Pinnacle^3 Scripting Guide

External Beam and Brachytherapy Treatment Planning

Philips Radiation Oncology Systems—CONFIDENTIAL

Table of Contents

1: Introduction

2: Objects and Messages

3: Container Objects

4: Control Commands

5: Find Valid Object Messages

6: Advanced Scripting and Store Variables

7: Example Scripts

1 Introduction

This document describes the scripting capabilities of the Pinnacle^3 system. It describes how a user can write scripts to customize the system or extend the system for research purposes.

Great care should be taken with scripting. It is possible with scripts to cause system crashes, to modify dose distributions, or to corrupt data. User-defined scripts are meant for straightforward customizing, or for non-clinical research purposes.

2 Objects and Messages

Pinnacle^3 is developed using object-oriented techniques. The scripting capability is one benefit of this design.

2.1 Model-View-Controller Architecture

The Pinnacle^3 system is developed using an object-oriented development environment, which is implemented using the ANSI C programming language, but provides many of the object-oriented features provided by languages like Smalltalk or Objective-C.

The **model** is the core of the application. It includes the objects and inter- and intra-object dependencies which provide the core functionality of the application. Examples in Pinnacle^3 include a beam, a CT data set, or a dose computation engine.

The model is presented to the end user using **views**. The user interface is an obvious view of the model. Reports are another view, as are saved data files.

A **controller** is an element which modifies the model. Text fields and other widgets on the user interface are controllers. A script file is a controller. When a saved data file is read, it also acts as a controller.

The true benefit of this architecture comes from the separation between the model, views, and controllers. Multiple views and controllers interact with the model using messages. The model has no knowledge of the views or controllers. It simply returns information in response to query messages and performs operations in response to command messages.

When a script is played back, it sends messages to objects in the model. These messages are interpreted using the identical mechanisms used to process messages issued from the user interface or from a data file, including the application of any filters or limits.

2.2 Objects

An object is an encapsulation of the data and methods (functions) used to model a real-world object. For example, the Beam object consists of a representation of all beam attributes (a structure) and all methods which operate on the beam attributes (subroutines.)

Objects in Pinnacle^3 are similar to Smalltalk objects. They are treated as generic objects which respond to certain messages.

Each object is associated with a Class. The Class contains a template for the object and has knowledge of its attributes and messages. It has the ability to create or destroy individual instances of the Class.

When a message is sent to an object, the message list for the object is obtained from the Class. If a matching message is found, the Class knows how to process it, which may involve calling some function, or directly interacting with some attribute of the object. If no matching message is found, a given Class may ignore the message, issue an error message, or pass the message on to a Class from which it inherits behavior.

The *root object* is the highest level object. In Pinnacle^3, the root object is named Pinnacle and contains all elements of the Pinnacle^3 application. All script messages are sent to the root object, which then propagates the messages on to the proper sub-objects.

2.3 Messages

A message is a command which is sent to an object. In its simplest form, a message is simply a string on the left hand side and a value on the right hand side, like: Name = "test";

2.3.1 Basic message syntax

The basic syntax of a command message is:

<*SetMessage*> = <*Value*>*;*

where <*Value*> can be one of the following:

- A string enclosed in double quotes.
- A floating point or integer number.
- A Query message.

Note the semicolon which must be used to terminate each message.

The message may identify an object or an attribute of an object. A message is routed through the hierarchy of objects using key pieces delimited by periods. For example, the following message modifies the gantry angle for the current beam in the current trial (sub-plan):

```
TrialList.Current.BeamList.Current.Gantry = 180.0;
```

This message is sent to the TrialList sub-object of the root object. The TrialList is an ObjectList which contains a list of Trial objects. The Current message tells it to route the message to the currently selected Trial in the list. The current Trial accepts the message and routes it to it's BeamList sub-object, which is an ObjectList containing Beam objects. Eventually the message reaches the current Beam, which recognizes the Gantry message and sets it's gantry angle accordingly.

The right hand side of the message can be a Query message. For example, the following example sets the couch angle to be equal to the gantry angle:

```
TrialList.Current.BeamList.Current.Couch =
   TrialList.Current.BeamList.Current.Gantry;
```

Note that messages can be split across lines. All white space between messages is ignored. So the following messages are equivalent:

```
TrialList. Current. BeamList. Current. Gantry = 180.0;
TrialList.Current.BeamList.Current.Gantry = 180.0;
TrialList.
   Current.
      BeamList.
         Current.
            Gantry = 180.0;
```

2.3.2 Query messages

If a message has no right hand side, a query is performed with the message and the result is echoed to standard output (the console). The following message echoes the value of the beam gantry angle:

```
TrialList.Current.BeamList.Current.Gantry;
```

For this reason, it is sometimes necessary to include a "dummy" right hand side argument for a message which has no arguments. Typically an empty string is used:

```
Quit = "";
```

The Quit command has no arguments, but the empty string is necessary so the system knows that a command is being sent and not a query.

2.3.3 Nesting messages

Curly brackets can be used to nest messages. For example, a series of messages can be sent to an object by using curly brackets at a given level.

For example, the following message sets the couch, gantry, and collimator angles for the current beam:

```
TrialList.Current.BeamList.Current = {
   Couch = 180.0;
   Gantry = 90.0;
   Collimator = 0.0;
   };
```

Multiple levels of nesting are possible:

```
TrialList.Current = {
   DoseGrid = {
      VoxelSize.X = 0.2;
      VoxelSize.Y = 0.2;
      VoxelSize.Z = 0.2;
      ;}
   BeamList.Current = {
      Couch = 180.0;
      Gantry = 90.0;
      Collimator = 0.0;
      };
   };
```

The previous script is equivalent to the following script:

```
TrialList.Current.DoseGrid.VoxelSize.X = 0.2;
TrialList.Current.DoseGrid.VoxelSize.Y = 0.2;
TrialList.Current.DoseGrid.VoxelSize.Z = 0.2;
TrialList.Current.BeamList.Current.Couch = 180.0;
TrialList.Current.BeamList.Current.Gantry = 90.0;
TrialList.Current.BeamList.Current.Collimator = 0.0;
```

2.3.4 Special characters

Certain characters must be escaped when present in scripts. Two common characters which must be escaped are the asterisk and pound sign.

