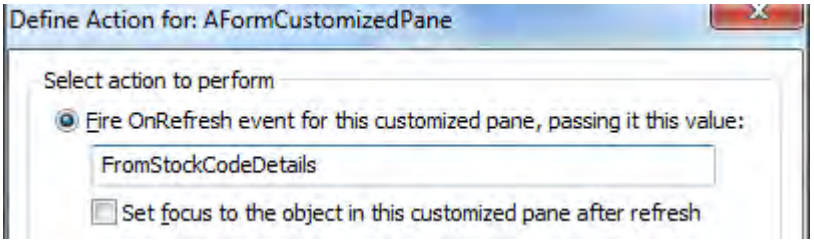


Figure 21-8: Changing the value to be passed through

You are not limited to passing through one piece of information. You may want to pass through the name of the pane and another piece of information. When doing this you need to have a character to delimit where the different pieces of information start and stop. In the code below the space character is used as a delimiter, but this may be an issue if one of your pieces of data contains a space:

```
CustomizedPanes.CodeObject.AFormCustomizedPane = "FromStockCodeDetails A100"
```

Alternatively you can use a comma as the delimiter:

```
CustomizedPanes.CodeObject.AFormCustomizedPane = "FromStockCodeDetails,A100"
```

When the customized pane is refreshed the value passed through is available within the *RefreshValue* variable in the *Variables* pane (see **Figure 21-9**).

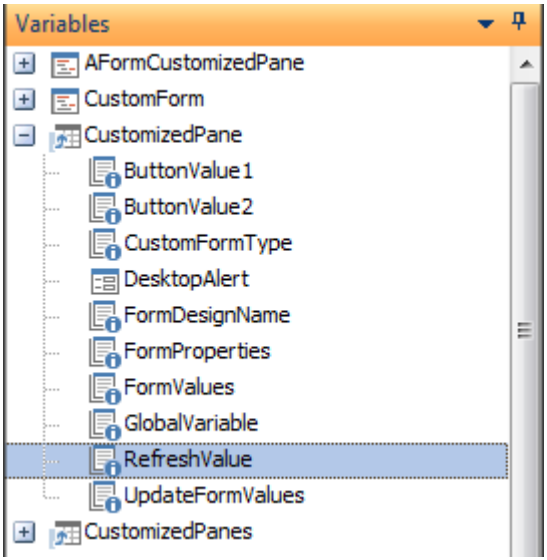


Figure 21-9: The *RefreshValue* variable within the *Variables* pane

If you are passing through multiple pieces of information in the *RefreshValue* variable they will appear as one string. You need to be able to separate them out within the VBScript code of the customized pane.

This can be performed using the *Split* function. The *Split* function needs to be given the full name of the *RefreshValue* variable, along with the delimiter so that it knows how to split up the contents of the variable.

The following code snippet would be placed within the customized pane's *OnRefresh* function. This uses the *Split* function with the delimiter of a comma to separate out the different parts of the message. The different parts are placed into an array called *BitsOfRV* so that the first part can be referenced using *BitsOfRV(0)*, the second part using *BitsofRV(1)*, etc.

In this case each of these parts is assigned to another variable, and these are displayed using a message box. It is more likely that an *IF* statement would be used against the *Wherefrom* variable so that different logic would be applied depending on which pane caused the refresh.

```
Dim BitsOfRV, WhereFrom, WhatParam
BitsOfRV = Split(CustomizedPane.CodeObject.RefreshValue,",")
WhereFrom = BitsofRV(0)
WhatParam = BitsofRV(1)
msgbox Wherefrom,,"Where did the refresh come from"
msgbox WhatParam,,"What parameter was passed"
```

When passing multiple pieces of information in this way, do not use a semi-colon (;) as the delimiter. SYSPRO already uses this as its own delimiter. If the semi-colon is used as a delimiter, only the first piece of information will be received by the customized pane and placed in the *RefreshValue* variable.

When You Call a Business Object and it Doesn't Appear to Return Results

When you call a business object from within a function and it doesn't populate the listview, or display the value against the form field, there are a couple of places to start looking. Business objects are driven by consuming XML and returning XML.

Check to see if the XML that you are Supplying appears to be Correct
If you do not supply the correct XML to the business object it will not return what you are expecting. The simplest test that you can do is to display the input XML using a message box statement. The default name of the variable containing the XML supplied to the business object (if you used the *Call Business Object* wizard to build your XML to call a *Query* class business object) is *XMLParam*. The line of code to add to display the contents of the XML would be:

```
Msgbox XMLParam
```

If the business object is either a *Transaction* class *Post* method business object, or one belonging to the *Setup* class, there are two input XML strings. One will contain the parameters and one will contain the document to be processed. In this case you will need to display both the *XMLParam* and *XMLDoc* variables.

There are two potential problems with using the message box statement to display the XML. The first is that the content of the *XMLParam* (or *XMLDoc*) variable is listed out as one long string with no formatting, which makes it difficult to see typos in the element name or missing opening/closing elements. The second is that the message box statement only displays the first 1,000 characters, so if your XML is longer than 1,000 characters the display will be truncated.

Check to see if the XML Supplied is Well-formed
For the business object to process the XML it needs to be well-formed. This means that it conforms to the XML standards.

Is there a root element? Does every element have an opening/closing element? Is the nesting done properly?

Make sure that you are not using any characters with special meanings to XML (such as an ampersand, greater-than sign, less-than sign, apostrophe, and quote) without using their entity reference, or putting them within a CDATA section.

For example, you may have a customer name containing an ampersand (&) which is a special character to XML, such as James & Alexander. You could either convert this to the entity reference:

```
<Name>James &amp; Alexander</Name>
```

Or enclose the data within a CDATA section:

```
<Name><![CDATA[James & Alexander]]</Name>
```

It is simple to overlook the basics. One developer built a system to interact between SYSPRO and their product. They were pushing out XML in their own format, and using XSL/T to convert the XML output from their format to the input required by SYSPRO's business object. But every time they started the process there was no output from the SYSPRO business object. After a little investigation it was found that one of the element names that they were using started with a number, such as *<3rdParty>*. XML element names are not allowed to start with numbers.

The following is an extract from the Microsoft Developer Network (MSDN) on the subject of elements:

> "All elements must have names. Element names are case-sensitive and must start with a letter or underscore. An element name can contain letters, digits, hyphens, underscores, and periods."

To prove that the XML is well-formed you can write it out to a text file with the suffix of `.xml`, and then try to open it using a browser. The browser will report a fault if it does not meet the requirements of an XML file. Alternatively, you can write the XML out to a text file, and then paste it into the *e.net Diagnostics* utility and select *Data* | *Tidy XML*.

The following sample code can be added to your VBScript after the point where you have built up the XML to be posted to the business object. It outputs the content of the *XMLParam* variable to a text file.

```
Dim fs, MyOutput
Set fs = CreateObject("Scripting.FileSystemObject")
Set MyOutput = fs.CreateTextFile("c:\Messages\testfileChapter9.xml", True)
MyOutput.WriteLine(XMLParam)
MyOutput.Close
```

Note that this has been hard-coded to write the text file to a folder called *Messages*. This is relative to the client where the script is being run. This also assumes the XML input is held in a variable called *XMLParam*, which is the default if you used the *Call Business Object* wizard to build up the script. If you are calling a business object that requires two XML strings you will also need to write out the contents of the *XMLDoc* variable (assuming that you are using the default name created using the *Call Business Object* wizard).

Figure 21-10 shows where the XML output has been pasted into the *e.net Diagnostics* utility and the *Tidy XML* option has been used. If this XML was well-formed it would have put each new element on a new line and indented it with two space characters. The *Tidy XML* option reported that there was an element that started with a number, which is not allowed, so reported an error.

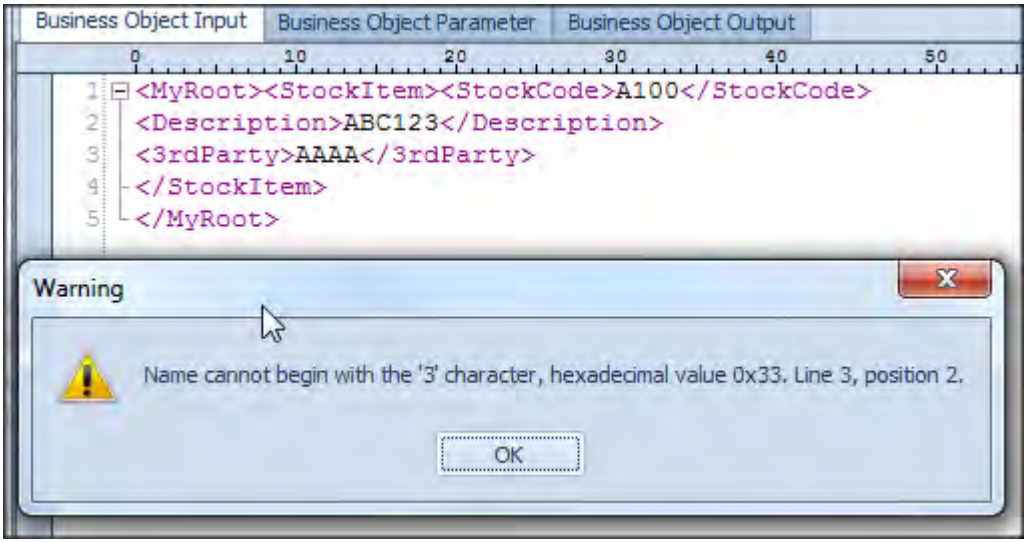


Figure 21-10: Using the *Tidy XML* option to highlight XML that is not well-formed

Check to see if there is Output from the Business Object

In a similar way to the XML input, you can write out the XML returned by the business object to see what this contains. It is possible that it has thrown an exception, it is returning an error message in the XML, or because of how you have supplied you XML the returned XML does not match what you were expecting.

To write out the XML you can use the same code as above, but replace the name of the *XMLParam* variable with the name of the variable containing the output. If you used the *Call Business Object* wizard the name will default to *XMLOut*.

```
Dim fs, MyOutput
Set fs = CreateObject("Scripting.FileSystemObject")
Set MyOutput = fs.CreateTextFile("c:\Messages\testfileChapter9.xml", True)
MyOutput.WriteLine(XMLOut)
MyOutput.Close
```

If there is no output from the business object at all it is likely that it has thrown an exception. If you used the *Call Business Object* wizard to build the code to call the business object, this will contain some lines of code that check for an exception being thrown by the business object, and notify the operator with a message box.

The code below is an extract from the code built by this wizard to call the *Inventory Query* business object. It then checks to see if an exception has been thrown, and reports it with a message box:

```
XMLOut = CallBO("INVQRY",XMLParam,"auto")
if err then
  msgbox err.Description, vBCritical, "Calling Business Object"
  exit function
end if
```

If your code does not contain a means of trapping/displaying an exception, you should add something similar. **Figure 21-11** shows a message box containing the exception that is thrown if the XML supplied to a *Query* class business object contains an invalid key field value.

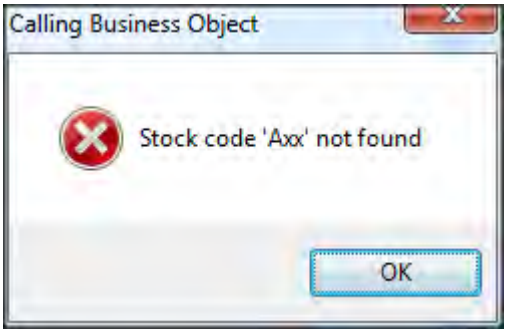


Figure 21-11: The exception thrown when calling a *Query* class business object with an invalid key

The business object could also be returning an error message from SYSPRO within the XML. In this case the elements that you are checking for in the returned XML to use in the form/listview are not contained in the XML. You should load the returned XML into the *Document Object Model* (DOM) and perform a check looking for a node called *ErrorMessage* or *ErrorDescription* before attempting to process the output. This way you know that the returned XML contains an error.

