Smartist scripting

headquarters
Reg. Gurey Zona Industriale 5/bis
11020 Donnas (AO) - Italy
TEL.: +39 – 0125 – 812811
FAX: +39 – 0125 - 812858

DPSS Laser division
Via Oneda, 11
21018 Sesto Calende (VA) - Italy
TEL.: +39 – 0331 - 918001
FAX: +39 – 0331 - 918032

www.laservall.com

Published on:

Updates

Author	Topics of the new edition	Release date
Manul Lupato	1st draft	
Manul Lupato	Release	
Manul Lupato	Added section on Project.Close() function	25/01/2008
Manul Lupato	Adaptations to ship this manual with Smartust installer	13/02/2008

Preface

This document was realized with the goal to introduce to VB script programming using Smartist.
This document is not Smartist manual: please refer to it for information on using the program. The manual can be downloaded from Laservall web site in case you do not have one: http://www.laservall.com
This document focuses on scripting. Script allows Smartist to interact with other programs and to make its operation automatic.

Audience

Basic script programming skills are suggested. If the reader has no experience at all, an introduction to basic scripting is given.
More information can be easily found (and in greater detail) online. Check http://msdn.microsoft.com as a starting point

Structure

This document focuses on Smartist scripting interface; since its core technology is Microsoft ActiveX the first part is dedicated to understanding similarities and differences between script and ActiveX clients.
The second part goes deep into describing the documentation.

Feedback

If you find errors or would like to report suggestions, please email the author here:
swdept@laservall.com

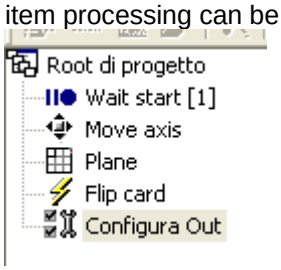
Smartist

Smartist is Laservall's program suite for industrial and ID marking. It includes a layout editor, a laser control program, a program for tuning the optical field and other applications.
Smartist main application is Laser Editor: this is used to edit the layout to be marked, control axis movement, build simple projects and, finally, write scripts using the built-in editor. Smartist and Laser Editor are usually interchangeable terms.
This is not Smartist manual. Please refer to Smartist manual if you have any doubt on the terms being used

Focus of this section will be illustrating the way the user has to make Smartist operation automatic

Building a project

Each Smartist project comprises a list of one or more ***project items***. Items are associated to specific behavior of the program: when they are *processed* the program performs peculiar actions. Some items react to external signals: since Smartist was born for industrial environment, this allows the program to interact with programmable logics (PLC) to realize simple program flows. Usage of project items is the simplest way to control program flow as item processing can be non-sequential.



Scripting

Scripting is an easy and flexible way to control item processing flow, modify layout, interact with other programs.

What's a script?

The distinction line between scripts and programs is that the first are written in interpreted languages, the latter in compiled languages. Example of compiled languages are C or C++: a *compiler* produces the binary code from the source files and the resulting program can work stand-alone. Examples of interpreted languages are VBscript, Jscript, PerlScript: the program need an *interpreter* on the host machine to work. The interpreter parses the script at run-time and perform the operation it requires.
Smartist is a **script host**: not only it includes a script engine that works as an interpreter, but exposes its own interface in that engine. The scripts executed in Smartist context can therefore access Smartist own script interface.

