Data Reduction and Visualization Environment

Version 2.0

Introduction to programing using the iTool Framework

Richard Tumanjong Azuah

(document version 1.0 – March 2008)

Table of Contents

Introduction

The purpose of this course is to provide a general introduction to the IDL iTool framework for developers who wish to contribute to the Data Analysis and Visualization Environment (DAVE) project. The iTool framework forms an integral part of a redesign of DAVE. The framework itself is very extensive and it is just not possible to adequately cover a substantial part of it in a short course such this one. The manual therefore will focus on some key aspects of the iTool framework especially as they relate to DAVE.

Since the iTool framework is built using the object oriented programing (OOP) paradigm, a brief discussion on OOP and how it is supported in IDL is given. This is followed by an overview of the iTool framework. DAVE is then introduced and the remainder of the manual focuses on tutorials about how various parts of the application are implemented.

It is worth noting that in preparing this manual, I have borrowed heavily from the book **iTool Developer's Guide** published by RSI Research Systems Inc (currently ITT Visual Information Solutions), the IDL supplier. The book is the primary source of information for developers about the iTool framework and is available in print as well as online. You are encouraged to consult it for a more detailed consideration of some of the topics introduced in the manual. And, of course, the online documentation that comes with your IDL distribution should be used to look up detailed function or class references.

Chapter 1: OOP in IDL

What is Object Oriented Programing (OOP)

Object orientation is an approach to software development in which we focus on **objects** and their **attributes** and **responsibilities**. It provides a framework in which a **direct mapping** of problems to computer code can be most easily realized.

OOP vs Procedural Programing

In conventional or procedural programing an application consists of a set of functions which operate on a set of data variables to produce the desired outcome. The functions and data are distinct and the emphasis is on the interaction between the two. Separating data from functionality typically leads to software that is difficult to maintain and understand – especially as the program size grows.

In OOP, data is treated as a critical element and is not allowed to flow freely. It bounds data closely to the functions that operate on it and protects it from accidental modification from *outside* functions. Essentially, OOP allows decomposition of a problem into a number of **well defined entities** consisting of **both data and functions**.

Advantages of OOP

- Provides a clear *modular* structure for programs.
- Easier to maintain, modify and extend existing code

Note: OOP is more than just learning new syntax, etc; it requires a new way of thinking

Basic Concepts

1. ***Objects***. An object is a computer representation of some real-world thing (i.e person, place) or event. Objects can have both attributes and behaviours.
2. ***Classes***. A class is the definition, or blueprint, for an object. Classes do not really ever exist – they are plans.

We use the class to create an object which has identity. That is, an object exists, has values in its properties and can execute behaviors. Once a class has been defined, we can create any number of objects belonging to that class. “Larry” is an instance or object of the class of “human beings”.

More about Classes

- A class defines an **abstract data type** (ADT) which can be used to instantiate or create an object of the class. In this context, an object is like a new user-defined type.
- A class definition consists of variables and functions/procedures.
- Common names for class variables are class data or **data members**.
- Common names for class functions are member functions or **methods**.

The integration of data and functions into a class is known as **encapsulation.** Encapsulation is generally considered one of the three fundamental principles of OOP. Encapsulation encourages information hiding with the use of **private** and **public** access specifiers in a class definition:

- A **private** method or data is known only within a class definition and is **not accessible** from outside the class.
- A **public** method or data is the **"published" interface** for the class and is the only way in which an object of the class communicates with the outside.

Building object based application

1) Identifying classes to define

In OOP, an application consists of a set of interacting objects. After specifying a problem, the first task is to identify the relevant players or objects in the system – this may be straightforward in some cases but tricky in general. The following guideline is useful:

In OOP, the names of places or things (nouns) are clues to potential objects and consequently classes to be defined. Actions (verbs) are clues to functions that should be associated with a class while attributes or characteristics help identify data members.

So essentially:

- Noun or noun phrases suggest objects
- Verbs suggests methods
- Attributes suggests data members

In simulating a chemical reaction, atoms, molecules and temperature would be considered objects. In a banking application, a bank account is an obvious object.

2) Flesh out class details

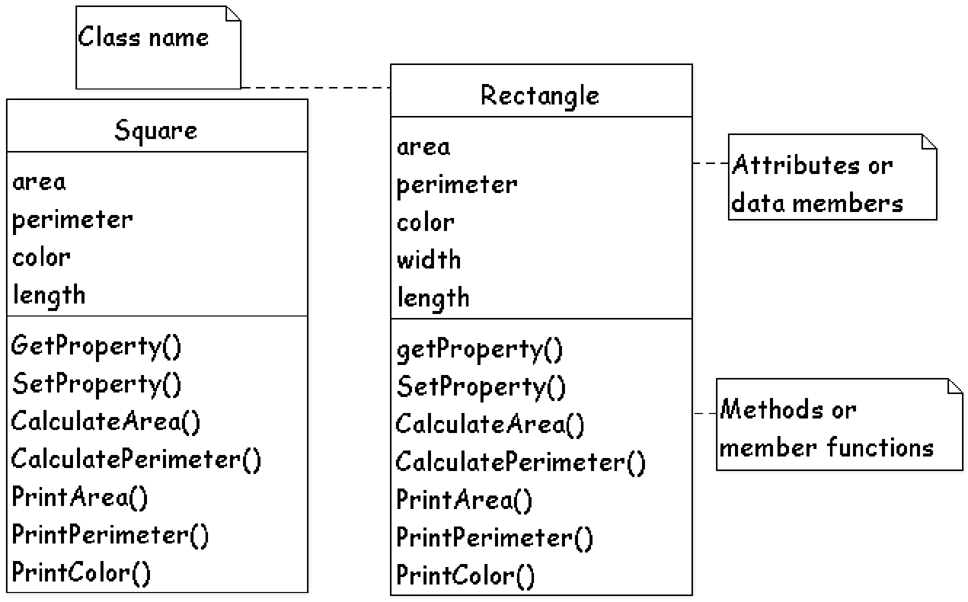
For each class, identify all the possible data members and methods that objects of the class will require. Ensure classes are modular and fairly self-contained with a clear defined interface (methods).

An atom's mass would be a data member of an atom class. A method would be required to change the oxidation state of an atom. The bank account class will include methods for withdrawing and depositing funds and for retrieving the current balance.

3) Design and Coding

Complete the design by putting it all together to form a system of interacting objects rather than simply a collection of objects. The challenges here are similar to a conventional program where functions and data need to be assembled together to obtain a working application.

Then start coding!



Defining Classes in IDL

For illustration purposes, lets imagine we are working with an application in which geometric shapes are being used. Hence objects such as square, rectangle, circle, etc have been identified. Further, careful consideration has shown that the square and rectangle classes should contain the following details as indicated in the UML diagram representation of the classes.

Data members. A named structure is used to declare data members – the name of the class structure must be the same as the class name. For the Square class this is achieved in the following procedure

```
pro Square__define
  void = {Square $
          ,area:0.0 $
          ,perimeter:0.0 $
          ,length:0.0 $
          ,color:[0B,0B,0B] $
         }
end
```

The procedure name square__define is significant (and should be saved in a file called square__define.pro).

Methods. Methods are defined in IDL as either procedures or functions with a slightly modified syntax:

The CalculateArea method can be defined as a procedure:

```
pro Square::CalculateArea
self->GetProperty, length=length
Self.area = length^2

end
```

or as a function:

```
function Square::CalculateArea
self->GetProperty, length=length
Self.area = length^2

return, 1
end
```

Note: that the procedure or function name is prefixed by the class name followed by a double colon (Square::) to indicated that it is a method of the Square class. It is customary to store all the class structure definition together with all the method definitions into the same file namely: <classname>__define.pro

Lifecycle methods. In IDL two methods are usually defined for each class

1. The **Init** method. The Init method is called during the creation or construction of an object of the class. It is generally known as the class *constructor*. An Init method *must be provided* for every class. Its purpose is to initialize an object so it has a default state that is proper for an object of that class.
2. The **Cleanup** method. This method is called whenever an object of the class is kill or destroyed. It is more generally known as the class *destructor*. A Cleanup method is optional but should be provided whenever data members use heap memory.

Note: the Init and Cleanup methods are called lifecycle methods because they are called only when an object is created or destroyed.

Working with objects in IDL

Creating an object

The **OBJ_NEW()** function in IDL is used to instantiate an object in IDL. To create a circle object, the following syntax is used:

myCircleObject = OBJ_NEW('Circle')

where the argument is a string specifying the class name of the object to be created. Once created, the object myCircleObject is just like any other IDL variable in that it has a state or value. It is distinct from any other object including other circle objects that may be created.

Note: The OBJ_NEW() function automatically calls the Init method of the class

Destroying an object

The OBJ_DESTROY procedure in IDL is used to delete or destroy an existing object. The following command will destroy the circle object created previously

OBJ_DESTROY, myCircleObject

Like pointers, *objects are heap variables* so it is necessary to explicitly destroy them when they are no longer needed in an application. Failure to do so will result in memory leaks.

Note: The OBJ_DESTROY procedure automatically calls the Cleanup method, if it exists.

Interacting with objects

Interaction with objects occur through its public interface. In IDL, **all methods are public** while **all data members are private**. Therefore, only methods of a class have direct access to class data while all methods can be called from anywhere. The syntax of calling a method in IDL depends on whether it is implemented as a function or procedure.

For procedures:	*myCircleObject->SetProperty, width=10.0*
For functions:	result = *myCircleObject->SetProperty(width=10)*

(Within a method, direct access to private data is possible. Use the **Self** statement to access the current objects data. Data members are referenced as **Self.<data member>** (eg Self.length) and methods as **Self-><method name>** (eg Self->SetProperty).

The complete implementation of Square (square__define.pro) and Rectangle (rectangle__define.pro) are available.

More Complex Classes

Usually, it is possible to define classes using simple built-in data types for the data members. However, in many situations, the objects have a complex nature and there are ways of dealing with this complexity.

Composition

For example, a car object has an engine, seats, doors, etc. But the engine, door and seat are also objects! Therefore, an object may contain other objects. In OOP, the term **aggregation** or **composition** is used to describe the process of creating new objects from other objects. In this case, the **data members** of the composite or aggregate class will have one or more **objects of another class**. The relationship between the parent object and the included object is known as a "**has a**" relationship. In our example the car "has a" door. If the relationship between two objects cannot be expressed in this form then composition is not appropriate!

Inheritance

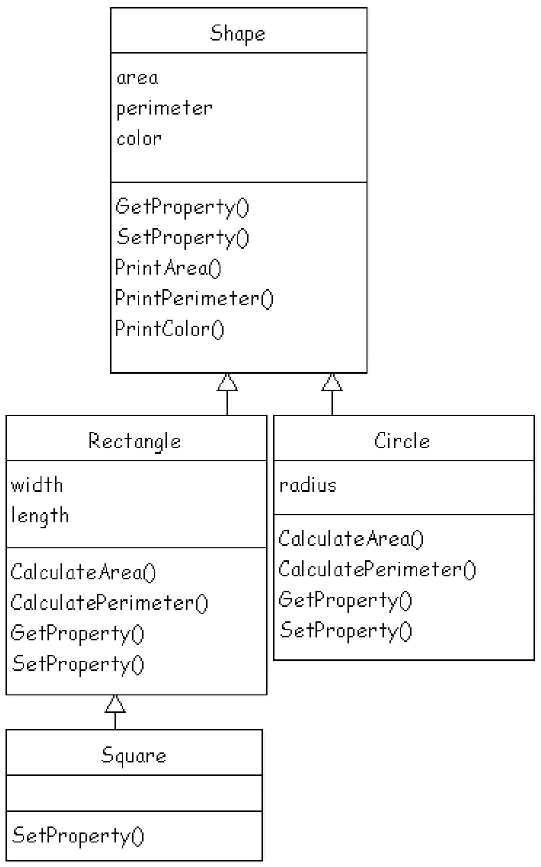
There is another relationship between objects which is slightly more subtle. An object may be almost like some other object except that it may have additional properties or behaviour which make the two slightly distinct. For example a savings account is a bank account which is used mainly for depositing money for a longer term purpose – in addition, there are other types of bank accounts such as a checking account. This is clearly not a composition relationship (between bank account and savings account). A bank account class and a savings account class can certainly be defined but it is clear that there is a high degree of overlap between the two – a savings account is a special kind of bank account. The concept of **inheritance** is used to deal with this type of relationship. The savings account inherits all the properties of a bank account but imposes certain restrictions on it. The parent class (back account) is commonly referred to as the **superclass** or **base class** while the child class is called the **subclass** or d**erived class**. The relationship between the subclass and superclass is an "**is a**" relationship. In our example, the savings account "is a" bank account. If the relationship between two objects cannot be expressed in this form then inheritance is not appropriate!

Inheritance is is generally considered the second of the three fundamental principles of OOP. Inheritance forms the basis of code reuse since it enables the extension of existing classes to add new capability without affecting the old functionality.

Note: Although object hierarchies linked by inheritance are frequently identified during the initial design of an application, inheritance is more commonly used afterwards to extend applications or re-factor (optimize) the code.

Example inheritance with geometric objects

In a previous example, a circle, square and rectangle were objects identified in an application involving geometric shapes. The details of the square and rectangle classes reveal a significant level of overlap which usually suggests the need for some refactoring. One way to deal with this is to introduce a shape class which will acts as a parent to all geometric objects. Thus all geometric objects are shapes. In addition, a square is a rectangle with equal sides – hence implying another inheritance relationship. The following class inheritance hierarchy would therefore be suitable for modeling a system of geometric shapes which include a square, rectangle and circle.



In the class diagram, the arrows represent inheritance with the arrow pointing from the derived class/subclass towards the base class/superclass. Rectangle and Circle classes are derived from the

Shape class. Square derives from a Rectangle and hence Shape. So a Square is also a Shape. However, a Square **is not** a circle!

Rectangle, Square and Circle all inherit the area, perimeter and color attributes. However, only Circle has a radius.

Implementing Inheritance in IDL

The major difference from before is in the declaration of the class structure. Assuming the Shape class exists, then the Rectangle class structure should be defined as follows:

```
pro Rectangle__define

struct = {Rectangle $
          ,inherits Shape $ ; inherit all variables from Shape class structure
          ,width:0.0 $
          ,length:0.0 $
         }

end
```

where the **inherits** statement is used to indicate that the Shape class is being inherited. IDL allows multiple inheritance – in this case each parent class should be referenced by a separate inherits statement.

The derived class should either

- override or replace methods defined in the base class or
- extend the base class methods
- leave them unaltered.

Each method of the same name that is defined in the derived class will automatically supersede that of the base class

To extend (rather than override) a method, it must also be defined in the derived class but it must directly reference or call the base class version.

For example:

- Square extends the SetProperty method.
- Circle adds the CalculateArea and CalculatePerimeter methods and extends the GetProperty and SetProperty methods.

The complete implementation of Shape (shape__define.pro), Circle (circle__define.pro), Square (square__define.pro) and Rectangle (rectangle__define.pro) are available.

Chapter 2: Overview of the iTool Framework

What is it?

The **iTool framework** is a set of classes with well defined application programing interface (API) that can be used to write applications with a rich set of built-in functionality. The framework makes extensive use of object oriented programing and consists of dozens of classes, many of which are written in IDL.

There is a subtle distinction between the iTool framework and the **Intelligent Tools** (or **iTools** for short). iTools are pre-built IDL applications that are based on the iTool framework. Essentially, these are flagship applications that are designed to showcase the capabilities of the framework especially with regards to data visualization. The iTool Framework and iTools are often used interchangeably due to the small difference between the two. In this tutorial, every effort will be made to use the iTool framework when referring to the core classes and APIs and iTools when referring to the example applications. However, there may be instances in which they are incorrectly used.

As of IDL 7.0, there are a total of seven iTools which are designed around a specific data or visualization type. These are

1. iPlot – used for 1D and 2D plots (line, scatter, polar and histogram style)
2. iContour – used for 2D contour displays
3. iImage – used for 2D image displays
4. iVector – used for visualizing vector (magnitude and direction) data
5. iSurface – for surface visualizations
6. iVolume – for volume visualizations
7. iMap – for displaying map information and data

In addition to the visualization functionality, these iTools also include many useful image analysis capabilities.

The classes that form part of the iTool framework significantly enhance the IDL language and provide standard features that can be integrated into any application. These features include:

- creation of visualization graphics
- mouse manipulations of visualization graphics
- annotations
- management of visualization and application properties
- undo and redo capabilities
- data import and export
- printing
- interface element event handling

Most of these features are provided as fully implemented classes while others are in the form of well defined APIs that requires the developer to implement prescribed classes or methods. In either case, the net result is that the features are incorporated into applications as objects or components. The framework is designed to be very modular and as such is also referred to as the **iTool Component Framework**.

Key Features of the iTool Framework

As mentioned earlier, the framework consists of several dozen core classes that form part of an overall design or architecture which introduces several new concepts to IDL application developers. The most important of these are:

- A platform-independent text-based object identifier system for labeling or identifying objects. There is no need to keep track of all object references that are created since they can be subsequently located using their object identifiers in a manner similar to the way in which IDL widgets can be found using their uvalue property.
- A messaging system that allows notification messages to be sent and received between components. This way, one component can observe another and react accordingly.
- A separation between the non-interactive functionality and the presentation layer. The non-interactive functionality here refers to that which executes without direct user intervention. The presentation layer encompass user-interface elements defined using the widget toolkit. This separation promises the possibility that future changes to the presentation layer will have minimal overall impact to existing applications.
- A registration mechanism that allows whole components to be added or removed from an application – only registered components are accessible within an application. The registration scheme is also used to selectively control which properties of an object are to be exposed.

The above are the most significant ideas even though there are many additional examples of how the iTool framework has introduced a fresh approach to application development in IDL. To achieve this level of sophistication in new functionality, the iTool framework contains a vast amount of new classes that were added as standard to the IDL development package. The diagram in Figure 1 shows the core classes that define the base functionality of the framework.

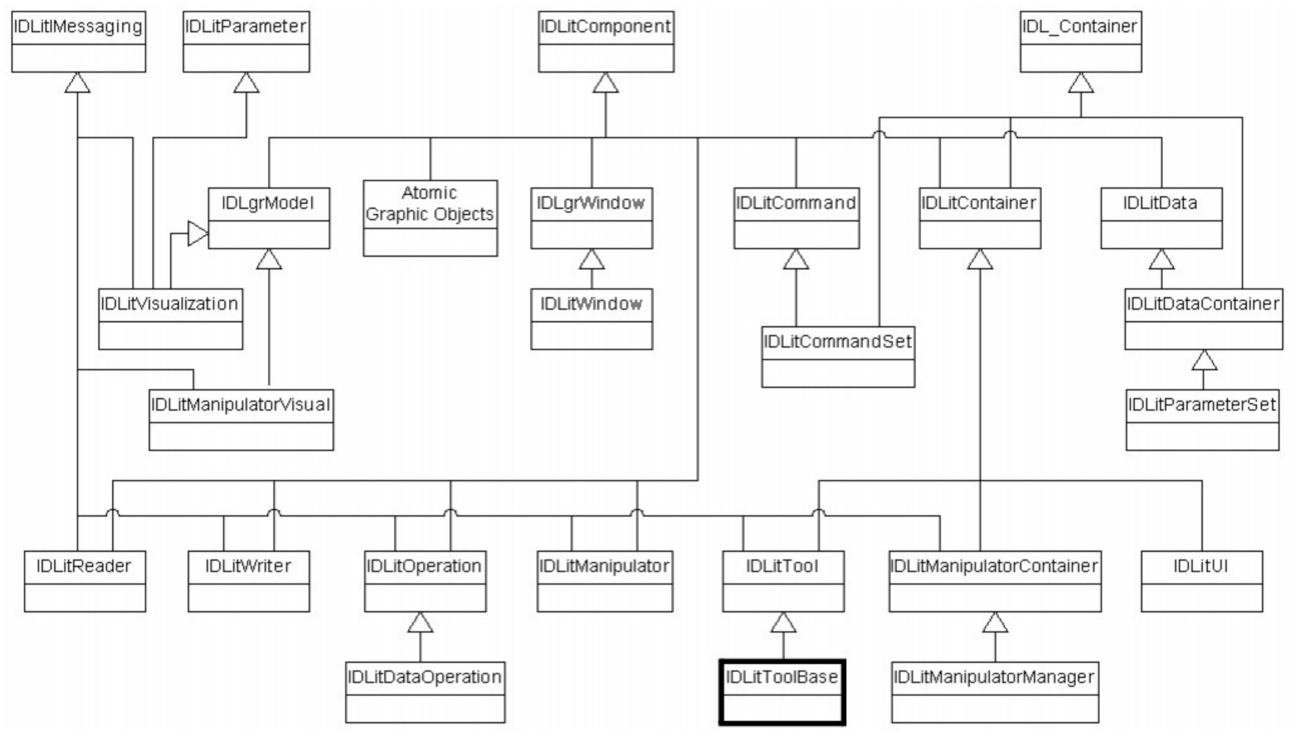


Figure 1: Visual overview of the inheritance relationship of the core classes in the iTool framework. Note that virtually every class inherits from IDLitComponent. (source: iTool Developer's Guide)

Note: Many of the classes are related through inheritance (arrow points from base class to superclass). There are only a few (four) root classes that have no superclass and multiple inheritance is also widely used.