The following is an extract from the XML returned by the *Sales Order Import* business object (***SORTOI)*** when a check of the price percentage above cost failed. This prevented the order from being processed, so the code looking at the output from the business object for the element containing the sales order number will fail, as this element is not in the XML (note that the contents of the *ErrorDescription* element have wrapped around on this page).

```
<ErrorMessages>
  <ErrorDescription>The price does not exceed the cost by the specified
margin</ErrorDescription>
</ErrorMessages>
```

Another type of error message is contained within the *WarningMessage* node. This does not stop the XML from being processed, but might supply information about something that requires remedial action.

The following is an extract from the XML returned by the *Sales Order Import* business object when there was an insufficient quantity of part *B100* to ship all that was required. The business object accepted the order but put the quantity into backorder, and the sales order into the backorder status. This requires some intervention from the operator before the item can be shipped. Therefore it is also good practice to also check for *WarningMessage* nodes.

```
<WarningMessages>
  <WarningDescription>Line 0001 for stock code 'B100' was placed on back
order</WarningDescription>
</WarningMessages>
```

Business objects that post transactions can also contain other elements that specify that the posting failed. The following XML is also from the *Sales Order Import* business object. When the parameter XML contains the *<Process>IMPORT</Process>* element the business object first performs a validation of the whole XML, and if this is successful it attempts to process it. If the validation fails the *ValidationStatus* node will contain a status element showing that the validation phase failed.

```
<ValidationStatus>
  <Status>Failed</Status>
</ValidationStatus>
```

Another check that can be made with the output from business objects that work in the same way as the *Sales Order Import* business object, is to see if any items were flagged as being invalid.

```
<StatusOfItems>
  <ItemsProcessed>000001</ItemsProcessed>
  <ItemsInvalid>000000</ItemsInvalid>
</StatusOfItems>
```

Does the Output from the Business Object Match your Listview Design

When you design the structure of a customize pane listview you need to specify the node name of the repeating node that contains the data. This appears against the *PrimaryNode* attribute of the *Columns* element. The following is an extract from the *Columns* element where the *PrimaryNode* has been set to the element name of *Item*:

```
<Columns PrimaryNode='Item' Style='DataGrid' AutoSize='false'
```

Using the XML below, if you want to display all the stock codes and their matching descriptions you must specify the element name *MyRow* against the *PrimaryNode* attribute in the listview structure. This is the repeating node that contains the elements that you need.

Note that this is case-sensitive. If you supply the incorrect name for the *PrimaryNode* the business object will run and return the results, but the listview will not be populated.

```
<MyRoot>
  <MyHeader>
    <Customer>0000001</Customer>
    <PostCode>12345<PostCode>
  <MyHeader>
  <MyRow>
    <StockCode>ABC1</StockCode>
    <Description>Description1</Description>
  </MyRow>
  <MyRow>
    <StockCode>ABC2</StockCode>
    <Description>Description2</Description>
  </MyRow>
  <MyRow>
    <StockCode>ABC3</StockCode>
    <Description>Description3</Description>
  </MyRow>
</MyRoot>
```

When you are defining the *Column* elements (the individual columns within the listview), the *Name* attribute must exactly match the element name in the XML; both the spelling and the case.

Below is an extract from the structure of a customized pane listview. This *Column* element has *OrderStatus* against the *Name* attribute. Therefore it is looking for an element called *<OrderStatus>*, not one called *<Orderstatus>* or *<orderstatus>*. If it does not find an element called *<OrderStatus>* nothing will appear against this column for this row.

```
<Column Name='OrderStatus' Description='Order status'…
```

Programatically Force the Operator to Exit the Program

There are times when you want to force an operator to exit a program. This could be when they have completed a transaction, or maybe because another operator has already completed the task.

The following line of code can be used to perform the task of exiting the operator from the current program back to the menu. This uses the logic built into the screen of this program so will only allow the operator to exit if the program would normally allow them to exit at this point.

```
SystemVariables.CodeObject.ActionToInvoke = "ExitApplication,60000"
```

An example of where the operator would be prevented from closing the program is where the operator is adding a sales order in the *Sales Order Entry and Maintenance* program. If they have already entered the first detail line (causing the sales order to be created) they would not be allowed to just exit the program. They would need to save the order, or cancel out of it. Calling the line of code above would return the same message as if they had tried to manually close the program at that point.

Populating the Listview without Using XML

Using XML is the simplest way to populate a customized pane listview. It was designed this way so that it could match up with the output of the business objects, which is typically where the data to populate the listview will come from.

When the data to be displayed is not too large, or it is already in XML, the results are rendered immediately. However, if there is a lot of information to be displayed, and it is not already in XML (such as being extracted from SQL Server), it can be more efficient to put the data into the listview's native format and let it handle the display. In the case of data being retrieved from SQL Server there is an overhead when building the XML, only to have SYSPRO unpack it so that it is rendered in the listview.

The structure of the listview is always defined using XML (as covered in Chapter 11) regardless of whether you are going to use XML to populate it.

The native format of the listview is that each cell of the listview row is separated using the ASCII code 255. The last cell on a row has the *Carriage Return* and *Line Feed* characters. There is no ASCII 255 character after the last cell of a row before the *Carriage Return* and *Line Feed* characters. This is probably best explained with an example.

Non-XML Example
This example was built using the *Application Builder* option from the *Administration* tab of the *SYSPRO Ribbon bar*. The *Application Builder* dropdown list contains 20 empty applications (***IMPDH1 - 9*** and ***IMPDHA - H***). These are empty shells to which you can insert one or more customized panes. In this example a single customized pane listview has been added. The *Application Builder* is covered in detail in Chapter 9.

Use the *Application Builder* dropdown list to select the application to be used. In this example *Application 2* was used. Not strictly necessary, but worth doing to make life easier later, is to change the display name of the application. This is performed from the toolbar of the application (see **Figure 21-12**).

Click on the *Add a Customized Pane* button on the toolbar to start adding the customized pane listview. Set the *Object type* to listview, add a *Window title*, and click on the *Edit VBScript* button on the toolbar. The *VBScript Editor* screen will be displayed. On the *VBScript Editor* double-click the *OnLoad* event name. The screen where you edit the VBScript will be displayed, the *CustomizedPane_OnLoad* function will be created for you, and the cursor placed within it.

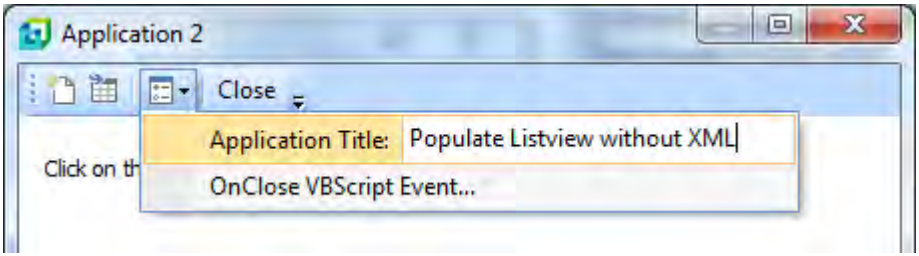


Figure 21-12: Changing the title of an *Application Builder* application

Double-click on the *ListviewDesigner* option within the *CustomizedPane* section of the *Variables* pane. This will load the *Listview Designer* screen, which you should use to build the listview structure. The *PrimaryNode* should be set to *MyRow*.

The listview should have three columns called *When*, *Quality*, and *Amount*. The *When* field should be configured as *alphanumeric*, with a description of *When this was sold*. The *Quality* field should be configured as *alphanumeric*, with a description of *Unit quality*. The *Amount* field should be defined as numeric with two decimals, and have the description of *How many tons were sold*.

Once you have built this listview structure, use the *Insert VBScript* button on the toolbar to add the code to your *OnLoad* function.

Exit back to the *VBScript Editor* screen, and double-click the *OnRefresh* event name to create this function for you. If you were using XML to populate this listview the XML would be similar to the following:

```
dim ListData
ListData = "<MyRoot>"
ListData = ListData & "<MyRow>"
ListData = ListData & "<When>2014-06-25</When>"
ListData = ListData & "<Quality>Good</Quality>"
ListData = ListData & "<Amount>123.99</Amount>"
ListData = ListData & "</MyRow>"
ListData = ListData & "<MyRow>"
ListData = ListData & "<Amount>30.25</Amount>"
ListData = ListData & "<When>2014-08-25</When>"
ListData = ListData & "<Quality>Bad</Quality>"
ListData = ListData & "</MyRow>"
ListData = ListData & "</MyRoot>"

CustomizedPane.CodeObject.ListviewData = ListData
```

Note that the last line of code passes the XML to the *CustomizedPane.CodeObject.ListviewData* variable which processes it and displays it. The sequence of the elements within the *MyRow* node is

not important as the data is extracted using the element name. In this XML example the sequence is different for each *MyRow* node to emphasize the point. If there is no element for one of these columns in one of the rows, it is ignored, as it is not mandatory.

However, the following example does not use XML, so the values to be passed to the individual cells must be in the correct sequence and separated by the ASCII character 255. The rows must also be terminated with the *Carriage Return* and *Line Feed* characters. If a cell needs to be empty then the ASCII character 255 for this cell must still be present, even though there is no value.

The following VBScript code builds up the contents of four rows for this listview and passes it to the *CustomizedPane.CodeObject.ListviewData* variable. Note: a blank line has been inserted between each of these lines to make it easier to read, as several of the lines wrap around on this page.

```
Dim MyValues

MyValues = "2014-01-31" & Chr(255) & "Best" & Chr(255) & "100.25" & VbCrLf

MyValues = MyValues & "2014-01-31" & Chr(255) & "Not so good" & Chr(255) & "30.5" &
VbCrLf

MyValues = MyValues & "2014-02-28" & Chr(255) & "Best" & Chr(255) & "105.10" &
VbCrLf

MyValues = MyValues & "2014-02-28" & Chr(255) & "Not so good" & Chr(255) & "22.75"
& VbCrLf

CustomizedPane.CodeObject.ListviewData = MyValues
```

The results of this can be seen in **Figure 21-13**.

The first line defines the variable that is used to contain all of the information as it is built up.

The second line adds the values of the three cells of the first row to the variable *MyValues*, separated with the ASCII character 255. After the third value is a *String Constant* of *VbCrLf*, which is used to provide the *Carriage Return* and *Line Feed* characters.

The third, fourth and fifth lines are similar to the second line except that they append to the content of the *MyValues* variable.

The last line takes the content of this variable and passes it to the *CustomizedPane.CodeObject.ListviewData* variable so that it is processed/displayed.