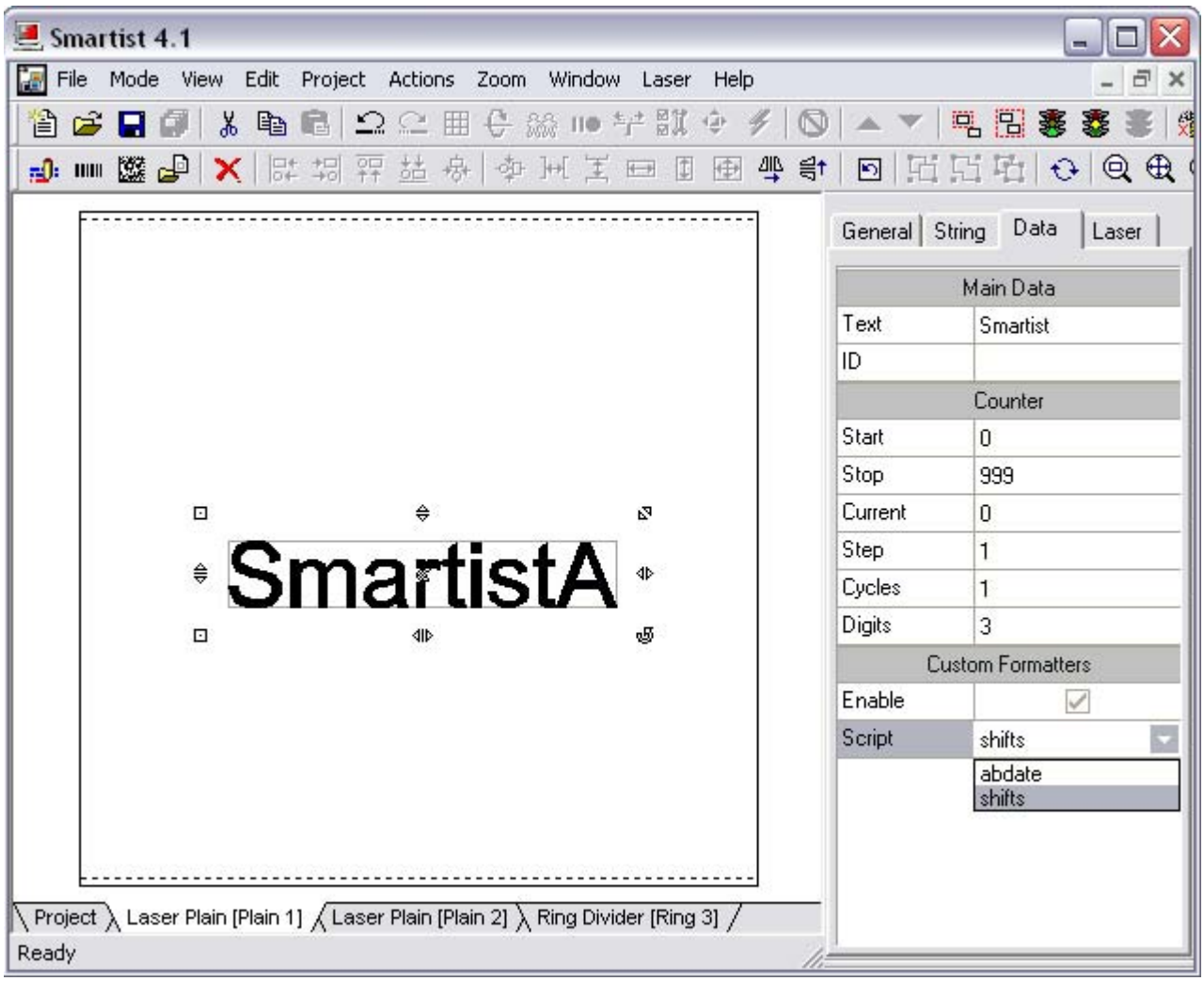
Adding a script to a project

To add VBScript to a project, the ***Project Customization*** tag must be enabled. An additional page labeled SCRIPT will appear to the side of the project pages.
A pre-set script can be entered using the SCRIPT combo box located in the properties window or a new one can be written.
The highlighting syntax is used to quickly display the basic elements that make up the script.
Any errors during execution of the script will be indicated by messages on the page.

Formatters

It is possible to apply customised formatting to single objects (text, bar codes or data matrix) using formatters. Formatters are simple VBScripts executed into objects context: they execute before a START command, thus being updated each time the object is sent to the laser print spooler.
It is possible to format an object by selecting the formatter as shown in the following figure.

PLEASE NOTE: formatters are not saved with the project as scripts: you need ti manually copy the formatter in the target machine into the “Formatters” folder.



The formatting scripts are saved in the folder ***../smartist4.1/formatters/*** .

The keyword *Text* can be used inside the formatter to access the text contained in the object.

ActiveX

ActiveX is a Microsoft technology that enables software components to interact with one another in a networked environment, regardless of the language(s) used to create them. An ActiveX server is a program that exposes one or more interfaces to the environment. Accessing those interfaces an ActiveX client can interact with the server.
Smartist in an ActiveX server; this technology is the core of scripting as well: the very same interface can be accessed by an ActiveX client and by a script (that is actually seen by the system as client). This is the reason the documentation of the interface is only one.

Documentation

The ActiveX documentation ships with the installer.
The document is extracted from the source code of the objects or method exported: being Smartist written in C++, the syntax of documented methods and objects is C++ and MFC (Microsoft Foundation Class). Script developers must be aware of some information here:

- ***Pointers:*** C-style pointers are not supported in VBscript. Functions using pointers cannot be used in scripts.
- ***IDispatch:*** functions returning pointers to IDispatch objects (or better: interface) translate into reference to objects (therefore they can be used in client applications)
- ***Constants:*** constants referenced in the examples must be re-implemented in scripts as they can't be exported in script context
- ***Dispatch and Events:*** objects interface has been divided in two. The Dispatch part exposes objects properties and methods; the Event part exposes the event only. This division is due to technology and implementation limitations.

Script limits

Since interface is unique for scripts and ActiveX clients one would think that it is useless to write a custom application if everything could be done with a script. Reality is that script has limitations that only custom application can get over:

- **User interface:** scripts can customize user interface in very limited ways; highly personalized interfaces require custom application
- **Loading projects:** scripts run in their project context, therefore they cant load a project file, only document files
- **Object destruction:** due to limitations in script language it is possible to create objects in laser layout but not deleting them
- **Pointers support:** functions using pointers as arguments cannot be used

ActiveX client

An ActiveX client can be roughly described as a stand-alone program realized in any language whose development environment supports ActiveX technology (e.g. Microsoft's Visual Studio or Borland Delphi and C Builder). The program statically "links" an interface to its ActiveX server at design time. At run-time the program queries Windows to see if its server is installed in the system[1]. If not the client fails, otherwise a session of the server starts (either visually or in the background) together with the client. The client communicates with the server trough the "linked" interface model.
Writing an ActiveX client is more complicated (as it requires greater programming skills) and more time consuming than writing a script. A custom application however offers more flexibility in designing the user interface and allows better functionality.

[1] The server usually installs itself as an ActiveX server during the installation procedure. If not running the program once should install it (the console command regsrv32 can be used either)

Choosing a solution

So far it should be clear that neither scripts nor ActiveX clients are feasible way to overcome Smartist limitations: they can be used only to make Smartist operations automatic and integrate it with other (ActiveX) applications.
A brief description of other software solutions Laservall provides can help choosing the right tool.

xLaser

This is a collection of functions and (for the most part) objects. Goal of the library is to offer the programmer willing to implement Laservall technology into its application, a way to access the laser, initialize and set DSP board, build a layout and engrave it

xLaser is an open source project. Cooperation to extend its capabilities is welcome.

