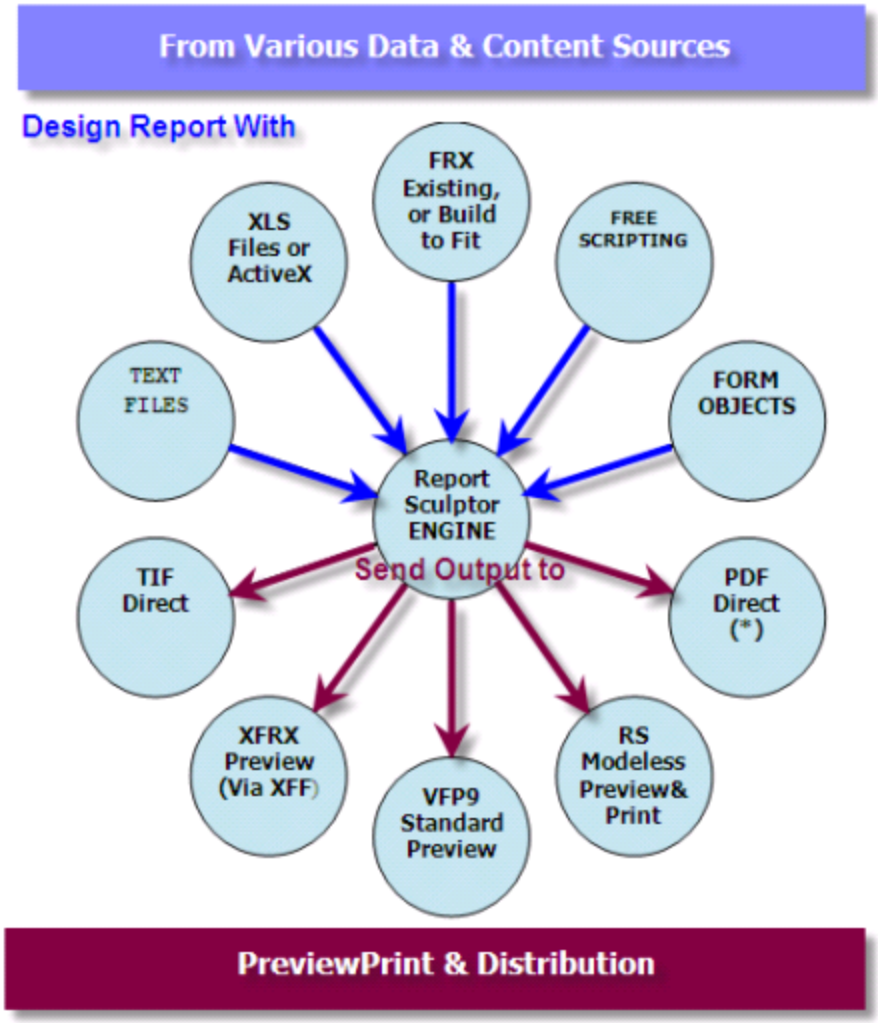
Version: 0.95 Beta Build
Last update: April 28, 2009

Report Sculptor Foundation

Free Reporting Framework for VFP9

Developer's Guide



By Djordjevic Srdjan
Limassol, Cyprus

* * *

About Report Sculptor

Project History / Mission

For years, Report Designer / FRX was only reporting tool native to FoxPro. Basic FRX concept was conceived back in DOS days, primarily as idea of reporting flow of sorted/related data and producing some hard-copy printed output. Although in fact very efficient concept, this (mandatory) alias-bound execution pattern unfortunately imposed certain design limitations on our reporting.

While this was not that much of a problem back in DOS times when we could code our reports around those limitations, it become real issue when we switched to Windows environment. At some point, FRX paradigm became almost like a *bird cage*, because there was simply **no way** of building reports almost any other way.

Along these years, industry requirements shifted in every conceivable respect, whereas FoxPro reporting capabilities changed in fact very little in the past 15 years. For instance, screen presentation of reports became very important point, which in many cases even overweights actual need for hardcopy printout, while our native Preview/Print facility barely changed look of it's buttons! Granted, we got better Gdi+ rendering and zooming, but native Report Preview remained just as monotone and gray looking today, as it used to look back in 90-ies!

As we all know, numerous requests by community for improving FoxPro reporting features and turning FRX into OOP environment were all turned down, and fingers pointed towards third party reporting tool market.
While third part reporting tools indeed started emerging, *it looked as time stood still for native FoxPro reporting*.
However third party tools focused mainly on end-user reporting and/or converting/exporting FRX to other formats, while again, everything was revolving around underlying FRX reporting pattern. In order to produce complex non-standard reports I stated using OLE automations and PDF libraries, which were at the time only way of circumventing these conceptual limits.

Result of these 'struggles' were my earlier reporting projects ,namely Rep2excel and CodeRep published back in 2002 and 2004 respectively. Later on as it came out, I adopted XFRX scripting and started slowly shaping up Report Sculptor using XFRX as underlying reporting engine. This framework was never published, because at some point, I finally realised that I cannot really pursue my ideas in full, unless I provide solid reporting engine and infrastructure myself. So I took plunge into VFP9/Gdi+x, and finally Report Sculptor saw the light of the day as completely free and independent project.
If you look historically, RS is by now 3rd or 4th generation of my own *free form* and *cross-format* reporting frameworks.

Report Sculptor Mission
When VFP9 came out , I was like many others very much excited about it, because major highlight and accent of the whole version was on new reporting engine with flexible and open architecture. However when it came to practical implementations, I eventually realised that very few things changed in area that was bothering me the most; Everything was still very tightly coupled with underlying data traversing pattern, without providing much in terms of **freestyle report creation** that I was after all these years.

However, one of the greatest *features* of new VFP9 reporting architecture was exactly that opennes and flexibility.
As someone said, Microsoft completely 'blew the lead off' the reporting engine, and openned up endless possibilites.
Community was invited to use these possibiltities, and this way extend and improve FoxPro reporting. One of authors of new VFP9 reporting made comment like; *We did what we did with VFP9 reporting, now you take that ball and roll it...*
Well this is exactly what Report Sculptor Framework is trying to accomplish;

Mission and Purpose of Report Sculptor Foundation Project is to provide infrastructure and building blocks for future developments of reporting tools by and for FoxPro community.
Major goal of this project is Fully OOP Reporting, CrossFormat, Open, Extensible and Free of all known constraints.
With Idea to improve, enrich and extend our reporting and bring it up in line or even exceed prevailing industry standards.

Core of Report Sculptor is independent reporting engine, which is surounded by various content interpreters on report design side, and output handlers on Output & report presentation side. Together, they provide rich set of functionalities which range from primitive page drawings, through powerfull Print-as-You-Go scripting, and reaches all the way into true OOP reporting.
There are several new reporting concepts introduced and made possible by RS, which I hope you will find usefull in your every day life and also improve / extend on your own.

At first, RS core engine is class object which you can subclass and extend it to do literally anything you want.
But not only; By enabling our ordinary foxpro controls to be used in reporting context, and bringing master control over report driving and execution to our code, RS creates possibility to aproach our reporting in a true OOP manner.
So, just as we have built our classes to make our life easier with forms, we can now also build real classes for our reporting.

To top it all comes again fully customizable (released in source) **RS Live Preview/Print** facility, which is now completely overhauled to support zooming, search, content navigation, exports to TIF/PDF, hook methods for email support etc.