The class diagram shows only the core classes – there are many more undocumented classes that form an integral part of the framework. Some of the class names indicated in the diagram (IDLitTool, IDLitVisualization, IDLitData, IDLitReader, IDLitWriter, IDLitOperation, IDLitManipulator, IDLitManipulator, etc) are **superclasses representing a subset or category** of classes with similar functionality. For example, there are approximately 23 other visualization classes that are subclassed from IDLitVisualization – each of these implements a specific type of visualization such as a surface. It is not particularly useful to list all the subclasses within these main categories here but if you are interested, see Chapter 2 of the iTool Developer's Guide. However, it is important to highlight a few main class categories since they provide an indication of the types of components that are provided by the framework:

- **IDLitTool**. A full featured iTool-based application is usually an object of this class or subclass.
- **IDLitVisulization**. Subclasses of this class provide visualization or graphic components
- **IDLitData**. Used for storing data of any type
- **IDLitOperation**. Used for creating components that perform specific actions with undo/redo capability.
- **IDLitReader**. Used for defining input filters for reading data from specific file types.
- **IDLitWriter**. Used for definig output filters for writing data to specific file types.
- **IDLitManipulator**. For defining keyboard and/or mouse interactions with a visualization.

The underlined class types will be discussed in more detail in this manual.

Composition of an iTool Application

Any non-trivial iTool-based application will typically contain a few essential components as suggested in Figure 2. A system object (IDLitSystem), one or more tool objects (IDLitTool) and one or more UI objects (IDLitUI) are part of any iTool based application as shown in the diagram. The tool object deals with the non-interactive functionality while the UI objects handles the user interface (widgets). Also, depending on the nature of the application, there will be additional components such as operations, visualizations, user interfaces services, etc.

In the diagram, components indicated to the left of the dash line in blue are non-interactive while those to the right of the line in green are widgets or user interface elements.

Lets examine the various components in a bit more detail.

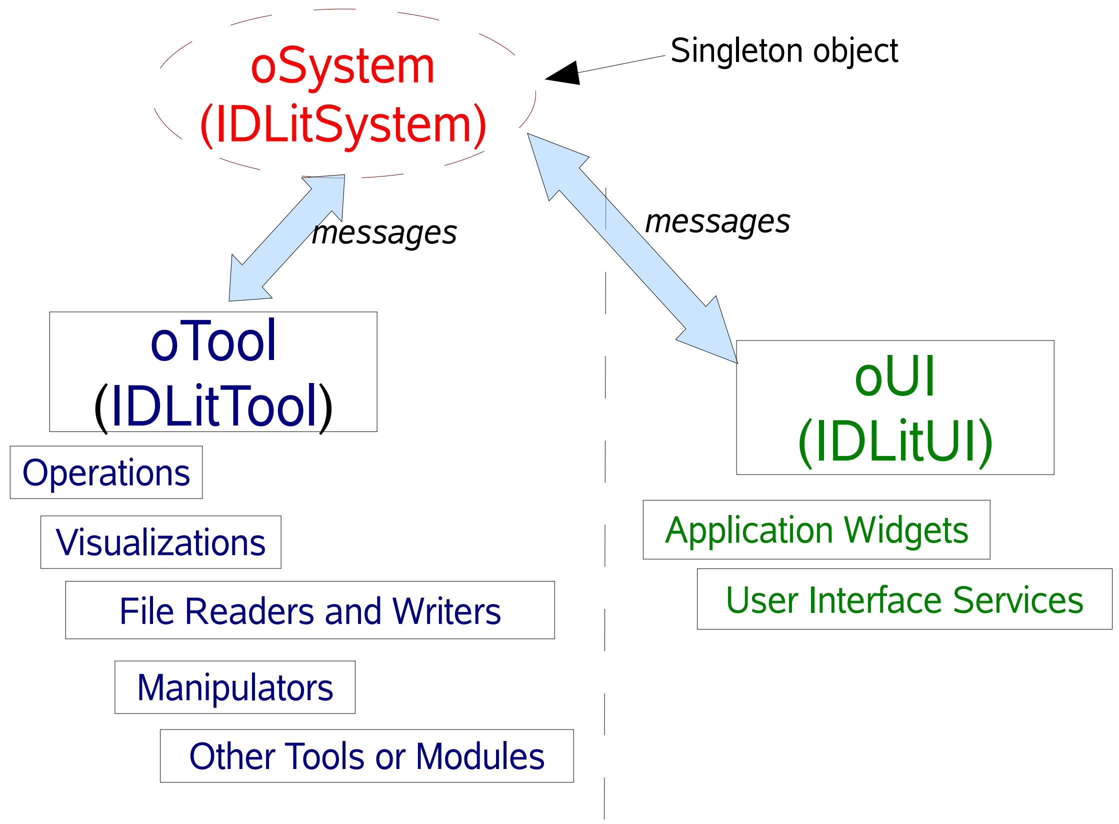


Figure 2: Typical components of an iTool based application. The vertical line indicates that there is a loose coupling between the tool and UI objects – the system object coordinates notification messages between the two.

The system object.

Every application has one system object – indeed, only one system object can be created during an IDL session regardless of whether there are multiple applications running. The system object is therefore a **singleton.**

Note: In OOP a singleton class such as IDLitSystem is designed such that only precisely **one object** of the class can exist at any one time.

The system object is created automatically as part of the registration or instantiation of certain critical framework elements. Earlier, a registration mechanism and messaging system were mentioned as some of the key features of the iTool framework. The system object deals with component registration and also manages the messaging system. It maintains a hierarchy of containers or folders, similar to the directory hierarchy in a UNIX file system, for storing and retrieving these components. Example folders include '/TOOLS','/REGISTRY','/DATA MANAGER' for storing tool objects, registered components and data respectively.

Object Descriptors

Special lightweight objects known as **object descriptors** are used to represent registered components. The object descriptor knows how to create an instance of the registered component *when it is needed*. Object descriptors play an important role in component registration because without them the potentially heavyweight component objects will be created and stored at registration time even though they may not be used during the life time of the application. Object descriptors therefore significantly reduce memory overhead in an iTool application. Fortunately, the developer rarely has to explicitly manipulate object descriptors as they are handled seamlessly within the system or tool class. Object descriptors representing registered components are all stored underneath the '/REGISTRY' folder. Sub folders are created based on the component type being registered. For example, '/REGISTRY/VISUALIZATIONS' contains registered visualization while '/REGISTRY/OPERATIONS' will contain registered operations.

Object Identifiers

All iTool objects have an associated identifier string (ie like a filename in a file system) and this, in conjunction with the folder name, is used to uniquely identifier any object or component that is registered with the system object. Object Identifiers strings are similar to path specifications in a UNIX file system and are either fully qualified (ie starting from the root hierarchy '/') or relative to some other folder. For example a tool named 'MyApplication' will probably have the fully qualified identifier '/Tools/MyApplication'

Non-interactive Components

The core of an iTool application is a tool object which is an instance of either the IDLitTool class or its subclass. The class should develop the infrastructure necessary to accomplish the goals of the application. In addition, any components that may be potentially useful have to be registered by the tool. By default, the system object registers many components that come standard with the framework. Therefore, if some of these standard components are not needed, they should be *unregistered.* These components that must be registered before use are essentially classes that are written to accomplish a specific functionality. The bulk of the functionality in an iTool application will actually be performed by these components. By default therefore, iTool applications are very modular in nature.

User Interface Components

User interface components are essentially conventional widget programs that are necessary to implement the graphical user interface elements of an iTool application. Categories of user interface components include:

- Main Widget Interface. These are used to create the main canvas for an application and woud normally include menus, toolbars, other standard IDL widgets such as buttons and text widgets and perhaps even a graphics window.
- User Interface Panel (UI Panel). These are dialogs that are meant to dynamically extend the main application window by attaching themselves to it. For example the standard iTools (iPlot, iContour, iImage, etc) all share the same Main Widget Interface. However, for an iImage, the main window is extended using a UI Panel that adds image processing controls.

- User Interface Service (UI Service). A UI Service is typically a floating dialog that can be used to retrieve/display specific information from/to the user. A UI Service is typically implemented as a modal dialog.

Since widget routines are conventional procedures and functions (not classes), after registration, they are controlled entirely through a user interface object (instance of IDLitUI) which provides a proxy object for the widget routines.

Chapter 3: Programing in DAVE

Data Handling

One important requirement for the DAVE redesign was to present a data-centric interface to the user. To achieve this goal, it is essential that there is a solid framework for the management and presentation of data. Luckily, the iTool framework comes with several build-in classes for handling and dealing with various types of data. The diagram below shows the class names, their relationship to each other and links with other parts of the framework.

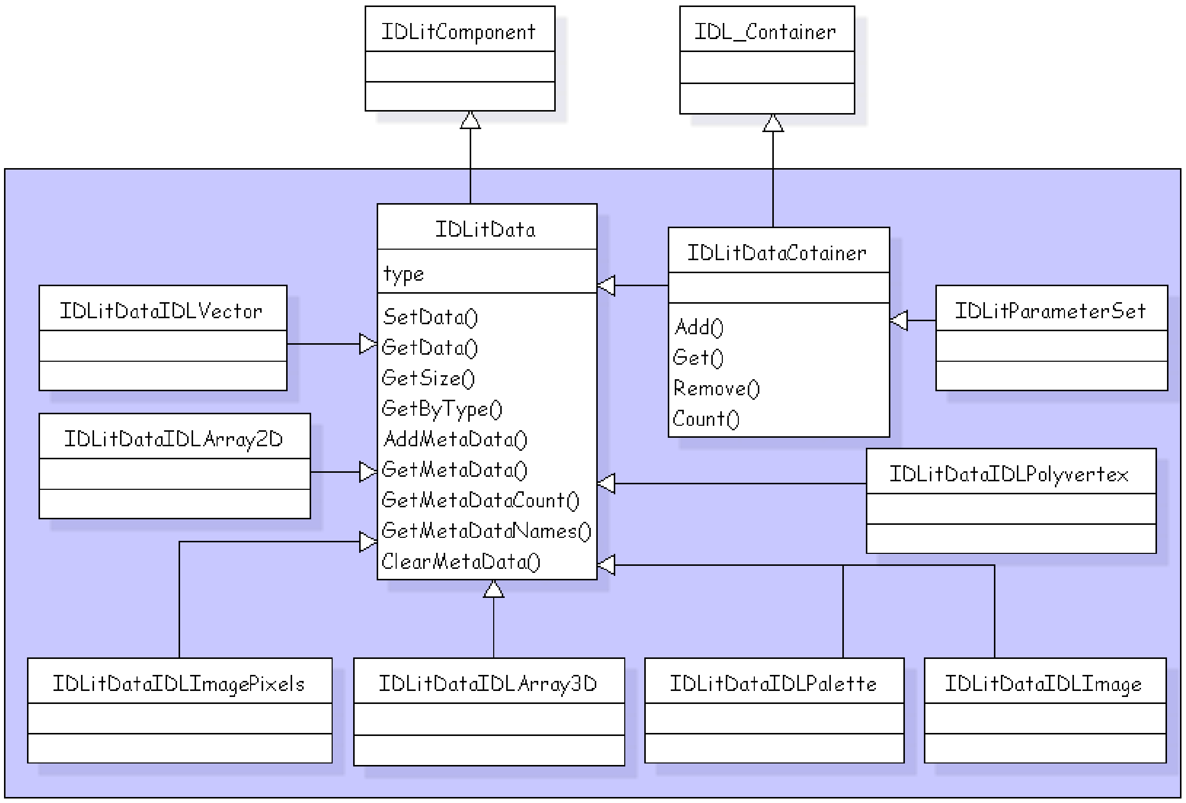


Figure 3: Figure 3: Class diagram of the built-in data classes in the iTool framework. For clarity, only class names are shown except for IDLitData and IDLitDataContainer where some additional (but not all) details are presented.

IDL_Container, a container class, and IDLitComponent are the superclasses for the data classes in the framework. The data classes make it easy to package various types of data and the following table provides a brief description of each. IDLitData is clearly the base data class in the iTool framework from which all the others derive from.

Let us now look at a few examples of how these classes are used to quickly create and use data objects.

Basic data objects

Create an IDLitData object and store a scalar float value in it. (You may find it more beneficial if you type in these examples at the IDL command prompt!)

```
IDL> fltObject = obj_new('IDLitData')
IDL> void = fltObject->SetData(10.0)
IDL> void = fltObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       FLOAT     =       10.0000
```

The SetData and GetData methods are used to set and retrieve the object's stored data.

IDLitData	Stores data of any IDL generic data type
IDLitDataContainer	Implements a container that can hold multiple data items or other containers. Very useful for organize data into hierarchies.
IDLitParameterSet	A specialized container in which the contained data can be associated with a label known as a 'parameter name'
IDLitDataIDLVector	Storage for a 1D array of any IDL generic type.
IDLitDataIDLArray2D	Storage for a 2D array of any IDL generic type
IDLitDataIDLArray3D	Storage for a 3D array of any IDL generic type
IDLitDataIDLPolyVertex	Used to store vertex and connectivity lists suitable for use with IDLgrPolygon and IDLgrPolyline objects.
IDLitDataIDLImage	Used to store two-dimensional image data. Images can be constructed from multiple image planes.
IDLitDataIDLPalette	Stores palette data
IDLitDataIDLImagePixels	Stores raw image data (pixels).

Table 1: Built-in IDL data classes and their intended uses.

The object creation and data assignment could have been achieved in a single command as follows

```
IDL> fltObject = obj_new('IDLitData',10.0)
IDL> void = fltObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       FLOAT     =       10.0000
```

Other types of data objects can be created in a similar fashion to hold a scalar byte, a scalar string, an integer vector and a 2D float array:

```
IDL> byteObject = obj_new('IDLitData',255B)
IDL> void = byteObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       BYTE      =  255
IDL> strObject = obj_new('IDLitData','A string example')
IDL> void = strObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       STRING    = 'A string example'
IDL> integer1DObject = obj_new('IDLitData',[2,4,6,8,10])
IDL> void = integer1DObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       INT       = Array[5]
IDL> flt2DObject = obj_new('IDLitData',[[1.1,1.3,1.5],[2.2,2.4,2.6]])
IDL> void = flt2DObject->GetData(dataValue)
IDL> help,dataValue
DATAVALUE       FLOAT     = Array[3, 2]
IDL> print,dataValue
      1.10000      1.30000      1.50000
      2.20000      2.40000      2.60000
```

As indicated in the above class diagram, IDLitData objects have a 'type' data member or property in addition to 'name' and 'identifier' that are inherited from IDLitComponent. If no type is specified, it defaults to an empty string as was the case for all the data objects created above. (By default the name and identifier property are set to 'Data' for all IDLitData* classes)

```
IDL> byteObject->GetProperty, type=type, name=name, identifier=identifier
IDL> help,type,name,identifier
TYPE            STRING    = ''
NAME            STRING    = 'Data'
IDENTIFIER      STRING    = 'DATA'
```

The type property accepts any scalar string and provides a means of tagging an object with any label that best describes the contents of the object or a label which is suggestive of the intended use of the object's data. The following commands set the type property for the example objects

```
IDL> byteObject->SetProperty, type='Byte'
IDL> strObject->SetProperty, type='Title'
IDL> integer1DObject->SetProperty, type='Even Numbers'
IDL> flt2DObject->SetProperty, type='2D Floats'
IDL>
IDL> flt2DObject->GetProperty, type=type
IDL> help,type
TYPE            STRING    = '2D Floats'
```

As you can see, the type property can be set to any scalar string. Later on, its usefulness will be demonstrated.

With the exception of IDLitDataContainer and IDLitParameterSet, the remaining data classes in the above diagram are simply specializations that are designed to contain specific kinds of data. The type property for objects of these classes default to defined iTool data types shown in the following table.

iTool Data Type	Contents
IDLVECTOR	A vector of any IDL data type
IDLARRAY2D	A two-dimensional array of any IDL data type
IDLARRAY3D	A three-dimensional array of any IDL data type
IDLCONNECTIVITY	A vector containing connectivity list data
IDLIMAGE	A composite data type that includes IDLIMAGEPIXELS and IDLPALETTE data
IDLIMAGEPIXELS	One or more two-dimensional image planes
IDLPALETTE	A 3 x 256-element byte array
IDLPOLYVERTEX	A composite data type that contains a vector of vertex data and a vector of connectivity data
IDLVERTEX	A vector containing vertex data
IDLOPACITY_TABLE	A 256-element byte array

Table 2: iTool data types (source: iTool Developer's Guide)

For example, the following commands create IDLitDataIDLVector object which is expected to store data of type IDLVector.

```
IDL> myVec = obj_new('IDLitDataIDLVector',['a','b','c','d'])
IDL> myVec->getProperty, type=type
IDL> help,type
TYPE            STRING    = 'IDLVECTOR'
```

Unfortunately, the IDLitData* classes do not have any type-checking mechanism to prevent data of the wrong type from being stored. In the following example,

```
IDL> myVec1 = obj_new('IDLitDataIDLVector',1.0)
IDL> myVec1->getProperty, type=type
IDL> void = myVec1->GetData(dataValue)
IDL> help,type,dataValue
TYPE            STRING    = 'IDLVECTOR'
DATAVALUE       FLOAT     =       1.00000
```

I accidentally assigned a scalar to an IDLitDataIDLVector object and hence the object's type and content are not consistent! The bottom line is that the developer should use the type property as a label but should not rely on it to determine the object's content.

Note: Do not rely on the type property to determine the contents of a data object. Use the object's data_type property to determine the ***intrinsic IDL data type*** (specified as an integer) of the object's contents.

Data Hierarchies

IDLitDataContainer inherits from IDLitData and IDL_Container and hence is also a container class. A container object can be used to *store* **other objects**, *retrieve* them when needed, and to *remove* them when no longer needed. Containers are most often used for storing a set of related objects together. Another important use is for constructing hierarchical data structure from a set of data objects.

