and

Scripting on the Pinnacle^3 Treatment Planning System

by

Sean Geoghegan, PhD MACPSEM
Medical Physicist

Medical Engineering and Physics
Royal Perth Hospital

29 August 2007

WARNING

This document is written for use at Perth Radiation Oncology and Royal Perth Hospital in Western Australia. The information contained in this document may not be suitable for use at other institutions. Any use of the information contained in this document at sites other than Perth Radiation Oncology and Royal Perth Hospital in Western Australia is at the user's own risk. Any use of information contained in this document at Perth Radiation Oncology or Royal Perth Hospital in Western Australia must be approved by the Medical Physicist responsible for the Pinnacle^3 treatment planning system.

1 Contents

2 Overview

This document describes scripting techniques available with the Pinnacle^3 treatment planning system and provides descriptions of some of the scripts used at Perth Radiation Oncology and Royal Perth Hospital in Western Australia. Users at Perth Radiation Oncology and Royal Perth Hospital use Pinnacle^3 scripts to modify their system for clinical and research purposes. Under no circumstances is scripting to be used for any purpose unless the user of the script fully understands the possible consequences of using the script including any possible errors that may be introduced into a treatment plan or treatment data. Any script should be used carefully because Pinnacle^3 scripts can cause serious errors on the system including system crashes and loss of data. It is possible for data to be corrupted and, if used clinically, for incorrect treatments to be planned and possibly delivered. This document is produced for Perth Radiation Oncology and Royal Perth Hospital and any use of scripts based on this document at locations other than Perth Radiation Oncology or Royal Perth Hospital is at the user's own risk. Even at Perth Radiation Oncology and Royal Perth Hospital, all scripts must be validated by the Medical Physicist responsible for the Pinnacle^3 treatment planning system prior to use on the clinical system. A similar procedure is recommended for other sites.

The scripting language available in the Pinnacle^3 treatment planning system is extremely versatile and powerful. It can be used to automate tasks in planning, such as simple scripts that define what colours are used for beams, to more complicated scripts that generate an entire plan based on outlined regions of interest. Pinnacle^3 scripting can also be used by the medical physicist when commissioning the system, reducing time taken to import electron output factors and other tedious and error prone tasks. Radiation oncologists can get some benefit from Pinnacle^3 scripting with each plan being loaded with identical settings making the system consistent and simpler to use.

Unfortunately, due to the very versatility and power of Pinnacle^3 scripting, scripting is not supported by Phillips except for very specific uses, and most Pinnacle^3 scripting development is left to experienced Pinnacle^3 users. This means that various centres have developed their own scripting tools. Some centres have limited themselves to very simple scripts. Other centres have put a great deal of effort into script development once they have recognised the advantages of using them.

This document covers all aspects of scripting that are familiar to the Medical Physicists at Royal Perth Hospital and Perth Radiation Oncology. This document provides examples of script development and application by going through a series of scripts that various authors from around Australasia and the rest of the world have kindly made available to the author or were made available through the Pinnacle^3 mailing list. There is a CD containing a library of scripts that can be implemented at other centres using the Pinnacle^3 treatment planning system with only minor modifications to suit each centre. The author hopes that this document and script library will become the basis of a collaborative working group to further the development and distribution of useful and easily adaptable Pinnacle^3 scripts.

Because there is little Pinnacle^3 scripting support available from Phillips, some of the text in this document is based on educated guesses on the internal workings of Pinnacle^3. As such, there is scope for

improvement to this document, and any errors should be reported to the author. All scripts in this document must be tested before use, and it is possible that they will not work as intended. This is especially true at other institutions where the scripts have not been tested on nor configured for use on these computing systems. This document and the script library are to be used at your own risk.

3 Acknowledgements

This document and the associated script library were developed over many years by many workers, some of whom are unknown to the author. The following people are known to the author who have either contributed to the scripts in the script library used at Perth Radiation Oncology and Royal Perth Hospital or provided script examples and discussions that helped the author write this document:

Wendy Arancini, Perth Radiation Oncology, Australia
Gary Goozee, Liverpool Hospital, Australia
Joe Hanley, Hackensack University Medical Center, USA
Scott Neil, Radiation Oncology Resources, USA
Alberto Perez-Rozos, Hospital Universitario Virgen de la Victoria, Spain Pierre-Alain Tercier, Hôpital Fribourgeois, Switzerland
Erik Van Dieren, Haga Hospital, The Netherlands
Simon Woodings, Royal Perth Hospital, Australia
Chuan Wu, University of California – Davis, USA

If there are any errors in this document (which there are bound to be) then none of errors in this document are attributable to any of these people.

4 Basic Pinnacle^3 Control Language

4.1 Pinnacle^3 Framework

The Pinnacle^3 treatment planning system is designed and programmed using an object oriented approach, that is, the features of the treatment planning system that are used by clinicians, planners and physicists have an intuitive feel of being actual real or abstract objects, such as a patient or a beam. Pinnacle^3 is designed around three essential components, the model, the controllers and the views. The model calculates dose distributions and images (such as DRRs) and responds to messages from the controllers delivered to the model using text messages that form the basis of the Pinnacle^3 scripting language. These text messages are sent by the controllers either by the user modifying parameters in the graphical user interface (the Planning windows), which sends Pinnacle^3 text messages (equivalent to Pinnacle^3 script commands) to the model one at a time, or through provision of commands directly to the controller via a script file. The views also interrogate the model to display the data in a form convenient to the user, such as via the graphical user interface, printout or a data file. These three components essentially operate independently of each other allowing the user to view changes in the model as the commands given to the model via the controller are interpreted.

Because the underlying communication between the three components of Pinnacle^3 is in the form of messages, it is possible to script a sequence of messages or record the messages sent while the user is operating the system. This allows the emergence of powerful user-written scripts that enable the user to customise and automate the system allowing significant improvements in productivity and consistency when planning.

4.2 Objects and Classes

The fundamental programmatical elements of Pinnacle^3 are objects. An object is a grouping of data and computer processes designed in such a way that real objects (such as a patient) or virtual objects (such as a beam) are represented and modelled. For example, the patient may be represented by the CT dataset, their name, and other personal details (the data) which have the processes attached so as to allow the user to change the patient's name and other details. A beam object, which is abstract and not completely physically real, has data describing its orientation, energy, field size and other details as well as having processes attached to allow the addition of a block or wedge or other features. Each object has a certain Class which is the template for the object. Examples of different instances of objects include a patient and a beam. Some classes of objects can create subclasses of objects, for example a block does not exist without a beam, and a beam does not exist outside a trial. Messages are sent to the highest class object available (in planning this is the Trial object) which propagates the message down the hierarchy of objects until an object is found to accept the message and interpret it. If the message is not matched to any object then no action is taken. Because all planning is done from within a particular plan, the higher level objects that may exist, such as Pinnacle and Plan, are not directly referenced in scripting and the highest level of class available for an object is Trial. As such, the addressing of messages is fairly straight forward, so long as the user knows the class structure.