I hope you will enjoy exploring all these new possibilities, and make the best out of it in your every day life.
Project development continues.

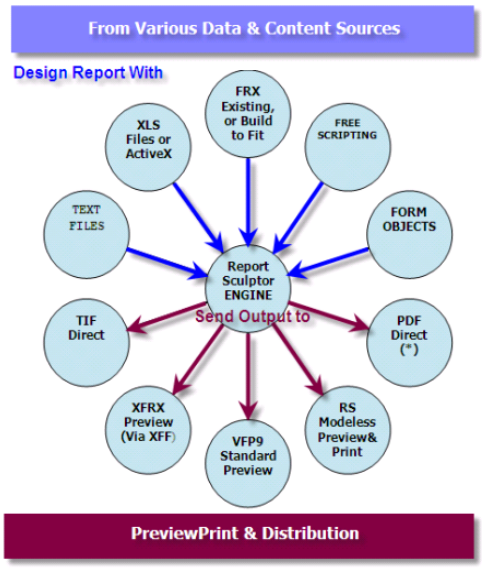
Enjoy using Report Sculptor :)

Djordjevic Srdjan

Founding Author of Report Sculptor Framework
Limassol, Cyprus

Overview

Core of Report Sculptor is independent reporting engine based on GdiPlus(X) and VFP's fast cursor engine, designed from ground up for flexibility and extensibility. RS Engine is class object acting as 'proxy' or 'device', capable of receiving simple drawing instructions so called 'reporting primitives' and recording them in memory cursors as actual pages.



Those drawing intructions can be sent directly by calling engine object methods, or can come from higher level objects such are File Interpreters, FRX Interpreter, Form Objects, Scripts etc. Results (pages) will be all sustained in memory cursors, and then they later shown to the user for Preview/Print and/or exported to various supported output file formats.

Idea is basically 'Report from anywhere and export everywhere'. CrossFormat, meaning that next to standard databound FRX reports, you can now use various new *content sources* and combine them all together into single report extraction.

Theoretically, anything can be Content source, provided that there is interpreter object written, capable of bringing it into our report session, and any file format can be output destination, if there is output handler written supporting it.

When it come to output, I already built and attached few important output handlers myself, such are native VFP9 report preview feed, RS Live Preview output and also exports to PDF*,TIF and XFF**.
[PDF Provided by using Dorin Vasilescu's wrapper class for ultrafast Haru dll library written originally by Takeshi Kanno]

Since Report Sculptor focus is primarily on improving report design side, I did not see any merit in developing more export types, considering the fact that RS can be directly ported to XFRX, for additonal output file formats.
Therefore if you need to support more file export types, I strongly suggest purchasing XFRX, which is tool specialised in converting reports to broad spectar of common file formats. This is the tool I am still using myself.
Report Sculptor was originally designed on top of XFRX scripting, so XFF intermediate format is directly supported as output type, making it possible to seamlessly export anything you do with RS, to all file formats supported by XFRX.
(If you are XFRX licenced user that is)

Reporting With Report Sculptor

Back to report design side;
So what is conceptually new here ? First and foremost, instead of being tied to only FRX with it's driving alias, here it is really you who runs the show! You can use variety of design elements, objects, functionalities and/or file formats as design 'assemblies'. Additionally you can expand this as much as you want, by subclassing RS engine and adding more power to it, and also build your own classes which produce specific or general purpose reporting contents.

So, Report Sculptor is not in any way replacement for FoxPro native FRX/RD. Instead Report Sculptor is platform which makes it possible to further expand our reporting capabilities beyond standard FRX reporting. It also makes FRX reporting to be much more usefull and powerfull, being that it can be now combined with various other things.

In this context, FRX becomes just another design element next to meny other exciting concepts and possibilities.
What brings it all together is RS engine, with OOP script being direct way to 'talk' to it. Aldough very powerfull, Report Scripting is not main idea of Report Sculptor. In this context scripting has more of a binding role.
Consider it just as very powerfull way to direct the 'show'. You will mostly use it to call inn what you want and in order you want it to be presented on report. Be that FRX or part of Excel File or Text File, som code, some object etc.

Power & Flexibility

Unlike with FRX processing engine, where everything is happening 'on the fly', in a single run through data, FRX report layout and adjoined listener chain - RS engine is always in 'wait state' , simillar to let say, active form or menu.
This 'wait state' is what makes whole world of difference! It alows us to freely add and combine as many reports and report parts as we need. But not only.
Those 'parts' are not necessarily just FRX-es files, but they can be as I mentioned many other report content sources.
Objects in charge for bringing various content sources into our reporting session are so called 'content interpreters'.
So we have FRX interpreter, Text Interpreter, Form Object interpreter and so for. They are all separate objects called by engine to perform specific task when you ask for it.

RS Engine itself is completely neutral as to from where drawing intructions are sent, or what is going to be final destination of reporting content. Role of RS Engine is simply to record those drawing instructions into memory cursors and sustain them there, until writing session is finished and report is ready to be shown to the user.

What is written inside memory cursors are lowest level reporting primitives, which only when executed together can represent page in meaningfull form. So pages are basically 'shreded to bits' and preserved in this granullar state (as records) until writing session is commenced.
From there and on it will be 'Output handlers' job to create meaningfull page representation again.

Extensibility
This separation between initial page writing session and later page (re)interpretation makes whole concept very extensible and flexible. So, just as there are no preconditions as from where writing of content can come, the same applies for final output. Enabling RS for another file format is matter of adding another *output handler*, just like enabling some file format to become valid 'content source' is is matter of adding another *content interpreter* object.

So with this multi-tiered architecture with core engine in the middle (see RS schema picture again) we get maximal extensibility on both input and output side, and also unified class interface for simplicity of use. All surrounding objects are invoked, used and then disposed by engine on demand, while from developer's standpoint entire RS Engine appears as simple flat set of methods to a single class object.

CrossFormat
Manifestation of this conceptual flexibility and extensibility can be observed if you let say take chunk of excel file and parse it through RS engine. By passing proper parameter to .Output() method that excel will be directly exported into TIF or PDF for instance.

```
=Xls2Rs(cXlsFile,cRange,4)   &&Excel range > TIF
=Xls2Rs(cXlsFile,cRange,5)   &&Excel range > PDF
```

This at this point just *interesting possibility*, is why I coined term *CrossFormat Reporting* in relation to RS reporting. However this term is not to be taken literally. It does not mean that RS is aiming to become universal File2File converter, instead it represents an idea/prospect of combining multiple content sources together into a single report, and then consecutively exporting result into any format we want. To me personally, this is really fascinating concept, and I will certainly try to expand possibilities in this respect.

Report Presentation & Distribution
Once writing session is closed, final interpretation of paged content will be delegated to respective Output handler object to be finally materialised into some meaningfull data presentation. That data presentation can be eighter some Preview/Print facility or export to some of commonly used file formats.
Report Sculptor up to now have 5 basic output types. These are in the same time numeric parameters (1-5) passed to .Output() method of RS engine. First 2 types are Report Preview facilities, Type 3 is actually both (xff intermediate format shown in XFRX Preview) while 4,5 are physical file formats;

1) RS Live Preview / Print
2) FoxPro9 Native Preview/Print

3xx) Xff / XFRX Preview Print (For XFRX licenced users)

4) TIF (GdiPlusX)
5) PDF (Haru PDF wrapper class by Dorin Vasilescu)

3xx **Conversions to file formats via XFRX commercial tool
(Applicable for XFRX licenced users only!)
---
301 PDF
302 Word
.
See Output method syntax for full list

RS Live Preview/Print

Highlight of this version is by all means improved preview/print facility. RS Live Preview & Print facility is now radically improved to support zooming, report navigation, report search and many other features which our users now days expect to have while viewing reports.
Unlike before you can now easily customise and localise this facility. RsDefaultViewer.scx form is now called via engine property, so all you need to do is make copy of it, customize it as you wish (as in translate to your own language) and then tell engine to use your own form instead. More on that later in this document.