The following commands create a IDLitDataContainer object. The Count method is used to determine the number of items in the container and the Add method adds an object to the container.

```
IDL> myContainer = obj_new('IDLitDataContainer',name='Example Container', $
                                 type='Miscellaneous')
IDL> print,myContainer->Count()
           0
IDL> myContainer->Add, byteObject
IDL> print,myContainer->Count()
           1
IDL> items = myContainer->Get()
IDL> help,items
ITEMS           OBJREF    = <ObjHeapVar51801(IDLITDATA)>
```

The IDL_Container::Get method is used to retrieve the contents of a container. If used without any arguments as in the example above or with the /ALL keyword, then all the objects in the container will be returned as an object array. If the order in which the items were added to the container is significant, the POSITION keyword can be used with the Get method to specify which item to retrieve.

```
IDL> myContainer->Add, strObject    ; add a second object to container
IDL> print,myContainer->Count()
           2
IDL> items = myContainer->Get(/all) ; retrieve all items in container
IDL> help,items
ITEMS           OBJREF    = Array[2]
IDL> item = myContainer->Get(position=0) ; retrieve first item only
IDL> help,item
ITEM            OBJREF    = <ObjHeapVar51801(IDLITDATA)>
IDL> item = myContainer->Get(position=1) ; retrieve second item only
IDL> help,item
ITEM            OBJREF    = <ObjHeapVar40795(IDLITDATA)>
```

Currently myContainer has 2 items and certainly more can be added as needed to form a flat structure. However, more interesting data hierarchies can be constructed by using sub-containers to create a tree structure as in the following example.

```
IDL> cont1D = obj_new('IDLitDataContainer',type='1DContainer')
IDL> cont1D->Add, integer1DObject
IDL> cont2D = obj_new('IDLitDataContainer',type='2DContainer')
IDL> cont2D->Add, flt2DObject
IDL> myContainer->Add, [cont1D,cont2D]         ; add sub-containers to the main one
IDL> print,myContainer->Count()
           4
IDL> items = myContainer->Get(/all)
IDL> print,items
<ObjHeapVar32(IDLITDATA)><ObjHeapVar63(IDLITDATA)><ObjHeapVar193(IDLITDATACONTAINER)>
<ObjHeapVar227(IDLITDATACONTAINER)>
```

Two new containers are created and an object is placed in each. The two conatiners are then added to myContainer. The Count method now indicates that there are four items in myContainer. Retrieving and printing all the contents of myContainer shows that the four items are the two data objects that were previously added and the two sub-containers that were just added. The data objects added to the two sub-containers are not retrieved! That is, the Get method does not descend into sub-containers. To obtain the items in the sub-container, the Get method must be executed on the sub-container.

Obviously, using the Get method's POSITION keyword to retrieve a specific item from a container gets tricky very quickly if one cannot keep track of the order in which the items were added to the container. Earlier, the IDLitData's type property was introduced. It turns out that it is possible to use it to retrieve a specific object from a container by using the IDLitData::GetByType method. In the last code snippet, the first sub-container was given the type '1DContainer' and this can be used to retrieve the object from the myContainer object:

```
IDL> item = myContainer->GetByType('1DContainer')    ; retrieve object whose type is
'1DContainer'
IDL> help,item
ITEM            OBJREF    = <ObjHeapVar193(IDLITDATACONTAINER)>
IDL> print,item->Count()
           1
```

It is therefore useful to always assign a meaningful type property to IDLitData* objects since that information can be later used to search for objects contained in a hierarchy.

Parameterset Containers

Parameter sets are encapsulated in the IDLitParameterSet class which is a specialization of IDLitDataContainer. The main difference is that it is possible to associate contained objects with a 'parameter name' where parameter name can be any scaler string. However, unlike the type property, the parameter name is not a property of the object being added but rather serves as a way of *tagging* an object *within* a particular parameterset container – the object itself is not aware of the tag!

IDLitParameterSet objects are commonly used when packaging 'plottable' data for visualization. In this case, the parameter names are used for specifying how the data objects in the container are to be used in a visualization. The developer is responsible for coming up with a consistent scheme within their application that enables data to be more easily associated with *visualization parameters*.

The following code snippet illustrates the use of an IDLitParameterSet object.

```
IDL> ; Create x,y,error data for a line plot
IDL> xdata = (findgen(11) - 5)*0.1*!pi
IDL> ydata = sin(xdata)
IDL> error = 0.1*(abs(ydata) > 0.1)
IDL> ; Encapsulate data into IDLitDataIDLVector objects
IDL> xObj = obj_new('IDLitDataIDLVector',xdata)
IDL> yObj = obj_new('IDLitDataIDLVector',ydata)
IDL> eObj = obj_new('IDLitDataIDLVector',error)
IDL> ; Create an IDLitParameterSet object to hold dataset
IDL> pSetObj = obj_new('IDLitParameterSet',name='ParameterSet Object', $
                              type='1DPlottableData')
IDL> ; Add data to parameterset object
IDL> pSetObj->Add, xObj, parameter_name='Independent' ; can add a single object at a
time
IDL> pSetObj->Add, [yObj,eObj], parameter_name=['Dependent','Error']  ; or object array
```

In this example, three pieces of data have been grouped together into a dataset in a paramterset object. The parameter names associated with the data objects means that a visualization module will be able to automatically determine how the dataset should be used in a line plot. Of course, the visualization module would have to know that an object with the parameter name 'Independent' should be used as the independent variable in a plot and that the 'Dependent' and 'Error' objects should also be used as the dependent and error data respectively. Within the visualization module, given the parameterset object, the dataset can be easily resolved as follows:

```
IDL> ; Assuming a valid parameterset object pSetObj
IDL> ; Retrieve the independent data object
IDL> oX = pSetObj->GetByName('Independent')
IDL> void = oX->GetData(xdata)     ; Get the independent data
IDL> ; Retrieve the dependent object
IDL> oY = pSetObj->GetByName('Dependent')
IDL> void = oY->GetData(ydata)     ; get the dependent data
IDL> plot,xdata,ydata
```

The above code sample would typically be located within a visualization module even though the packaging of the parameterset object happens in another module. The only requirements is that both modules agree on a set of parameter names with consistent meaning.

Metadata

Metadata is data about data! Attributes associated with data are prime examples of metadata. For example, it is important to know the units of most types of data. Furthermore, the units may change during data interpretation in an analysis program. It is useful therefore to have an easy means of associating metadata and data in a transparent way.

Fortunately, the IDLitData class provides full metadata support which allows an unlimited number of attributes to be defined for a data object. In the IDLitData implementation, the metadata is defined as keyword/value pairs where the keyword is any valid scalar string and the value is any valid IDL data type!

Note: For some unknown reason, the metadata support in IDLitData is undocumented! However, ITT have given assurances that this functionality will not be removed in future versions of IDL.

The IDLitData::AddMetaData method is used for setting a new metadata. For example, in the previous parameterset example, let us assign a few attributes to the independent variable object:

```
IDL> xObj->AddMetaData, 'Units', 'Radians'
IDL> xObj->AddMetaData, 'Title', 'X Axis Title'
IDL> xObj->AddMetaData, 'Periodicity', !pi     ;value can be any valid data type!
IDL> print,xObj->GetMetaDataCount()
            3
IDL> print,xObj->GetMetaDataNames()
UNITS TITLE PERIODICITY
```

In the above example, 'Units' is a the metadata keyword or name and 'Radians' is the metadata value. In the first two cases, the metadata value is a string while in the third it is a float. Remember, the metadata value can be any valid data type while the matadata name must be a scalar string.

The GetMetaDataCount method returns the number of metadata currently define for an object.

The GetMetaDataNames method returns the names of all defined metadata.

Use the GetMetaData method to retrieve a metadata value and the AddMetaData method again to redefine an existing metadata:

```
IDL> void = xObj->GetMetaData('PERIODICITY', metaValue)
IDL> print, metaValue
      3.14159
IDL> xObj->AddMetaData, 'Periodicity', 2*!pi   ;redefine periodicity
IDL> void = xObj->GetMetaData('PERIODICITY', metaValue)
IDL> print, metaValue
   6.28319
```

To delete metadata, use the ClearMetaData method. Unfortunately, there is no option to delete selectively!

```
IDL> xObj->ClearMetaData           ; Delete all metadata
IDL> print,xObj->GetMetaDataCount()
      0
```

Examining contents of data in DAVE

So far in this section we have explore the available data classes in the iTool framework and shown how data can be organized into hierarchies using containers or datasets using parametersets. Later, we will see how this can be put together to store the contents of a file. The dataset created will depend on the file format being read. File formats such as the DAVE data format and other ASCII formats like the SPE format are currently supported in DAVE.

The last thing remaining in this section is to show how data structures can be easily loaded into DAVE and examined.

DAVE has a Data Manager module which accepts any IDLitData* object as input. The Data Manager has an associated Data Browser which represents the data object using a tree widget hierarchy and enables the user to perform various actions on its contents.

Without going into too much detail, it is possible to programatically load both data hierarchies that have been created so far.

First, make sure the DAVE project has been created in the IDL development environment (IDLDE). Then compile and run the project. The DAVE application should lunch without any errors.

At the IDLDE command line, execute the following:

```
IDL> oDaveTool = GetDaveTool()

IDL> help,oDaveTool
ODAVETOOL       OBJREF    = <ObjHeapVar18404(DAVETOOL)>
```

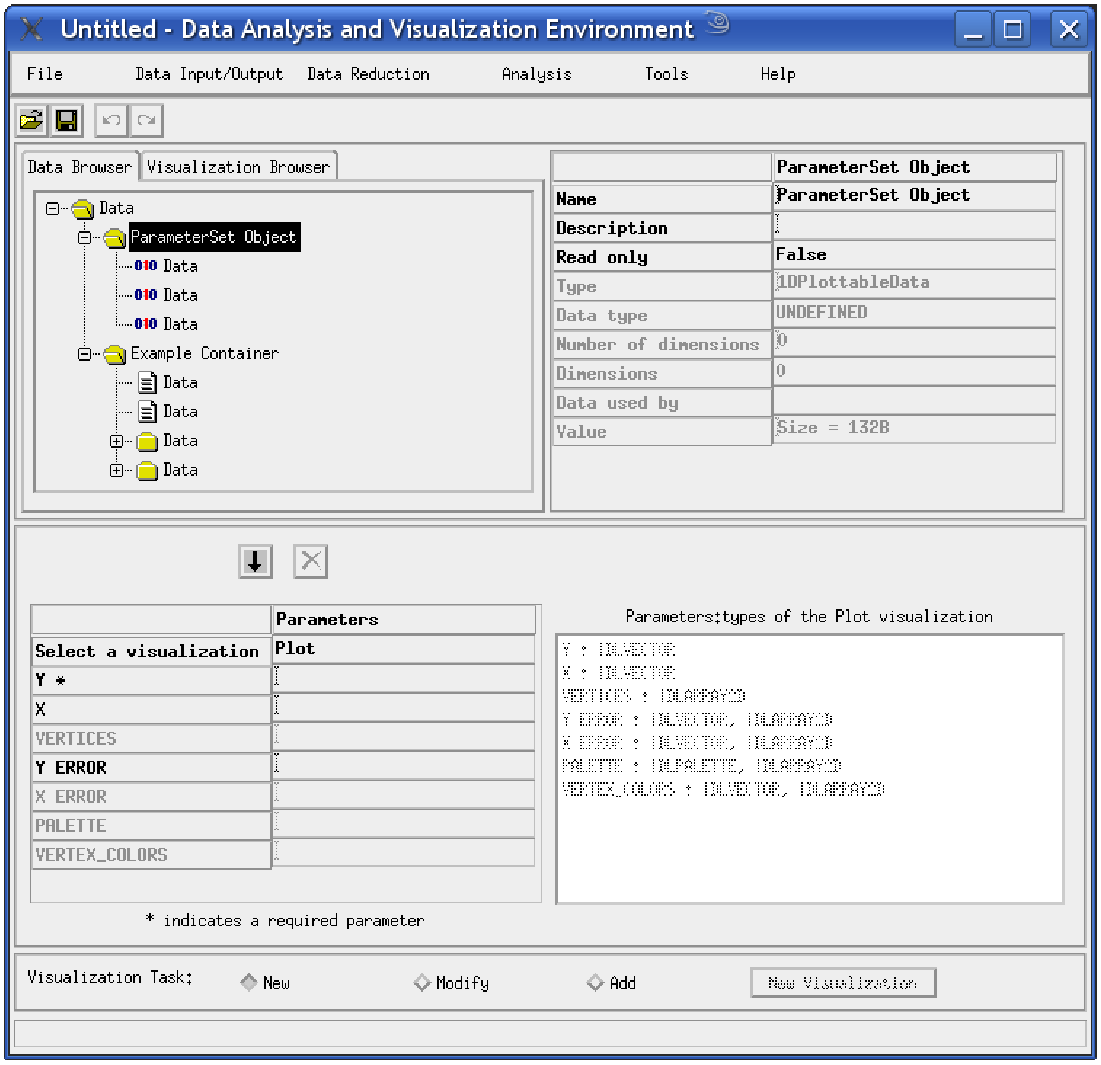
to retrieve an object reference to the DAVE tool that is created as part of the application.

If you have been executing the code snippets presented so far, then you should have created two object hierarchies pSetObj (an IDLitParameterSet object) and myContainer (an IDLitDataContainer object). Both of these can be loaded into the Data Manager in DAVE using the following commands

```
IDL> oDaveTool->AddByIdentifier, 'DAVE Data Manager', pSetObj
IDL> oDaveTool->AddByIdentifier, 'DAVE Data Manager', myContainer
```

The IDLitContainer::AddByIdentifier (DAVETool inherits from IDLitContainer, a generic container class) method adds an object to the tool at a location/folder specified by the given identifier string. The identifier string, 'DAVE Data Manager', is the identifier for the Data Manager folder.

As seen in the sceenshot, the loaded data appear in the Data Browser in the order in which they were added. A few observations should be made:



	ParameterSet Object
Name	ParameterSet Object
Description	
Read only	False
Type	1DPlottableData
Data type	UNDEFINED
Number of dimensions	0
Dimensions	0
Data used by	
Value	Size = 132B

	Parameters
Select a visualization	Plot
Y *	
X	
VERTICES	
Y ERROR	
X ERROR	
PALETTE	
VERTEX_COLORS	

* indicates a required parameter

Parameters:types of the Plot visualization

```
Y : IDLVECTOR
X : IDLVECTOR
VERTICES : IDLARRAY2D
Y ERROR : IDLVECTOR, IDLARRAY2D
X ERROR : IDLVECTOR, IDLARRAY2D
PALETTE : IDLPALETTE, IDLARRAY2D
VERTEX_COLORS : IDLVECTOR, IDLARRAY2D
```

Figure 4: Displaying dataset hierarchies in the Data Browser

- Each container is treated as a dataset and appears as a branch in the tree widget representation of the browser.
- Expanding on each of the branches, reveals the contents of each dataset. Containers are represented by branch nodes and are expandable while other data objects are represented by leaf nodes.
- The names of items in the Browser reflect the name property of the objects. Since the *name* property was not set for most of the objects, it defaulted to the value 'Data' – it is advisable to always specify sensible names to data objects you create.
- The embedded property sheet widget to the right of the Data Browser, displays the details of any item that is currently selected in the browser. Note the *type* property for each of the items.

Note that the contents of any leaf node can be viewed by right-clicking on the item and selecting the **View Details** option from the context menu that is displayed. If the data object has associated metadata, that too will be shown.

Brief Outline of DAVE Application

DAVE consists of several modules that are spread over numerous class definitions and conventional procedures. For the developer, it can be very difficult to appreciate how the whole thing fits together. For a better understanding of the code structure, it is helpful to consider the application using the following classifications or categories:

1. The application launcher. This obviously launches the application. However, before an iTool application can be launched, all *potentially* useful components and modules that have been developed must be registered with *system object*. If a component is not registered it will not be available for use!

   Note that most components that come standard with the framework will already be registered with the system object and as such are available for use. Thus it may also be necessary to unregister those standard components that are not needed in your application. In summary therefore, the launch routine:

   - Registers components that have been created by the developer.
   - Unregisters standard iTool framework components that are not required
   - Instantiates the tool object (item 2 below) to launch the application.

2. The main application which consists of two separate parts. First, the non-interactive part which, like any other iTool based application, must be a subclass of the IDLitTool class – this is the ***DAVETool*** class. The second is the user interface part of the application and is implemented almost like a standard widget procedure. The widget procedure is called ***wd_DAVETool***. The DAVETool class determines which registered components to incorporate – it essentially performs a component integration role. The class also provides any basic infrastructure required to ensure that the components function well together. wd_DAVETool is responsible for constructing the user interface based on the components that have been incorporated by the DAVETool class. It also performs any required event handling including dealing with notification messages generated by the framework. In summary, the main application consists of:

   - The DAVETool class that is responsible for integrating useful application components
   - A user interface definition (wd_DAVETool) that constructs the main application window and handles widget events and other notification messages.

3. The third category of code in DAVE consists of everything else that does not belong to the first two! Due to the OOP nature of the iTool framework, functionality using modular components which can be later integrated using the main application tool class. In an application the size of DAVE the number of required components are too many and all cannot be listed. Most of these components come standard with the iTool framework while many more have been (or will be) added by DAVE developers. In terms of functionality, these are the major domains that DAVE covers:

   - Project management. To handle saving and resumption of user sessions.
   - Data Manager. To better deal with neutron scattering datasets.
   - Visualizations. Support for various types of data visualization.
   - Support for various data file formats (file readers) – DAVE, ASCII (various), NeXus
   - Support for various data file formats (file writers) – DAVE ASCII (various), NeXus
   - Data reduction support for various instruments at the NCNR and PSI. Initially the data reduction modules will be the same as in the previous version of DAVE (DAVE 1.x).
   - Data analysis functionality. Data operations and curve fitting.

- Numerous miscellaneous tools useful for neutron scattering.
- Planning tools for neutron scattering instruments.

Components covering some of these functionality areas have been developed although there is still a lot left to be accomplished.

In the remainder of this manual, we will examine how some of these components are implemented in order to get a better understanding of DAVE development using the iTool framework.

The Launch Procedure

The launch procedure for DAVE is implemented in the file dave.pro. dave.pro calls on other supporting routines as well.

The main goals of the launch routine are:

- to register certain basic components, including the main application tool.
- to instantiate the application class object.

Component Registration

Registration links a *class or procedure* with an ***object identifier***. The object identifier has a name and is used in essence to *represent* that class or procedure subsequently. A few key points

- Components can be registered with the system object or with an individual tool.
- If registered with the system object, a component will have global scope and can be used by any tool.
- If registered with a tool, the component will be visible only to that tool
- Tools must themselves be registered with the system object. Also, main widget interface code (for creating the main application window) can only be registered with the system object.

Registration is accomplished either by using the ITREGISTER wrapper procedure or various methods of the IDLitSystem/IDLitTool classes. (The ITREGISTER procedure indirectly calls the IDLitSystem methods.)

After registration, components of the same type are grouped together in the same folder. For example all registered tools are stored in the '/Registry/Tools' folder while visualizations are grouped in '/Registry/Visualizations'. For this reason, at registration, you have to indicate the component type (if using the itregister procedure) or use a specific method for that component type.

Example: let us assume that we have implemented a NeXus file reader in a class called nexusfilereaderclass__define.pro. This can be registered *globally* using either of the following:

```
itregister, 'NeXusReader', 'NeXusFileReaderClass', /FILE_READER
oSystem->RegisterFileReader, 'NeXusReader', 'NeXusClassName'
```

where oSystem is the singleton instance of IDLitSystem. Both statements accomplish the same thing. An object descriptor will be created with the name 'NeXusReader' to represent the NeXus file reader class. Subsequently, whenever a NeXus file reader is needed, the 'NeXusReader' component will be used!