4.3 Lists

Access to the various objects in Pinnacle[3] is through lists of the objects. The message must specify for which object in the list the message is intended. For example, each patient may have associated multiple plans, each plan multiple trials and each trial multiple beams. Sometimes the lists are associated with a higher class level object than the active view, for example the list of regions of interest and the list of points of interest are associated with the plan and not the trial. This explains some of the features in Pinnacle[3] that sometimes confuse new users to the system, for example changing the isocentre point of interest in one trial will invalidate dose in the other trials if the same point is used as an isocentre in the other trials. Examples of objects referenced using lists are:

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

TrialList.Current.BeamList.Current.Collimator
```

Both these objects refer to the currently selected beam in the currently selected trial with the first referring to the gantry angle and the second to the collimator angle. Note that a period is used between the elements of the object reference to delimit the keywords. The object references

```
TrialList.ph1.BeamList.rt_lat.Gantry

TrialList.ph2.BeamList.ap.Gantry
```

refer to the gantry angles of the beams named “rt_lat” and “ap” in the trials “ph1” and “ph2” respectively. There are other reference methods that are described in Section 6.4.

4.4 The Escape Character “#” and Special Characters “#”, “*” and “@”

When naming elements of an object list there is a special escape character which enables an individual element of the list to be identified by number or allow the asterisk character to be used which allows all elements of the list to be referenced in one go. The hash sign “#” is used as the escape character which makes Pinnacle literally interpret any following string including any special characters that are contained in the string (a hash “#” again, an asterisk “*” or an at symbol “@”). For example, the statements

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

and

```
TrialList.#"Current".BeamList.#"Current".Gantry
```

all refer to the identical object whereas

```
TrialList."Current".BeamList."Current".Gantry
```

refers to the beam named “Current” in the trial named “Current” which may not be the currently selected beam in the currently selected trial. Note that the name “Current” should not be used for any Pinnacle object because of the possibility of confusion, especially since the name of the object does not need to be included in quotes for Pinnacle to interpret the reference correctly (so long as it is unambiguous).

If the hash character or the asterisk character is used in the string escaped by the hash character, then these are interpreted literally by Pinnacle and identified as special cases. The statement

```
TrialList.#"#0".BeamList.#"#2".Gantry
```

refers to the third beam in the first trial in the current plan. Note that Pinnacle starts counting at zero and the first element is "#0". The statement

```
TrialList.#"#1".BeamList.#"*".Gantry
```

refers to all beams in the second trial in the current plan.

The at symbol "@" can be used in place of the asterisk "*" to iterate through the elements in a list. This is used when an action is applied to each element in the list in turn. Examples of the various methods of using these special characters are given in Section 6.

4.5 Actions, Data and Data Types

Because the objects have two types of functions, one to store data (for example the gantry angle of a beam) and the other to perform actions (for example to add a block to a beam), there are two types of object items that can be accessed: the action and the data. The data is made up of data elements that can be any combination of two types of data. The two types of data are string and float. A string is used to contain text data and a float is used to contain numeric data.

4.6 Messages

There are four types of messages and one type of fork that are used in Pinnacle. The first type of message is an action command to be executed in a script, the second is a query of an action which may call the action itself depending on the action, the third is an assignment message assigning a value to object data and the fourth is a query of the object data that returns that value of that data. The syntax of these four types of messages is

```
<ObjectAction> = "";

<ObjectAction>;

<ObjectData> = <Value>;

<ObjectData>;
```

The fork that is used in Pinnacle is the if-then-else construction

```
IF.<ObjectData1/Value1>.<TEST>.<ObjectData2/Value2>.THEN.
<ObjectAction1>.ELSE.<ObjectAction2/ObjectData3> = <Value3>;
```

where the test comes from one of the keywords in Table 1. The test key word used depends on the type of data being compared; that is, whether the type is either a float or a string. It is possible to replace the test with a number which evaluates to false if the number is zero or true otherwise. If an assignment message is required if the test is true, then the following construction for the complete if-then-else statement involving the negation operator "!" is used:

Table 1 Boolean test keywords and special tests and operators used in the if-then-else fork construction in the Pinnacle[3] treatment planning system scripting language.

Float	*String*	*Special*
EQUALTO	*CONTAINS*	*0 = false*
GREATERTHAN	*IS*	*1 = true*
GREATERTHANOREQUALTO	*ISNULL*	*! = not*
LESSTHAN	*STRINGEQUALTO*	*AskYesNo*
LESSTHANOREQUALTO	*STRINGNOTEQUALTO*	
NOTEQUALTO		

```
IF.<[ObjectData1|Value1]>.<TEST>.<[ObjectData2|Value2]>.
THEN.<[ObjectAction2|ObjectData3]> = <Value3>;
IF.!<[ObjectData1|Value1]>.<TEST>.<[ObjectData2|Value2]>.
THEN.<[ObjectAction3|ObjectData4]> = <Value4>;
```

Note that the endings of the messages and fork are indicated by a semicolon. The value used in an action message can be a string, a float (that is a number) or a query. If the object item refers to a process, then to make it an action instead of a query an empty string is used for the value. All strings need to be enclosed in double quotation marks. For example:

```
TrialList.Current.BeamList.Current.Name = "rt_lat";
```

Sets the beam name for currently selected beam in the currently selected trial to "rt_lat",

```
TrialList.Current.BeamList.Current.Gantry = 90;
```

Sets the gantry angle for currently selected beam in the currently selected trial to 90 degrees,

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

sets the gantry angle for currently selected beam in the currently selected trial to be the same as the collimator angle for currently selected beam in the currently selected trial.

Because a semicolon is required to indicate the end of a message, messages can be spread over multiple lines. It is also possible to nest messages by using braces. For example:

```
TrialList.Current.BeamList.Current = {
    Name = "sup";
    Gantry = 270;
    Collimator = 180;
    Couch = 90;
    };
```

sets the beam name to "sup", gantry angle to 270 degrees, collimator angle to 180 degrees and the couch angle to 90 degrees for currently selected beam in the currently selected trial.

4.7 Comments

A message that is preceded by a double slash "//" is ignored by the Pinnacle interpreter allowing the user to add comments to messages and scripts. For example the following message

```
// Set up the current beam in the current trial
//
// Here we set up the beam according to protocol
//
TrialList.Current.       // With the current trial
  BeamList.Current       // select the current beam
    = {                  // and
      Name = "sup";      // set the name to "sup",
      Gantry = 270;      // the gantry to 270,
      Collimator = 180;  // the collimator to 180
      Couch = 90;        // and the couch to 90
      };
```

is equivalent to

```
TrialList.Current.BeamList.Current.Name = "sup";
TrialList.Current.BeamList.Current.Gantry = 270;
TrialList.Current.BeamList.Current.Collimator = 180;
TrialList.Current.BeamList.Current.Couch = 90;
```

It is important to use comments in any script to make it understandable to future users of the script, especially if the script is to be used routinely and if it is to be maintained as treatment protocols and data requirements change.

5 Creating Pinnacle^3 Scripts

5.1 Recording Scripts

A user is able to record scripts directly from the Pinnacle^3 treatment planning system. It is useful to start a script by first recording it and then editing it to make it run more efficiently. More information on recording scripts is found in the Pinnacle^3 manuals.

5.2 Saving Scripts

It is possible, from within a script, to save the values of the parameters of an object by using the "Save" message. For example

```
TrialList.#"#0".Save = "/home/p3rtp/trial_1.dat";
```

saves all the data and construction of the first trial in the current plan to the file "trial_1.dat" in the p3rtp home directory. This saved data is in the form of a script and can be loaded as a script using the "ExecuteNow" message.

The location of the script files should be consistent. In this document, and at Perth Radiation Oncology, the scripts are stored in the default user script location

```
/usr/local/adacnew/PinnacleSiteData/Scripts
```

which is also the "Public" scripts folder in the standard Pinnacle^3 configuration. Within this scripts folder is the HotScriptList.p3rtp file which is used by Pinnacle^3 to list the hot scripts that are available to the user. At Perth Radiation Oncology the scripts are stored in subdirectories