Please note: The xLaser toolkit is a development platform designed for laser marking applications. The xLaser toolkit consists of multiple software modules, some developed by Laservall and many developed by other members of the open source community (and released under "GNU Lesser General Public License"2). Those authors hold the copyrights in the modules or code they developed. At the same time, the combined body of work that constitutes xLaser toolkit is a collective work which has been organized by Laservall, and Laservall holds the copyright in that collective work.

xLAL

xLAL is an ActiveX library wrapping xLaser objects. ActiveX technology allows fast and easy integration with many development languages (Visual Basic, C#, Borland C Builder for example) paying a little fee in performance time and reduced flexibility.

xLAL is a copyrighted product

Update: xLAL is now out-of-maintenance, this is no more a supported project

The right tool

The following table illustrates what solution is best for specific problems

Tool	Application
xLaser	high specialization applications: high productivity, special layouts, high integration with existing application, high flexibility
xLAL	Integration with existing application, like xLaser but with lower flexibility and time time-critical issues
ActiveX client	Applications were Smartist is sufficient but its interface need to be customized
VBscript	Automation of all operations Smartist can perform
Item	Integration with PLCs for very simple projects

[2] "GNU Lesser General Public License" terms and conditions could be found here: http://www.gnu.org/licenses/lgpl.html

Basic scripting syntax

This chapter is not a manual for scripting syntax. More detailed documentation can be found all over the web, starting from Microsoft's web site(s).
This chapter is an introductory to scripting syntax and writing the simplest program.

Variables

VBScript is derived from Visual Basic and the variables behave in almost the same way.
The instruction **Dim** is used to declare them:

```
Dim variable1
Dim variable2
etc.
```

Or

```
Dim variable1, variable2, etc.
```

The Variant type: weakly and strongly typed languages

Unlike other languages variable declaration does not require to declare variable type, such as a number, a string, a date etc. This is why VBScript belongs to the category of **weakly typed** languages as opposed to **strongly typed** languages like C or C++ where all variable type need to be declared. All declared variables belong to a generic container-type called **Variant**. It is VBScript interpreter that converts variant to the proper type required in the operation. This makes it very easy to program, but requires to be careful manipulating variables as interpreter will not warn us when performing ambiguous functions (as using a string in a mathematical operation, for example)

```
dim a,b,c
a = "1"
b = 5
c = a + b   'interpreter won't issue an error until run-time
```

Things can go subtler and errors more difficult to find:

```
Dim a,b,c
a = InputBox("Insert a number")
b = 5
c = a + b
```

InputBox prompts the user to insert data; these are always returned as a string, no matter what the input was.

Subtypes

Please note: types do not exist, but subtypes which characterize the variables do, they are:

Boolean:Can contain True or False.
Byte : Contains an integer between 0 and 255.
Integer: Contains an integer between -32,768 and 32,767.
Currency: Value between -922,337,203,685,477.5808 and 922,337,203,685,477.5807.
Long: Contains an integer between -2,147,483,648 and 2,147,483,647.
Single: Contains a floating point number with single precision between –3.402823E38 and 3.402823E38
Double: Contains a floating point number with double precision between –1.79769313486232E308 and 1.79769313486232E308
Date: Contains a number representing a date between 1 January 100 and 31 December 9999.
String: Contains a string with a variable length composed of a maximum of around 2 billion characters.
Object: Contains an object.

Errors can be avoided by *casting* variables (i.e. the transforming a variable from one type to another) explicitly

```
dim a,b,c
a = InputBox("First Number:")
b = InputBox("Second Number:")
c = Cint(a) + Cint(b)
MsgBox "Result " & c
```

InputBox is a function that allows the user to enter a custom datum. Return value is always a string.
CInt is a cast function: it tries to force the value of the variable into an integer.
MsgBox displays the result in the form of a box. & is the concatenation operator for strings.

Arrays

Variant types can be used as data arrays as well. Declaration is similar to regular variables using the **Dim** statement:

```
Dim array(10)           'declare a 10 element array
Dim dynarray()          'declare an array whose dimension is not
kown
Redim dynarray(100)     'allocates storage space
```

Declaration

Using **Dim** to declare a variable is not mandatory: if a variable is used but not declared the interpreter takes care of creating it (creation is usually the main purpose of declaration). This is an advantage because you have no need to keep track of all variables declared but can be misleading because sometimes a simple error in typing a variable can result in long debug (the interpreter "thinks" the mistyped variable is a new declared one)
When creating a complicated program I recommend to use the **Option Explicit** command at the beginning of the program. This command makes variable declaration compulsory: if a variable is used but not declared an error is generated.

```
Option Explicit
dim a,b,c
a = InputBox("First Number:")
b = InputBox("Second Number:")
c = Cint(a) + Cint(b)
MsgBox "Result " & c
```

The name of a variable must be chosen according to some rules:

- It needs to start with a letter of the alphabet.
- It cannot include periods and spaces.
- It must not be composed of more than 255 characters.
- There is no difference between upper and lower case letters.
- Assign the variable a name that reminds of its logic function inside the program. This is not a not compulsory but is used to make programs easier to read. E.g. a variable that must contain a date of birth could be called dtDate_Of_Birth, where dt indicates that it is a date type.

Operators

VBScript operators can be divided into types:
Arithmetic Operators

		a=5 b=2	**Result**
+	Add	c=a+b	7
-	Subtract	c=a-b	3
*	Multiply	c=a*b	10
/	Divide	c=a/b	2,5
\	Whole Division	c=a\b	2
Mod	Module	c=a Mod b	1
^	Increase exponent	c=a^b	25
&	String chaining	c=a & b	52

For complex formulas, addition and subtraction from left to right have precedence.