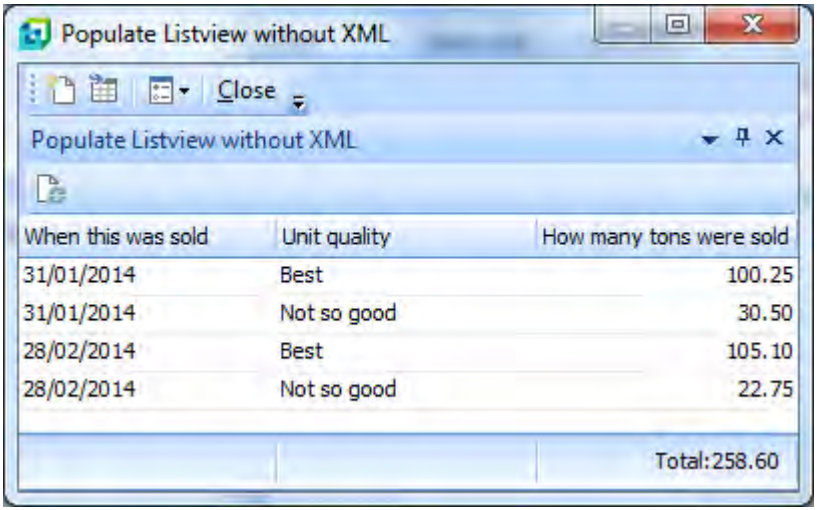


Figure 21-13: The finished result of populating the listview without using XML

Finding Out Which Macro Events Fire and When

The simplest way of finding out when a macro event fires is by using message box statements against the functions within the VBScript.

Using Message Box Statements

Although adding message box statements against each event is simple to do, you need to make sure that the name of the function is included in the message box. Else you will know that an event has fired, but not necessarily which one. As you add message box statements to more functions, this becomes more important.

The following is a simple example that appears against the *Asset Details* pane in the *Asset Query* program. Both the *OnLoad* and *OnRefresh* functions use a message box statement to display the name of the function.

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function AssetDetails_OnLoad()
  msgbox "AssetDetails_OnLoad"
End Function

Function AssetDetails_OnRefresh()
  msgbox "AssetDetails_OnRefresh"
End Function
```

This is fine if you are only tracking events for one pane. However, if you are tracking multiple panes and multiple events on each pane you will have many message boxes appearing, and each one requires operator intervention before they will go away and processing continues.

Events and Messages Window

In the *Administration* section of the *Administration* tab of the *SYSPRO Ribbon Bar* is the *Diagnostics* menu. On this menu is the *Show Events Window* option which loads the *Events and Messages* program (see **Figure 21-14**).

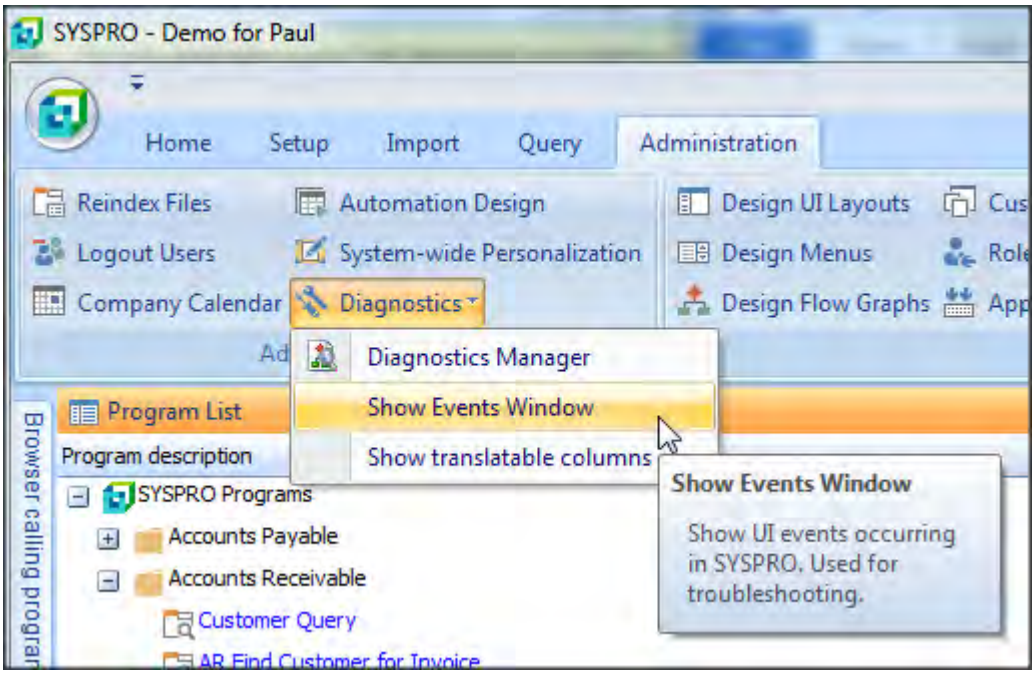


Figure 21-14: The *Show Events Window* menu item

The *Events and Messages* program contains a floating pane that is used to display a log of when events are fired and messages sent. Most of these events and messages are only useful to SYSPRO developers, but there is an option to include when the VBScript events are fired. This option is under the *Options* menu on the toolbar, and is called *Show VBScripts* invoked. To best use this option, unselect all the other options on this menu, otherwise the results will be very cluttered (see **Figure 21-15**).

Only macro events that have a function and are fired are included in this list. If there is no function created for a macro event the event will not fire, and it will not appear in this list.

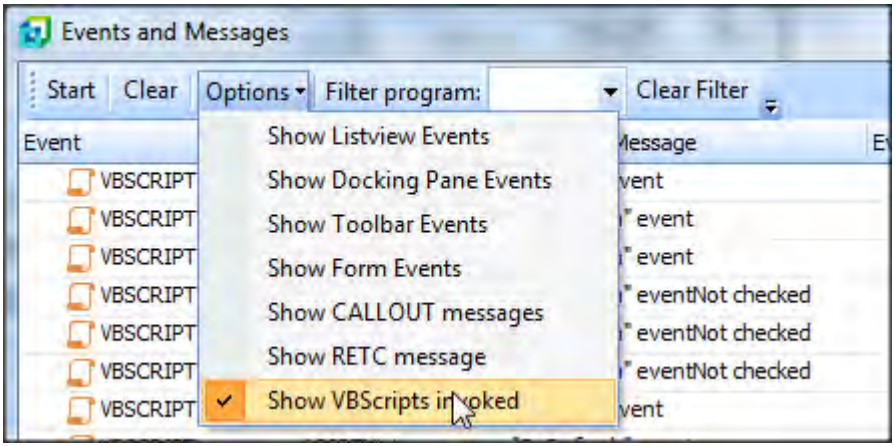


Figure 21-15: The option within the *Events and Messages* program to show the VBScript events

The functions for the events to be monitored must exist, but do not need to contain any code. Using the *Asset Details* pane of the *Asset Query* program (as used in the message box example above), the following code would cause the *OnLoad* and *OnRefresh* events to fire:

```
' This script contains functions for form and field events.
' You must not modify the name of the functions.
Option Explicit

Function AssetDetails_OnLoad()

End Function

Function AssetDetails_OnRefresh()