```
admin
clinical
development
library
physics
private
```

Each of these subfolders is used to store particular types of scripts. The "admin" folder contains those scripts used in the system administration of the Pinnacle^3 treatment planning system software, patient data and physics data which can be shared with other sites. The "clinical" folder stores all scripts used in the preparation of a clinical plan which can be shared with other sites. The "development" folder contains the scripts under development which should not be shared with other sites except in the development of the scripts. The "library" folder contains common scripts which are used by other scripts, for example scripts to get input from the user or other scripts that use common variables. The "physics" folder contains scripts that are particular to commissioning of the Pinnacle^3 treatment planning system, for the checking of plans or for other quality assurance or data acquisition purposes. The "private" folder contains scripts that are for the use of the site that are not to be shared and should be protected for some reason. The "private" folder may include examples of working scripts from the other groups (clinical, library or physics).

For consistency, all scripts should be saved in these locations in directories within each of these subdirectories. Old versions of the scripts should be moved to a folder named "old" within each individual script folder, for example the folder

```
/usr/local/adacnew/PinnacleSiteData/Scripts
       /physics/handcalc/old
```

contains the old versions of the handcalc script that is contained in the handcalc directory of the physics directory within the default script location.

The filename to which the scripts are saved should end with an appropriate extension. Pinnacle^3 scripts that are to be available to the user from the browse window when in planning mode should use the "Script" extension whereas those Pinnacle^3 scripts that are not to be seen by the default script browse settings should use the "script" extension.

5.3 Editing Scripts

The scripts are saved as text files that can be edited using any standard text editor including "vi". It is preferable to use a graphical text editor available on most systems. Care must be taken when editing script files that backup copies of the original files are kept. This is easily done by copying the script files and using version numbering to identify which file is the most recent copy of the script. This allows the user to back track to an earlier point if an error is found in a newer version of the script. For example the file

```
handcalc.4.3.8.script
```

is version eight of the third minor revision of the fourth major release of the handcalc script. All older versions of the script are kept in the old directory within the handcalc directory.

5.4 Transcript Logs

When a plan is open, every message sent by the controller is recorded in a transcript file. These transcript files are useful for finding out the keywords and other operations that are run in a Pinnacle^3 planning session. The transcript files are saved in the patient folders in the Pinnacle^3 patient database.

6 Using Objects

6.1 Creating an Object

A new element of a list of objects can be created by sending either a message containing the class name of the object or using the message "CreateChild", for example the action messages

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

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

both create a new beam in the list of beams in the current trial. The message

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

is even more useful in that the beam is created and the name of the beam defined. The new object is added to the end of the list of objects and can be referenced using the "Last" keyword. Using the class name when creating a new object can be used to set the initial values for that object, for example

```
TrialList.Current.BeamList.Beam = {
    Name = "sup";
    Gantry = 270;
    Collimator = 180;
    Couch = 90;
    };
```

creates a new beam in the currently selected trial, adding it to the end of the list of beams in that trial, and sets the values for the gantry, collimator and couch angles.

6.2 Sorting a List of Object

It is possible to sort a list of objects by using the "SortBy" message and applying that to a keyword or list of keywords in the object. The "SortBy" message immediately follows the name of the list to be sorted. For example the action message

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

sorts the list of beams in the currently selected trial in ascending alphabetical order of the beam names. To sort the list in descending alphabetic order of the beam names the message "D" is used preceding the sorting keyword, for example in the action message

```
TrialList.Current.BeamList.SortBy.D.Name = "";
```

sorts the list of beams in the currently selected trial in descending alphabetical order of the beam names. More than one keyword can be used to sort the list, for example

```
TrialList.Current.BeamList.SortBy.D.Gantry.Collimator.D.Name
= "";
```

sorts the list of beams in the currently selected trial first by descending order of the gantry angles, then by ascending order of the collimator angles, then by descending alphabetical order of the beam names.

6.3 Selecting an Object

An existing element of a list of objects can be made the currently selected object by sending the message "MakeCurrent", for example the action message

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

makes the last beam in the list of beams in the currently selected trial the currently selected beam.

6.4 Referencing an Object

Referencing an object in a list of objects has already been described in various sections above. All references are prefaced by a period and, contrary to elsewhere in scripts, white space is important. To summarise the various ways to reference an object in a list of objects refer to Table 2 below.

Table 2 Object references that can be used in the Pinnacle[3] treatment planning system scripting language.

Reference	***Use***	***Example***
<Name>	*Direct named reference*	*TrialList.ph1* *References trial "ph1" in the open plan*
Current	*Direct reference to the currently selected object (note that the user should avoid "Current" as a name)*	*TrialList.Current* *References the currently selected trial in the open plan*
#"<Name>"	*Direct named reference where the string "<Name>" is interpreted literally and can include special characters*	*TrialList.#"ph1"* *References trial "ph1" in the open plan*
#"#<Index>"	*Direct indexed reference with zero as the first index number*	*TrialList.#"#0"* *References the first trial in the open plan*
#""*	*Reference to all elements in a list of objects*	*TrialList.#"*"* *References all trials in the open plan*
#"@"	*Iterator that moves through all elements in a list of objects*	*TrialList.#"@"* *Iterates through each trial in turn in the open plan*

6.5 Iterating Through Each Object in a List of Objects

To cycle through each object in a list of objects and execute a script on each object in turn the at symbol "@" is used, for example

```
TrialList.Current.BeamList.ChildrenEachCurrent.#"@".Store.At.
TempCommand.Execute = "";
```

executes the command stored in the variable Store.At.TempCommand on each beam in the currently selected trial.

6.6 Destroying an Object

An existing element of a list of objects can be destroyed by sending the message "Destroy", for example the action message

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

removes the currently selected beam in the currently selected trial from the list of beams. The user must be careful before destroying any object because other objects may reference it, for example a point of interest may be referenced by a beam as its isocentre. To avoid this situation the "RelyOnList" object should be queried to check that it has no elements, for example the fork

```
IF.PoiList.Current.RelyOnList.HasNoElements.
THEN.
PoiList.Current.Destroy.
ELSE.
WarningMessage = "The POI is referenced by some other
object!";
```

will only destroy the currently selected point of interest if there are no dependencies on it.

7 Using Variables, References and Stores

7.1 The Store

Variables are saved to the Store which is a temporary object created each time a plan is opened. The Store is able to contain user defined variables of two types, Float and String as well as references to objects and other stores. The Store is an element of the Root object and is referenced as either

```
Store
```

or

```
Root.Store
```

depending on the context. The Store can hold references to objects within Pinnacle by assigning the reference to a variable within the store and it can store other stores.

It is also possible to create user defined variables inside existing objects and reference these variables using the "<*ObjectReference*>" reference instead of a reference to "Root". This can result in an ambiguous reference to a variable if the "Store" reference is used without reference to "Root" or the object reference. If ambiguity is possible then it is important to use the "Root.Store" or "<*ObjectReference*>.Store" construction, example use either

```
Root.Store
```

or

```
<ObjectReference>.Store
```

for example

```
TrialList.#"#0".Store
```

The reason that someone may want to create a custom store within an existing object is to allow only those variables to be referenced if the store is displayed or edited as a whole, for example through the use of the CreateStoreEditor message.

7.2 Creating Objects in the Store

7.2.1 Creating a Variable

A variable in the Store is created in a script using the "FloatAt", "StringAt" or "At" message passed to the Store, for example the message

```
Store.FloatAt.TempFloat = 1.234;
```

or

```
Store.At.TempFloat = Float {Value = 1.234;};
```

creates the variable TempFloat if it doesn't already exist and assigns the value 1.234, overwriting any existing value. For strings the messages