Setup And Installation

ReportSculptor is normally shipped as two distinct ZIP files. They are named RsDev.zip and rsServer.zip respectively. First zip contains demo project which is intended to be your development time playground for RS reporting, while second one is conviniently packaged server deployment, which you will simply drop on your shared server along with your application files for future run time deployment.

Playing with ReportSculptorDemo project

So, to immediately start playing with your new toy, unzip rsDev.zip on your development machine, presummably as C:\rsDev After you have done this, open your VFP9 and look for project ReportSculptorDemo.pjx located in that folder. After project is openned, go under project tab 'Code' and run prg

C:\rsDev\progs\ **myRS.prg**

*Note
Remember to always run this prg first. When executed, this prg will set you up for running all samples contained in this demo project. This way you will avoid errors happening when code/forms are run without proper environment being set.

Although set as **main prg** of the project, do not try to compile project into exe or app and then run it, since this project was not designed for such thing.

Sole purpose of this project is to be your VFP Development time PlayLab, and nothing else.
After going through and running all form and code samples, you will eventually use this project as solid source for our favourite Copy/Paste practices. You will just grab some existing example, copy it somewhere in your project, then twist it a bit and turn it into your own report. There is around 20 sample forms at this point and there will be hopefull more published. Eventually you will build your own pool of reports and hopefully even report class libraries to streamline your reporting with RS.

Run Time Deployment

*** To Deploy at run time / Compile Your application

1) Unzip **rsServer.zip** alongside your application (can be in the same folder) . So for example, if your application is deployed somewhere on the server as

\\servermachine\myAppFolder Unzipping should result in creating ;
\\servermachine\myAppFolder **\rsServer**

And that is all you need to do on server side. I call it *dump once* deployment.
Hence, you could unzip rsServer folder on your own development machine right away, and then eventually simply copy the whole folder to your server machine. Or you can wrap it up in your own distribution (Inno) setup files.

To Compile Your application;

2) Add \rsDev\progs\dummy.prg to your project

3) Add \rsDev\libs\ReportSculptor.vcx to your project

4) Copy functions rsGlobalObjectSetup() / RsFrx() and engine subclass 'myRS' from \rsDev\progs\myRs.prg to some of your procedure files which are run time visible
(Together with your own common Functions and procedures, or at the bottom of your main prg itself)

5 At the beginning of your main module include RS Initalisation Sequence with proper paths supplied.

```
*********************  RS /  GDI+X Initialisation Sequence ********************************
Local cRSServerFolder,cGdiPlusX
cRSServerFolder = '\\MyServerMachine\myVFPAppFolder\rsServer'  &&supply proper path
do ( addbs(cRSServerFolder)+'ReportSculptor.App' )

***Now we also enable GdiPlusX application supplied in subfolder of rsServer.
***(system_lean.app can be kept elsewhere, but make sure version is 1.20 with EMF support!)
cGdiPlusX = Addbs(_oRSGO.RS_Root) + 'GdiPlusX\system_lean.app'
do (cGdiPlusX)
*******************************************************************************************
```

Now you are ready to compile your exe. DO NOT include RS app to your project at all. Once initialised (executed) as above, ReportSculptor.App will sets procedure to itself at run time, and everything will work. Your exe should not gain more then some kilobytes in 'weight 'as result of implementing RS.

Integrating ReportSculptor into your own environment

To enable RS in your own development environment (without going to rsDemoProject.pjx) , include RS Initalisation sequence (copy/amend initalisation sequence provided within **myRS.prg**) and place it in your own init routine.
Normally, I keep one prg called setup_dev.prg for each project, which initalises common procedures, paths and whatever else I use at development time. Something like this would be ideal place for putting this initalisation sequence. Path string in this Initalisation sequence can point to ReportSculptor.App contained in RsDev folder, (which you kept on your machine for future references) or directly to your server deployment folder. (Or bettter, some local test copy of your production site/data)
Since RS requires those few functions and at least one RS Engine subclass to be embeded to your source, you should also add those to some prg containing your common functions. This prg should be 'visible' at all times by executing 'set procedure to' it.

RS Global Object
It was mentioned earlier that for sucessfull deployment of Report Sculptor you need to add some functions and one subclass to your common library of functions (prg) which is 'visible' all the time Here is why;
In order to avoid dealing with INI files which could be in wrong path, missing etc, we now use single function call to set up all global variables which are conviniently grouped into - Global object.

RS Global Object is instantiated once you executed ReportSculptor.App during initialisation sequence. Reference to it is available via single public variable used/created by RS which is called ;
`_oRSGO`

Global object is used to handle and control all instantiated engine objects. It's properties (settings) directly affect instantiation of consecutive RS Engine objects used throughout your application. Those objects will directly obtain certain set of properties.

Using XFRX with ReportSculptor
For instance, in order to enable RS to directly export to XFRX for additiona file exports, we will simply amend global object settings via INI function which we previosely added to our common procedures. See below;

```
*********************************************************************************/
***  INI Function for global RS settings
*********************************************************************************/
Function rsGlobalObjectSetup
    with _oRSGO

        **Global properties applied at RS Engine instantiation

        .xfrx_active = 'N'  &&Change to 'Y' if you own and use XFRX
        .xfrx_path   = '\\SomeServerMachine\XFRX125'   &&Set your own XFRX run time path

        ***Subclassing of RS Engine
        .rs_EngineClass = 'MyRS'    && RS Engine Subclass used to customise RS Engine


    endwith

*********************************************************************************/
```

Global Object can be acessed and changed later at any point outside this INI function.
Here are some usefull usage tips;

```
_ORSGO.rsExportFolder  = cSomeFolder  &&Set destination for exporting files
_ORSGO.rsExportStem    = cFileNameStem &&Set name of export file stem (extensions will vary)
```

After that, wrapper functions such are Frx2RS() or ReportGrid() with canned reporting functionality will know where to send files and call them names you want.

*** Note
You probably noted oRSGO.Property .rs_EngineClass which contain name of current subclass of RS Engine you are using. Since RS Engine is an object, just as we did with our base classes - we subclass it right away and use subclass down the line. This will alow us to use multiple customizations of RS Engine for whatever purpose this might be used in the future.

Take advantages right away, Learn Later

In our world everybody wants things done as they say 'for yesterday'. This applies very well on our users, but also on us as developers. Since patience is rare virtue nowdays, {g} Let's see what is it from ReportSculptor that we can use and benefit from immediately, without getting deep into the bones...

Use RS Live Preview & Print with your existing reports

New RS Live Preview and Print facility is one of the first big advantages that you can use without much of a hassle. Normally we execute our reports using commands like

```
Report Form myReport.frx to printer Preview
**or lately
report form myReport.frx  object loReportListener ...
```

To take advantage of RS Live Preview/Print you can simply amend your report calling code as below

```
* Report Form myreport.frx to Printer Preview
=Frx2RS('myreport.frx')
```

This function will intercept FRX output and send it to RS Live Preview for further viewing, consecutive exports, emails etc. See RS Live Preview chapter in this document for all benefits it has to offer.

