Tally

POWER OF SIMPLICITY

TDL Reference Manual

The information contained in this document represents the current view of Tally Solutions Pvt. Ltd., ('Tally' in short) on the topics discussed as of the date of publication. Because Tally must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Tally, and Tally cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. TALLY MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Version: TDL Reference Manual/9.0/August 2012

Preface

Tally Definition Language (TDL) is the development of Tally.ERP 9. This allows the programmers to develop and deploy faster, effective Tally Extensions with ease.

The book, TDL Reference Manual, divided into two sections. First section begins with the Introduction to TDL and focuses on basic concepts of TDL i.e, TDL Components, Symbols used in TDL, Dimensions and Formatting, Usage of Variables, Buttons and Keys.

Thereafter the emphasis is on the coverage of core concepts of Objects, Methods and Collections, Actions and UDF creation. After gaining a reasonable amount of depth and confidence in understanding the above, the focus of the book progresses towards the application of all covered topics i.e., the creation of various types of Reports, Printing and Voucher/Invoice customisations.

Second section devoted to a detailed discussion of TDL language enhancements for Tally.ERP 9. This section describes the new features, Writing Remote Compliant TDL Reports and User Defined Functions respectively. The What’s new section gives an insight about the enhancements in the latest Tally.ERP 9 Releases.

This book is for anyone who wants to explore TDL as a development language of Tally and how to write TDL programs effectively. Absolutely no previous TDL experience is necessary. Even advanced users will find this book useful, as the changes to TDL are dealt from the developers and the user's point of view.

You will enjoy reading this book, as it is rich in concepts.

Happy programming folks!

Contents

Section I. TDL – The Development Language of Tally.ERP 9

Section II. TDL – Language Enhancements

Section I

TDL – The Development Language of Tally.ERP 9

Tally Definition Language – An Introduction

Introduction

Tally Solutions has been in the business of providing complete business solutions for over 20 years to MSME (Micro, Small and Medium Enterprise) and to a large extent for LE (Large Enterprise) businesses. With over 3 million users in over 100 countries Tally, the flagship product continues to be the preferred IT solution for a majority of businesses every year.

Tally – the flagship product (which started as a simple bookkeeping system, 20 years ago), is today a comprehensive, integrated solution – covering several business aspects of an enterprise. These include Accounting, Finance Management, Receivables/Payables, Inventory Accounting, Inventory Management, BoM based manufacturing inventory, multi-location/multi-currency\multi-unit handling, Budgets and Controls, Cost and Profit Centres, Job Costing, POS, Group Company consolidations, Statutory Taxes (Excise, VAT, CST, TDS, TCS, FBT, etc), Payroll Accounting, and other major and minor capabilities. It has served as an ERP for small enterprises over the past 12 years.

With the introduction of Remote Access, Remote Authentication, Support Centre, Central Administration and Account Management inherently supported in the product it can be formally labeled as Tally.ERP 9. With this capability, it is possible that the owner or an authorized user will be able to access all the reports and information from a remote location. With each forthcoming release subsequent to Tally.ERP 9 Release 3, additional capabilities will be delivered to cater to large business enterprises. The major functional areas in Tally are:

Order to Payment (Purchase Processes)

Simple (Cash Purchase) to Advanced Purchase Processes - including Ordering, Receipting, Rejections, Discounts, etc.

Order to Receipt (Sales Processes)

Simple (Cash Sales) to Advanced Sales Processes - including Orders Received, Delivery, Invoicing, Rejections and Receipting, POS Invoicing at Retail.

Material to Material (Manufacturing Processes)

Simple to Multi-step material transformations, Discrete and Process Industry cycles, Work in progress and valuations.

Payroll

Simple to Complex Payrolls – including working with different Units of Measures (e.g. Job rates). Statutory compliances, their specifications and usage.

MIS

A complete set of reports for Business requirements are as follows:

Financial, Inventory, MIS & Analysis. Budgeting & Controls with advanced classification and filtering techniques. Group Companies and multiple consolidation views. Cross-Period Reporting, Forex handling, Bank Reconciliation. There is also an Export option to port data into other applications (e.g. Spreadsheets) for additional manipulation.

Statutory Compliance

The Compliance Requirements and related configurations in Tally.ERP 9 are as follows with regard to the implementation of :

- Direct Taxes: TDS/TCS, FBT
- Indirect Taxes: Excise, Service Tax, VAT, CST

Enabling Environment for Remote - Tally.NET

Tally.NET is overall responsible for the Remote Access Services. It allows:

- Remote Access - It is now possible for an authenticated user to access Tally.ERP 9 from any computer system.
- Tax Audit Tools - The CA community will now be able to deliver affordable services to clients addressing their Security and Privacy concerns.

1. Tally Definition Language

Tally Definition Language is the application development language of Tally. TDL is developed to provide the user with flexibility and power to extend the default capabilities of Tally and integrate them with the external applications. TDL provides a development platform for the user. The entire User Interface of Tally.ERP 9 is built using TDL. TDL as a language, provides capabilities for Rapid Development, Rendering, Data Management and Integration.

TDL is an Action driven language based on definitions. It emphasizes strongly on the concept of reusability. It comprises of Interface and Data objects. Interface Objects mainly determines the behavior of the product in terms of user experience. Data objects are mainly used for data persistence in the Tally Database.

Any user of Tally.ERP 9 can learn TDL and develop extensions for the product. The entire source code of the product is available as part of the Tally Development Environment i.e. with our product Tally Developer.

1.1 Comparison with other Languages

Today there are many languages in the world which are used to develop applications.These languages are developed keeping some specific areas of application in mind. Some languages are good for developing front end applications while others may be good for writing system programs. The various categories of languages available today are as follows:

Low Level Languages

Low level Languages are languages that can interact directly with the hardware. They comprise instructions which are either directly given in computer-understandable digital code or in a pseudo code. These languages require very sound knowledge in hardware. For e.g. Assembly language or any native machine language.

Middle Level Languages

Middle Level Languages consists of syntax, rules and features just like high level languages. However they can implement low level languages as part of the code. For e.g., C, C++, etc.

High Level Languages

High level languages are very much like the English language. They are easy to learn, program and debug. High level programming languages are sometimes divided into two categories: Third Generation and Fourth Generation languages.

Third Generation Languages

Most High Level languages fall in the category of Third Generation Languages. Third Generation languages are procedural languages i.e. the programmer specifies the sequence of the execution and the computer strictly follows it. The execution starts from the first line of the code to the last line, taking care of all the control statements and loops used in the program.

Fourth Generation Languages

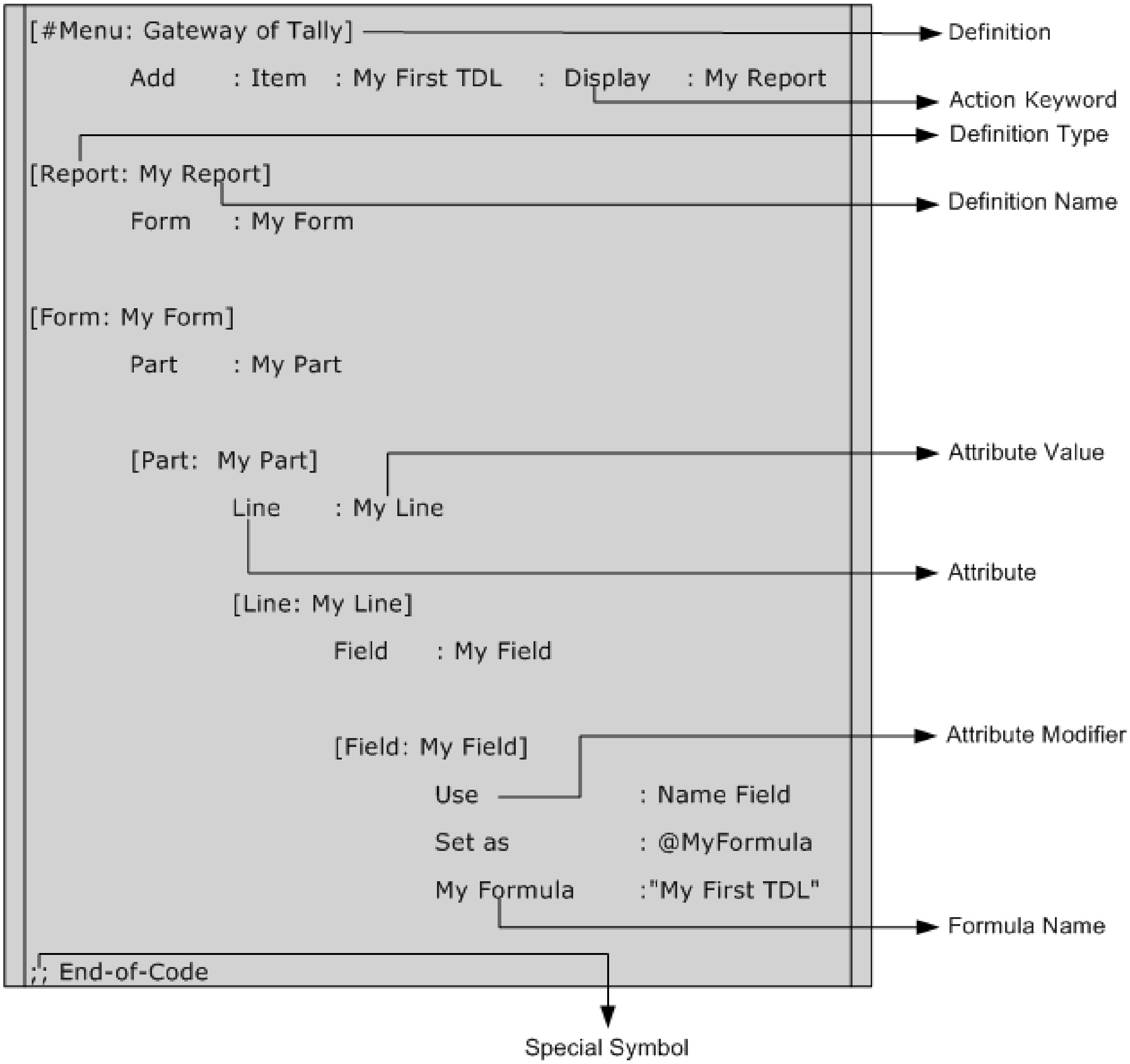
There is no clear cut definition for the Fourth Generation Languages (4GL). Normally the 4GL are high level languages which require significantly fewer instructions to accomplish a task. Thus a programmer is able to quickly develop and deploy the code. Most 4GL are non procedural languages.

E.g: Some 4GL are used to retrieve, store and modify data in the database using a single line instruction whereas 4GL use report generators to generate complex reports. It is sufficient to specify headings and totals using the language and the report is generated automatically. Certain 4GL can be used to specify the screen design which will automatically be created.

On having understood the categorization of computer languages, TDL can be categorised as a Fourth Generation High Level Language. The capabilities which TDL provides to the users is much more than what other 4GL languages provide. This may extend to meeting specific purposes like database management, report generation, screen design etc. TDL is a comprehensive 4GL language which gives tremendous power in the hands of the programmer by providing data management, complex report generation and screen design capabilities using only a few lines of code, leading to rapid development. Let us now analyze the features in detail which help us in understanding and appreciating the capabilities provided by the development language of Tally i.e., ‘TDL - Tally Definition Language’.

2. The TDL Program - At a Glance

Before we discuss the capabilities and features of TDL in detail, let us have a look at the basic TDL program. The following figure describes all the components in a TDL Program. The description, usage and detailed explanation of each component will be taken up in the subsequent chapters.



```
[#Menu: Gateway of Tally]
        Add      : Item   : My First TDL   :  Display    : My Report

[Report: My Report]
        Form     : My Form

[Form: My Form]
        Part     : My Part

        [Part:  My Part]
                Line     : My Line

                [Line: My Line]
                        Field     : My Field

                        [Field: My Field]
                                Use             : Name Field
                                Set as          : @MyFormula
                                My Formula      :"My First TDL"
;; End-of-Code
```

Figure 1.1 TDL Components

3. TDL Capabilities

Rapid Development

TDL is a language based on definitions. It is possible to reuse the existing definitions and deploy them. This is a language meant for rapid development. It is possible to develop complex reports within minutes. The user can extend the default functionalities of the product by writing a code consisting of a few lines.

Multiple Output Capability

The same language can be used to send the output to multiple output devices and formats. Whenever an output is generated, it can be displayed on the screen, printed, transferred to a file in particular format and finally mailed or transferred to a webpage using Http protocol. All this is made possible just by writing a single line of code. Just imagine the technology used to develop the platform that such a complex task is developed and implemented using only a few lines.

Data Management Capability

As we have discussed earlier, the data is stored and retrieved as objects. There are a few internal objects predefined by the platform. Using TDL, it is possible to create and manipulate information on these with ease. Suppose, an additional field is required by the user to store information as a part of the predefined object, then that capability is also provided, i.e. by using TDL the user can create a new field and store a value into it which can be persisted in the Tally.ERP 9 database.

Integration Capability

To meet the challenges of the business environment it becomes absolutely mandatory to share information seamlessly across applications. Integration becomes a crucial factor in avoiding the duplication of data entry. The Tally.ERP 9 platform has a built in capability of integrating data with other applications.The following are the different types of integrations possible in Tally.ERP 9.

- Tally.ERP 9 to Tally.ERP 9 using Sync
- Tally.ERP 9 to external applications in various data formats
- External DB to Tally.ERP 9 using XML and SDF formats
- Tally.ERP 9 DB to external applications using ODBC
- External DB to Tally.ERP 9 using ODBC

4. TDL – Features

Definition Language

A definition language provides the users with ‘Definitions’ that can be used to specify the task to be performed. The user can specify the task to be performed, but has no control over the sequence of events that occur while performing the specified task. The sequence of events is implicit to the language and cannot be changed by the user. TDL works on Named Definitions, which means, that every definition should have a name and that it should be unique. TDL has User Interface Objects like Reports, Forms, Parts, Lines and Fields as definitions.

TDL can define Reports, Menus, Forms, and so on, but the Definitions will not have any relevance unless they are used. Definitions are deployed by use, not by existence.

TDL is based on concepts pertaining to Object Oriented Programming. This language has been created for reusability. Once a definition is created, it can be reused any number of times. Besides the reusing capability, the user can also add new features along with the existing definitions.

Tally.ERP 9 has a singular view of all the TDL Definitions, which means the Tally.ERP 9 executable reads TDL (user defined and default) as one program. On invoking Tally.ERP 9, all the default TDL files of TDLServer.DLL will be loaded. The user TDLs will be subsequently loaded as specified in Tally.ini.

Non Procedural Language

Most of our programming experience has been in dealing with a procedural language where we define a sequence of actions to define the sequence of events that take place. The entire control is with the programmer. The programmer is able to determine the start and end-point of the program. The programmer cannot control the sequence. All the sequences are implicit in the program. The programmer cannot write his/her own procedure. The platform provides a set of functions for the TDL programmer.

Action Driven Language

The programmer can only control as to what happens when a particular event takes place. While interaction, the user can select any sequence of action. Based on his/her action a particular code gets executed.

Rich Language

TDL is a rich language, that refers to a list of functions, attributes, actions etc. which are provided by the platform. It is possible to develop a complex report or modify the existing one within no time. Imagine how many lines of code would be required if a simple button were to be added using a traditional programming language.

Flexibility and Speed

The architecture of the software and the language provide extraordinary flexibility and speed. Speed in this regard refers to the speed of deployment. With Tally.ERP 9 the deployment is extremely rapid.

Tally.ERP 9 is flexible enough to change its functionality based on the customer’s business requirements. Most of the time customer specific requirements may seem like a majority of functional changes that have to be done but they may only be minor variations of the existing functionality which can be done within no time.

Learning Outcome

- The major functional areas of Tally.ERP 9 are Purchase processes, Sales processes, Manufacturing processes, Payroll, MIS, Statutory Compliance and Tally.NET.
- TDL is the application development environment of Tally.ERP 9.
- TDL is a Fourth Generation High Level Language.
- TDL is not only a definition language but also a non-procedural action driven language.

TDL Components

Introduction

As we have already discussed in the previous lesson, TDL is a language based on definitions. It is an action driven language i.e. whenever the user performs an action a particular segment of code gets executed. In this lesson we will provide an overview and basic functionality of each component involved in a TDL program.

1. Writing a Basic TDL Program

TDL allows us to define tasks in standard English statements. This simplifies the process of definition, allowing even a person without any programming language background to work on TDL. The TDL statements required to perform a particular task can be created in a file using IDE provided by Tally.ERP 9 such as Tally Developer. Such a file is called TDL file. Let us begin our discussion by writing the basic TDL program.

The Steps to create a TDL Program

- Open any ASCII text editor such as notepad or use the IDE Tally Developer provided by Tally.ERP 9 .
- Create a new file.
- Type TDL statements in the file.
- Save the file with a meaningful name an extension as applicable to the editor. The editor can save the file with an extension '.txt', '.tdl'
- The file can be compiled into a file with an extension .tcp (Tally Compliant Product). It is possible to compile the file for a particular Tally serial number.
- It is possible to run all files i.e., (.txt,.tdl and .tcp) in Tally.ERP 9.

1.1 Specification of TDL Files

There are two ways of implementing the TDL code:

- Specifying TDL files in Tally.ini (Configuration Settings File)
- Specifying TDL file through Tally.ERP 9 application configuration screen

Specifying TDL files in Tally.ini

The path of the TDL program has to be included in the Tally.ini file, using a parameter called 'TDL'. If the parameter 'User TDL' is set to No, Tally.ERP 9 will not read any TDL parameters specified in Tally.ini file.

Syntax

```
        User TDL    = Yes
        TDL         = <Path\filename> with extension
```

Example

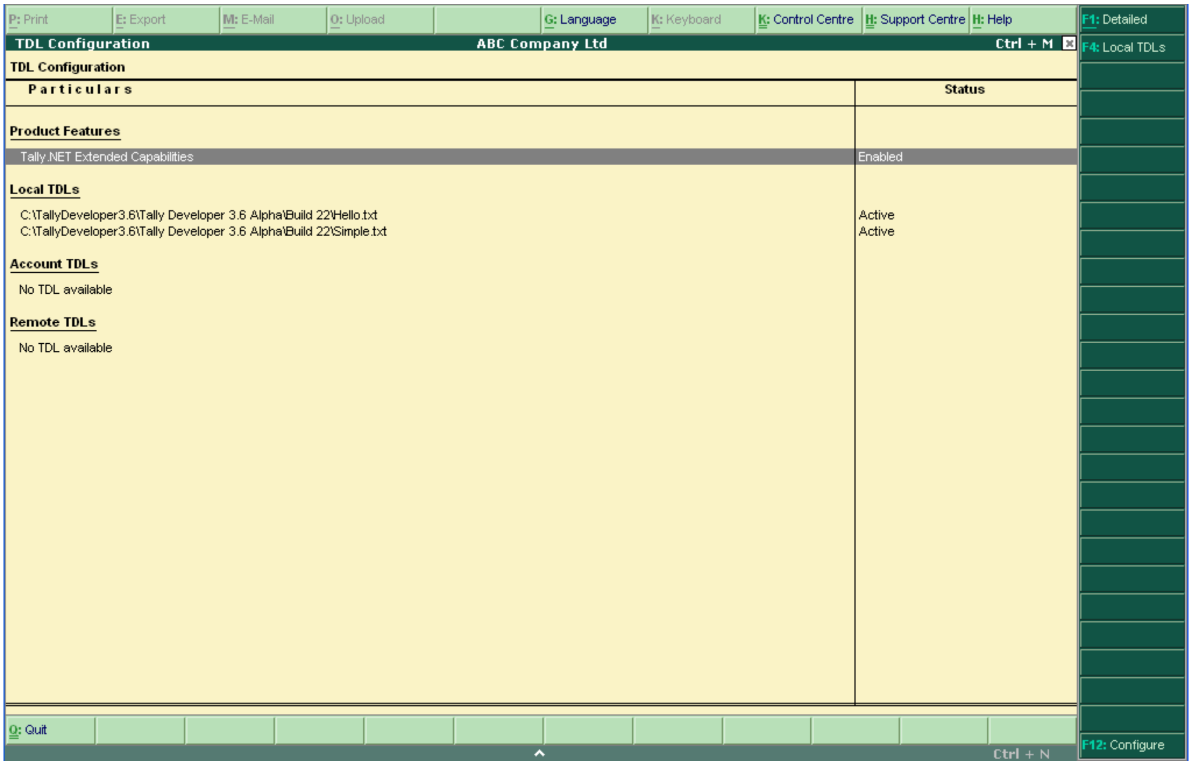
```
    User TDL = Yes
    TDL      = C:\Tally.ERP 9\MyReport.tcp
                     or
    TDL      = C:\Tally.ERP 9\MyReport.txt
```

When Tally.ERP 9 starts, it looks for a file named 'MyReport.tcp' or 'MyReport.txt' in the directory C:\Tally.ERP 9. On loading the default TDL files into memory, Tally.ERP 9 reads and loads every TDL file mentioned in Tally.ini into memory before displaying the first Menu, 'Gateway of Tally'.