Without going into details of registration syntax, let us examine the contents of dave.pro

```
pro dave, identifier=identifier, _EXTRA=extra
compile_opt idl2

; version info: == 2.0
DAVE_MAJOR_VERSION=2
DAVE_MINOR_VERSION=0

; Where are we executing from (=install directory, hopefully)
defsysv,'!DAVE_DIR', sourcepath()


; DAVE 1.x compatibility setup
;colorset
DAVE1Setup

; DAVE 2 System setup - register/unregister various components
if (~DAVESetup()) then return

;register the Main Tool
itregister, "DAVEMain", "DAVETool"

;register the main user interface widget creation routine
itregister, "DAVEMain_UI", "wd_davetool", /USER_INTERFACE

;Okay, launch the Main Tool
title = "Data Analysis and Visualization Environment"
identifier = IDLitSys_CreateTool("DAVEMain", name="DAVE Main Tool" $
                                  ,title=title $
                                  ,USER_INTERFACE="DAVEMain_UI" $
                                  ,dave_major_version=DAVE_MAJOR_VERSION $
                                  ,dave_minor_version=DAVE_MINOR_VERSION $
                                 )

; Set system preferences
oSystem = _IDLitSys_GetSystem()
oDAVETool = oSystem->GetByIdentifier(identifier)
if (obj_valid(oDAVETool)) then  oDAVETool->InitPreferences


end
```

The routine is very concise and accomplishes the following:

- Initially a call is made to DAVE1Setup which defines various symbols and attributes that are necessary for backward compatibility with modules ported from older versions of DAVE.
- Then DAVESetup is called to perform a bunch of component registrations (see below).
- The main tool defined in class DAVETool is then registered and given the component name 'DAVEMain'.
- Also, a widget interface routine which is implemented in the file 'wd_davetool.pro' is registered using the component name 'DAVEMain_UI'.
- The DAVE tool or core application is then launched using the IDLitSYS_CreateTool() function. The argument indicates the tool to be created (specified using the registered component name) The USER_INTERFACE keyword is very important since it specifies the widget component to be used to construct the application's main window.
- Finally, when the application is up and running, an object references to the tool is obtained and some application specific initialization is performed.

DAVESetup essentially registers several components that have been created for DAVE. Note that components registered here are global in scope because they are registered here with the system object – components with limited scope to a specific tool should be registered in that tool as will be shown later for the DAVETool class.. DAVESetup should be updated to include any global components that are developed in future. The following table highlights the components registered in DAVESetup

Component type	**Component Name** **(object descriptor name)**	**Class / Procedure name**	**Description**
Visualization	Plot	DAVEvisPlot	Creates a 1D line plot
	Plot3D	IDLitVisPlot3D	Creates a 2D line plot
	Image	DAVEvisImage	Creates an image
	Contour	DAVEvisContour	A contour
	Surface	DAVEvisSurface	A surface plot
UI Service	AboutDAVE	ui_aboutdave	Provides dialog displaying DAVE version info
	DavePreferences	IDLitUIPrefs	Preferences dialog
	SelectDirectory	ui_selectDir	Directory chooser for data and work directories
	SetDataAxesLabels	ui_setDataAxesLabels	Dialog requesting axes label info.
	AddVisItem	ui_AddVisItem	Adds a new entry in Visualization Browser tree
	FANS Data Reduction	ui_launchFANSReduction	Launches FANS data reduction module
File Reader	DAVE Project	DAVEreadISV	Load a DAVE project file (*.disv)
	DAVE1x	DAVEreadDAVE1	Load a DAVE 1.x file (*.dave)
	ASCII text	DAVEreadASCII	Load an ASCII file – 3COL, SPE, GRP (.txt,.spe,.grp,.dat)
File Writer	DAVE Project	DAVEwriteISV	Save a DAVE project file (*.disv)
	DAVE1x	DAVEwriteDAVE1	Save a DAVE 1.x file (*.dave)
	ASCIICOL	DAVEwriteASCIICOL	Save an ASCII 3COL file
	ASCIIGRP	DAVEwriteASCIIGRP	Save an ASCII GRP file
	ASCIISPE	DAVEwriteASCIISPE	Save an ASCII SPE file
Tool	DAVE VisTool	DAVEvisToolbase	Visualizations are created in a separate module. This defines the visualization tool module.
User Interface	DAVE VisTool Interface	wd_DAVEVisToolbase	Main widget interface for visualization tool module

The DAVETool Class

The DAVETool class (DAVETool__define.pro) inherits from IDLitTool and implements the standard tool functionality required of an iTool application. For now, it is probably only necessary to focus on what happens in the class constructor (the Init method).

```
function DAVEtool::Init, _REF_EXTRA=etc
compile_opt idl2
```

```
; call superclass
if (~self->IDLitTool::init(_EXTRA=etc $
                           ,help='DAVETOOL_HELP' $ ; help keyword for this class
                          )) then return, 0

; Add a 'DAVEDATA_MANAGER' service
self->AddService, OBJ_NEW("DAVEsrvDatamanager", name="DAVEDATA_MANAGER")

; Create the dave data manager folder
oDMFolder = Obj_New("IDLitDataManagerFolder", identifier='DAVE Data Manager' $
                     ,name='Data', description='DAVE Data Manager Container',tool=self)
self->Add, oDMFolder

; Do any tool customization now
if (~self->ModifyComponents()) then return, 0

; Define required menu containers and menu items for this tool
; by linking in registered components
if (~self->MenuLayout()) then return, 0

; set flag to disable re-registration of DAVE specific readers/writers
self.reqReadWriteReg = 0

; return success
return, 1

end
```

First a Data Manager service is created as well as folder or container for storing datasets. The Data Manager service class provides basic generic functionality (delete, duplicate, view contents) for all datasets loaded in DAVE.

Second, the MenuLayout method is used to define the contents and structure of the application. *All application menus* are defined in this method. Menus are essentially links to components known as **operations**. *Operations are components that modify a selected data or the state of the application in some fashion*. In a typical iTool application, operations are the most common class type that would be used by the developer to implement various functionality.

Just like other components, operations need to be registered before they can be used. The operations that come with the iTool framework are already registered by the system object. Custom operations can be registered globally or with the application's tool object as necessary.

The DAVETool::MenuLayout is implemented in the file davetool__menulayout.pro. If you examine the file, you find that it consists mostly of a operation registration statements using the RegisteOperation method which has the following syntax:

```
oTool->RegisterOperation, '<Component name>', '<Class name>', $
                          identifier='<identifier string>'
```

where :

– '<Component Name>' becomes the given name for the operation

– '<Class name>' is the class that defines the operation

– and the identifier keyword specifies the identifier string for the component. The identifier string may sometime include a relative folder path of where the component should be located in the tool's folder hierarchy.

Note: all registered DAVE operation components are located underneath '/Tools/DAVEMain/Operations'. DAVEMain is the name given to the DAVE tool when it was registered!

From davetool__menulayout.pro, the statement

```
self->RegisterOperation, 'Transpose','DAVEopDataTranspose' $
  ,description='Transpose dataset (swap independent axes)' $
  ,identifier='Analysis/DataTrans/Transpose'
```

registers a new operation called 'Transpose' and stores the object descriptor with full identifier (including path) '/Tools/DAVEMain/Operations/Analysis/DataTrans/Transpose'. Subsequently, whenever a 'Transpose' operation is requested, an instance of the DAVEopDataTranspose class will be created.

Please examine the registereOperations statements in the DAVETool::MenuLayout method and pay particular attention to the folder structures created. Then execute DAVE and look at its menu structure. You should find that the menu structure is defined by the structure created in DAVETool::MenuLayout!

Note: The menu structure of an iTool application is defined by the contents of the '/Tools/<toolname>/Operations/' folder!

The User Interface procedure in DAVE

The main application user interface in DAVE is defined in wd_DAVETool.pro and registered in dave.pro. It is a conventional widget program consisting of widget construction and event handling code. However, there are some new concepts or iTool specific features:

- The procedure declaration must use the following API

  ```
  pro wd_DAVETool, oTool, title=title, user_interface=oUI
  ```

  where

  oTool is an input parameter that will be set to the object reference of the tool whose user interface is being constructed.

  user_interface keyword is an output keyword that contains a IDLitUI object on exit
- An IDLitUI object must be created and passed to the calling program using the user_interface keyword. The UI object 'connects' the widget program with the tool class.
- The TLB must be registered with the IDLitUI object as a widget adapter. The widget adapter must be setup to observe messages from the tool object. Notification messages must be handled by a callback procedure which must be defined (wd_DAVETool_callback).
- iTool specific compound widgets are used to construct menus and toolbars.

Please, examine the file wd_davetool.pro for more details on how a user interface widget procedure is written.

Writing a File Reader for DAVE 1.x format files

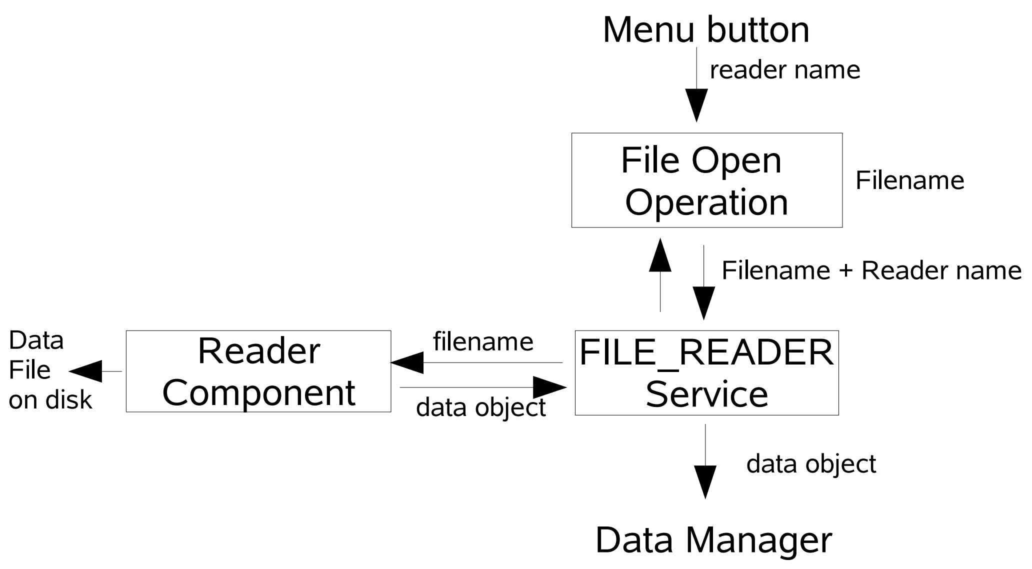


Figure 5: Schematic of the file loading process in an iTool application

A file reader component is used for reading data from a specific data file format. The standard process of initiating and reading a file from disk is illustrated in the above diagram and can be summarized as follows:

- A reader name is associated with the menu button that initiates the file loading action.
- When the menu button is selected a File Open operation is launched. The File Open operation retrieves the FILE_READER Service. This service is used to obtain a list of file filters that are defined for the given file reader. It then presents a file selection dialog with the file filter restrictions for the user to choose a filename to be read.
- If a filename was selected, the FILE_READER Service is again called to retrieve the data from the file. The FILE_READER service then calls on the the appropriate File Reader for that file type which then reads the file and generates a dataset object.
- Finally, the FILE_READER Service places the read dataset object in DAVE's Data Manager container.

The File Open operation and FILE_READER Service are standard components that have already been created for DAVE and do not depend on a particular file type. The File Reader, however, is file format dependent and in general a File Reader should be written for each supported file format. We will now study in detail how to write a File Reader for DAVE 1.x (*.dave) data files.

A custom File Reader class should inherit from the IDLitReader (or its subclass) and adopt the following simple API

– implement an IsA method. The IsA method is used to validate any given file to ensure it is consistent with the reader.

– implement a GetData method. The GetData method reads the data from the file and creates an appropriate data structure.

That is it!

The DAVE 1.x file reader class name is DAVEReadDave1

The DAVEReadDave1 class structure

The class structure is straightforward – we simply inherit from IDLitReader

```
pro DAVEreadDAVE1__Define
compile_opt idl2

void = {DAVEreadDAVE1, inherits IDLitReader}

end
```

The DAVEReadDave1::Init method

The constructor is also quite simple:

```
function DAVEreadDAVE1::Init, _REF_EXTRA=etc
compile_opt idl2

; Init superclass
if (self->IDLitReader::Init('dave'  $             ; <-- acceptable file extension(s)
                            ,NAME='DAVE 1.x' $
                            ,DESCRIPTION="DAVE 1.x data file (dave)" $
                            ,_EXTRA=etc) eq 0) then return, 0

; call setproperty method, if necessary.
if (n_elements(etc) gt 0) then $
  self->SetProperty, _EXTRA=etc

return, 1

end
```

It does not have any required arguments and returns 1 on success or 0 on failure. The superclass Init method is called and the first argument passed, 'dave', is a string representing the common **extension** for files to be read by this reader. Note that the extension is specified without a period. Also, it is a scalar string because DAVE 1.x files only use *.dave extension. However, in cases where multiple extensions are valid, a vector of the allowable extensions should be supplied instead as in ['dave','sav'] meaning *.dave and *.sav files are both valid.

The DAVEReadDave1::IsA method

This method determines whether any particular data file contains DAVE 1.x format data.

```
function DAVEreadDAVE1::IsA, strFilename
compile_opt idl2

; Basic error handler
catch, iErr
if(iErr ne 0)then begin
    catch, /cancel
    return, 0
endif

; Call the base class method to check for proper extension for this reader
if (~self->IDLitReader::IsA(strFilename)) then return, 0

; Check file contents to make sure this is a valid DAVE 1.x file.
; Using the IDL_Savefile class avoids the need to restore the file
oSaveFile = obj_new('IDL_Savefile', strFilename) ; if not a sav file error handler will trigger!
structs = oSaveFile->Names(count=nstruct, /structure_definition)
obj_destroy, oSaveFile
```

```
; A valid DAVE file must contain the named structure 'DAVEPTRSTR'
if (nstruct eq 0) then return, 0
res = where(structs eq 'DAVEPTRSTR', count)
if (count eq 0) then return, 0

return, 1
end
```

The IsA method is a function that must accept one argument: a string representing the filename to be validated. If the file is valid the function returns a 1 and if the file is invalid it returns a 0.

Some basic error handling code is included to deal with unexpected errors – it is a good precaution to take since we would be opening files.

Next, the superclass IsA method is called to check whether the filename has an appropriate extension for this reader. Remember, in the Init method, it was declared that only *.dave files should be acceptable.

However, just because a file ends in .dave does not make it a valid DAVE 1.x file so additional checks are required. If you recall, DAVE 1.x files contain a data structure called the davePtr (DAVE pointer) which is a pointer to a named structure DAVEPTRSTR and s stored using IDL's internal sav file format. So a good test is to check the file to see if it contains such a data structure.

The method used here involves opening the file as an IDL_Savefile object. This avoids the need to actually restore the file's contents.

Note: Restoring an IDL sav file without full knowledge of its contents is dangerous because it will not be possible to delete all restored heap variables if you decide that the restoration was in error.

Use the IDL_Savefile::Names method to retrieve any named structures in the file. If there are no named structures in the file then return a failure (0). If there are named structures, then check to see if any of hem matches DAVEPTRSTR. If there is no match then return failure else return success.

The DAVEReadDave1::GetData method

The GetData method does the work of actually reading the data file and converting the contents into a data object hierarchy that can be used by other iTool components.

```
function DAVEreadDAVE1::GetData, oData
compile_opt idl2

catch, iErr
if(iErr ne 0)then begin
    catch, /cancel
    goto, ioerr  ;; do any cleanup needed
endif
on_ioerror, ioerr

oTool = self->GetTool()
if (not OBJ_VALID(oTool)) then $
  return, 0


filename = self->GetFilename()

; use IDL restore to retrieve info from file
restore,filename, /relaxed_structure_assignment

; No davePtr? Can't continue
if (~ptr_valid(davePtr) || (n_elements(*davePtr) eq 0)) then return, 0
```

```
; Convert the davePtr into an object hierarchy
oData = self->DavePtrToObject(davePtr,filename)
if (~obj_valid(oData)) then begin
    heap_free, davePtr
    return, 0
endif

heap_free, davePtr
help,oData
return, 1


ioerr: ;; IO Error handler
if(n_elements(unit) gt 0)then begin
    sStat = fStat(unit)
    if(sStat.open ne 0)then $
      free_lun, unit
endif
self->SignalError, !error_state.msg, severity=2
return, 0

end
```

GetData is a function that must accept one output argument, oData, that on exit should contain the object reference of a data hierarchy representation of the contents of the file. As usual, a return value of 0 is failure and 1 is success.

Basic error handling code is included for safety.

The filename is retrieved and its contents read using the restore procedure.

A davePtr variable is expected after restoration so if one is not found then exit with failure (contents of a DAVE 1.x format file is discussed below).

Use the helper method DavePtrToObject (discussed below) to convert the davePtr variable to an object hierarchy. If the conversion is successful, then return success otherwise return failure. Note that no matter whether the conversion succeeds or not, the davePtr heap variable is freed.

The object data hierarchy is return to the caller using the output argument, oData.

For further details about how the davePtr is converted to an object hierarchy, it is useful to examine the DavePtrToObject method. First, let us take a brief look at the description of the contents of a davePtr.