```
Store.StringAt.TempString = "Hello world!";
```

or

```
Store.At.TempString = SimpleString {
       String = "Hello world!";};
```

creates the variable TempString if it doesn't already exist and assigns the value "Hello world!", overwriting any existing string.

7.2.2 Creating a Reference

To create a reference to an object, only the "At" message is used, for example

```
Store.At.TempReference = TrialList.Current.Address;
```

creates the variable TempReference if it doesn't already exist and assigns the reference to the currently selected trial in the open plan, overwriting any existing reference.

7.2.3 Creating a Store

To create a store within the Store, only the "At" message is used, for example

```
Store.At.TempStore = StringKeyDict {};
```

creates the variable TempStore if it doesn't already exist overwriting any existing values.

7.3 Changing Objects in the Store

7.3.1 Changing a Variable

Apart from redefining a variable from scratch, it is possible to modify the value or string of a variable or an element of the referenced object by standard simple operations. For Float variables the standard mathematical operations are available. For example the messages

```
Store.FloatAt.TempFloat = 1.234;  //  1.234
Store.At.TempFloat.Add = 1;       //  2.234
Store.At.TempFloat.Subtract = 2;  //  0.234
Store.At.TempFloat.Multiply = 3;  //  0.702
Store.At.TempFloat.Divide = 4;    //  0.1755
Store.At.TempFloat.Square = "";   //  0.03080025
Store.At.TempFloat.Root = "";     //  0.1755
Store.At.TempFloat.Negate = "";   // -0.1755
Store.At.TempFloat.Invert = "";   // -5.69800570
Store.At.TempFloat.Absolute = ""; //  5.69800570
Store.At.TempFloat.Round = "";    //  5
```

end up assigning the value 5 to TempFloat. Another message available to send to Float variables is "Nint" which finds the nearest integer to the value. Note that the "Round" message rounds down. For String variables, string concatenation is possible. The following messages

```
Store.StringAt.TempString = "Hello";
Store.At.TempString.AppendString = " world!";
```

end up assigning the string "Hello world!" to TempString.

7.3.2 Changing a Reference

The values of the referenced object can be changed, for example the messages

```
Store.At.TempReference = TrialList.Current.Address;
Store.At.TempReference.Name = "ph1";
```

changes the name of the currently selected trial in the open plan to "ph1".

7.3.3 Changing a Store

The variables within a store can be changed using the existing messages already listed, for example the messages

```
Store.At.TempStore = StringKeyDict {};
Store.At.TempStore.FloatAt.TempStoreFloat = 9.876;
Store.At.TempStore.StringAt.TempStoreString = "Bye!";
Store.At.TempStore.At.TempStoreReference =
       TrialList.Current.Address;
```

creates a new store named TempStore within the Store and adds the variables TempStoreFloat, TempStoreString and TempStoreReference with initialisation value, string and reference respectively. This is particularly useful when the "CreateStoreEditor" message is used with the argument of the address of the new store, which limits the number of variables that are available for user editing.

7.4 Clearing Objects in the Store

7.4.1 Deleting a Variable

The "FreeAt" message is used to delete variables that are of the Float or SimpleString type. For example, the following two lines of code deletes both the TempFloat and TempString variables.

```
Store.FreeAt.TempFloat = "";
Store.FreeAt.TempString = "";
```

7.4.2 Removing a Reference

The "RemoveAt" message is used to remove references. For example, the following line of code deletes the TempReference reference and the object which it references.

```
Store.RemoveAt.TempReference = "";
```

This message must be used with care to avoid creating conditions where an object is removed where it is referred to later in the script. Do not use the "FreeAt" message when removing a reference because, although the referenced object is destroyed, any dependencies on that object are not removed and the system can crash.

7.4.3 Clearing a Store

The "FreeAt" or the "RemoveAt" messages can be used to delete or remove values within store as well as the store itself. For example, the following line of code deletes the TempStore reference and the object which it references.

```
Store.FreeAt.TempStore = "";
Store.RemoveAt.TempStore = "";
```

These messages must be used with care to avoid creating conditions where an object is removed where it is referred to later in the script. Do not use the "FreeAt" message when removing a reference because, although the referenced object is destroyed, any dependencies on that object are not removed and the system can crash. It is best to delete or remove all variables and references before clearing the store.

8 Subroutines in Pinnacle^3 Scripts

8.1 The Reload Script Concept

There is no simple facility within the Pinnacle^3 scripting language for creating subroutines, that is routines that can be defined at one place within a script and which can called again and again with different parameters. The facility to do this can be relatively easily created by calling external shell scripts that can take arguments and which generate another Pinnacle^3 script (a "reload" script) that is called by the original script. There are several types of shells that can be used including sh, csh, ksh and perl (which actually runs under sh). The following example shows the generation of a "reload" script and its execution

```
// Define where the home is for our scripts.
Store.At.TempScriptHome = SimpleString {
    AppendString = "/usr/local/adacnew/PinnacleSiteData";
    AppendString = "/Scripts/myscript";
    };
// Define the file name of the external script
// "my.external.scr" that generates the reload
// script "my.reload.script".
Store.At.TempExternalScript = SimpleString {
    AppendString = Root.Store.At.TempScriptHome.String;
    AppendString = "/my.external.scr";
    };
// Define the reload script file name to be
// "my.reload.script" in the script home directory.
Store.At.TempReloadScript = SimpleString {
    AppendString = Root.Store.At.TempScriptHome.String;
    AppendString = "/my.reload.script";
    AppendString = ".";
    AppendString = GetEnv.SESSION_SVR;  // Unique identifier
    };
// Build up the arguments for the csh program named
// "my.external.scr" to generate "my.reload.script".
Store.At.TempCommand = SimpleString {
    AppendString = Root.Store.At.TempExternalScript.String;
    AppendString = " ";
    AppendString = "rt_lat";  // Parameter #1 - more can used
    AppendString = " > ";
    AppendString = Root.Store.At.TempReloadScript.String;
    };
// Execute the command that will run the csh program named
// "my.external.scr" which will generate "my.reload.script".
SpawnCommand = Store.At.TempCommand.String;
// Execute the reload script "my.reload.script".
ExecuteNow = Store.At.TempReloadScript.String;
```

The reload script generated by the "my.external.scr" csh script can take multiple parameters and may look like the following

```
#!/bin/csh
# This script is used to generate a Pinnacle3 reload
# script that sets the beam name of the currently
# selected beam.

printf "TrialList.Current.BeamList.Current.Name = /"$1/""
```

Note that this particular example is an overly complicated way of achieving something that could be achieved from with the Pinnacle3 scripting language itself, however some mathematical operations and string manipulations are better and more easily performed in sh, csh, ksh or perl.

8.2 Reload Conventions

8.2.1 Standard File Locations

It is important, when developing and using scripts, for all files that are generated by the scripts (whether permanent or temporary) to be containing within the calling script's home directory. It is convenient and consistent to define the variable "TempScriptHome" to be this location, that is for example

```
// Define where the home is for our scripts.
Store.At.TempScriptHome = SimpleString {
    AppendString = "/usr/local/adacnew/PinnacleSiteData";
    AppendString = "/Scripts/myscript";
    };
```

8.2.2 Standard Variable Names

The external script and reload script file locations should be stored in the "TempExternalScript" and "TempReloadScript" variables, for example