Specifying TDL file through Tally.ERP 9 application configuration screen

Alternatively, the TDL file name can be specified in the configuration screen displayed by selecting menu item 'TDL Configuration' from the F12 Configuration menu. In this screen click the button Local TDLs or press F4, set the value Yes for 'Load TDLs on Start up' and specify the **<Path\filename>** with extension in 'List of TDLs to preload on Tally Startup' field.

Following figure shows the TDL configuration screen:



TDL Configuration   ABC Company Ltd

TDL Configuration

Particulars	Status
Product Features	
Tally.NET Extended Capabilities	Enabled
Local TDLs	
C:\TallyDeveloper3.6\Tally Developer 3.6 Alpha\Build 22\Hello.txt	Active
C:\TallyDeveloper3.6\Tally Developer 3.6 Alpha\Build 22\Simple.txt	Active
Account TDLs	
No TDL available	
Remote TDLs	
No TDL available	

Figure 2.1 Specification of TDL files

To load a Default Company in Tally.ERP 9, the 'Load' parameter used is as stated below:

Example

```
Default Companies    = yes
Load                 = 00002
```

Here 00002 is the company folder that resides in Tally.ERP 9\Data. The data path can be specified with the parameter Data.

Example

```
Data      = C:\Tally.ERP 9\Data
```

Restart Tally.ERP 9 whenever there are changes made in the TDL program, so that they can be implemented.

2. TDL Interfaces

We have already seen that TDL is a language based on definitions. When we start Tally.ERP 9 the Interfaces which are visible on the screen are Menu, Report, Button and Table. In TDL specific definitions are provided to create the same.

A Report and Menu can exist independently. A Menu is created by adding items to it while a Report is created using Form, Part, Line and Field. These are the definitions which cannot exist without a Report. TDL operates through the concept of an action which is to be performed and Definition on which the action is performed. The Report is invoked based on the action.

TDL program to create a Report contains the definition Report, Form, Part, Line and Field and an action to execute the Report. A Report can have more than one Form, Part, Line and Field definitions but at least one has to be there. The hierarchy of these definitions is as follows:

- Report uses a Form
- Form uses a Part
- Part uses a Line
- Line uses a Field
- A Field is where the contents are displayed or entered

The Report is called either from a Menu or from a Key event.

3. Hello TDL Program

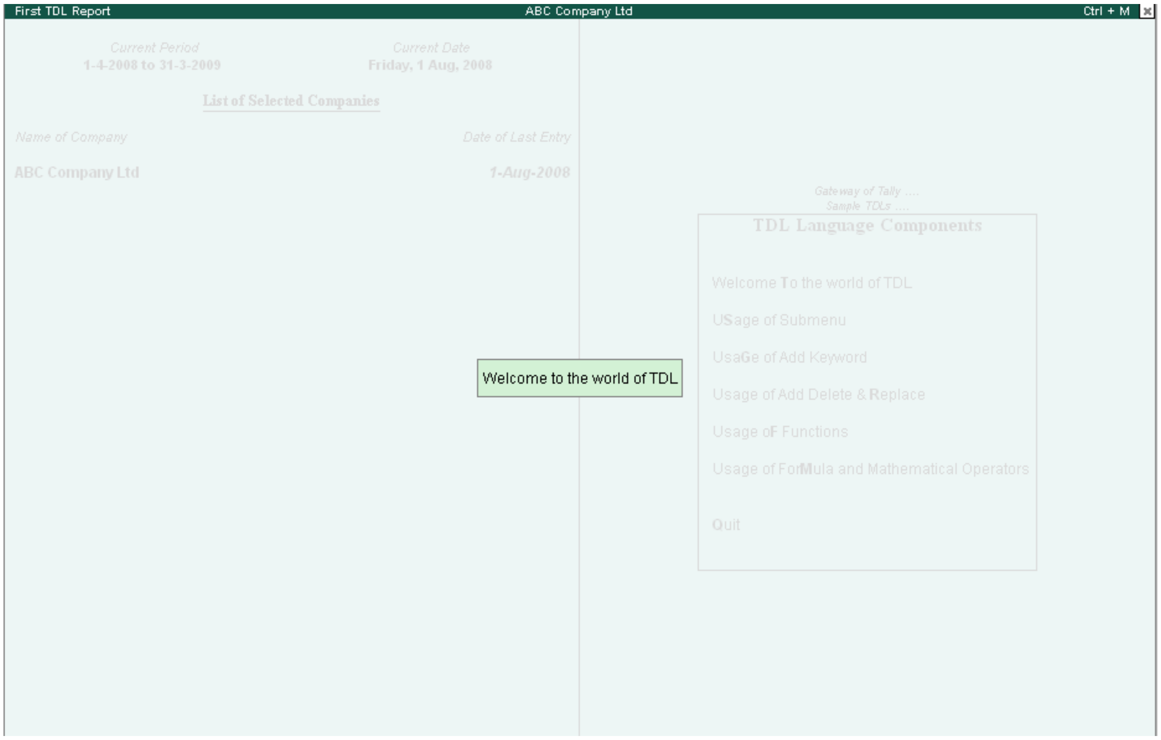
The Hello TDL program demonstrates the basic structure of the TDL. The Report is executed from the existing Menu ‘Gateway of Tally’.

To invoke a new Report displaying the text “Welcome to the world of TDL” from the main Menu ‘Gateway Of Tally’.

```
[#Menu: Gateway of Tally]
   Item : First TDL : Display : First TDL Report

[Report: First TDL Report]
   Form     : First TDL Form

[Form: First TDL Form]
   Parts    : First TDL Part

[Part: First TDL Part]
   Lines    : First TDL Line

   [Line: First TDL Line]
      Fields : First TDL Field

      [Field: First TDL Field]
         Set as   : "Welcome to the world of TDL"
```

The TDL code adds a new Menu Item ‘First TDL’ in the ‘Gateway Of Tally’ menu. When the Menu Item is selected the report, First TDL Report is displayed. The report is in display mode as the action ‘Display’ is specified while adding the menu item ‘First TDL’. The user input is not accepted in this report. The text ‘Welcome to the world of TDL’ is displayed in the Report since it contains only one field.

Figure 2.2 shows the output of the code mentioned above :



First TDL Report | ABC Company Ltd | Ctrl + M

Current Period
1-4-2008 to 31-3-2009

Current Date
Friday, 1 Aug, 2008

List of Selected Companies

Name of Company	Date of Last Entry
ABC Company Ltd	1-Aug-2008

Welcome to the world of TDL

Figure 2.2 Output of Welocme to the world of TDL program

3.1 Executing Multiple Files using Include Definition

Since TDL can span or exist across files, the definition ‘INCLUDE’ provides the convenience of modularizing the application and specifying all of them in one TDL file. It allows the user to include TDL code existing in separate file/files to be included into the current file. ‘Include’ as the name suggests, gives you the ability to include another TDL file into a file, instead of declaring it in Tally.ini separately.

Syntax

```
[Include: <path/filename>]
```

In case the TDL file is in the same directory, give either the filename or give the complete path for the file.

Example

Let us assume we are using two files, sample1.txt and sample2.txt. To run both the files, we have to include sample2.txt in sample1.txt.

```
[Include: sample2.txt]
```

4. TDL Components

The TDL consists of Definitions, Attributes, Modifiers, Data Types, Operators, Symbols and Prefixes, and Functions. Let us now analyze the components of the language.

4.1 Definitions

Tally Definition Language (TDL) is a non-procedural programming language based on definitions. TDL works on named definitions. The biggest advantage of working with TDL is its reusability of definitions. All the definitions are reusable by themselves and can be a part of other definitions. Whenever a change in code needs to be reflected in a program, Tally.ERP 9 must be restarted.

Syntax

```
        [<Definition Type> : <Definition Name>]
```

All definitions start with an open square bracket and end with a closed bracket.

<Definition type> It is the name of predefined definition types available in the platform, e.g. Collection, Menu, Report, Form, Part, Line etc.

<Definition Name> This refers to any user defined name which the user provides to instantiate the definition i.e. whenever a definition is created, a new object of a particular definition type comes into existence.

Example

```
    [Part: PartOne]
```

In the example mentioned above, the type of definition is Part and the name of definition is PartOne.

4.1.1 Types of Definition

The various definitions in TDL are categorized as follows:

- Interface Definitions – Menu, Report, Form, Part, Line, Fields, Button, Table
- Data Definitions – Object, Variable, Collection
- Formatting Definitions – Border, Style, Color
- Integration Definitions – Import Object, Import File
- Action Definitions – Key
- System Definitions

Interface Definitions

Definitions which are used in creating a user interface are referred to as an interface definition. The definitions in this category are Menu, Report, Form, Part, Line, Fields, Button and Table.

Menu: A Menu displays a list of options. The Tally.ERP 9 application determines the action to be performed on the basis of the Menu Item selected by the user. The 'Gateway of Tally' is an example of a 'Menu'. A Menu can activate another Menu or Report.

Report: This is the fundamental definition of TDL. Every screen which appears in Tally.ERP 9 i.e. the input screen or output screen is created using the report definition. A Report consists of one or more Forms.

Form: A Form consists of one or more Parts.

Part: Part consists of one or more Lines.

Line: A Line consists of one or more Fields.

Field: A field is place where the data is actually displayed or entered. The data can be a constant or variable data.

Button: The user can perform an action in three ways i.e. by selecting a menu item, by pressing a key and by clicking on a button. The Button definition allows the user to display a button on the Button bar and execute an action.

Table: The Table definition displays a list of values as Tables. Data from any collection can be displayed as a Table.

Data Definitions

Definitions which are used for storing the data are referred to as a Data Definitions. The definitions in this category are Object, Variable and Collection.

Object: An object is the definition which consists of a data and the associated / related functions, commonly called as methods that manipulate the data. TDL is made up of User interface and Info Objects. Info Objects can be External (user defined) or Internal (platform defined). External or user defined objects are not persistent in the Tally database. It is not possible to create an Internal Object Definition in TDL i.e. they are predefined by the platform . It is possible to perform modifications on it. An object can also further contain an object/objects. A Ledger/Group is an example of an internal object.

Collection: A Collection is a group of objects. Collections can be made up of internal or external objects. These can be based on multiple collections also. We can create a collection by aggregating the collections at a lower level in the hierarchy of objects.

Variables: Variables are used to control the behavior of reports and its contents. The variables can assume different values during the execution and based on those values the application behaves accordingly. The option Plain Paper/Pre-Printed while printing the invoice is an example of a variable controlling the report

Formatting Definitions

Definitions which are used in formatting a user interface are referred as Formatting Definition. The definition in this category are Border, Style and Color.

Style: The Style definition determines the appearance of the text to be displayed by using a font scheme. The Font name, Font style and Font size can be changed/defined using the style definition. In default TDL the pre-defined Style definitions are Normal Bold, Normal Italic and Normal Bold Italic.

Border: This introduces a single/double line as per user specifications. Thin Box, Thin Line, Common Border are all examples of pre defined borders.

Color: The Color definition is used to define a color. A name can be given to an RGB value of color. Once a name is assigned to an RGB color value, it can be expressed as an attribute. In TDL the only color names that can be specified are Crystal Blue and Canary Yellow.

4.1.2 Integration Definitions

Definitions which makes the import of data available in SDF (Standard Data Format) are referred to as Integration Definitions. Import Object and Import File are the two definitions classified in this category.

Import Object: This identifies the type of information that is being imported into Tally.ERP 9. The importable objects can be of the type groups, ledgers, cost centre, stock items, stock groups, vouchers etc.

Import File: The Import file allows the user to describe the structure of each record in the ASCII file that is being imported. The field width is specified as an attribute of this definition.

4.1.3 Action Definitions

The action definition allows the user to define a action when a key combination is pressed. It also associates an object on which the action is performed. The Key definition falls in this category.

Key: The Key Definition is used to associate an action with the key combination. The action is performed when the associated key combination is pressed.

4.1.4 System Definitions

System Definitions are viewed as being created by the administrator profile. Any items defined under System Definitions are available globally across the application. System Definitions can be defined any number of times in TDL. The items defined are appended to the existing list. System Definitions cannot be modified.

E.g. of System Definitions are System: Variable, System : Formula, System : UDF and System : TDL Names

4.2 Attributes

Each definition has properties referred to as 'Attributes'. There is a predefined set of attributes provided by the platform for each definition type. The attribute specifies the behavior of a definition. Attributes differ from Definition to Definition. A Definition can have multiple attributes associated with it. Each attribute has a 'Name'(predefined) and an assigned value (provided by the programmer). A value can be either directly be associated to a given attribute or through symbols and prefixes. Apart from a direct value association of the attribute, there are ways to associate alternate values based on certain conditions prevailing at runtime.

Syntax

```
[<Definition Type> : <Definition Name>]
      <Attribute Name> : <Attribute Value>
```

<***Attribute Name***> It is name of the attribute, specific for the definition type.

<Attribute Value> This can be a constant or a formula.

Example

```
[Part:  PartOne]
   Line    : PartOne
```

4.2.1 Classification of Attributes

The classification of an attribute is done on the basis of the number of values it accepts and if they can be specified multiple times under the definition i.e. based on the number of sub attributes and the number of values.There are seven types of attributes.

Single and Single List

A **Single** type attribute accepts only one value and can't be specified multiple times. The attributes Set As, Width, Style etc are all of a single type.

Example

```
[Field : Fld 1]
   Set As : "Hello"
   Set As : "TDL"
```

In the field the string "TDL" is displayed as Set As as a Single type attribute. The value of the last specified attribute will be displayed.

A Single List type attribute accepts one value which can be specified multiple times. These attributes also accepts a comma separated list.

Example

```
[Line   : Line 1]
   Field : Fld 1, Fld 2
   Field : Fld 3
```

The line Line 1 will have three fields Fld 1, Fld 2 and Fld 3.

Dual and Dual List

Dual type attributes accept two values and can't be specified multiple times. The attributes Repeat is an example of a Dual type.

Example

```
Repeat : Line 1 : Collection 1
```

Dual List type attributes accept two values and can be specified multiple times.

Example

```
Set : Var 1 : "Hello"
Set : Var 2 : "TDL"
```

Triple and Triple List

Triple type attributes accept three values.

Example

```
Object : Ledger Entries : First  : $LedgerName = "Tally"
```

Triple List type attributes accepts three vales and can be specified multiple times.

Example

```
Aggr Method : TrPurcQty : Sum : $BilledQty
Aggr Method : TrSaleQty : Sum : $BilledQty
```

The Attribute type Menu item

The attribute type Menu Item allows the user to add a menu item in the given Menu definition.

Example

```
[#Menu: Gateway Of Tally]
   Item  : Sales Analysis    : Display : Sales Analysis
   Item  : Purchase Analysis : P : Display : Purchase Analysis
```

In the example mentioned above, the options Sales Analysis & Purchase Analysis are added to the Gateway of Tally Menu. For a Purchase Analysis, the character 'P' is explicitly specified as a hot key.

Attributes of Interface Definitions

Frequently used attributes of interface definitions like Report, Form, Part, Line and Field are explained in this section.

Report Definition Attributes

Form

Every report requires one or more Forms. If you have more than one form, then the first form is displayed by default. When you are in print mode, all the forms will be printed one after the other.

Syntax

```
      Form : <Form Name>
```

Example:

```
   [Report : HW Report]
      Form : HW Form
```

This code defines the report 'HW Report', using the form HW Form.

If you choose a Report that has no Forms defined, Tally.ERP9 assumes that the Form Name is the same as the Report Name and looks for it. If it exists, Tally.ERP9 displays it. Otherwise, Tally.ERP9 displays an error message '*Form :<Report Name> does not exist'*.

Title

The Title attribute is used to give a meaningful title to the Report.

Syntax

```
      Title : <String or Formula>
```

By default, Tally.ERP 9 displays the name of the Report as Title, when it is invoked from the menu. If the title attribute is specified, then it overrides the default title.

Example:

```
   [Report : HWReport]
      Form  : HWForm
      Title : "Hello World"
```

Here, "Hello World" is displayed as the title of the Report, instead of HWReport.

Form Definition Attributes

Part / Parts

The attribute Part defines Parts in a Form. Part and Parts are synonyms of the same attribute. This attribute specifies the alignment of the Parts in a Form. By default, the Parts are aligned vertically.

Syntax

```
      Part / Parts : <List of Part Names>
```

Example

```
   [Form : HW Form]
      Part : HW Title Partition, HW Body Partition
```

This part of the code defines two parts, HW Title Partition and HW Body Partition which are vertically aligned, starting from the top of the Form.

Part Definition Attributes

Line / Lines

This attribute determines the Lines of a Part.

Syntax

```
      Line / Lines : <list of line names>
```

Example

```
   [Part : HW Part]
      Line  : HW Line1, HW Line2
```

Line Definition Attributes

Field / Fields

The attribute, Field and Fields are similar. They start from the left of the screen or page in the order in which they are specified.

Syntax

```
      Field / Fields : <List of Field Names>
```

Example

```
   [Line : HW Line]
      Fields : HW Field
```

Set as

This attribute sets a value to the Field.

Syntax

```
Set as : <Text or Formula>
```

Example

```
[Field : HW Field]
   Set as :”Hello TDL”
```

Here, the text “Hello TDL” is displayed in the report.

Info

This attribute is used typically to set text for prompts and titles as display strings. Even when used in Create/ Alter mode, this attribute does not allow the cursor to be placed on the current field as against the Attribute Set as. However, in display mode the Attributes Set as and Info function similarly.

Syntax

```
Info : <Text or formula>
```

Further, if both the attributes (Set as and Info) are specified, then the value set with the attribute Set as overrides the value set with the attribute Info.

Skip

This attribute causes the cursor to skip the particular field and hence, the value in the field cannot be altered by the user, even if the report is in Create or Alter mode.

Syntax

```
Skip : <Logical Formula>
```

Example

```
[Field : HW Field]
   Type    : String
   Set as  : “Hello World“
   Skip    : Yes
```

This code snippet sets the value in the ‘HW Field’ as ‘Hello World’ and forces the cursor to skip the field.

The above code snippet can also be rewritten as:

```
[Field : HW Field]
   Type    : String
   Info    : “Hello World“
```

The attribute Info at Field combines both Skip and Set As.