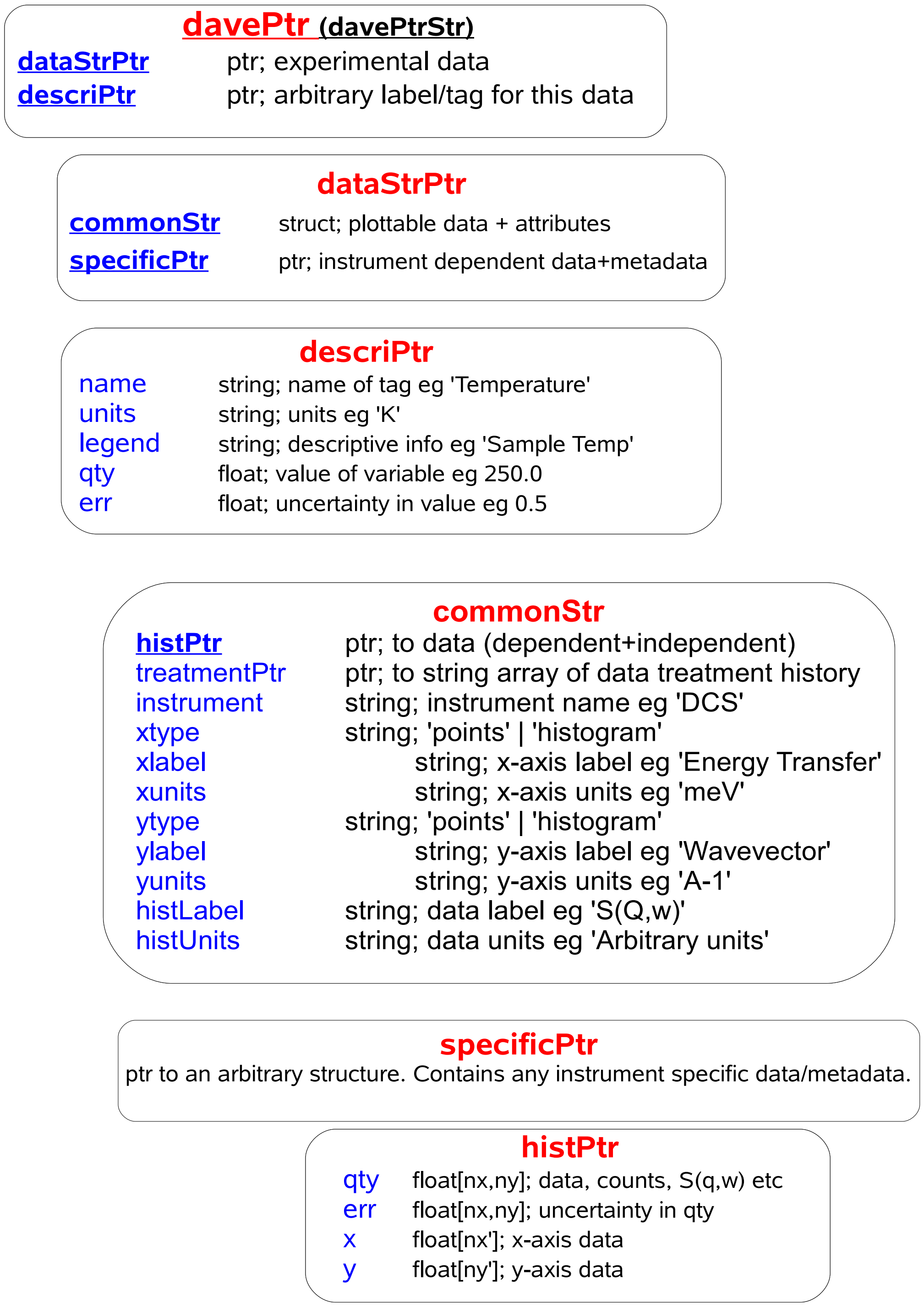

davePtr (davePtrStr)
dataStrPtr ptr; experimental data
descriPtr ptr; arbitrary label/tag for this data
dataStrPtr
commonStr struct; plottable data + attributes
specificPtr ptr; instrument dependent data+metadata
descriPtr
name string; name of tag eg 'Temperature'
units string; units eg 'K'
legend string; descriptive info eg 'Sample Temp'
qty float; value of variable eg 250.0
err float; uncertainty in value eg 0.5
commonStr
histPtr ptr; to data (dependent+independent)
treatmentPtr ptr; to string array of data treatment history
instrument string; instrument name eg 'DCS'
xtype string; 'points' | 'histogram'
xlabel string; x-axis label eg 'Energy Transfer'
xunits string; x-axis units eg 'meV'
ytype string; 'points' | 'histogram'
ylabel string; y-axis label eg 'Wavevector'
yunits string; y-axis units eg 'A-1'
histLabel string; data label eg 'S(Q,w)'
histUnits string; data units eg 'Arbitrary units'
specificPtr
ptr to an arbitrary structure. Contains any instrument specific data/metadata.
histPtr
qty float[nx,ny]; data, counts, S(q,w) etc
err float[nx,ny]; uncertainty in qty
x float[nx']; x-axis data
y float[ny']; y-axis data

Figure 6: Data structure of the davePtr

So essentially a davePtr can be reduced to three important sections:

- the commonStr, the most important, contains the plottable data (histPtr) and additional metadata
- the descriPtr block is used for labeling or tagging the davePtr
- the specificPtr section contains instrument specific data and has no defined structure.

The commonStr and descriPtr sections are dealt with in the DavePtrToObject while the specificPtr is handled by yet another helper method called ReadSpecific.

The DAVEReadDave1::DavePtrToObject method

This method is called from GetData. It is responsible for dealing with the commonStr and descriPtr components of the data structure. Then it calls the ReadSpecific method to address the specificPtr.

This method is fairly long so it is better to review fragments of it at a time.

The method has two input arguments – the first is the davePtr to be converted and the second is the filename the davePtr was restored from.

```
function DAVEreadDAVE1::DavePtrToObject, davePtr, filename
compile_opt idl2

; both the davePtr and filename arguments are required
if (n_params() ne 2) then begin
    errorMsg = 'DAVEreadDAVE1::DavePtrToObject: Missing parameters'
    self->SignalError, errorMsg, severity=2
    return, obj_new()
end

dims = (size((*(*(*davePtr).dataStrPtr).commonStr.histPtr).qty))[0]

if (dims ne 1 && dims ne 2) then begin
    errorMsg = 'Empty data block'
    self->SignalError, errorMsg, severity=2
    return, obj_new()
endif

; strip dir path and extension from the filename
description = filename
filename = file_basename(filename,'.dave',/fold_case)

; iTool does not handle '_[0-9]' characters in filenames well eg _0, _1, etc
; replace all '_' with '-'
while ((pos = stregex(filename,'_[0-9]')) ne -1) do begin
    strput, filename,'-',pos
endwhile
```

The initial part of the function above simply performs some basic checks: the number of parameters must be two and the number of dimensions for the plottable data should be either 1 or 2.

The filename's extension is removed and also stripped of any '_' that is followed by a number (eg '_0', '_8', etc). The filename is used later in the name and identifier of the dataset object and an underscore+number combination has a special meaning with these two properties – they are used to signify a duplicate entry.

The next piece of code:

```
; Create a data container using filename as its name
oData = obj_new('IDLitDataContainer', name=filename $
                ,description=description,identifier='id'+filename $
                ,type='DAVE1DATASET')
```

creates an IDLitDataContainer object to represent the davePtr. Note that the filename is used in the *name* and *identifier* properties of the object. Also, the *type* property is set to 'DAVE1DDATASET' – the type property is very significant and is used to identify DAVE 1.x format datasets subsequently within the application.

Next, two IDLitData objects are created to store two pieces of information – the instrument name and treatment history – and are added to the dataset container as shown:

```
; The instrument name and treatment history
childData = (*(*davePtr).dataStrPtr).commonStr.instrument
```

```
child = obj_new('IDLitData',childData,name='Instrument',type='DAVE1INST'$ ;partype(childData) $
                ,description='Instrument Name')
if (obj_valid(child)) then oData->add, child

childData = (*(*(*davePtr).dataStrPtr).commonStr.treatmentPtr)
child = obj_new('IDLitData',childData,name='Treatment History',type='DAVE1HISTORY' $
                ,description='Treatment history')
if (obj_valid(child)) then oData->add, child
```

The descriPtr section is handled next:

```
;_________________________________________________
; Deal with the descriPtr
void = where(tag_names(*davePtr) eq 'DESCRIPTR', cnt)
if (cnt eq 1) then begin
    ;; A descriPtr entry is present.
    ;; If it is valid, create an entry for it
    if (ptr_valid((*davePtr).descriPtr) && $
        (n_elements(*(*davePtr).descriPtr) gt 0)) then begin

        descriPtrStr = (*(*davePtr).descriPtr)
        ;; Create a container for the descriPtr and fill it with the
        ;; descriPtr contents
        oDescriPtr = obj_new('IDLitParameterSet', name='Sample Information Tag' $
                             ,description='Sample information tag (descriPtr)'  $
                             ,type='DESCRIPTR')

        oQty = obj_new('IDLitData',descriPtrStr.qty,name=descriPtrStr.name,  $
                       type=partype(descriPtrStr.qty),uvalue=0.0)
        oErr = obj_new('IDLitData',descriPtrStr.err,name='Error in '+descriPtrStr.name,$
                       type=partype(descriPtrStr.err))
        oUnits = obj_new('IDLitData',descriPtrStr.units,name='Units', $
                       type=partype(descriPtrStr.units))
        oLegend = obj_new('IDLitData',descriPtrStr.legend,name='Description', $
                       type=partype(descriPtrStr.legend))

        parameter_names = ['Description','Value','Error','Units']
        oDescriPtr->add, [oLegend,oQty,oErr,oUnits],parameter_name=parameter_names
        oData->add, oDescriPtr
    endif
endif
```

The structure of the descriPtr is shown in Figure 6. It is a pointer to a named structure 'DESCRIPTRSTR' containing five fields – name, units, legend, qty, err. An IDLitParameterSet object is created to hold the descriPtr. The quantities: qty, err, units and legend, are each stored in an IDLitData object – the name item is used as the name property of the qty object. The objects are then added to the parameterset object using the parameter_names shown.

The commonStr contains the plottable data – see Figure 6. The next few code snippets show how the conversion is accomplished.

```
;_________________________________________________
; Deal with commonStr

; Create a container for the data (x,y,z axes)
icon = (dims eq 1)? 'plot' : 'surface'
oCommonCon = obj_new('IDLitParameterSet',name='Experimental Data' $
                     ,description='Experimental Data Section',type='DAVE1COMMONSTR',icon=icon)
oCommonCon->AddMetaData,'DatasetName',filename ; give it same name as the filename
oData->Add, oCommonCon
```

An IDLitParameterSet object is created to hold the contents of the commonStr. Note the value of the name, 'Experimental Data', and type, 'DAVE1COMMONSTR', properties. This type will be used to identify the plottable data section of the DAVE 1.x dataset. The commonStr parameterset is then added to the overall dataset container.

The following code deals with the x-axis data or the first independent axis data:

```
; First Independent Axis
; Store in IDLitDataDave object and include additional metadata
desc = (*(*davePtr).dataStrPtr).commonStr.xlabel
xname = (strtrim(desc) eq '')? 'X Axis' : desc
xAxisType = '0'
data = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).x
dist = (strcmp((*(*davePtr).dataStrPtr).commonStr.xtype,'HISTOGRAM',4,/FOLD_CASE) gt 0)? $
       'HISTOGRAM' : 'POINTS'
oX = obj_new('IDLitDataDave',data,name=xname,type=partype(data),description=desc,axisType=xAxisType)
oX->AddMetaData,'Long_name',desc
oX->AddMetaData,'Units',(*(*davePtr).dataStrPtr).commonStr.xunits
oX->AddMetaData,'Distribution',dist

oCommonCon->Add, oX, parameter_name=xAxisType
```

The first few lines extract the x-axis data and additional metadata from the data structure. The xAxisType variable is set to '0' and the distribution is set to either histogram or point mode. As before, an object is created to store the data. Note, however, that we are using custom class called IDLitDataDave which is derived from IDLitData.

Note: The IDLidDataDave class is a subclass of IDLitData and adds an axisType property. The axisType property is used to denote the axis type of a data object. The axisType property is a string that accepts the values '0','1','2','3','4' corresponding to axis types 'X','Y','DATA','ERROR','UNDEFINED' respectively.

Next, 3 metadata are added to the object. The 'Long_name' item contains the axis title, 'Units' specifies the data units and 'Distribution' indicates whether this axis is in histogram or point mode. Finally, the object is added to the commonStr parameterset using the xAxisType as the parameter_name.

If the plottable data is 2D, then the second independent axis or y-axis data plus metadata are retrieved and also stored in an IDLitDataDave object as shown below. Note that the axisType property is set to '1':

```
if (dims eq 2) then begin
    ;; Second Independent Axis
    ;; Store in IDLitDataDave object and include additional metadata
    desc = (*(*davePtr).dataStrPtr).commonStr.ylabel
    yname = (strtrim(desc) eq '')? 'Y Axis' : desc
    yAxisType = '1'
    data = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).y
    dist = (strcmp((*(*davePtr).dataStrPtr).commonStr.ytype,'HISTOGRAM',4,/FOLD_CASE) gt 0)? $
           'HISTOGRAM' : 'POINTS'
    oY = obj_new('IDLitDataDave',data,name=yname,type=partype(data),description=desc,axisType=yAxisType)
    oY->AddMetaData,'Long_name',desc
    oY->AddMetaData,'Units',(*(*davePtr).dataStrPtr).commonStr.yunits
    oY->AddMetaData,'Distribution',dist

    oCommonCon->Add, oY, parameter_name=yAxistype
endif
```

The dependent data section is dealt with in the same way as for the independent axes a indicated in the next code segment. However, the metadata are slightly different. The 'Signal' item is set to 1 and is not really useful in the current context but is included for forward compatibility reasons. The 'Axes' item is set to the names of the independent axes and it is also included here for forward compatibility reasons only. Note that the axisType property is set to '2'.

```
; The data
; Store in IDLitDataDave object using a suitable type and additional metadata.
desc = (*(*davePtr).dataStrPtr).commonStr.histlabel
zname = (strtrim(desc) eq '')? 'Data' : desc
zAxisType = '2'
axes = (dims eq 1)? xname : [xname,yname]
data = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).qty

oZ = obj_new('IDLitDataDave',data,name=zname,type=partype(data),description=desc,axisType=zAxisType)
oZ->AddMetaData,'Signal',1
oZ->AddMetaData,'Axes',axes
oZ->AddMetaData,'Long_name',desc
oZ->AddMetaData,'Units',(*(*davePtr).dataStrPtr).commonStr.histunits

oCommonCon->Add, oZ, parameter_name=zAxisType
```

The uncertainty in the dependent data is stored in a separate object as shown next. There is no metadata for this object and the axisType property is set to '3'.

```
; The data error
; Store in an IDLitDataDave object
ename = 'Error'
eAxisType = '3'
desc = 'Uncertainty in '+zname
data = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).err
oErr = obj_new('IDLitDataDave',data,name=ename,type=partype(data),description=desc,axisType=eAxisType)
oCommonCon->Add, oErr, parameter_name=eAxisType
```

This concludes the commonStr section.

The last section of this method deals with the contents of the specificPtr. The specificPtr is optional and may not always be present in the davePtr.. The code snippet below first checks for the existence of the specificPtr. If it is present then call the helper method DAVEReadDave1::ReadSpecific is called to deal with it (ReadSpecific is discussed below):

```
;_________________________________________________
; Deal with specficPtr
specificPtr = (*(*davePtr).dataStrPtr).specificPtr
if (ptr_valid(specificPtr) && $
    (n_elements(specificPtr) gt 0)) then begin

    void = self->ReadSpec(specificPtr,oData,name='Miscellaneous' $
                          ,description='Instrument Specific Details' $
                          ,type='DAVE1SPECIFICPTR')
endif

return, oData

end
```

After the dataset object is fully constructed, it is return to the caller (GetData) of this method.

The DAVEReadDave1::ReadSpecific method

This method is responsible for converting the data in the specificPtr component of the davePtr structure. It is written as a recursive function. A recursive function is one that calls itself! Its implementation is now shown:

```
function DAVEreadDAVE1::ReadSpecific, dataStr, parent, type=type, _EXTRA=etc
compile_opt idl2

switch size(dataStr,/tname) of
```

```
    'UNDEFINED':
    'OBJREF': break

    'POINTER': begin
        if ptr_valid(dataStr) then begin
            if (~keyword_set(type)) then type='POINTER'
            void = self->ReadSpecific(*dataStr,parent,type=type,_EXTRA=etc) ; ,type='POINTER'
        endif
        break
    end

    'STRUCT': begin
        tags = tag_names(dataStr)
        ntags = n_tags(dataStr)
        if (ntags le 0) then break

        if (~keyword_set(type)) then type='STRUCTURE'
        child = obj_new('IDLitDataContainer',type=type, _EXTRA=etc) ;,type='STRUCTURE'
        if (~obj_valid(child)) then break

        parent->add, child

        for i=0,ntags-1 do $
          void = self->ReadSpecific(dataStr.(i),child,name=tags[i] $
                                ,description=tags[i])

        break
    end

    else: begin
        if (n_elements(dataStr) gt 0) then begin
            if (~keyword_set(type)) then type=partype(dataStr)
            child = obj_new('IDLitData',dataStr,type=type,_EXTRA=etc)
            if (obj_valid(child)) then parent->add, child
        endif
        break
    end

endswitch

return, 1
end
```

The content of the specificPtr is not predetermined but is expected to point to a structure. The method is therefore implemented in a generic fashion so that it can cope with any type of data. Each item in the specificPtr is examined and depending on the intrinsic data type of the item, an appropriate block of code is executed. The most important observation is that the method recurses or descends into a structure to retrieve all its field data. If a pointer is encountered, it is dereferenced and the method called again. Data items are stored in IDLitData objects and structures are represented by IDLitDataContainers. Also, because ReadSpecific is a recursive function, only a single call to it (from DavePtrToObject) is required to traverse the entire contents of the specificPtr.

Of course, to use any defined reader class, it must be registered. This is done in davesetup.pro using the following syntax:

```
oSystem->RegisterFileReader, 'DAVE1x','DAVEreadDAVE1',icon='open'
```

where the name 'DAVE1x' now becomes the component name for this reader. Subsequently, when the File Open operation is being created in the DAVETool class (see DAVETool::MenuLayout method) to handle DAVE 1.x files, the registered component name is specified through the readerNames keyword:

```
self->RegisterOperation, 'DAVE 1.x', 'DAVEopFileOpen', ICON='open' $
  ,DESCRIPTION='Open DAVE 1.x file(s)',IDENTIFIER='DataIO/Read/READDAVE1',readerNames='DAVE1x'
```

Thus indicating that when the

Data Input/Output => Read Dataset From => DAVE 1.x

menu is selected the registered reader component 'DAVE1x' is be used.

Writing Operations

Operations are iTool components that modify data or the state of an application is some way. A developer writing a new iTool based application, will probably find that much of their effort will be spent on writing operations. The operations API provide a well defined interface for modifying data in a way that gives the end user adequate control over their actions. If properly implemented, every operation has a built-in undo and redo functionality.

For the developer, operations play an important role in DAVE and as such this manual will review a few examples on creating an operation. The examples expose different levels of difficulty and functionality and should hopefully help you in writing your own custom operations.

Writing a custom operation involves subclassing from either *IDLitOperation* or *IDLitDataOperation*.

You should use the IDLitDataOperation as a superclass if your operation modifies data only. In addition, the data is usually selected from a visualization or plot window and is restricted to the precise item that is selected. So, if the line in a line plot is selected, only the dependent data will be passed through to the operation for modification! For these reasons, IDLitDataOperation based operations are mostly useful if you are writing data visualization or image processing functionality. IDLitDataOperation operations are called *data-centric* operations.

You should use the IDLitOperation as a superclass if you are writing a more general operation. For most cases in DAVE, this would be the more appropriate choice particularly when writing data analysis operations. The main reason is because data analysis tasks will usually require complete datasets (independent, dependent, and uncertainty components). For example, it is not advisable to scale the dependent data without doing the same to the dependent data error. Operations that are based on IDLitOperation are referred to as *generalized* operations.

All example operations discussed here are generalized operations. (See the iTool Developer's Guide for examples of data-centric operations.)

A generalized operations class must implement the following 3 public methods (in addition to life cycle, get and set methods)

DoAction
UndoOperation
RedoOperation

As their names suggest, these three methods are used by the operation to respectively do the main operation task, undo the operation and to redo the operation when requested.

If it is important to keep a record of the operation and/or application properties before and after the operation is performed, then the following two additional public methods must be implemented also:

RecordInitialValues
RecordFinalValues

Note that the five methods mentioned refer to the public methods of the class – additional helper methods to accomplish the goals of the public methods can be used as necessary.

The Transpose operation

The objectives of the Transpose operation are straightforward: for a 2-dimensional dataset, transpose the dependent data and swap the independent axes.

The operation is implemented in the DAVEopDataTranspose class which inherits from IDLitOperation. The public interface for the class are

DoAction
UndoOperation
RedoOperation

The class structure definition is outlined below:

```
pro DAVEopDataTranspose__define

compile_opt idl2

struct = {DAVEopDataTranspose $
          ,inherits IDLitOperation $
         }

end
```

DAVEopDataTranspose::Init method

```
function DAVEopDataTranspose::Init, _REF_EXTRA=etc
compile_opt idl2

; call superclass init
if (~self->IDLitOperation::Init(types=['DAVE1COMMONSTR','ASCIISPE','ASCIIGRP'] $
                               ,NAME='Data Transpose' $
                               ,_EXTRA=etc)) then return, 0

; Unhide the SHOW_EXECUTION_UI property
self->SetPropertyAttribute, 'SHOW_EXECUTION_UI', hide=1

; This operation is reversible
; This operation is not expensive
self->SetProperty, reversible_operation=1, expensive_operation=0

; return success
return, 1

end
```

First, a call to the superclass Init method is made. Note the use of the *types* keyword which is set to an array of types. Remember that IDLitData objects have a type property which can be set to any scalar string. By defining a types property for this operation, we are stating that this operation will only accept data objects whose type property matches the specified list.