Comparison operators

=	Equal
>=	Greater than or equal to
<=	Less than or equal to
<>	Different
<	Less than
>	Greater than

Comparison between two variables gives True or False

Logical Operators

Not	Negation
And	Logical Conjunction
Or	Logical Disjunction

These are used for operations with Boolean variables. They will be greatly used in the next chapter which examines conditional instructions.

The following program is used to check arithmetic operators:

```
Option Explicit
dim a,b,c
a=5
b=2
MsgBox "c=a + b ---->" & a + b
MsgBox "c=a - b ---->" & a - b
MsgBox "c=a * b ---->" & a * b
MsgBox "c=a / b ---->" & a / b
MsgBox "c=a \ b ---->" & a \ b
MsgBox "c=a Mod b ---->" & a Mod b
MsgBox "c=a ^ b ---->" & a ^b
MsgBox "c=a & b ---->" & a & b
```

Conditional Structures and flow control

To perform operations conditionally the main command is **If**. The syntax is as follows:

```
If <condition> Then
      <operations if the condition is true>
Else
      <operations if the condition is false>
End If
```

The following example uses the function Month(), which gives the month in a numeric form, and Now(), which gives the current date and time:

```
if month(Now())=6 then
      MsgBox "It is June"
else
      MsgBox "It is not June"
end if
```

It is also possible to nest structures:

```
if month(Now())<=6 then
      MsgBox "We are in the first half of the year <br>"
      if month(Now())<=3 then
            MsgBox "it is the first quarter."
      else
            MsgBox "it is the second quarter."
      end if
else
      MsgBox "We are in the second half of the year <br>"
      if month(Now())<=9 then
            MsgBox "it is the third quarter."
      else
```

```
                MsgBox "it is the fourth quarter."
      end if
end if
```

Notice the importance of indenting when using structures that are this complex.
Another use of the **If** instruction is the give as many conditions as possible:

```
if month(Now())<=3 then
      MsgBox "We are in the first quarter."
elseif month(Now())<=6 then
      MsgBox "We are in the second quarter."
elseif month(Now())<=9 then
      MsgBox "We are in the third quarter."
else
      MsgBox "We are in the fourth quarter."
end if
```

You can see that in this type of structure only one operation is carried out. Starting from the top and moving towards the bottom, at the first condition the corresponding operations is effected and the structure is left. If it was January all the conditions would have been true but only the first sentence would have been printed.

The logical operations **Not**, **And** and **Or** can be use to combine multiple conditions.

When using structures with many choices based on the value of a parameter, you can use a structure with **Select**. This is the same program as above but with a **Select** structure:

```
Select case month(Now())
case 1,2,3
  MsgBox "We are in the first quarter."
case 4,5,6
  MsgBox "We are in the second quarter."
case 7,8,9
  MsgBox "We are in the third quarter."
case else
  MsgBox "We are in the fourth quarter."
end select
```

In this structure the variable to check is set with the instruction "Select case Variable"., and then a list of possible values that the variable can assume with the instruction" case *value1, value2, value3*"; the operations to be carried out follow this instruction. In the example the variable is numeric, in the case of string variables the values are placed between quotation marks: case *"value1", "value2", "value3".*

Structures for cycles

A cycle is used to repeat operations for a certain number of times or until a certain condition occurs.
The **For - Next** cycle increments by a variable at each cycle. When the variable reaches an established value the cycle finishes.

```
Option Explicit
Dim i
For i=4 to 20
  MsgBox i & "<br>"
next
```

The program prints the values from 4 to 20 in a column.
Using the **Step** parameter, I can count backwards and use different steps.

```
Option Explicit
Dim i
For i=20 to 4 step -2
  MsgBox i & "<br>"
next
```

This program will display the numbers 20, 18, 16, 14, 12, 10, 8, 6, 4

The **For - Each** cycle is similar: a loop is performed for all the elements contained in an array

```
Option Explicit
Dim vector(5),element
vector(0)="Html"
vector(1)="Asp"
vector(2)="Php"
vector(3)="JavaScript"
vector(4)="VBScript"
For Each element In vector
  MsgBox element & "<br>"
next
```

The **Do - Loop** cycle is repeated until a condition becomes false. The following code will be repeated until a second has elapsed from start.

```
Option Explicit
dim StartTime, EndTime, count
count=0
StartTime = Now()
EndTime = Now()+1/100000
Do While now() <= EndTime
  count=count+1
Loop
MsgBox "The cycle has been repeated " & count & " times in 1 second"
```

If the condition never occurs, the code in the cycle is never executed. Moving the condition to the bottom the test executes at least once as the condition is evaluated at the bottom. In the following example the message "Test Until" is displayed even if the condition is false.

```
Option Explicit
dim a
a=10
Do
  MsgBox "Test Until"
Loop While a < 0
```

The cycles can also be nested. In the following code the contents of a matrix is displayed.

```
Option Explicit
dim cLines, Columns, Matrix(3,2)
matrix (1,1)=1
matrix (2,1)=2
matrix (3,1)=3
matrix (1,2)=4
matrix (2,2)=5
matrix (3,2)=6
for cLines=1 to 3
      for cColumns=1 to 2
            MsgBox mstrix(cLines,cColumns) & " "
      next
      MsgBox "<br>"
next
```

Functions and Sub

These two keywords can be used to declare a procedure that perform a series of statements, and change the values of its arguments (i.e. what the majority of programming languages call a function). However, unlike a **Sub** procedure, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function, such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

```
Function myFirstFunction
      myFirstFunction = "Hello"     'this is function's return value
End Function
```

```
Sub myFirstSub
      myFirstSub = "Hello"
End Sub

MsgBox myFirstFunction              'prints Hello
MsgBox myFirstSub                   'error: type not corresponding
```

There's no declaration order: i.e. function can be called before or after they have been declared.