Export FRX to various file formats

To get TIF or PDF version of your report, all you will have to do is supply few more parameters to the same function;

```
=Frx2Rs('myReport.frx' , 4 , .t.  )  &&to TIF with Preview

=Frx2Rs('myReport.frx' ,  5 , .t.  )  &&to PDF with Preview
```

For licenced users of XFRX, list of output options is much bigger, starting from

```
=Frx2Rs('myReport.frx' ,  3 , .t.  )  &&to XFRX Preview

=Frx2Rs('myReport.frx' ,  302 , .t.  )  && to Word via XFRX
**From 301 to 315 are file exports via XFRX
```

See .Output() method syntax for full list of output types

Since this function is very likely to be used let see full syntax;

Function Frx2RS

Parameters cFrx [nOutput,lShowPreview, cFor,cWhile]

cFrx - Name/Path of FRX you want to run

***Optional

nOutput - Ouput Type Default is 1 for RS Preview (See .Ouput Method for full list of ouput type numbers)

lShowPreview - Show exported file with default preiewer - default is .f.

cFor FOR clause for FRX execution
cWhile WHILE clause for FRX execution

Next in line of things which are extremely easy to use are functons

ReportGrid() , ReportPageFrame()

As it's name implies, ReportGrid() is used to create simple listing report from our regular grids. Many of our forms contain grids and in many cases we need to build report which will print whatever user sees in grid to the printer.
To your aid comes function ReportGrid() which can turn all your gridss into run time resizable simple reports.

ReportGrid()
Parameters oGrid , [nOutputType , lShowPreview]

Creates simple report listing which resembles in full your actual grid

oGrid - Object Reference to the grid you want to form grid

nOutputType - Ouput type (Default is 1 RS Live Preview)
lShowPreview - Show Preview / Open Exported File

Returns cFleName / or .t. / .f.

ReportPageFrame()
Parameters oPageFrame , [nOutputType , lShowPreview]

Creates report which replicate each individual pagerframe page as one report page. This can be very usefull for creating reports with prefixed number of pages in fully WYSIWYG mode. Simply create number of pages in your pageframe fill them up with content (Textboxes,labels,lines,shapes,grids,ActiveX etc) .
Note: You can use native FoxPro controls or any framework controls you are normally use.

oPageFrame - Object Reference to the PageFrame you want to cast as report

nOutputType - Ouput type (Default is 1 RS Live Preview)
lShowPreview - Show Preview / Open Exported File

Returns cFleName / or .t. / .f.

Moving on with ;

Basic Concepts

First point you have to understand about Report Sculptor is that it's report engine is fully fledged object! In order to easier understand things , consider it simply as 'device', So as you would do;

```
Set Device to Printer
Set Printer ON
```

And then start issuing writing commands to printer or screen, hereby you instantiate RS Engine object

```
local oRS as myRS
oRS=GetRsObject()
oRS.OpenSession()
```

From this point and on RS Engine object is loaded in memory as idle datasession object awaiting your writing instructions. So since we have device on we start writing *things with it;*

```
with oRS
   .DrawString(100,100, 'Hello World!')
   .
   .
   .
```

And so for. Once we are done drawing, we will simply 'Close Device' and call engine method to send output to actual printer or other device.

```
   .
   .
   .
endwith
oRS.CloseSession()
oRS.Output(2 , .t.)    &&Send To Native FoxPro Preview
```

Code Structure Sequence

1) GetRsObject() 2) .OpenSession() 3) .CloseSession() 4) .Output()

***** Important Note.**
Above presented order of events is mandatory. Code sequence should be always exactly 1-2-3-4.
Any other variation (1-3-2-4), ommision (1-2-3) , or repetition (1-2-3-4-4) will get you eventually in trouble.

So at first we instantiate object by using factory function GetRsObject(), then start writing session by issuing .OpenSession() and then we start sending writing commands or calling inn reports and report parts.

Once we are done writing we will close writing session by issuing '.CloseSession()'
Last and only method call should be always .Output() ,which will then send report to desired *Output type*
and release all engine dependancies. Just as we would issue 'Set Printer To' and this way actually release previous writing to printer or close file we were writing into.

However, unlike having 'vanila' printer driver as device where you basically talk to OS proxy component by issuing primitive drawing instructions, here you are 'talking to' pure FoxPro object that you have almost full control over. But not only; Rs Engine is rich object supercharged with powerfull drawing methods now being all under your disposal. Additionally you can also subclass that object it as you wish to better suit you reporting requirements.

Setting up Report

As we normally do with our objects, hereby we control our RS Engine object by setting properties and/or calling some object methods. There is various aspects that can be very effectively controled by setting apropriate property or issuing apropriate mthod. Let start with most important ones;

Report Layout
Report is usualy preset for one particular paper size and orientation. So this is by all means first two properties you will need to learn. As they invented Copy&paste you don't really have to memorise this but but just be aware of it.

```
oRS.rsPageFormat='LETTER'    &&Default 'A4'
oRS.rsPageOrientation='LANDSCAPE'     &&Defualut = 'PORTRAIT'
```

Supported Paper sizes
(Property values for **rsPageFormat**)

A3,A4,A5,LETTER,LEGAL,FOLIO,11X17,STATEMENT,ENVL_MON

These properties should be normally set prior creating first page , which is normally triggered by .OpenSession() method. Therefore these two proprties should be naturally set up front. In case we don't set them directly prior creating first page, RS engine will assume defaults (A4/PORTRAIT)

Next to already supported page sizes we can also add our own cstom page sizes by issuing method calls like this.

```
.AddPageLayout('MYCUSTOM' , 'PORTRAIT', 1000,800  , _oRSGO.rs_root+'\reports\myCustom_P.frx' )
.AddPageLayout('MYCUSTOM' , 'LANDSCAPE', 800,1000, _oRSGO.rs_root+'\reports\MyCustom_L.frx' )
```

In order for this to work, you will need to create aproppriate pair of dummy FRX-es with these pixel dimensions preset. Tip: Use **RS_A4_L.frx** dummy frx template provided under \rsServer\Templates\Frx to produce your own custom paper size dummy FRX-es. Place them under templates\frx folder and then use it as shown above.

Another property important for this initial set of events and this is

```
oRS.lAutoFirstPage=.t. &&Default .t.
```

If this property is set to .f. then .OpenSession() method will not add initial page, assuming that we are going to do it at some later stage. This is necessary in particular situations, so this is also something you should be aware of.

Next set of commonly used properties are for enabling and setting default PageHeader and PageFooter.

```
**Enable
oRS.lAutoPageHeader=.t. &&Default .t.
oRS.lAutoPageFooter=.t. &&Default .t.
```

If we build **form based reports** then we can also assign directly form containers which will be used as automatic PageHeader/PageFooter.

```
oRS.oPageHeader = Thisform.Container1
oRS.oPageFooter = Thisform.Container2
```

Bear in mind that there is also Default PageHeader/Footer available at engine level, and you can also customize them by subclassing RS Engine object.

There are more properties which play the part in controling RS engine but most of them are used by engine itself and you will very rarely need to know and use more then what is exposed above.
Full set of usable engine properties will be supplied later in this document

Another property that might be of good use is

```
.PgRemainder   &&Remainder in pixels
```

Since free OOP reporting is done following 'write-as-you-go' philosophy this property will come usefull when you want to keep content together. Based on this property value (number of remaining pixels to the page bottom) we can use method

```
.Eject()
```

to force RS engine to continue on new page.