```
// Define the file name of the external script
// "my.external.scr" that generates the reload
// script "my.reload.script".
Store.At.TempExternalScript = SimpleString {
    AppendString = Root.Store.At.TempScriptHome.String;
    AppendString = "/my.external.scr";
    };
// Define the reload script file name to be
// "my.reload.script" in the script home directory.
Store.At.TempReloadScript = SimpleString {
    AppendString = Root.Store.At.TempScriptHome.String;
    AppendString = "/my.reload.script";
    AppendString = ".";
    AppendString = GetEnv.SESSION_SVR;  // Unique identifier
    };
```

Storing the script file locations in these standard variables makes it easier for the developer of a script to maintain the script as well as for other users to understand how a script works.

8.2.3 Uniquely Named Reload Scripts

When creating a reload script, it is important that the reload script is generated with a unique filename to avoid the situation of a script being called at the same time by different users on different workstations in a networked environment. As such, it is essential that all temporary scripts files are named with the "GetEnv.SESSION_SVR" message. This is still not completely safe because it is possible that a user can call the same script from multiple plans on the same workstation before the first script has finished with its reload script.

8.2.4 Safe Permissions

Because external scripts need to be executable, it is preferable to have the scripts readable and executable by only the users who belong to the "pinnacle" group. It is also important that the script folders are also readable, writable and executable by the users who belong to the "pinnacle" group. Ensuring the ownership of the scripts stay with p3rtp can be achieved by executing the commands

```
cd /usr/local/adacnew/PinnacleSiteData/Scripts
chown –R p3rtp:pinnacle *
```

when running under super user mode. This is especially important after any upgrade to the system. In addition the permissions to the files may need to be changed by using the commands

```
chmod g+x *.scr
chmod o+x *.scr
```

(or equivalent for other external script files) so that the users who belong to the "pinnacle" group can execute these files. This convention allows other users in the "pinnacle" group (such as pinnbeta or equivalent) to use these scripts.

One method to overcome this limitation is to use the "exec" or "perl" commands to load the external script files (they only need to be readable), however this may mean that temporary files cannot be created because the p3prt use does not have write permissions to the folder.

8.3 Default Values

It is possible to use the reload script concept to create a script that is intended to be loaded when the Pinnacle^3 script is initially run and to be reset just prior to the Pinnacle^3 script ending. This allows the establishment of default values or an initialisation of Pinnacle^3 parameters that may be based on the ending conditions of the previous running of the script. For example, this could be useful when trying to set standard colours or standard isodose lines relevant to a particular type of plan that may not be possible using the existing Pinnacle^3 preferences.

8.4 External Scripting Languages

8.4.1 Bourne Shell

Bourne Shell is usually found at /bin/sh on the Solaris system. It has fairly limited utility as a scripting tool compared to other shells or languages, however it can be used for relatively simple scripts or loops. More information about Bourne Shell can be obtained by executing the command

```
man sh
```

from an xterm window.

8.4.2 C Shell

C Shell has more advanced features than the Bourne Shell and is the most advanced shell option available under the older releases of the Solaris system that came with the earlier releases of the Pinnacle[3] treatment planning system. Although C Shell is more advanced than Bourne Shell, it is still limited and requires several tricks to enable a programmer to use it for more complex scripts and loops. More information about C Shell can be obtained by executing the command

```
man csh
```

from an xterm window. Until the provision of more advanced scripting tools on the Pinnacle[3] treatment planning system all external scripts written at Perth Radiation Oncology and Royal Perth Hospital were written in C Shell.

8.4.3 Korn Shell

Korn Shell has more advanced features than either the Bourne Shell or C Shell and was not available on the Pinnacle[3] treatment planning system until more recently. More information about Korn Shell can be obtained by executing the command

```
man ksh
```

from an xterm window.

8.4.4 Perl

Perl is a scripting language that is almost perfectly designed for the tasks required of an external script called by Pinnacle[3] scripts. Perl is a language that has been described as a practical extraction and reporting language (also Pathologically Eclectic Rubbish Lister) allowing a script to interrogate a text file, perform some operation on the data contained within that file and produce a report based on those data. This is ideal for reading in the text files that the Pinnacle[3] treatment planning system produces, extracting the relevant data from these files and producing a derived script file as output (or summary report if required) either as a serial job or to loop through a series of options.

Perl only recently became available on the Pinnacle[3] treatment planning system and most of the scripts written at Perth Radiation Oncology and Royal Perth Hospital are still written in C Shell code. Because Perl is based on Bourne Shell scripting including awk and sed, it is relatively easy to learn Perl if the user is already familiar with Unix shell scripting.

Detailed information on Perl can be obtained from www.perl.com and www.cpan.org as well as the books produced by O'Reilly. A good distribution of Perl is available for free from www.activestate.com, however this distribution should only be used for learning and development on computers other than those of Pinnacle[3] treatment planning system.

All future external scripts at Perth Radiation Oncology and Royal Perth Hospital should be written in Perl.

9 Pinnacle^3 Objects and Object Lists

9.1 BeamList

The "BeamList" lists all of the beams in the currently selected trial of the currently open plan. The structure of a beam is complicated and can be obtained by saving a beam to a text file using the "Save" message, for example

```
TrialList.Current.BeamList.Current.Save =
        "/home/p3rtp/beam.dat";
```

saves the currently selected beam in the currently selected trial to the text file "beam.dat" in the p3rtp home directory.

9.2 ColorList

The "ColorList" lists all of the colours available in the currently open plan. The structure of this list is very simple and can be obtained by using the "Save" message, for example

```
ColorList.Save =
        "/home/p3rtp/colours.dat";
```

saves the colour list to the text file "colours.dat" in the p3rtp home directory.

9.3 ControlPanel

The "ControlPanel" gives the elements in the control panel of the Pinnacle^3 treatment planning system. The structure of the control panel can be obtained by using the "Save" message, for example

```
ControlPanel.Save =
        "/home/p3rtp/controlpanel.dat";
```

saves the control panel to the text file "controlpanel.dat" in the p3rtp home directory.

9.4 DVHList

The "DVHList" lists all of the DVHs calculated in the currently open plan. The structure of this list is relatively simple and can be obtained by using the "Save" message, for example

```
DVHList.Save =
        "/home/p3rtp/dvh.dat";
```

saves the DVH list to the text file "dvh.dat" in the p3rtp home directory.

9.5 ElectronEnergyList

The "ElectronEnergyList" lists all of the loaded electron energies available in the currently selected machine physics tool. The structure of an electron energy is complex and can be obtained by saving the electron energy list to a text file using the "Save" message, for example

```
MachineList.Current.ElectronEnergyList.Save =
        "/home/p3rtp/electrons.dat";
```

saves the list of electron energy data to the text file “electrons.dat” in the p3rtp home directory.

9.6 KeyDependencyList

The “KeyDependencyList” lists all of the dependencies between various objects within the currently open plan. It is possible for script messages to be tied to certain actions via a dependency, that is a script message is executed whenever a certain event happens. A dependency is established in the “KeyDependencyList” object list. Each new instance of a dependency is created as an object using the “CreateChild” message. The elements of the dependency are Name, KeyString and AddAction. The string in Name is used to reference the dependency, the string in KeyString identifies the object name to which the dependency is tied and the AddAction string defines the message to be interpreted when the value of the named object changes. For example the messages

```
KeyDependencyList.CreateChild = "";
KeyDependencyList.Last =
        {
        Name = “TempDependency”;
        KeyString = “TrialList.Current.Name”;
        AddAction = “Script.ExecuteNow =
                    #/home/p3rtp/scripts/dependency.script”;
        };
```

runs the script “/home/p3rtp/scripts/dependency.script” whenever the name of the currently selected trial changes. Note the use of the escape character the hash “#”. A dependency can be removed in the same way that any object is destroyed using the Destroy message.