4.3 Modifiers

Modifiers are used to perform a specific action on definition or attribute. They are classified as Definition Modifiers and Attribute Modifiers. Definition Modifiers are #, ! and *. Attribute Modifiers are Use, Add, Delete, Replace/Change, Option, Switch and Local. They are classified into two:

- Static/Load time modifiers : Use, Add, Delete, Replace/Change
- Dynamic/Real time modifiers : Option, Switch and Local

4.3.1 Static/Load time Modifiers

These modifiers do not require any condition at the run time. The value is evaluated at the load time only and remains static throughout the execution. Use, Add, Delete, Replace are static modifiers.

Use

The USE Keyword is used in a definition to reuse an existing Definition.

Syntax

```
      Use : <Definition Name>
```

Example

```
   [Field   : DSPExplodePrompt]
      Use        : Medium Prompt
```

All the properties of the existing field definition Medium Prompt are applicable to the field DSPExplodePrompt.

Add

The ADD modifier is used in a definition to add an attribute to the Definition.

Syntax

```
      Add :<Attribute Name>[:<Attribute Position>:<Attribute Name>]
            :<Attribute Value>
```

<Attribute Name> This is the name of the attribute specific to the definition type

<Attribute Position> It can be any one of the keywords Before, After, At Beginning and At End. By default the position is At End.

<Attribute Value> This can either be a constant or a formula.

Example

```
    [#Form  : Cost Centre Summary]
      Add   : Button    : ChangeItem
```

A new button ChangeItem is added to the form Cost Centre Summary.

Example

```
   [#Part : VCH Narration]
      Add : Line : Before : VCH NarrPrompt : VCH ChequeName, VCH AcPayee
```

The lines VCH ChequeName, VCH AcPayee are added before the line VCH NarrPrompt in the part VCH Narration.

Delete

The Delete modifier is used in a definition to delete an attribute of the Definition.

Syntax

```
      Delete :<Attribute Name>[:<Attribute Value>]
```

<Attribute Value> This is optional and can either be a constant or a formula. If the attribute value is omitted, all the values of the attribute are removed.

Example

```
   [Form: Cost Centre Summary]
      Use    : DSP Template
      Delete : Button   : ChangeItem
```

The button ChangeItem is deleted from the form Cost Centre Summary. The functionality of the button ChangeItem is no longer available in the form Cost Centre Summary.

If the Button name is not specified, then all the buttons will be deleted from the Form.

Replace

The Replace modifier is used in a definition to alter an attribute of the Definition.

Syntax

```
      Replace : <Attribute Name> :<Old Attribute Value>: <New Attribute Value>
```

<Attribute Name> This is the name of the attribute specific for the definition type.

< Old Attribute Value > *and* ***<New Attribute Value>*** can either be a constant or a formula.

Example

```
[Form: Cost Centre Summary]
   Use     : DSP Template
   Replace: Part : Part1: Part2
```

The part Part1 of form Cost Centre Summary is replaced by the Part2. Now only the Part2 properties are applicable.

4.3.2 Dynamic/Real time modifiers

Dynamic modifiers get evaluated at the run time based on a condition. These modifiers are run every time the TDL is executed in an instance of Tally. Option, Switch and Local are the Dynamic modifiers.

Local

The Local attribute is used in the context of the definition to set the local value within the scope of that definition only.

Syntax:

```
Local        : <Definition Name> :<Old Attribute Value>
             : <New Attribute Value>
```

Example

```
Local  : Field  : Name Field  : Set As  : #StockItemName
```

The value of the formula #StockItemName is now the new value for the attribute Set As of the field Name Field applicable only for this instance. Elsewhere the value will be as set in the field definition.

Option

Option is an attribute which can be used by various definitions, to provide a conditional result in a program. The ‘Option’ attribute can be used in the ‘Menu’, ‘Form’, ‘Part’, ‘Line’, ‘Key’, ‘Field’, ‘Import File’ and ‘Import Object’ definitions.

Syntax

```
Option : <Optional definition>: <Logical Condition >
```

If the ‘Logical’ value is set to ‘True’, then the ‘Optional definition’ becomes a part of the original definition and the attributes of the original definition are modified based on this definition.

<Modified Definition> This is the name of a definition defined as optional definition using the definition modifier !.

Example

```
[Field : FldMain]

   Option : FldFirst : cond1

   Option : FldSecond: cond2
```

The field FldFirst is activated when the cond1 is true. The field FldSecond is activated when the cond2 is true. Optional definitions are created with the symbol prefix "!" as follows:

```
[!Field  : FldFirst]

[!Field  : FldSecond]
```

Switch - Case

The Switch - Case attribute is similar to the Option attribute but reduces code complexity and improves the performance of the TDL Program.

The Option attribute compulsorily evaluates all the conditions for all the options provided in the description code and applies only those which satisfy the evaluation conditions.

The attribute Switch can be used in scenarios where evaluation is carried out only till the first condition is satisfied.

Apart from this, the Switch can be grouped using a label. Therefore, multiple switch groups can be created and zero or one of the switch cases could be applied from each such group.

Syntax

```
      Switch: Label: Desc name : Condition
```

Example

```
[Field: Sample Switch]

       Set as   : "Default Value"

       Switch   : Case1: Sample Switch1: ##SampleSwitch1

       Switch   : Case1: Sample Switch2: ##SampleSwitch2

       Switch   : Case1: Sample Switch3: ##SampleSwitch3

       Switch   : Case2: Sample Switch4: ##SampleSwitch4
```

4.3.3 Sequence of Evaluation – Attributes

The order of evaluation of the attributes is as specified below:

1. Use
2. Normal Attributes
3. Add/Delete/Replace
4. Option
5. Switch
6. Local

4.3.4 Delayed Attributes

Add/Delete/Replace are referred to as Delayed attributes because even if they are specified within the definition in the beginning, their evaluation will be delayed till the end, within the static modifier and normal attributes.

4.4 Actions in TDL

TDL is an action driven language. Actions are activators of specific functions with a definite result. Actions are performed on two principal definition types, 'Report' and 'Menu'. An action is always associated with an originator, requestor and an object. All the actions originate from the Menu, Key and Button. An action is evaluated in the context of the Requestor and Object.

Typically, actions are initiated through the selection of a Menu item or through an assignment to a Key or a Button. Examples of Actions are: Display, Menu, Print, Create, Alter etc.

Syntax

```
      Action : <Action Name> [: <Definition/Variable Name> : Formula]
```

<Action Name > It is the name of the action to be performed. It can be any of the pre-defined actions.

<Definition/Variable Name> It is the name of the definition/variable on which the specified action is to be performed.

Example

```
   Action : Create : My Sample Report
```

4.5 Data Types

The Data Types in TDL specify the type of data stored in the field. TDL being the business language, supports business data types like amount, quantity, rate apart from the other basic types. The data types are classified as Simple Data Type and Compund Data Type.

Simple Data Type

This holds only one type of data. These data types cannot be further divided into sub-types. String, Number, Date and Logical data types fall in this category.

Compound Data Type

This is a combination of more than one data type. The data types that form a compound data type are referred to as sub-data type. The Compound Data types in TDL are: Amount, Quantity, Rate, Rate of exchange and Aggregate.

Table 2:1 shows the Sub-Types under a particular Data Type.

Data Types	Sub - Types
Simple Data Types	
Number	
String	
Date	
Logical	
Compound Data Types	
Amount	Base / Direct Base
	Forex
	Rate Of Exchange
	DrCr
Quantity	Number
	Primary Units/ Base Units
	Secondary Unit/ Alternate Units/ Tail Units
Rate	Price
	Unit Symbol
Rate of Exchange	

TABLE 2.1   Data Types and its Sub- Data Types

The type for the field definition is specified using the **Type** attribute.

Syntax

```
      [Field: <Field Name>]
            Type : <Data type> : <Sub-type>
```

Example

```
   [Field   : Qty Secondary Field]
      Type  : Quantity : Secondary Units
```

4.6 Operators in TDL

Operators are special symbols or keywords that perform specific operations on one, two or three operands and then return a result.

The three types of operators in TDL are as follows:

4.6.1 Arithmetic Operators

The arithmetic operators supported by TDL are as shown in Table 2:2:

+	Addition
-	Subtraction
/	Division
*	Multiplication

TABLE 2.2 Arithmetic Operators

4.6.2 Logical Operators

The logical operators used are: OR, AND ,NOT , TRUE/ON/YES and FALSE/OFF/NO

OR	Returns True if either of the expression is true.
AND	Returns True when both the expressions are True
NOT	Returns True if the expression value is False and False when expression value is True
TRUE/ON/YES	Can be used to check if the value of the expression is True.
FALSE/OFF/NO	Can be used to check if the value of the expression is False.

TABLE 2.3 Logical Operators

4.6.3 Comparison Operators

A Comparison Operator compares its operands and returns a logical value based on whether the comparison is True. The Comparison Operator returns the value as True or False. TDL supports the following Comparison Operators:

= /Equal/ Equals	Checks if the values of both the expressions are equal.
</LessThan/Lesser Than / Lesser	Checks if the value of the <expression 1> is less than the value of <expression 2>.
> / Greater Than/ More	Checks if the value of the <expression 1> is greater than the value of <expression 2>.
In	Checks if the value is in the List of comma separated values.
Null	Checks whether the expression is Empty.
Between …. And	Checks if the expression value is in the range

TABLE 2.4 Comparison Operators

4.6.4 String Operators

String operators facilitate the comparison of two strings. The following are the String operators:

Contains/ Containing	Checks if the expression contains the given string
Starting With / Beginning With/ Starting	Checks if the expression starts with the given string
Ending With / Ending	Checks if the expression ends with the given string
Like	Checks if the expression matches with the given string pattern

TABLE 2.5 String Operators

Notes

The operator = is a comparison operator, not an assignment operator. There is no assignment operator in TDL. While evaluating the expression some keywords are ignored. The keywords which are not considered are ***Than, With, By, To, Is, Does, Of.***

4.7 Special Symbols

The Symbol Prefix in Tally Definition Language (TDL) has different usage and behavior when used with different definitions and attributes of definitions.

Special Symbols used in TDL are $, $$, @, @@, #, ##, ;, ;;, ;;;, /* */, + , ! , * and _ . Each of these symbols are used for a specific purpose. The usage of each of these symbols will be discussed in detail in the subsequent chapters.

4.8 Functions

A function is a small code which accepts a certain number of parameters performs a task and returns the result. The function may accept zero or more arguments but returns a value.

The functions in TDL are defined and provided by the platform. These functions are meant to be used by the TDL programmer and are specifically designed to cater to the business requirement of the Tally.ERP 9 application.

TDL has a library of functions which allows performing string, date, number and amount related operations apart from the business specific tasks. Some of the basic functions provided by TDL are $$StringLength, $$Date, $$RoundUp, $$AsAmount. TDL directly supports a variety of business related functions such as $$GetPriceFromLevel, $$BillExists, $$ForexValue.

Syntax

```
      $$<Function Name>: <Argument List>
```

Example

```
$$SysName:EndOfList
```

The function returns True if the parameter passed is a TDL reserved string.

Learning Outcome

- In a TDL program, the Report and Menu definitions can exist independently.
- The hierarchy of definitions in a TDL program are as follows:
  - Report uses a Form
  - Form uses a Part
  - Part uses a Line
  - Line uses a Field and
  - A Field is where the contents are displayed or entered.
- The Report is called either from a Menu or from a Key Event.
- TDL consists of Definitions, Attributes, Modifiers, Data Types, Operators, Symbols and Prefixes, and Functions.

Symbols and Prefixes

Introduction

In the previous lesson, we have discussed the various TDL Components like definitions, attributes, functions, symbol prefixes, variables etc.

In TDL, there are a few symbols which are used for a specific purposes. Some symbols are used as access specifiers i.e. mainly used to access value of a method, variable, field, formula etc. Some are used for a general purpose such as modifiers. Figure 3.1, Let us refer to the table below to understand the categorization of the various symbols and their usage at a glance.

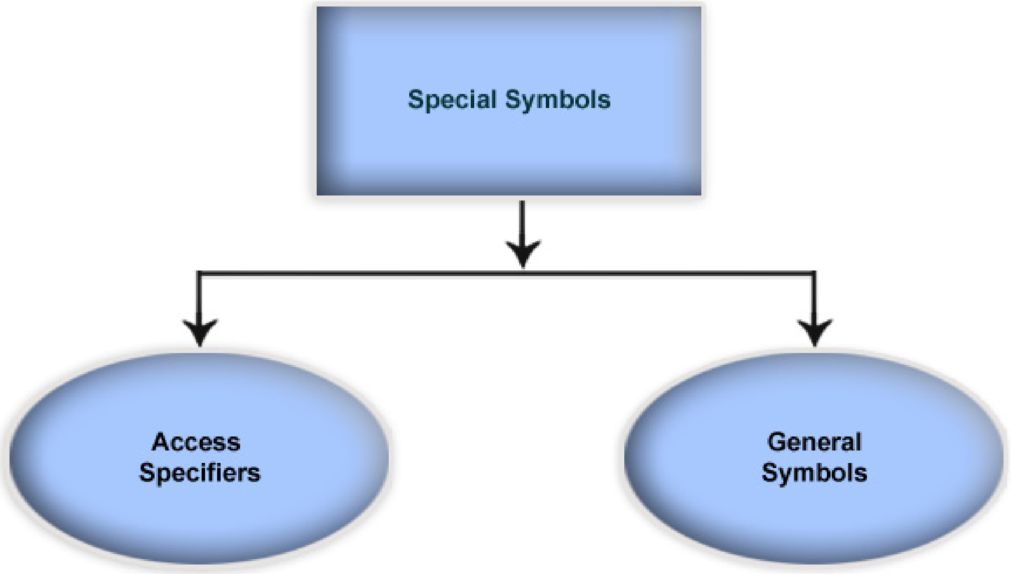


Figure 3.1 Symbol Categorization

1. Access Specifiers/Symbol Prefixes

Symbols	Usage
@	Used to access Local formula.
@@	Used to get the value of a System formula
#	When prefixed to Field name gives the value of the field
##	Used to get the value of a global variable
$	Used to access the value of an Object Method
$$	Used to call the Function

TABLE 3.1 Access Specifiers

2. General Symbols

Symbols	Usage
; ;; ;;; /* */	Used for adding comments in TDL
+	Used as line continuation character
_ (underscore)	Used to expose methods to ODBC and for creating SQL Procedure
*	Used to Reinitialize a Definition
!	Used to create an Optional Definition
#	Used as a definition modifier

TABLE 3.2 General Symbols

3. The Usage of @ and @@

3.1 Formula

In TDL, large complex calculations can be broken down to smaller simple calculations or expressions expressed as a Formula. The values computed using these formulae can be accessed using the symbol prefixes @ and @@.

3.1.1 Naming Conventions for Formula

- Case insensitive
- Only alphanumeric characters are allowed
- Space insensitive at Definition time. However, during deployment or usage of the same, spaces are not allowed

3.1.2 Classifications of formulae

- Local Formula
- Global Formula

3.1.3 Local Formula

A Local Formula is one which can be defined and retrieved at any Interface Definition. The scope of the local formula is only within the current definition. A local formula is usually defined if the formula is specific to a definition and not required by any other definition. The value of a Local Formula can be accessed using the Symbol Prefix @.

Example

```
[Field: CompanyNameandAddress]
   Set as: “Tally India Pvt Ltd, No 23 & 24, AMR Tech Park II, Hongasandra,+
            Bangalore”
```

The above could be written, using the Local Formula as:

```
[Field   : CompanyNameandAddress]
   Company     : “Tally India Pvt Ltd, ”
   Address     : “No 23 & 24, AMR Tech Park II, Hongasandra, ”
   City        : “Bangalore”
   Set as      : @Company + @Address + @City
```

3.1.4 Global Formula

A Global Formula, is one which when defined once, is available globally. In other words, the Global Formula value can be accessed by all the Definitions. A Global formula is defined when a formula is required at many locations. The value of a Global Formula can be accessed using the Symbol Prefix @@. A Global Formula can also be referred to as a System Formula. All the Global Formulae must be defined within the **[System: Formula]** Definition Section.

Example

```
[System: Formula]
   AmtWidth : 20

[Field: RepTitleAmt]
   Width    : @@AmtWidth

[Field: RepDetailAmt]
   Width    : @@AmtWidth

[Field: RepTotalAmt]
   Width    : @@AmtWidth
```

In the example mentioned above, all the Fields assume the same width. If the width of the fields need to be altered, a change is made only at the **[System: Formula]** Section. This change will be applied to all the Fields using the Global Formula AmtWidth.

4. The Usage of # and

In TDL, the Symbol Prefix # can be used for:

- Referencing a field using #
- Modifying the existing definitions using #

4.1 Referencing a Field using

The Symbol Prefix # is used to retrieve the value from another Field.

Example

```
[Field: HW]
   Set as : "Hello World"

[Field: HW1]
   Set as : #HW
```

In the example mentioned above, the value within the Field HW is being set to the Field HW1. In other words, the contents of the Field HW i.e., The Field HW1 is set to "Hello World" by using #HW.

4.2 Modifying existing Definitions using

The Symbol Prefix # is also used to modify existing definitions. One can alter the attributes of the definition. For e.g., adding a new Field within a Line definition.

Example

```
[#Menu: Gateway of Tally]
   Add   : Key Item: Hello World: H : Display: HWReport
   Title : "Tally Gateway"

[#Field: LedParticulars]
   Width : 50
```

In the example mentioned above, the existing Menu Gateway of Tally (default Menu) has been altered to add the Item 'Hello World' and the Title of the Menu is changed to Tally Gateway. The existing Field LedParticulars have also been altered to set its width attribute to the value of 50.

4.3 Accessing value from a Variable using

As the name suggests, a Variable is a named container of data which can be altered as and when required. In TDL, Variables can be classified as Local and Global Variables. Local variables retain their value only within a particular Report. Global variables on the other hand, retain their values throughout the session or permanently, based on the Variable Definition. We will learn more about Variables later.

The value of a Variable can be accessed using symbol prefix ##. Both Local and Global Variables can be retrieved using ##. Local variable is being checked for first. In cases where the Local Variable is not found, it assumes the Global Variable value.

Example

```
[Field   : FGField]
   Set as: ##RTitle


[Report  : DBLedReport]
   Title : if ##LedgerName = " " then "Daybook" else "Ledger Report"
```

5. The Usage of $ and $$

5.1 Accessing a Method using $

Any information from an Object can be extracted by using a Method or UDF. The $ Prefix is used to invoke or deploy the value from a Method or UDF of any Object, where the term Method and Object are TDL specific. This will be covered in greater depth in the sections to follow.

Context Fall Through for $

- Check if it is an internal method or UDF within the current object
- User Defined Method
- System Formula
- Change context to parent object and repeat the above steps

Example

```
[Field   : My Field]
    Set as : $Name
```

The previous code snippet displays the value of the method Name of the associated object.

5.2 Calling an Internal Function using $$

In TDL, functions are inbuilt and TDL Programmers can make use of the same. A function can accept zero or more arguments to perform a specific task on the arguments and return a value. While passing arguments to functions, spaces and special characters except bracket () are not allowed. If the function parameter requires an expression, it can be enclosed within bracket () so as to return the result of the expression as a parameter to the Function.

Example

```
   [Field   : Current Date]
      Set as  : $$MachineDate

   [Field   : Credit Amt]
      Set as  : if $$IsDr:$ClosingBalance then 0 else $ClosingBalance

   [Field   : StringPart Field]
      Set as  : $$StringPart($Email:Company:##SVCurrentCompany):0:5
```

6. Commenting a Code using ;, ;; and /**/

Commenting increases readability. In TDL, Comments can be given using symbol prefixes viz. ;, ;; and /* */. Symbol Prefix ; is used for Part line commenting, ;; is used for Single Line Commenting and /* */ is used for Multi Line Commenting. All the lines enclosed within /* and */ will be ignored by the TDL Interpreter as a comment.