Note: an application menu linked with an operation will be activated or deactivated based on whether the *type* of the selected data (or its subcomponent) matches the *types* property of the operation.

The *SHOW_EXECUTION_UI* property of an operation determines whether a dialog (UI service) should be displayed to retrieve information from the user before completing the operation. If its *hide* attribute is set to 1, as it is in this case, then it means no dialog should be shown and if it is set to 0, then a dialog should be displayed. If not set, the default for hide is 1.

A key property of an operation which must be set in the Init method is the *REVERSIBLE_OPERATION*. If this is set to 1, as it is here, it means this operation is reversible and if set to 0, it is not. For a reversible operation, the the changes that are made to data or application state can be reversed. In an operation with the REVERSIBLE_OPERATION set to 0, the data or application state is cached *prior* to making any changes; afterwards, undoing the action simply involves retrieving the cached data and restoring it. When an operation is reversible, it can be undone by applying an operation rather than restoring a stored value. By default, operations are not reversible.

Another important property that should be set in the Init method is the *EXPENSIVE_OPERATION* property. It is used to determine if the operation is computational expensive (set to 1) or not (set to 0). The default is 0 as in this case. Expensive computations are those that require significant memory or processing time to execute. In an expensive operation, the computed information is stored *afterwards* and is used to *redo* the operation if requested.

Note: REVERSIBLE_OPRATION and EXPENSIVE_OPERATION properties are must be set for a data-centric operation data caching is handled by the IDLitDataOperation superclass.

DAVEopDataTranspose::DoAction method

This method should implement the tasks required of the operation. It is called by the iTool system whenever the operation is used. If successful, it should return an IDLitCommandSet object – a CommandSet object is used to store the commands that were used in carrying out the task for this operation.

Note: the DoAction method serves as the entry point to the operation.

The code to transpose the data is is now discussed in sections.

```
function DAVEopDataTranspose::DoAction, oTool
compile_opt idl2

; oTool should be valid
if (~obj_valid(oTool)) then return, obj_new()
```

The function accepts a single input argument which would be set to the object reference of the tool requesting the operation.

A simple check is made to ensure that the input object is valid. Note that since this method is expected to return an object if successful, you should return a NULL object if exiting prematurely

```
; Get the selected dataset(s)
oSelections = oTool->GetSelectedData()
void = where(obj_valid(oSelections),cnt)
if (cnt eq 0) then begin
    oTool->StatusMessage, 'No valid data to operate on! Select a dataset from the Data Browser 
tree.'
    return, obj_new()
endif
```

In the above code, the currently selected datasets are retrieved. Print a helpful message in the status bar and exit, if there are no selected data.

```
; Locate valid dataset containers that can be transposed
self->GetProperty, types=validTypes ; expect ['DAVE1COMMONSTR','ASCIISPE','ASCIIGRP','ASCIICOL']
nValid = n_elements(validTypes)
for i = 0,n_elements(oSelections)-1 do begin
   ;; search for one of the valid types from this selction
   j = 0
   repeat begin
      oRes = oSelections[i]->GetByType(validTypes[j],count=found)
   endrep until (found || ++j ge nValid)

   if (~obj_valid(oRes)) then continue
   oTarget = (n_elements(oTarget) gt 0)? [oTarget,oRes] : oRes
   oSel = (n_elements(oSel) gt 0)? [oSel,oSelections[i]] : oSelections[i]
endfor
if (n_elements(oTarget) eq 0) then begin
    oTool->StatusMessage, 'Invalid data selection(s) - no data that can be transposed!'
    return, obj_new()
endif
```

The above code snippet shows how to determine valid datasets that can be transposed. First, retrieve the list of valid types from the operation.

Then loop through the selected dataset and find data items that match the valid types. Note the use of the IDLitData::GetByType method that was introduced in the Data Handling section. The valid data objects are stored in the oTarget object array..

If after completing the search, no valid objects were found, print out a status message and exit.

```
; Create a command set obj by calling the base class DoAction
oCmdSet = self->IDLitOperation::DoAction(oTool)
```

So far, we have some valid data to operate. Now get a new CommandSet object by calling the superclass method as shown above.

Finally, loop through the valid targets and transform their data as shown next:

```
; Transpose data in selected targets
; NB: no properties are being modified for the target objects hence
; there are no initial/final values to be recorded
for i = 0,n_elements(oTarget)-1 do begin

    ;; transpose data in target
    if (~self->Transpose(oTarget[i])) then continue

    ;; record transposed targets on a command object
    oCmd = obj_new('IDLitCommand',target_identifier=oTarget[i]->getfullidentifier())
    oCmdSet->Add, oCmd
endfor

; return the command set obj
return, oCmdSet

end
```

A helper method DAVEopDataTranspose::Transpose is called to perform the actual transpose.

For each target that is successfully transposed, create a IDLitCommand object to store the full object identifier of the target object (note that the identifier not the object reference is used). The IDLitCommand class provides a tag-based mechanism for storing and retrieving information which allows the iTool developer to easily implement undo and redo functionality for an operation. The full functionality of the Command object is not exposed in this example but will be in a later example.

For each successful target, a command is added to the CommandSet.

Lastly, the CommandSet object is returned. This object contains all the relevant information about what was done by this operation and is used by the application tool to create an undo/redo buffer.

The DAVEopDataTranspose::Transpose method.

This method is called by the DoAction method to transpose the data contained in a parameterset object. The function returns a 1 if successful.

The function accepts a single argument which is the object reference to a ParameterSet containing the data to be transposed. The code listing is now discussed.

```
function DAVEopDataTranspose::Transpose, oParmSet
compile_opt idl2

oDep = oParmSet->GetByName('2')
if (~obj_valid(oDep) || ~obj_isa(oDep,'IDLitData')) then return, 0
if (~oDep->GetData(zData)) then return, 0
if (size(zData,/n_dimensions) ne 2) then begin
    oTool->StatusMessage,'Can only transpose datasets with 2 independent variables!'
    return, 0
endif
oErr = oParmSet->GetByName('3') ; the error
errPresent = obj_valid(oErr)
oInd1 = oParmSet->GetByName('0') ; first independent
oInd2 = oParmSet->GetByName('1') ; second independent
if (~obj_valid(oInd1) || ~obj_valid(oInd2)) then return, 0
```

The first task is to retrieve the various components of the parameter set object by using the IDLitParameterSet::GetByName method in association with the parameter_name that was used when adding the data to the container. (See the DAVE1DReader section, especially the construction of the DAVE1COMMONSTR parameterset). Recall that the labels '0','1','2' and '3' were used to indicate the first independent axis (x-axis), the second independent axis (y-axis), the dependent axis (z-axis) and the error in the z-axis respectively. These labels are now used to selectively retrieve objects from the parameterset.

The dependent data object is first obtained and its data extracted using the IDLitData::GetData method. If the dimensions of the dependent data is not 2 (that is if the data is not 2D) then print out an appropriate status message and return failure.

Retrieve the remaining data objects and ensure they are valid.

```
;; Swap axis types for the independent axes. The axis type property
;; is used to automatically match data and visualisation parameters
oInd1->GetProperty, axisType=xType
oInd2->GetProperty, axisType=yType
oInd1->SetProperty, axisType=yType
oInd2->SetProperty, axisType=xType

;; Swap the parameter names for the independent axes data
oParmSet->remove, [oInd1,oInd2]
oParmSet->Add, [oInd1,oInd2], parameter_name=strtrim(string(fix([yType,xType])),2)
```

If the dependent data is to be transposed, the axes data must be swapped also! The above code accomplishes this by swapping the axisType property of the independent axes objects.

After swapping the axisType property, also change the parameter_name used to associate the objects in the ParameterSet. This is done by removing the axes object from the ParameterSet and reinserting them using the swapped axis types. yType and xType are bytes so must be converted to strings.

```
;; transpose the data and error
void = oDep->SetData(transpose(zData),/no_copy,/no_notify)
if (errPresent) then begin
    void = oErr->GetData(eData)
    void = oErr->SetData(transpose(eData),/no_copy,/no_notify)
endif
```

Next, transpose the data as shown above. The dependent data was previously extracted and stored in the zData variable. Use the IDLitData::SetData method to replace the dependent objects data with its transpose. Setting the /no_copy keyword means zData is placed in the object and not a copy. The /no_notify keyword is used to suppress any notifications messages about the change in data that would otherwise be sent out to observers of the dependent data object.

Note: by default, whenever the data in an IDLitData object changes, a notification message is sent out to all observers of the object. Setting the no_notify keyword prevents this from happening. This is important when making a series of changes and we want to avoid making updates in response to partial or incomplete changes!

If the error object is present, also transpose its data while suppressing messages.

```
;; If there are visualizations using this dataset, the parameter data
;; of the visualization need to be swapped to reflect the transposed dataset.
oVis = oDep->getdataobservers(/all,ISA='IDLitVisualization',count=visCnt)
if (visCnt gt 0) then begin
    for i=0,visCnt-1 do begin
        if (obj_valid(oVis[i]) && obj_isa(oVis[i],'IDLitVisualization')) then begin
            oX = oVis[i]->GetParameter('X',count=xCnt) ; Note: oX should be equiv to oInd1, etc
            oY = oVis[i]->GetParameter('Y',count=yCnt)
            if ((xCnt eq 1) && (yCnt eq 1)) then begin
                void = oVis[i]->setData(oX,parameter_name='Y',/no_update)
                void = oVis[i]->setData(oY,parameter_name='X',/no_update)
            endif
        endif
    endfor
endif
```

If there are any existing visualizations created with this dataset, then it is necessary to swap their x- and y-axis as shown above. First retrieve all observers of the dependent data that are visualizations. If there are, then loop through the visualizations, retrieve their independent axes parameters

Note: All visualizations in the iTool framework use the labels 'X' and 'Y' to specify their x-axis and y-axis parameters respectively.

Then, swap the data attached to the parameters to reflect the current axes data. The /no_update keyword prevents the visualization from updating itself.

```
;; notify observers about data change
oDep->NotifyDataChange
```

```
if (errPresent) then oErr->NotifyDataChange
oInd1->NotifyDataChange
oInd2->NotifyDataChange

oDep->NotifyDataComplete
if (errPresent) then oErr->NotifyDataComplete
oInd1->NotifyDataComplete
oInd2->NotifyDataComplete

return, 1
end
```

Now that all changes are complete, it is okay to update observers. This is accomplished as shown above by first calling the IDLitData::NotifyDataChange method on all the data objects followed by a call to the IDLitData::NotifyDataComplete method. These calls will trigger all observers to update themselves accordingly.

The DAVEopDataTranspose::UndoOperation method

This method is called whenever the operation's actions need to be undone. This typically occurs when the user selects the Undo functionality from the application's menu or toolbar.

The operation is reversible, and is accomplished as follows:

```
function DAVEopDataTranspose::UndoOperation, oCmdSet
compile_opt idl2

; Retrieve the command object (there is only one for this operation)
; from the command set
oCmds = oCmdSet->Get(/all,count=nCmds)
if (nCmds lt 1) then return, 0

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0

; Loop through rest of commands and undo the changes that were made
; to the targets
for i=0,nCmds-1 do begin
    oCmds[i]->GetProperty, target_identifier=idTarget
    oTarget = oTool->GetByIdentifier(idTarget)
    if (~obj_valid(oTarget)) then begin
        oTool->StatusMessage,'Missing dataset! Undo could not be completed'
        continue
    endif

    ;; transpose data in target
    if (~self->Transpose(oTarget)) then continue
endfor

return, 1

end
```

The function accepts a single input argument which is the CommandSet object that was generated by DoAction method. The CommandSet object contains a list of commands that were executed when the action was performed.

First, retrieve the commands from the CommandSet. Exit if no commands are found. Also get the tool object.

Loop through the command objects and do the following:

- Get the target identifier for the command. Recall that this was the only information retained in the command. The identifier refers to the dataset object that was transposed.
- Use the IDLitTool::GetByIdentifier method to retrieve the target object, knowing its identifier string.
- If the target object is not valid, display a status message and continue to the next loop
- Transpose the target object by calling on the Transpose method – this reverses the previous transpose.

The DAVEopDataTranspose::RedoOperation method

This method is called to repeat an action that was previously performed. This typically occurs when the user selects the Redo functionality from the application's menu or toolbar.

Note: You can only redo an action that was previously undone.

This operation is not expensive (so it is not being redone from a cache) and is accomplished in the following code:

```
function DAVEopDataTranspose::RedoOperation, oCmdSet
compile_opt idl2

; Retrieve the command object (there is only one for this operation)
; from the command set
oCmds = oCmdSet->Get(/all,count=nCmds)
if (nCmds lt 1) then return, 0

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0


; Loop through command objects and redo the transpose
for i=0,nCmds-1 do begin
    oCmds[i]->GetProperty, target_identifier=idTarget
    oTarget = oTool->GetByIdentifier(idTarget)
    if (~obj_valid(oTarget)) then begin
        oTool->StatusMessage,'Missing dataset! Redo could not be completed'
        continue
    endif

    ;; transpose data in target
    if (~self->Transpose(oTarget)) then continue
endfor

return, 1

end
```

The method is virtually identical to the UndoOperation. It essentially cycles through the valid targets and transpose the data again. It assumes that the initial transpose has been undone.

After an operation class is defined, it must be registered before it can be used. This should be done in the tool class definition from which you wish the operation to appear. DAVEopDataTranspose is registered in the DAVETool class in the MenuLayout method (and called by DAVETool::Init) using the following syntax

```
self->RegisterOperation, 'Transpose','DAVEopDataTranspose' $
  ,description='Transpose dataset (swap independent axes)' $
  ,identifier='Analysis/DataTrans/Transpose'
```

Thus DAVEopDataTranspose becomes registered as a 'Transpose' component and appears in the Analysis menu folder.

To use the Transpose operation:

- First, select a dataset you wish to transpose from the Data Manager folder.
- Then select the

  Analysis => Data Transform => Transpose

  menu item. This would cause the selected data to be transposed, if it contains data that can be transposed.
- An entry will be automatically created in the applications undo/redo buffer.
- If there are any components (for example a visualization) currently using this data, they will be automatically updated.
- If you hover your mouse over the undo toolbar button, you should observe a new tooltip containing the text 'Undo Transpose'. If you click on this button, the transpose action will be undone.
- If the transpose is undone, you will be able to redo the action. Hover your mouse over the redo toolbar item. Again you should observe a tooltip containing the text 'Redo Transpose'. Click on the button to redo the action.

Writing a UI Service

In many situations, an iTool component may need to either present information or receive input from the user in order to complete its function. Modal dialogs are typically required in these situations and are included in the iTool scheme as a user interface service or UI service. Fortunately, there are UI Services that come bundled with the framework – the most famous of these is the PropertySheet UI service which can be use to display/retrieve information about an object. However, in situations where the keyword/value pair interface of the property sheet is too restrictive, you can build your own custom UI Service.

A UI service traditionally has two parts:

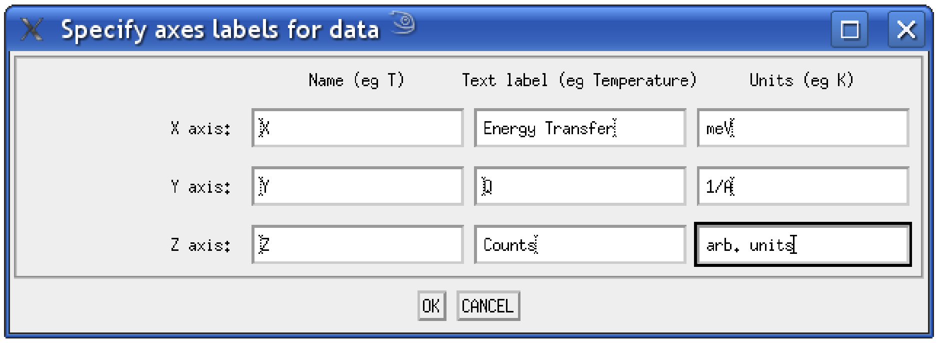
– a conventional widget creation routine that constructs the modal dialog and handle events

– a wrapper function that is used to connect the conventional dialog to the iTools framework.

Note: It is helpful to maintain a convention of using the same base name for the two parts but distinguishing between them by adding a prefix or suffix to the base name. In DAVE, the prefixes 'wd_' and 'ui_' are used for the conventional and wrapper functions respectively.

In this example we will explore a UI Service that is used by the ASCII Text reader component to retrieve axis label and units information from the user when loading GRP and SPE files.

When an ASCII SPE or GRP file is being loaded, there is insufficient information in the file to determine things like axis name, title or units. The UI Service is therefore called to let the user provide the information if they so wish. If the information is supplied, the reader then integrates it with the rest of the data read from the file. A screenshot of the UI Service is shown.



	Name (eg T)	Text label (eg Temperature)	Units (eg K)
X axis:	X	Energy Transfer	meV
Y axis:	Y	Q	1/Å
Z axis:	Z	Counts	arb. units

Figure 7: SetDataAxisLabel UI Service dialog

The conventional dialog: wd_SetDataAxisLabels

This dialog widget is implemented in the following code:

```
pro wd_setDataAxesLabels_event, event
  compile_opt idl2

  uname = widget_info(event.id,/uname)
  widget_control, event.top, get_uvalue=sPtr

  list = ['XN','YN','DN','XT','YT','DT','XU','YU','DU']

  case uname of
     'OK': begin
        for i=0,n_elements(list)-1 do begin
           id = widget_info(event.top,find_by_uname=list[i])
           widget_control, id, get_value=lab
           (*sPtr).label[i] = lab
        endfor

        (*sPtr).ok = 1

        widget_control, event.top, /destroy
     end

     'CANCEL': begin
        (*sPtr).ok = 0
        widget_control, event.top, /destroy
     end

     else: begin
        (*sPtr).ok = 0
        widget_control, event.top, /destroy
     end
  endcase
end

function wd_setDataAxesLabels, label, group_leader=gpl
  compile_opt idl2

  title = 'Specify axes labels for data'
  if (n_elements(gpl) gt 0) then begin
     modal = 1
     wTLB = widget_base(/col,title=title,/base_align_center,group_leader=gpl,modal=modal)
  endif else begin
     modal = 0
     wTLB = widget_base(/col,title=title ,/base_align_center,modal=modal)
  endelse
  cb0 = widget_base(wTLB,/frame,/col)
  labs = ['X axis: ','Y axis: ','Z axis: ']
  uN = ['XN','YN','DN']
  uT = ['XT','YT','DT']
  uU = ['XU','YU','DU']
  label = (n_elements(label) eq 0)? strarr(9) : label

  xs = 600
  cb = widget_base(cb0,/grid,col=4,scr_xsize=xs)
  void = widget_label(cb,value='')
  void = widget_label(cb,value='Name (eg T)')
  void = widget_label(cb,value='Text label (eg Temperature)')
  void = widget_label(cb,value='Units (eg K)')
  for i=0,2 do begin
     cb = widget_base(cb0,/grid,col=4,scr_xsize=xs)
     void = widget_label(cb,value=labs[i],/align_right)
     void = widget_text(cb,uname=uN[i],value=label[i],/editable)
     void = widget_text(cb,uname=uT[i],value=label[i+3],/editable)
     void = widget_text(cb,uname=uU[i],value=label[i+6],/editable)
  endfor

  cb1 = widget_base(wTLB,/row)
  void = widget_button(cb1,value='OK',uname='OK')
  void = widget_button(cb1,value='CANCEL',uname='CANCEL')

  sPtr = ptr_new({label:label,ok:0B})
  widget_control, wTLB, /realize, set_uvalue=sPtr

  xmanager,'wd_setDataAxesLabels',wTLB,no_block=modal

  label = (*sPtr).label
  retVal = (*sPtr).ok
  ptr_free, sPtr

  return, retVal

end
```

The function is written as modal dialog if the group_leader keyword is supplied by the caller.