Objects

VBScript is a object-oriented programming language. Object-oriented programming languages offer the ability to organize the information into groups called classes. Each instance (i.e. allocation in physical memory) of a class is called an object.
A class is composed of properties (that is characteristics, represented by variables) and methods (that is behaviors, implemented as functions). Just like in the real world a person is characterized by properties like sex, height, hair color and methods like how they talk, walk and throw, a "person" class can be modeled with variables corresponding to the characteristics and function that describe the actions the person can perform.
VBScript has some built-in classes but others can be created by user. Smartist itself adds several objects to its context (i.e. they are accessible in the built-in editor): they will be described more accurately in the next chapter.

Smartist Objects

The following is a list of all objects Smartist declares. This means that in Smartist context they became sort of reserved keywords (i.e. users can not declare variables or functions having same name) and this words are highlighted in Smartist built-in editor.

- **Project**
- **Document**
- **Spooler**
- **Control**
- **Axis**
- **ComPort**
- **InputObject**
- **IoPort**
- **Tmr**
- **TextFile**

Project

A project is a collection of items. Items can be either **Document**s (plane or ring) or special "instructions" that Smartist interprets to perform specific actions (cloning a document, setting outputs, waiting fro specific input etc).

Processing items

Smartist default behaviour when script is not enabled is to process items in a sequential order from top to bottom: when the current item being processed is completed, the next one is started automatically. With script enabled there's a different behaviour between Smartist 4.0x and Smartist 4.1x. Smartist 4.0x processes items from top to bottom as without script. Smartist 4.1x default is to not process items at all. This is because the script itself can be required to take control over the processing sequence. *ProcessActiveItem*, *ProcessNextItem*, *SelectActiveItem* are some of **Project**'s functions that can be used to control the process. If the user want the script to behave the other way the *ProcessItemsAsDefault* function must be used.

Loading projects

Projects cannot be loaded within a script. The Load function (as the Hide/Show and the Close functions) are meaningful only in ActiveX clients (although the latter can be used)

Getting interfaces

The *GetXXX* functions in **Project**'s interface allow to have access to other objects interface.

Document

A **Document** is a collection of marking objects such as string, barcodes, data matrix, vectorial objects, raster objects (bitmaps and jpeg). **Document**s are represented in **Project** as items (but not all items are **Document**s).

Coordinates system

The marking field is represented as a Cartesian plane with the origin at the centre of the marking filed itself. This origin cannot be changed.
Objects position is always referred to plane origin. Objects origin (the point the coordinates are measured with respect to plane origin) is placed by default at the centre of the rectangle surrounding the object. The origin can be moved to any angle of the box or to the middle of any box side.

Modifying Document appearance

The *Move* and *Rotate* functions provide a way to modify **Document'**s appearance. Remember to call *UpdateView* to apply changes and redraw screen layout.

Accessing Objects

Smartist interface allows to assign IDs to objects. IDs should be unique alphanumeric string: they are used in scripts to access those objects interfaces. The *GetObject* and *GetObjectIds* functions provide functionality to get the list of accessible objects in **Document**s and to object interface.

Laser Parameters

A **Document** is associated with laser parameters in the way that the laser engraves all objects contained in the document with the parameters associated to the document unless differently specified. *Power*, *Frequency*, *Passes*, *Delay* and *Speed* functions set the parameters.

Synchronization

Strings and objects inheriting from these (i.e. barcodes and data matrix) have built-in formatters that allow simple operations like formatting an automatic counter or representing a date. Updating happens before or after marking (dates are updated right before, counters right after); you can force updating of these properties by calling the *PreSyncObjects* and *PostSyncObjects* functions. Forced updates is necessary when default item processing is not enbled

Spooler

The **Spooler** object allows direct interaction with the raw buffer data being sent to the DSP board for controlling laser emission and scanners movement. Dynamic **Spooler** management can be used when tracking **Document**'s limit or to gain control over the synchronization procedure described in previous section. In fact pre- and post-synchronization requires the program to re-send data to the spooler (as some data may have been changed). This operation can be time-consuming: if you are sure data have not been changed you can simply re-execute the **Spooler.**

Busy and Ready

The *IsLaserBusy* and *IsLaserReady* function may seem to produce the same (complementary) information. Actually, busy refer to the status of the DSP board and the function directly accesses the underlying hardware to get the information, while ready refers to the **Spooler** status: it is ready when it has been correctly closed and can be sent to the laser.

Calling sequence

Call *Break* before using the **Spooler** if you are not sure if the laser is busy or not. *Reset* allows to clear **Spooler**'s memory. You then need to *Open* the spooler, fill it with data trough the *SendXXX* functions, finally *Close* the **Spooler** and Execute it.

Control

The **Control** object allows control over the parameters characterizing the system. Please consult "Introduction to Laservall technology" for an overview of the meanings of the parameters.

Axis

The **Axis** object controls axis movement

Comport

The **ComPort** object allows script users to control RS232 ports for I/O operation.
Please note: **ComPort** object is not available in Smartist ActiveX interface intentionally as ActiveX client have better ways to directly access serial ports.