A Single Semi-Colon (;) is allowed as a comment for single line commenting but as a standard coding practice, it is recommended to use Double Semi-Colon (;;).

Example

```
/*
This code explains the usage of Multi-Line Commenting
as well as Single Line Commenting.
*/
;; Altering Menu Gateway of Tally

   [#Menu   : Gateway of Tally]
      Add   : Key Item  : Comment   : C   : Display : Comment
;; Menu Item alteration ends here
```

7. Line Continuation Character (+)

A Line Continuation Character (+) is used to split a lengthy line into number of shorter lines. By doing this, the programmer can see the entire line without scrolling to the left or right. This can also help in understanding and debugging the code faster.

Example

```
/*
This code explains the mechanism of breaking a line into Multiple Lines using +
*/
;; Altering Menu Gateway of Tally
   [#Menu: Gateway of Tally]
      Add : Key Item : Before : @@locQuit : +
            LineCtn : C : Display : LineCtn : +
            NOT $$IsEmpty : $$SelectedCmps
```

8. Exposing Methods and Creating Procedures (_)

The Symbol Prefix (_) is used to expose Methods to ODBC. By prefixing _ to a Collection Name, it turns into a procedure which can be referenced externally by passing the parameter as a Variable.

Example

```
;; Exposing Methods within the Objects to ODBC
   [#Object : Ledger]
      _Difference : $ClosingBalance - $OpeningBalance
;; Creating Procedures to be referenced externally

   [Collection: _LedBills]
      Type       : Bills
      Child of   : #UName
      SQLParms   : UName
      SQLValues  : Bill No    : $Name
      SQLValues  : Bill Date  : $$String:$BillDate:UniversalDate
```

9. Reinitialize Definitions (*)

This is similar to operators such as '#' (Modify) and '!' (Option). When * is used for an existing definition; all the attributes of the definition are overridden. This is very useful when there is a need to completely replace the existing description content with a new code.

Example

```
   [*Field  : MyField]
      Width : 20% Page
      Set as: "This Field has been reinitialized"
```

10. Optional Definitions (!)

The Symbol Prefix ! is used to define optional definitions. Switch and Option are attributes which can be used by various definitions like Menu, Form, Part, Line, Field, Collection, Button, Key, Import File and Import Object to provide a conditional result in TDL. However, they cannot be used with Report, Color, Style, Variable, System Formula, System Variable, System UDF, Border and Object definitions.

The attributes of the original definition are overridden by the attributes of the optional definition only if the Logical Condition is satisfied. In other words, if the Logical Condition returns True, the attributes of the optional definition become a part of the original definition else it is ignored, leaving the original definition intact.

Syntax

```
      Option : <Optional Definition> : <Logical Condition>
      Switch : Label : <Optional Definition> : <Logical Condition>
```

The difference between Switch and Option is that Switch statements bearing the same label are executed till a satisfying condition is found. On the other hand, option executes all the Option statements matching the given conditions sequentially. Switch statements bearing different labels are similar to Option statements as all the Switch statements will be executed for the given conditions.

Example - Option

```
   [Line: MFTBDetails]
      Fields       : MFTBName
      Right Fields: MFTBDrAmt, MFTBCrAmt
      Option       : MFTBDtlsClsgG1000 : $ClosingBalance > 1000
      Option       : MFTBDtlsClsgL1000 : $ClosingBalance < 1000

   [!Line: MFTBDtlsClsgG1000]
      Local        : Field : MFTBDrAmt : Style : Normal Bold
      Local        : Field : MFTBCrAmt : Style : Normal Bold

   [!Line: MFTBDtlsClsgL1000]
      Local        : Field : MFTBDrAmt : Style : Normal
      Local        : Field : MFTBCrAmt : Style : Normal
```

In the above code snippet, the condition specified in both the options, will be checked and it will execute the option satisfying the given condition. In this case, there is a possibility that more than one condition might satisfy and get executed.

Example - Switch

```
   [Line: MFTBDetails]
      Fields       : MFTBName
      Right Fields: MFTBDrAmt, MFTBCrAmt
```

```
    Switch      : Case 1: MFTBDtlsClsgG1000 : $ClosingBalance > 1000
    Switch      : Case 1: MFTBDtlsClsgL1000 : $ClosingBalance < 1000

 [!Line: MFTBDtlsClsgG1000]
    Local : Field : MFTBDrAmt : Style : Normal Bold
    Local : Field : MFTBCrAmt : Style : Normal Bold

 [!Line: MFTBDtlsClsgL1000]
    Local : Field : MFTBDrAmt : Style : Normal
    Local : Field : MFTBCrAmt : Style : Normal
```

In the previous code snippet, the condition specified in the switch statements, will be checked one after another. The first statement satisfying the given condition will be executed and all other statements grouped within this label, ‘Case 1’ will not be executed further unlike Option. The similar behavior of Option can be achieved by specifying different labels, if required.

Learning Outcome

- Access Specifiers and General symbols are the two different special symbols used in TDL.
- The Access Specifiers @ and @@ is used for accessing the value of Local and global formula respectively.
- # can be used for referencing a field or modifying the existing definition.
- ## is used for accessing the value from a Local or global variable.
- $ is used for accessing a method or UDF and $$ is for calling an internal function.

Dimensions and Formatting

Introduction

Dimensions are specifications. Dimensions in TDL are effective either in the display mode or in the print mode. The data in TDL does not have an absolute position of the dimensions specified but a relative .

There are four definitions in TDL that attract dimensions. They are:

- Form
- Part
- Line
- Field

1. Unit of Measurement

A Unit of Measurement can be any of the following:

- Millimeters/ mms
- Centimeters/ cms
- Inch(es)
- Number of Characters/ Number of Lines
- % Screen/ Page
- Number – Points (where 1 Point = 1/72 Inch)

It is advisable to follow uniform Units of Measurement throughout the Report in order to avoid confusion.

2. Dimensional Attributes

Dimensional Attributes can be classified into two i.e., Specific and General Attributes. They are as shown in Table 4:1:

Definitions	Specific Dimensions	General Dimensions
Form	Height, Width, Space Top, Space Bottom, Space Left, Space Right	Horizontal Align, Vertical Align, Full Height, Full Width
Part	Height, Width, Space Top, Space Bottom, Space Left, Space Right	Horizontal Align
Line	Height, Space Top, Space Bottom, Indent	Full Height
Field	Width, Space Left, Space Right, Indent	Full Width, Widespaced

TABLE 4.1 Dimensional Attributes

2.1 Sizing/Size Attributes

2.1.1 Height and Width

The attribute Height is used to specify the Height required for the Form, in the Part and Line Definition whereas the attribute Width is used to specify the Width required for the Form, Part and Field Definition. The Height and Width can be specified in terms of any of the above Units of Measurement. In the absence of any Unit of Measurement, the Height assumes a certain number of lines and similarly, the Width assumes number of characters. The entire Height and Width is in the proportion of the available paper/ screen dimensions.

Syntax

```
      Height      : <Measurement Formula>
      Width       : <Measurement Formula>
```

2.1.2 Height and Width – Form Definition

The Height and Width when specified in a Form Definition implies that it is the available Height and Width which can be utilized by all the Parts, Lines and Fields within the Form. If the contents of the Part and Line exceed the available Height and/or Width, the contents of the Form are squeezed to accommodate the same within the available Height and Width. In the absence of any Height and Width specified, the Form Definition assumes the Height and Width required by the contents of the Form comprising of Parts, Lines and Fields.

Example

```
   Height    : 10 inch
   Width     : 8.50 inch
```

2.1.3 Height and Width – Part Definition

Subsequently, Height and Width when specified in a Part Definition implies that it is the available Height and Width that can be utilized by all its Sub-Parts, Lines and Fields. If the contents of the Sub-Parts, Lines and Fields exceed the available Height and Width, the contents of the Part are squeezed to accommodate the same within the available Height and Width.

Example

```
Height    : 10% Page
Width     : 60% Page
```

2.1.4 Height – Line Definition

Similarly, the Height when specified within a Line Definition restricts the contents of the Lines to the available Line Height. Generally, specifying the Line Height is not required since the contents of the lines are controlled by the given Part Height.

2.1.5 Width – Field Definition

The Width when specified within a Field Definition limits the contents of the Field within the defined boundary. If the contents are longer than the available Width, the Field contents are squeezed to accommodate the same within the defined width.

2.1.6 FullHeight and FullWidth

The Attribute FullHeight can be specified in a Form or a Line Definition and Attribute FullWidth can be specified in a Form or a Field Definition. FullHeight is used to instruct the Form or a Line to utilize the required Height while FullWidth is used to instruct the Form or a Field to utilize the required Width.

Syntax

```
      FullHeight  : <Logical Value>
      FullWidth   : <Logical Value>
```

Example

```
FullHeight      : No
FullWidth       : No
```

2.1.7 FullHeight and FullWidth – Form Definition

The attribute FullHeight decides whether to allow the form to consume the required Height or not depending on the logical value set. By default, the value set for this attribute is Yes. If the current Form uses Bottom Parts or Bottom Lines, then the Height required/ utilized by the Form will be 100% Page/ Screen.

Similarly, the attribute FullWidth decides whether to allow the Form to consume the available Full Width or not depending on the logical value set. By default, the value set for this attribute is Yes.

If the current Form uses the Right Parts or Right Lines, then the Width required/ utilized by the Form will be 100% Page/ Screen.

2.1.8 FullHeight – Line Definition

The attribute FullHeight decides whether the line can consume the required Height or not depending on the logical value set. By default, the value set to this attribute is Yes.

2.1.9 FullWidth – Field Definition

The attribute FullWidth decides whether the Field can consume the required Width or not depending on the logical value set. The value set to this attribute by default, is Yes.

2.2 Spacing/Position Attributes

2.2.1 Space Top, Space Bottom, Space Left and Space Right

Attributes Space Top, Space Bottom, Space Left and Space Right are used to specify the spaces to be kept to the Top, Bottom, Left and Right of the Definition. Space Top and Space Bottom can be used in a Form, Part and Line Definition. Space Left and Space Right can be used in a Form, Part and Field Definition. When Space Top, Space Bottom, Space Left and Space Right are used in a definition, these spaces are included in the Height and Width specified within the definition.

Syntax

```
      Space Top          : <Measurement Formula>
      Space Bottom       : <Measurement Formula>
      Space Left         : <Measurement Formula>
      Space Right        : <Measurement Formula>
```

Example

```
   Space Top       : 1.5 inch
   Space Bottom    : If ($$IsStockJrnl:##SVVoucherType OR +
                     $$IsPhysStock:##SVVoucherType) then 0 else 0.25
   Space Left      : @@DSPCondQtySL + @@DSPCondRateSL + @@DSPCondAmtSL
   Space Right     : 1
```

2.2.2 Space Top, Space Bottom, Space Left and Space Right – Form / Part Definition

The attributes Space Top, Space Bottom, Space Left and Space Right are specified in a Form or a Part Definition, by leaving the appropriate spaces before displaying / printing a Form. These spaces are included in the Height / Width of the Form Definition.

2.2.3 Space Top and Space Bottom – Line Definition

The attributes Space Top and Space Bottom when specified in a Line Definition, leave the appropriate spaces before/ after the Line. These spaces are inclusive within the Height of the specific

Part in which the current Line Definition resides. If the Height of the Part is unable to accommodate the same, it compresses the line to fit it within the available Height.

2.2.4 Space Left and Space Right – Field Definition

The attributes Space Left and Space Right when specified in a Field Definition leave the appropriate spaces before/ after the Field. These spaces are inclusive within the Width of the Part and Field. If the Width of the Part is unable to accommodate the same, it compresses the Fields within the Parts and Lines to fit it within the available Width.

2.2.5 Indent

An Indent can be specified either in a Line or a Field Definition. It is similar to the Tab Key which is used to specify a starting point for a Line or a Field.

Syntax

```
Indent: <Measurement Formula>
```

Example

```
Indent: @@IndentByLevel
```

2.2.6 Indent – Line Definition

This attribute in the Line Definition specifies the space to be left from the Left margin before the contents of the line begin.

2.2.7 Indent – Field Definition

This attribute in the Field Definition is similar to the Space Left attribute, except that this attribute indents the field independent of width of the field. Space Left indents the field within the width available. However, Indent indents the field exclusive of the width. It can either take a formula as a parameter or you can give the expression itself as a parameter. The formula can decide as to what extent each instance of the field has to be indented from the initial place. This attribute is typically used while displaying reports like list of accounts, Trial Balance, etc., where the groups and ledgers under a particular group are recursively indented inside the group, based on the order of the groups and ledgers.

3. Alignment Attributes

3.1 Top Parts, Bottom Parts, Left Parts and Right Parts

These attributes are used to place different parts at different positions in a particular Form or Part. The attributes Top Parts and Bottom Parts can be specified both in Form as well as Part Definition whereas Attributes Top Parts, Bottom Parts, Left Parts and Right Parts can be specified in a Part Definition.

Syntax

```
      Top Parts          : <Part1, Part2, ….>
      Bottom Parts       : <Part1, Part2, ….>
      Left Parts         : <Part1, Part2, ….> ;; Only for Part Definition
      Right Parts        : <Part1, Part2, ….> ;; Only for Part Definition
```

Example

```
   Top Parts       : ACLSFixedLed, TDSAutoDetails
   Bottom Parts    : PJR Sign
   Left Parts      : EXPINV Declaration
;; Only for Part Definition
   Right Parts     : STKVCH Address
;; Only for Part Definition
```

3.1.1 Top Parts and Bottom Parts – Form Definition

In cases where the Top Part or Bottom Part is specified within a Form Definition, it occupies the Top Section or Bottom Section of the Form respectively, keeping in account the Space Top and Space Bottom of the Form. The attribute Space Bottom impacts the Bottom Parts by moving it from the bottom in order to leave appropriate spaces. Similarly, Space top impacts the Top Parts by moving it from the top in order to leave appropriate spaces.

The Bottom Parts/ Bottom Lines start printing from bottom to the top of the Form. If Height is specified at the Form Definition, then the Bottom Parts/ Lines start printing from the bottommost line within the specified Height.

3.1.2 Top Parts, Bottom Parts, Left Parts and Right Parts – Part Definition

In cases where the Left Part or Right Part is specified within a Part Definition, it occupies the Left Section or Right Section of the Part respectively keeping in view the Space Left and Space Right of the Part. The attribute Space Right impacts the Right Parts by moving it from the right in order to leave appropriate spaces. Similarly, Space Left impacts the Left Parts by moving it from Left in order to leave appropriate spaces. If the intent is to have multiple parts printed horizontally, then the Part Attribute Vertical should be set to No. Incases where the Vertical Attribute is set to Yes, then all the parts within this part will be printed vertically. In such circumstances, the Left Parts will position at the Top of the Screen/ Page and the Right Parts will position at the Bottom of the Screen/ Page.

Incases where the Top Part or Bottom Part is specified within a Part Definition, it occupies the Top Section or Bottom Section of the Part respectively keeping Space Top and Space Bottom of the Part in account. The attribute Space Bottom impacts the Bottom Parts by moving it from the bottom in order to leave appropriate space. Similarly, the attribute Space Top impacts the Top Parts by moving it from the Top in order to leave appropriate spaces. If the intent is to have multiple parts printed vertically, then the Part Attribute Vertical should be set to Yes. If the Vertical Attribute is set to No, then all the parts within this part will be printed horizontally. In such circumstances, the Top Parts will be positioned at the Left of the Screen/ Page while the Bottom Parts are positioned at the Right of the Screen/ Page.

Both Parts and Lines are not allowed within a Part. They are mutually exclusive entities. Either Parts or Lines can be used.

3.2 Top Lines and Bottom Lines

These attributes are used to place different lines at different positions in a particular Part Definition. The attributes Top Lines and Bottom Lines can be specified in a Part Definition. However, the attributes Top Lines/Lines can only be used in a Line and Field Definition.

Syntax

```
Top Lines   : <Line1, Line2,…..>
Bottom Lines: <Line1, Line2,…..>
```

Example

```
Top Lines       : Form SubTitle, CMP Action
Bottom Lines    : VCHTAXBILL Total
```

3.2.1 Top Lines and Bottom Lines – Part Definition

The attribute Top Lines is used to place lines at the top while the attribute Bottom Lines is used to place the lines at the bottom of the Part with respect to the Height specified within the Part Definition.

3.3 Left Field and Right Field

The attribute Left Fields can be specified in both Line and Field Definition whereas the attribute Right Fields can only be specified in a Line Definition.

Syntax

```
Left Fields        : <Field1, Field2, ….>
Right Fields       : <Field1, Field2, ….>
```

Example

```
Left Fields              : Medium Prompt, Chg SVDate, Chg VchDate
Right Fields             : Trader TypeofPurchase, Trader QtyUtilisedTotal
```

3.3.1 Left Fields and Right Fields – Line Definition

The attribute Left Fields and Right Fields specified in a Line Definition places the fields at their respective position. The Left Fields starts printing from the Left to the Right of the Line while the Right Fields starts printing from the Right to the Left of the Line. If Repeat Attribute is used in a Line, specification of Right Fields are not allowed as by default, Repeat Attribute places the Field specified to the Right of the Screen/Page.

3.3.2 Left Fields / Fields – Field Definition

The attribute Field is used to create fields containing one or more fields, like Group fields.

We can create multiple fields inside a single field using the Fields attribute.

The attribute Fields is useful when multiple Fields are required to be repeated in a Line. For example, in case of a Trial Balance, two Fields i.e., Debits and Credits are required to be repeated together if a new column is added by a user. The new column thus added, should again contain both these fields, i.e., Debit and Credit. In a Line Definition, only one Field can be repeated. So, a Field is required within a Field if more than one field requires to be repeated.

3.4 Align

The attribute Align aligns the contents of a Field as specified. The permissible values to this attribute are Left, Center, Right, Justified and Prompt.

Syntax

```
      Align : <String Value>
```

Example

```
   Align    : Right
```

3.4.1 Horizontal Align and Vertical Align

Horizontal align sets the alignment of the Form or Part horizontally while Vertical align sets the alignment of the Form vertically.

Syntax

```
      Horizontal Align  : <Logical Value>
      Vertical Align    : <Logical Value>
```

Example

```
   Horizontal Align      : Right
   Vertical Align        : Bottom
;; Only for Form Definition
```

The alignment of the Form or Part across the width of the page is set by the attribute Horizontal Align. The default alignment of the Form and Part is positioned in the Centre onscreen and in the

Left on print. Depending on the width of the Form and page, the Form or Part will be displayed/printed leaving equal amount of space on the left and right of the Form.

The alignment of the Form across the height of the page is set by the attribute Vertical Align. The default alignment of the Form is Centre on screen and Top on print. Depending on the height of the form and page, the form will be displayed/printed leaving equal amount of space on the top and bottom of the form.

4. Some Specific Attributes

4.1 Inactive

The Inactive attribute can be used in both a Field Definition and a Button Definition.

When the attribute Inactive is set to Yes in a Field Definition, the Field loses its content but the size of the Field remains intact. In cases where a Button Definition, is used, the Button becomes Inactive.

Syntax

```
Inactive     : <Logical Formula>
```

Example

```
[Field: TBCrAmount]

   Set as       : $ClosingBalance

   Inactive     : $$IsDr:$ClosingBalance
```

In the previous example, the Field TBCrAmount is used to display the Credit Amount of the Ledger in a Trial Balance. When the Ledger Balance is Debit, the amount should not be displayed in the Credit Column but the Width should be utilized to avoid the Debit Field being shifted to the Credit Field. The Credit Totals to be calculated and displayed will also exclude the Debit Amount.