The pound sign transforms its string argument into a string literal without interpretation of special characters. The two statements in the following example are identical:

```
TrialList.# "Current".Name = "test"; // Escaped
TrialList.Current.Name = "test"; // Not escaped
```

In the following example, the first line will generate a syntax error because of the special character (asterisk.) The second line will be processed correctly.

```
TrialList.*.Name = "test"; // Syntax error
TrialList.# "*".Name = "test"; // Escaped - no syntax error
```

Unfortunately, the pound sign itself is used in some messages, which leads to some fairly confusing syntax, as in the following example:

```
TrialList.# "#0".Name = "test";
```

The #0 message selects the first (index 0) trial object.

3 Container Objects

Pinnacle^3 uses lists extensively to store collections of objects. For example, Pinnacle^3 has lists of beams, windows, regions of interest, points of interest, etc.

The ObjectList object is a generic container object which can store an array of *child* objects. Most ObjectList instances have an installed child class, and can create instances of this class. In other words, the list knows what type of object is going to be stored in the list.

3.1 Creating child objects

The "CreateChild" message creates a new instance of the child class and adds it to the list. Objects are always added to the end of the list.

The following message creates a new beam:

```
TrialList.Current.BeamList.CreateChild = "";
```

In addition, sending the class name to the ObjectList performs an identical function.

```
TrialList.Current.BeamList.Beam = "";
```

This is useful because the "Beam" message will also route messages to the object which was last added to the list. Combined with nesting, this is useful for adding a new object and setting some of its parameters, as in the following example:

```
TrialList.Current.BeamList.Beam = {

    Couch = 180.0;
    Gantry = 90.0;
    Collimator = 0.0;
    };
```

The "Beam" message creates a new beam. The "Couch" message is routed through Beam which sends it to the last added object.

3.2 Referencing child objects

Various messages can be used to route a message to a child object.

3.2.1 Identifying a child object by name

Most objects respond to the message "Name". It is possible to route a message to a child object by specifying the child object's name, as in the following example:

```
TrialList.Current.BeamList.Lateral.Couch = 180.0;
TrialList.Current.BeamList.Ant Post.Couch = 180.0;
```

This message sets the couch angles of the beam objects named "Lateral" and "Ant Post". Note that white space inside a message is significant.

There can be ambiguity when addressing objects by name. For example, a child object which was named "Current" would cause problems since this name conflicts with the ObjectList message named "Current". A message is first compared to known ObjectList messages before being compared to child object names. Most ObjectList's prevent the name of a child object from being set to an ObjectList message name.

3.2.2 Identifying a child object by number

Child objects can also be specified by index. Indexing in an ObjectList starts at zero.

The following message sets the couch angle of the third beam (at index 2).

```
TrialList.Current.BeamList.# "#2".Couch = 180.0;
```

Note that the pound sign is a special character which must be escaped.

The pound sign can be omitted, as in the following example:

```
TrialList.Current.BeamList.2.Couch = 180.0;
```

But this is risky since a beam may have a numeric name. For example, if a beam existed which was named "2", the message above would be routed to this beam even if it was not at index 2.

3.2.3 Referencing the current child object

The ObjectList maintains a *current* index which points to one of the child objects. Messages are often routed to the current object, so a given script will operate on the beam or other object which is currently selected by the user.

For example:

```
TrialList.Current.BeamList.Current.Couch = 180.0;
```

3.2.4 Referencing the last child object

The "Last" message references the last object in the list.

For example:

```
TrialList.Current.BeamList.Last.Couch = 180.0;
```

Since newly created objects are added to the end of the list, the "Last" message can be used to reference an object which has just been created.

3.2.5 Passing a message to all child objects

The wildcard message (*) can be used to send a message to all members of the list. For example, the following message sets the couch angle for all beams to 180.0.

```
TrialList.Current.BeamList.#"*".Couch = 180.0;
```

Multiple wildcards can be used. The following message sets the couch angle for all beams in all trials.

```
TrialList.# "*".BeamList.# "*".Couch = 180.0;
```

Note that the asterisk message must be escaped.

3.3 Destroying child objects

To destroy a child object, pass the message "Destroy" to the child after referencing it using one of the techniques above.

For example, the following commands will destroy the current beam.

```
TrialList.Current.BeamList.Current.Destroy = "";
```

Note that it is not always safe to destroy an object because other objects may reference it. For example, a point of interest object may be referenced by a beam which is using it as an isocenter. Some objects have a sub-object called a "RelyOnList" which can be used to test whether it is safe to destroy the object.

For example, the following could be used to delete a point of interest:

```
IF.PoiList.Current.RelyOnList.HasNoElements.
THEN.PoiList.Current.Destroy.
ELSE.WarningMessage = "Unable to delete POI";
```

3.4 Making a child object the current object

The "MakeCurrent" message is used to make a child object current.