So now that we rounded up real basics, let see couple of working examples;

Example 1

```
**Instantiate RS Object
local oRS as rsEngine
oRS = GetRsObject()    && (1)

**Set up report layout
oRS.rsPageFormat='A4'
oRS.rsPageOrientation='PORTRAIT'

with oRS
    .OpenSession()  && (2) Open writing session

    .lw(25, 'Hello','Arial',14,4,4, rgb(255,0,128) )
    .lf(2)  &&Two Line Feeds
    .lw(25, 'World','Arial',14,4,4, rgb(255,0,128) )
    .lf(3)
    .hl(25,700)

    .CloseSession()   && (3) Close Writing session

endwith
oRS.output(2 , .t. )  && (4) Send to desired Preview/Print or FileExport format
```

As you have noticed I used couple of new methods which I did have chance yet to describe, but I believe you can aready get a picture. Methods .LF() / .LW() are counterparts to DOS style commands ?/??.

However we will not stay with them (this is dark ages anyway) and rather move to more productive kind of report scripting using visual objects. As you have read before RS supports and provides possibility to use visual objects from within report scripting so here is another scripting example that illustrates use of scripting combined with visually composed elements (Forms & Form Objects)
We will assume that you created one blank foxpro form and oppened table 'Customers' using form DE.
Also on the same form you will create 3 white borderless containers and call them respectively ;

ListHeader
ListDetail
ListFooter

Drag and dropp fields from DE on that form and then place all labels to ListHeader container and also all textboxes to ListDetail containers. After that you can put folowing code anywhere in your forrm;

Example 2

```
**Instantiate RS Object
local oRS as rsEngine
oRS = GetRsObject(thisform)
**Set up report layout
oRS.rsPageFormat='A4'
oRS.rsPageOrientation='PORTRAIT'

with oRS
    .OpenSession()  &&Open writing session (Mandatory/Once Only)

    .FlashContainer(thisform.ListHeader)
     select customers
     scan
        .FlashContainer(thisform.ListDetail)
     endscan
    .FlashContainer(thisform.ListFooter)

    .CloseSession()   &&Close Writing session (Mandatory/Once Only)
endwith
**Send to desired Preview/Print or FileExport format
oRS.output(2 , .t. )  &&VFP9 Preview
```

Run form and see what happens :)

Creating Navigation Tree

Report Sculptor in general, aims at elevating user experience while using our reports. In this respect RS LIve Prevew/Print facility introduces treeview as navigation aid for our report content. To set up navigation tree is very simple by using few engine methods designated for this purpose. General philosophy when creating navigation tree complies with 'Print-as-you-go' philosophy. So at first we create Root node which is mandatory;

```
oRS.rsDocumentTreeRootNode = 'Products By Category'
```

and then during report run we simply add nodes on the fly

```
.AddNode('Beverages')
.
.
```
.Write some detail records, and then again
```
.
.
.AddNode('Group2')
```

And so for.

RS Engine will automatically record current page coordinates for subsequent navigation through report.
If Navigation tree need deeper structure (in case of multilevel grouping) then we can use folowing code logic;

```
.
.
.AddNode('Condiments')
.AddNode('Confections')    &&This is the node we want to expand;

            **Expand tree ->
           .BranchDown('Chocollates')
              *detail
              *detail
           .AddNode('Cookies')   &&The same level as 'Chocollates'
              *detail
              *detail

   .BranchUp()     &&  <-- Decrease Level (Jump one level up)

.AddNode('Dairy Products')     &&The same level as 'Condiments' and 'Confections'
.
.
```

Note*** Report Navigation will work only within RS Live Preview / Print Facility.

Internal Report Bookmarking

Not available yet

Report Output and Distribution

One of RS Engines methods that you will be using the most is by all means .Output() .
Therefore it will be explained here in in more detail. So let's start with syntax first ;

.Output()
Sends all pages spooled within RS Engine object to selected output device, preview facility or export to some file format

Parameters nOutputType ,lShow

lShow - Show In Preview mode / Open generated file with default application
nOutputType - Number representing preview or export file type. See below lists for values accepted

Returns
cFileName - path/name of the file generated or .t. / .f.

* * *

RS Basic Output Types List

nType	Output Goes to	File Format
1	Rs Preview	None
2	VFP9 Preview	None
3	XFRX Preview	XFF***
4	Multipage TIF	TIF
5	PDF File	PDF

*** **Extended Output Types List**
Applicable for licenced XFRX users only

301	PDF PDF
302	DOC Word Document
303	FDOC Word document with flow layout □
304	RTF RTF document with absolute layout □
305	FRTF RTF document with flow layout □
306	HTML HTML document
307	MHT HTML document with all graphics included
308	XML XML Document
309	XPS XPS Document Format
310	XLS Excel document
311	PLAIN Plain text document
312	ODT OpenOffice Writer document with absolute layout
313	FODT OpenOffice Writer document with flow layout
314	ODS OpenOffice Calc spreadsheet with absolute layout
315	FODS OpenOffice Calc spreadsheet with flow layout

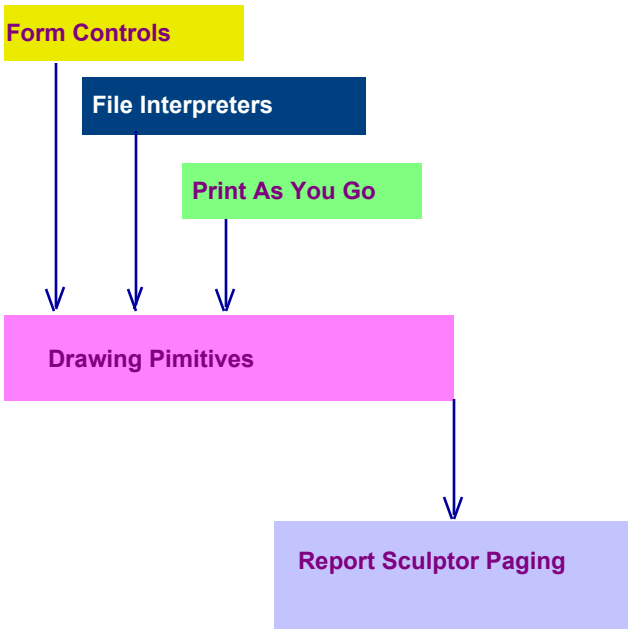
In addition to all this, RS Live Preview Facility also supports interactive exports to PDF/TIF with posibility to email this output directy from preview form.

Report Sculptor Engine Methods

Methods calls in further text called also 'Script' methods are generally divided in 3 categories.

1. Reporting/Drawing Primitives
2. Print-As-You-Go / Line by Line reporting support
3. High level scripting (Content Interpreters)

See picture on the right where they are presented in top to bottom order;



At very bottom we have real drawing primitives, which are used to describe pages in memory. Each page can be described very precisely by distinct number of theese drawing primitives. That is very core of ReportSculptor scripting and engine itself. On top of this all other scripting methods are built using these very methods calls.
So any higher level script method is just code construction that produces one or more of these primitive drawing instructions, which put together produce desired report result.
(Example; Drawing textbox off your screen form.)