4.2 Invisible

This attribute can be specified in a Part, Line or a Field Definition. Based on the logical condition, this attribute decides whether the contents of the definition should be displayed or not. When this attribute is set to Yes, it does not display the contents but the contents are retained for further processing. In this case, contrary to Inactive, the size of the entire field is reduced to null but the value is retained.

Syntax

```
Invisible    : <Logical Formula>
```

4.2.1 Invisible – Field Definition

The invisible attribute when specified in a Field denotes that the current field is excluded from all the further processing based on satisfying certain condition.

Example

```
[Field: Attr Invisible]

   Set as       : "Invisible Attribute"

   Invisible    : Yes
```

In the previous example, the Field Attr Invisible is used to display Credit Amount of the Ledger in a Trial Balance. When the Ledger Balance is debit, the amount should not be displayed/printed in the Credit Column and the Width is not utilized allowing the other fields to utilize the space. The Credit Totals being calculated and printed will also exclude the Debit Amount.

In a Report, at least one Part, Line and Field should be visible.

4.3 Widespaced

This attribute is used in a Field Definition to allow increased spacing between the characters of the string value specified in the field. This attribute is used to create titles for the report / columns.

Syntax

```
      Widespaced  : <Logical Value>
```

Example

```
   Widespaced     : Yes
```

5. Definitions and Attributes for Formatting

5.1 Border

The Definition Border determines the type of lines required in a border which can be used by a Part, Line or a Field which means that this definition can define customized borders for the user. But it is ideal to use the predefined borders which are part of the default TDL instead of the user defined, since almost all possible border combinations are already defined in the Default TDL.

Syntax

```
      [Border: <Border Name>]
            Top         : <Values separated by a comma>
            Bottom      : <Values separated by a comma>
            Left        : <Values separated by a comma>
            Right       : <Values separated by a comma>
            Color       : <Color Name – B&W, Color Name - Color>
            PrintFG     : <Color Name>
```

Top, Bottom, Left and Right

The Top, Bottom, Left and Right attributes in a Border Definition are used to add appropriate lines which constitute the Border defined. The permissible values for these attributes are:

- **Thin/Thick** : This specifies whether the Line should be thin or thick.
- **Flush** : The border includes the spaces on the Top, Bottom, Left or Right.
- **Full Length** : This ignores the space given at the Top, Bottom, Left or Right and prints the border for the whole length.
- **Double** : This parameter forces double line to be printed. In its absence, it is assumed to be single line.

Example

```
[Border: Thin Bottom Right Double]
   Bottom       : Thin, Flush, Full Length
   Right        : Thin, Double

[Field: Total Field]
   Set AS       : $Total
   Border       : Thin Bottom Right Double
```

Color

The Color attribute of the Border Definition is used to specify the Color required for the border in display mode. In a Border definition the attribute Color requires two values to be specified, viz., First is for a Black and White Monitor and the second in case of a Color monitor.

```
[Border: Top Bottom Colored]
   Top          : Thin
   Bottom       : Thin
   Color        : "Deep Grey, LeafGreen"

[Field: Total Field]
   Set AS       : $Total
   Border       : Top Bottom Colored
```

PrintFG

The PrintFG attribute of Border Definition is used to specify the Color required for the border during printing.

```
[Border: Top Bottom Colored]
   Top          : Thin
   Bottom       : Thin
   Color        : "Deep Grey, Leaf Green"
   Print FG     : “Leaf Green”
```

```
[Field: Total Field]
   Set AS       : $Total
   Border       : Top Bottom Colored
```

5.2 Style

The Definition Style can be used in the Field Definition only. This definition determines the appearance of the text being displayed/printed by using a corresponding font scheme, Bold, Italic, Point Size, Font Name, etc.

The Style attribute in Field Definition is used to format the appearance of the text appearing within the Field, both in display and print mode provided the Print Style attribute is not used within the current Field. The Print Style attribute is used in Field, if the Style required while displaying is different from the Style required while printing.

Syntax

```
[Style: <Style Name>]
      Font        : <Font Name>
      Height      : <required Font Height in Point Size>
      Bold        : <Logical Formula>
      Italic      : <Logical Formula>
```

Font

This is the generic name of the Font supported by the Operating System. A Font is system dependent and we do not have any control over them. However, one can select the required fonts from among those available.

Example

```
[Style   : Normal]
   Font         : if $$IsWindows then "Arial" else "Helvetica"
   Height       : @@NormalSize

[Style   : Normal Bold]
   Use          : Normal
   Bold         : Yes

[Field   : Party Name]
   Set AS       : $PartyLedgerName
   Style        : Normal
   Print Style : Normal Bold
```

Height

The Height attribute within the Style Definition should be specified without any measurement specified, since it is always measured in terms of Points. The value for the attribute Height can have fractions and can be denoted by a formula which returns a number.

One can also grow or shrink the Height by a multiplication factor or percentage.

Example

```
[Style    : Normal Large]
   Use    : Normal
   Height: Grow 25%
```

Bold

The attribute Bold can only take logical values/ formula. In other words, it can either take a Yes or No. This signifies that the Field using this Style should be printed in Bold.

Example

```
[Style    : Normal Bold Large]
   Use    : Normal Large
   Bold   : Yes
```

Italic

The attribute Italic can only take logical values/ formula. In other words, it can either be set to Yes or No. This signifies that the Field using this Style should be printed in Italics.

Example

```
[Style    : Normal Large Italics]
   Use    : Normal Large
   Italic: Yes
```

5.3 Color

The Definition Color is useful to determine the Foreground and Background Color for a Form, Part, Field or Border, both in Display as well as in Print Mode.

A Color specification can be done by specifying the RGB Values (the combination of Red, Green and Blue - each value should range from 0 to 255).

Syntax

```
[Color: <Color Name>]
      RGB   : <Red>, <Green>, <Blue>
```

RGB

This is the second way of specifying color. One can specify the RGB value from a palette of 256 colors to obtain the required color. i.e., the values Red, Green and Blue can each range from 0 to 255. This gives the user an option to select from 24 bit color.

Example

```
[Color    : Pale Leaf Green]
   RGB    : 169, 211, 211

[Field    :  Party Name]
   Set as       : $PartyLedgerName
   Color        : Pale Leaf Green
   Print FG     : Pale Leaf Green
```

5.4 Background and Print BG Attribute

The attribute Background is used to set the Background Color of a Form, Part or a Field in display mode. The Print BG attribute is used to set the Background Color of a Form, Part or a Field in print mode.

Syntax

```
[Form: <Form Name>]
      Background   : <Color Name Formula>
      Print BG     : < Color Name Formula>

[Part: <Part Name>]
      Background   : <Color Name Formula>
      Print BG     : <Color Name Formula>

[Field: <Field Name>]
      Background   : <Color Name Formula>
      Print BG     : <Color Name Formula>
```

Example

```
[Form     : Salary Detail Configuration]
   Background   : @@SV_CMPCONFIG

[Part     : Party Details]
   Background   : Red
   Print BG     : Green
```

```
    [Field    : Party Ledger Name]
       Background   : Yellow
       Print BG     : Red
```

5.5 Format Attribute

The attribute Format is used in the Field Definition which determines the Format of the value being displayed/ printed within the Field.

Syntax

```
        [Field: <Field Name>]
              Format      : <Formatting Values separated by comma>
```

The value for the Attribute Format, varies based on the data type of the Field.

Field of Type Number

Example

```
    [Field: My Rate of Excise]
       Set AS       : $BasicRateOfInvoiceTax
       Format       : "No Comma, Percentage"
```

Field of Type Date

Example

```
    [Field: Voucher Date]
       Set AS       : $Date
       Format       : "Short Date"
```

Field of Type Amount

Example

```
    [Field: Bill Amount]
       Set AS       : $Amount
       Format       : "No Zero, No Symbol"
```

Field of Type Quantity

Example

```
    [Field: Bill Qty]
       Set AS       : $BilledQty
       Format       : "No Zero, Short Form, No Compact"
```

Learning Outcome

- The following four definitions in TDL attract the dimensions:
  - Form
  - Part
  - Line
  - Field
- In TDL, Dimensional attributes are used for specifying the dimensions.
- The Definition Style determines the appearance of the text being displayed/printed by using the corresponding Font scheme, Bold, Italic, Point Size, Font Name, etc.
- The Definition Color is useful to determine the Foreground and Background color for a Form, Part, Field or Border both in Display as well as Print Mode.
- The attribute Format is used in the Field Definition which determines the Format of the value being displayed/ printed within the Field.

Variables, Buttons and Keys

Introduction

A Variable is a storage location or entity. It is a value that can change, depending on the conditions or on the information passed to the program.

In TDL, a Variable is one of the important definitions since it helps control the behavior of Reports and their contents. Variables assume different values during execution and these values affect the behavior of the Reports

A Variable definition is similar to any other definition.

Syntax

```
      [Variable: <Variable Name>]
            Attribute : Value
```

A Variable should be given a meaningful name which determines its purpose.

1. Attributes of a Variable

The attributes of a Variable determines its nature and behavior. Some of the widely used attributes are discussed below:

1.1 Type

This attribute determines the Type of value that will be held by the variable. The Types of values that a variable can handle are String, Logical, Date and Number. In the absence of this attribute, a variable assumes to be of the Type String by default.

Syntax

```
      [Variable: <Variable Name>]
            Type   : <Data Type>
```

Example:

```
[Variable : ICFG Supplementary]
   Type      : Logical
```

A logical variable ICFG Supplementary is defined and used to control the behavior of certain reports based on this logical value as configured by the user.

1.2 Default

This attribute is used to assign a default value to a variable, based on the 'Type' defined.

Syntax

```
[Variable: <Variable Name>]
      Default     : <Initial Value>
```

Value of the should adhere to the data type specified with Type Attribute.

Example

```
[Variable : DSP HasColumnTotal]
   Type        : Logical
   Default     : Yes
```

The Default Initial Value for the logical Variable **DSP HasColumnTotal** is set to **Yes**. This variable will begin with an initial value Yes in the Reports unless overridden by the System Formula. We will learn about the System Formula in the coming sections.

1.3 Persistent

This attribute decides the retention periodicity of the attribute. If the attribute Persistent is set to Yes, then the latest value of the variable will be retained across the sessions, provided the variable is not a local variable. We will learn about the concept of local and global variables shortly.

Syntax

```
[Variable: <Variable Name>]
      Persistent: <Logical Value>
```

Example

```
[Variable : SV Backup Path]
   Type        : String
   Persistent  : Yes
```

The attribute Persistent of the variable SV Backup Path has been set to Yes which means that it retains the latest path given by the user even during the concurrent sessions of Tally.

1.4 Volatile

In cases where the Volatile attribute in the variable definition is set to Yes, then the variable is capable of retaining multiple values i.e., its original value with its subsequent values are stored as a stack. The default value of this attribute is Yes.

In cases where a new report R2 is initiated, using a volatile variable V, from the current report R1, the current value of a volatile variable will be saved as in a stack and the variable can assume a new value in the new report R2. Once the previous report R1 is returned back from R2, then the previous value of the variable will be restored. A classic example of this is a drill down Trial Balance.

Syntax

```
      [Variable: <Variable Name>]
            Volatile: <Logical Value>
```

Example

```
   [Variable : GroupName]
      Type        : String
      Volatile    : Yes
```

The Volatile attribute of a Group Name Variable is set to **Yes**, which means that the Group Name can store multiple values which have been recieved from multiple reports.

1.5 Repeat

This attribute is mainly used to achieve the Auto Column behaviour in various Reports. Each Column is created with a subsequent Object in a Collection automatically till all the columns required for Auto Columns exhaust. The Repeat attribute has its value as a variable which has the collection of Objects for which columns need to be generated. Every time the Repeat is executed, the column for subsequent Object is added.

Syntax

```
      [Variable: <Variable Name>]
            Repeat      : <Variable Value>
```

Example

```
   [Variable : SV FromDate]
      Type        : Date
      Volatile    : Yes
      Repeat      : ##DSPRepeatCollection
```

DSPRepeatCollection Variable receives the Collection Name from a Child Report which accepts input from the user regarding the columns required. Variable SVFromDate gets repeated over the subsequent period in the Collection each time the column repeats.

2. The Scope of a Variable

The scope of a Variable can be broadly classified as follows:

- Local
- Global
- Field acting as a variable

2.1 Local

A Variable is termed to be a local variable when it is associated to a Report. This means that the scope of the variable covers only the current report and its components. It is not mandatory for local variables to have an initial value.

Syntax

```
[Report : <Report Name>]
      Variable: <Variable Name>
```

Example

```
[Report   : Balance Sheet]
   Variable     : Explode Flag
```

Explode Flag Variable is made local to Report Balance Sheet by associating it using the Report attribute Variable.

This variable retains its value as long as we work with this Report. On exiting the Report, the original value if given is returned and the value modified within this report is lost. For example, consider a situation where Stock Summary Report is being viewed with Opening, Inwards, Outwards and Closing Columns enabled through Configuration settings. Once we quit this Report and re-enter the Report, the variables return to the default settings.

2.2 Global

A Variable is termed to be a global variable when it is defined under System Variable Section. This means that the scope of the variable covers all the reports. An initial value is mandatory for global variables.

A Global Variable can also be made local to a Report by associating it to a Report as discussed in the Local Variables mentioned above.

```
[System: Variable]
      Variable: <Initial Type Based Value>
```

Example

```
[System   : Variable]

   BSVerticalFlag    : No
```

The BSVerticalFlag Variable is made Global. Hence, this variable value being modified in a Report is retained even after we quit and re-enter the Report.

The retention of a Global Variable can be done on two levels, i.e., either within the current session or across the sessions. If the Variable attribute Persistent is set to **Yes**, then the modified variable value is retained across the sessions else the value defaults back to initial value on re-entering another session of Tally.

All the Persistent Variable Values are stored in a File Named TallySav.Cfg in the folder path specified in Tally.ini. Each time Tally is restarted, these variable values are accessed from this file.

2.3 Field Acting as a Variable

The Variable attribute in a Field Definition is used to make the Field behave as a Variable. This means that as soon as some value is entered/ altered in a Field, the variable assumes the same value with immediate effect. The Variable need not be defined previously since it inherits its data type from the Field itself.

For example: In a Trial Balance Report which is a drill down report there is a need to retain the Group Name which has been selected by the user. So each time the user scrolls up and down, the field value changes and the current field value is passed on to the variable immediately so that if the current group is selected and drilled down, the report begins with the sub groups and ledgers of the selected group.

The Variables used in a Field Acting as a Variable are local variables and are local to the Report

Syntax

```
[Field: <Field Name>]

      Variable: <Variable Name>
```

Example

```
[!Field   : DSP Group Acc]

   Variable     : Group Name
```

This is used in the List of Accounts Report in Tally.ERP 9 wherein the optional Field DSP Group Acc is made to act as a variable by using the Field attribute Variable and the value selected by the user is passed on to this variable for further use.

3. Modifying the Variable Value

A Field attribute **Modifies** is used to modify the value of a variable.

Syntax

```
[Field : <Field Name>]
      Modifies : <Variable Name>
```

Example

```
[Field    : SLedger]
   Modifies    : SLedger
```

The SLedger Variable is modified with the value stored/keyed in the Field **SLedger**

4. Example - Variables

The following code snippet explains the usage of Local variable.

```
[Variable: LocVar]
   Type        : String
   Default     : "This is the default value"
;; Variable LocVar of Type String is defined and it is assigned a Default Value

[Report: Local Variable]
   Variable    : LocVar
;; At this point, Variable LocVar becomes a Local Variable for this Report

   [Field: Local Variable Field]
      Set As   : "This is a Local Variable in Report"
      Modifies : LocVar
;; Modifies the variable value with this Field contents
```

In the above code snippet, a local variable locvar is defined and locally attached to the Report Local Variable. This Report modifies the Variable Value to **'This is a Local Variable in Report'**. Once we exit from this Report, the value of the variable locvar modified in this Report is lost.

5. Buttons and Keys

The actions in TDL can be delivered in three ways: either by activating a Menu Item, by pressing a Key or by activating a Button.

The definition of both Buttons and Keys are the same but at the time of deployment, Keys differ from Buttons.

All the Buttons used within the attribute Buttons are visible on the button bar so that the user can either click it or press the unique key combination. All the Buttons used within the attribute Keys are invisible entities and key combination associated in the Key must be pressed to activate a key, whereas to activate a button, either it can be clicked or key combination assigned for the button can be pressed.

5.1 Attributes of Buttons/ Keys

5.1.1 Title

The Title attribute can be used to give a meaningful Title to the Button being displayed on the Button Bar. This attribute is optional.

In case the Title is not specified, then by default, it assumes the Button Name as its title. In cases where it is used as a Key, then the Title is ignored since the Keys are hidden in a Menu or a Report.

Syntax

```
[Key/Button: <Key/Button Name>]
      Title : <Button Title>
```

Example

```
[Button : NonColumnar]
   Title : “No Columns”
```

5.1.2 Key/ Keys

The Key attribute is used to give a unique key combination which can be activated by pressing the same from any Report or Menu. This attribute is mandatory if action is specified in this definition.

Syntax

```
[Key/Button: <Key/Button Name>]
      Key   : <Combination of Keys>
```

Example

```
[Button: NonColumnar]
   Key   : Alt + F5
```

5.1.3 Action

The **Action** attribute is used to associate an Action with this Button. Every Button or Key defined is for the purpose of executing certain predefined actions.

Syntax

```
[Key/Button: <Key/Button Name>]
      Action: <Required Action>
```

Example

```
[Button: NonColumnar]
   Action      : Set : ColumnarDayBook: NOT ##ColumnarDayBook
```

5.1.4 Inactive

The **Inactive** attribute is used to activate the Button based on some condition. If the condition is false, the button will be displayed but it cannot be activated.

Syntax

```
[Key/Button : <Key/Button Name>]
      Inactive : <Logical Condition>
```

Example

```
[Button : Close Company]
   Inactive: $$SelectedCmps < 1
```

Learning Outcome

- A variable is a storage location or an entity. It is a value that can change, depending either on the conditions or on the information passed on to the program.
- The Variable attribute ‘Type’ determines the Type of value that will be held within it.
- The attribute ‘Default’ is used to assign a default value to a variable, based on the ‘Type’ defined.
- The attribute ‘Persistent’ decides the retention periodicity of the attribute.
- The attribute ‘Modifies’ in a field definition is used to modify the value of a variable.
- Title, Key, Action and Inactive are the attributes of a button definition.

Objects and Collections

Introduction

In the previous lesson the usage of Variables, Buttons and Keys were explained. In this lesson the concept of ‘object and collection’ will be discussed in detail. Let us try to understand what an object is in general, its importance and usage in TDL.

1. Objects

Any information that is stored in a computer is referred to as Data. Database is a collection of information organized in such a way that a computer program can quickly select desired data. A database can be considered as an electronic filing system. To access information from a database a Database Management System (DBMS) is used. DBMS allows to enter, organize, and select data in a database.

The organization of data in a database is referred to as the ‘Database Structure’. The widely used database structures are hierarchical, relational, network and object-oriented.

In the hierarchical structure the data is arranged in a tree like structure. This structure uses the parent –child relation ships to store repeating information. A parent can have multiple children but a child can have only one parent. The child in turn can have multiple children. Information related to one entity is referred to as an object. A database is a group of interrelated objects.

An object is a self-contained entity that consists of both data and procedures to manipulate the data. It is defined as an independent entity based on its properties and behavior/functionality. Objects are stored in a data base.

A relationship can be created between the objects. As discussed, the hierarchical structure has a parent-child relationship. For example, child objects can inherit characteristics from parent objects. Likewise, a child object can not exist with out a parent object.

After discussing the object concept in general, let us examine the Tally object structure in the following section.