For example, the following message will make the last object in the list the current object:

```
TrialList.Current.BeamList.Last.MakeCurrent = "";
```

3.5 Sorting the ObjectList

An ObjectList can be sorted by any child object key. The following example sorts the beam list by couch angle in ascending order:

```
TrialList.Current.BeamList.SortBy.Couch = "";
```

Multiple keys can be specified:

```
TrialList.Current.BeamList.SortBy.Couch.Gantry = "";
```

To do a descending sort, precede the attribute with "D" as in the following example where the primary sort key is the couch angle in ascending order, and the secondary sort key is the gantry angle in descending order.

```
TrialList.Current.BeamList.SortBy.Couch.D.Gantry = "";
```

4 Control Commands

4.1 Exiting the system

The "Quit" and "QuitWithSave" messages exit the core planning software. The "QuitWithSave" message saves all plan data before quitting.

```
    Quit = "";
```

or

```
    QuitWithSave = "";
```

4.2 Saving the plan

To save the current plan, use the "SavePlan" message.

```
   SavePlan = "";
```

Note that there is no argument for this command. The plan files are saved in the plan directory.

4.3 Writing a message to the console

The "Echo" message will send a string to the console window.

```
   Echo = "Starting My Script";
```

4.4 Turning on the hourglass cursor

If an operation is going to be lengthy, an hourglass cursor can be installed and a status message can be displayed.

```
   WaitMessage = "Doing something slow...";
   ...
   WaitMessageOff = "";
```

Multiple "WaitMessage" messages can be used to show the status of a multi-part process.

```
WaitMessage = "Step 1...";
...
WaitMessage = "Step 2...";
...
WaitMessage = "Step 3...";
...
WaitMessageOff = "";
```

4.5 Issuing a warning message

To issue a warning message which appears in a modal dialogue window, use the "WarningMessage" message.

```
WarningMessage = "This is a test.";
```

The script will continue to execute after the warning message dialogue is dismissed by the user.

4.6 Executing a system command

The "SpawnCommand" and "SpawnCommandNoWait" messages can be used to spawn a new process and execute the specified command. The "SpawnCommand" message will not return control to Pinnacle^3 until the spawned process completes. The "SpawnCommandNoWait" message starts the new process and returns control immediately to Pinnacle^3.

```
WaitMessage = "Checking disk space...";
SpawnCommand = "df > /usr/tmp/MyDiskSpace";
SpawnCommand = "xterm -e vi /usr/tmp/MyDiskSpace";
SpawnCommand = "rm -f /usr/tmp/MyDiskSpace";
WaitMessageOff = "";
```

or

```
SpawnCommandNoWait = "xclock";
```

4.7 Launching other programs

`Store.At.MyString.Execute`	Executes a command
`Script.ExecuteNow`	Executes a script
`SpawnCommand`	Executes an external program
`SpawnCommandNoWait`	Executes an external program and continues processing

4.8 Running a script from a script

The ExecuteNow command executes the specified script, then returns control to the "calling” script.

```
TrialList.Current...   //Do something
...
Script.ExecuteNow = "/home/pinnbeta/Scripts/MyOther-
Script.Script";
...
TrialList.Current... //Do more stuff.
```

4.8.1 Asking the user a Yes/No question

```
AskYesNoPrompt = "Are you having fun?";
AskYesNoDefault = 1;  // Default yes
IF.AskYesNo.THEN.WarningMessage = "Having fun!";
```

4.8.2 Automatically answering an expected Yes/No question

```
Test.ExpectAskYesNo=1;
Test.ExpectedAskYesNoReply=1; // yes=1, no=0
```

4.8.3 Automatically dismissing a warning

```
Test.ExpectWarning = "Text";
```

Note Text can be used to dismiss a certain expected warning message. Set Text to 1 to dismiss the next warning message.

4.9 If...then...else

```
IF.key1.OP.key2.THEN.action1.ELSE.action2 = value;
```

Where OP is:

Numeric	EQUALTO
	NOTEQUALTO
	GREATERTHAN
	LESSTHAN
	GREATERTHANOREQUALTO
	LESSTHANOREQUALTO
String	STRINGEQUALTO
	STRINGNOTEQUALTO
	IS
	ISNULL
	CONTAINS
Logic	AND
	OR
	XOR

```
IF.key.THEN.action1.ELSE.action2 = value;
```

Where key is a boolean query.

```
IF.key.THEN.action = value;
```

4.10 Iterating

```
RoiList.ChildrenEachCurrent.#"@".Root.Script.ExecuteNow = "/
home/pinnbeta/Scripts/MyScript.Script";
```

The Root message redirects the action to the Root object. The script makes each beam current, then executes the rest of the command.

5 Find Valid Object Messages

The Pinnacle^3 development team uses various code and runtime browsers to view available messages for objects. At this time these browsers are not accessible to end users. In addition, at this time it is not possible to make our internal documentation public because it contains proprietary information.

Pending the development of additional documentation on specific messages, there are a number of methods of finding available messages for an object or feature:

5.1 Recording commands

The script record capability lets you record a series of operations. The resulting script file will contain script commands. It is often useful to record a script as a starting point, and then edit this script to customize or generalize it. Consult the user manual for information on recording scripts.

5.2 Transcript files

Every command issued from the user interface is recorded in a plan transcript file, which is a regular script file. Contact ADAC for information on retrieving this file.

The file can be used in a similar fashion to a recorded script.

Note that the transcript files are removed periodically by the system.

5.3 Extracting objects

A given object can be saved to a file. This file is a script and will list persistent attributes of the object.

For example, to view the persistent attributes of the beam object, the following script could be used:

```
TrialList.Current.BeamList.Current.Save = "/files/rtp/
Beam.out";
```

The file will contain the following:

```
IsocenterName = "Poi_1";
PrescriptionName = "Prescription_1";
PrescriptionPointName = "isocenter";
Name = "AP";
MachineNameAndVersion = "PhotonQAMachine2: 96/03/29 15:59:29";
Modality = "Photons";
MachineEnergyName = "10 MV";
...
```

Note that most messages have fairly clear names. All of the messages in this file could be used in a script to set the corresponding parameter.