The main function has an input variable which is an input/output argument called label which may or may not be defined. If it is defined it should contain a string array of default values of the pieces of information that the user is being asked to supply.

A dialog with 9 text fields is created and the user is asked to supply the requested information. Any information entered by the user is packaged into the label variable.

If the user exits the dialog by selecting either the CANCEL or OK buttons – in either case the choice is recorded and return to the calling program.

That is the end of the conventional dialog function. Obviously, this can be used by any IDL program since it contains no reference to any iTool functions.

The wrapper function: ui_SetDataAxisLabels

The wrapper function must follow an API that is required by the iTool framework. The function is typically very brief and is outline below:

```
function ui_setDataAxesLabels, oUI, oRequester
compile_opt idl2
```

```
oUI->GetProperty, group_leader=wTLB

oRequester->GetProperty, label=label

success = wd_setDataAxesLabels(label, group_leader=wTLB)

if (success) then oRequester->SetProperty, label=label

return, success

end
```

It must accept two input parameters.

The first is an IDLitUI object and the second is an object reference of the component that requested the UI Service.

Retrieve the widget ID of the main application's top level base (TLB) from the UI object.

It is expected that any component that is requesting this service should implement a label keyword in their GetProperty and SetProperty methods.

Note: In general, a UI Service is setup to handle a specific kind of information and the requester is expected to know about this. They would then present a way (usually through the Get/SetProperty methods) for the ui wrapper to receive/supply that information from/to the requester.

Call the requester object to retrieve the current setting for the label property.

Execute the modal dialog so the user can make changes

If the user exited the dialog successfully (clicking OK button in this case), then send the modified label information to the requester by calling its GetProperty method.

Then exit with either 1 for success or 0 for failure.

As with all components in the iTool framework, it has to be registered before it can be used. Only the wrapper function should be registered and it is accomplished as followed in davesetup.pro:

```
oSystem->RegisterUIService, 'SetDataAxesLabels', 'ui_setDataAxesLabels'
```

The component name 'SetDataAxesLabels' will then become available for use.

To utilize a UI Service component, you have to use the following syntax:

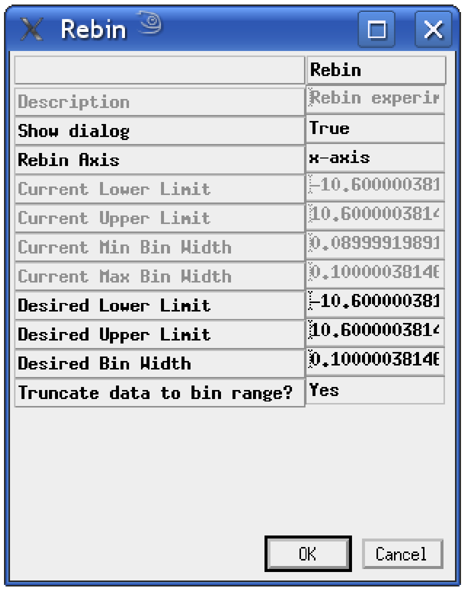
```
success = oTool->DoUIService('SetDataAxesLabels',oRequester)
```

from within the component that wishes to retrieve label information from the user. In DAVE this call is made in the ASCII Text reader (DAVEReadAscii class) and in this case oRequester is set to Self.

The Scale Operation

The Rebin Operation

The rebin operation is used to rebin a 1D or 2D dataset. If it is a 2D dataset, then the rebin can be done with respect to either the x- or y-axis but not both simultaneously. The rebin operation launches a UI service which request inputs from the user as shown in the following screenshot:



	Rebin
Description	Rebin experir
Show dialog	True
Rebin Axis	x-axis
Current Lower Limit	-10.600000381
Current Upper Limit	10.6000003814
Current Min Bin Width	0.08999919891
Current Max Bin Width	0.1000003814
Desired Lower Limit	-10.600000381
Desired Upper Limit	10.6000003814
Desired Bin Width	0.1000003814
Truncate data to bin range?	Yes

Figure 8: The rebin operation's information dialog

The operation is implemented in the DAVEopDataRebin class which inherits from IDLitOperation. The public interface for the class are

DoAction
RecordInitialValues
RecordFinalValues
DoExecuteUI
UndoOperation
RedoOperation

These methods were all introduced earlier except for DoExecuteUI. DoExecuteUI is used for launching a user interface dialog to retrieve information from the user.

We will now examine all the methods of the class

The class structure definition is outlined below:

```
pro DAVEopDataRebin__define

compile_opt idl2

struc = {DAVEopDataRebin $
         ,inherits IDLitOperation $
         ,dMin:0.0 $              ; bin axis minimum of existing data
         ,dMax:0.0 $              ; bin axis maxinum of existing data
         ,dBinMin:0.0 $           ; min bin width of existing data
```

```
        ,dBinMax:0.0 $          ; max bin width of existing data
        ,min:0.0 $              ; min of axis range to be rebin
        ,max:0.0 $              ; max of axis range to be rebin
        ,bin:0.0 $              ; bin width to be rebin to
        ,axis:0 $               ; rebin axis (for 2D)
        ,truncate:0 $           ; set to 1 if data to be truncated
        ,setdefault:0 $         ; a control flag
        ,oTarget:obj_new() $    ; the current target object.
        ,oSel:obj_new() $       ; the current selected data item
       }

end
```

As expected, DAVEopDataRebin inherits from IDLitOperation and adds a bunch of data members that are used to characterize the binning operation.

The DAVEopDataRebin::Init method

The class constructor is outline below

```
function DAVEopDataRebin::Init, _REF_EXTRA=etc
compile_opt idl2

; call superclass init
; This operation is not reversible
; This operation is expensive
if (~self->IDLitOperation::Init(types=['DAVE1COMMONSTR','ASCIISPE','ASCIICOL','ASCIIGRP'] $
                                ,NAME='Data Rebin' $
                                ,reversible_operation=0,expensive_computation=1 $
                                ,_EXTRA=etc)) then return, 0

;self->SetProperty, reversible_operation=0, expensive_computation=1

; Unhide the SHOW_EXECUTION_UI property
self->SetPropertyAttribute, 'SHOW_EXECUTION_UI', hide=0

; Register an offset property for this operation
self->RegisterProperty, 'axis', enumlist=['x-axis','y-axis'], description='Rebin Axis' $
  ,name='Rebin Axis',sensitive=1
self->RegisterProperty, 'dmin', /float, description='Lower limit of data' $
  ,name='Current Lower Limit',sensitive=0
self->RegisterProperty, 'dmax', /float, description='Upper limit of data' $
  ,name='Current Upper Limit',sensitive=0
self->RegisterProperty, 'dbinmin', /float, description='Mininum bin width of data' $
  ,name='Current Min Bin Width',sensitive=0
self->RegisterProperty, 'dbinmax', /float, description='Maximum bin width of data' $
  ,name='Current Max Bin Width',sensitive=0
self->RegisterProperty, 'min', /float, description='Desired lower limit' $
  ,name='Desired Lower Limit',sensitive=1
self->RegisterProperty, 'max', /float, description='Desired upper limit' $
  ,name='Desired Upper Limit',sensitive=1
self->RegisterProperty, 'bin', /float, description='Desired bin width' $
  ,name='Desired Bin Width',sensitive=1
self->RegisterProperty, 'truncate', enumlist=['No','Yes'], description='Truncate data to rebin
range?' $
  ,name='Truncate data to bin range?',sensitive=1

; init offset to 0.0
self.axis = 0
self.truncate = 1
self.setdefault = 1
```

```
; return success
return, 1

end
```

The superclass Init method is called using the types keyword to define acceptable dataset types. In addition, the reversible_operation and expensive_operation keywords are used to indicate that the rebin operation is not reversible and expensive!

The hide property of the SHOW_EXECUTION_UI is set to 0 meaning that the UI Service dialog should be displayed.

After this, a bunch of properties are defined for the class. The intention is therefore to use the PropertySheet UI Service.

Note: Properties that are registered using IDLitComponent::RegisterProperty method become visible if an object of this class is displayed in a property sheet widget.

The registered properties are all parameters that can be used to control the rebin operation. Some of the registered properties will be used for informational purposes only and so their sensitive attributes are set to 0 – they will be appear grayed out in the property sheet.

Finally, some data members are initialized – the axis defaults to 0 (x-axis), the rebinned data will be truncated to the min and max range supplied by the user.

The DAVEopDataRebin::DoAction method

This method is called whenever the rebin operation is called into action.

```
function DAVEopDataRebin::DoAction, oTool
compile_opt idl2

; oTool should be valid
if (~obj_valid(oTool)) then return, obj_new()

; Get the selected dataset(s)
oSelections = oTool->GetSelectedData()
void = where(obj_valid(oSelections),cnt)
if (cnt eq 0) then begin
    oTool->StatusMessage, 'No valid data to operate on! Select a dataset from the Data Browser
tree.'
    return, obj_new()
endif
```

As discussed previously, a few checks are performed initially. Exit, if the tool is not valid or there are no selected datasets checks the validity of the tool object.

```
; Locate valid dataset containers that can be rebinned
self->GetProperty, types=validTypes ;expext ['DAVE1COMMONSTR','ASCIISPE','ASCIIGRP','ASCIICOL']
nValid = n_elements(validTypes)
for i = 0,n_elements(oSelections)-1 do begin
   ;; search for one of the valid types from this selction
   j = 0
   repeat begin
      oRes = oSelections[i]->GetByType(validTypes[j],count=found)
   endrep until (found || ++j ge nValid)
   if (~obj_valid(oRes)) then continue
   oTarget = (n_elements(oTarget) gt 0)? [oTarget,oRes] : oRes
   oSel = (n_elements(oSel) gt 0)? [oSel,oSelections[i]] : oSelections[i]
endfor
if (n_elements(oTarget) eq 0) then begin
    oTool->StatusMessage, 'Selection does not contain data that can be rebinned!'
    return, obj_new()
endif

; Make the first valid target the operation's target dataset
self.oTarget = oTarget[0]
self.oSel = oSel[0]             ; also record the selected data containing the target dataset

; Create a command set obj by calling the base class DoAction
oCmdSet = self->IDLitOperation::DoAction(oTool)
```

In the code segment above, get the data types defined for this operation and locate valid data targets from within the data selection. If there are no valid data targets, exit.

Only the first valid target is processed further. Essentially it is not possible to rebin multiple datasets simultaneously. The basic reason is to reduce the possibility of error in attempting to apply the same bin parameters to multiple datasets that are unrelated!

Create the CommandSet object to hold the commands created for this operation by calling the superclass DoAction method.

```
; Is some UI needed prior to execution?
self->GetProperty, show_execution_ui=doUI
hasPropSet = 0b
if (doUI) then begin

    ;; Some operation properties depend on the selected data: update them
    self.setdefault = 1
    self->_UpdateCurrentBinSettings

    ;; Record initial properties for this operation.
    if (~self->RecordInitialValues(oCmdSet,self,'')) then begin
        obj_destroy, oCmdSet
        return, obj_new()
    endif

    ;; Perform our UI.
    if (~self->DoExecuteUI()) then begin
        obj_destroy, oCmdSet
        return, obj_new()
    endif

    ;; The hourglass will have been cleared by the dialog.
    ;; So turn it back on.
    ;void = oTool->DoUIService("HourGlassCursor", self)

    ;; Record final properties for this operation.
    if (~self->RecordFinalValues(oCmdSet,self,'')) then begin
        obj_destroy, oCmdSet
        return, obj_new()
    endif
endif
```

In the above code, retrieve the SHOW_EXECUTION_UI setting. If it is set, then do the following:

- Use the private method DAVEopDataRebin::_UpdateCurrentBinSettings to reset the bin properties of the operation so that they are consistent with the target object data.

- Record the initial properties of the operation by calling the RecordFinalValues method.
- Display a dialog so the user can specify bin settings.
- Record the final properties of the operation using the RecordFinalValues method. This keeps a record of the user's choices.

```
; Rebin the data and record changes in undo/redo buffer
f (~self->Rebin(oTarget, oCmdSet)) then begin
    obj_destroy, oCmdSet
    return, obj_new()
endif

; return the command set obj
return, oCmdSet

end
```

Finally, the target object is rebinned by calling on the DAVEopDataRebin::Rebin method and the CommandSet object is returned.

The DAVEopDataRebin::_UpdateCurrentBinSettings method

This is a method used to retrieve the data from the target object and to update the operations bin settings accordingly. This way, the user is guided to entering sensible bin settings that are compatible with the data.

```
pro DAVEopDataRebin::_UpdateCurrentBinSettings
compile_opt idl2

if (~obj_valid(self.oTarget)) then return
oAxis = (self.axis eq 0)? (self.oTarget)->GetByName('0') : $
        (self.oTarget)->GetByName('1')
if (~obj_valid(oAxis) || ~obj_isa(oAxis,'IDLitData')) then return
if (~oAxis->GetData(axisData)) then return
```

First, retrieve the appropriate independent axis object based on the value of self.axis – can rebin either with respect to x or y-axis but not both at the same time. Validate the object and retrieve the independent axis data.

```
n = n_elements(axisData)
dMin = min(axisData)
dMax = max(axisData)
bins = axisData[1:n-1] - axisData[0:n-2]
dBinMin = min(bins, max=dBinMax)

self.dmin=dmin
self.dmax=dmax
self.dbinmin=dbinmin
self.dbinmax=dbinmax
if (self.setdefault) then begin
    self.setdefault = 0
    self.min = dmin
    self.max = dmax
    self.bin = dbinmax
endif
```

Extract information from the axis data and fill out the operation data. This includes the axis data range (dmin and dmax) and the minimum and maximum bin widths of the data.

```
; If there is only one independent axis modify the axis drop-list.
if (obj_valid((self.oTarget)->GetByName('1'))) then begin
    self->SetPropertyAttribute, 'axis', enumlist=['x-axis','y-axis']
endif else begin
    self->SetPropertyAttribute, 'axis', enumlist=['x-axis']
endelse
```

If there is a second independent axis, modify the axis property so that there is an option to choose between the x-axis and y-axis.

```
; Impose certain bounds
self.bin = self.bin > self.dbinmin
self.min = (self.min > self.dmin) < self.dmax
self.max = (self.max < self.dmax) > self.dmin
if (self.min eq self.dmax) then self.min = self.dmax - self.bin
if (self.max eq self.dmin) then self.max = self.dmin + self.bin

end
```

Finally, ensure that the bin settings are consistent with the data. That is, the bin width must greater than the current minimum bin width of the data, the minimum and maximum bin ranges must be within the data limits.

The DAVEopDataRebin::RecordInitialValues method

This method is used to record the current property settings into a Command object.

```
function DAVEopDataRebin::RecordInitialValues, oCmdSet, oTarget, idProp
compile_opt idl2

; create a command object to store the values, This is the first one
; that is created for this oCmdSet
oCmd = obj_new('IDLitCommand',target_identifier=oTarget->getfullidentifier())
if (~obj_valid(oCmd)) then return, 0

; Get the value to be stored and add to command obj
oTarget->GetProperty,min=min,max=max,bin=bin,axis=axis,truncate=truncate
void = oCmd->AddItem('OLD_MIN',min)
void = oCmd->AddItem('OLD_MAX',max)
void = oCmd->AddItem('OLD_BIN',bin)
void = oCmd->AddItem('OLD_AXIS',axis)
void = oCmd->AddItem('OLD_TRUNCATE',truncate)

; Add the command to command set
oCmdSet->Add, oCmd

return, 1

end
```

The method takes three input parameters. The first is the CommandSet object for the operation and the second id the target object. The third parameter is not used.

First, create a new Command object – set the target_identifier keyword to the target object's full identifier.

Get the current values for the property's bin parameters.

Create a new entry in the Command object representing each of the bin parameters. Note the 'OLD_' prefix that is used for each of the item tags.

Finally, add the Command object to the CommandSet object.

The DAVEopDataRebin::RecordFinalValues method

This method is identical to the RecordInitialValues method.

```
function DAVEopDataRebin::RecordFinalValues, oCmdSet, oTarget, idProp
compile_opt idl2

; Retrieve the first command object from the command set
oCmd = oCmdSet->Get(position=0)
if (~obj_valid(oCmd)) then return, 0

; Get the value to be stored and add to command obj
oTarget->GetProperty,min=min,max=max,bin=bin,axis=axis,truncate=truncate
void = oCmd->AddItem('NEW_MIN',min)
void = oCmd->AddItem('NEW_MAX',max)
void = oCmd->AddItem('NEW_BIN',bin)
void = oCmd->AddItem('NEW_AXIS',axis)
void = oCmd->AddItem('NEW_TRUNCATE',truncate)

return, 1

end
```

There are 3 basic differences.

First, a new Command object is not created – the one created by RecordInitialValues is retrieved.

Second, RecordFinalValues is called after the user has specified their choice for the bin settings.

Third, the tags used to store the entries in the Command object have the prefix 'NEW_'.

The DAVEopDataRebin::DoExecuteUI method

This is a simple method that launches dialog from which the user can enter parameters to control the operation.

```
function DAVEopDataRebin::DoExecuteUI
compile_opt idl2

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0

; Use the build-in 'PropertySheet' UI service to let the user
; customize the operation's property.
return, oTool->DoUIService('PropertySheet',self)

end
```

The dialog launched is a property sheet and the object displayed in the property sheet is the operation – note the self parameter passed to DoUIService. The user is then able to set the rebin parameters by modifying the operation properties directly.

Note the PropertySheet UI Service is registered by the System object.

The DAVEopDataRebin::Rebin method

This is a helper method that actually rebins the data in the target object based on user bin settings.

```
function DAVEopDataRebin::Rebin, oTarget, oCmdSet
compile_opt idl2

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0

; Get current operation settings
self->GetProperty,min=lolim,max=uplim,bin=binw,axis=axis,truncate=truncate $
                  ,reversible_operation=rev_op,expensive_computation=exp_comp

; uplim must be greater than lolim
if (lolim gt (uplim-binw)) then begin
    oTool->StatusMessage,'Data Rebin: lower limit should not be >= upper limit!'
    return, 0
endif
```

First, get the current bin settings and perform a few checks.

```
; Get the independent data
oInd = (self.axis eq 0)? oTarget->GetByName('0') : $
       oTarget->GetByName('1')
if (~obj_valid(oInd) || ~obj_isa(oInd,'IDLitData')) then return, 0
if (~oInd->GetData(vin)) then return, 0
if (~oInd->GetMetaData('DISTRIBUTION',distribution)) then distribution = 'POINTS'
; The dependent data
oDep = oTarget->GetByName('2')
if (~obj_valid(oDep) || ~obj_isa(oDep,'IDLitData')) then return, 0
if (~oDep->GetData(zin)) then return, 0
; The error in the dependent data
oErr = oTarget->GetByName('3')
errExist = 0
if (obj_valid(oErr) && obj_isa(oErr,'IDLitData')) then begin
    errExist = oErr->GetData(zein)
endif

; Store copies of original data because rebin operation is not
; reversible.
oCmd = obj_new("IDLitCommand", TARGET_IDENTIFIER=oTarget->GetFullIdentifier())
oCmdSet->Add, oCmd
void = oCmd->AddItem('INDDATA_IN',vin)
void = oCmd->AddItem('DEPDATA_IN',zin)
void = oCmd->AddItem('INDID',oInd->GetFullIdentifier())
void = oCmd->AddItem('DEPID',oDep->GetFullIdentifier())
void = oCmd->AddItem('ERREXIST',errExist)
if (errExist) then begin
    void = oCmd->AddItem('ERRDATA_IN',zein)
    void = oCmd->AddItem('ERRID',oErr->GetFullIdentifier())
endif
```

Get the independent data, the dependent data and the errors.