End Function
```

An extract from an *Events and Messages* screen appears in **Figure 21-16**. The information displayed is from loading the *Asset Query* program where the *OnLoad* and *OnRefresh* functions are present for the *Asset Detail* pane (***ASSPENL1***) and *Purchasing Details* pane (***ASSPENL2***), and the *OnPopulate* function is present for the *Asset Entries* listview (***ASSPENLI***).

The *Events and Messages* pane is itself a listview, so most of the functions of a normal listview are available from the menu that is displayed when right-clicking on the column header. In **Figure 21-16** the columns have been re-sequenced from the default, so that the most important ones appear on the left of the listview.

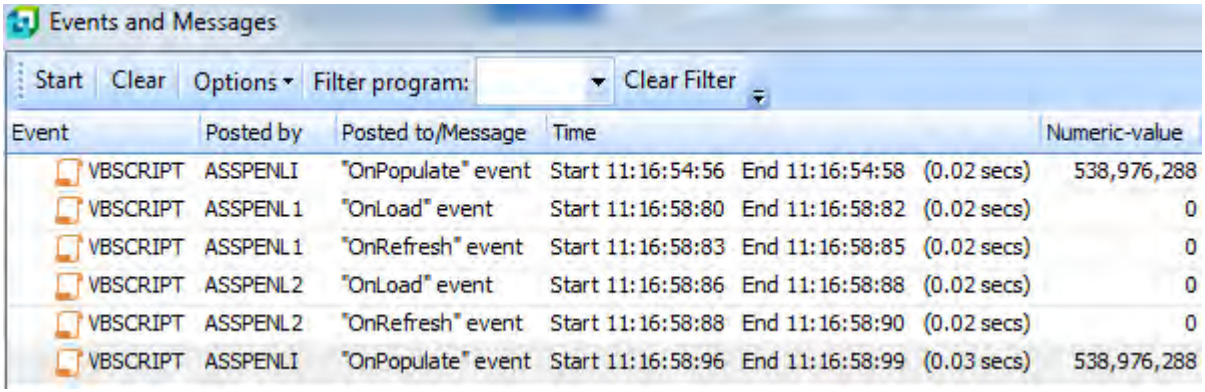


Figure 21-16: The *Events and Messages* pane

If you are only going to view VBScript events you could drag the *Event* column header from the listview, and it would no longer appear. This column would be available in the *Field Chooser* if you needed to return it to the *Events and Messages* listview at a later time (see **Figure 21-17**).

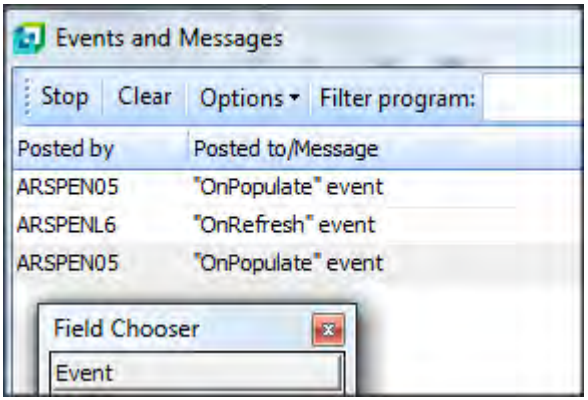


Figure 21-17: The *Event* column available in the *Field Chooser* after being removed from the listview

Each time that the *Events and Messages* program is loaded all of the different types of events and messages are set to be tracked. So if you only want the VBScript ones, you will need to unselect all the others each time.

When the *Events and Messages* program is loaded it is always in recording mode, and the *Stop*/*Start* button will display *Stop*. Clicking on the *Stop* button will stop the recording and toggle the *Stop*/*Start* button. Typically you would stop the recording until you have called up the program to be tracked, and you are ready to start tracking the events. Then you would start recording.

If you are only interested in recording events from a particular program, you can enter the six character program name against the *Filter program* prompt. Only events belonging to this program will be recorded. The filter can be removed using the *Clear Filter* button.

You can clear out the contents of the listview using the *Clear* button. You do not need to stop recording to clear out the listview. If you did not stop the recording, as soon as another VBScript event fires it will be recorded as the first entry in the listview.

You can only close the *Events and Messages* screen when you are at the *SYSPRO Main Menu*. You cannot close this program while you have another program open. When you exit SYSPRO the *Events and Messages* screen closes. The next time that SYSPRO is loaded by this operator this *Events and Messages* program will not load by default. If it is still required, it must be loaded again.

Another standard listview feature that is available to the *Events and Messages* program is to *Export to Excel*. This enables you to retain this information even after closing the *Events and Messages* screen, or even exiting SYSPRO.

Manipulating SYSPRO Dates using VBScript

SYSPRO stores dates internally using the CCYYMMDD format (Century, Century, Year, Year, Month, Month, Day, Day), even though they are displayed in the format that you selected when SYSPRO was installed. When interacting with SYSPRO forms and listviews using VBScript, dates are retrieved in the format CCYYMMDD, and when using entry forms (or the *_OUT* array during the *OnPopulate* function of a listview), dates can be updated using this format too.

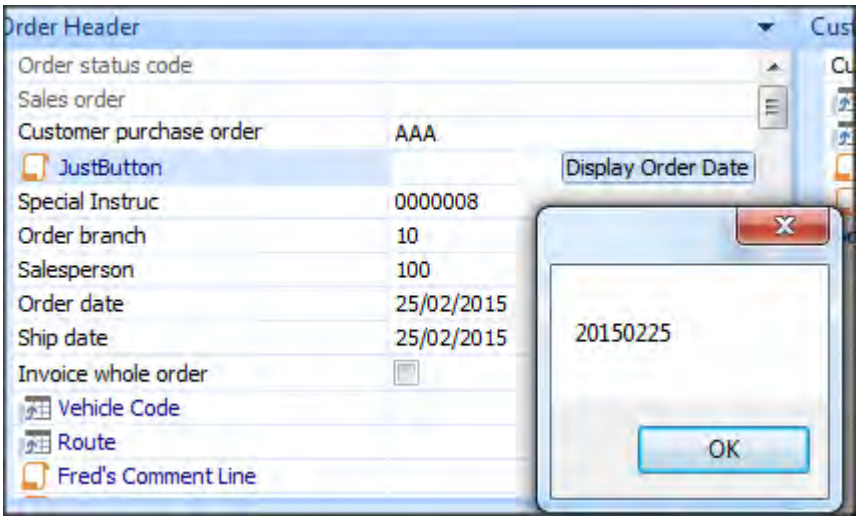


Figure 21-18: Using a button to display the *Order date*

In **Figure 21-18** the *Order date* has been displayed in a message box to show its true format.

VBScripting has built-in functions for manipulating dates such as *DateDiff*, *DateAdd*, *IsDate* and *DateValue*. However, VBScript date functions expect dates to be in the *Short date* format configured within the operating system on your machine.

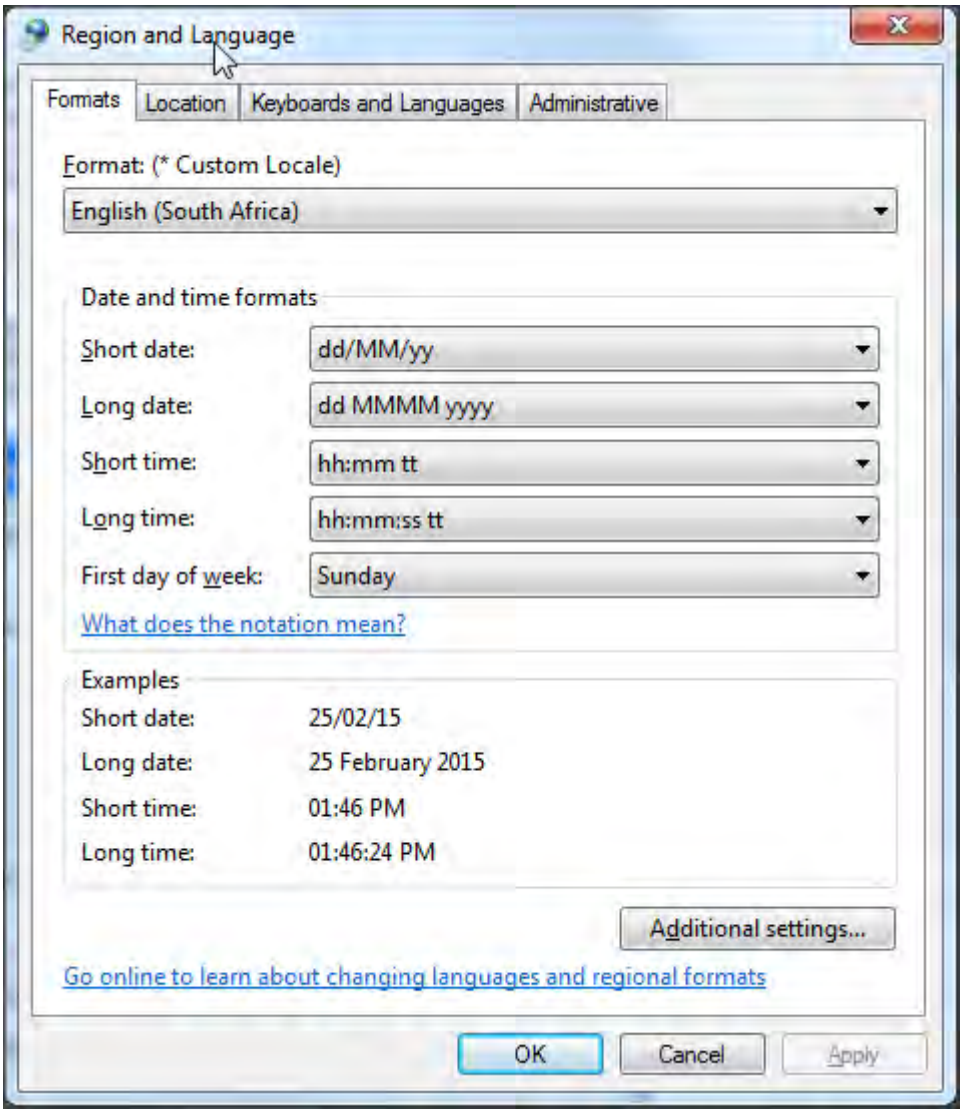


Figure 21-19: For Windows 7, *Short date* appears on *Formats* tab of the *Region and Language* app

The format can be seen/changed using the *Region and Language* applet on the Windows *Control Panel*. The *Short date* entry appears on the *Formats* tab (see **Figure 21-19**). Other versions of Windows may be slightly different.

If you intend to manipulate dates using any of the VBScript date functions you must convert the date supplied by the SYSPRO form to the one that is stored as your *Short date* format. If you need to write the date back after manipulation, you must convert the date back to the CCYYMMDD format before writing it back.

In this example, code is placed against the *OnLostFocus* function against the *Order date* field. The code takes the order date and adds 20 days to it, and then writes out the *Ship date*.

The dates to be manipulated need to have slashes (/) inserted (see line four). After the date manipulation only the numbers are extracted and written to the CCYYMMDD format date, the slashes are dropped.

Below is the code added to the *OnLostFocus* function for dd/mm/yy format (note: each line is preceded by a comment line and followed by a blank line. Some lines wrap around):

```
' Create the variables to be used
dim SysproDate, DaysToAdd, MyShipDate, NewShipDate

' Specify the number of days to be added
DaysToAdd = 20

' Read in the Order date
SysproDate = OrderHeader.CodeObject.OrderDate

' Convert it to my Short date format
MyShipDate = DateValue(Right(SysproDate,2) & "/" & Mid(SysproDate,5,2) & "/" &
Left(SysproDate,4))

' Add the number of days and populate NewShipDate
NewShipDate = dateadd("d", DaysToAdd, MyShipDate)

' Extract the pieces and write out in CCYYMMDD format
SysproDate = "20" & Right(NewShipDate,2) & Mid(NewShipDate,4,2) &
Left(NewShipDate,2)

' Populate the Ship date field
OrderHeader.CodeObject.ShipDate = "<Field Value='" & SysproDate & "' > </Field>"
```

Figure 21-20 shows where the operator has entered the date of the 28th August 2015 and tabbed off the field. When focus was lost, 20 days were added to this date and the new date (17th September 2015) used to update the *Ship date* field.

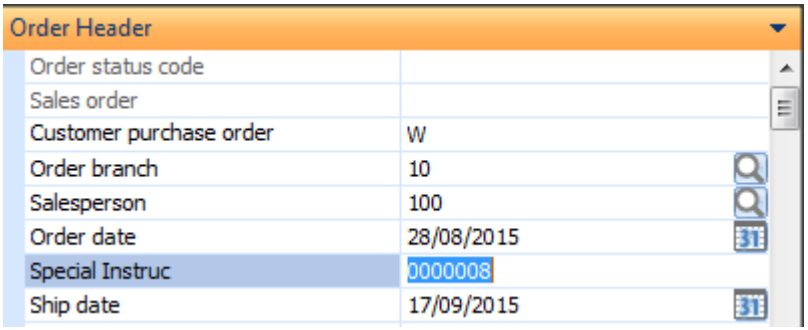


Figure 21-20: Increasing the *Order date* by 20 days and populating the *Ship date*

Below is the same code, but for mm/dd/yy format (note: each line is preceded by a comment line and followed by a blank line. Some lines wrap around):

```
' Create the variables to be used
dim SysproDate, DaysToAdd, MyShipDate, NewShipDate

' Specify the number of days to be added
DaysToAdd = 20

' Read in the Order date
SysproDate = OrderHeader.CodeObject.OrderDate

' Convert it to my Short date format
MyShipDate = DateValue(Mid(SysproDate,5,2) & "/" & Right(SysproDate,2)  & "/" &
Left(SysproDate,4))

' Add the number of days and populate NewShipDate
NewShipDate = dateadd("d", DaysToAdd, MyShipDate)

' Extract the pieces and write out in CCYYMMDD format
SysproDate = "20" & Right(NewShipDate,2) & Left(NewShipDate,2) &
Mid(NewShipDate,4,2)

' Populate the Ship date field
OrderHeader.CodeObject.ShipDate = "<Field Value='" & SysproDate & "' > </Field>"
```

Below is the same code, but for yy/mm/dd format, which is only required because of the slashes (note: each line is preceded by a comment line and followed by a blank line. Some lines wrap around):

```
' Create the variables to be used
dim SysproDate, DaysToAdd, MyShipDate, NewShipDate

' Specify the number of days to be added
DaysToAdd = 20

' Read in the Order date
SysproDate = OrderHeader.CodeObject.OrderDate

' Convert it to my Short date format
MyShipDate = DateValue(Left(SysproDate,4) & "/" & Mid(SysproDate,5,2) & "/" &
Right(SysproDate,2))

' Add the number of days and populate NewShipDate
NewShipDate = dateadd("d", DaysToAdd, MyShipDate)

' Extract the pieces and write out in CCYYMMDD format
SysproDate = "20" & Left(NewShipDate,2) & Mid(NewShipDate,4,2) &
Right(NewShipDate,2)

' Populate the Ship date field
OrderHeader.CodeObject.ShipDate = "<Field Value='" & SysproDate & "' > </Field>"
```

Note that if your *Short date format* also includes the “20” of “2015” you will need to tweak the code above.

Adding Extra Lines to a Customized Pane Listview

If you have a customized pane listview, you can allow the changing of values within the listview by making individual columns editable when building the listview structure. This is done by adding the *Editable=’true’* attribute to the individual *Column* elements within the *Columns* node. An example appears below:

```
<Column Name='CustomerPoNumber' Description='Customer po number' Editable='true'/>
```

This allows you to change the content of this listview column. To allow the operator to add a new row to this listview the attribute *AutoInsert=’true’* must be added to the *Columns* element, an example appears below:

```
<Columns PrimaryNode='SalesOrderItem' Style='DataGrid' AutoSize='false'
FreezeColumn='1' AutoInsert='true' >
```

The *Columns* element describes how the whole listview works, whereas the *Column* element describes how an individual column will behave. To be able to add extra lines to the listview the

AutoInsert attribute of the *Columns* element must be true, and at least one *Column* must have its *Editable* attribute set to *True*.

Telling a SYSPRO Form that the Form has been Updated

When the content of a standard SYSPRO form is modified by an operator, SYSPRO enables the *Save* or *Post* button on the toolbar (for forms that have these buttons). This can be after changing one field when editing an existing item, or after certain required fields have been populated when adding a new item.

Figure 21-21 shows where a new product class has been added using the *Product Class Maintenance* program. As soon as the *Product Class* and *Description* fields have been populated the *Save* button becomes enabled.

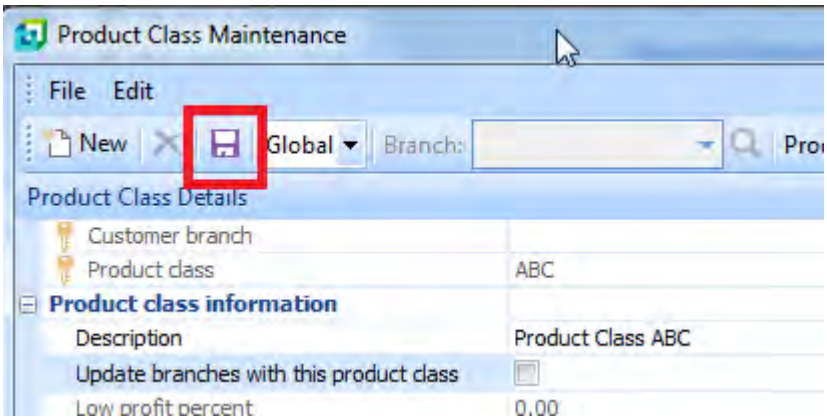


Figure 21-21: The *Save* button being enabled

However, if the operator calls up a program and a VBScript event makes a change to the current item, the SYSPRO program does not know that this change has happened, so the *Save*/*Post* button is not enabled. You need to enable this button programmatically by setting the *PostChangeEvent* variable to 1.

Using the *Product Class Maintenance* program as an example, when the operator calls up a product class the *Update branches with this product class* checkbox is automatically checked by the following code:

```
Function ProductClassDetails_OnRefresh()
  ProductClassDetails.CodeObject.UpdateBranchesWithThisProductClass = "1"
End Function
```

Because this change was not performed by an operator, the *Product Class Details* form does not know that this has been changed so does not enable the *Save* button on the toolbar (see **Figure 21-22**).

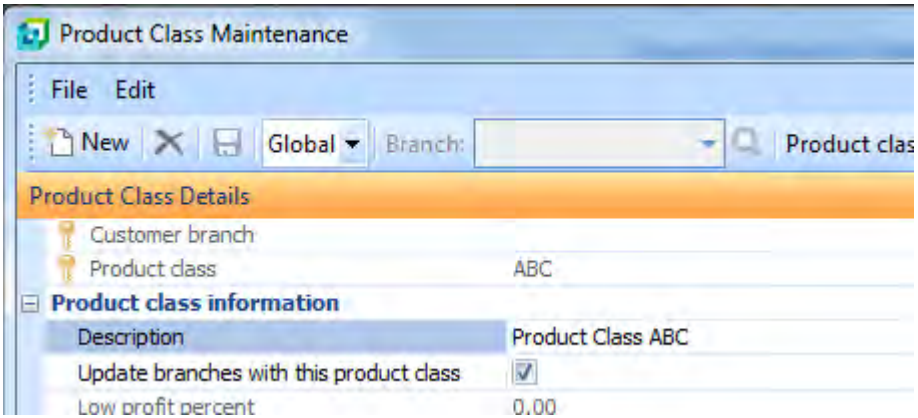


Figure 21-22: The checkbox was checked programmatically but the *Save* button was not enabled

The *PostChangeEvent* variable resides under the *SystemVariables* section on the *Variables* pane within the VBScripting environment (see **Figure 21-23**).

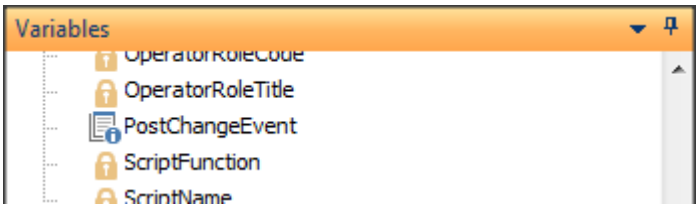


Figure 21-23: The *PostChangeEvent* variable

Double-clicking on this will add the full name of this variable to your code. Add this variable to your code after the point where you set the checkbox status, and set its value to 1. The *Save* button will be enabled as it is in **Figure 21-21**.

```
Function ProductClassDetails_OnRefresh()
  ProductClassDetails.CodeObject.UpdateBranchesWithThisProductClass = "1"
  SystemVariables.CodeObject.PostChangeEvent = "1"
End Function
```

Note that this performs slightly differently if the operator had made the change. It just enables the *Save* button. Although the *Save* button is enabled, if the operator just exits the program there is no prompt to warn the operator that these settings will be lost. Whereas if the change had been made by the

operator and they attempt to close the program without saving, a message prompting them to save the changes is displayed (see **Figure 21-24**).

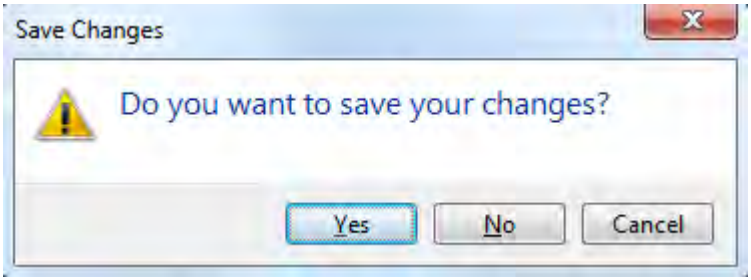


Figure 21-24: Prompting to save changes when exiting after the operator made changes

Prevent Tampering with your VBScripts/Protect your Intellectual Property

The VBScripts that are used with customized panes, listviews, and forms are stored as plain text. You can prevent operators from viewing or editing them from within SYSPRO by unchecking the operator security activity *VBScript editing* (*SYSPRO Ribbon Bar* | *Setup* tab | *Security* section | *Operators* | select operator | *Change* button | *Security* tab | *Activities* section). **Figure 21-25** shows where the operator's activity *VBScript editing* has been unchecked.

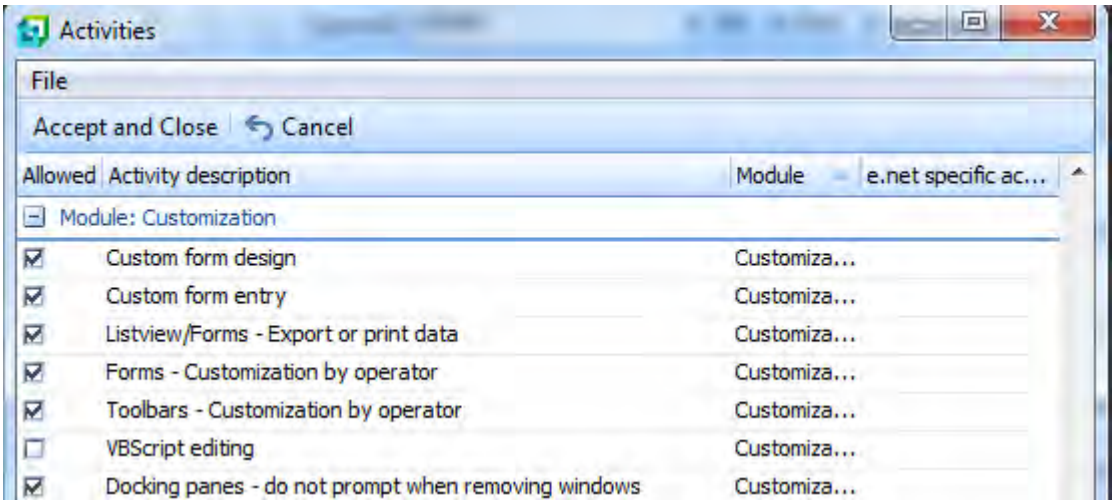


Figure 21-25: The *VBScript editing* operator activity

However, the script must still be moved from the server to the client by SYSPRO, because this is where it is executed by the program. It is possible for the operator to use a text editor to manipulate

this text file if they can work out where it resides and what it is called (which will not be mentioned here).

There are two options under the *VBScript Security* section of the *System-wide Personalization* program that can be used to prevent tampering with your scripts, or protect your intellectual property contained within them. These options are *VBScripts are saved on the server only* and *VBScripts are encrypted on client* (see **Figure 21-26**).

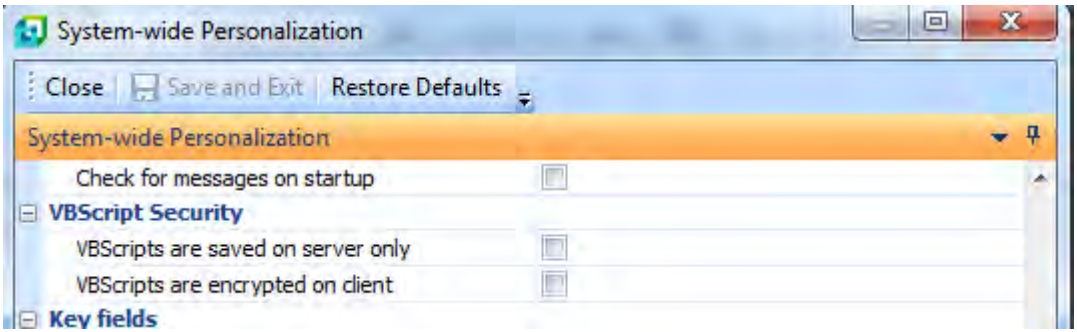


Figure 21-26: *VBScript Security* options under *System-wide Personalization*

VBScripts are Saved on the Server

The *VBScripts are saved on the server* option prevents unauthorized access to VBScripts by ensuring that they are only stored on the server. When this option is initially enabled, all VBScripts are copied back to the server and removed from the client. When an operator logs in, the VBScripts are copied to the client, cached into memory and then immediately removed from the client's hard drive. When the operator logs out, the cache is cleared. Disabling this option after it has been enabled will cause all the scripts to be copied back to the client.

Note that any `.vbs` files in the *vbsmodules* folder will remain on the client when this option is checked.

VBScripts are Encrypted on Client

When the *VBScripts are encrypted on client* option is checked, the next time that operator logs in, all the scripts are self-healed to the client machine. They are then encrypted and are given a `.snk` (*Strong Name Key*) suffix. The *date-time stamp* is then updated on this file. Any change to the script on the server causes the later version of this script to be downloaded to the client and encrypted the next time that the operator logs in.

Note that this encryption will not affect the performance of the script.

Prevent Having the same VBScript code in Multiple Places

If you have the same (or similar) logic in multiple VBScripts, consider extracting this logic and placing it into a *VBScript Module*. These are VBScript files with the suffix of `.vbs` that reside in the *vbsmodules* folder. You then include the contents of this module *(*`.vbs` file) by placing the following statement near the top of your script (assuming that your module is called *MyModule.vbs*):