6 Advanced Scripting and Store Variables

6.1 Variables - The Store

The Store is a dictionary that associates objects with user-specified strings

MyFloatAttribute	ptFloatObject
MyStringAttribute	ptStringObject
MyBeamReference	ptBeamReference

The Store is a non-persistent attribute of the Root object, accessed via the "Store” message. Store objects can be created for any object.

6.1.1 Creating a Float Entry in the Store

```
Store.FloatAt.MyFloatAttribute = 1.23;
```

Label for variable		MyFloatAttribute
New object reference	Class name	Float
	Class attributes	Value = 1.23;

6.1.2 Creating a String Entry in the Store

```
Store.StringAt.MyStringAttribute = "ABC";
```

Label for variable		MyStringAttribute
New object reference	Class name	SimpleString
	Class attributes	String = "ABC";

6.1.3 Creating a Beam Reference in the Store

```
Store.At.MyBeamReference = TrialList.Current.BeamList.Current.Address;
```

Label for variable	MyBeamReference
Existing object reference	TrialList.Current.BeamList.Current.Address;

The object at "MyBeamReference” is a reference to the beam in the beam list. If the beam in the list is deleted, the Store will nto be aware of it and will reference an invalid object.

6.1.4 Modifying a Float Entry in the Store

```
Store.At.MyFloatAttribute.Value = 2.34;
```

Label for variable	MyFloatAttribute
Attribute name	Value

Alternative:

```
Store.FloatAt.MyFloatAttribute = 2.34;
```

6.1.5 Modifying a String Entry in the Store

```
Store.At.MyStringAttribute.String = "DEF";
```

Label for variable	MyStringAttribute
Attribute name	String

Alternative:

```
Store.StringAt.MyStringAttribute = "DEF";
```

6.1.6 Modifying a Beam Reference in the Store

```
Store.At.MyBeamReference.Gantry = 123;
Store.At.MyBeamReference.Couch = 234;
```

Label for object	MyBeamReference
Attribute name	Gantry, Couch

Alternative:

```
Store.At.MyBeamReference.Gantry = {
   Gantry = 123;
   Couch = 234;
   };
```

6.1.7 Deleting an Entry in the Store

```
Store.FreeAt.MyFloatAttribute  = "";
Store.FreeAt.MyStringAttribute  = "";
```

Label for object	MyFloatAttribute, MyStringAttribute
Value not used	"";

6.1.8 Removing an Entry in the Store

```
Store.RemoveAt.MyBeamReference  = "";
```

Label for object	MyBeamReference
Value not used	"";

```
Store.FreeAt.MyBeamReference  = "";
```

Do not use this command. The beam is destoyed but the beam list is unaware of the deletion and the system will crash.

6.1.9 Saving and Reloading the Store

```
Store.Save = "/home/pinnbeta/Scripts/MyStore.Script";
```

The file name is /home/pinnbeta/Scripts/MyStore.Script.

```
Store.Save = "/home/pinnbeta/Scripts/MyStore.Script";
```

Note that the existing Store is not emptied before loading.

6.2 String Operations

Strings can be concatenated to create file names, unique beam names, etc. The SimpleString object and the Store are used to build strings.

6.2.1 Building Strings

```
   Store.StringAt.MyString = "ABC";
   Store.At.MyString.AppendString = "DEF";
```

Now the stored string is "ABCDEF”

```
   Store.At.MyString.AppendString = "xyz";
```

Now the stored string is "ABCDEFxyz”

6.2.2 Useful String Queries

Code	String
`Store.StringAt.MyString = GetEnv.HOME;`	/home/
`Store.StringAt.MyString = UserName;`	pinnbeta
`Store.StringAt.MyString = LexicalTimeStamp;`	1999/11/22 12:20:23
`Store.StringAt.MyString = ReportComment;`	whatever is typed in the report comment field
`Store.StringAt.MyString = PatientDirectory;`	…/Institution_0/plan_42
`Store.StringAt.MyString = PlanName;`	plan
`Store.StringAt.MyString = PublicScriptDirectory;`	.../PinnacleSiteData/Scripts
`Store.StringAt.MyString = SystemScriptDirectory;`	…/PinnacleStatic/Scripts
`Store.StringAt.MyString = PlanInfo.PatientName;`	Smith, John
`Store.StringAt.MyString = PlanInfo.PlanName;`	HeadAndNeck
`Store.StringAt.MyString = PlanInfo.Institution;`	ClinicAB
`Store.StringAt.MyString = PlanInfo.Planner;`	Bunker, Gloria
`Store.StringAt.MyString = PlanInfo.MedicalRecord-Number;`	345126
`Store.StringAt.MyString = PlanInfo.Physician;`	Doctor, John
`Store.StringAt.MyString = PlanInfo.Physicist;`	Physicist, Jane

6.2.3 Determining Query Keys

```
PlanInfo.Save = "/home/Temp.PlanInfo";
```

yields...

```
Institution = "University of XYZ";
PatientName = "Johnson, John";
MedicalRecordNumber = "123-45-6789";
PlanName = "4-field boost";
Planner = "Smith";
Physician = "DocName";
Physicist = "PhysName";
```

6.2.4 File Name Example

```
// Just in case these didn't get freed for some reason.
Store.FreeAt.TempBaseFileName = "";
Store.FreeAt.TempInputFileName = "";
Store.FreeAt.TempOutputFileName = "";
```