Since the rebin operation is expensive, create a Command object and store the data as well as the full identifiers of the data objects. Note the use of the prefix '_IN'.

```
; Get treatment history
trmtType = 'DAVE1HISTORY'
oTrmt = (self.oSel)->GetByType(trmtType)
if (~obj_valid(oTrmt)) then begin
    ;;Try going (only) one level up to locate the treatmenet history
    ;; *** This may have to change if the data structure becomes more
    ;; complicated (eg for NeXus data) ***
    (self.oSel)->GetProperty, _parent=oSelParent
    if (obj_valid(oSelParent) && obj_hasmethod(oSelParent,'GetByType')) then $
       oTrmt = oSelParent->GetByType(trmtType)
endif
trmt = ''
if (obj_valid(oTrmt)) then begin
    void = oTrmt->GetData(trmt)
    void = oCmd->AddItem('TRMTID',oTrmt->GetFullIdentifier())
    void = oCmd->AddItem('TREATMENT_IN',trmt) ; treatment info
    ;; modify treatment info accordingly
    line = '=================================================='
    trmt = [trmt, $
            line, $
            'Timestamp: '+systime(), $
            'Perform a rebin of the data']
endif
```

If a treatment history data object can be found, then also save the treatment history in the Command object before modifying it.

```
; If 2D data and rebin axis is y-axis then transpoase data
transpose = 0
if ((self.axis eq 1) && ((size(zin))[0] eq 2)) then begin
    zin = transpose(zin)  ; transpose data so that y-axis data become the rows to be rebinned
    zein = transpose(zein)      ;
    transpose = 1
endif

nin = n_elements(vin)
lolim_mod = (lolim lt vin[0])? vin[0] : lolim
uplim_mod = (uplim gt vin[nin-1])? vin[nin-1] : uplim

;; determine output bins
nout = fix((uplim_mod - lolim_mod)/binw) + 1
vout = lolim_mod + findgen(nout)*binw
;; do the rebinning
if (~errExist) then zein = 0.01*zin ; fake some error if necessary!
case strupcase(strmid(distribution,0,4)) of ; switch between histogram or points data rebinning
    'HIST': drebin,vin,zin,zein,vout,zout,zeout,/hist,/to_hist,err=err,emsg=emsg
    else: drebin,vin,zin,zein,vout,zout,zeout,/points,/to_points,err=err,emsg=emsg
endcase
if (err ne 0) then begin
    oTool->StatusMessage,emsg
    return, 0
endif
```

The above lines of code rebin the data by calling on John Copley's drebin procedure.

```
;; if the data outside the rebinning range is not to be
;; truncated then this data needs to be included in the output
if (~truncate) then begin
    ;; get any data below lolim
    res = where(vin lt lolim_mod, cnt)
    if (cnt gt 0) then begin
        vout = [vin[res],vout]
        zout = [zin[res],zout]
        zeout = [zein[res],zeout]
    endif

    ;; get any data above uplim
    res = where(vin gt uplim_mod, cnt)
    if (cnt gt 0) then begin
        vout = [vout,vin[res]]
        zout = [zout,zin[res]]
        zeout = [zeout,zein[res]]
    endif
endif
```

The drebin procedure truncates the data to within the output bins specified by the vout variable. If the user selected the option *not to truncate* the results, then the above code snippet is used to include any unbinned data present below and above vout in the original data before rebinning was performed. First the unbinned data below vout is included followed by the data above vout.

```
; transpose the data back, if necessary
if (transpose) then begin
    zout = transpose(zout)
    zeout = transpose(zeout)
endif

; Store rebinned data because rebin operation is expensive
void = oCmd->AddItem('INDDATA_OUT',vout)
void = oCmd->AddItem('DEPDATA_OUT',zout)
if (errExist) then void = oCmd->AddItem('ERRDATA_OUT',zeout)
```

In the above code, if a transpose was performed earlier, then this needs to be reversed. Then, copies of the rebinned data are stored in the Command object.

```
; Update data objects with rebinned data using the no_notify keyword
; so that observers are not informed
void = oDep->setData(zout,/no_copy,/no_notify)
if (errExist) then void = oErr->SetData(zeout,/no_copy,/no_notify)
void = oInd->SetData(vout,/no_copy,/no_notify)

; Update treatment information
if (obj_valid(oTrmt)) then begin
    trmt = [trmt, $
            'From '+ $
            strtrim(string(lolim_mod),2)+' to '+strtrim(string(uplim_mod),2)+ $
            ' in steps of '+strtrim(string(binw),2)]

    void = oCmd->AddItem('TREATMENT_OUT',trmt) ; save modified treatment info
    void = oTrmt->SetData(trmt,/no_copy,/no_notify)
endif
```

Modify the data objects with the rebinned data. Take note of the /no_notify keyword to the SetData method – again, this is used to suppress any notification messages from being generated. Also save a copy of the final treatment history which now includes a reference to the rebin that has just been performed.

```
; Notify observers about the change!
oDep->notifyDataChange
oDep->notifyDataComplete
oInd->notifyDataChange
oInd->notifyDataComplete
oErr->notifyDataChange
oErr->notifyDataComplete

return, 1
end
```

Finally, notify data observers of the rebin changes.

The DAVEopDataRebin::UndoOperation method

This method is used to reverse the operation actions. When the Undo function is initiate from the application's menu or toolbar, this method is called.

```
function DAVEopDataRebin::UndoOperation, oCmdSet
compile_opt idl2

; Retrieve the command objects.
oCmds = oCmdSet->Get(/all,count=nCmds)
if (nCmds lt 1) then return, 0

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0

; Get various stored properties and restore the operation with them
void = oCmds[0]->GetItem('OLD_MIN',min)
void = oCmds[0]->GetItem('OLD_MAX',max)
void = oCmds[0]->GetItem('OLD_BIN',bin)
void = oCmds[0]->GetItem('OLD_AXIS',axis)
void = oCmds[0]->GetItem('OLD_TRUNCATE',truncate)
self->SetProperty, min=min,max=max,bin=bin,axis=axis,truncate=truncate, /no_update
```

First, retrieve the Command objects from the CommandSet that is passed to the method. Retrieve the bin parameters that were stored in the first Command object and reset the operation's properties to their original settings.

```
; Loop through rest of commands and undo the changes that were made
; to the targets
for i=1,nCmds-1 do begin
    void = oCmds[i]->GetItem('INDID',indID)
    oInd = oTool->GetByIdentifier(indID)
    if (~obj_valid(oInd)) then begin
        oTool->_RemoveCommand, oCmdSet ; remove and destroy undo/redo buffer from tool
        return, 0
    endif
    void = oCmds[i]->GetItem('INDDATA_IN',vin)

    void = oCmds[i]->GetItem('DEPID',depID)
    oDep = oTool->GetByIdentifier(depID)
    if (~obj_valid(oDep)) then begin
        oTool->_RemoveCommand, oCmdSet
        return, 0
    endif
    void = oCmds[i]->GetItem('DEPDATA_IN',zin)

    void = oDep->SetData(zin,/no_copy,/no_notify)
    void = oInd->SetData(vin,/no_copy,/no_notify)

    void = oCmds[i]->GetItem('ERREXIST',errExist)
    if (errExist) then begin
        void = oCmds[i]->GetItem('ERRDATA_IN',zein)
        void = oCmds[i]->GetItem('ERRID',errID)
        oErr = oTool->GetByIdentifier(errID)
        if (~obj_valid(oErr)) then begin
            oTool->_RemoveCommand, oCmdSet
            return, 0
        endif
        void = oErr->SetData(zein,/no_copy,/no_notify)
    endif

    void = oCmds[i]->GetItem('TRMTID',trmtID)
    if (n_elements(trmtID) gt 0) then oTrmt = oTool->GetByIdentifier(trmtID)
    if (obj_valid(oTrmt)) then begin
        void = oCmds[i]->GetItem('TREATMENT_IN',oldTrmt)
        void = oTrmt->SetData(oldTrmt,/no_copy,/no_notify)
    endif

    ;; Notify observers about the change!
    oDep->notifyDataChange
    oDep->notifyDataComplete
    oInd->notifyDataChange
    oInd->notifyDataComplete
    oErr->notifyDataChange
    oErr->notifyDataComplete
endfor

return, 1

end
```

Loop through the remaining commands (there should actually only be one more since we created two Command objects only during the initial rebin process) and retrieve the original data, object identifiers and treatment history and reset everything back to their original values like they were before the rebin was performed. Thus the rebin action is undone.

The DAVEopDataRebin::RedoOperation.

The RedoOperation is triggered when the Redo function is activated. The code is listed below and it is similar to the UndoOperation except that the operation settings and data are reset to the values after the rebin was last performed. This way, the rebin action is redone.

```
function DAVEopDataRebin::RedoOperation, oCmdSet
compile_opt idl2

; Retrieve the command objects
oCmds = oCmdSet->Get(/all,count=nCmds)
if (nCmds lt 1) then return, 0

; Get the tool
oTool = self->GetTool()
if (~obj_valid(oTool)) then return, 0

; Get various stored properties and restore the operation with them
void = oCmds[0]->GetItem('NEW_MIN',min)
void = oCmds[0]->GetItem('NEW_MAX',max)
void = oCmds[0]->GetItem('NEW_BIN',bin)
void = oCmds[0]->GetItem('NEW_AXIS',axis)
void = oCmds[0]->GetItem('NEW_TRUNCATE',truncate)
self->SetProperty, min=min,max=max,bin=bin,axis=axis,truncate=truncate, /no_update

; Loop through rest of command(s) and redo the changes that were made
; to the targets
for i=1,nCmds-1 do begin
    void = oCmds[i]->GetItem('INDID',indID)
    oInd = oTool->GetByIdentifier(indID)
    if (~obj_valid(oInd)) then begin
        oTool->_RemoveCommand, oCmdSet ; remove and destroy undo/redo buffer from tool
        return, 0
    endif
    void = oCmds[i]->GetItem('INDDATA_OUT',vout)

    void = oCmds[i]->GetItem('DEPID',depID)
    oDep = oTool->GetByIdentifier(depID)
    if (~obj_valid(oDep)) then begin
        oTool->_RemoveCommand, oCmdSet
        return, 0
    endif
    void = oCmds[i]->GetItem('DEPDATA_OUT',zout)

    void = oDep->SetData(zout,/no_copy,/no_notify)
    void = oInd->SetData(vout,/no_copy,/no_notify)

    void = oCmds[i]->GetItem('ERREXIST',errExist)
    if (errExist) then begin
        void = oCmds[i]->GetItem('ERRDATA_OUT',zeout)
        void = oCmds[i]->GetItem('ERRID',errID)
        oErr = oTool->GetByIdentifier(errID)
        if (~obj_valid(oErr)) then begin
            oTool->_RemoveCommand, oCmdSet
            return, 0
        endif
        void = oErr->SetData(zeout,/no_copy,/no_notify)
    endif

    void = oCmds[i]->GetItem('TRMTID',trmtID)
    if (n_elements(trmtID) gt 0) then oTrmt = oTool->GetByIdentifier(trmtID)
    if (obj_valid(oTrmt)) then begin
        void = oCmds[i]->GetItem('TREATMENT_OUT',oldTrmt)
        void = oTrmt->SetData(oldTrmt,/no_copy,/no_notify)
    endif

    ;; Notify observers about the change!
    oDep->notifyDataChange
    oDep->notifyDataComplete
    oInd->notifyDataChange
    oInd->notifyDataComplete
    oErr->notifyDataChange
    oErr->notifyDataComplete
endfor

return, 1

end
```

Attaching an existing DAVE 1.x data reduction module

A scheme has been developed to launch existing data reduction module that work with DAVE 1.x. The scheme has two components: an operation and a UI Service. The operation is generic and the same one should work for all data reduction modules. The existing application is wrapped into a UI Service. The UI Service part is dependent on the application to be launched. An example is presented here for the FANS data reduction module.

The DAVEopLaunchApp operation

The operation is used to create menu entries that can launch a conventional data reduction module. This operation implements the following methods:

DoAction

It does not support the undo/redo functionality and as such the UndoOperation and RedoOperation methods are not implemented. The DoAction method is shown below.

```
function DAVEopLaunchApp::DoAction, oTool
compile_opt idl2

;oSystem = oTool->_GetSystem()
self->GetProperty, IDENTIFIER=id
name = STRMID(id, STRPOS(id, '/', /REVERSE_SEARCH) + 1)

;call_procedure,name
void = oTool->DoUIService(name,self)

return, obj_new()

end
```

An id is extracted from the operation using the GetProperty method. This id is used to determine the name of the UI Service that has been written for the module. The UI Service component is then launched by calling the DoUIService method of the tool. The UI Service will then activate the data reduction module as explained later.

Note that the return value of the DoAction method is a null object (instead of a CommandSet object that is normally returned). The reason for this is because we don't wish to be able to undo the application launch!

Note: When the DoAction method returns a null object, the application tool does not create an undo/redo command buffer and consequently the operation's actions cannot be undone or redone.

To create a menu entry for a data reduction module, you must include an entry like this one:

```
self->RegisterOperation,'FANS...','DAVEopLaunchApp',identifier='DataRed/NCNR/FANS Data Reduction'
```

in the DAVETool::MenuLayout method (implemented in davetool__menulayout.pro). The above statement registers a component called 'FANS...' using the identifier indicated. Thus a menu item 'FANS...' will be created under the 'NCNR' submenu within the 'Data Reduction' menu. The specified identifier, the part after the last /, will be used later in the UI Service registration.

All modules should use a similar entry – the first argument and identifier keyword should be different, of course, and reflect the name of the data reduction module. For example, for the MARS instrument at the PSI, the entry could be like this:

```
self->RegisterOperation,'MARS...','DAVEopLaunchApp',identifier='DataRed/PSI/MARS Data Reduction'
```

The UI Service launcher

The module-dependent user interface service (UI Service) forms the second part of the launch scheme.

The example UI Service code was created for the FANS data reduction module:

```
function ui_LaunchFansReduction, oUI, oRequester
compile_opt idl2

; Get group leader from main UI object
oUI->GetProperty, group_leader=group_leader

; Get requesting Tool object from main UI object
oDAVETool = oUI->GetTool()

; Retrrieve the current data and working directories from requesting tool
oDAVETool->GetProperty, data_directory=dataDir, working_directory=workDir

; Start FANS data reduction application. This app is a direct port
; from the DAVE 1.x version.
FANS_DatReduc, group_leader=group_leader, workDir=workDir, dataDir=dataDir, DAVETool=oDAVETool

return, 1
end
```

As you can see, it is very short. A few observations:

- A UI Service is a function that accepts two *input* arguments

  The first is an object reference to an IDLitUI object and the second is a reference to the component that activated the UI Service.
- Retrieve a group_leader widget identifier and the tool object from the UI object.
- Retrieve the globally set working directory and data directory from the tool object.
- Then launch your DAVE 1.x data reduction application.
- FANS_DatReduc is almost unchanged except there is no davePtr argument. The group_leader, data_directory and working_directory keywords have the identical meaning as in DAVE 1.x
- There is a new keyword – DAVETool – which is set to the tool object. This object reference will be used within FANS_DatReduc to connect back to DAVE to pass datasets (that are packaged in a davePtr) into the application's Data Manager.

The UI Service must be registered before it can be used and this should be done in davesetup.pro as follows:

```
oSystem->RegisterUIService, 'FANS Data Reduction','ui_launchFANSReduction'
```

where the first argument is the UI service component name and the second is the function name that implements the code shown above.

Note: The UI Service component name 'FANS Data Reduction' must match the last part of the identifier string used earlier in the RegisterOperation statement that created the menu entry.

When the 'FANS...' menu is clicked, the DAVEopLaunchApp::DoAction method is called. The identifier string is obtained and the name variable is set to 'FANS Data Reduction' which is the name of the UI Service component registered above. Hence the DoUIService method is then able to locate and execute it.

Required changes to the existing data reduction module

There are six, mostly minor, changes that would be required before the data reduction module can be launched in DAVE.

1. The arguments to the module's main function or procedure must be altered. Currently, the FANS data reduction module, a typical DAVE 1.x data reduction module, has the following signature:

   ```
   pro FANS_DatReduc, davePtr,notifyIds=notifyIds,group_leader=group_leader
   ```

   where the davePtr keyword is a pointer to a davePtr dataset, notifyIds keyword specifies an array of widget identifiers from the main application and the group_leader keyword specifies a group_leader widget identifier to be used by FANS_DatReduc.

   In the current version of DAVE, only the group_leader keyword should be retained. notifyIds is no longer used since widget event messages will not be passed back to the caller.

   The new signature should now look like this:

   ```
   pro FANS_DatReduc, group_leader=group_leader, workDir=workDir, dataDir=dataDir, $
                      DAVETool=oDAVETool
   ```

   where davePtr and notifyIds have been replaced by three additional keywords. workDir and dataDir are input keywords that should contain the globally set working and data directories respectively. DAVETool is also an input keyword that is set to the object reference of the DAVE tool object. The values for these keywords, in addition to the group_leader keyword, are all set by the calling program (ui_LaunchFansReduction in this case).
2. Use the dataDir and workDir values supplied in the usual way – in DAVE 1.x, these would normally be obtain from the variables (*!dave_defaults).datdir and (*!dave_defaults).workdir that are maintained globally in DAVE. Some changes may be required in your code to adjust for the way in which the work and data directories are defined.
3. If your data reduction module sends events back to DAVE using the notifyIds widget identifiers, this should be eliminated – in DAVE 1.x this was normally used to notify DAVE that a data reduction module has been killed so that DAVE can be sensitized, if it was desensitized when the reduction module was launched.
4. Add a menu entry in your reduction module to export data to the Data Manager in DAVE. For FANS_DatReduc the entry looks like this:

   ```
   wID = widget_button(fileMenu,value='Export to DAVE Data Manager',uname='sdavetool')
   ```

   where the uname will be used to identify events from this menu.
5. Store the DAVETool object reference in your module's state structure so it can be retrieved in the event handler for the menu item added.
6. The event handler for the menu item should contain code similar to the following:

```
'sdavetool': begin

   ; Update the davePtr with most current output
   fans_2Dave, sPtr, err=err, emsg=emsg, nameTag=nameTag, /no_output
   if (err ne 0) then return

   if ((n_elements(nameTag) le 0) || (strtrim(nameTag,2) eq '')) then nameTag='Untitled'

   ; Send data to DAVE Data Manager Folder
   ((*sPtr).DAVETool)->AddDavePtrToDataManager, (*sPtr).davePtr, nameTag
  break
 end
```

In this event handler, the davePtr is updated with the most current output in the data reduction program by calling on the fans_2Dave procedure. Obviously this is very specific to the FANS Data reduction module.

The basic expectation here is that your data reduction module supports storing data using a davePtr. So in whatever way it is done, retrieve the most up to date version of the davePtr to be exported.

It would be nice to have the data appear in the Data Manager with a sensible name – it is up to you to specify one using the nametag variable shown here. In FANS, this would typically be set to the file name from which the raw data was read from!

Finally, call the DAVETool::AddDavePtrToDataManager method as shown to export the davePtr into the Data Manager.

These are the basic steps required to adapt your data ruction module to work in DAVE 2.x. When this is done your module will be launched from DAVE and you will be able to export reduced data that is packaged in a davePtr into the DAVE's Data Manager folder. Once in the Data Manager folder, the user is able to make use of any available build-in functionality.