```
LoadModule("MyModule.vbs")
```

The benefit of placing this code in one place is that if it needs to change, you only need to change it in one place and is automatically available in all scripts that reference it.

VBScript Modules were covered in detail in the *VBScript Modules* section of Chapter 6.

SYSPRO32.DLL and Encore.DLL

Versions prior to SYSPRO 7 shipped with the e.net Solutions COM object called `Encore.dll`. This has been retained with SYSPRO 7 for backwards-compatibility, so all existing customization, power tailoring and integration will continue to work as before. `SYSPRO32.dll` ships with SYSPRO 7 as well. This means that if you have a test environment that has both SYSPRO 7 and a prior version, SYSPRO 7 can be configured to use the `SYSPRO32.dll` and the prior version can be configured to use `Encore.dll`.

Note that the currently shipping version of *e.net Diagnostics* can only use `Encore.dll`, so if you need to use this product with SYSPRO 7 you will need to configure SYSPRO 7 to use `Encore.dll`.

Using the Option to Add Custom Columns to Listviews

As covered in Chapter 8 on Listviews, and Chapter 11 on Customized Pane Listviews, it is possible to add custom columns to listviews. There are settings that prevent the operator from using the *Add Custom Column* option, as well as cases where it will not be available (see **Figure 21-27**).

Standard Listview
The operator will not be able to use the *Add Custom Columns* option if:

- they are a member of a role and are not in design mode
- the operator is denied access by either of these two security options being unchecked (see **Figure 21-28**):
  Forms – Customization by operator
  Forms – Caption adjustments for group/company

Standard Data Grid
You cannot use the *Add Custom Column* option for a *Data grid* because it contains columns that are editable.

Customized Pane Listview
The operator will not be able to use the *Add Custom Columns* option if:

- they are a member of a role and not in design mode
- the operator is denied access by either of these two security options being unchecked (see **Figure 21-28**):
  Forms – Customization by operator
  Forms – Caption adjustments for group/company
- One or more of the columns in the listview is defined as being editable (has a column containing *Editable='true'* configured against it)

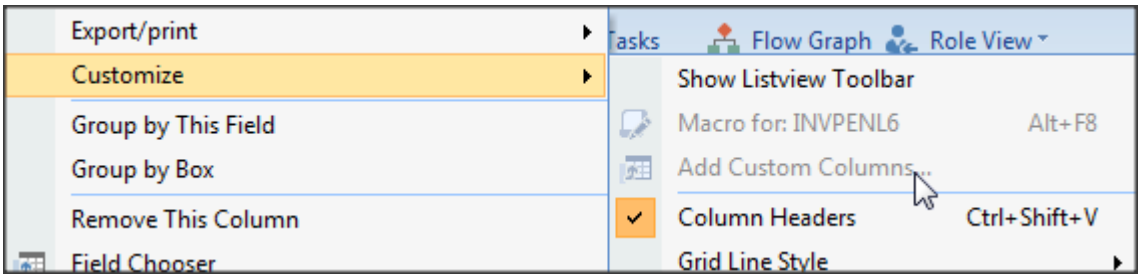


Figure 21-27: The *Add Custom Columns* option disabled

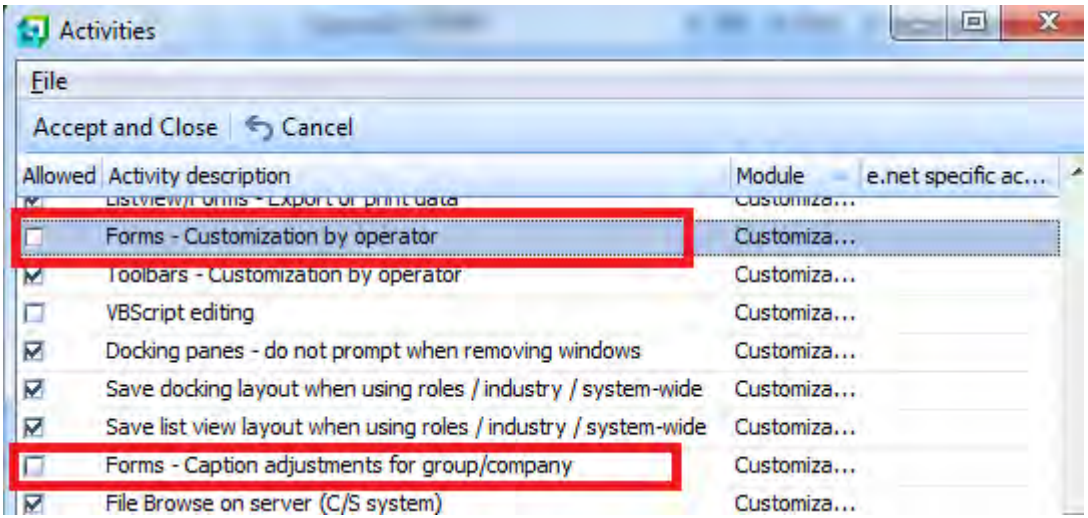


Figure 21-28: The two operator activities that disable the *Add Custom Columns* option

The Different Diagnostics Options that can be used with e.net Solutions

SYSPRO can be configured to retain diagnostic information about several different e.net Solutions transactions. These diagnostics are configured system-wide. The *Diagnostics* dropdown menu appears on the *General* tab of the *System Setup* screen (*SYSPRO Ribbon Bar* | *Setup* tab | *Setup options* section | *General Setup* | *System Setup* | *General* tab). **Figure 21-29** shows the *Diagnostics* being configured to use *enet01*.

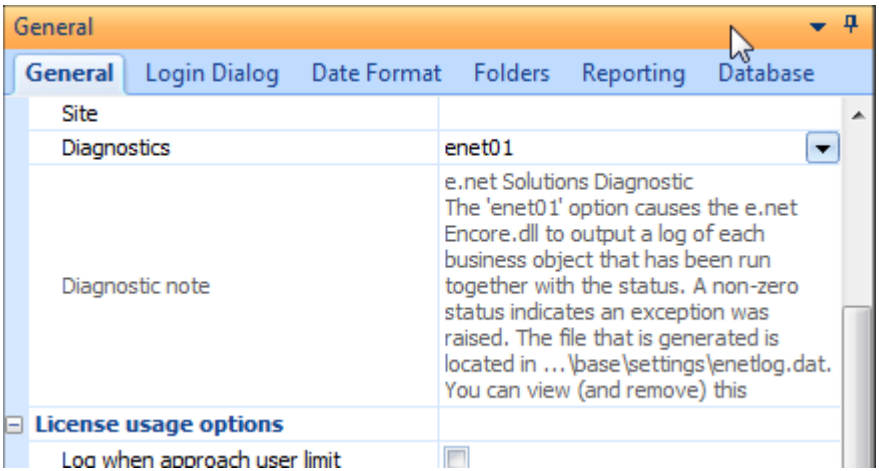


Figure 21-29: Selecting the *enet01* diagnostics option

Configuring and Using Enet01 Diagnostics

The *enet01* diagnostics option tracks all transactions that use the e.net Solutions COM object. It tracks the date, time, operator, business object, return status, and error message. This information is stored in the ***enetlog.dat*** and ***.idx*** files under the `Base\Settings` folder. Note that as transactions made from within SYSPRO do not use the e.net Solutions COM object they are not tracked using *enet01*.

The output from *enet01* can be viewed under the *Diagnostics Manager* (*SYSPRO Ribbon Bar* | *Administration* tab | *Administration* section | *Diagnostics* menu | *Diagnostics Manager*). On the right at the top of the screen is a pane containing three tabs, *Application Server Checks*, *e.net Transaction Log* and *e.net Admstate Log*.

Selecting the *e.net Transaction Log* tab displays an empty listview and a toolbar. The toolbar contains a *Refresh* button, a *Delete the transaction log* button, a *print* button, and *export to Excel* button, a search box, and a *Clear* button. The *Refresh* button populates the listview with the contents of the *enetlog.dat* file. The *Delete the transaction log* button deletes the transaction log files. The *Print* and

Export to Excel buttons behave as expected. Entering text against the search box and pressing the *Enter* key filters the content of the listview to only include rows that contain the search text. The *Clear* button clears this filter so the full content of the listview is displayed.

Figure 21-30 shows a section of the *e.net Transaction Log*, where some error messages can be seen. These are exceptions thrown by the business objects, not error messages included in the returned XML. The *Return status* is the error number that was returned by the business object.

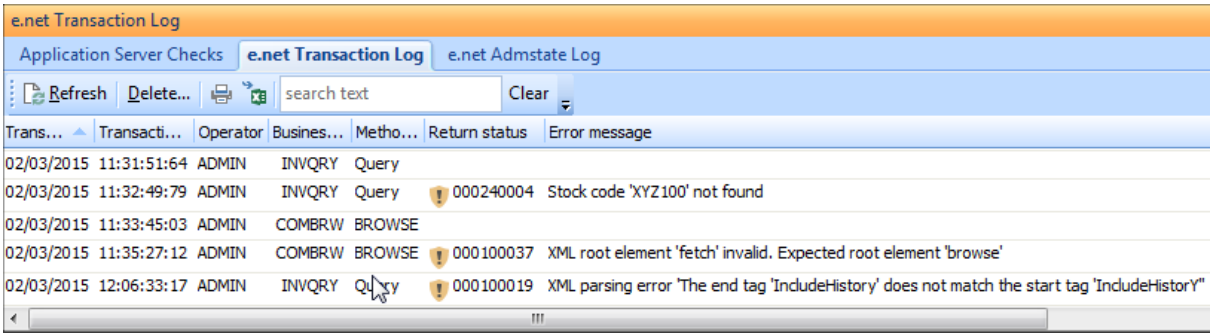


Figure 21-30: Viewing the *e.net Transaction Log*

Configuring and Using Enet02 Diagnostics

The *enet02* diagnostics is similar to the *enet01* diagnostics, but instead of tracking transactions that are processed by the e.net Solutions COM object it tracks transactions that are processed by the *Document Flow Manager*, which does not use the COM object.

Configuring and Using Enetxx Diagnostics

The *enetxx* diagnostics causes the XML inputs and outputs from the business objects to be written to the `Base\Settings\diag_xml` folder as individual XML files. The filenames consist of the date, time, entry number, SYSPRO operator code, business object name, and the text *XMLIn* or *XMLOut*. For business objects requiring two XML input files/strings, the second one will be *XMLIn_01*.

If there are many transactions, having this diagnostics configured can impact on the performance and produce a large number of files.

Embedding VBScript Code in the .NET User Control

When using a *.NET User Control* customized pane you can embed the VBScript directly in your user control rather than having to add it to the customized pane itself. This is covered in detail in Chapter 16 on *.NET User Controls*.

If you have a *.NET User Control* customized pane that can be imported into many programs, and you want it to be able to call different functionality depending on which program it resides in, you should consider adding this functionality to a *VBS Module* (*VBS Modules* are covered in Chapter 6 on *Using the VBScript Editor*). This has the added benefit that if a change needs to be made it can be made to this text file without having to supply (and replace) the *.NET User Control*. You may not even be aware of all of the programs where the operator has installed this customized pane.

Documenting Your Power Tailoring

It is always a good idea to document your Power Tailoring. You may well remember what you did (and why you did it) this week or this month, but will you remember the nuances in six months' time. It may also be someone else that has to decipher what you did. Documenting your coding is always a good start. It is also a good idea to standardize the way you name variables, constants, etc., so that it is obvious what it is used for without having to work through your code.

As a bare minimum, any scripts should contain the author's name, date of creation, description of what the script does, and details of any dependencies such as custom form fields or scripted field that must be present. With the author's name, the person looking at the code knows who to ask about the coding if they get stuck.

It is also recommended that you try to simplify your code. It is easy to start off trying to achieve an objective and find that it can't be done that way. Instead of starting from scratch again you try to leverage off of this code, and maybe have another false start. When you finish the code it is always worth going back through it while it is still fresh in your mind and remove variables that are no longer required, remove functions that are not being used, document what you have done, and generally tidy it up.

It is usually better to create several functions and call them when required than to have one long sequential script.

When you have finished you may find that you have used the same, or similar, code in this script as you have in another. Now is the time to consider stripping this out and adding it into a *VBS Module* so that the code only exists in one place, and is called by both places.

Using Writeline with Windows 8.1

SYSPRO uses the same VBScripting engine as Internet Explorer. Internet Explorer on Windows 8.1 has a security setting that prevents the *writeline* function from writing out the text. The *writeline* function has been used in several of the examples above where the XML has been written to a text file, such as in the *Check to see if the XML Supplied is Well-formed* section. The code from this section appears below. When this is used under Windows 8.1 the file is created but never populated.