Note The FreeAt command does nothing if the string doesn’t exist.

```
// Create a base name for all files and scripts.
Store.At.TempBaseFileName = SimpleString{};
Store.At.TempBaseFileName.AppendString = PatientDirectory;
Store.At.TempBaseFileName.AppendString = "/.Temp.MachineUniquenessList";

// Create input file name
Store.At.TempInputFileName = SimpleString{};
Store.At.TempInputFileName.String = Store.At.TempBaseFileName.String;
Store.At.TempInputFileName.AppendString = ".in";

// Create output file name
Store.At.TempOutputFileName = SimpleString{};
Store.At.TempOutputFileName.String = Store.At.TempBaseFileName.String;
Store.At.TempOutputFileName.AppendString = ".out";
```

6.3 String Conditional Statements

Is

```
IF.Store.At.MyString.Is.xyz.
THEN.WarningMessage = "It’s xyz!";
```

IsNull

```
IF.Store.At.MyString.IsNull.
THEN.WarningMessage = "It’s empty!";
```

Contains

```
IF.Store.At.MyString.Contains.xyz.
THEN.WarningMessage = "Contains xyz!";
```

6.4 The Execute Statement

```
Store.StringAt.MyCommand = "RoiList.Current.Density = 1.23";
Store.At.MyCommand.Execute = "";

Store.StringAt.MyCommand =
         "RoiList.Current.Density = RoiList.Bone.Density";
Store.At.MyCommand.Execute = "";
Store.StringAt.MyCommand = "RoiList.Current.Name = #Bone";
Store.At.MyCommand.Execute = "";
```

Note #Bone is a string literal delimiter.

6.5 Numeric Operations

Code	Value
Store.FloatAt.MyFloat = 1.23;	1.23
Store.At.MyFloat.Add = 23.0;	24.23
.Subtract = 1.0;	23.23
.Multiply = 2.0;	46.46
.Divide = 2.0;	23.23

Code	Value
Store.FloatAt.MyFloat = 9.0;	9.0
Store.At.MyFloat.Square Root = "";	3.0
.Square = "";	9.0
.Negate = "";	-9.0
.Invert = "";	1/9.0 = 0.111111

Code	Value
Store.FloatAt.MyFloat = 1.78;	1.78
Store.At.MyFloat.Round = "";	1.0

Code	**Value**
`Store.FloatAt.MyFloat = 1.78;`	1.78
`Store.At.MyFloat.Nint = "";`	2.0

Code	**Value**
`Store.FloatAt.MyFloat = -1.78;`	-1.78
`Store.At.MyFloat.Absolute = "";`	1.78

These operatores support queries:

```
Store.FloatAt.MyFloat = Store.FloatAt.MyOtherFloat.Round;
Store.FloatAt.MyFloat = Store.FloatAt.MyOtherFloat.Nint;
Store.FloatAt.MyFloat = Store.FloatAt.MyOtherFloat.Absolute;
```

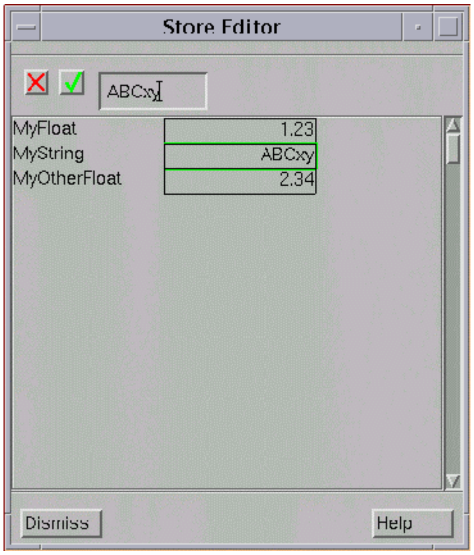
6.5.1 Numeric Operation Example

```
Store.FloatAt.MyFloat = TrialList.Current.BeamList.Current.Gantry;
Store.At.MyFloat.Add = 10.0;
TrialList.Current.BeamList.Current.Gantry =
Store.FloatAt.MyFloat;
Store.FreeAt.MyFloat = "";
```

6.6 Editing the Store

- Supports Strings and Floats
- Relaunch using the same command (existing window will be launched)
- Parameters can be edited by the user.

```
WindowList.MyStoreEditor.CreateStoreEditor = "Store.Address";
```



ABCxy

MyFloat	1.23
MyString	ABCxy
MyOtherFloat	2.34

6.7 Forcing User Entry

```
WindowList.MyStoreEditor.CreateStoreEditor = "Store.Address";
// Dismiss the window
WindowList.MyStoreEditor.Unrealize = "";
// Make it modal
WindowList.MyStoreEditor.IsModal = 1;
//Re-launch the window
WindowList.MyStoreEditor.Create = "";
// Continue with script
...
```

6.8 Nesting Stores

- StringKeyDict is the class name for the Store object.
- Nesting stores allows editing of a limited set of parameters.

```
Store.At.MyPrivateStore = StringKeyDict{};
Store.At.MyPrivateStore.FloatAt.MyFloat = 1.23;
```

6.9 Custom Data

- A Store object can be created for any object in the hierarchy (e.g. ROI, beam, wedge)
- All functionality is identical to the root Store, except that:
    - The data is persistent
    - The data is copied with the object.
- The Store is specific to each instance of the class, not the object class. (e.g. two beams might have different sets of parameters.)