1.1 Tally Object Structure

The Tally data base is hierarchical in nature in which the objects are stored in a tree like structure. Each node in the tree can be a tree in itself. An object in Tally is composed of methods and collection. Method is used to retrieve data from the database. A collection is a group of objects. Each object in the collection can further have methods and collection. The structure is as shown in Figure 6.1.

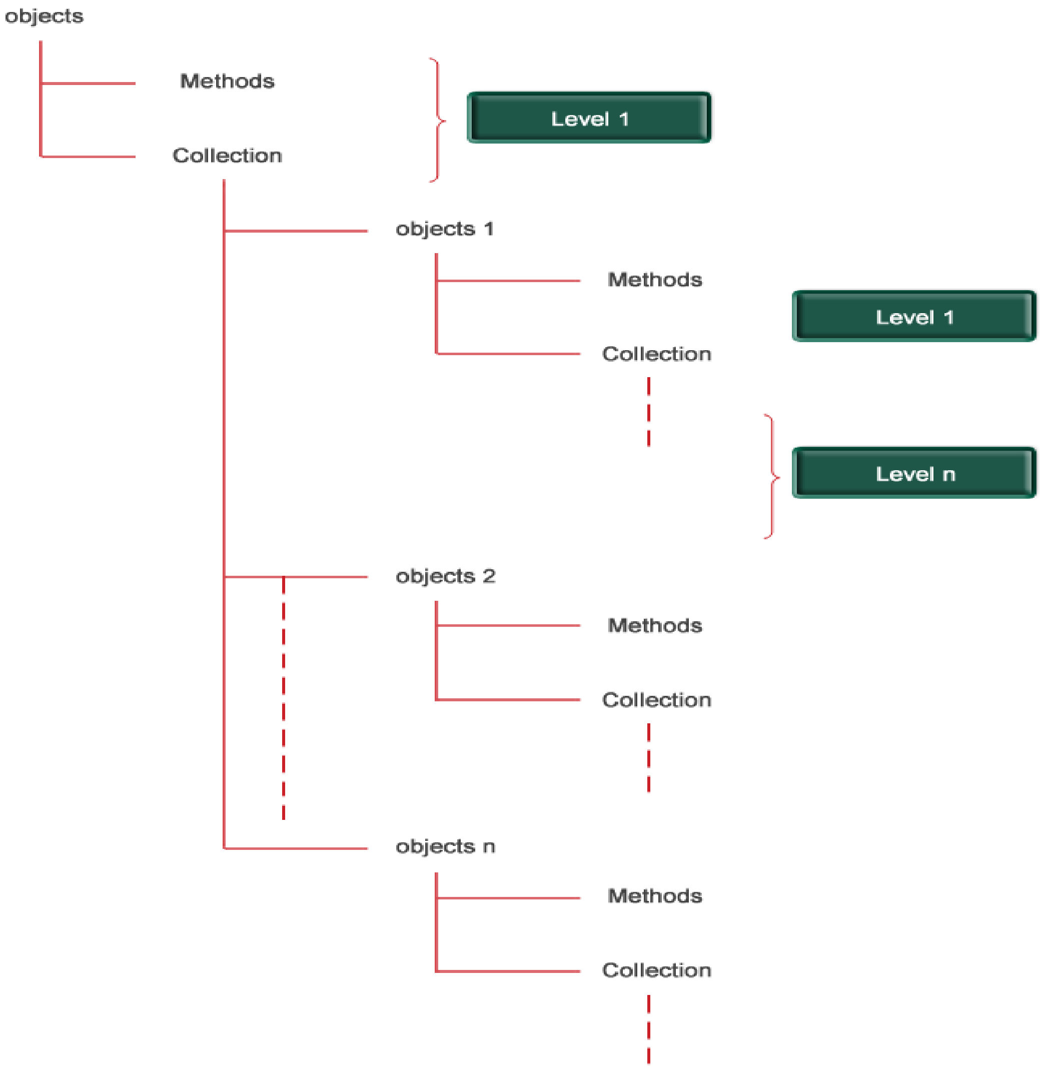


Figure 6.1 Tally Object Structure

Everything in TDL is an Object. As mentioned in the earlier chapters, Report , Menu, Company, Ledger all are objects in TDL. The properties of objects in TDL are called Attributes. For example, the attributes Object, Title, Form are all properties that define the Report object.

An object can have Methods and Collections as mentioned earlier. For example, the Object Ledger contains the Methods Name, Parent etc. and the collections Address and Billwise Details.

As shown in the Figure 6.1, the Objects available at Level 1 are referred to as Primary objects and objects which are at Level 2-n are referred as Secondary objects.

Two different types of objects are available in TDL. The following section describes the classification of objects in TDL.

1.2 Tally Objects Types

The objects in TDL are classified in two types based on the their usage and behaviour as follows :

- Interface Objects
- Data Objects

Interface objects define the user interface while Data objects store the value in the Tally Primary or Secondary database. Any data manipulation operation on the data object is performed through Interface objects only. Figure 6.2 shows the classification of objects in TDL.

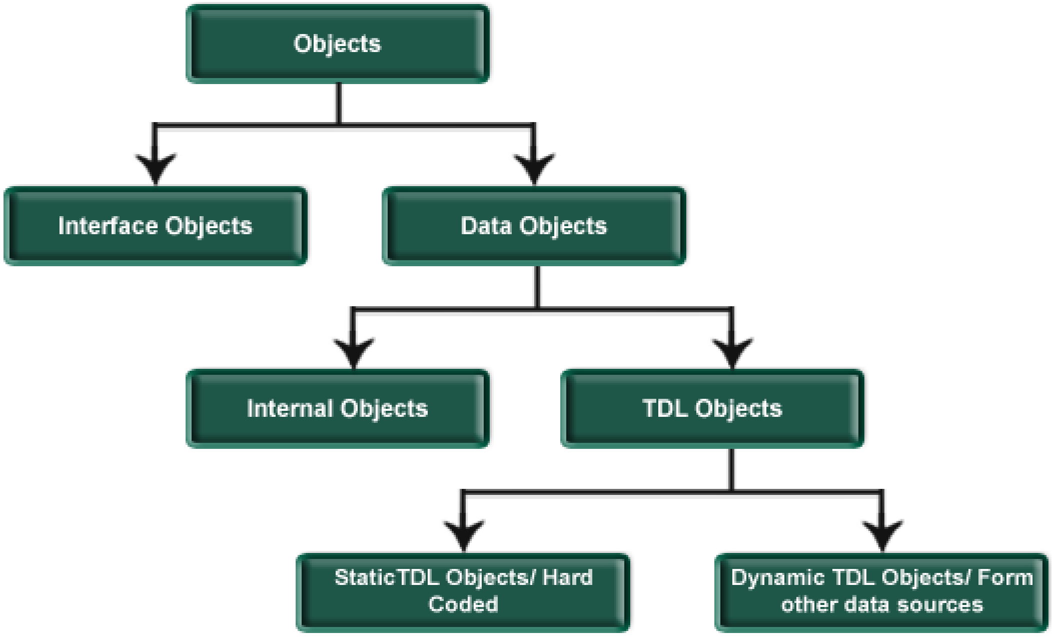

Objects
Interface Objects
Data Objects
Internal Objects
TDL Objects
StaticTDL Objects/ Hard Coded
Dynamic TDL Objects/ Form other data sources

Figure 6.2 Classification of objects

1.2.1 Interface Objects

Objects used for designing the User Interface are referred to as Interface objects. Report, Form, Menu etc. are interface objects. Interface objects like Report and Menu are independent items and can exist on their own. The objects Form, Part, Line, Field can't exist independently. They must follow the containment hierarchy as mentioned in the section Basic TDL Structure of Lesson 2 – TDL components.

Example

```
[Field: Sample Fld]
   Width: 22

[Line: Sample Ln]
   Field: Name Field
```

TDL allows a re-usage of all the objects.There are two ways to obtain some more properties that are required in an object:

- The existing object can either be used in the new objects or in lieu of defining a new object.
- The existing object can be modified to add new properties.

The interface objects can be shared by other interface objects. For example, a single field can be used in multiple lines. The following examples describe the discussed scenarios.

Example 1

```
[Field : Sample Fld]
   Width        : 22
   Set As       : "TDL Demo"

[#Field : Sample Fld]
   Style        : Normal Bold
```

The field **Sample Fld** will have both the properties. The width of the field is 22 and text is displayed using the style Normal Bold.

Example 2

```
[Field: Sample Fld]
   Width        : 22
   Set As       : "TDL Demo"

[Field: Sample Fld1]
   Use          : Sample Fld
   Style        : Normal Bold
```

The field **Sample Fld1** will have both the properties. The width of the field is 22 and the text is displayed using the style Normal Bold.

Example 3

```
[Line : TitleA]
   Field        : Name Field

[Line: TitleB]
   Field        : Name Field
```

The field **Name Field** is used in both the lines TitleA and TitleB.

A set of available attributes of interface objects are predefined by the platform. A new attribute can not be created for an interface object.

Interface objects are always associated with a Data Object and essentially add, retrieve or manipulate the information in Data Objects.

1.2.2 Data Objects

Data is actually stored in the Data Objects. These objects are classified into two types viz., Internal objects and User defined objects / TDL objects.

Internal Objects – Internal objects are provided by the platform. They are stored in the Tally Database. Multiple instances of internal objects can exist. In Tally.ERP 9, internal objects are of several types. Examples of internal objects are Company, Group, Ledger, Stock, Stock Item, Voucher Type, Cost Centre, Cost Category Budget, Bill and Unit of Measure.

User Defined Objects /TDL Objects – All the Objects which are defined by the user in TDL are referred to as User Defined Objects or TDL objects. User defined objects are further classified as Static Objects or Dynamic Objects.

Static TDL Objects cannot be stored in Tally Database. The data for the Static object is hard coded in the program and can be used for the display purpose only.

Dynamic TDL Objects can be created from the data available in any of the following external data sources:

- XML Files from remote HTTP server
- DLL files
- From any type of database through ODBC

In TDL, the data from all these external data sources is available in a collection.

1.3 Object Context

Each Interface object exists in the context of an Data Object. An Interface object either retrieves information from the Data Object or stores information onto the Data Object.

The association of the Interface object with a Data Object can be done at the Report, Part, Line and Field level. All the methods of the associated Data Object are available in the Interface object which is said to be in the 'Context' of the associated Data Object.

The data is always retrieved from the database in context of the current object. All the data manipulation operations are performed on the object in context only.

Any expression such as Formulae, Methods and so on which are evaluated in the Interface object will be in the 'Context' of the Data Object. To understand the concept of an object context consider the following example:

Example

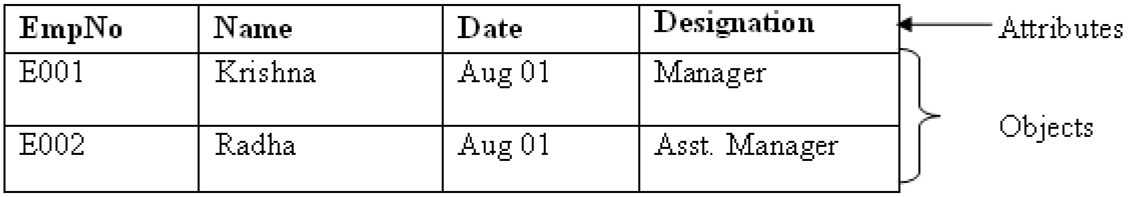
When the Interface object Report is associated to the Data Object Ledger, then all the methods and collection of the Ledger object can be referenced in the associated report. The Method $Name when used in the field will display the name of the Ledger object associated at the Report level. If no object is associated at the Report level then no data will be displayed in the field ~~as~~ since there is no object in the context.

1.3.1 Example of Internal and TDL Object

Static TDL Objects / External Objects

As discussed earlier, a user can create Static TDL Objects for which the data is hard coded. Consider the following examples of Employee Details.

Employee Details



EmpNo	Name	Date	Designation
E001	Krishna	Aug 01	Manager
E002	Radha	Aug 01	Asst. Manager

Attributes
Objects

In TDL two objects have to be created such that the EmpNo, Name, Date and Designation becomes the attribute of the object. The code snippet to create these objects is as shown.

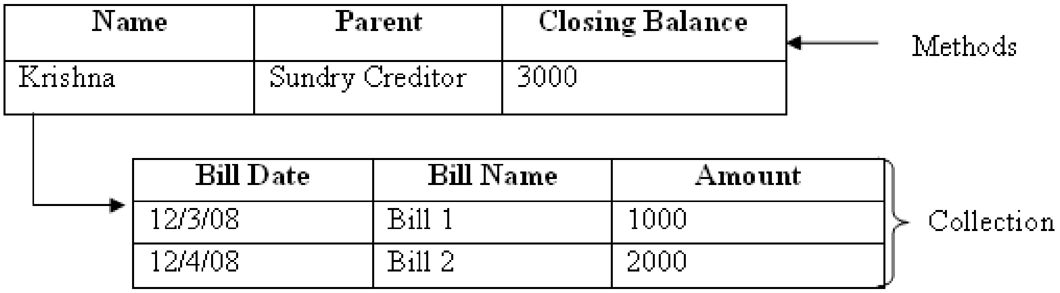
```
[Object : Emp 1]
   EmpNo       : E001
   Name        : "Krishna"
   Date        : Aug 01
   Designation : Manager
```

```
[Object : Emp 2]
   EmpNo       : E002
   Name        : “Radha”
   Date        : Aug 01
   Designation : “Asst. Manager”
```

Internal Objects

Consider the data for a ledger object which has multiple bill details associated with it.

Ledger Details



Name	Parent	Closing Balance
Krishna	Sundry Creditor	3000

Bill Date	Bill Name	Amount
12/3/08	Bill 1	1000
12/4/08	Bill 2	2000

The above hierarchical structure shows that the ledger Krishna is created under the group Sundry Creditors. It further contains multiple bill details. The Ledger Name is Krishna, the parent group is ‘Sundry Creditors’ and the closing amount is 3000. The two bills Bill 1 for the amount 1000 and Bill 2 for the amount 2000 are associated with the ledger Krishna.

Please refer to the Appendix for the detailed structure of Internal Objects and Methods.

2. Collections

A Collection is termed as a group of objects. It refers to a collection of zero or more objects. The objects in the collection can be obtained from the Tally database or from external data sources e.g. XML file.

In default TDL many collections are defined which are referred to as an Internal Collection. The collections created by a user are called user defined collection. Object in a collections follow the Tally object structure. That is each object of the collection can contain the Methods and Collection and so on.

A collection can be a collection of objects or a collection of collections.

Figure 6.3 shows the collection of objects.

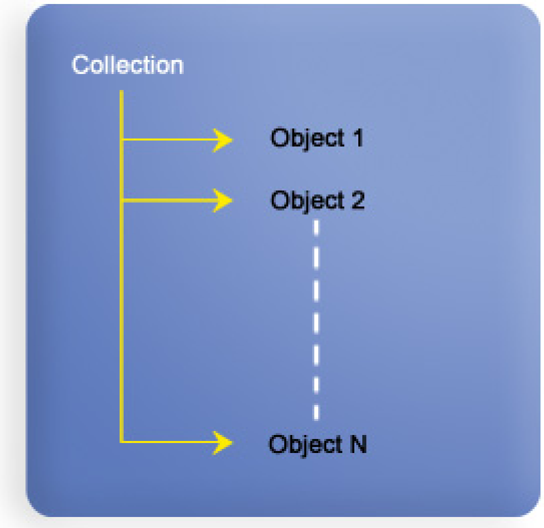


Figure 6.3 Collection of objects

The collection of collections is referred to as a Union of collection. This capability will be discussed in detail in the section Collection Capabilities.

In TDL, the collections are of two types: Simple collection and Compound collections.

2.1 Simple and Compound Collections

Collections can have multiple Methods and Collection. They are classified as Simple Collection and Compound Collection based on the constituents of the collection.

Figure 6.4 shows the classification of collection.

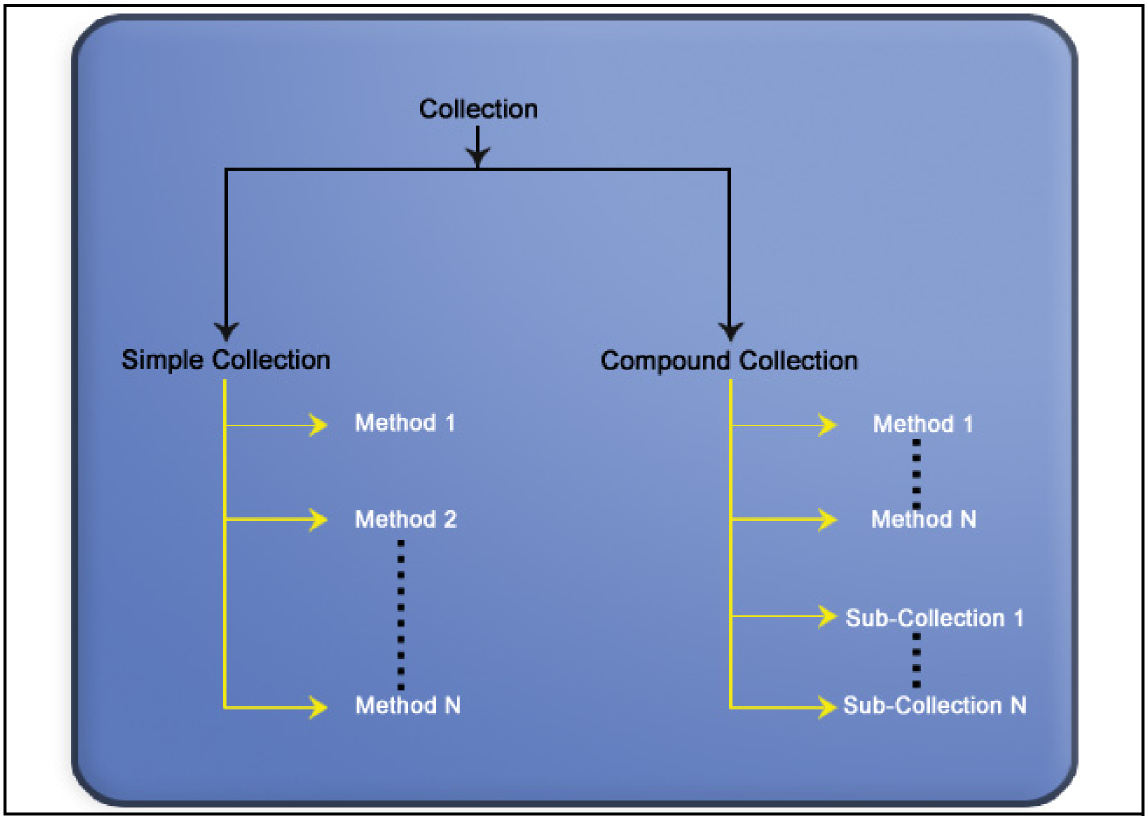


Figure 1.4 Classification of collections

Simple Collections

Simple collections only have a single method which is repeatable. Simple Collections cannot have sub-collections. The Name and Address are examples of Simple Collections.

Compound Collections

The collections which have sub-collections and multiple methods are called Compound Collection. Any Internal or External Collections of Primary or Secondary or user defined objects is an example of a Compound Collection. In both Simple and Compound Collections, the index can be used to fetch user-defined or internal methods of the Object. The Index can be either First or Last.

After describing the classification of a Collection, the following topic describes the various data sources of a Collection.

2.2 Sources of Collection

Collection, the data processing artifact of TDL provides extensive capabilities to gather data not only from Tally database but also from external sources using ODBC, DLLs and HTTP.

Based on the source of data, the collections are referred to as External collection, ODBC collection, HTTP XML collection and Aggregate/summary collection.

