

Beginner's Manual

**By
Ajith.R**

**with the help of
Suneido Team**

A Beginner's Manual

[Introduction](#)

[Chapter 1](#) - Define the tables

[Chapter 2](#) - Data dictionary entries

[Chapter 3](#) - More Data dictionary entries

[Chapter 4](#) - Rules

[Chapter 5](#) - The Data Entry Screen

[Chapter 6](#) - Print

[Chapter 7](#) - The details table dataentry

[Chapter 8](#) - Formats used in Reports

[Chapter 9](#) - Running standalone

[Chapter 10](#) - Running Suneido over a network

[Chapter 11](#) - Maintenance of Suneido

[Chapter 12](#) - Improve your code

[The Program](#)

[Epilogue](#)

A Beginner's Manual

You may be thinking that I made a mistake with the apostrophe in *A Beginner's Manual* - No, I myself am the beginner I've referred to. I am a beginner not only in Suneido, but also in computer programming. I happened to read about Suneido in a computer magazine and because I wanted to try computer programming, I decided to give Suneido a trial. With the help of the Suneido team, now I am able to use Suneido, though I am still a beginner. Suneido has only started its journey and so there is still a lot to improve. One such area where I thought I could help is a section for beginners to guide them through a small project.

So, here I present my first project (unfinished when this was written). I am a medical professional and so I chose something related to medicine. My project was intended to make a small program to keep track of the HIV infected employees in my organization. I do not think the program as such will be useful anywhere other than for the organization it is made for. But the steps in making it will, I think, help all beginners to get started in Suneido. It is a beginners' manual as well as beginner's manual. Do allow tolerance for mistakes if any that I may have made. I apologize for my English as well.

Requirements

I suppose that-

You have basic computer knowledge - moving about, using a mouse etc

You have Suneido's latest version installed in your computer (I was using the 2003 April release)

You have read the Users Manual chapters Introduction, Getting Started, Tools. If you read Going Further and Language chapters as well, it will be more advantageous.

Conventions followed in the manual

The black colored letters represents the text, bold indicating the headings. The Suneido code and words that are of special significance in Suneido are represented by colored letters. **Blue** color denotes words that are typed as they appear in this manual. **Green** indicates that you can substitute another word for that. Bold letters (whether **blue** or **green**) represent that such piece of code cannot be left out in that particular context. I had used * to mark areas where I was not sure of myself. They should be removed by the time this manual is ready for use. If you find one do let me know (through the Suneido forum).

Possible errors

My disclaimer is that I am a beginner and this manual cannot be without errors, though nothing is intentional. You use this manual at your own risk and all responsibility is yours.

My project was being changed all the time this was being written and so there might be some discrepancy between the library provided and the manual. I will try to eliminate them as much as possible, but some might escape my attention.

There will be something extra in the library not explained in the manual. Probably, they are to be removed or is beyond the scope of this manual.

I wanted the reference section to be true links, but that may take time. As I did not have enough time to sort out the Suneido forum topics, I may have left out some topics.

Acknowledgments

To the Suneido team for providing such a good piece of work.

To the entire Suneido forum members for their support.

To Mr B Vijayan for going through the manual and correcting the grammatical errors.

To Julien Marmin, Jean-Luc Chervais, Louis Marmin-Cormary and Philippe Gouillou for their encouraging emails, for pointing out the errors in the manual and for their suggestions.

I think I should dedicate this to Mr Jeff Ferguson for having answered more than $\frac{3}{4}$ of my silly questions in the forum with great patience and thereby making me confident to contribute this.

Special Thanks to Mr Andrew McKinlay for taking up the job of educating me from Jeff and for all the support extended in preparing this book.

License

This is not copyrighted, but written by me Ajith.R and I am submitting this work to Suneido team for any use they think this is fit for.

Chapter 1 - Define the tables

Suneido is designed to work with tables and so obviously the first step is to make a table.

How to do that?

I can suggest 3 ways:

- You can open the QueryView window and type the command

```
create tablename (column1, column2) key (column1) index (column2)
```

and execute the command in the QueryView window itself by pressing F9 .

- You can make a new item after opening the LibraryView window, give it a name you wish (**Nameofitem**) and write code as given below

```
function()
{
  Database ("create tablename (column1
            , column2) key (column1) index (column2)")
}
```

and execute it from the Workspace by calling the item in LibraryView as **Nameofitem()** or from the LibraryView itself by pressing F9 or using Run Command.

- The third option is similar to the second except that instead of **create** you use **ensure**. So the code will look like

```
function()
{
  Database ("ensure tablename (column1
            , column2) key (column1) index (column2)")
}
```

Both the **Database** commands can be executed from Workspace as well, without the **function()** and the enclosing braces.

What exactly does that mean?

All the 3 options listed above creates a table with the table name you have assigned (**tablename**) with the column names as specified (**column1** and **column2**) .The column names can start with a capital letter in which case Suneido will take it as column to have calculated values and will try to get its value calculated using a **Rule** of its name. The calculated columns are not stored in the database, but are calculated every time it is accessed. Even when a column name starts with a capital letter, it is referred to without the caps in all other places. The **key** that you specify should be one of the columns specified earlier and denotes that column which should not have a duplicate entry. The **index** specifies other columns which should be indexed but can have duplicate entries. Indices may speed up access to data. They're not essential to any queries. If you specify an **index** on a column, it cannot be the one that is specified as the **key**. The Users Manual says that you can specify multiple keys - probably by repeating **key(column)** with the names of the different columns. It also says that we can specify multiple columns as a single **key** - by specifying the different column names

inside one `key()` separated by commas. The same holds true for `index` as well.

What more?

The commands can accept one option along with `index` as given below:

```
index (columnname) in mastertablename (masterkeycolumn) cascade update
```

This particular kind of indexing is done in a details table (the many in a 1 to many relation) to specify that its master table is the one specified as `mastertablename` and that the column specified by `columnname` is the one in the details table that will have values similar to those entered in the `masterkeycolumn` of `mastertable`. The `masterkeycolumn` should be a `key` column. If the names of both `columnname` of details table and `masterkeycolumn` of `master table` are same, then you need not specify the `(masterkeycolumn)`. The `cascade` option is needed if you want to make sure that as a record of master table is removed or updated corresponding changes should take place in the details table. If you want only updating, but no deletes to be applicable, then specify `updates` as well, following the word `cascade`

What difference is there between the options ?

Understand option 1 (`create`) as the basic option. It allows all that were discussed. The minimum requirements should be specifying a table name, at least one column and one `key` column. You have to run it from the QueryView. The next time you run it , it shows an error message that such a table already exists.

The `Database` options are written in the LibraryView and can be run from WorkSpace. Your code will be there if you dump your library and load it elsewhere. If the first option of queryview is used, then you will not have this advantage - you have to reenter the code.

The `ensure` is different, smarter than `create`. Though I have specified it as an argument of `Database` function only, you can take out the `Database` and outer braces and quotes and enter the rest in QueryView as with `create` and execute it there as well. It will behave like `create` if there is no table with the specified name. The advantage lies with a second run - if you run it with more options say a new `key` or a new `columnname` etc it won't return an error message, but will update the table. One important thing - it won't remove anything, only adds if needed. One more thing, if you use only `create`, then you have to `destroy` your table and `create` one fresh if you want to make changes or use the `alter` command. With `ensure` no data entered is lost while updating.

What did I use?

I chose to use in LibraryView, under the name

`Ensure_hivmain`

```
function()
{
  Database("ensure hiv_main (firstname
           ,middlename,lastname,Fullname,sexes,dobirth,Age
           ,emp_dep,emp_name,persnum,tick_num,factory,address
           ,phone,hivnumber,dodiagnosis,do_death,causedeath,riskfactors)
  index (firstname)
  index (lastname)
  index (persnum)
  index (tick_num)
```

```

        key (hivnumber) ")
    }

```

,under the name

Ensure_hivinvestigations

```

function()
{
    Database("ensure hiv_investigations (inv_recordnumber
            ,hivnumber,hivinvestigations,inv_date,inv_result)
            index (hivnumber) in hiv_main (hivnumber) cascade
            key (inv_recordnumber) ")
}

```

,under the name

Ensure_hivadmissions

```

function()
{
    Database("ensure hiv_admission (andnumber
            ,hivnumber,do_admission,do_discharge,hivdiagnosis)
            index (hivnumber) in hiv_main (hivnumber) cascade
            key (andnumber) ")
}

```

& under the name

Ensure_hivcontacts

```

function()
{
    Database("ensure hiv_contacts (contactrecordnumber
            ,contactee,contactnumber,typecontact,elisadate,elisaresult
            ,vdrldate,vdrlresult,hepbdate,hepbresult)
            index (contactee) in hiv_main (hivnumber) cascade
            key (contactrecordnumber) ")
}

```

The aim was to create 4 tables. The *hivmain* was made to store details of HIV patients that will have only one instance of the information at a time. By this, I mean information like name, address etc that need not be reentered a second time. That they may have to be changed should be understood. Among others, patient identification data, columns to store few (minimal, in fact) clinical data and a unique clinic number (*hivnumber*) are also created using this command in the table.

The second table (*hivinvestigations*) was designed as a details table which carries many rows corresponding to each row of *hivmain*. It was designed to store multiple investigation results of all patients

The third table (*hiv_admission*) was again a details table designed to store the different hospital admission dates and some details of admissions.

The fourth table (*hivcontacts*) was designed to store information relating to the contact details of the patient. Again this is a details table that can have any number of rows with information related to one *hivnumber*.

Related commands

The other database commands like

- [alter](#)
- [view](#)
- [destroy](#)
- [rename](#)

and the corresponding [Database](#) variations are worth mentioning. Read about them in the Users Manual.

Reference

Users manual -> Requests -> create

Users manual -> Requests -> ensure

Users manual -> Getting Started -> Create a Table

Users manual -> Going further -> Master-Detail Relationship

Users manual -> Database -> Foreign Keys

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=723

Chapter 2 - Data dictionary entries

You might have noticed that we gave only the names of the columns in each table. So how does the program understand that each column has to store a particular kind of data? Suneido may be unique here - it need not know what kind of data has to be stored in each column when they are declared. It just allows any kind of data to go inside any column and even different kind of data to different cells of a column.

No, it does not create any problem. (In fact I think Suneido has done it the better way).You can however restrict what is passed on to be stored (but not what is stored) by defining Data dictionary entries.

How to do that?

You have to do it from LibraryView by making new items, the data dictionary entries. The items must be named in a standard way - their name should begin with **Field_** followed by the columnname for which it should apply. For example, if you want to make a data dictionary entry to limit data entered in a column named *mydata*, the data dictionary entry should be given the name **Field_mydata**. (Note that it should not have any space in between). Now the data dictionary entry will be used for all columns with the name *mydata* in any table (currently present or newly made as long as the library is being used) and it is done automatically by the program. No need for you to link the data dictionary entry to the column of a particular table.

What more?

What should be entered in each data dictionary item? You can opt to enter any one of the previously defined data dictionary entries as a parent entry from which your data dictionary entry inherits and then supply the parameters as per the parent data dictionary entry. The parameters that can be supplied to different data dictionary entries as I know are -

Prompt

Control

Format

Heading

The code is written as

```
Field_name
{
  Prompt: 'String'
  Control: (ControlName_and_parameters)
  Format: (FormatName_and_parameters)
  Heading: 'String'
}
```

The **Field_name** denotes the parent data dictionary entry from which your present data dictionary inherits and rest of the code is the parameters or options that you specify. The **Prompt** is to be supplied as a string and should always be enclosed in quotes. You can use either double or single quotes. What you enter inside the quotes will be used as column

heading in different data entry screens and prints. If you do not supply any **Prompt**, then the column name as specified while creating the table, will be used. The **Control** specifies what widget should be used to enter the data in data entry screens. This means, whether you want a list to pop up to select from, a checkbox to enter your option... The name of the control and its parameters are to be entered inside parentheses. There are many controls and the list is increasing. I will explain the controls I have used in the pages to come. Likewise **Format** options dictate the way the value you entered (through the **Control** options) should be displayed. Its options should be specified inside parentheses. Here also you have different options based on the data type expected which will be discussed later. The **Heading** is again a string and always put inside quotes. If supplied and different from **Prompt**, the **Prompt** is used in screens and **Heading** will be used in prints. If not supplied, **Heading** will default to **Prompt** .