```
KeyDependencyList.TempDependency.Destroy = “”;
```

The action message interpreted by a dependency does not have to be the execution of a script and can be a single line message using the dependency defined in “KeyString”. For example the “AddAction” string in the dependency defined above could be of the form

```
        AddAction = “TrialList.Current.BeamList.Current.Name =
                    TrialList.Current.Name”;
```

This would make the currently selected beam in the currently selected trial have the same name as the trial when the name of the trial is changed.

9.7 MachineList

The “MachineList” lists all of the loaded machines available in the physics tool. The structure of a machine is complex and can be obtained by saving the machine list to a text file using the “Save” message, for example

```
MachineList.Save = “/home/p3rtp/machines.dat”;
```

saves the machine list to the text file “machines.dat” in the p3rtp home directory.

9.8 MeasureGeometryList

The "MeasureGeometryList" lists all of the measurement geometries available in the currently selected modality for the currently selected machine in the physics tool. The structure of a measurement geometry energy is complex and can be obtained by saving the list of measurement geometries to a text file using the "Save" message, for example

```
MachineList.Current.PhotonEnergyList.Current
       .MeasureGeoemtryList.Save =
                  "/home/p3rtp/geoemtries.dat";
```

saves the list of measurement geometries to the text file "geometries.dat" in the p3rtp home directory.

9.9 PanelList

The "PanelList" lists all of the panels available in the windows in currently open plan. The structure of a panle list is fairly simple and can be obtained by saving the panel list to a text file using the "Save" message, for example

```
WIndowList.Current.PanelList.Save = "/home/p3rtp/panels.dat";
```

saves the panel list of the currently selected window to the text file "panels.dat" in the p3rtp home directory.

9.10 PhotonEnergyList

The "PhotonEnergyList" lists all of the loaded photon energies available in the currently selected machine physics tool. The structure of an photon energy is complex and can be obtained by saving the electron energy list to a text file using the "Save" message, for example

```
MachineList.Current.PhotonEnergyList.Save =
       "/home/p3rtp/photons.dat";
```

saves the list of photon energy data to the text file "photon s.dat" in the p3rtp home directory.

9.11 PlanInfo

The "PlanInfo" object provides all the information relating to the patient including "PatientName", "Institution", "PlanName", etc. The structure of a "PlanInfo" is fairly simple and can be obtained by using the "Save" message, for example

```
PlanInfo.Save = "/home/p3rtp/planinfo.dat";
```

saves the information relating to the currently open plan to the text file "planinfo.dat" in the p3rtp home directory.

9.12 PoiList

The "PoiList" lists all of the points of interest in the currently open plan. The structure of a point of interest is fairly simple and can be obtained by saving the points of interest list to a text file using the "Save" message, for example

```
PoiList.Save = "/home/p3rtp/pois.dat";
```

saves the points of interest list to the text file "pois.dat" in the p3rtp home directory.

9.13 PrescriptionList

The "PrescriptionList" lists all of the prescriptions in the currently selected trial in the currently open plan. The structure of a prescription is complicated and can be obtained by saving a prescription to a text file using the "Save" message, for example

```
TrialList.Current.PrescriptionList.Current.Save =
       "/home/p3rtp/prescription.dat";
```

saves the currently selected prescription in the currently selected trial to the text file "prescription.dat" in the p3rtp home directory.

It is possible to make the reference points for all beams the same as the relevant prescription points by either using a series of dependencies or running a script that iterates over all beams, for example the message

```
TrialList.Current.BeamList.ChildrenEachCurrent.#"@".Script.Ex
ecuteNow =
       "/home/p3rtp/scripts/synch_prescriptions.script";
```

iterates through all beams in the current trial and executes the "synch_prescriptions.script" script which contains

```
Store.At.TempPrescriptionName =
       SimpleString{};
Store.At.TempPrescriptionName.AppendString =
       TrialList.Current.BeamList
       .Current.PrescriptionName.String;
TrialList.Current.PrescriptionList.MakeCurrent =
       Store.At.TempPrescriptionName;
Store.At.TempPrescriptionPoint =
       SimpleString{};
Store.At.TempPrescriptionPoint.AppendString =
       TrialList.Current.PrescriptionList
       .Current.PrescriptionPoint;
TrialList.Current.BeamList.Current.PrescriptionPointName =
       Store.At.TempPrescriptionPoint;
```

Based on the contribution of Pierre-Alain Tercier, Hôpital Fribourgeois Switzerland

9.14 RelyOnList

The "RelyOnList" lists all of the object that rely on the specified object. The rely on list has is a fairly simple list and can be obtained by saving the rely on list to a text file using the "Save" message, for example

```
TrialList.Current.BeamList.Current.RoiList.Save =
       "/home/p3rtp/relyonlist.dat";
```

saves the list of objects that rely on the currently selected beam in the currently selected trial of the open plan to the text file "relyonlist.dat" in the p3rtp home directory. Refer to the "RelyOnList" message in Section 10.

9.15 RoiList

The "RoiList" lists all of the regions of interest in the currently open plan. The structure of a region of interest is fairly simple and can be obtained by saving the regions of interest list to a text file using the "Save" message, for example

```
RoiList.Save = "/home/p3rtp/rois.dat";
```

saves the regions of interest list to the text file "rois.dat" in the p3rtp home directory.

9.16 Store

The "Store" object contains the definitions of all the variables in the "Store", the definitions of which can be obtained by saving the "Store" to a file by using the "Save" message, for example

```
Store.Save = "/home/p3rtp/store.dat";
```

saves the "Store" to the text file "store.dat" in the p3rtp home directory.

9.17 TrialList

The "TrialList" lists all of the trials in the currently open plan. The structure of a trial is complicated and can be obtained by saving a trial to a text file using the "Save" message, for example

```
TrialList.Current.Save = "/home/p3rtp/trial.dat";
```

saves the currently selected trial to the text file "trial.dat" in the p3rtp home directory.

9.18 ViewWindowList

The "ViewWindowList" lists all of the viewing windows available in the currently open plan. The structure of a viewing window is fairly simple and can be obtained by saving a window to a text file using the "Save" message, for example

```
ViewWindowList.Current.Save = "/home/p3rtp/viewwindow.dat";
```

saves the currently selected view window to the text file "viewwindow.dat" in the p3rtp home directory. It is perhaps more useful to save the entire list of viewing windows by using the message

```
ViewWindowList.Save = "/home/p3rtp/viewwindowlist.dat";
```

to save the list of viewing windows in the currently open plan to the text file "viewwindowlist.dat" in the p3rtp home directory.

9.19 VolumeList

The "VolumeList" lists all of the volumes available in the currently open plan. The structure of a volume cannot be obtained by saving a volume to a text file using the "Save" message, for example

```
VolumeList.Current.Save = "/home/p3rtp/volume.dat";
```

saves the currently selected volume to the text file "volume.dat" in the p3rtp home directory, however there is little information available from this file. It is perhaps more useful to save the entire list of volumes by using the message

```
VolumeList.Save = "/home/p3rtp/volumelist.dat";
```

to save the list of volumes in the currently open plan to the text file "volumelist.dat" in the p3rtp home directory, however the specific data pertaining to each volume is not available in this file.

9.20 WindowList

The "WindowList" lists all of the windows available in the currently open plan. The structure of a window cannot be obtained by saving a window to a text file using the "Save" message, for example