The Collection of Internal Objects

In cases where a collection contains objects from Tally database, it is referred to as an Internal Collection. In the collection of internal objects the attributes used are Type, Child Of, Belongs To.

External Collection

The collection of static TDL objects are referred to as an External Collection. The attribute used to create an external collection is Object.

ODBC Collection

The Data Objects populated in the collection are from an external database using ODBC. The attributes used are ODBC, SQL, SQL Objects, SQL Parms and SQL Values.

HTTP XML Collection

The Object of a collection is obtained from the XML file using HTTP. The file can be made available either on the local machine or on the remote server. The attributes used in creating an XML collections are Remote URL, Remote Request, XML Object Path and XML Object.

DLL Collection

A collection can be populated with objects obtained by executing a DLL file. The DLL’s can be written using an external application to extend the existing functionality of Tally. This allows the users to extend the kernel capability by adding their own functions.

External Plug-Ins are written as DLL’s and can be of two types:

- C++ DLL’s
- ActiveX DLL’s

In order to create the Collection that calls an external PlugIn the following attributes are used. Values can be passed to the DLL’s as parameters.

Syntax

```
[Collection: My DLL Collection]
      Plug-In          : <path to dll>.<pInput param>
      ActiveX Plug-In  : <Project Name>.<Class Name>.<pInput param>
```

The value returned by executing the DLL will be available as objects in the collection.

2.3 Creating a Collection

TDL provides a set of attributes to create a collection and populate it with objects obtained from various data sources. The set of attributes used in the collection is based on the data source as mentioned in the section Sources of Collections. This section describes the attributes used in the creation of an internal and external collection. Creating collections from various data sources will be explained later.

2.3.1 Collection of Internal Objects

To create a collection of internal objects the attribute Type is used.This attribute accepts the object type name, as a value. The collection definition for creating an internal collection has the following syntax.

Syntax

```
      [Collection: <Collection Name>]
            Type : <Object Type>
```

<Collection Name> This is a user defined name for the collection.

<Object Type> This is the name of any of the internal objects. Eg.Group, StockItem, Voucher etc.

Attribute – Type

This attribute is used to define a collection of a particular Type or Subtype. The 'Type' can take values of the default TDL objects as well as the user defined fields (UDF).

Syntax

```
      Type : <ObjectType>[: <ParentType>]
```

<Object Type> This is the name of the object type or its sub-type.

<Parent Type> This is optional and is required if the subtype is to be specified.

Example

```
   [Collection: My Collection]
      Type : Ledger
```

The code snippet, My Collection consists of a collection of Ledgers which is an Internal object.

2.3.2 External Collection

To create a collection of Static TDL objects the attribute used is Object. The collection definition for creating external collection has the following syntax:

Syntax

```
      [Collection: <Collection Name>]
            Object : <Object  Name>, <Object  Name>, …….., <Object Name>
```

<Collection Name> This is the user defined name for the collection.

<Object Name> These are names of user defined objects.

Attribute – Object

The Object attribute is used to create a collection of user defined objects. A collection can have multiple collections/objects in it.

Syntax

```
      Object : <List of Objects>
```

<List of Objects> This is a comma separated list of objects.

Here, the objects are defined using the Object definition as shown in the following example.

Example

```
[Collection : Emp]
   Object   : Emp1, Emp2
[Object : Emp1]
   EmpName : "Ram Kumar"
   Age      :  "25"
[Object : Emp2]
   EmpName : "Krishna Yadav"
   Age      :  "30"
```

The Objects of Collection Emp has the Methods EmpName and Age.

In TDL, Methods are used to retrieve data from Objects and Collections. The following section explains the Usage and Types of method.

3. Object Association

Object Association is the process of linking an Interface Object with one or more Data Objects. Each Interface Object must be in the context of a Data Object. A TDL programmer can associate an Interface Object with any Data Object. If a Interface object is not explicitly associated with any Data Object, then Anonymous Object is associated to it. Anonymous Object is a Primary data Object provided by platform. It has no methods, sub-collections, or parameters.

Object Association can be done at the following levels:

- Report Level Association
- Part Level Association
- Line Level Association
- Field Level Association

Once an Object is associated at the Top level, the child level Interface Objects inherits it, unless it is explicitly overridden. If there is no explicit association of the Data Object at the Report level, it is associated with the Anonymous Object.

In Release 3.0, the Object association becomes more natural and simpler.

3.1 Report Level Object association

A Report is normally associated with a data object, which it gets from the previous Report and if not, will be associated with the anonymous object. From Release 3.0 onwards, the syntax for association has been enhanced to override the default association as well. The Report attribute ‘Object’ has been enhanced to take an additional optional value as ‘Object Identifier Formula’.

Syntax

```
      Object: <ObjectType> [: <ObjectIdentifierFormula>]
```

Where:

<ObjectType>This is a Type of Primary Object

<ObjectIdentifierFormula> This is an optional value and ~~is~~ refers to any formula which evaluates the name of Primary Object.

Example 1: Without the Object Identifier

```
   [#Form: Sales Color]
      Delete       : Print
      Add          : Print: New Sales Format

   [Report: New Sales Format]
      Object       : Voucher
```

Default **Sales Color** Form is modified to have a new print format ‘New Sales Format’. This Report gets the voucher object from the previous Report.

Example 2: With the Object Identifier

```
   [Report: Sample Report]
      Object       : Ledger     : "Cash"
```

The Ledger ‘Cash’ is associated to the Report ‘Sample Report’. Now components of a ‘Sample Report’ by default, inherit this ledger object association.

3.2 Part Level Object Association

Part inherits the Object from the Report/Part/Line, by default. This can be overridden in two ways.

3.2.1 Using the ‘Object’ attribute specification in the Part definition

The syntax of an Object attribute at the part level is as follows :

Syntax

```
Object : <SupplierCollection> : <SeekTypeKeyword> [: <SeekCondition>]
```

Where:

<SupplierCollection> This is the name of the Collection of Secondary Objects.

<SeekTypeKeyword> This can be First or Last which denotes the position index.

<SeekCondition> This is an optional value and is a filter condition to the supplier collection.

Example: Part in the Context of Voucher Object

```
[Part: Sample Part]

   Line          : Sample Line

   Object        : InventoryEntries:First:@@StkNameFilter

   Scroll        : Vertical


[System: Formula]

   StkNameFilter : $StockItemName = "Tally Developer”
```

The first inventory entry which has stock item “Tally Developer” is associated with Part ‘Sample Part’.

Only sub-objects can be associated at part level for which the primary object is associated at the Report level. To overcome this limitation a new attribute ‘Object Ex’ is introduced at part level in release 3.0.

3.2.2 Using ‘Object Ex’ attribute specification in Part definition

The attribute Object Ex provides the ease of using enhanced method formula syntax while specifying the object association. Now even the Primary Object can be associated to a Part, which was not possible with the Object attribute of Part Definition.

Syntax

```
Object Ex   : <Method Formula Syntax>
```

Where:

<Method formula syntax> is, ***<Absolute Spec>.[<SubObjectSpec>]***

<Absolute Specification> is (***<Object Type>, <Object Identifier Formula>***). If only Absolute Spec. is given then it should end with dot (‘.’).

<Sub Object Specification> is CollectionName[Index,<Condition>]

Example 1

```
[Part: Sample Part]

   Object Ex: (Ledger,"Customer")
```

The Ledger object “Customer 1” is associated to the Part ‘Sample Part’. Since only the absolute specification used, the Object specification is ends with ‘.’

Example 2

```
[Part: Sample Part]
   Object Ex: (Ledger,"Customer").BillAllocations[1,@@Condition1]

[System: Formula]
   Condition1: $Name = "Bills 2"
```

The Secondary Object ‘Bill Allocation’ is associated with the Part ‘Sample Part’.

The Data Object associated to some other Interface Object can also be associated to a Part. This aspect will be elaborated in the section ‘Object Access via UI Object’ of the Enhancement training.

The enhanced method formula syntax is discussed in detail under the section ‘Accessing Methods’.

3.3 Line Level Object Association

An object can be associated to a Line by Part attribute ‘Repeat’. Now, the Part attribute ‘Repeat’ is enhanced to support the following.

- Extraction of collection from any Data object.
- Extraction of collection from UI Object associated Data object. This aspect will be elaborated in the section “Object Access via UI Object”.

3.3.1 Attribute - Repeat

```
Repeat : <Line Name>: <Coll Name>: [<Supplier Coll> : <SeekTypeKeyword> :
         <SeekCondition>]
```

Where:

<Coll Name> This the name of the Collection and if that Collection is present in one level down of the object hierarchy then Supplier Collection needs to be mentioned.

<SupplierCollection> This the name of the Collection of secondary Objects.

<SeekTypeKeyword> This can either be First or Last which denotes the position index.

<SeekCondition> This is an optional value and is a filter condition to the supplier collection.

Example: Part in the context of Voucher Object

```
[Part: Sample Part]
   Line        : Sample Line
   Repeat      : Sample Line: Bill Allocations: Ledger Entries: First: +
                 @@LedFormula
[System: Formula]
```

```
      LedFormula  : $LedgerName = "Customer"
```

The Line ‘Sample Line’ is repeated over Bill Allocations of first Object Ledger entries which satisfies the given condition.

Alternate Repeat

Instead of specifying the ‘<Coll Name>: [<Supplier Coll> : <SeekTypeKeyword> : <SeekCondition>]’ the new method formula syntax can be used as shown below:

Syntax

```
      Repeat : <Line Name> : <MethodFormulaSyntax>
```

Where:

<MethodFormulaSyntax> is *<Absolute Spec>.<SubObjectSpec>*

<Absolute Spec> is (*<Object Type>, <Object Identifier Formula>*)

<Sub Object Spec> is *CollectionName[Index,<Condition>]*

Example

```
   [Part: Sample Part]
      Line  : Sample Line
      Repeat: Sample Line: (Ledger, "Customer").BillAllocations
```

The Line ‘Sample Line’ is repeated over Bill Allocations of Object Ledger for Customer ledger.

3.4 Field Level Object Association

By default, it is inherited from the Parent line or Field (if field inside a field). This cannot be overridden. However Field also allows Object Specification syntax. This association if specified acts as the ‘Secondary Context Object’ for the Field. During any formula evaluation, if the formula / method fails in the context of the Primary Object, the Secondary Object is tried then.

4. Methods

Each piece of information stored in the data object can be retrieved using a method. A method either performs some operation on the object or retrieves a value from it. To retrieve the value from the database, the storage name is prefixed with the $ symbol. TDL provides a pre-defined Methods and allows the user to create methods as well.

Methods are classified as Internal or External methods.

4.1 Internal Methods

The methods which are defined by the platform are called as Internal Methods. For example the methods Name, Address, Parent are the internal Methods of Object Ledger.

4.2 User Defined/External Methods

A user can change the behaviour or perform an action on the internal object by defining new Methods. Methods defined by the user are referred to as External methods or User defined methods.

Example: A Method DiffBal can be created for an Object Ledger which gives the difference of the total debit amount and total credit amount.

4.3 Accessing Method

The Method of an object can be accessed in TDL in three different ways, based on the context of an Object.

Accessing data from the current Object

Incase you are already in the object context, use the Method name prefixed with $ directly.

Syntax

```
$<MethodName>
```

<Method Name> This is the name of the Method of the object in context.

Example

```
$CompanyName
```

Accessing by Reference

In cases where the user is not in the object context, or is in a different object context then following syntax may be used:

Syntax

```
$<Method Name>:<Object Name>:<formula>
```

<Method Name> This is the name of the Method which belongs <*object name*>.

<Object Name> This is the name of the object.

<Formula> This is the value based on which the Method value is retrieved.

Example

```
$Name:Ledger:##SVLedgerName
```

Accessing by using the Index

In cases where the user is not in the object context, or in a different object context then the following syntax may be used:

Syntax

```
$<Method Name>:< Collection Name>: <Seek Type>
```

<Method Name> This is the Name of the Method which belongs <*Collection Name*>.

<Collection Name> This is the Name of the Collection.

<Seek Type> This is the searching direction. It can either be the First or Last.

Example

```
$LedgerName:LedgerEntries:First
```

4.3.1 Directly Accessing Data from Any Object

The Method formula syntax allows direct access to any object Method including its sub-collections to any level with a dotted notation framework.The values from any object anywhere can be accessed without making the object as the current object. This syntax is introduced to support access out of the scope of the Primary Object and to access the Sub object at any level using (.) dotted notation with index and condition support.

Syntax

```
$<PrimaryObjectSpec>.<SubObjectPathSpec>.MethodName
```

Where:

<Primary Object Spec> can be (*<Primary Object Type Keyword>, <Primary Object Identifier Formula>*)

<SubObjectPathSpec> This is given as the CollectionName *[<Index Formula>, [<Condition>]]*

<MethodName> This refers to the name of the Method in the specified path.

<Index Formula> This should return a number which acts as a position specifier in the Collection of Objects matching the given *<condition>*.

Example: Assuming that the Voucher is the current object

1. To get the Ledger Name of the first Ledger Entry from the current Voucher:
```
Set As           :  $LedgerEntries[1].LedgerName
```

2. To get the amount of the first Ledger Entry on the Ledger Sales from the current voucher (Sales Invoice):
```
Set As           : $LedgerEntries[1,@@LedgerCondition].Amount
LedgerCondition  : $LedgerName = "Sales"
```

3. To get the first Bill Name of the first Ledger entry on the Party Ledger from the current voucher (Sales Invoice):
```
Set As: $LedgerEntries[1,@@LedgerCondition].BillAllocaions[1].Name
LedgerCondition  : $LedgerName = @@InvPartyName
```

4. To get the OpeningBalance of the first Bill for the Party Ledger Acme Corp:
```
Set As:  $(Ledger,@@PartyLedger).BillAllocations[1].OpeningBalance
PartyLedger     :  "Acme Corp"
```

The Primary Object specification is optional. If it is not specified, the current object will be considered as the Primary Object. A Sub-Collection specification is optional. If not specified, Methods from the current or specified primary object will be made available. The Index specifies the position of the Sub-Object to be picked up from the Sub-Collection. This Condition is 'Filter' which is checked on the objects of the specified Sub-Collection.

<Primary Object Identifier Formula>**, **<Index Formula> and Condition can be a value or formula.

The Index Formula can be any formula evaluating to a number. The Positive Number indicates a forward search while a negative number indicates a backward search. This can also be a keyword First or Last which is equivalent to specifying 1 or -1 respectively.

In cases where both the Index and Condition are specified, the index is applicable on the Object(s) which satisfies the condition so that one gets the nth Object which clears the condition. Let's say for example, if the Index specified is 2 and Condition is Name = "Sales", then the second object which matches the name Sales will be picked up.

The Primary Object Path Specification can either be Relative or Absolute. A Relative Path is referred to by using empty parenthesis () or a dotted path to refer to the Parent object relatively. A SINGLE DOT denotes the current object, DOUBLE DOT the Parent Object, TRIPLE DOT the Grand Parent Object and so on within an Internal Object. The Absolute Path refers to the path in which the Primary Object is explicitly specified.

To access the Methods of Primary Object using a Relative Path, the following syntax is used.

Syntax

```
$().<MethodName> or $..<MethodName> or $…<MethodName>
```

Example

With regard to the context of LedgerEntries Object within Voucher Object, the following have to be written to access the Date from its Parent Object which is the Voucher Object.

```
$..Date
```

To access the Methods of Primary Object using the Absolute Path :

Example

```
$(Ledger, "Cash").OpeningBalance
```

5. Collection Capabilities

Having understood the concept of Objects, Collection, Methods and Object association, let us now concentrate on understanding the concept of a Collection as a Data Processing Artifact in TDL.

In the previous sections, we have already seen that a Collection can contain objects from the Tally Database or populate objects from an external data sources as well. In the coming sections we will discuss on the capabilities of collection from the perspective of data processing capabilities. Let us segregate these capabilities into:

- Basic Capabilities
  - Union
  - Filtering
  - Sorting
  - Searching

- Advanced Capabilities
  - Extraction
  - Aggregation
  - Usage As Tables
  - Integration Capabilities using HTTP XML Collection
  - Dynamic Object Support
  - Collection Capabilities for Remoting

In this training program we will be covering the Basic capabilities in detail with all the relevant attributes and functions for achieving the same.

Some portions of Advanced capabilities which were available prior to Tally.ERP 9 will be covered here.The latest developments pertaining to this, will be covered in our training program 'TDL Enhancements for Tally.ERP 9'

5.1 Basic Capabilities

5.1.1 Union

A Collection can be created as a combination of multiple Collections. The total number of objects in the resultant Collection is the sum of objects of the subsequent Collections. This is known as a Union. The following figure shows a Collection of Sub-collection. The Sub-collection. which can further be a Union of Collections and so on.

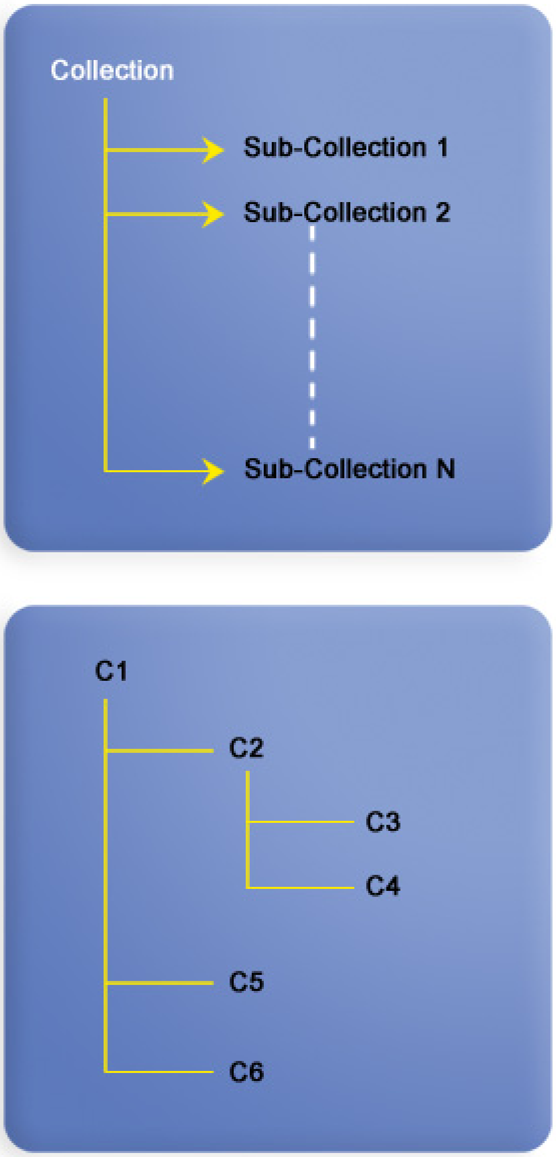


Figure 1.5 Collection of Sub-collection

The example shows that Collection C1 contains collections Collection C2 and Collection C3. Likewise, Collection C2 further contains collections Collection C4 and Collection C5.

The attribute Collection is used to create a Union as follows:

The attribute Collection

The attribute Collection is used to specify a Collection under the main Collection. All the objects belonging to the Sub collections are available in the resultant Collection.

Syntax

```
Collection : <List of Collections>
```

<List of Collections> This is a comma separated list of collection. The Collections that are used can be of different types.

Example

```
[Collection: Groupandledger]
   Collections : Group, Ledger
```

In the example above, both the Group Collection and Ledger Collection are used under the main collection Group and Ledger.

5.1.2 Filtering

This is required to retrieve only a specific set of objects from a Collection, then the collection needs to be filtered. Filtering is applied on the Collection based on a condition. All the objects which satisfy the given condition are retrieved and are available in the Collection.