What different data dictionary entries do we have?

You take a look in the Datadict folder of the standard library (stdlib) and you can see some 40 items with their name beginning with **Field_** and another four in the subfolder Base and many more in the System subfolder. Other libraries may have additional data dictionary entries.

What did I use?

I mostly used the Base data dictionary entries to build my own entries. You can find my data dictionary entries in *Field* subfolder of *hiv_lib*.

The data dictionary entry for the field *firstname* of table *hiv_main* was given the name **Field_firstname** and the following code was entered

```
Field_string
{
  Prompt: 'First Name'
  Control: (Field width: 20 mandatory: true)
}
```

Field_string means that the data is going to be strings. If the rest of the code is not entered, the defaults as per **Field_string** data dictionary entry will be used.

By specifying **Prompt** we are specifying Suneido to display the column as *First Name* in all data entry screens.

Control specifies how the data entry field should be. All parameters to **Control** should be given as an object within parentheses. The first member should be the name of the control and the rest will be arguments for the control. In our example we use **FieldControl** (written without **Control**) which will provide a single line rectangular field to enter data. It will take all types of characters and if the characters fall out of the visible area, you can use navigation keys to browse the beginning and end. It supports cut, copy and paste functions. The parameters accepted by **FieldControl** are as follows, with their default value -

width : The value of **width** should be a number and the default is 20. This determines how many characters wide should be the rectangular area. It has nothing to do with the number of characters accepted into the field.

status : It should be a string specified inside quotes and determines what should appear in the status bar when the field is accessed. By default nothing is displayed.

readonly : It is specified as **false** or **true** without quotes to specify whether the field should only be displayed or editing should be allowed. By default, editing is allowed, i.e. the value is **false**.

If `true`, the field is grayed out.

`font` : The entry made inside quotes should be a valid font name, a list of which I cannot provide. The default font is the system GUI font. The `font` parameter should be specified if you want to set the next 3 parameters.

`size` : It should be a number. It will not work if you do not specify the `font` option. Again the default value is that of system font.

`weight` : It can be a number from 0 to 1000. Note that 400 is normal while 600 is bold.

`underline` : It can be `true` or `false` without quotes and determines whether the text has to be underlined or not. The default is `false`.

`password` : This again is specified as `true` or `false`. It determines whether the typed characters should be displayed as they are or as asterisks (*) as in passwords. Needless to say that the default value is `false`.

`justify` : It should be a string in all caps to specify the alignment. So it can be `LEFT` or `RIGHT` or `CENTER` (not `JUSTIFY`) all inside quotes. The default is `LEFT` .

`style` : It is specified as a number. Users manual says that "The window style can be passed by specifying the style in the `style` parameter." But I do not fully understand it.

`set` : The initial value that is to be set is specified here. If the value is string specify it inside quotes. By default nothing is specified. I could not get this working. Later I could understand that `set` option will not work inside `RecordControl` which is used by `AccessControl` and `ExplorerListViewControl`.

`mandatory` : This can be either `true` or `false`. It determines whether the field can be left empty or not. This parameter can be specified since `FieldControl` inherits from `EditControl`. The default is `false`.

This will help you understand the basic data dictionary entry. If you do not specify a data dictionary entry for any column, the default of `Field_string` is used and default control for string, `FieldControl` is used with its default values with the default `Format` of `Text`.

I will continue with the different options on data dictionary entries in the next chapter.

Note:

The better way to make sure that the things that you make do not conflict with anything else is to add a unique 2 or 3 letters to all items of one project. I have not done this in my example because, I learned this only half way through my project. The `address` and `phone` data dictionary entries will be conflicting with those in the `stdlib`.

Chapter 3 - More Data dictionary entries

The types of data you can enter are string, number, date and boolean. Objects come from elsewhere. So the data dictionary entry should inherit from either [Field_string](#) or [Field_number](#) or [Field_date](#) or [Field_boolean](#) or any other data dictionary entries that inherit from them either directly or indirectly.

Some of the different controls you can use for [Field_string](#) are : [FieldControl](#), [ChooseListControl](#), [ComboBoxControl](#), [ChooseTwoListControl](#), [ChooseManyControl](#), [EditorControl](#), [FieldHistoryControl](#), [PatternControl](#) and [RadioButtonsControl](#)

For [Field_number](#), you can have [NumberControl](#) and [SpinnerControl](#).

[Field_date](#) can have [ChooseYearMonthControl](#), [ChooseMonthDayControl](#), [DateTimeControl](#), [ChooseDateControl](#) as its [Control](#) options.

[Field_boolean](#) can have [CheckBoxControl](#).

This is not an exhaustive list, but only representative. There are many composite controls that make use of the different controls already named.

I will attempt to take examples from my data dictionary entries to explain a few of the controls. I will explain [Formats](#) as and when they are encountered.

A Few Examples

In [hiv_lib](#) you can find [Field_firstname](#), [Field_middlename](#), [Field_lastname](#), [Field_fullname](#), [Field_contactee](#), [Field_causeddeath](#), [Field_address](#), [Field_contactnumber](#) and [Field_tick_num](#) all of which are defined using [Field_string](#) and [Field](#) as control . The difference in definition of each data dictionary can be understood easily if you refer to the previous discussion. They won't be discussed further.

ChooseListControl

The data dictionary entries [Field_emp_dep](#), [Field_factory](#), [Field_vdrresult](#), [Field_hepbresult](#), [Field_elisaresult](#), [Field_typecontact](#), [Field_sexes](#), [Field_hivcategory](#), [Field_hivinvestigations](#) and [Field_inv_result](#) are all defined using [Field_string](#) and [ChooseListControl](#) .

The [ChooseListControl](#) makes a rectangular field with an arrow button at its right end, pressing which pops up a predefined list. As mentioned earlier the control name is specified inside parentheses without the [Control](#) in its name. The [ChooseList](#) takes the following arguments:

list : The **list** should be an object so the items of list are given inside parentheses separated by commas and a hash{#} is put before the opening parentheses. If you have both number and text , the text has to be specified inside quotes. If you have only text even if you do not put each item inside quotes there is no trouble([Field_hivcategory](#)). If you want the list to show a particular thing (say, a number) in the list and the value to be stored is a different one (a string for instance), the Users Manual gives an example. You have to specify the value to be displayed first followed by the what is to be stored both inside a common quotes and separated by a hyphen (-). I do not have an example for this. You should understand that, though you can specify numbers in the list supplied to [ChooseListControl](#), they are converted to text and used.

listField : is used if you want the list be derived from another field. The field need not be existent, but can be 'virtual', i.e. derived from a [Rule_](#). For the time being, just understand that

Rules are items in library that are named as `Rule_name_of_field` and they are functions and return some value. The Rules you want to use in a `listField` option should return a comma separated string. The `listField` option is given as the name of the field (or Rule without the `Rule_`) and put inside quotes. In my examples, `Field_typecontact`, `Field_hivcategory`, `Field_emp_dep`, `Field_factory`, `Field_sexes`, `Field_vdrlresult`, `Field_hepbresult`, `Field_elisaresult` and `Field_inv_result` are using the `list` and the rest are using `listField` options. In case of those using `listField`, there will be an item in the library in the subfolder Rules named `Rule_name` where `name` will correspond to that in the `listField` option. The general format of the code inside the Rule will be

```
function ()
{
  return "a,b,c,d"
}
```

`allowOther` : This can be either `true` or `false` and the default is `false`. The advantage by specifying `true` for this argument is that we can type values other than in the list and the advantage in keeping it as `false` is that the field will auto select a member of the list as you type in the first unique letter(s).

`selectFirst` : It can also be `true` or `false` depending on whether you want the first item in the list to be selected as default. The default value is `false` and the first item is not selected. I could not find this option working, though it is specified. The probable reason is the same as for `set` of `FieldControl`.

`set` : This is also used to set a default value. You have to specify the value you want as default. Depending on whether the value is string or number you may or may not specify quotes.

Though the inheritance is not stated, the Users Manual does state that `mandatory` and `width` (with a default of `10`) can also be specified as discussed for `FieldControl`

Related Controls

Some of the other controls similar to the above are `ChooseManyControl`, `ComboBoxControl`, `ChooseTwoListControl`, `FieldHistoryControl`, `TwoListDlgControl` and `ListBoxControl`. The `RadioButtonsControl` can be substituted when the number of items is less. Find their usage from the Users Manual.

ChooseManyControl

The data dictionary entries `Field_hivdiagnosis` and `Field_riskfactor` are defined using `Field_string` and `ChooseManyControl` which is very similar to `ChooseListControl` described earlier.

The `ChooseMany` also makes a rectangular field but with an ellipsis button at its right end, pressing which pops up a predefined list. The list is different from that of `ChooseList` in that each item of the list has a check box beside it allowing you to select more than one item. You have All and None Buttons to select all or none from the list. Once you have made your choice, you can press the OK button or if you do not want to make the selection, press the Cancel button. The data returned by `ChooseMany` is a string always with commas separating the different selected options. The `ChooseMany` takes the following arguments:

- 1) The `list` or a `listField` is specified exactly as for `ChooseList`
- 2) `saveNone` : can be `true` or `false` and the default is `false`. The `true` condition specifies the

[ChooseMany](#) to return "None" if none of the values is selected.

The Users Manual says that [ChooseMany](#) derives from [ChooseFieldControl](#), but the parameters that can be specified that way is not known to me.

Related Controls

Have already been listed along with [ChooseList](#).

ChooseDateControl

The data dictionary entries [Field_do_admission](#), [Field_do_diagnosis](#), [Field_do_discharge](#), [Field_hepdata](#), [Field_elisadate](#), [Field_do_death](#), [Field_vdrldate](#) and [Field_inv_date](#) are all defined using [Field_date](#) and [ChooseDateControl](#) .

The [ChooseDate](#) makes a rectangular field with an ellipsis button at its right end, pressing which brings up a calendar. As mentioned earlier the control name is specified inside parentheses without the Control in its name. The calendar has areas for dates, months and years. By clicking on the arrow buttons by the side of the months area, the date is incremented by one month. You can click on the required day and the date corresponding to the day clicked will be taken as the value of the field. In case, you want a year that is long back, click over the year and you will get a spinner to increment the years fast. Clicking over the month displayed will bring you a list of months from which you can select easily. The [ChooseDate](#) starts with the current system date. In case you want the calendar to start from a particular date, set it as the default value. Whether [ChooseDate](#) accepts any parameters is not stated explicitly in the Users Manual. But it is said that it inherits from [ChooseField](#) and hence should be able to accept [width](#) . It is also said that [ChooseField](#) accepts a [field](#) option. But, what is the significance of [field](#) is not known to me.

For any data dictionary defined using [Field_date](#), the default control is [ChooseDate](#). If you don't want the [ChooseDateControl](#), but some other control to restrict data entry of that field, then you have to specify that control and should not choose to leave the [Control](#) option empty. You can see that being used in defining [Field_contactrecordnumber](#) and [Field_inv_recordnumber](#) where I wanted a plain rectangular field and hence [FieldControl](#) was used.

Now I will tell you something about [Format](#) parameter of data dictionary entries. I used them mostly when using [Field_date](#). A date is actually date plus time. If one of the halves is not specified, it is assumed. For example, if no time is specified 00.00.00 is assumed in place of time. If you want a date to be displayed in a particular format, then you have to specify them. The valid options are [ShortDate](#), [LongDate](#), [DateTime](#), [Text](#) ... (there will be more format options even for date and definitely for other data types that I am not aware of). Specifying a format does not affect the type of data as such, but only affects the way it is presented.

For example, [Field_contactrecordnumber](#) and [Field_inv_recordnumber](#) are dates. But I wanted them to be shown as a unique identifying number. So, I opted for the [TextFormat](#) as the [Format](#) member of their data dictionary entry. In all other data dictionary entries where the data type is a date and I wanted them to be displayed as per my windows setting short date format, I opted for the [ShortDateFormat](#) or left it blank as [ShortDate](#) is the default [Format](#).