Drawing Primitives

.DrawString() Draw string
.DrawText() Draws rectangle filled with textt
.DrawLine() Draws line
.DrawBox() Draws Rectangular Shape
.PaintBox() Paints Rectangular Shape
.DrawPicture() Draws Bitmap
.DrawArc() Draw Arc
.DrawShape() DrawShape (with rounding)

Print-As-You-Go

Next level of scripting is support for 'Print-as-you-Go'. So it is sort of 'automation' of issuing above mentioned drawing primitives to mimick those old ?/?? , eject etc reporting commands which were used to in DOS coded reports. So we have

.lf() && Line Feed
.eject() && Eject Page
.lw() && Write in current line, left to right.
.hl() && Draws horisonatal line floating at current vertical position
.ltx() && Sort of like inline Textbox which can present properly aligned text and numbers
But also Visualy designed elements;
.FlashContainer() && Replicate form container at current page position

High Level Scripting

This category of scripting will be very wide collection of various report content generating methods. At the time been RS replicate visual controls, text files, long memo content (as is this page!) excel file ranges, include entire FRX files etc. This area will be hopefully greatly expanded in the future.

- Replicate FoxPro Native Controls
- Replicate external Files or file parts
- Interprete FRX based reports
- Custom content writing classes (Bulleted Lists, Charts, Gauges, Text Styling etc.)

Drawing Primitives ..... DrawString() , DrawText() , DrawLine() , DrawBox() , PaintBox() , DrawPicture()

DrawString()
Draws string on given page position.

Parameters
nTop,nLeft,cExpression [cFontName , nFontSize, nFnStyleNo ,nFrgb ,nRotate]

nTop,nLeft - Top and Left Page Coordinates of string
cExpression - String To Be Written

cFontName - Font Family name
nFontSize - Font Size

nFnStyleNo - Number representing desired font specs in terms Bold/Italic/Underline etc.

0 = Normal
1 = Bold
2 = Italic
4 = Underlined
128 = Strikethrough
*** Composite Values
3 = BoldItalic (1+2)
5 = BoldUnderlined (1+4)
6 = ItalicUnderlined (2+4)
7 = BoldItalicUnderlined

129 Bold+StrikeTrough (128+1)
etc

Note: At the time been, not all font specs are posible for all
output types. Use combinations which ive you consistent results.
This will be however improved in the future.

DrawText()
Draws block of text within rectangle defined by 4 coordinates

Parameters;
nTop,nLeft,nDown,nRight,lcText [,cFontname,nFontSize, nFnStyleNo , nAlign , nFrgb,nBrgb ,nRotate]

nTop,nLeft - Top and Left Page Coordinates of bounding rectangle
nDown,nRight - Down and Right Coordinates of bounding rectangle

lcText - Text to be written

cFontname,nFontSize - Font Family name / Font Size
nFnStyleNo - Number representing desired font specs in terms Bold/Italic/Underline etc. (See DrawString for details)

nAlign - Text Alignment mode
* 0 && Left Align
* 1 && Right Align
* 2 && Centered
* 3 && Full Justify

nFrgb - Fore Color of text [Use rgb() function to pass parameter]
nBrgb - (Back)Color of bounding Rectangle [Use rgb() function to pass parameter]

nRotate - Rotation Angle

DrawLine()

Draws line between two points (TopLeft & DownRight)

Parameters; ntop,nleft,nDown,nRight [, nPenSize , nForeRgb , nPenPat]

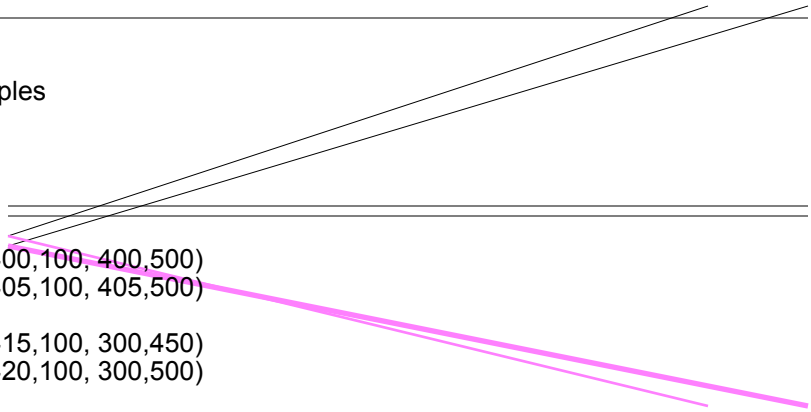
nTop,nLeft - Top and Left Page Coordinates
nDown,nRight - Down and Right Coordinates

nPenSize - Pen Size
0,1 Hair (Default 1) ,2-8

nForeRgb Color as RGB Number [Use rgb() function to pass parameter]

nPenPat Pen Pattern
8 = Solid [default]
1-7 (See FRX help)

Usage Samples



.DrawLine(400,100, 400,500)
.DrawLine(405,100, 405,500)

.DrawLine(415,100, 300,450)
.DrawLine(420,100, 300,500)

.DrawLine(415,100, 500,450,2, rgb(255,128,255))
.DrawLine(420,100, 500,500,3, rgb(255,128,255))

DrawBox()

Draws Rectangle defined by 4 coordinates (TopLeft & DownRight)

Parameters: nTop,nLeft,nDown [,nRight,nPenSize,nForeRgb, nCurvature]

nTop,nLeft - Top and Left Page Coordinates
nDown,nRight - Down and Right Coordinates

nPenSize - Pen Size
0,1 Hair (Default 1) ,2-8

nForeRgb Color as RGB Number [Use rgb() function to pass parameter]

nCurvature - Roundation effect (not operational yet)

Usage Samples
.DrawBox(820, 20 , 880 , 500)
.DrawBox(822, 22 , 882 , 502, 2, rgb(255,128,255))

.PaintBox()

Parameters: nTop,nLeft,nDown,nRight , nColor

nTop,nLeft - Top and Left Page Coordinates
nDown,nRight - Down and Right Coordinates

nPenSize - Pen Size
0,1 Hair (Default 1) ,2-8

nColor Color as RGB Number [Use rgb() function to pass parameter]

Usage Sample
.PaintBox(300,25,360,300, rgb(213,213,255))

.DrawPicture()

Parameters: nTop,nLeft,nDown,nRight,cFile [,nAdjType,nRotate]

Bounding Rectangle Coordinates
nTop,nLeft - Top and Left Page Coordinates
nDown,nRight - Down and Right Coordinates

cFile Bitmap File/Path
nAdjType Stretch
nRotate Rotation Angle

Usage
.DrawPicture(400,400,700,800, '\reportsculptor\gdiplusx\graphics\climber.jpg' , 0 , 45)

Print-As-You-Go

.lf() , lw() , .ltx() , .hl() , .Eject() .FlashContainer()

.LF()

Emulates Line Feed

Parameters: [(+/-)nNumberOfRows]
Default is 1 row

It simply moves current vertical position downward (or upward) the page by number of rows.
Row is actualy just number of pixels defined in property 'rsDefaultRowHeight'

After issuing this command we can then use inline writing methods such are .LW(nAt) and .LTX() to write within that line. (See below)

Note

When issuing consecutive linefeeds within some loop and writing along as we go, at certain point, curent vertical position will come close to the end of the page. So when certain point is reached defined by property 'rsPageFooterAt', then engine will automatically execute default PageFooter method (if specified) and append new page.
On the new page, it will automatically execute PageHeader() meethod (again if specified).
Decision, weather engine will automatically execute PageHeader/Footer methods lies with properties; 'lAutoPageHeader' and 'lAutoPageFooter'

After new page is added vertical pointer is set at the top of it (or right below pageheader) and we are set to continue to write next page. And so on we write-as-we-go until the end of the report.