Flags

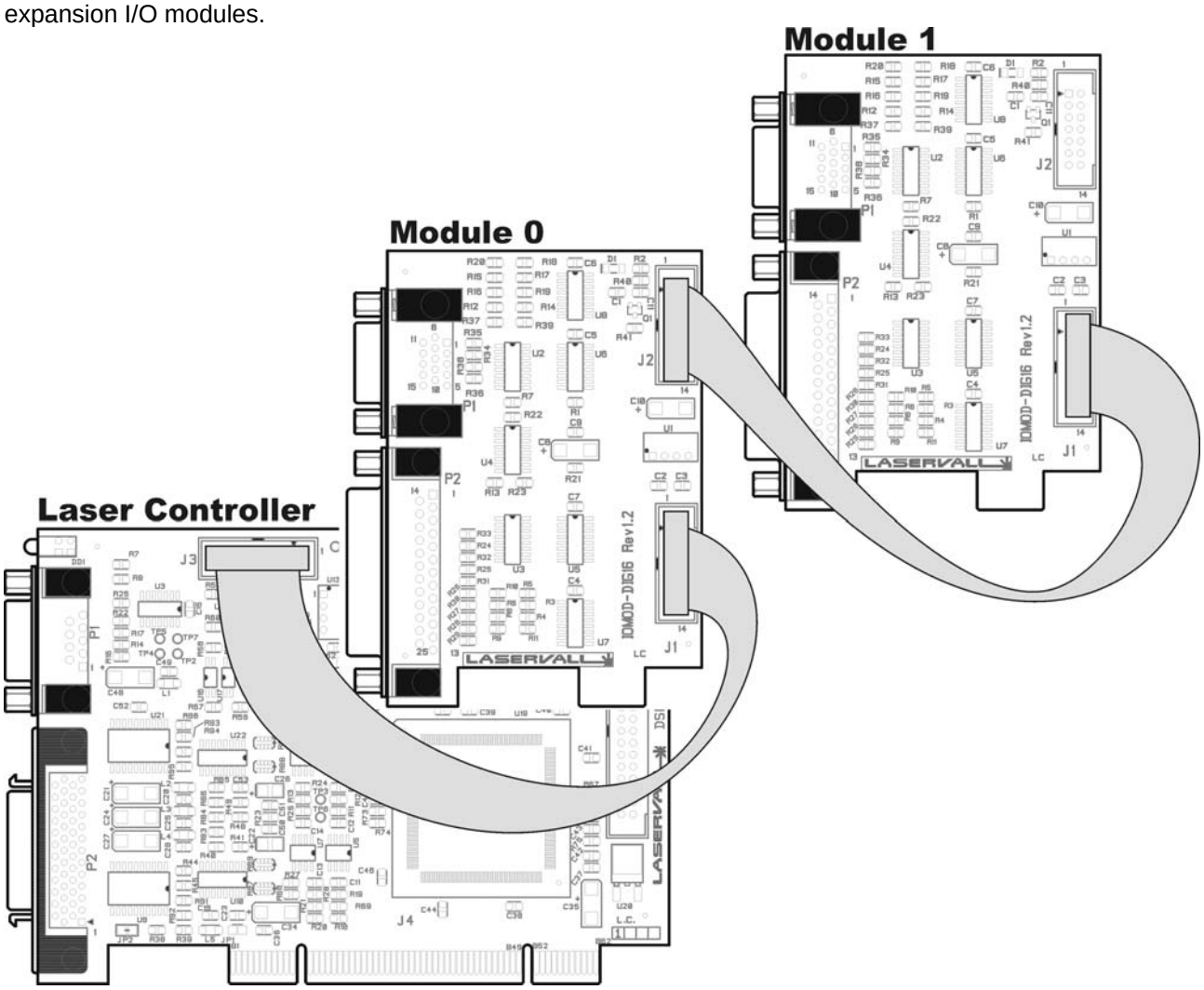
ComPort interface allows to set a number of flag whose usage makes sense only with the event notification system (which I will talk about more later). Every time a flag changes its status an event is fired.

InputObject

This object extends the capabilities of the standard InputBox object: it is possible to prompt the user for more information in different formats (strings, numbers, strings in a drop down list)

IoPort

The DSP2 board is actually composed of one laser controller module connected to one or more expansion I/O modules.



Each module offers 16 inputs and 16 outputs: in the first module only a part of these are available to the user. The following table list all pins and their use:

DB 25 pin male - Axis motor connection.

Pin out

1:	+12V 500mA
2:	Step Y
3:	Step Z
4:	Brake X
5:	Brake Y
6:	Brake Z
7:	Zero X
8:	Zero Y
9:	Zero Z
10:	Disable X
11:	Disable Y
12:	Disable Z
13:	GND.

Pin out

14:	+5V 500mA.
15:	Step X
16:	Dir Z
17:	Dir Y
18:	Dir X
19:	Input 8
20:	Input 4
21:	Input 2
22:	Input 1
23:	Output 9
24:	N.C.
25:	GND.

DB 15 pin male – Laser controls/status..

Pin out

1:	Out – Laser End. Active at end of engraving.
2:	Out – Laser Busy. Active during engraving.
3:	Out – Laser Ready. Active when system is ready.
4:	In – External Start. Starts engraving when activated.
5:	In – External Stop. Stops engraving in progress when activated.
6:	Input 13
7:	Input 15.
8:	Output 15
9:	12V.
10:	DSP Ready
11:	N.C.
12:	GND
13:	Input 12
14:	Input 14
15:	Output 14

Pins available to the user are Input and Output only. I/Os can be used to provide simple communication with devices like PLCs. The **IoPort** object is an interface to the board I/O and provide I/O communication.