[Format](#) is more significant in relation to printouts where I have taken up the topic in a better way.

Related Control

Few of the other controls similar to the above are [ChooseDateTimeControl](#), [ChooseMonthDayControl](#) & [ChooseYearMonthControl](#).

PatternControl

The data dictionary entries [Field_hivnumber](#), [Field_phone](#), [Field_persnum](#) and [Field_andnumber](#) are all defined using [Field_string](#) and [PatternControl](#) .

The [PatternControl](#) makes a rectangular field without any buttons and it restricts data entry based on a pattern specified. The pattern characters can be string or numbers. It can also have special characters. The pattern control inherits from [FieldControl](#) and hence should handle all parameters specified along with the [FieldControl](#). In addition it can have a [pattern](#) specified. The [pattern](#) is specified as string within quotes. The different patterns are separated using the pipe character (|). The [pattern](#) has a variable component as well as constant characters. The variables are specified below-

A - stands for any capital letter

a - stands for any small letter

- stands for any digit

< - stands for a digit or a letter that will be converted to lower case

> - stands for a digit or a letter that will be converted to upper case

The constants are specified as they should be.

^ - the character following **^** is taken as that character itself. This is useful in denoting one or more of the code characters that denote variable characters (**#**, **A**,..) as constants.

[Field_hivnumber](#) is an example where the [pattern](#) is "[H#####|H###|H##|H#](#)" which means that it will allow you to type a **H**- followed by either a single, two, three or four digits.

The [PatternControl](#) also auto completes the constant characters in case you did not enter them. So, when you use the above example for data entry, you need not type **C-22**, all you need to type to enter data is **22**.

Related Control

The other control similar to the above is [NumberControl](#). [PhoneControl](#) and [ZipPostalControl](#) derive from [PatternControl](#).

NumberControl

The data dictionary entry [Field_age](#) use the [NumberControl](#) .

The [NumberControl](#) is similar to [PatternControl](#) in its working except that it is meant for numbers, but actually not related. Here you can specify a [mask](#) (similar to [pattern](#) of [PatternControl](#)) and a valid range of values. The control allows you to enter numbers in any of the valid formats. However, the [mask](#) doesn't support displaying exponents. The alignment ([justify](#)) is **RIGHT**. The arguments taken are:

mask : It is specified as a string which contains **#** to represent one digit. Also a **-** or **+** can be specified to denote a sign for the number as well as **.** for specifying the decimal place.

rangefrom and rangeto : These should be numbers and denote the valid range of numbers that can be entered. If none is entered the range defaults from zero to the highest supported by the [mask](#).

As the [NumberControl](#) derives from [FieldControl](#), it should support all arguments taken by [FieldControl](#).

One [readonly](#) argument is also specified in the Users Manual which can be [true](#) or [false](#) to allow or not to allow editing. I doubt whether this is an argument for the [NumberControl](#) or whether it actually belongs to [FieldControl](#).

Related Controls

As stated earlier, [NumberControl](#) and [PatternControl](#) are similar.

Reference

Users manual -> Going Further-> Report control Breaks

Users manual -> Going Further-> Master Detail Relationship

Users manual -> User Interfaces -> Reference -> FieldControl

Users manual -> User Interfaces -> Reference -> ChooseFieldControl

Users manual -> User Interfaces -> Reference -> ChooseListControl

Users manual -> User Interfaces -> Reference -> ChooseDateControl

Users manual -> User Interfaces -> Reference -> ChooseManyControl

Users manual -> User Interfaces -> Reference -> NumberControl

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=821

http://www.suneido.com/forum/topic.asp?TOPIC_ID=720

http://www.suneido.com/forum/topic.asp?TOPIC_ID=638

http://www.suneido.com/forum/topic.asp?TOPIC_ID=629

http://www.suneido.com/forum/topic.asp?TOPIC_ID=598

Note:

Suneido's ease of use rests mostly in the different [Controls](#) it has to offer. My example shows just a part of the variety of useful controls Suneido has to offer. There are a lot more like [KeyControl](#) which by themselves require a chapter to describe them. Do not forget to learn them from the Users Manual. Though the presentation of Users Manual is a little confusing, it is quite useful.

Chapter 4 - Rules

Now it is time for **Rules**. I am not a programmer and so won't be much helpful for this topic. I would advise you to read the Users Manual => Language chapter as well.

The uses of **Rules** in relation to tables and data entry as known to me are:

- To return a default value for a field.
- To return the value for a calculated field
- To return arguments for certain **Controls** like **ChooseList** and **ChooseMany**.
- To make individual fields protected.
- To check whether the values entered in individual fields is valid or not.
- To make all or part of the fields in an **Access** or **Browse** or **ExplorerListView** protected.
- To check whether the record entered in an **Access**, **Browse** or **ExploreListView** is valid or not.

How to do that?

To return a default value, the **Rule** must be defined as an item in the library using the following convention. The name should begin with **Rule_** and should be followed by the name of the column to which it should apply as **Rule_contactnumber** in my example. All my examples are in the subfolder Rules.

A calculated field is one in which the value keeps on changing based on the values in other fields. They are defined when creating the table by starting their names with a capital letter. The difference between a usual column and a calculated one is that, the value of the field keeps on changing and it is not stored, but calculated each time it needs to be displayed or accessed. Except for the first time when they are made, they are referred to by their name changing the caps to small letter. Here again the name of the **Rule** that does the calculation is as in the previous example, but the caps in the name of the column is changed to small letter.

For the different **Controls** the name of the **Rule** will be that used inside the control. This has been dealt with earlier.

To check an individual field's value, the name should be **Rule_fieldname__valid**, while to determine whether a field has to be protected or not, the **Rule** should be named as **Rule_fieldname__protect**. These two types of **Rules** are called Control rules. Note that the names have two underscores before the last part.

The use with **Access** will be covered later

What more?

A **Rule** is a function. I understand function as a piece of code that does some thing (calculations, comparisons ...) and returns some value .It can also accept some information for doing the work (called arguments which are actually the values passed on and parameters which actually are the local variables inside the function that accept the arguments. But they are used somewhat interchangeably.). The first piece of code in a rule is to declare that the item is a function. This is done using the keyword **function()**. This is essential. If the function takes any arguments, then they can be specified inside the parentheses separated by commas. The rest of the code is enclosed inside **{}**. The next important thing is a **return**

statement which returns a value to whatever has called the function. The calculations are made using the different control statements, keywords, other functions and operators which are described better in the Users Manual. The `return` is not essential, the value of the last expression is automatically returned. But to avoid confusion, I think it is better to have one.

What did I use?

For the first type, you can see `Rule_contactrecordnumber`, `Rule_inv_recordnumber` and `Rule_contacttee`, `Rule_hivnumber2`, `Rule_hivnumber3`, `Rule_hivnumber4`, `Rule_emp_name`, `Rule_factory` and `Rule_sexes`.

The second group has `Rule_age` (Thanks to Jeff Ferguson) and `Rule_fullname`. The third has got `Rule_hivinvesigation`, `Rule_compli_list` and `Rule_risklist` as examples.

`Rule_protecthiv` is the example in connection with `Access`.

The control rules I have are `Rule_emp_dep_protect`, `Rule_factory_protect`, `Rule_persnum_protect` and `Rule_tick_num_protect`.

`Rule_contactrecordnumber` and `Rule_inv_recordnumber`

```
function ()
{
    return Timestamp()
}
```

This `Rule` is written so that it returns an unique datetime as the system time when the `Rule` was accessed. The returned datetime has the current date and time up to the precision of milliseconds. This is accomplished by using the `Timestamp()` function. We use this to ensure uniqueness of the key fields of `hiv_contacts` and `hiv_investigations` tables. Normally this is an internal column meaning nothing to an end user. But even then I prefer to display the number. But as it is a date, the default data dictionary displays it as dd/mm/yy hh/mm/ss PM or AM. Hence I decided to use the data dictionary entries described earlier.

`Rule_contacttee`, `Rule_hivnumber2`, `Rule_hivnumber3` and `Rule_hivnumber4` were defined using:

```
function ()
{
    return .hivnumber
}
```

This examples return the value of the column `hivnumber` in the current record.

The period "." signifies the current object and the current object for a rule is the record . To quote Jeff's words "Within a rule definition, the context is the record. When you access a member (the leading '.') you are accessing the record's fields."

Why I wanted a field to have the value of another field will be evident later.

`Rule_sexes`

```
function()
{
    return 'Male'
}
```

The rule just returns a string (hence the quotes) viz `Male` as the default value for the field.

Rule_emp_name is defined as

```
function ()
{
  if .emp_dep is 'Employee'
  return .fullname
}
```

This rule checks for the value of the field *emp_dep*. If it is *Employee*, the value of *fullname* is returned. In case it is not, nothing is returned. **Rule_factory** is also made along the same lines.

My example for a calculated column is *Age* in *hivmain*. The rule for that field is **Rule_age** (remember only when creating the column, do we need caps) and the code (thanks to Jeff) is:

```
function ()
{
  if (not Date?(.dobirth))
  return
  years = Date().Year() - .dobirth.Year()
  if (Date().Replace(month: .dobirth.Month(), day: .dobirth.Day()) > Date())
  years -= 1
  return years
}
```

This was a little complicated for me. First of all it checks whether the value of *dobirth* (which is another column) is a date or not, using the function **Date?** using the statement **if (not Date? (.dobirth))**. If it is not a date, nothing further is done and it returns back. If the first part is positive i.e. **if (not Date?(.dobirth))** returns **false**, then a variable *years* is created which has the value of the difference between current date and *dobirth* in years. The statement that performs this is

```
years = Date().Year() - .dobirth.Year()
```

and uses the function **Date()** and the method **.Year()**. The **Date()** returns the system date and **.Year()** takes out the year portion. Likewise the second **.Year()** takes out the year portion of *dobirth*. Then the difference between them is calculated using the operand **-** and then assigned to *years* using the operand **=**.

Next whether the month and day of *dobirth* are yet to come this year or not is checked using the statement

```
if (Date().Replace(month: .dobirth.Month(), day: .dobirth.Day()) > Date())
```

which uses the methods **.Replace**, **.Month()** and the functions **Day()** and **Date()** to do this.

day: .dobirth.Day() takes out the day portion of *dobirth* and assigns it to the variable *day*
month: .dobirth.Month() takes out the month portion of *dobirth* and assigns it to the variable *month*

Now the variables are passed on to **.Replace** as its arguments. (Note that the above two assignments are done inside the parentheses that follow **.Replace** and are separated by commas to mean that they are really arguments for the **.Replace** method) and then **.Replace** replaces the current date supplied by **Date()** (The first one). This is checked with the

unchanged system date again supplied by `Date()` (the last one) using the operand `>`. If the changed date is greater than current date (i.e. if the day is yet to come), then the next line is executed which is `years -= 1` This is to reduce the value of years by one if the previous statement was `true`. The operand is `-=` which actually means `years = years - 1`.

The final statement returns the value of `years`.

As Jeff says, the `Rule_age` is not perfect as it decreases the `years` even if only one more day is left to reach the `dobirth`. I was too lazy to change that. Perhaps you are not.

My second example for a calculated column is `Fullname`, the `Rule` for which is `Rule_fullname`. The code is

```
function()
{
return  .firstname.Trim().Lower().Capitalize() $" " $
        .middlename.Trim().Lower().Capitalize() $" " $
        .lastname.Trim().Lower().Capitalize()
}

```

the lines return the value of `firstname`, `middlename` and `lastname` of current record joined together after processing it using `Trim()` and then `Lower()` and then `Capitalize()`. `Trim()` removes the leading and trailing space if any from its argument, `Lower()` changes all letters to lower case and `Capitalize()` makes the first letter of the string caps and others small letters, i.e., sentence case. This rule uses `$` to join the 3 parts with a single space separating them to make a single string out of them.

The control rules are four in number in my library, all of them are constructed on the same lines.