```
WindowList.Current.Save = "/home/p3rtp/window.dat";
```

saves the currently selected window to the text file "window.dat" in the p3rtp home directory, however there is little information available from this file. It is perhaps more useful to save the entire list of windows by using the message

```
WindowList.Save = "/home/p3rtp/windowlist.dat";
```

to save the list of windows in the currently open plan to the text file "windowlist.dat" in the p3rtp home directory, however the specific data pertaining to each window is not available in this file. Known elements of the WidnowList include "CTSim" and "NewROISpreadsheet".

10 Pinnacle^3 Scripting Messages

10.1 Absolute

The "Absolute" message finds the absolute value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Absolute = "";
```

10.2 Add

The "Add" message adds the value of the argument to the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Add = <Value>;
```

10.3 Address

The "Address" message returns the memory address location of the referenced object. Its syntax is

```
<ObjectData>.Address;
```

For example the message

```
Store.At.TempReference = TrialList.Current.Address;
```

stores the address of the currently selected trial in the open plan in the user defined variable TempReference.

10.4 AppendString

The "AppendString" message concatenates the string in the argument to the string of the referenced String variable. Its syntax is

```
<StringVariableReference>.AppendString = <String>;
```

10.5 AskYesNo

The "AskYesNo" message opens the Pinnacle^3 dialogue box that is used when a yes-no question is asked and waits until an answer is provided. It returns a numeric value equivalent to true or false that is used in forks. Its usual syntax in an if-then-else statement is

```
IF.AskYesNo.
THEN.<ObjectAction1>.
ELSE.<[ObjectAction2|ObjectData2]> = <Value2>;
```

or

```
Store.FloatAt.TempAnswer = AskYesNo;
IF.Store.At.TempAnswer.Value.
       THEN.<[ObjectAction1|ObjectData1]> = <Value1>;
IF.!Store.At.TempAnswer.Value.
       THEN.<[ObjectAction2|ObjectData2]> = <Value2>;
```

Note the use of the not operator "!" in the second test where two values need to be set depending on the answer. The "AskYesNo" message uses the default choice and prompts defined by the "AskYesNoDefault" and "AskYesNoPrompt" messages.

10.6 AskYesNoDefault

The "AskYesNoDefault" message set the default choice (the choice that is selected when the user presses the enter key) of the prompt of the Pinnacle[3] dialogue box that is used when a yes-no question is asked. Its syntax is

```
AskYesNoDefault = [0|1];
```

A value of 0 sets the default choice to no, a value of 1 sets the default choice to yes.

10.7 AskYesNoPrompt

The "AskYesNoPrompt" message sets the text of the prompt of the Pinnacle[3] dialogue box that is used when a yes-no question is asked. Its syntax is

```
AskYesNoPrompt = <String>;
```

10.8 At

The "At" message creates a variable (either a Float or a String) or reference in the Store. Its syntax is

```
Store.At.<VariableName> =
       [
       Float {Value = <Value>;};|
       SimpleString {String = <String>;};|
       <ObjectAddress>;
       ]
```

10.9 ChildrenEachCurrent

The "ChildrenEachCurrent" message is used to iterate through each item in a object list and make each item the currently selected item for at least a short period of time. Its usual syntax is

```
<ObjectList>.ChildrenEachCurrent.#"@".<Action>;
```

10.10 ContainsObject

The "ContainsObject" message is used to test whether a list of object has an element, returning true if the list of object has an elements. Its syntax is

```
<ObjectList>.ContainsObject;
```

10.11 Count

The "Count" message returns the number of objects in <ObjectList>. Its syntax is

```
<ObjectList>.Count;
```

10.12 Create

The “Create” message opens a window from the window list. The argument on the right hand side should be a string with the text that appears in the top bar of the window. Its syntax is

```
<WindowObject>.Create = “[<WindowName>]”;
```

10.13 CreateReadOnly

The “CreateReadOnly” message opens a window from the window list and does not allow the user to modify any values within text boxes and attached to widgets associated with the specified window. The argument on the right hand side should be a string with the text that appears in the top bar of the window. Its syntax is

```
<WindowObjectReadOnly>.Create = “[<WindowName>]”;
```

This message can be used to provide a view for a script that allows the inspection of values without the risk of those values being changed.

10.14 CreateChild

The “CreateChild” message creates another instance of the objects in *<ObjectList>* and appends the object to the end of the list of these objects. The created object can be referenced using the “Last” message. Its syntax is

```
<ObjectList>.CreateChild = “”;
```

10.15 CreateStoreEditor

The “CreateStoreEditor” message is a special instance of the “Create” message that opens a special window from the window list that allows the user to edit values of Float and String variables in the Store. The argument on the right hand side is a string with the reference address of the Store. Its syntax is

```
WindowList.<Window>.CreateStoreEditor = "Store.Address";
```

For example, the message

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

creates the Store editor and identifies the it with the window name TempStoreEditor. It is possible to force the user to edit the values in the Store by using the IsModal message.

10.16 Current

The “Current” message references the currently selected item in the object list. Its syntax is

```
<ObjectList>.Current;
```

10.17 D

The “D” message is a sort key used to sort the list of object in ascending order when the “SortBy” message is used. Its syntax is

```
<ObjectList>.SortBy
        [.D].<ValueReference>
        [[.D].<ValueReference>…] = "";
```

Refer to the description of the "SortBy" message.

10.18 Destroy

The "Destroy" message destroyed the specified instance of the object in <*ObjectList*>. Its syntax is

```
<Object>.Destroy = "";
```

10.19 Divide

The "Divide" message divides the value of the referenced Float variable by the value of the argument. Its syntax is

```
<FloatVariableReference>.Divide = <Value>;
```

10.20 Echo

The "Echo" message sends a string to the console window. Its syntax is

```
Echo = <String>;
```

For example, the message

```
Echo = "Hello world!";
```

prints the string "Hello world!" to the console for when Pinnancle3 is loaded directly from the console. Loading Pinnacle^3 directly from the console is unusual and not the normal way for a normal user to load Pinnacle^3. The "Echo" message is not used except for developmental or debugging work.

10.21 Execute

The "Execute" message interprets the command that is stored in a string. Its usual syntax is of the form

```
<StringVariable>.Execute;
```

For example, the message

```
Store.StringAt.TempCommand.Execute;
```

executes the string in TempCommand where the string variable TempCommand has been created and defined to have a string value of a command line containing a script message.

10.22 ExecuteNow

The "ExecuteNow" message passes control to a specified script file which, when complete, returns control to the original script. Its syntax is

```
ExecuteNow = <String>;
```

For example, the message

```
ExecuteNow =
       "/usr/local/adacnew/PinnacleSiteData/
       Scripts/my.script";
```

runs the script "my.script" in the "Public" scripts folder (/usr/local/adacnew/PinnacleSiteData/Scripts) of the Pinnacle[3] treatment planning system.

10.23 Float

The "Float" message specifies that a variable can contain numerical data. Its syntax is

```
<Variable> = Float {Value = <Value>;};
```

Refer to the description of the "Value" message.

10.24 FloatAt

The "FloatAt" message creates a Float variable in the Store. Its syntax is

```
Store.FloatAt.<VariableName> = <Value>;
```

10.25 FreeAt

The "FreeAt" message deletes a variable (either a Float or a String) in the Store. Its syntax is

```
Store.FreeAt.<VariableName> = "";
```

The "FreeAt" message should never to be used to delete a reference.

10.26 GetEnv.SESSION_SVR

The "GetEnv.SESSION_SVR" message returns the name of the current workstation on which the plan is open. Its syntax is