```
Dim fs, MyOutput
Set fs = CreateObject("Scripting.FileSystemObject")
Set MyOutput = fs.CreateTextFile("c:\Messages\testfileChapter9.xml", True)
MyOutput.WriteLine(XMLParam)
MyOutput.Close
```

However, if you wrap the code in a VBScript class and then call the class methods in our VBScript it will all work. The following script is slightly different in that it appends the text to an existing file (and creates it if it does not exist) instead of creating a new one each time:

```
Dim wl
Set wl = New cls_LogFile
wl.WriteLogLine("Update all Branches with this Product Class - changed")
Set wl = Nothing

'Declare class
Class cls_LogFile
  Function WriteLogLine(strLogLine)

    Dim fs, MyOutput
    Set fs = CreateObject("Scripting.FileSystemObject")
    Set MyOutput = fs.OpenTextFile("C:\Messages\Chapter9.txt",8,True)
    MyOutput.WriteLine(strLogLine)
    MyOutput.close

  End Function
End Class
```

You could go a step further and add this class to a *VBS Module*. That way you only need to have it in one place, and by using the *LoadModule* statement you immediately have access to write out the log.

Using the SendKeys Method

The *SendKeys* method is used to supply keystrokes to the currently active window, as if the operator had typed them on the keyboard. These can be supplied as individual characters, or as a string of characters. Some characters such as the plus sign “+”, caret “^”, percent sign “%”, tilde “~” and parentheses “(“ and “)” have special meanings to *SendKeys*. To use their normal meaning these must be wrapped in braces “{}”. Square brackets “[]” must also be wrapped in braces, but they have no special meaning.

The following is a simple example showing how *SendKeys* can be used. It comes from *the Inventory Movements* program. When focus is set on the *Quantity* field a 1 is inserted and the *Tab* is used to move to the next field.

```
Function Quantity_OnGainFocus()
  dim objShell
  Set objShell = CreateObject("WScript.Shell")
  'Populate the Quantity field with 1
  objShell.SendKeys "1"
  'Send Tab character to move off the quantity field
  objShell.SendKeys "{TAB}"
  set objshell = nothing
End Function
```

As in the example above, there are times when you need to represent characters such as the *Tab* key. **Table 21-2** shows the codes used to represent these keys.

Key	Code
Backspace	{BACKSPACE}, {BKSP} or {BS}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Delete	{DELETE} or {DEL}
Down Arrow	{DOWN}
End	{END}
Enter	{ENTER} or ~
Escape	{ESC}
Help	{HELP}
Home	{HOME}
Insert	{INSERT} or {INS}
Left Arrow	{LEFT}
Num Lock	{NUMLOCK}
Page Down	{PGDN}
Page Up	{PGUP}
Print Screen	{PRTSC}
Right Arrow	{RIGHT}

Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up Arrow	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

Table 21-2: The codes used to represent certain keys

There are also times when you need to represent keystrokes that require multiple keys to be pressed at the same time, such as *Alt+F8* or *Ctrl+F*.

Table 21-3 contains the codes to enable this. For example, *Alt+F8* would be represented by using the following:

```
objShell.SendKeys "%{F8}"
```

And *Ctrl+F* would be represented using:

```
objShell.SendKeys "^F"
```

Key	Code
Alt	**%**
Ctrl	**^**
Shift Lock	**+**

Table 21-3: The codes use to represent *Alt*, *Ctrl*, and *Shift* keys

The Customization Profiler

The *Customization Profiler* provides a complete customization analysis of the current program. The customization profiler is loaded by selecting the *Menu* button on one of the panes within the program, and selecting *Customization Profiler* from the displayed menu (see **Figure 21-31**).

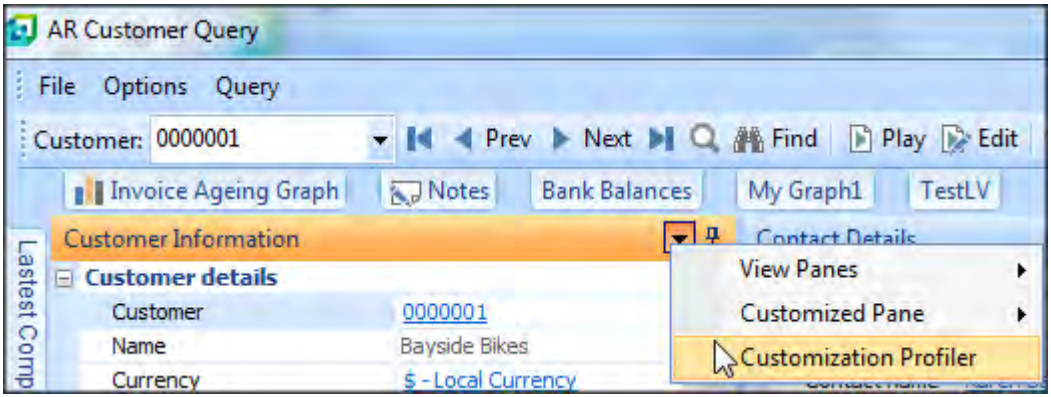


Figure 21-31: Loading the *Customization Profiler*

When the *Customization Profiler* is loaded it shows information about all the customization for all panes and listviews in this program.

This includes listing all customized panes within this program, their types, and the script associated with them (see **Figure 21-32**). For *Form* customized panes that have a separate XML containing the form's structure, the XML filename is supplied. For *.NET User Control* customized panes the *Assembly name* and *Namespace* are included. Any associated panes are included in the *Customized panes* section.

Custom form fields that have been added directly to a form are listed, as well as those that have been added to a listview using the *Add Custom Columns* option. *Form Actions* that have been added to a form appear in their own section, along with their associated VBScripts.

Customization section	Column 1	Column 2	Column 3	Column 4
Customized panes	**Description**	**Type**	**Information**	**VBScript nam**
	Dummy Listview	Listview		ARSPEN01
	My Listview2	Listview		ARSPEN04
	My Listview3	Listview		ARSPEN05
	Bank Balances	Graph		ARSPEN06
	My Graph1	Graph		ARSPEN07
	Lastest Company Information	Browser		ARSPEN08
	Movements - Custom	Listview		ARSPEN09
	My RTF Notepad	Notepad		ARSPEN11
	Test of Address	Form	AssemblyName settings\ADMIN_VBS_ARSPEN13.XML Namespace	ARSPEN13

Figure 21-32: An extract from the *Customization Profiler* showing the *Customized panes* section

VBScripts	Description	Loaded from	Procedures	Special calls
ARSPENL7	ContactDetails	C:\TST700\...		
ARSPENLV	Customer Invoices	C:\TST700\...		
ARSPEN13	TestOfAddress	settings\ADM...		
ARSPEN11	My RTF Notepad	settings\ADM...	CustomizedPane_OnLoad	
ARSPENTB		C:\TST700\...	OnDalesParam	
ARSPENLE	Quotations	C:\TST700\...		
ARSPEN08	Lastest Company Information	settings\ADM...		
ARSPENLA	ExtraCustomerDetails	C:\TST700\...		
ARSPEN06	Bank Balances	settings\ADM...	CustomizedPane_OnRefresh CustomizedPane_OnClick CustomizedPane_OnLoad RefreshGraph	SYSPROXmlout = CallBO("COMQEX",SYSPR Set xmlDoc = createobject("Msxml2.Fre
ARSPENL6	CustomerInformation	C:\TST700\...	CustomerInformation_OnGoToMasterAcco... CustomerInformation_OnRefresh	XMLOut = CallBO("ARSQRY",XMLPar Set XMLDoc = createobject("MSXML2.D "<Toolbar Name='ARSI Action='Execute' Value=" Caption='Customer:'/>" &

Figure 21-33: An extract from the *Customization Profiler* showing the *VBScripts* section

The *VBScripts* section shows the name of the VBScript and its description (see **Figure 21-33**). It also shows where the script is loaded from, which functions exist within the script, and extracts from the script.

Copying Customization from one Role to Another

There are times when you have created a specific customization against one *Role* and you need the same customization, or similar, against another *Role*. In these cases you can use the *Customization Management* program to copy between roles (*SYSPRO Ribbon Bar* | *Administration* tab | *Customization* section | *Customization Management*). The *Customization Management* program can be seen in **Figure 21-34**, where the *Account Manager* role has been selected.

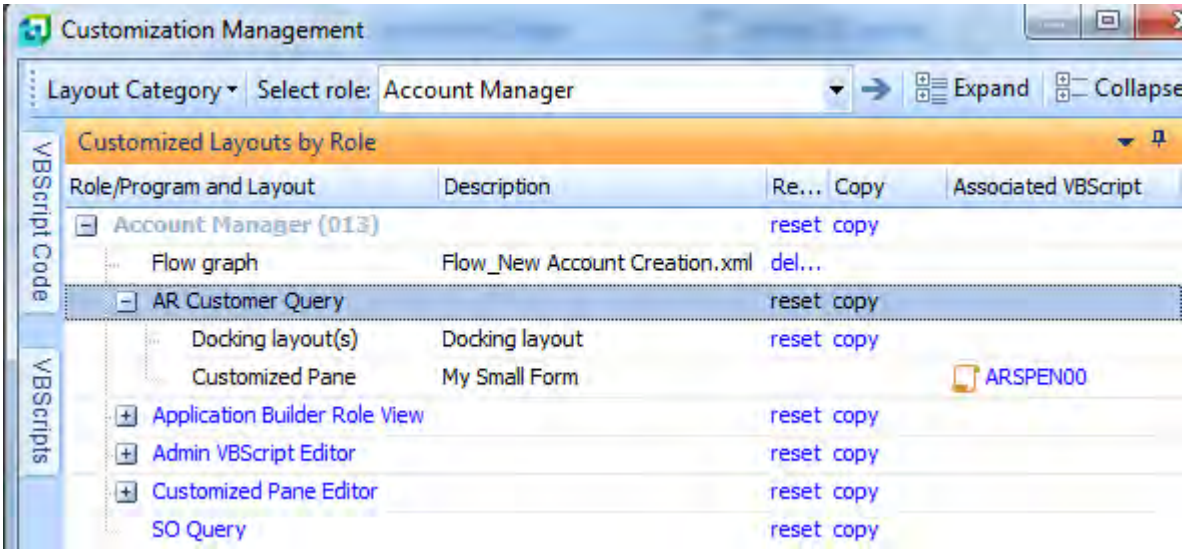


Figure 21-34: The *Customization Management* program

To copy the customization you must first locate the required customization. The *Layout Category* button enables you to choose to display *Role*, *System-wide*, or *Industry* layouts, so select the *Role Layout* option. Once you have selected the required role, the customization for this role is displayed. Locate the specific customization and click on the *copy* hyperlink. You can copy the entire customization for the role by selecting the *copy* hyperlink against the role, for individual programs by selecting the *copy* hyperlink against the program, or for individual components/panes by clicking on the *copy* hyperlink against the individual item.

After clicking on the *copy* hyperlink the *Copy Layout* screen is displayed (see **Figure 21-35**). This enables you to select one or more roles to copy to, with the options to select/unselect all of them. When you click on the *Copy* button the customization is copied to the role(s).

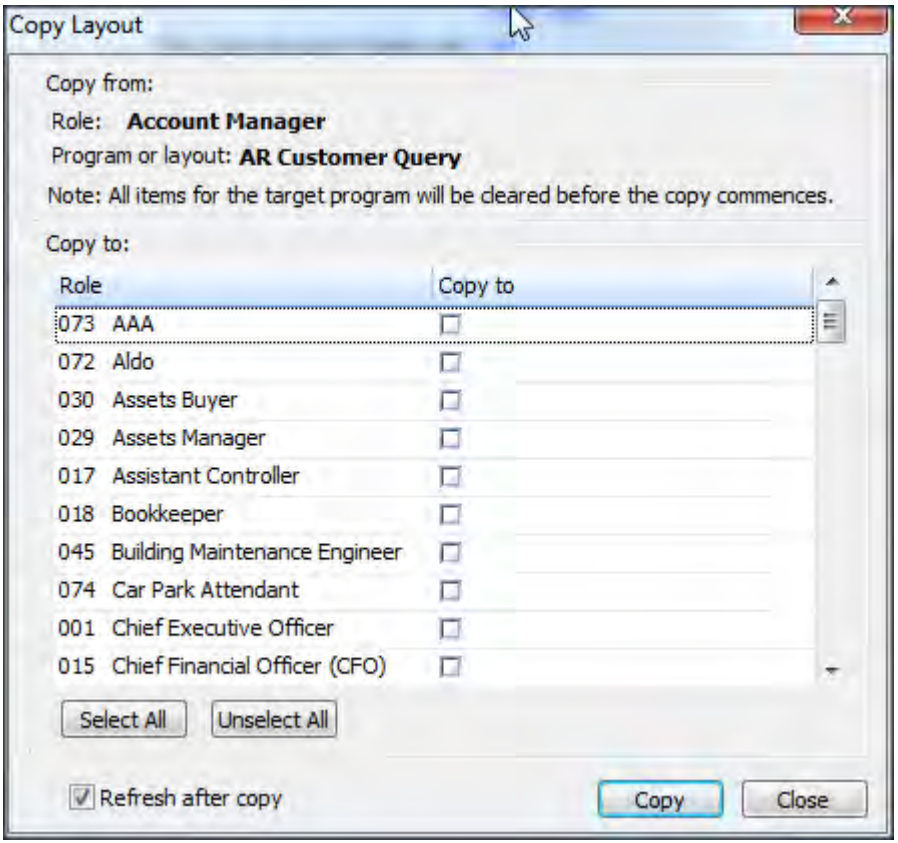


Figure 21-35: Selecting while role(s) to copy to

Adding more than 20 Items to a Dropdown List against a Form Field

When designing a customized pane *Form,* a field can be defined as having a *Field type* of *Drop down* (see **Figure 21-36**). You supply a list of up to 20 items to appear in the dropdown list using the *Enter items* hyperlink (see **Figure 21-37**).

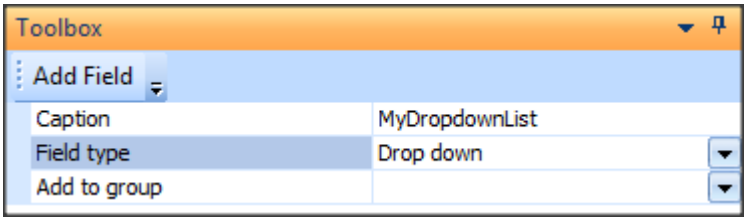


Figure 21-36: Defining a field as having a *Field type* of *Drop down*

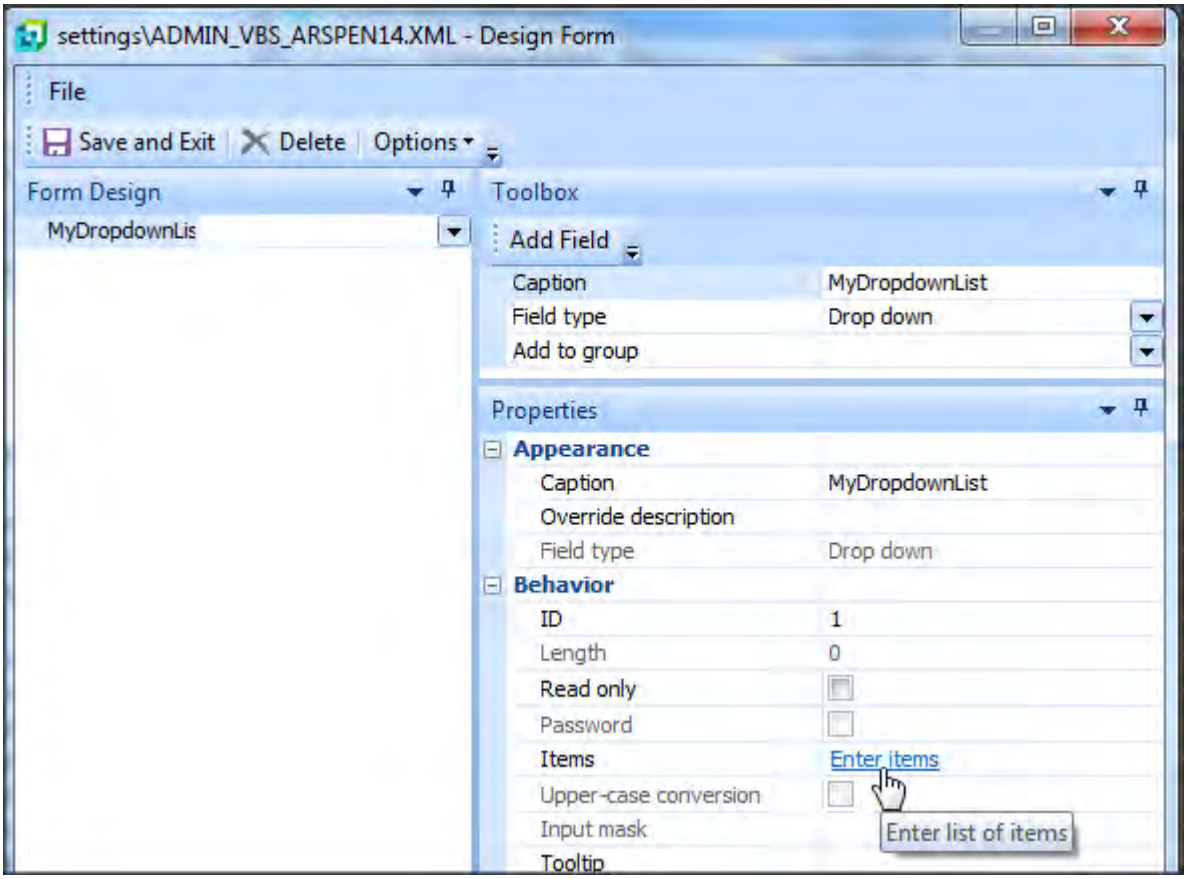


Figure 21-37: Adding the content to the dropdown list

This displays the *List Editor* screen where you can enter the items (see **Figure 21-38**). If you enter more than 20 items this list is truncated at 20 items when you click on the *OK* button.

However, there is a way to get more than 20 items into the dropdown list, but it requires using the customized panes' *OnRefresh* function. You can specify the full name of the variable representing the field followed by the *Field* element with a *List* attribute containing a semi-colon delimited list. A simplified example appears below, showing just three values:

```
TestFormXML.CodeObject.MyDropdownList = "<Field List='A;B;C'></Field>"
```

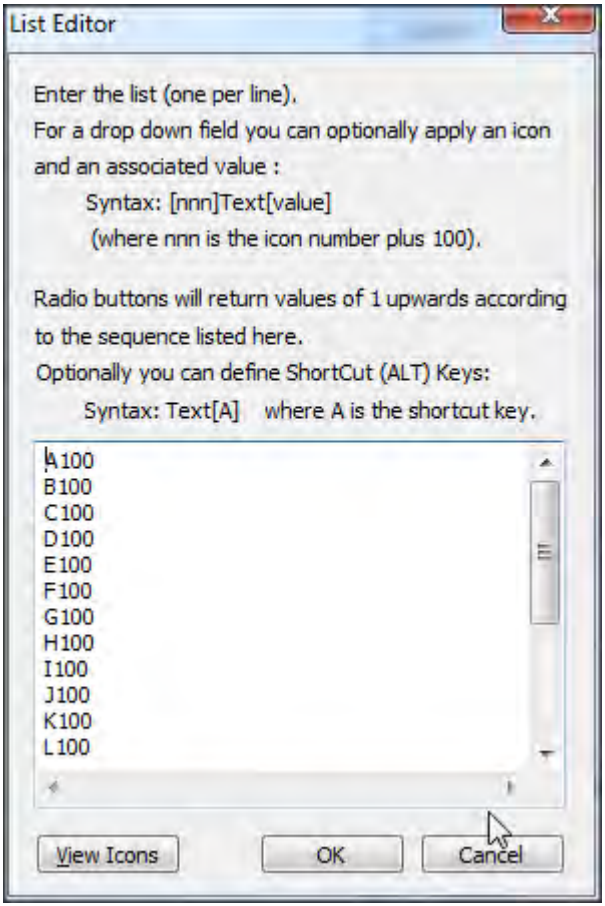


Figure 21-38: Using the *List Editor* to add the items for the dropdown list

An example containing 48 entries appears below. This is one line that has wrapped around multiple times:

```
TestFormXML.CodeObject.MyDropdownList = "<Field List='Auburn, WA;Auburn, WA
CY;Austin, TX;Bloomingdale, GA;Burlingame, CA;Carson, CA;Chino, CA;City of
Industry, CA;Dallas, TX;Denver, CO;Elwood, IL;Elwood, IL CY;Flower Mound,
TX;Glendale Heights, IL;Grapevine, TX;Greensburg, PA;Hanover Park, IL;Hebron,
KY;Kansas City, MO;Kennesaw, GA;La Crosse, WI;Lincoln, NE;Livermore, CA;Los
Angeles, CA;Los Angeles, CA CY;Louisville, KY;Louisville, KY CY;Maple Grove,
MN;Memphis, TN;Nashville, TN;New Orleans, LA;Northborough, MA;Ontario,
CA;Origin;Orlando, FL;Raleigh, NC;Redlands, CA;Richmond, VA;Saint Rose,
LA;Savannah, GA;Savannah, GA CY;Seattle, WA;Suwanee, GA;Swedesboro, NJ;Tempe,
AZ;Tracy, CA;Urbancrest, OH;Winchester, VA' > </Field>"
```

SYSPRO App Store

The *SYSPRO App Store* is available at http://appstore.syspro.com and contains many samples that can be downloaded and used at no cost.

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Every business is unique. Having chosen SYSPRO to run your business, you will inevitably feel the need to make small changes to suit your particular requirements. It could be in the way information is captured, how it is combined with other products, or simply how it is represented when you need to access it.

'Power Tailoring' is the best way to describe the extensive customization capabilities of SYSPRO. These range from changing the font of a caption to adding new fields, using SYSPRO's embedded VBScripting capabilities, embedding user-developed .NET User Controls and – last but definitely not least - getting your users to interact with SYSPRO.

Power Tailoring – A developer's guide is the second of a two-part series on power tailoring, and covers what can be achieved if you have development skills. If you know a little VBScript, it explains how you can use this to change the way that SYSPRO looks, change the way that it behaves, change the toolbars, or create your own customized panes. If you are able to develop a .NET User Control, it explains how this can be embedded in a customized pane, and how it can interact with other parts of SYSPRO.

Produced by: SYSPRO Press
Reader level: SYSPRO Developer

SYSPRO™
Simplifying your Success