`Rule_emp_dep_protect`

```
function()
{
if .emp_dep is 'Non entitled' or .emp_dep is 'Employee'
return true
else
return false
}

```

The control rules should have a `true` or a `false` as their return. If the field has to be protected (or to declare the value as valid, in case of `__valid` rules), the value should be `true` otherwise `false`. My rule checks whether the `emp_dep` field has got the value of `'Non entitled'` or `'Employee'` in which case my field will be protected. If `emp_dep` has a different value, the field is editable. All other `__protect` rules are almost same except that they lack the `or` part, i.e., they do not check whether the value is `'Employee'`.

Further reading

I do not think this is an easy task as it is related to the core of any programming language, the language itself. Anyway, at least the Language section of Users Manual should help many.

Reference

Users Manual -> Going further -> Master-Detail Relationship

Users Manual -> Going further -> Creating a Form Report

Users Manual -> Database -> Reference->Timestamp

Users Manual -> Language -> Reference-> Date

Users Manual -> Language

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=860

http://www.suneido.com/forum/topic.asp?TOPIC_ID=844

http://www.suneido.com/forum/topic.asp?TOPIC_ID=840

http://www.suneido.com/forum/topic.asp?TOPIC_ID=789

http://www.suneido.com/forum/topic.asp?TOPIC_ID=774

http://www.suneido.com/forum/topic.asp?TOPIC_ID=760

http://www.suneido.com/forum/topic.asp?TOPIC_ID=703

Note:

You may encounter some problems with Rules as mastering them demands mastering the language. Help is always available from the Suneido Forum.

Chapter 5 - The Data Entry Screen

Now you have all the ingredients to make a data entry screen.

How to do that?

You have three options in designing a data entry screen -use an [Access](#), a [Browse](#) or an [ExplorerListView](#).

An [Access](#) is designed to show one row of the table at a time and the layout is a form (the screen shows different rectangular fields for entering the column values for one record). A [Browse](#) shows all records in a tabular layout. An [ExplorerListView](#) shows all records in a tabular layout. As well, it provides a form to edit any one record selected in the tabular section.

What exactly does that mean?

[Access](#) is the more sophisticated of the three. It is meant primarily to view one record a time and even to locate one or print one. It can also restrict the records shown and select records based on values you can specify on screen. But, it can support the other two inside it to display details table of the primary displayed in the [Access](#) proper. It also allows you to see summaries of different columns of the table or print all records of the table in a predefined format. Also, it provides a CrossTable to get statistics related to the table you provide.

[Browse](#) is meant to view the entire table at a time. It is just to see every record simultaneously and edit them as needed. [Browse](#), if used for a big table, can consume a lot of system memory as it reads all the records simultaneously. It supports deleting of old records and adding new records.

[ExplorerListView](#) is a combination of the above two. But does not support print and summaries like [Access](#). The interface provides all you want - view all and edit one. The disadvantage is also the same - it restricts the space available for the form area (but scroll support is there).

What more?

I will first try to sketch the use of [Access](#). The code for your data entry screen should be entered as an item in the library . The convention used for name of the item is that it should start with a caps and has '_Access' as the ending. The caps is essential while the ending is not. The code can start with

```
#(Access
```

or with

```
Controller  
{  
  Controls:  
    (Access
```

and then the arguments are provided.

If you want to use the [Access](#) as a stand alone, follow the second method and end the name of the item with [Control](#).

The first and only essential argument for [Access](#) is a [query](#), which can be as simple as a table name or can be a complex one as per the criteria given in Users Manual. The [query](#) is

provided as a string. There will be no problem even if it is given without quotes if the [query](#) is just a table name.

The name you want for the access is specified as [title](#) and should be a string. It defaults to the query specified. It is aligned in the center on the top and uses a bold font.

Next argument is called [control](#) and is the screen layout as desired. The default will lay the different fields one below the other aligned together at the Prompt and Field interface and one field per line. The control is specified as an object. But even if you do not specify the <#>, then also there is no problem. I will deal with this option and possibilities later.

Now there are two options [startLast](#) or [startNew](#) one of which can be [true](#) .As you might have guessed, they specify whether the [Access](#) should start with the last record or new record respectively. Obviously both cannot exist together. The default is to start on the first record.

The next option is [stickyFields](#). It is an object and so has to start with '<#>(' and end with ')'. The members can be any field in the [Access](#) and the names are written inside quotes. This option allows you to specify fields whose values have to stick from the previous new record entered. This will happen during the current session only. Once you leave the screen, the previous values of sticky won't be there. An example could be -

```
stickyFields: #('fieldname1' 'fieldname3' 'fieldname4')
```

Now you can have a [protectField](#) option which has to be the name of a [Rule](#) without the [Rule_](#) and is specified as string. The [Rule](#) should return [true](#) or [false](#) or an object containing the name of fields. If the value of [Rule](#) is [true](#), then all fields are protected from editing and the record is protected from deletion. If it is [false](#), then the reverse happens. If the [Rule](#) returns an object, then the field names in the object are protected and they are grayed out and editing is not allowed on those fields. If the first member (member 0) of the object is '[allbut](#)' , then all fields other than those specified are protected. If one of the members is [allowDelete](#) then deleting the record is allowed otherwise it is not possible.

A [Rule](#) that is used for [protectField](#) could be as below -

```
function()
{
  return #({allowDelete: , fieldname2:, fieldname5:, fieldname8:})
}
```

Another option is [validField](#) which also should be the name of a [Rule](#) and hence specified as a string inside quotes. This rule is constructed as per your wish and should check the values inside the current record and assess whether it is a valid record and if so should return either empty string or if the record is invalid another message, a string to be displayed. This rule is called only when a record is saved using the buttons or on closing the Access. The default checks whether the controls are valid. If not valid, the message shown is the name of the fields whose [Controls](#) are not valid.

Then you can specify a [select](#) which is used to restrict the [query](#) initially. It is specified as a container object (containing other objects) with three parameters per object - a field, an operator and a value. As Jeff says, operation should be the actual operator, not the description of the operation that is displayed in Select, i.e. "=" instead of "equals". So a valid [select](#) can be:

```
Object(#{(fieldname1, '<', 100) #(field2, '=', 'something')})
```

The [Access](#) has a select button that allows you to select the records that you wish to view or edit based on specified values of particular fields. By default all fields are shown. If you do not want certain fields to be shown in the select option, then you can use [excludeSelect](#) for specifying those fields. This is specified as an object with the members being the name of the fields. The construction is the same as for [stickyFields](#)

The Users Manual also tells us about few more options - a [locate](#) option that allows you to customize the locate button and the window that comes on pressing the button. The locate option is specified as an object and so has a <#> and the members are enclosed in a pair of parentheses. There has to be two named members - [keys](#) and [columns](#). The arguments for each member are specified inside a pair of parentheses each and should include the field names separated by space. The names of the fields can be enclosed in quotes, though not essential. The [key](#) member determines in which all field locate can be done. The field specified in [key](#) should have only unique records, i.e. there should not be duplicate values in that field. You are allowed to select one field for locate by providing a list which contains the prompt of the fields you have specified in the [key](#) member. The [column](#) member determines the columns that should be shown in the window that comes up on pressing the ellipsis button on the locate field. If you do not specify a [locate](#) option, then only the key field is available for locate and all columns are shown. An example of a valid [locate](#) option could be

```
locate: #(keys: (fieldname1 fieldname2) columns: (fieldname2 fieldname3))
```

What determines the order in which the fieldnames appear in the list or of the columns in the window are not known to me.

[dynamicTypes](#) I think this is a powerful option allowing you to customize the menu buttons and [newOptions](#) to specify the order of the menu labels. But I do not know the usage of these options. Some of you can teach me.

What did I use?

You can find my code in [hiv_lib](#) under the name [Hiv_AccessControl](#) . I will put a cut down version here for easy understanding.

```
Controller
{
  Controls:
  (Access 'hiv_main'
   title: "HIV CLINIC"
   (Vert
    (Horz
     (Center
      (Horz(Personal, title: 'Personal Information')
       (Skip 10)
       (Employee, title: 'Employee Information'))))
     (Horz (Center(Contact, title: 'Contact Information'))
      (Tabs
       (Horz
        (Center
         (GroupBox 'Case Details'
          (Vert
           (Horz (hivnumber) (Skip 18) (dodiagnosis) (Skip
            (Horz (Skip 16) (do_death) (Skip 143) (causede
             )
            )
           )
          )
         )
        )
       )
      )
     )
    )
   )
  )
}
```

```

        )
    )
    // I have something more here
)
)
startNew: true
protectField: 'protecthiv'
)
}

```

So what did you understand by -

```

Controller
{
  Controls:
  (Access 'hiv_main'
  title: "HIV CLINIC"

```

? This means that the code is an **Access** that uses the table *hiv_main* as its **query** and the heading will be *HIV CLINIC* and everything inherits from **Controller**. The last **startNew: true** tells the **Access** to start a new record whenever the **Access** is used. The rest of the code from (**Vert** till **)** prior to **startNew:** is the **control** option which I will describe in detail now. The final **)** and **}** are the closing braces for the (**(** and **)** prior to **Access** .

The **control** as I told earlier determines how a data dictionary field is displayed in relation to others in the data entry screen, i.e., the layout. It uses certain controls.

The first control that I used is **Vert** the arguments of which can be one or more controls including the data dictionary entries without the **Field_** in their name, each specified inside a pair of parentheses. My **Vert** has 3 Controls (and so three horizontal rows of items) inside it

1.

```
(Horz
  (Center
    (Horz
      (Personal, title: 'Personal Information')
      (Skip 10)
      (Employee, title: 'Employee Information'))))
```
2.

```
(Horz
  (Center(Contact, title: 'Contact Information')))
```
3.

```
(Tabs
  (Horz
    (Center
      (GroupBox 'Case Details'
        (Vert
          (Horz (hivnumber) (Skip 18) (dodiagnosis) (Skip 27) (risk)
            (Horz (Skip 16) (do_death) (Skip 162) (causedeath))
          ))))
    // I have something more here
  )
)
```

All the three **Controls** are placed one below the other vertically . That is what **Vert** does - arranging its member **Controls** in a single column one below the other . If the **Controls** provided to **Vert** are data dictionary entries, then the Prompt-Field junctions are aligned in a vertical line.

Now what is the first member for the **Vert**? It is a **Horz**, the function of which is to align its members in a horizontal row one by the side of the other. Like **Vert** it can take as many members as provided and they can be other **Controls** or data dictionary entries and each item should be enclosed in a pair of parentheses.

In my example it takes a **CenterControl** which aligns its member to the middle of the current item (the item that has called it) which is **Horz** for us. So the **Center** centers horizontally in our example. What does it center? It takes only one control and that is centered. In our example I wanted two items (taken together) to be centered. So a **Horz** which can take more than one member was specified as the **Center**'s argument and I put two of my custom controls inside it. The custom controls were groupboxes which held a few data dictionary entries. Their names are **PersonalControl** and **EmployeeControl**. [The **title** specified along with them is the name of the groupbox name that has to be displayed and has got nothing to do with our topic at present]. Wait, there is one more thing inside the **Horz** and that is **Skip** which takes an argument with it specified as a number. This **Skip** puts non stretchable (even when the window or its containing control is maximized or resized, the space is maintained as such and is not changed proportionally) space in the context it is called. We called it inside a **Horz** and so a horizontal space is put in between **Personal** and **Employee**. Finally you can see that in the first row, I have two of my custom controls with an intervening space of 10 pixels in between them centered horizontally in the middle of the screen.

The second line inside the **Vert** is similar to the previous except that the **Center**'s child control is a single item, again a custom control, a groupbox. So this line makes the horizontal row where **Contact** is centered.

The third line contains a new control - **TabsControl** which can take one or more controls as its argument. In my example, I have placed a **Center** and 3 **ExplorerListViews** along with its arguments inside the **TabsControl**. Each control should be placed inside a pair of braces and should in addition to the **Control** and its arguments (if any) have a member **Tab: 'tabname'** where **tabname** is the text that appears on that particular tab.