Example 1

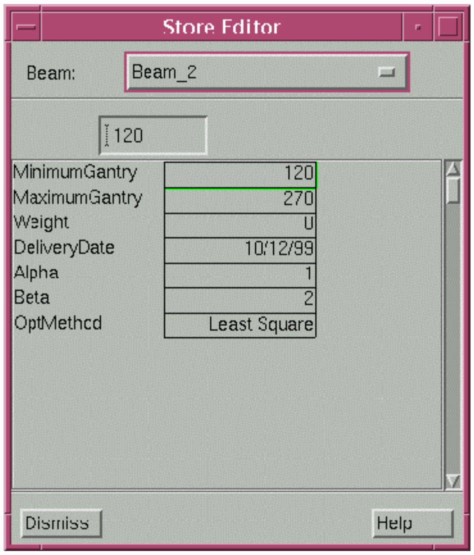
```
TrialList.Current.BeamList.
   Current.Store.FloatAt.MyFloatAttribute = 123.4;
```

This command:

- Creates a Store specific to the current beam.
- Adds a float object to the Store named MyFloatAttribute with a value of 123.4

Example 2

```
TrialList.Current.BeamList.Current.Store = {
   FloatAt.MinimumGantry = 120.0;
   FloatAt.MaximumGantry = 270.0;
   FloatAt.Weight = 0.0;
   StringAt.DeliveryDate = "10/12/99";
   FloatAt.Alpha = 1.0;
   FloatAt.Beta = 1.0;
   StringAt.OptMethod = "Least Square";
};
WindowList.BeamStoreEditor.
CreateStoreEditor = "TrialList.Current.BeamList.
      Current.Store.Address";
```



Beam: Beam_2

120

MinimumGantry	120
MaximumGantry	270
Weight	0
DeliveryDate	10/12/99
Alpha	1
Beta	2
OptMethod	Least Square

6.10 Dependencies

- When X occurs, do Y
- X is some event in Pinnacle (e.g. the current beam’s gantry angle changing, or the current beam changing)
- Y is an action (e.g. executing a script)

Example

```
KeyDependencyList.CreateChild = "";
KeyDependencyList.Last = {
   Name = "IsocenterDependency";
   KeyString = "TrialList.Current.BeamList.Current.Iso-
center.ZCoord";
   AddAction = "ViewWindowList.#0.SlicePosition =
              TrialList.Current.BeamList.Current.Isocenter.ZCo-
ord";
    };
```

- Pans the first viewing window to the isocenter slice.
- Pans the image when:

• The isocenter Z coordinate changes	ZCoord
• The beam’s isocenter changes	Isocenter
• The current beam changes	BeamList.Current
• The current trial changes	TrialList.Current

6.10.1 Removing a Dependency

```
KeyDependencyList.CreateChild = "";
KeyDependencyList.Last = {
   Name = "IsocenterDependency";
   KeyString = "TrialList.Current.BeamList.Current.Iso-
center.ZCoord";
   AddAction = "ViewWindowList.#0.SlicePosition =
      TrialList.Current.BeamList.Current.Isocenter.ZCoord";
    };
KeyDependencyList.IsocenterDependency.Destroy = "";
```

6.10.2 Executing a Script from a Dependency

- Executes MyRoiScript.Script whenever the number of ROIs changes.
- ExecuteNow executes the specified script immediately.

```
KeyDependencyList.CreateChild = "";
KeyDependencyList.Last = {
   Name = "NewRoiDependency";
   KeyString = "RoiList.Count";
   AddAction = "Script.ExecuteNow = #/home/pinnbeta/Scripts/
MyRoiScript.Script";
   };
```

6.11 Avoiding ‘vi’

Example 1

```
WaitMessage = "Checking disk space...";
SpawnCommand = "df > /usr/tmp/MyDiskSpace";
Store.StringAt.MyEditCommand = "xterm -e ";
Store.At.MyEditCommand.AppendString = GetEnv.EDITOR;
```

Note Use the EDITOR environment variable.

```
Store.At.MyEditCommand.AppendString = " /usr/tmp/MyDiskSpace";
SpawnCommand = Store.StringAt.MyEditCommand;
Store.FreeAt.MyEditCommand = "";
SpawnCommand = "rm -f /usr/tmp/MyDiskSpace";
WaitMessageOff = "";
```

Example 2

```
WaitMessage = "Checking disk space...";
SpawnCommand = "df > /usr/tmp/MyDiskSpace";
Store.StringAt.Editor = GetEnv.EDITOR;
IF.Store.StringAt.Editor.IsNull.THEN.Store.StringAt.Editor =
"vi";
```

Note The line above does error checking.

```
Store.StringAt.MyEditCommand = "xterm -e ";
Store.At.MyEditCommand.AppendString = Store.StringAt.Editor;
Store.At.MyEditCommand.AppendString = " /usr/tmp/MyDiskSpace";
WarningMessage = Store.StringAt.MyEditCommand;
```

Note The line above is a debuggin message and displays the edit command.

```
SpawnCommand = Store.StringAt.MyEditCommand;
Store.FreeAt.MyEditCommand = "";
Store.FreeAt.Editor = "";
SpawnCommand = "rm -f /usr/tmp/MyDiskSpace";
WaitMessageOff = "";
```

7 Example Scripts

7.1 Loading a dose distribution

It is possible to compute a dose distribution outside of Pinnacle^3 and load this distribution into a beam. Obviously great care must be taken with this option and it should be used in RESEARCH MODE ONLY.

The dose is loaded from a binary file. Dose is assumed to be either a relative quantity or in units of cGy/MU. Each element of the dose array is a 4-byte floating point number. The data is stored like a C array with the X dimension changing fastest.

The message used to insert a dose distribution is "DoseVolume", which is sent to a beam object, as in the following example:

```
TrialList.Current.BeamList.Current.DoseVolume = \BOB{L}:/files/
rtp/MyDose.img\;
```

The DoseVolume attribute points to a binary file. The format of the right hand side of the DoseVolume line is: \BinaryType:FileName\

There are various binary file types which can be used:

- External Data Representation. Best when you have various platforms with different byte order. (This is what Pinnacle^3 uses to save the dose distributions.)
- {L} Binary data with little-endian byte order.
- {B} Binary data with big-endian byte order.
- Hexadecimal data with big-endian byte order.

The FileName is a standard Unix filename. Alternatively, an index can be specified, in which case the system looks for the name of the script with ".binary.<Index>" appended.