Masking

The *SetPort* and *ResetPort* functions take a mask as second parameter. The mask is used to select or de-select the bit(s) going ON or OFF.
The mask is interpreted in binary formats: each input or output is associated with the corresponding bit in the mask (numbered right to left). Active bits (i.e. set to 1) in the mask are either set (i.e. moved to ON with the *SetPort* function) or reset (i.e. moved to 0 in the *ResetPort* function); inactive bits (i.e. set to 0) are left unchanged.

Checking

The CheckPort and UncheckPort functions allows to monitor port state: if the checked port changes its state, an event is fired (see Events later)

Tmr

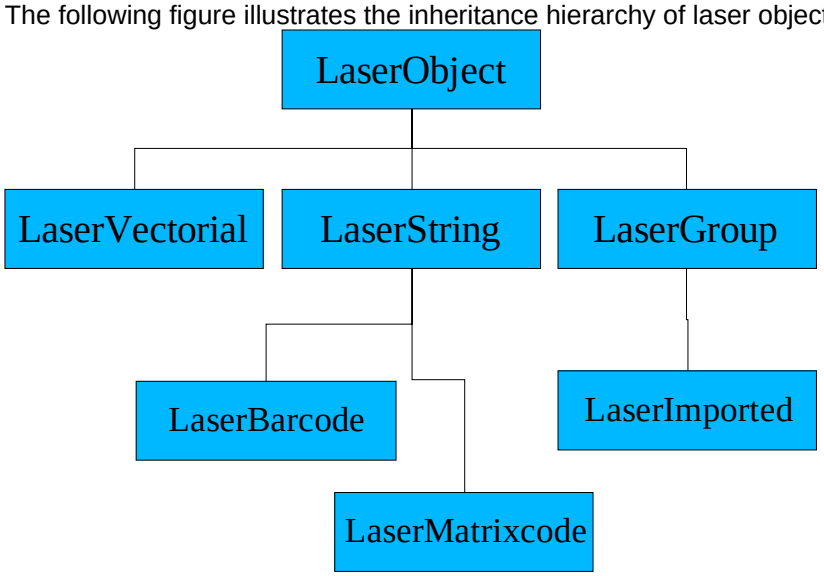
Allows to set/reset a timer that periodically fires an event

TextFiles

Allows access to text files. VBScript facilities to access files are usually preferred.

Graphical Objects

The following figure illustrates the inheritance hierarchy of laser objects.



The hierarchy is *single rooted,* meaning that all objects inherit from the base class LaserObject. Inheritance means that derived objects get the same interface as the base class object and add some more properties and/or functions depending on their characteristic.

Transformation

LaserObject were born vectorial therefore all objects have the interface to perform vectorial transformation over them even if this does not make sense (as for LaserImported objects containing raster image).
A generic vectorial transformation can always be summarized in the product of the coordinates of the points constituting the object with a 2x2 matrix. Transformations as *Scale*, *Rotate*, *Shear* are cases of the more general algorithm accessible trough the *Transform* function.
Remember to call *Update* after applying any transformation to an object,

Parameters

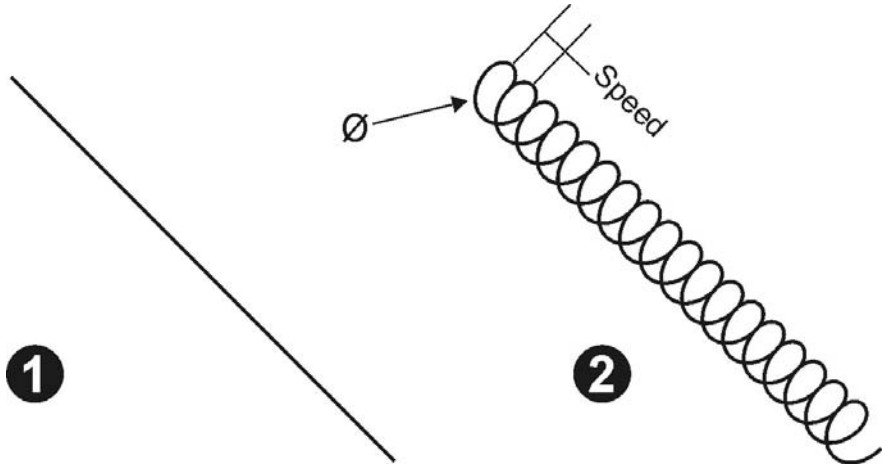
Laser parameters are the same as **Document's**. Refer to **Document** section.

Box and Extend points

The surrounding box is the minimum rectangle (i.e. having sides parallel to the axis) containing the object.
Extend points belong to a bigger rectangle surrounding the box itself. This is used inside Smartist to place command icons around the object

Wobble

The purpose of this parameter is to engrave the single lines of vectorial graphics, either True Type characters or imported figures, with thicker lines. The line thickness if normally equal to the dimension of the laser spot. However, for some applications this thickness may not be sufficient, the wobble function solves this problem.
The figure below represents the functioning principle of the wobble. Figure 1 shows the result of a vector engraving with the wobble disabled. Figure 2 shows the same vector but with the wobble enabled. The conversion of the vector into a dense spiral provides greater thickness of the engraved line. The same function can be applied to any vectorial graphic that is going to undergo plain or ring engraving.