We will examine the **Center** inside the **Tab** -

This line is also similar to the one above but for the fact that **Center** uses a **GroupBoxControl** directly instead from a predefined control. The **GroupBox** draws a border around its child controls and also puts the name specified in the **title** option in the left upper corner of the border. If no **title** is specified, a full rectangle is drawn. It also allows only one control inside it which in my example is a **Vert** which contains inside it two **Horz** and each of them carrying few data dictionary inside them. **Skip** is used to keep the alignment between the fields and to space them.

The **startNew** as told earlier makes my **Access** start with a blank record.

The **protectField: 'protecthiv'** tells the **Access** to use **Rule_protecthiv** to find out which all fields to protect i.e. non editable. The **Rule_protecthiv** is simple,

```
function ()
{
    return #({allowDelete:, fullname:, age:})
}
```

The fields **fullname** and **age** are protected always, as no particular condition is specified. The **allowDelete** allows deleting of records.

Related commands

The [Horz](#) and [Vert](#) and [Center](#) and [Skip](#) do all my work. If you find them tiresome to use, you can use other controls like [FormControl](#), [GridControl](#) and [PairControl](#). The [EtchedlineControl](#), [BorderControl](#), [StaticControl](#) and [FillControl](#) can be used to beautify the layout in addition to [Skip](#).

As stated earlier, [Browse](#) and [ExplorerListView](#) are other options for entering data in tables. Learn about these controls from Users Manual.

Reference

Users Manual -> Getting Started -> Create an Access

Users Manual -> Going further -> Master-Detail Relationship

Users Manual -> User Interfaces -> Reference -> [AccessControl](#)

Users Manual -> User Interfaces -> Reference -> [HorzControl](#)

Users Manual -> User Interfaces -> Reference -> [VertControl](#)

Users Manual -> User Interfaces -> Reference -> [CenterControl](#)

Users Manual -> User Interfaces -> Reference -> [SkipControl](#)

Users Manual -> User Interfaces -> Reference -> [GroupBoxControl](#)

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=869

http://www.suneido.com/forum/topic.asp?TOPIC_ID=868

http://www.suneido.com/forum/topic.asp?TOPIC_ID=861

http://www.suneido.com/forum/topic.asp?TOPIC_ID=856

http://www.suneido.com/forum/topic.asp?TOPIC_ID=853

http://www.suneido.com/forum/topic.asp?TOPIC_ID=848

http://www.suneido.com/forum/topic.asp?TOPIC_ID=844

http://www.suneido.com/forum/topic.asp?TOPIC_ID=841

http://www.suneido.com/forum/topic.asp?TOPIC_ID=840

http://www.suneido.com/forum/topic.asp?TOPIC_ID=806

http://www.suneido.com/forum/topic.asp?TOPIC_ID=803

http://www.suneido.com/forum/topic.asp?TOPIC_ID=799

http://www.suneido.com/forum/topic.asp?TOPIC_ID=789

http://www.suneido.com/forum/topic.asp?TOPIC_ID=779

http://www.suneido.com/forum/topic.asp?TOPIC_ID=759

http://www.suneido.com/forum/topic.asp?TOPIC_ID=703

Note :

All controls used for screen formatting have a [xmin](#), [ymin](#), [xstretch](#) and [ystretch](#) - understanding these will help you use the right controls effectively.

Chapter 6 - The details table dataentry

I left out something in the previous chapter - what was inside three of the [Tabs](#). All three contained one [ExplorerListView](#) each. We will see how to make the details part of a data entry screen.

How to do that?

Using an [ExplorerListView](#) is easy. The power of [ExplorerListView](#) lies when used in connection with an [Access](#) to display a details table (The [Access](#) showing the master table). [Browse](#) can also be used for the same purpose, but my personal preference is [ExplorerListView](#).

What exactly does that mean?

The [ExplorerListView](#) can be used as a data entry screen for a 'standalone' table or can be used to display the details table of a master details relation ship. The arguments differ slightly under these circumstances. The arguments taken by [ExplorerListView](#) are

1. The [model](#) with its arguments and specified as an object. So the first argument for [ExplorerListView](#) has a <#> and the members are enclosed in parentheses separated by commas. The first member is the name of the [model](#) which can be [ExplorerListModel](#) if the [ExplorerListView](#) is used for a 'standalone' table and [ExplorerListModelLinked](#) if used in a master details circumstance.

The second member is the [query](#) and is specified as a string inside quotes. In a master details circumstance, there is no need for you to restrict the [query](#) based on the master field. I mean that if your [ExplorerListView](#) is showing a details table and you want the [ExplorerListView](#) to show only the records that have a specific value in the foreign key, then do not do that restriction in the [query](#) - do not make the query something like '[table where column is...](#)', but let it be only '[table](#)'. The restriction is taken care of by [ExplorerListView](#) itself.

The third member is the [keyfields](#) specified as an object and so, has parentheses around the name of the keyfield which itself is enclosed in quotes. You can also have many fields which together make a key. The [keyfields](#) are those of the details table. What is the need of this member when already it is specified while creating the table is unknown to me.

The fourth member is the [headerFields](#) which are fields from the header(master) table which have to be copied to the details table. This member should be specified only in case the [model](#) is [ExploreListModelLinked](#). If the [model](#) specified is [ExplorerListModel](#), then you have to leave this member empty as this model is not intended to be used in a master details setup. If the details table has a column name with the same name as in the master table and the name is specified on the [headerFields](#), then the value from the master table is copied to the column of the details table. If the details table has no column as specified in the [headerFields](#), then the value from the master table is shown in the [ExplorerListView](#) in screen, but is not stored in the details table. The [headerFields](#) can be any other field than the one specified in the [name](#) option discussed below.

2. The second is [view](#) which is to be specified as a control and hence an object and so has a <#> and the members enclosed in parentheses. The first member should support [Get](#) and [Set](#). This member is similar to the control member specified in the [Access](#). This determines the layout of the form area of the [ExplorerListView](#)

3. The third is the [query](#) in which you specify the query you want the [ExplorerListView](#) to use and is supplied as a string. This is already specified in the [model](#) argument and can be left out. I do not know what will happen if you specify a query different from what you specified in the [model](#) is specified here and also why an option to specify a [query](#) is supplied twice(may be for compatibility reasons).
4. The fourth is [columns](#) which are the columns you want to specify to be displayed in the list view. The default value is none from the query specified, though all columns from the specified query would have been a better default. The order the columns are specified in this option determines the order in which they are displayed from left to right. Though the manual says that the [column](#) argument need not be specified(it can be passed on to the [model](#) argument), I am not aware of how to accomplish it. The [columns](#) argument is given as an object with <#> and parentheses the column names being separated using commas.
5. The next important argument is the [linkfield](#) and is needed only in the master details scenario and it is the field in the details table based on which you want the restriction to be done. By this I mean that if you want the [ExplorerListView](#) to show only a few records of the details table based on a value in another field in the master, then the name of the column in the details field is given here. It is not necessary that the column need to be the foreign key of the details table. This option is specified as a string inside quotes.
6. The [name](#) argument is the field of the master table (or can also refer to a 'virtual' field that exists only as a rule) to which the [linkField](#) is to be 'linked'. Taken together, [ExplorerListView](#) will show in the [ExplorerListView](#), the records from details table with values in [linkField](#) that match the value of [name](#) field of master table. Again this is needed only in master details scenario. The [name](#) is specified as a string. One important thing about [name](#) - If you have more than one [ExplorerListView](#) inside an [Access](#) as I have, then each [ExplorerListView](#) has to have different name. But probably you may want every [ExplorerListView](#) to be linked to the same field, in which case you have to use a 'virtual' field existing only as a rule (need not be defined in the table)which returns the value of the original field. The [linkField](#) is filled up automatically with the value in the [name](#) field of the master table.
7. The [title](#) argument serves to provide a title of your choice to the [ExplorerListView](#). It is specified as a string inside quotes and will be displayed on the top of the [ExplorerListView](#) in bold font.
8. [reverse](#) can be [true](#) in which case the order of records shown in the list view of [ExplorerListView](#) will be reversed.
9. [readonly](#) if [true](#) will not allow any changes to be made in the [ExplorerListView](#).
10. [status](#) if [false](#) turns off the status bar at the bottom of the [ExplorerListView](#). The default is [true](#).
11. If the option [noShading](#) is [false](#), then the shading given to alternate rows of the list view is turned off.
12. By default you can sort on a field by clicking on the column headings in the list view portion of [ExplorerListView](#). If you do not want this, [noHeaderButtons](#) should be specified as [false](#).
13. [validFields](#), [stickyFields](#) and [protectFields](#) are also supported as in [Access](#), but [startNew](#) is not.
14. A [primary_accessobserver](#) option is also specified in the Users Manual, but I am not sure

of its use and usage.

What did I use?

I will explain one of the [ExploreListViews](#) used in the [Tab Investigations](#)

```
ExplorerListView
#("ExplorerListModelLinked", "hiv_investigations" ("inv_recordnumber") ()),
#(Center
  (GroupBox ''
    (Vert
      (Horz (hivinvestigations) (inv_date) (inv_result))
      (Skip 5)
      (Horz
        (Center
          (Horz
            (Button 'New'xmin:100)
            (Skip 10)
            (Button 'Delete'xmin:100)
            (Skip 10)
            (Button 'Save' xmin: 100)
          )
        )
      )
    )
  )
)
columns: #(inv_recordnumber,hivnumber,hivinvestigations,inv_date,inv_result
linkField: 'hivnumber',
title: "Investigations",
name: 'hivnumber3'
```

Note that inside the [Tabs](#), the whole code is enclosed in a pair of parentheses.

What I wanted was to show one of my details table [hiv_investigations](#) in an [ExplorerListView](#) inside an [Access](#). The rows of details table should only be those that correspond to the value of [hivnumber](#) in the master. I wanted the field [hivnumber](#) of [hiv_investigations](#) to be filled up automatically with the value in [hivnumber](#) of master table , [hiv_main](#).

As I wanted a master details relationship, I chose to use [ExplorerListModelLinked](#) as the model name. Note that, though I wanted only some of the records of details table (as dictated by the value of [hivnumber](#) in the [Access](#)) to be shown in the ExplorerListView, my query is only [hiv_investigations](#). This restriction as well as copying the [hivnumber](#) of [hiv_main](#) (which is not specified in the code) to [hivnumber](#) of [hiv_investigations](#) is done by the [linkField](#) the value for which is [hivnumber](#) (of [hiv_investigations](#)). I wanted the [hivnumber](#) to be copied from [hivnumber](#) of [hiv_main](#). But only one [hivnumber](#) in [hiv_main](#) was a problem as a field can be used only once. So I made a [Rule_hivnumber3](#) which returns [hivnumber](#) and used [hivnumber3](#) as the [name](#). As I had no other field to be copied from the master table, I left the [headerFields](#) option empty - the empty pair of parentheses in

```
#("ExplorerListModelLinked", "hiv_investigations" ("inv_recordnumber") ())
```

The [view](#) starts with [#\(Center](#) and ends with [\)\)\)](#). You should be able to understand what I have put inside the [view](#) from the previous discussion. Note that inside the view I have 3 [Buttons](#) for starting a new record, deleting a record and for saving one. The default [ExplorerListView](#) uses context menu to add or delete or restore a record and records are saved as a new one is

started or the [ExplorerListView](#) is quit. [New](#), [Save](#), and [Delete](#) are the main buttons that can be specified without specifying an [On_method](#).

Inside each pair of parentheses that is used to define the [Buttons](#), the word [Button](#) tells the [ExplorerListView](#) that a press-button control is to be used, the string that follows is the string that appears on the button and the [xmin](#) determines how many pixels wide the button should be.

Related commands

In addition to [ExplorerListView](#), there is one [ExplorerVirtualListView](#) which also do the same work and usage is also probably similar which is more suited for large amounts of data.

As told earlier, [Browse](#) can also be used to display a details portion in a master details scenario.

Reference

Users manual ->User Interfaces->Reference-> ExplorerListViewControl

Users manual ->Going further -> Master-Detail Relationship

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=866

http://www.suneido.com/forum/topic.asp?TOPIC_ID=839

http://www.suneido.com/forum/topic.asp?TOPIC_ID=768

Note :

[Browse](#) would fit in data entry screen much better than [ExplorerListView](#).