.LW()

Write string in current line at specified position left to right.

It is actual wrapper for DrawString() method, but because vertical position is already predefined it is enough to pass only one parameter for offset toward right. It emulates '?? ... cExpression at... '

Parameters: nAt , cString [, cFontname, nNontsize, nFnStyleNo , nFrgb , nRotate]
See sintax for .DrawString() method because this being wrapper means parameters are the same

Usage samples

```
.
.lf()  &&Advance one line down
.lw(50, 'Product' , 'Arial',12,1,rgb(0,0,255) )
.lw(330,'Unit Quantity', 'Arial',12,1,rgb(255,0,255))
.
```

Produces;

Product **Unit Quantity**

tip:
It can be very efficiently used from within text file or memo (as in this manual)

.LTX()

Writes justified text within current line.

Sort of like textbox placed within current line.
It is wrapper for .DrawText() method with 2 out of 4 coordinates being current (nTop,nHeight) .
So what needs t be passed is only left position and width of bounding rectangle.
And of course the rest of parameters for mentioned DrawText() method ;

Parameters: nleft,nWidth,lcText [,cFontname,nFontsize, nFnStyleNo, nAlign,nFrgb,nBrgb,nRotate]
(See details under .DrawText())

Usage Samples;
.
.
.ltx(80,70, str(Products.unitsinstock,8,2) , 'Arial',9,1,1, rgb (0,0,0), rgb (236,236,0))
.ltx(180,70, str(Products.reorderlevel ,8 ,2) , 'Arial',9,1,1)
.
Produces result like

1234.56 **789.10**
tip*** Combine with transform() function for better number presentation.

.Eject()

Adds new page to the ongoing report composition.

Parameters [cPageFormat,cPageOrientation]

.HL()
Draws horisontal line at current vertical position
(As one right below here)

Parameters nFrom,nTo [, nVerticalAdjustment]
nFrom - left point
nTo - to right point
nVerticalAdjustment - Float line few pixels up or down as we might need

*****And for most powerfull/productive report scripting**

.FlashContainer()

Replicate container from active form at current vertical position on the page,
and advances that position for the height of the container itself.

Parameters: oContainer [,nTop,nLeft]
oContainer - direct object reference to a container object

nTop Vertical position on page (if not passed assumes current page position)

nLeft Horisontal position (if not passed, assumes default left position stored in engine property .nColumnLeft

---

This method of writing alone is capable of cutting down code normally required to write scripted report by almost 90%. It is basically bridge between scripted and wysiwig reporting. So instead of coding every single dot or number to appear on report, we can design report parts visually (on form) and use them programmaticaly, providing for maximum precision while minimising code needed to produce report content.

See OOP reporting later in this document for more hints.

Content Interpreters

Report Sculptor engine itself is not responsible or capable of producing report content. This is done by surrounding higher level objects which I call content interpreters. These object are actually ones capable of producing ready made pages or page parts on demand, from some physical file content or som other entity. In this respect we have FRX interpreter which interprets output of FRX report execution into RS reporting session, then Text interpreter interprting plain text file from hard disc or memo field, then Excel File interpreter, then Form Content Interpreter and so for.

FRX Interpreter

Due to history of having FRX report designer as only tool for building reports in the past, most of our reports are actually done as FRX-es and this is not going to change any time soon. We need to use those reports as they are, and occasionally we want to merge more of them together into single report extraction. In addition to this RS introduces concept of combining FRX with other content sources such are text, excell ranges etc.

So RS engine method (actually method+function) responsible for incorporating result of FRX execution inside ongoing report composition is;

.LoadFrx(oRptPages) / together with function **.LoadFrx(rsFrx(cFrxFile))**
parameter oRptPages - Object produced by factory function as shown above right

.LoadFrx() method actually receives entire pageload from FRX execution as an object, which comes as result of special factory function that is used to actually invoke FRX execution.

Function rsFrx(cFrxName [cFor,cWhile])
Parameters: cFrxName [cFor,cWhile]
cFrxName - FRX file name
cFor - For Clause
cWhile - While Clause

returns oRptPages (Object required by .LoadFrx())

This little syntax hurdle is necessary because only way for FRX to see resources compiled within an exe (bitmaps) is if call to FRX is made from within that particular exe. Therefore folowing function is provided as surce, and you are to compile it inside your exe. So this way FRX will be executed and interpreted properly.

This might sound confusing at first, but usage is not really complicated at all. (See sample below)
When we put it all together, complete syntax looks like

Usage Samples

FRX-es can be called from any RS report writing session. One effective use can be seen below where it is used to effectively merge two FRX based reports and shows them together inside VFP9 preview.
Note*** This is otherwise impossible with VFP9 itself.

Code Example - Merging multiple FRX reports into singe report preview

```
local oRS
oRS = GetRsObject()
oRS.lAutoFirstPage=.f.
local cFrx1,cFrx2
with oRS

    .OpenSession()

    cFrx1 =   HOME() + 'SAMPLES\SOLUTION\REPORTS\' +  'ledger.FRX'
    cFrx2 =   HOME() + 'SAMPLES\SOLUTION\REPORTS\' +  'Colors.FRX'
    .LoadFrx( rsFrx( cFrx1              ))  &&Import result of frx for given scope
    .LoadFrx( rsFrx( cFrx2              ))  &&Import result of frx for given scope

    .CloseSession()

endwith
oRs.Output(2,.t.)  &&Foxpro9 native preview
```

Tip***

Frx can be included very effectively from text like this. See below;

* /& .LoadFrx(rsFrx(addbs(_oRSGO.RS_ROOT) + 'REPORTS\employees.FRX'))

If you remove asterisk in line above, entire report would appear directly as next page of this document.

Since most of the time we will want to use single frx, simplest way of using Frx within RS environment is via function call ! Rather then instantiatin RS object directly, opening session and then calling, it is much easier to call single function ;

```
=Frx2RS('ledger.FRX')
```

This fuction internally rises RS object, runs FRX properly and then directly (by default) calls RS Preview Print. However this function can do much more then just that. If we see fully expanded syntax

Function Frx2Rs()
Parameters: cFrxFileName , [nOutput] , [lShowPreview , [cFor] , [cWhile]]
nOutput , ShowPreview See .Output() Method
returns: cFileName (Name of export file created)

You will realise that this function is little powerhouse in itself .
For example you get to export FRX to PDF in a single line of code, or incorporate RS Preview within your application with very little effort.

To take advantage of RS Live Preview/Print you can simply amend your report calling code as below

```
* Report Form myreport.frx to Printer Preview
=Frx2RS('myreport.frx')
```

RS Smart Text

Plain Text files are something we all used. VFP is very good at reading and creating them. However including them into our reports was not all that easy in the past. With FRX you could present them by reading them into variable, but that was limited to current band you placed them inn, with possibly some some streach/overflow. Not much else we could do with them. With report Sculptor they become totally acceptable report content source.
Then can span to unlimited number of report pages, and they can be combined and enriched with scripting.
Report Sculptor will read file and then interprete it to report line by line. Prior executing simple HTML like text formatting tags it performs TextMerge() so we can easily merge our variables and table fields into it.