For example, if the script is named /files/rtp/InsertDose.Script and it contains the following:

```
TrialList.Current.BeamList.#"#0".DoseVolume = \BOB{L}:0\;
TrialList.Current.BeamList.#"#1".DoseVolume = \BOB{L}:1\;
```

It will look for the binary data in files:
/files/rtp/InsertDose.Script.binary.000 and /files/rtp/InsertDose.Script.binary.001.

The system will assume that the array dimensions match the dose grid dimensions. Thus it is often necessary to specify the dose grid geometry before inserting the dose distribution. The dose grid geometry is set using the "Dimension", "VoxelSize", and "Origin" messages, which are sent to the DoseGrid sub-object of the Trial object.

```
TrialList.Current.DoseGrid =
   Dimension.X = 44;
   Dimension.Y = 73;
   Dimension.Z = 35;
   VoxelSize.X = 0.4;
   VoxelSize.Y = 0.4;
   VoxelSize.Z = 0.4;
   Origin.X = -8.022;
   Origin.Y = -29.866;
   Origin.Z = -6.216;
   ;
TrialList.Current.BeamList.Current.DoseVolume = BOBL:/files/
rtp/MyDose.img
TrialList.Current.BeamList.Current.MonitorUnitsValid = 1;
```

For an array with units of cGy/MU, the "MonitorUnitsValid" attribute of the beam should be set to 1. If the dose is relative, set "MonitorUnitsValid" to 0.

7.2 Extracting the block outline for a beam

The blocks for a beam are stored by definition. The actual final block outline is not stored. For example, the beam data file will store the rule that the block should expose some structure with a given margin. It is sometime useful to extract the actual block contour, for export to a block cutter, for example.

The internal block contour which results from all automatic and manual block definitions can be extracted by messaging the BlockContourList sub-object of the Beam object. The BlockContourList stores an array of contours which define the edge of the blocks. Note that all edges will be included, so there may be some ambiguity with "island" blocks where two contours will be present in the list.

If the beam blocking is described with a single contour, this contour can be extracted to a file as follows:

```
TrialList.Current.BeamList.Current.BlockContourList.
Current.WriteToFile = "/files/rtp/Blocks.out";
```

The "WriteToFile" message is a special message of the CurveNd object which writes the points in the curve to the file preceded by the number of points. For example:

```
24
22.3 11.2
22.5 10.4
...
```

For a beam with multiple block contours, the entire block contour list can be sent to a file. In this case, the regular object persistence mechanism is used.

```
TrialList.Current.BeamList.Current.BlockContourList.
Save = "/files/rtp/BlockList";
```

This results in a file like the following:

```
CurveNd =
   NumberOfDimensions = 2;
   NumberOfPoints = 64;
   Points[] =
      -0.125,-5.275,
      1.375,-5.275,
      1.925,-5.225,
      ...
   ;
;
 CurveNd =
    NumberOfDimensions = 2;
    NumberOfPoints = 34;
   Points[] =
       -2.875,-2.525,
      -2.725,-2.925,
      -2.525,-3.175,
      ...
   ;
;
```

7.3 Changing an automatic block to a manual block

It is sometimes desirable to generate a block automatically and then edit the resulting block outline. To do this the block must be converted from an automatic block to a manual block. The following script converts an automatic block to a manual block:

```
// This script will copy an automatically generated block contour into
// a manual block contour.
// NOTE: This script will only copy the first automatically generated contour.
// It will not copy multiple automatic contours
// Before running this script, create a beam, add a block to the beam,
// and generate an automatic block for the target volume.
// Delete any existing manual contours.
TrialList.Current.BeamList.Current.
   ModifierList.Current.ContourList.DestroyAllChildren = "";
// Create a new contour
TrialList.Current.BeamList.Current.
   ModifierList.Current.ContourList.CreateChild = "";
// Copy the number of points.
TrialList.Current.BeamList.Current.
   ModifierList.Current.ContourList.Current.NumberOfPoints =
TrialList.Current.BeamList.Current.BlockContourList.Current.NumberOfPoints;
// Copy the points. (In version 1.5, a simple Copy could be used. For older
// versions, must use PointsAsBlob.)
TrialList.Current.BeamList.Current.
   ModifierList.Current.ContourList.Current.PointsAsBlob =
TrialList.Current.BeamList.Current.BlockContourList.Current.PointsAsBlob;
// Set the margin to zero. Otherwise the margin will be applied to the
// new manual contour and it won't match the auto-generated contour.
TrialList.Current.BeamList.Current.
   ModifierList.Current.Margin = 0.0;
// Set the block to be manual instead of automatic.
TrialList.Current.BeamList.Current.
   ModifierList.Current.StructureToBlock = "Manual";
```

7.4 Forcing a viewing window to track the beam isocenter

Using a dependency, it is possible to specify that a given viewing window should display the slice containing the current beam's isocenter, and pan to the proper slice when the isocenter is moved, or when a new beam becomes current.

The following script will set up such a dependency. It sets the first window (window 0) to always display the slice corresponding to the isocenter for the current beam.

```
KeyDependencyList.CreateChild = "";
KeyDependencyList.Last =
   Name = "IsocenterDependency";
   KeyString = "TrialList.Current.BeamList.Current.Iso-
   center.ZCoord";
   AddAction = "ViewWindowList.0.SlicePosition =
TrialList.Current.BeamList.Current.Isocenter.ZCoord";
;
```

The first viewing window will be panned to the proper slice when any of the following occurs:

- Z coordinate of isocenter is modified.
- The beam is aimed at a different isocenter (point of interest)
- A different current beam is selected.
- A different current trial is selected.