Chapter 7 - Print

How will you rate a database management software which is unable to print its data in a readable format? Let us see the possibilities given by Suneido in printing reports.

How to do that?

Suneido uses `ParamsControl` to manage printing.

What exactly does that mean?

As any other control, it is to be written as an item in the LibraryView to be used in a program/standalone. The name of the Library record can be anything that starts with a caps. The ending has to be `Control` if the code need to be used in a stand alone. The code can also be executed from Workspace. `Params` pops up a window with three buttons - Print, Preview & Page Setup which are used for the purposes their names suggest.

What more?

The `ParamsControl` takes the following arguments -

1. A `title` which is a string and supplied inside quotes. It is the title that appears on the top of the `Params` Window in bold and left oriented. The same string is also used in the heading of the actual print. If you want different strings in both the places, then no straight forward arguments can be supplied, though there are ways to do that.
2. Be sure to read this twice - There is a `Params` argument that can be specified to the `ParamsControl`. (The name of the argument or of the control should have been changed to avoid confusion). Here you can specify a user interface to allow users to select what should be printed. It is constructed in the same way you create a `control` of `Access` or `view` of `ExplorerListView`. `Vert`, `Horz`, `Form` etc can be used to create the interface. Among other controls (which were discussed earlier) that can be specified, an object the first member of which is `ParamsSelect` and second member is `fieldname` can be specified for the interface. You can supply as many controls as you want. An example may be

```
Params: (Form (ParamsSelect fieldname1) (ParamsSelect fieldname2))
```

If `ParamsSelect` is there, then in the window that comes up, there will be one field with a list which contains the different operations possible (like starts with, ends with, less than, contains...), one or two fields (based on the operation selected) that derive from the data dictionary entry as specified by the `fieldname`. The user is allowed to select an operation from the list and to enter the value in the other field(s). The matter that needs to be printed out can be modified using the values entered. The `fieldname` specified need not be there in the table. How you will modify the report based on the values selected will be discussed later.

3. A `name` option , a string which can be specified without the quotes can be given to store the options specified in the `Params` argument. This is used to set the default for `Params` fields when the `Params` is used next time. This is stored in the system table `params` under the current user's name.
4. If you want a standard default to come up all the time rather than the saved values, then

you can specify a [SetParams](#) option. I have not used this option and so do not know the details.

5. A [validField](#) option which should be the name of a [Rule](#) similar to that used by [Access](#) or [ExplorerListView](#) can also be specified to check the validity of the parameters supplied by the user.
6. The [printParams](#) option if specified will print the *fieldnames*, the operation selected and the values entered in the [Params](#) argument in the default heading. This is printed in small letters (Arial 8) on the left lower line of the default page heading. The values for this argument should be an object specified inside parentheses, the members could be any or all of the *fieldnames* supplied in the [Params](#) argument. The default for [printParams](#) is that nothing will be printed.
7. A [header](#) argument should be specified as [false](#) if you do not want the default page heading provided by Suneido. The default heading centers the string "Suneido Software" in Arial size 13 font along with the date and time in [LongDate](#) format along the left upper corner and the page number on the right upper corner both using Arial size 8. If a [title](#) argument is specified, then that string is specified below "Suneido Software" in bigger font. If [printParams](#) is specified, then a corresponding string is specified on the lower left corner of the heading. You can also use any [Format](#) of your design as the argument for the [header](#).
8. The most important argument is the [format](#) the default for which is the [QueryFormat](#) written with or without the [Format](#). This accepts a lot of arguments, the most important of which is a [query](#). You can have formats other than [QueryFormat](#). But if what you want to type is data from a table (as is the usual case), then [QueryFormat](#) is the best option or rather the only one. The discussion on [QueryFormat](#) will be taken up in the next chapter.

What did I use?

I will put my code without most of [QueryFormat](#) here. I have 3 reports named [Print_ContactsControl](#), [Print_IndividualControl](#) and [Print_ListControl](#). The code for [Print_ContactsControl](#) and [Print_IndividualControl](#) are almost the same (except for the [title](#)) if [QueryFormat](#) is excluded and is

```
Controller
{
  Controls: (Params
    title: "Personal and Contact Information of Retrovirus Infected
    Params: (ParamsSelect collectname )
    QueryFormat
    {
      Query ()
      {
        where = _report.GetParamsWhere("collectname")
        return 'hiv_main rename hivnumber to collectname' $ whe
      }
      // I have more here
    }
  )
}
```

The [title](#) *Personal and Contact Information of Retrovirus Infected Cases* will be printed along with the heading. The same string is used in the Report window that comes up.

I did not want my [ParamsSelect](#) values to be saved. I wanted them to be blank all the time and so no [name](#) was specified. And I wanted the default Suneido page heading to be as it is(at least a little publicity for their great program) and so [header](#) was not specified as [false](#).

The [Params](#) argument has only one object the members for which are [ParamsSelect](#) and [collectname](#). Check out the data dictionary entry for [collectname](#) named [Field_collectname](#). If you followed my earlier advice, then you will be able to understand its control - [KeyControl](#). In short my [collectname](#) will use [hiv_main](#) table to show the records to show the records in it and the field will get filled with the value of [hiv_number](#) in the record selected and then the value of [collectname](#) changes to that [hiv_number](#). I could have used [hiv_number](#) instead of [collectname](#) or any other [fieldname](#). I selected a [KeyControl](#) to give the users the ease of selecting from a list.

Now the [query](#) option of [QueryFormat](#) -

```
where = _report.GetParamsWhere("collectname")
```

This sentence assigns the value from [ParamsSelect](#) to the variable [where](#). The [_report](#) is the reference to the current report and [GetParamsWhere](#) is the function that gets the value from its argument ([collectname](#) in this example) of [Params](#) argument and returns a string that begins with the word [where](#) followed by an appropriate operator (as = when the selected operation is equals) followed by the value of [collectname](#). An example of value assigned to [where](#) in my example could be `where collectname = "H-2"` . We need this sentence to be able to use the value specified in [Params](#) argument to limit what is printed. If you are not using [Params](#) argument, this sentence is not needed.

Now the [query](#) proper. This can be just a table name in which case all data from the table will be considered. In my example, I want to use the [hiv_main](#) table. And I want to restrict it using the value of [collectname](#). But I don't have field called [collectname](#) in my table. The data restriction actually has to be done as per the field [hiv_number](#). So I renamed the [hiv_number](#) to [collectname](#) (`'hiv_main rename hivnumber to collectname'`) and used the [where](#) defined earlier to restrict the data. So the code `'hiv_main rename hivnumber to collectname' $ where . '$'` joins two strings. If you want to print in an order or wants to use [QueryFormat](#) to its full potential, then a [sort](#) has to be done on a field. Hence my final [query](#) statement has become `'hiv_main rename hivnumber to collectname' $ where $ 'sort collectname'` . As the [Query](#) option is a method, it has to have [return](#) statement similar to a function.

The [Print_ListControl](#) is easy to understand if you understand the previous. It differs from the above in that the [Params](#) argument has four objects contained in a [Vert](#) which uses data dictionary of [emp_dep](#), [sexes](#), [factory](#) and [dodeath](#). As the data dictionary entries are used as they are in the table, the [query](#) for the [QueryFormat](#) does not contain a [rename](#). As I wanted to have the characteristics based on which the print is made in the print, I have specified a [printParams](#) as well.

We will continue with the rest of our discussions in the next chapter.

Chapter 8 - Formats used in Reports

[QueryFormat](#) which we supplied as the [format](#) argument to [ParamsControl](#) belongs to the generator group of formats. Generators give order and sequence to reports and uses other formats to print out material. [QueryFormat](#) is the only generator that I am aware of that deals with printing from a table / database.

The Report Breaks

As told in the previous chapter, [query](#) is the main argument that [QueryFormat](#) takes. If a [sort](#) is performed on the [query](#), then we can use the report breaks provided by [QueryFormat](#). The [sort](#) can be performed on as many columns as you wish.

[QueryFormat](#) provides us with report breaks which helps us to print our report in a specified order. Each report break is specified as a member of [QueryFormat](#) as [Reportbreak](#) : or as methods as [Reportbreak\(\)](#) {}. Specifying them as methods is more beneficial as extensive formats can be used. It also allows us to pass on [data](#) which contains the current record from [query](#) that is being processed. Whether specified as methods or as members, the content should evaluate to a [Format](#) or to [false](#) for all breaks except [Total](#).

I will take up the report breaks, but not in the order in which they are printed. I think that will help in better understanding.

The [query](#) returns a number of records. They are printed in the [Output](#) break. If this break is not specified, it is assumed and the default format used by [Output](#) viz [RowFormat](#) is used to print all records from the query in a columnar format. All columns of all records are printed by default. If the contents run over to another page, that is also handled. The [Output](#) does not take [data](#) as an argument.

If you want print something before each record in the [Output](#) break, then [BeforeOutput](#) is the break you need. If, in addition you want something after each record in the [Output](#) then [AfterOutput](#) is the break. Both these breaks are called as many times as there are records in the [query](#).

In [Before_fieldname](#) break, the [fieldname](#) refers to the column on which [sort](#) was performed. You can have as many [Before_fieldname](#) breaks as the number of fields on which [sort](#) was performed. This break is called before [Output](#) when the value of the [fieldname](#) changes in the [Output](#).

The [After_fieldname](#) break is similar to [Before_fieldname](#) except for that it is called after [Output](#) when the value of the concerned field changes in the [Output](#). Again this break requires [sort](#) in the [query](#).

If you have more than one [Before_fieldname](#) or [After_fieldname](#), then [Befores](#) are called in the same order as the [sort](#), and [Afters](#) are called in the reverse order. You can see that both these breaks are called as many times as there are different values in the concerned [sort](#) fields. To use these breaks, you need to have done a [sort](#) on the concerned fields.

The [Before](#) is the break that comes on the top of everything. It is called only once and before every other break.

The [After](#) break is called after every other break except [AfterAll](#). It is called only if there is any output in the report.

[AfterAll](#) break is called after everything else. It differs from [After](#) in that it is called even if there is no output in the report.

The **Header** is the break which determines the **QueryFormat**'s header but can also replace the standard Page heading. Whatever is specified in this break is displayed in all pages. In the first page, this comes before **Before** and in all pages, it becomes page heading or comes below the page heading. If you do not supply a **title** or **header** to the **ParamsControl**, then **Header** replaces the standard page heading. If a **title** is specified or **header** is specified, then **Header** comes below page heading in all pages. If **Output** is not **false**, but the default of **RowFormat**, then the **Header** automatically uses the format **ColHeadsFormat** to print the column heading of the column specified in the **Output**. If you specify something other than the default format to **Output** break, then **Header** automatically becomes **false**.

There is also **Total** which is always specified as a member and written as an object, the member of which should be a fieldname on which sort has been done. Automatic totaling is done on the field specified. **Total** can be used inside more than one break. I do not know anything regarding its usage. It is not a true break.

The default 'formats' used in different breaks default to **false** except for **Output** for which it is **Row** and uses all columns from the **query** supplied and for **Header** for which it is **Output**'s **Header** the default for which is **ColHeads**. The **query** is a necessary argument while others are not.

What did I use?

I will use **Print_ContactsControl** as an example for exploring the use of **QueryFormat** as well as to discuss about the different formats other than **QueryFormat**.

In my **QueryFormat** the **query** is a method and so has a **return** statement. The **query** is constructed using the value in the **printParams** as detailed in the previous chapter as

```
'hiv_main rename hivnumber to collectname' $ where $ 'sort collectname '
```

The **hiv_number** is a key field and so the **query** will be returning only one record. The **Output: false** will make the default break **Output** print nothing which otherwise would have printed the record in a tabular field. This automatically will make the **Header** also **false**.

```
Before_collectname(data)
{
    return Object ('Hiv_Main', data)
}
```

This break will be called each time the value of **collectname** in **Output** changes. But as our **Output** is **false**, this break will be called only once. The format that this break uses is **Hiv_MainFormat** which is a custom format that was made with the help of Tracy Arams. Each format that is specified in a break is presented as an Object along with its arguments, hence the parentheses. The argument that is provided to **Hiv_MainFormat** is **data** which is the current record that is being processed. The code for **Hiv_MainFormat** is in the record of the same name in **hiv_lib** in the folder **Common**. You will be able to understand the code when we finish our discussion on formats. This format is responsible for printing out the things between the first 2 bold lines (including the second one) that you see in the report.