Additionally you can also use RS methods within text lines and it will get executed as they come along, among ordinary text lines. This way you can effectively blend pictures, draw graphics or even include entire frx reports into this running text.

Text Formating Tags

, , , <u></u>
Bold,Italic,Underlined and combinations

Scripting from within text

// , /&

//
Comment within texts which does not appear on report. It gets simply ignored

---

/&
Executes single line script call by attaching command to engine reference via macrosubstitution

/& .ScriptMethod()

Translates directly into

oRS.ScriptMethod()

Therefore entire set of RS engine object methods can be utilised directly from within 'RS Smart Text'

. SetFont() / .SetColumnLeft() / .SetTop()
.ImportFrx() .RenderControl()

Etc.

Note:

This opens up **whole new paradigm** in our standard FoxPro reporting.
This combination of **formatted text** and **live data reports** (don't forget Excel interpreter) ,
where basically flow of text lines is the what actually drives your report, could be very effectively used for producing complex company reports that include manual writings (hence accountant writing year end / performance reports).

That process in many organisation involves pooling out numerous live data reports, and consolidating them into some other document (Word,Excel etc) or presentation tools (PowerPoint etc) for further rounding up and publishing.

With this feature added to out FoxPro reporting, you can write effective applications to streamline this kind of activitiies.
This was of course just one example. I cannot possibly immagine all use cases this concept might have, but I am sure you will find some good use for it
Also note that writing this manual is done by using this very concept. It is just that material presented here are actually help pages, rather then let say ballance sheet commentary by accountant, or full scale EOY management report.

If native FRX paradigm gave us chance to produce complex databound reports, RS expands our ability now to use the same reports, for producing much more complex, hierarchically organised multipart documents.

Programming Text - An Example

Since we have ability to execute engine methods from within text lines we can now actually venture into text programming area. This is of course hard and complex matter which goes well beyond original purpose/scope of ReportSculptor, but even with this limited text processing and programming abilities we do have some 'guns' if need arise to actually create more text oriented report stuff.

Example 1 Wrapping 'Lorem Ipsum' around my head...

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nulla facilisi. Quisque dolor leo, sollicitudin a, porta vel, faucibus id, nunc. Suspendisse mollis nonummy tellus. Sed auctor pulvinar odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus olutpat dui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. onec lorem ligula, pellentesque ac, sodales at, ornare non, lacus. Vivamus rutrum aliquam leo. orem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. ***Sed feugiat mauris quis velit.*** Computer Freak urna. Vivamus volutpat ui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. onec lorem ligula, pellentesque ac, sodales at, ornare non, lacus. Vivamus rutrum aliquam leo. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, viverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus volutpat dui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. Donec lorem ligula, pellentesque ac, sodales at, ornare non, lacus.

Example 2 Presenting Text in multiple columns.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendr-erit urna. Sed dictum est in Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendr-erit urna. Vivamus olutpat dui vel est. Sed dictum est in Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, erit urna. Vivamus olutpat Sed feugiat mauris quis velit.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus,

purus. Sed feugiat mauris quis velit. Etiam iaculis hendr-erit urna. Vivamus olutpat purus. Sed feugiat mauris quis velit. Etiam iaculis hendr-erit urna. Vivamus olutpat dui vel est.

Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus olutpat dui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus olutpat dui vel est. Sed dictum est in metus. Nullam facilisis aliquet Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendr-erit urna. Vivamus olutpat dui vel est. Sed dictum est in Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus olutpat dui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. onec lorem ligula, pellentesque ac, sodales at, ornare non, lacus. Vivamus rutrum aliquam leo. orem ipsum dolor sit amet, consectetuer adipiscing elit. Proin lorem lacus, mattis et, cursus ut, iverra faucibus, purus. Sed feugiat mauris quis velit. Etiam iaculis hendrerit urna. Vivamus volutpat ui vel est. Sed dictum est in metus. Nullam facilisis aliquet turpis. Duis varius enim ut orci. onec lorem ligula, pellentesque ac, sodales at, ornare non, lacus. Vivamus rutrum aliquam leo.

Excel Interpreter

To include part of excel fiile into your report, you can use folowing call.

.RenderXLRange(oXL,cRange,nTop,nLeft)

Parameters: oXL,cRange [,nTop,nLeft]

oXL - Object reference to an open excel sheet. See Function OpenExcel()
cRange - Range of Excel file to be rendered
nTop,nLeft - Distinct coordinates. If these are not supplied range gets rendered at default or current positions. Beware that flow of excel rows can turn new page if space on page is not enough.

Code Example

```
local oRS
oRS = GetRsObject()
oRS.lAutoFirstPage=.f.

local oXL , cXLS
with oRS

    .OpenSession()

     cXLS = addbs(_orsgo.rs_root)+'templates\xls\employer_policy.xls'
     cRange='A1:J62'
     oXL=OpenXls(cXls)  &&Create an instance of excel with given file open

      .RenderXLRange(oXL,cRange)  &&Now pass that object reference with range to be rendered

    .CloseSession()

endwith

oRs.Output(2,.t.)  &&Foxpro9 native preview
```

Next to this there is also Excel Wrapper class which can be used when reporting from FoxPro forms in WYSIWYG mode. You simply drop object on container or pageframe for instance, fill up excel file name and range and it will get render accordingly. See form sample #11.

Also you can use single function call to directly read and report excel sheet range by itself

Xls2rs()
Parameters: cXlsFile,cXlsRange [cPageFormat,cOrientation,nOutput,lShowPreview]
cXlsFile,cXlsRange File name and Excel range.

Optional / Defaults
cPageFormat ... Default 'A4'
cOrientation ... Default 'PORTRAIT'
nOutput ... Default 1
lShowPreview ... Default .t.

Note

Excel interpreter relly on Excel OLE Automation to read excell range cells and their graphic layouts, so it should be used with caution to avoid speed degradation. Use it only where applicable, to your advantage rather then dissapointment :)

RS Live Preview

Customize Rs Live Preview Form

To customize RS Live Preview form, use copy of actual form called myrspreview.scx and then change colors graphics etc at first without touching code. If you feel confident, you can improve code as well, I will be glad to include any improvement made to this form and publish it in last version (1.00) scheduled for late summer this year. As far as design is concearned, defferent people have different visual preferences, therefore feel free to customize visual appearance any way you like.

To test, and later use this amended form, all you have to do is supply name of it to the RS engine property called

```
.rsPreviewScx

**To set your own form change as below;

.rsPreviewScx = '.\forms\MyRsPreview.scx'
```

Naturally the best place to do this, is in RS engine subclass that you have previosely inserted into your code, as instructed in deployment instructions (see readme.txt)

Once you have done it, include/compile this form into your own exe and use that one instead of the one I supplied as default. Classes for RS Default Preview form are contained in .\libs\rsprev.vcx so include this library to your project as well.

* * *

Don't be shy!

If you by any chance mess up any code, simply make brand new copy (save as) and retry til you reach your perfect design. You can always revert back to RS default preview form, which is compiled within ReportSculptor.app just by reverting above mentioned property back to it's original value. (Simply comment out that line)

<b>I would love<b> to see your variant versions, and if you don't mind please send them over to me via email.
In return, I will publish and try to award the best customizations.

So, competition for best looking RS Live Preview Form is as of now open :)

To be continued...

Developer Guide is incomplete. It will be hopefully finished and published with final version (V 1.00) planned for late summer.

Thank You for your patience