The following script removes the dependency:

```
KeyDependencyList.IsocenterDependency.Destroy = "";
```

7.5 Timing a process

This example demonstrates how to create a Timer object to time a given process. It also demonstrates use of the temporary object database to store the temporary Timer object.

```
Store.At.Clock = Timer ; // Create Timer object. Reference as
"Clock".
SavePlan = ""; // Save all plan files.
Echo = "Elapsed time to store plan: ";
Echo = Store.At.Clock.ElapsedTime; // Query elapsed time from
timer.
Store.FreeAt.Clock = ""; // Destroy the timer.
```

The first line of the script creates a new Timer object. The curly brackets {} are the syntax used to indicate that a new instance of the class named "Timer" should be created. Initial values for the object could be placed inside these brackets. The "Store.At.Clock" message tells the system to store the object using the name "Clock". Messages can then be routed to the object through "Store.At.Clock".

7.6 Adding four blocked beams

The following is the "four field prostate" script which is one of the scripts shipped with Pinnacle^3.

```
TrialList.Current.BeginMessageBatch = "";
/* Add a new beam. */
TrialList.Current.BeamList.CreateChild = "";
/* Set the gantry angle to zero for the last (newly added) beam.
*/
TrialList.Current.BeamList.Last.Gantry = 0.0;
/* Add a new block. */
IF.RoiList.ContainsObject.prostate.THEN.
   TrialList.Current.BeamList.Last.ModifierList.Create-
   Child.ELSE.
   WarningMessage = "No blocking added because no ROI named
   'prostate'";
/* Automatically block the current ROI. */
TrialList.Current.BeamList.Last.ModifierList.Last.StructureToB-
lock
   = RoiList.Current.Name;
/* Set the automatic blocking margin to 1.0 cm. */
TrialList.Current.BeamList.Last.ModifierList.Last.Margin = 1.0;
/* ============ 90 ================================== */
/* Copy last beam (one which was set up above.) This copies block-
ing.*/
TrialList.Current.BeamList.Copy =
   TrialList.Current.BeamList.Last.Address;
/* Set gantry to 90 degrees. */
TrialList.Current.BeamList.Last.Gantry = 90.0;
TrialList.Current.BeamList.Last.NextColor = "";
/* ============ 180 ================================== */
TrialList.Current.BeamList.Copy =
   TrialList.Current.BeamList.Last.Address;
/* Set gantry to 180 degrees. */
TrialList.Current.BeamList.Last.Gantry = 180.0;
TrialList.Current.BeamList.Last.NextColor = "";
/* ============ 270 ================================== */
TrialList.Current.BeamList.Copy =
   TrialList.Current.BeamList.Last.Address;
/* Set gantry to 270 degrees. */
TrialList.Current.BeamList.Last.Gantry = 270.0;
TrialList.Current.BeamList.Last.NextColor = "";
TrialList.Current.EndMessageBatch = "";
```

This script demonstrates various features:

- The "Copy" message used to copy the first beam to the 3 final beams is a message handled by ObjectList. It creates a copy of the object specified on the right hand side of the message and adds it to the list.
- The "IF" conditional message can be used for simple branching.
- The "BeginMessageBatch" and "EndMessageBatch" messages can be sent to any object when a series of messages is going to be sent to the object. The object may or may not behave differently when in *message batch mode*. This prevents lengthy operations from being performed repeatedly during filing or script playback. For example, the Trial object does not sum together dose grids while in message batch mode. When it sees the "EndMessageBatch" message, it sums the beam dose grids.

7.7 Sorting a list of objects

An ObjectList can be sorted by any child object key. The following example sorts the beam list by couch angle in ascending order:

```
TrialList.Current.BeamList.SortBy.Couch = "";
```

Multiple keys can be specified:

```
TrialList.Current.BeamList.SortBy.Couch.Gantry = "";
```

To do a descending sort, precede the attribute with "D" as in the following example where the primary sort key is the couch angle in ascending order, and the secondary sort key is the gantry angle in descending order.

```
TrialList.Current.BeamList.SortBy.Couch.D.Gantry = "";
```

7.8 Adding an ROI curve

```
// Create a new region-of-interest.
CreateNewROI = "";
```

Note You must use CreateNewROI, not RoiList.CreateChild.

```
// Define the points for a new curve.
RoiList.Last.EditCurve = {
   SliceCoordinate = 0.7;   // Slice position (cm)
   Orientation = "Transverse";   // (Optional - transverse
default)
   Curve = {
      NumberOfPoints = 5;
      Points[] = {
         1.0, 7.0,     // These are CT coordinates (cm).
         1.0, -7.0,
         -2.0, -7.0,
         -2.0, 7.0,
         1.0, 7.0
         };
      };
   };
// Add the new curve to the ROI
RoiList.Last.CopyEditCurveToNewCurve = "";
```

Note You must use the indirect method since ROI curves do not support messaging.

7.9 Controlling Viewing Windows

For a window named XYZ: (See the BEV selection menu for names.)

```
ViewWindowList.XYZ.Orientation = "Sagittal";   (or "Transverse" or "Coronal")
ViewWindowList.XYZ.InterpMode = "Nearest Neighbor Interpolation";
(or "Linear Interpolation")
ViewWindowList.XYZ.Enter2dMode = 1;
   .Enter3dMode = 1;
   .SetNextTrial = 1;
   .NextImage = 1;
   .PreviousImage = 1;
   .SliceNumber = 23;
   .UserSliceNumber = 24;
   .SlicePosition = 13.5;
   .PanToFraction = 0.5; // Pan to slice which is 50% through data set.
   .ScaleZoom = 2.0; // Doubles the current zoom factor.
   .Zoom = 1.0; // Sets zoom to 1.0 (life size.)
   .ZoomToWindow = 1; // Fit slice to window
```