```
GetEnv.SESSION_SVR;
```

10.27 HasNoElements

The "HasNoElements" message is used to test whether a list of object has no elements, returning true if the list of object has no elements. Its syntax is

```
<ObjectList>.HasNoElements;
```

10.28 Invert

The "Invert" message finds the inverse of the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Invert = "";
```

10.29 IsModal

The "IsModal" message makes a window from the window list modal, that is the window must be dismissed before any further action can be taken with the currently open plan. The argument on the right hand side should be a string with the text that appears in the top bar of the window. Its syntax is

```
<WindowObject>.IsModal = "[<WindowName>]";
```

10.30 Last

The "Last" message references the last item in the object list. Its syntax is

```
<ObjectList>.Last;
```

10.31 MakeCurrent

The "MakeCurrent" message makes the specified object the currently selected object. Its syntax is

```
<ObjectReference>.MakeCurrent = "";
```

10.32 Multiply

The "Multiply" message multiplies the value of the referenced Float variable by the value of the argument. Its syntax is

```
<FloatVariableReference>.Multiply = <Value>;
```

10.33 Negate

The "Negate" message finds the negative of the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Negate = "";
```

10.34 NextCurrent

The "NextCurrent" message specifies the next object in a list of objects based on the referenced object. Its syntax is

```
<ObjectReference>.NextCurrent = <Value|String>;
```

10.35 Nint

The "Nint" message finds the closest integer to the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Nint = "";
```

The user will need to determine how "Nint" handles number such as 0.5 in each specific case because it is possible that bit errors could make a number that appears to be 0.5 (or similar) is actually slightly larger or smaller than the quoted 0.5. The user should use the "Round" message when possible.

10.36 Quit

The "Quit" message closes the current plan without saving. Its syntax is

```
Quit = "";
```

10.37 QuitWithSave

The "QuitWithSave" message saves all data for the current plan and closes the current plan. Its syntax is

```
QuitWithSave = "";
```

The "QuitWithSave" message is equivalent to

```
SavePlan = "";
Quit = "";
```

10.38 RelyOnList

The "RelyOnList" message creates the list of objects which are dependent on the specified object item. Its syntax is

```
<Object>.RelyOnList;
```

This message is often used with the "HasNoElements" message to check that the specified object can be removed without creating an inconsistency in the plan.

10.39 RemoveAt

The "RemoveAt" message removes a reference in the Store as well as the referenced object from the currently open plan. Its syntax is

```
Store.RemoveAt.<ReferenceName> = "";
```

The "RemoveAt" message should never be used to remove a variable of the Float or SimpleString type.

10.40 Root

The "Root" message takes the square root of the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Root = "";
```

10.41 Round

The "Round" message rounds down the value of the referenced Float variable to an integer. Its syntax is

```
<FloatVariableReference>.Round = "";
```

10.42 Save

The "Save" message saves all data for the specified object to a file. Its syntax is

```
<Object>.Save = <String>;
```

For example, the message

```
TrialList.#"#0".Save = "/home/p3rtp/trial_1.dat";
```

saves all the data and construction of the first trial in the current plan to the file "trial_1.dat" in the p3rtp home directory.

10.43 SavePlan

The "SavePlan" message saves all data for the current plan. Its syntax is

```
SavePlan = "";
```

10.44 SimpleString

The "SimpleString" message specifies that a variable can contain string data. Its syntax is

```
<Variable> = SimpleString {String = <String>;};
```

Refer to the description of the "String" message.

10.45 SortBy

The "SortBy" message sorts the list of object by the keywords given subsequent to the "SortBy" message. The sorting is done in ascending order. The "D" message can be used to sort in descending order. Its syntax is

```
<ObjectList>.SortBy
        [.D].<ValueReference>
        [[.D].<ValueReference>…] = "";
```

Refer to the description of the "D" message.

10.46 SpawnCommand

The "SpawnCommand" message transfers control to the underlying operating system and executes a command in that operating system. Command is returned to Pinnacle[3] when the command sent to the underlying operating system is completed. Its syntax is

```
SpawnCommand = <String>;
```

For example the message

```
SpawnCommand = "ls /home/p3rtp > /home/p3rtp/p3rtp_dir.lst";
```

Sends the command "ls /home/p3rtp > /home/p3rtp/p3rtp_directory_listing.dat" to the operating system and waits for this command to complete before returning control to Pinnacle[3] and resuming the interpretation of following messages. The command "ls /home/p3rtp > /home/p3rtp/p3rtp_dir.lst" creates a listing of the p3rtp home directory and writes that listing to the file named "p3rtp_dir.lst" in the same directory.

The "SpawnCommand" message is often used to run an operating system process, such as a csh or perl script, that generates a reload file containing Pinnacle[3] script messages. This reload file is then run using the "ExecuteNow" message.

10.47 SpawnCommandNoWait

The "SpawnCommandNoWait" message transfers control to the underlying operating system and executes a command in that operating system. Command is immediately returned to Pinnacle[3] even before the command sent to the underlying operating system is completed. Its syntax is

```
SpawnCommandNoWait = <String>;
```

The "SpawnCommandNoWait" message is equivalent to sending a no hang up command (post fixed with an ampersand "&") to the underlying operating system, for example the following two messages are identical

```
SpawnCommandNoWait = "<Command>";
SpawnCommand = "<Command> &";
```

10.48 Square

The "Square" message squares the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Square = "";
```

10.49 String

The "String" message specifies the string value of a variable of the "SimpleString" type that can contain string data. Its syntax is

```
<Variable> = SimpleString {String= <String>;};
```

or

```
<Variable>.String;
```

Refer to the description of the "SimpleString" message.

10.50 StringAt

The "StringAt" message creates a String variable in the Store. Its syntax is

```
Store.StringAt.<VariableName> = <String>;
```

10.51 Subtract

The "Subtract" message subtracts the value of the argument from the value of the referenced Float variable. Its syntax is

```
<FloatVariableReference>.Subtract = <Value>;
```

10.52 Unrelealize

The "Unrelealize" message closes a window from the window list. The argument on the right hand side should be a string with the text that appears in the top bar of the window. Its syntax is

```
<WindowObject>.Unrelealize = "[<WindowName>]";
```

10.53 Value

The "Value" message specifies the numerical value of a variable of the "Float" type that can contain numerical data. Its syntax is

```
<Variable> = Float {Value = <Value>;};
```

or

```
<Variable>.Value;
```

Refer to the description of the "Float" message.

10.54 WaitMessage

The "WaitMessage" message sends a string to the Pinnacle[3] status window and turns on the hourglass cursor. Its syntax is

```
WaitMessage = <String>;
```

The hourglass cursor will remain turned on (even after the script finishes) until the "WaitMessageOff" message is called.

10.55 WaitMessageOff

The "WaitMessageOff" message clears the Pinnacle[3] status window and returns the cursor to normal. Its syntax is

```
WaitMessageOff = "";
```

10.56 WarningMessage

The "WarningMessage" message opens a Pinnacle[3] dialogue box and displays some text. The dialogue box stops all processes in the current plan and must be dismissed before any further messages are interpreted. Its syntax is

```
WarningMessage = <String>;
```

11 Scripting Tips

11.1 MLC Leaf Positions

When recording leaf positions under MLC options, type in each leaf position explicitly, starting with the first leaf (even if this is not to change) and continue to the last of the leaves of interest with no gaps (ie, unedited leaves). There is no need to type positions for leaves beyond those of interest. Then the leaf positions 'stick' when the script is run. – Bob Chappell