Properties:

WobbleDiameter: Represents the diameter of the spiral curve

WobbleSpeed: Represents the frequency of spiral repetition

Events

Axis, ComPort, IoPort, Project and Tmr objects expose an event interface. Events are functions that Smartist fires when specific conditions occur. Scripts or AciveX client re-implement those functions to catch the condition occurring.
Event syntax is the same as the normal **Sub**s; the name of the **Sub** is composed adding the name of the event to the name of the object (separated by underscore) as in the following example:

Object: Project
Event: OnQueryStart
Sub name: Project_OnQueryStart

Project events

Project object associates most of the event. The **OnCustomEvent** family event is fired when the corresponding item in the project is being processed. Although the event is documented as having no arguments, the user can define a string in item's interface that can be used as parameter for the event. This example describe this behavior:

User adds a Custom Event item to the project. He assigns number zero to the event and writes this string as parameter: "12,30". The following code catches the event and interprets the parameter:

```
Sub Project_OnCustomEvent0(theString)
    ParamList = Split (theString, ",")          'gets parameter list from the string argument
End Sub
```

The **OnItemStarting** event differs from the **OnItemStart** as the latter fires before pre-synchronization takes place while the second fires after.

IoPort and ComPort events

IoPort's **OnImputChange** event fires only if the port is checked.
ComPort's **OnRxFlag** and **OnRxChar** events fire only if the Rx char has been defined trough the **SetFlag** method. Other ComPort events relates to well documented RS232 signals.

Script limitations

The goal of this chapter is to list the errors a programmer can be trapped in when developing a script in Smartist. Contents of this chapter results from Laservall experience in script development and are supposed to increase with it.

Use Option Explicit declaration with large scripts

Name mistyping can be a source of insidious bug in your script since the interpreter does not complain if a new variable is used without being initialized. These errors can be difficult to track down in complex programs: use **Option Explicit** to avoid mistypes.

DSP2 I/Os do not work while marking

This limitation relates to DSP2 more than scripting itself. While the board is busy marking or tracking the limits then I/O changes are not detected and events not fired.

Be careful when assigning object reference to global variables

When a Smartist object reference is assigned to a variable, then a “link” is created between the variable and the real object. Local variables go out of scope (i.e. out of visibility) when exiting the function they are placed in: in this case the link is automatically deleted by the script engine. Global variables can be accessed within any function in the script, therefore they “live” as long as the script is executed and never run out of scope. User need to be aware that they need to take care of the link between the variable and the Smartist object. See the following example to understand what happens under the scene:

```
Dim theRef                                          'global variable
Sub ComPort_OnRxFlag
      theFile = ComPort.Read                        'read the name of the
file to load in ComPort's buffer
      theRef = Nothing                              '!!Smartist crashes
      without this line!!
      Document.Load(theFile)
      Set theRef = Document.GetObject("ID")         'assign the global
      variable a reference to an object contained in the
                                                    'document
End Sub
```

This code crashes the second time the OnRxFlag event fires. The Load function deletes all the content of the document: if theRef was already assigned to an object in a previous call to OnRxFlag, then the interpreter will try to delete the link with that object before making a new assignment, but since the object has already been deleted by the Load function, then the engine crashes.

Object destruction

Object destruction is a delicate topic in ActiveX technology.
Basically it is not possible for an object to delete itself. If this is done, the reference to that object is still active and each successive use of the variable holding the reference to the object will crash the application. A method like `Project.Close()` can be misleading in an ActiveX client:

```
Dim WithEvents myProject As LaserManagerLib41.LaserProject

Private Sub Form1_Load(…) Handles MyBase.Load
      ' Connects to Smartist4.1 creating a new empty project
      myProject = New LaserManagerLib41.LaserProject
      myProject.Load("c:\temp\test.prj")
End Sub

Private Sub Button1_Click(…) Handles Button1.Click
      myProject.Close()
      myProject = Nothing
End Sub
```

This code crashes when dereferencing myProject because the object has already been destroyed but the ActiveX engine tries to decrement its internal reference counting before freeing the variable. The `Project.Close()` function should only be used when the variable referring to the project is never used anymore.

Activate Smartist ActiveX license

Changes between Smartist 4.1.2 and 4.1.3 involved the licensing mechanism: a new line has been added to the license file that enables ActiveX interface to be exported to external clinets; without this key it is impossible to build an ActiveX client.

How to build an ActiveX application

I use Microsoft Visual Studio as development environment and therefore I will describe the procedure to link a client application to the Smartist server with Visual Basic and Visual C. I will not go deep into the details of writing the application, linking objects and events as these topic can be better learned looking at some sample code you can require our support department.

Visual Basic ActiveX clients

Adding Smartist ActiveX interface to a Visual Basic project is easy: right click the “References” item in the Solution Explorer window of the project. Click “Add Reference” then select the “COM” tab and choose “Laser Manager Control 1.0”. Click “Select” then “OK”: the interface is added to the project and you can navigate trough it with the Object Browser.

Visual C ActiveX clients

Either one of the following procedures link an MFC application to Smartist trough ActiveX technology.

The #import directive

```
#import "LaserEditor.exe" no_namespace rename("GetObject","GetObj")
...
myPrj.CreateInstance(_T("LaserEditor.Project"))
```

The #import directive is used to incorporate information from a type library. The content of the type library is converted into C++ classes, mostly describing the COM interfaces. The compiler generates two files from the type library containing the type library interface. See Microsoft documentation form more details.
This is not the procedure vcclient uses but it is the way to go if you are not developing an MFC application.

Using the wizard

In Visual Studio IDE click “Project” then “Add class”. Pick “MFC Class from TypeLib” then “Open”. Now you can either select “Registry” then pick up “LaserManager Control 1.0” from the drop down list or select “File” then browse to “LaserEdito.exe”. Select the interface you want to integrate from the list: one header will be generated and must be included for each interface. See the vcclient sample for details on implementing the code.