Now the second break that I used - **After_collectname**

The code is

```
.Append (
```


Row. I wanted the 6 columns of the table to be printed in 4 columns. So my format is as I designed.

The format's first member is **Vert** which is a container and works similar to the **Vert** encountered while building data entry screen. They are not identical, though they perform similar work. Here they accept an **x** and a **y** argument along with the containing items to place the items in specific positions. The **x** and **y** specify the horizontal and vertical position respectively and they are in inches and are relative to the container and not the page.

The **Vert** contains an **Horz** and a **Vskip**. The **Horz** is again a container which positions its members objects in a row. The **Horz** in turn contains 4 **Vert**. Each **Vert** contains 2 objects each. Each member Object is made in the manner as specified below -

```
#(contactnumber " font: (name: Arial size: 12 weight: 600) x:0)
```

except for the second items inside the first and second **Verts**.

The first item inside each object is a field name. This asks the **Params** to use the format specified in the data dictionary entry to print the data inside that field for the present record. I have specified only the most essential **Format** details in the data dictionary entries. Of the fields used, only the dates have a **ShortDate** in their **Format** member. The default format for any strings is **Text** and so it is used in printing these fieldnames. **Text** does nothing but prints the material as it is. It can take a **font** argument which is specified along with each field name. The font can have a **name**, **size** and a **weight** members. The **name** can be any valid font name which in our example is **Arial**. The **size** determines the font size and the **weight** determines whether the font has to be bold or regular. The **weight** can be up to **1000**. The **Text** also takes **width** or **w** and **justify** arguments. The **width** is in average font width and **w** is in twips and determines how much wide should be the space allotted to this particular member in the print. No, the text won't be stretched to fill the entire width specified. The next item is placed only after that much space. If a **width** larger than the width of the text is specified, then you can specify a **justify** as to whether you want the text to be **left** (default), **right** or **center** aligned in that space. The default **width** is **20** for a data dictionary item. If the width specified is not enough to print the full material contained in the field, the material is printed up to near the end of the width and three periods are printed to indicate that not everything is printed. You can also see an **x** specified which is an argument to the **Horz** container as to where to keep this item on the page.

The second objects of the first two **Vert** are defined as

```
#("Text" " "font: (name: Arial size: 12))x:0)
```

Here the **TextFormat** is specified directly and the matter to print viz an empty string is specified as its argument along with its other arguments. I used this to keep the alignment between the other **Verts** which have two items and those two with only one.

The final **Vskip** helps to leave an empty horizontal line / space of 1/6 of an inch height below each record in **Output()**. This format can take an argument **height** which should be in inches. I could have used **AfterOutput()** also to put this format.

Now the **Header()** for the second **QueryFormat**. This defaults to **false** as **Output()** was not using the default format. All objects in this break are contained in a **Vert**. It contains a **Horz** which contains 4 objects. All objects use **Textformat** along with its arguments as in the previous example and are used to print the column headings in a bolder font for the columns of **Output()**. To keep alignment, the positions of corresponding objects (**x**) are same in **Output()**

and [Header\(\)](#).

The final [HlineFormat](#) in [Header\(\)](#) is used to print a horizontal line after the column heading. This format takes a [width](#) argument, which, if not specified, makes the line horizontally extend between both the page margins. The [thick](#) argument determines the thickness of the line for which the default is [10](#). An [after](#) and a [before](#) arguments determine how much should be left blank above and below the line. Their default value is [120](#). All values are in twips.

The [After\(\)](#) break adds a thick horizontal line once all records are printed.

The return 'pg' statement tells the [Params](#) to start a new page after the [Append](#) method. I returned a page break as all I had to print could be managed with just three breaks of the primary [QueryFormat](#) and no other break was used.

The second [Params](#) viz [PrintIndividualControl](#) is similar to the one explained. I will just touch upon the differences. This uses 3 [Appends](#). The first two uses a [QueryFormat](#) using details tables different from the one in this example. In the second [Append](#)'s [Output\(\)](#) I have specified a field [hivdiagnosis](#) with a [width](#) argument in addition to the usual ones. This field's data dictionary entry uses a [Wrap](#) as [Format](#). It is similar to [Text](#) except in that if the matter cannot be contained in the [width](#) specified, it is wrapped to the next line. The third [Append](#) adds a [Vskip](#) and an [Hline](#)

The third [Params](#) - [Print_ListControl](#) , I leave it to you to learn yourselves. I made it with the help of Tracy.

Related formats

The other formats which are available in the `std_lib` are as follows:

1. These formats can output only a field. These can be used in a data dictionary entry as well.
 - [TextFormat](#)
 - [WrapFormat](#)
 - [BooleanFormat](#)
 - [CheckBoxFormat](#)
 - [CodeFormat](#)
 - [CheckmarkFormat](#)
 - [EuroNumberFormat](#)
 - [NumberFormat](#)
 - [OptionalNumberFormat](#)
 - [ImageFormat](#)
 - [InfoFormat](#)
 - [IdFormat](#)
 - [PasswordFormat](#)
 - [UOMFormat](#)
 - [WrapGenFormat](#)

2. Those which are used to output more than one field:
 - Hfields
 - Vfields
 - RowFormat
 - GridFormat
 - AddressFormat
 - RecordFormat
 - WrapItemsFormat
3. Those which are used to maintain layout of the report:
 - VertFormat
 - HorzFormat
 - VskipFormat
 - HskipFormat
 - HfillFormat
 - VfillFormat
 - RectFormat
 - VlineFormat
 - HlineFormat
4. Also there are more formats like [TotalFormat](#), [GrandTotalFormat](#)..... which I think are used by [QueryFormat](#). I am not sure whether they are useful in a report of my kind.

Reference

Users manual ->Getting Started -> Create a Report

Users manual ->Going further -> Report Parameters

Users manual ->Going further -> Report Control Breaks

Users manual ->Going further -> Creating a Form Report

Users manual-> Reports-> Reference -> QueryFormat

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=852

http://www.suneido.com/forum/topic.asp?TOPIC_ID=846

http://www.suneido.com/forum/topic.asp?TOPIC_ID=821

http://www.suneido.com/forum/topic.asp?TOPIC_ID=818

http://www.suneido.com/forum/topic.asp?TOPIC_ID=813

http://www.suneido.com/forum/topic.asp?TOPIC_ID=798

http://www.suneido.com/forum/topic.asp?TOPIC_ID=787

http://www.suneido.com/forum/topic.asp?TOPIC_ID=772

http://www.suneido.com/forum/topic.asp?TOPIC_ID=751

http://www.suneido.com/forum/topic.asp?TOPIC_ID=708

Chapter 9 - Making a standalone

Now, we have almost all components of my little program. We will see how to bring all the different pieces together and make a standalone executable out of that.

How to do that?

The different ways that Suneido provides are:

- Make a **Book** and put the name of each one of the 'working' library items in one item of the **Book** each. You have to make as many items in your **Book** as the number of 'working' items you have in your library. You will be using the **BookEdit** tool from Workspace for this. I find **Book** not good for making program interfaces. The details regarding making a **Book** are well documented in the Users manual. If you want to know about this method, check the Users manual.
- The method I chose was to make my own composite **Control** and then make a persistent window out of that. That will be taken up in detail.
- Once you make a **MycompositeControl** or a **Book** for your program, you also have the option of not making a persistent set, but to provide a parameter to the Suneido command or shortcut that starts Suneido. The parameter can be a text file name which carries the commands listed below or can be the commands directly. The commands will be similar to:

```
Use("mylib");  
Window(Mycomposite, exitOnClose:)
```

Note that, though your composite control has a **Control** as its name, it is omitted in the **Window** command.

What exactly does that mean?

Any user interface in Suneido is a **Control**. A complicated **Control** (which I was calling as composite **Control**) which holds and manages many other **Controls** is made using **Controller** and should be named with **Control** as its ending.

Controller can be provided with 4 groups of arguments:

1. **Title** which is a string and hence written inside quotes. This string appears in the title bar of the user interface window left aligned.
2. **Commands** is provided as an object containing member Objects. Each member Object can have "*commandname*" which need not be enclosed in quotes. If you want spaces to be displayed in the command name, then substitute '_' (underscores) in place of them over here. A shortcut key is again a string. If you do not want a shortcut key, specify an empty string. The text to be displayed in the status bar is the next member, again inside quotes. The toolbar bit map picture to be used can then be specified next says Users manual. I think this again is specified as string. These toolbar bitmaps are built in. They're listed in **Resources > IDTB** in stdlib. A valid **Commands** option could be -

```
(Users_Manual, "F1", "Open the Suneido Help", Help)
```

- Now a **Menu** can be specified if you want one. This again is a single object with member objects. Each object can have as many strings as the number of commands you want in one menu. You should use the names as specified in the **Commands**, but with the underscores as true spaces. The first string is taken as the first level(main) menu button name and the rest of the commands as submenu names. Before the letter in the name of the menu button (same as that specified in the **Commands**) which you want to make as the hot key for the command, you can put an ampersand (&)as well.
- The **Controls** is again a single object which holds other **Controls** as its members. If you want a toolbar, you should specify an object with **Toolbar** and the commands (its arguments) you need separated using commas under **Controls**. Note that the controls are written without the **Control** in their name as in -

```
Controls: ( Vert ( Toolbar, Cut, Copy, Paste) Editor )
```

- You have to specify what should be done when a command is called using menu or toolbar or shortcut keys. For this you have to define command methods with names as **On_commandname**. The underscore in command names are preserved here. A probable command method (again from Users manual) is:

```
On_Users_Manual ()
{
    BookControl ("suneidoc")
}
```

What more?

It is not enough that we bring all that we want together. We (rather the end users for whom the program is designed) should be able to use this final composite **Control** with the click of a mouse rather than opening Suneido and typing in the WorkSpace the controls name.

Suneido allows us to create what is called a persistent set using the command **PersistentWindow** for the purpose of making a stand alone. A **PersistentWindow**'s position and window size are stored and then restored when it called a second time. They are saved in the **persistent** table. You have to define or make a persistent set only once and after that you can use it by providing the name of the set as an argument to the command Suneido. Once the set is running, none of the IDE windows like WorkSpace, LibraryView ... is available to the user, only the control specified while making the persistent set. When you close the Persistent Window, Suneido is shut down as well. The usage of the command for creating persistent sets is

```
PersistentWindow(#(Name_of_control), newset: "nameofset")
```

What did I use?

If you look back, I defined my **Access** as a **Control** as well as my reports. I did so, so that I could use them inside a **Controller** (which handles only other **Controls**). My first composite **Control** was **Print_AllControl** the code for which is -

```
Controller
{
    Controls: (Vert
                (EtchedLine)
                (Print_Individual)
```

```

        (EtchedLine)
        (EtchedLine)
        (Print_Contacts)
        (EtchedLine)
        (EtchedLine)
        (Print_List)
        (EtchedLine)
    )
}

```

You can see that I have put my three reports inside a `Vert` separated using `EtchedLines`. Observe that there are no `Title` or `Menu` or `Toolbar` or `Commands` specified. The only `Control` which I have not talked till now is `EtchedLineControl` which just makes a three dimensional line laid horizontally. I am not aware of any arguments it take. It fills the whole width of the window on resizing. The effect of this `Print_AllControl` can be seen if you run it directly from `LibraryView` or from `WorkSpace` using the command `Print_AllControl()`. It brings all three of my report windows in a single window.

Again, I made an `Hiv_WindowControl` which has a `Title` defined and one `Control` viz `TabsControl` specified in the `Controls` argument. The arguments for `Tabs` are objects and I have given two `Controls` as its arguments - the `Print_AllControl` and the `Hiv_AccessControl`. Note that in each member object of `Tabs`, there is `Tab` specified as a string. This is the label to be displayed on the top of the tab that carries that particular `Control`.

And to make the persistent set, I used the command -

```
PersistentWindow(#(Hiv_Window), newset: "hivset")
```

from the `WorkSpace`. And I created a shortcut to `Suneido` with `hivset` as the parameter, the directory where `Suneido` is installed as the working directory and renamed it as `Hivclinic`. Whenever I need to start `Hivclinic`, I double click on this shortcut and my little program pops up as a window that uses two tabs - one showing the `Hiv_AccessControl` and the other showing the `Print_AllControl` (together, the `Hiv_WindowControl`).

For distributing the program, you can make a fresh install, load the library you want, create the tables you want, run the `Persistent` command once, destroy unnecessary tables like `suneidoc`, compact it and use the database for inclusion in a zip file or in making an installation executable using a third party software.

Reference

Users manual ->Getting Started -> Create a Book

Users manual ->Getting Started -> Running Standalone

Users manual ->User Interfaces -> Introduction

Users manual ->User Interfaces -> Controls

Users manual ->User Interfaces -> Commands

Users manual ->Implementation -> Writing a Controller

Users manual ->User Interfaces -> Reference->PersistentWindow

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=829

http://www.suneido.com/forum/topic.asp?TOPIC_ID=805

http://www.suneido.com/forum/topic.asp?TOPIC_ID=761

Chapter 10 - Running Suneido over a network

Suneido is designed to work in a client - server basis over a network. Let us see how to do this.

What exactly does that mean?

Suneido can be installed in one computer of a network and can be set up to work as the server. Working as a server, the program does not do visible work. It just makes the database available for the clients. The client(s) from any other computer of the network can access the database served and do any specified work. The advantage of the client server capability of suneido is that working from different places, data can be entered into one database.

How to do that?

Though the client server can be made to run in one computer itself, as it offers no particular advantage, it should be understood that we are talking about a local area network. The LAN can be of any Windows platform from Windows 95 to XP. But the author does not recommend Windows 95 and 98. I ran the client server in peer to peer LAN where all computers used Windows 98. You can install Suneido in any computer in a peer to peer LAN, but probably you will install it in the server computer (server for the LAN) in a NT model LAN. The computer in which you install suneido will be the server for Suneido.

To run Suneido as a server, you have to start it with a command line option `-server`. Once you start suneido with this command, you won't see the usual IDE windows, but only a small blank window titled Suneido. The `-server` command should be followed by another word, denoting the name of a text file which contains the commands that will tell the server which all libraries to use. If you do not specify the text file name, then only the `stdlib` is served by the server and no other libraries are available to the clients. The text files are given the extension of `go` by convention, though anything is legal. The go file typically contains the command `Use`, which is a function that accepts a library name as its argument. It just makes the specified library available. We do not want to do all this exercise over and over. So, we create a short cut to suneido with the `-server` followed by the `textfile.go` added to end of the path in the *Target* parameter of the shortcut. Remember to specify the path to the directory where suneido is installed in the *Start in* parameter of the shortcut. You click on the shortcut and your server is ready. Note that there is no real need for the go file. You can enter the commands directly in place of the file name. But a file name would be better when you have more than one command.

For the client, again a shortcut has to be made. This time, the commandline option to be given is `-client`. Following the `-client`, you have to specify the IP address of the machine in which the server is situated. If you are making a shortcut for client in the same machine in which server is located, then a special IP address of `127.0.0.1` is to be used in place of the IP address of the server. After the IP address, you have to enter the command (usually the name of a persistent set) that the client has to execute. If you omit the IP address and / or the command, the client assumes that the server is in the same computer and uses the IDE set. Usually the client has to be in another computer. You have two options - to install just the executable suneido.exe in each computer, make a shortcut to that executable file with the IP address of the computer in which the database file is located, followed by the persistent set you want to execute. The better option is to make a shortcut to executable in the server computer itself. To do this, you have to make the folder in which you have installed the server suneido shared with rights to read and write. The advantage of the second method is that you needn't go and install suneido

in each computer, but need only a shortcut. Updating suneido is also easy as there will be only one copy to update.

What more?

The `-client` accepts along with it a command name or a persistent set's name as told earlier. If you specify a persistent set and is accessed by different users, then the instant one user changes the window size, the same happens in other places as well. This is because all users are logged as *default* in suneido database. To overcome this, preferably use a `Login (persistentset)` function. Using the `Login` ensures that each user is registered separately. The `Login` function supplied with `stdlib` is rather simple. It doesn't allow for a password. The `Login()` supplied in accounting library is better.

Both the `-server` and `-client` can have before them a `-port` command, followed by the port number. This is not needed usually. It specifies which TCP/IP port suneido should use for communication. By default `3147` is used. You will need this if you intend to have two suneido servers in one computer running simultaneously.

What did I use?

I decided to go without a go file. The first thing was a shortcut for the server which had as its *Target* parameter -

```
C:\Suneido\suneido.exe -server Use('hiv_lib')
```

The next step was making a shortcut for the client. I found out the IP address assigned to my server computer which has the name *Server* in the LAN from the properties tab of Network Neighborhood icon and it was `180.132.50.1`. You can use the tool `ipconfig` from a command prompt for finding out the IP address. Now the shortcut was made with -

```
\\Server\Suneido\suneido.exe -client 180.132.50.1 Login('hivset')
```

My little program is now ready to run over a LAN with these small steps! All these steps can be accomplished with the third party software you use in making a distribution package.

Reference

Users manual ->Getting Started -> Running Client Server

Users manual ->Introduction -> Command Line Options

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=1051

Note:

When you run suneido in LAN make sure that the antivirus software and firewall programs are not blocking suneido. Otherwise you will waste your time searching for network errors.

Chapter 11 - Maintenance of Suneido

Software is soft - It is damageable. So is the data contained in any program. We will see the steps in the guarding against loss and recovering from crashes.

What exactly does that mean?

The maintenance involves backing up your data and maintaining the size and integrity of the database file.

How to do that?

Suneido provides us with a bunch of command line options which are run from the DOS prompt as well as equivalent commands that can be used from inside the program to maintain backups. All command line options are run from the directory containing Suneido and its db file.

The relevant command line options are:

- -check
- -rebuild
- -compact
- -copy
- -dump
- -load

What more?

The **-rebuild** option is run from a DOS prompt as **suneido -r** or as **suneido -rebuild**. This does not have a corresponding command that can be run from the IDE. This command builds a new db file from the existing one(probably a damaged one). The existing db is renamed as **suneido.db.bak**. Everything up to the last good commit will be kept.

The **-check** option is used to check whether the db is O.K or not. You can run this to check whether your db is good after an improper shutdown as well as after a rebuild. This command does not have a shorthand notation or an IDE command.

After a 'crash' or an improper shutdown, Suneido automatically runs the **-check** and shows in a dialog the result and gives you the option to **-rebuild** whether or not the check found an error. My experience tells me not to believe the automatic starting of **-check** or the result it shows. I will advice you to do the **-rebuild** whether or not suneido asks for that automatically and in spite of the result given by **-check**.

That was what you would do in case some thing goes wrong. What should you do in anticipation of a calamity if **-rebuild** is not successful ?

The **-compact** option allows you to maintain the size of the db file by deleting all the history which it stores. What this history exactly is, is not known to me. You need to do this to make sure that size of the db does not go beyond what your processor can handle. The db is a memory mapped file and so consumes a significant amount of memory (so says its author). Again this option does not have a shorthand notation or an IDE counterpart.

The `-copy` option helps you make a copy of the db file. Its difference from `-rebuild` is that it does not check the integrity of the db, the `-copy` can give the copy any name you specify and does not keep any history. The default name for the copy is `suneido.db.copy` and can be any name you specify after the `-copy`. There is no need to have any quotes around the file name. The IDE counterpart is `Backup("filename")` for which an essential `filename` argument (the filename that is to be given to the backup) should be specified inside quotes inside the parentheses.

The `-dump` is the command that puts all the data from the db as a suneido file with the extension `su`. The structure of the `su` file is not published. My understanding is that `-dump` (shortened as `-d`) does not do anything to the db structure or its history. It just dumps the contents of the db in this particular format under the filename `database.su`. The advantage in dumping this way is that you can load it exactly as it is into another db, replacing its whole contents. The size of `database.su` is lesser than the db itself.

The `-dump` can accept an argument - a table's name. The table should exist in the db, otherwise an error occurs. If you provide a tablename, only that table is dumped in `su` format. Again, the tablename need not be inside quotes.

`Dump()` is the IDE counterpart of `-dump`. The table name should be specified inside the parentheses inside quotes. If an empty parentheses is provided, then the entire db is dumped.

The `-load` is similar to the corresponding `-dump`, but puts the specified `su` file (or implied `database.su`) into the database rather than taking it out of the db. It does not have a corresponding `Load()`.

What is the difference between the options ?

The major difference between the commands is whether they can be run from inside the program or from the command prompt when the db is not in use. Also the 'format' in which data is output is significant.

What did I use?

Of all the options to make a complete copy of db, I like `-rebuild`, though it is not intended for backup primarily. I like its error checking and correcting capacity.

But the disadvantage is that you have to use the command prompt every time you want to do that.

To overcome the difficulties I made a bat file which contains the following commands-

```
d: [ change to the drive having suneido]
cd d:\suneido [change to the directory having suneido]
start/w suneido -d hiv_main [dump the tables one by one. The start/w makes the DC
start/w suneido -d hiv_investigations
start/w suneido -d hiv_contacts
start/w suneido -d hiv_admission
start/w suneido -rebuild
start/w suneido -compact [ compact the new db]
```

Related commands

There are also the commands `DumpText` and `LoadText` to dump a table in CSV format from the program itself.

Also, you can use the `DL_lib` provided by Roberto Artigas Jr. to dump tables in formats other

than that provided by Suneido.

You also have the Export and Import routines (readily available from Access and Browse) to export and import records from different formats.

Also, you can see the use of New Release Tool by Roberto Artigas Jr. which can be modified for backup purposes.

You also have Task Scheduler that runs from inside Suneido that can execute commands as per a schedule.

Reference

Users manual ->Introduction->Command Line Options

Users manual ->Introduction->Crash Recovery

Users manual ->Tools->Scheduler

Users manual ->Language->DumpText

Users manual ->Language->LoadText

Users manual ->Database->Dump

Users manual ->Database->Backup

Suneido Forum Topics

http://www.suneido.com/forum/topic.asp?TOPIC_ID=843

http://www.suneido.com/forum/topic.asp?TOPIC_ID=828

http://www.suneido.com/forum/topic.asp?TOPIC_ID=745

http://www.suneido.com/forum/topic.asp?TOPIC_ID=712

Chapter 12 - Improve your code

Always there is scope for improvement. And probably my code is not error free. This section will continue in accordance with the feedback I receive on my code and ways to improve the code. These are arranged in no specific order.

Points for improvement

- Indent your code. It does not affect the functioning of the code, but adds to the readability. I have not implemented it (I am lazy ;)). Thanks to Jeff, I will try later.
- It is usually a good idea to not use specific [Skip](#)'s to align controls. The problem is that if you change the prompts or fonts, or translate the language, then they will not line up any more. Whereever possible, it is better to use [Vert](#) or [Form](#) to line things up. (Andrew)
- In control layouts, it is not necessary to enclose field names in parentheses. (Andrew)
- Calculated rule fields only have to be created in the database (i.e. with capitalized names) if you need to refer to them in queries, e.g., in [where](#) or [sort](#). Otherwise, you can put them on screens or reports without creating them in the database.(Andrew)
- Use the same name for a [key](#) on a table and a foreign key to the same table. This makes some of the header detail codes for user interface and reports a bit easier since only one field name is used. (Jeff)
- The [DumpText](#) and [LoadText](#) commands are not recommended. The [Import](#) and [Export](#) routines are probably better. (Andrew)

Epilogue

So, we reach the end of my endeavor. How was it? I would like to know from you, especially the flaws, both in my code as well as in the manual. You can mail me to ajithramayyan@yahoo.co.in. If you find that something is not working or returns an error, do find some time to inform me or the Suneido forum. The improvements section is also expected to grow.

Once again, Thanks to Suneido team for a great piece of work and for their invaluable support.

Ajith.R