Filtering Attributes

The attributes used for applying a filter are **ChildOf, BelongsTo** and **Filter**

The attribute Child Of

The ChildOf attribute helps to control the display of the contents of a collection. It retrieves only those objects whose direct parent is the string specified as the parameter of this attribute.

Syntax

```
ChildOf : <String Formula>
```

Example

```
[Collection : My Collection]
   Type          : Ledger
   ChildOf       : "Sundry Debtors"
```

It will return all the ledgers grouped directly under the group ‘Sundry Debtors’

The following definition code will return all the ledgers under the group blank. By default, Tally returns the ledger ‘Profit and Loss’

```
   ChildOf  : ""
```

The attribute BelongsTo

The attribute Belongs To is used along with the ‘Child Of’ attribute. This attribute determines whether to retrieve the first level of objects under the value specified in ChildOf or include all the objects upto the lowermost level .BelongsTo takes a logical value.

Syntax

```
      BelongsTo : <Logical Value>
```

<Logical Value> This can be either Yes or No.

Example

Consider the previous example of accounts. The following code is an extension to the previous code.

```
   [Collection: My Collection]
      Type         : Ledger
      ChildOf      : "Sundry Debtors"
      BelongsTo    : Yes
```

This code will retrieve all the objects directly under the group ‘Sundry Debtors’ as well as all the objects which are under the sub groups of ‘Sundry Debtors’.

The attribute Filter

The attribute ‘Filter’is used to specify the condition for filtering the objects. The Filter attribute takes a system formula. The condition is specified in the formula. If more than one filter has to be specified it can be separated by a comma.

Syntax

```
      Filter : <FilterName>
```

<Filter Name> This is the name of the global formula.

Example

```
   [Collection : LtdDebtors]
      Type         : Ledgers
      ChildOf      : “Sundry Debtors”
      Filter       : NameFilter
```

```
[System: Formula]
   NameFilter : $Name contains "Ltd" OR $Name contains "Limited"
```

The Namefilter is used to fetch only those objects whose name contains the string “Ltd” or “Limited” .

Filtering Functions

The function FilterAmtTotal

This is used to get the sum of the value returned by the specified expression when applied to all the Objects that satisfy the given filter expression in a Collection. The value returned is of the type Amount.

Syntax

```
       $$FilterAmtTotal:<CollectionName>:<FilterExpression>:<ValueExpression>
```

<CollectionName> This is the name of a Collection~~,~~

<FilterExpression> This is a System Formula.

<Filter Expression> This is evaluated for each Object and the resultant Objects that clear the filter are selected for further processing.

<ValueExpression> is any valid expression to be evaluated on each Object of the Collection.

Example

```
$$FilterAmtTotal : AllLedgerEntries : CashBankEntries : $Amount

[System :Formula]
   CashBankEntries : $$IsCashLedger : $LedgerName AND $$IsDr : $Amount
```

The filter in the example, checks whether the ledger is a Cash Ledger and the amount is of the type debit. **$$IsCashLedger** is a logical Function which checks whether the argument passed is a Cash Ledger or not. This statement can be evaluated only in the context of a Voucher Object.

The function FilterQtyTotal

This is similar to FilterAmtTotal except that the ValueExpression should evaluate to the type Quantity.

Syntax

```
      $$FilterQtyTotal:<CollectionName>:<FilterExpression>:<ValueExpression>
```

<CollectionName> This is the name of a Collection

<FilterExpression> This is a System Formula.The Filter Expression is evaluated for each Object and the resultant Objects that clear the filter are selected for further processing.

<ValueExpression> This refers to any valid expression to be evaluated on each Object of the Collection.

The Function FilterCount

This function is used to get the total number of Objects in a Collection after the filters are applied.

Syntax

```
        $$FilterCount : <CollectionName> : <FilterExpression>
```

<CollectionName> This is the name of a Collection,

<FilterExpression> This is a System Formula

Example

```
   $$FilterCount  :AllLedgerEntries:HasBankEntry > 0
   [System : Formula]
      HasBankEntry : ($$IsDr:$Amount != $IsDeemedPositive:+
                      VoucherType:$VoucherTypeName)+
                      AND($$IsLedOfGrp:$LedgerName:$$GroupBank+
                      OR $$IsLedOfGrp:$LedgerName:$$GroupBankOD)
```

The example confirms whether the voucher has any Ledger under the Group Bank or BankOD. IsLedOfGrp Function accepts two parameters and returns as true if parameter 1 is a ledger of a Group mentioned in parameter 2. GroupBankOD and GroupBank are functions which returns the name of the reserved groups Bank OD and Bank.

The Function FilterValue

This function is used to get the value of a specific expression based on the position specified in the set of objects filtered by the expression.

Syntax

```
        $$FilterValue     :<ValueExpression> : <CollectionName> :
                           <PositionSpecifier> : <FilterExpression>
```

<CollectionName> This is the name of the Collection.

<FilterExpression> This is the filter applied to get a set of filtered Objects.

<PositionSpecifier> This denotes the position.

<ValueExpression> This refers to any valid expression to be evaluated on each Object of the Collection.

Example

```
   $$FilterValue:$LedgerName:LedgerEntries:First:IsPartyLedger
```

The example filters all the objects within LedgerEntries to satisfy the filter condition IsPartyLedger and returns the first value of the requested method LedgerName that satisfies the condition.

Some other Functions Used

GroupSundryDebtors

It returns the name of the group Sundry Debtor even if the user re names it.

Syntax

```
      $$GroupSundryDebtors
```

Example

```
   [Collection :Sample Coll]
      Type        : Ledgers
      Child Of    : $$GroupSundryDebtors
```

The above code will populate the Collection Sample Coll with all the objects that are under the Group “Sundry Debtor”.

In case the user has renamed the group “Sundry Debtors” as “My Sundry Debtors”, the following code snippet won’t have any objects in the collection.

```
   [Collection :Sample Coll1]
      Type        : Ledgers
      Child Of    : "Sundry Debtors"
```

But in this case, the $$GroupSundryDebtor will populate the collection with all the objects that are under the Group Sundry Debtor even if the user renames the group.

GroupSundryCreditors

It returns the name of the group Sundry Creditor even if the user re names it.

Syntax

```
      $$GroupSundryCreditors
```

Example

```
   [Collection :Sample Coll]
      Type        : Ledgers
      Child Of    : $$GroupSundryCreditors
```

The above code will populate the Collection Sample Coll with all the objects that are under the Group “Sundry Creditor”.

In case the user has renamed the group “Sundry Debtor” as “My Sundry Debtors”, the following code snippet won’t have any objects in the collection.

```
[Collection :Sample Coll1]
   Type        : Ledgers
   Child Of    : "Sundry Creditors"
```

But in this case the $$GroupSundryCreditor will populate the collection with all the objects that are under the Group "Sundry Creditor" even if the user renames the group.

5.1.3 Sorting

Sorting refers to the arrangement of objects in a specific order within the collection.

The ordering is done on the basis of a specific method and the sort order can either be ascending or descending. The attribute Sort is used for this purpose.

The attribute Sort

A collection can be sorted by specifying the sort sequence using the 'Sort' attribute. The collection can be sorted by a combination of fields in ascending as well as in descending order.

Syntax

```
      Sort: < Sort Name> : <List of Methods>
```

<List Of Methods> Is a comma separated list of methods. The sorting will be done on the basis of value of the methods. The default sort order is ascending. Prefix the Methods name with '-' for the descending sort order.

Example

```
[Collection: My Collection]
   Collections : MyLedger
   Sort        : Default     : $ClosingBalance, $Name
```

5.1.4 Searching

Collection capabilities have been enhanced to enable the indexing of objects based on a particular method. Whenever a collection is indexed on a particular method, it allows instant access to the corresponding values without the need for a complete scan.

The Attribute –Search Key

Syntax

```
      Search Key : < Combination of Method name/s >
```

This implies that a unique key is created for every object which can be used to instantly access the corresponding objects and its values.

The function which is used to retrieve the values from a collection based on the key specified is expressed as the $$CollectionFieldByKey.

The Function CollectionFieldByKey

Syntax

```
$$CollectionFieldByKey:<Method Name>:<Key Formula>:<Collection Name>
```

Where:

<Method Name> This is the name of the method.

<Key Formula> This is a formula that can be mapped to the methods defined in the search key exactly in the same order.

Example

```
[Collection: My Ledgers]
   Type         : Ledger
   Search Key   : $Name

[Field : My Closing Bal Field]
   Set as       : $$CollectionFieldByKey:$ClosingBalance:@MySearchKey:+
                  My Ledgers
   MySearchKey : #LedName
```

In the above example we have defined a search key on $name for the collection **MyLedgers**. In the Field the value $Closing Balance is retrieved based on the name of the ledger. In this case the retrieval is much faster as compared to ordinary retrieval.

This capability is particularly useful in the case of matrix reports ie when two or more dimensions need to be represented as rows and columns. In that case defining the search key on a method combination and using $$CollectionFieldByKey for value retrieval improves the performance drastically.

The usage and examples based on the explanation above will be covered in detail in our training program “TDL Enhancements for Tally.ERP 9”

5.2 Advanced Capabilities

5.2.1 Extraction and Chaining

The Collection capabilities have been enhanced to extract information from the collection using other collections including its sub-objects with the choice of method(s), filter(s) and sort-order. Specific attributes have been added at the collection level to achieve the same.

Prior to Tally.ERP 9, extraction was possible using specific function *$$CollectionField.*

The Function CollectionField

This is used to get the value of a specified expression as applied on the nth Object of a Collection.

Syntax

```
        $$CollectionField:<ValueExpression>:<PositionNumber>:<CollectionName>
```

<CollectionName> This is the name of a collection.

<ValueExpression> This is any valid expression to be evaluated on the element at position <PositionNumber> in the collection.

Example

```
    $$CollectionField:$Amount:1:AllLedgerEntries
```

The example returns the first value of the Method, Amount from AllLedgerEntries Object

This function affects the performance of the report in terms of time taken to display the report.

A detailed discussion on the enhancements for extraction, chaining and reuse will be covered in the training program "TDL Enhancements for Tally.ERP 9"

5.2.2 Grouping & Aggregation

A major technological advancement in this release of Tally.ERP 9 is "Data Roll up in TDL Collection – GROUP BY", which is a part of the TDL language capabilities. This is a milestone achievement over the past 10 years. This will now facilitate the creation of large summary tables of aggregations in a single shot using the new attributes of the Collection description.This allows us to Walk down the object hierarchies and gather values to summarizes them in one scan. Overall, it reduces the TDL code complexity, resource require¬ment and increases performance drastically in case of reports generated using this new capability.

The attributes used for extraction, chaning,aggregation and grouping are Walk, By, Fetch, Compute, AggrCompute. A detailed discussion on the enhancements for aggregation and Grouping using the new attributes will be covered in the training program "TDL Enhancements for Tally.ERP 9"

Prior to Tally.ERP 9, the totals were generated using the Total and aggregation functions like CollAmtTotal or FilterAmtTotal on collections. These have certain advantages and disadvantages. While they provide excellent granularity and control, each call is largely an independent activity to gather the data set and then aggregate it. This significantly affects the performance of the reports.

Let us now discuss the various functions which are available for summarization and aggregation.

The Function CollAmtTotal

This function is used to get the sum of values of Type Amount returned by a specified expression when applied to all Objects in a given Collection. The return value is of type Amount.

Syntax

```
$$CollAmtTotal:<CollectionName>:<ValueExpression>
```

<CollectionName>This is the name of a Collection

<ValueExpression> This is any valid TDL expression to be evaluated on each Object in the Collection.

Example

```
$$CollAmtTotal:LedgerEntries:$Amount
```

This code snippet gets the sum of values in the Method Amount after it is applied on each Object in the Collection LedgerEntries. This statement will hold good only when you are in the context of Voucher Object.

The Function CollQtyTotal

This function is to get the sum of values of Type Quantity returned by the specified expression when applied to all Objects in a given Collection. The value returned is of a Type Quantity.

Syntax

```
$$CollQtyTotal:<CollectionName>:<ValueExpression>
```

<CollectionName> This is the name of a Collection

<ValueExpression> This refers to any valid TDL expression to be evaluated on each Object of the Collection.

Example

```
$$CollQtyTotal:InventoryEntries:$BilledQty
```

Each Inventory entry in the current Voucher Object is picked up and the Method BilledQty is evaluated on it. The resultant quantity is summed up to get the result of the statement.

The Function CollNumTotal

This function is used to get the sum of values of Type Number returned by the specified expression when applied to all Objects in a given Collection. The value returned is of the Type Number.

Syntax

```
$$CollNumTotal:<CollectionName>:<ValueExpression>
```

<CollectionName> This is the name of a Collection

<ValueExpression> This refers to any valid expression to be evaluated on each Object of the Collection.

Example

```
        $$CollNumTotal:InventoryEntries:$Height
```

Each Inventory entry in the current Voucher Object is picked up and the Method Height evaluated on it. The resultant height is summed up to get the result of the statement. Here, Height is an external Method of Object Inventory Entry in a Voucher.

5.2.3 Usage as Tables

TDL allows you to display the values obtained from the collection as a pop-up table. Earlier the values of voucher and the ODBC data can't be displayed as a collection. Now all limitations pertaining to usage of Collections as Tables have been completely eliminated. Any collection which can be created in TDL can be displayed as a table now. Collection with aggregation and XML Collections can also be used as Tables.

5.2.4 Integration Capabilities using HTTP XML Collection

The Collection capability has been enhanced to gather live data from HTTP/web-service delivering XML. The entire XML is automatically now converted to TDL objects and is available natively in TDL reports as $ based methods. Reports can be shown live from an HTTP server. The attributes in collection for gathering XML based data from a remote server over HTTP are RemoteURL, RemoteRequest, XMLObjectPath, and XMLObject.

A detailed discussion on this will be covered in the training program "TDL Enhanceemnts for Tally.ERP 9"

5.2.5 Dynamic Object Support

When a collection is used for editing (alter/create), objects are dynamically added to the collection when a new line is repeated over the same. The type of object which is added depends on the specification in the TYPE attribute. In case the TYPE attribute is not specified it defaults to adding a standard empty object.

However the following holds true for a COLLECTION keeping in mind the latest enhancements:

- It can be made up of multiple types of objects (say Ledgers and Groups)
- It can have TDL defined objects which are retrieved from XML file.They are specified using an XML Object
- It can have aggregated objects

Depending solely on the TYPE attribute to make a decision, the object type is a constraint with respect to the above facts. This is now being removed with the introduction of a new attribute which will independently govern the type of object to be added to the collection on-the-fly. The following is now supported in a collection.

```
        NEWOBJECT: type-of-object: condition
```

A detailed discussion on the subject can be accessed from our training program "TDL Enhancements for Tally.ERP 9"

5.2.6 Collection Capabilities for Remoting

Enabling access to your organizational data in an ‘any-time, any-where’ and yet being truly usable is what Tally.ERP 9 delivers.With Tally.NET enabled remote access, it will be possible for any authorized user to access Tally.ERP 9 from anywhere.

Major Enhancements have taken place at the collection level to achieve remoting capabilities.The attributes Fetch,Compute and AggrCompute provided at the Collection level and FetchObject and FetchCollection at the Report level significantly help in above functionality.

A detailed documentation on “Writing TDL Compliant Reports” can be downloaded from our website.

Learning Outcome

- An object is a self-contained entity that consists of both data and procedures to manipulate the data.
- Objects are stored in a data base.
- Tally data base is hierarchical in nature in which the objects are stored in a tree like structure.
- Everything in TDL is an Object.
- Objects used for designing the User Interface are referred as interface objects.
- Data is actually stored in the Data Objects. Data object are classified in two types namely Internal objects User defined objects / TDL Objects.
- A collection can be a collection of objects or a collection of collections.
- Collection, the data processing artifact of TDL provides extensive capabilities to gather data not only from Tally database but also from external sources using ODBC, DLLs and HTTP.
- In TDL, Object association can be done at following levels:
  - Report Level Association
  - Part Level Association
  - Line Level Association
  - Field Level Association
- Each piece of information stored in data object can be retrieved using a method. Methods are classified as internal or external methods.
- Union, Filtering, Sorting and Searching are the basic capabilities of collection.
- Extraction, Aggregation, Usage As Tables, Integration Capabilities using HTTP XML Collection, Dynamic Object Support and Collection Capabilities for Remoting are the advanced capabilities of collection.

Actions in TDL

Introduction

TDL is an event driven language. Events can be triggered through a Keyboard shortcut or a Mouse click. In an event, some predefined actions get executed. For example:

- The Ctrl+A Key pressed from a voucher accepts the Entry Screen
- Clicking on the F1 Button from the Gateway of Tally Menu results in the pop up of the Company Selection Screen.

Actions are activators of a specific task with a definite result. An action always originates from a User Interface Objects Menu, Form, Line or Field.

1. Categories of Action

Actions can be classified into two broad categories viz.,

- Global Actions
- Object Specific Actions

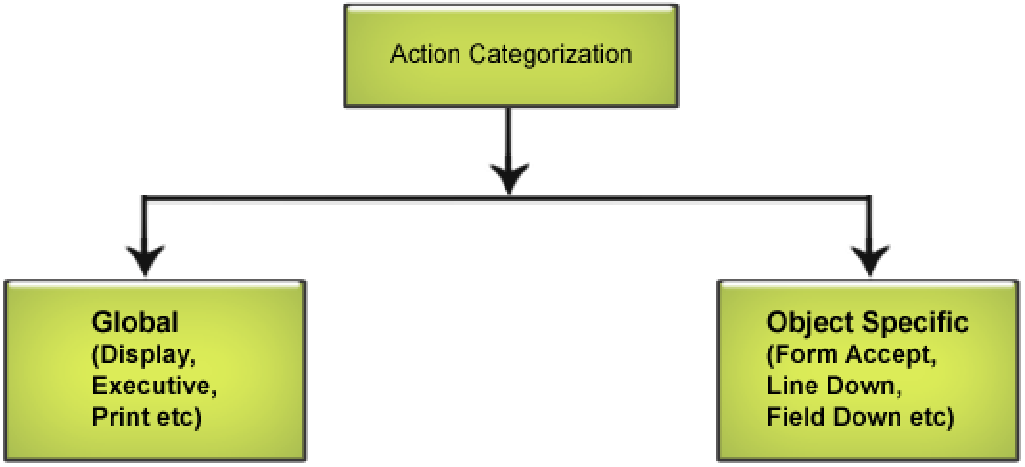


Figure 1.1 Action Categorization

Global Actions are not specific to any User Interface Object. For Example, Display, Create, Execute, Alter, etc. are Global Actions. They perform the action specified irrespective of the UI Object. Global Actions are performed on a Report or a Menu.

Object Specific Actions are actions which can act only upon specific UI Objects. For example, Line Down is a Part Specific Action since Part owns multiple lines and an individual Line cannot move the current focus to the subsequent line. Only Part can move the focus to the subsequent line. Object Specific Actions are performed on relevant User Interface Objects.

Global Actions	**Object Specific Actions**
Global Actions are not specific to any User Interface Object	These Actions are specific to any User Interface Object
Global Actions can be originated by a Menu, Button/ Key or a Field	Object Specific Actions can be originated by a Menu, Form, Line or a Field
Global Actions are performed on a Report or a Menu	Object Specific Actions are performed on relevant User Interface Objects
Example: Create, Display, Alter, Print, Print Report, Modify Object, Display Collection, etc.	Example: Line Up, Line Down, Explode (Actions - Line Object); Form Accept, Form Reject (Actions - Form Object), etc

TABLE 7:1 Action Categorisation

2. Action Association

Actions can be associated at various levels.

2.1 Action Association at Menu Definition

Action Association at Menu Definition is done through the Menu Item. Every Menu Item except Quit is associated with an Action. If an Item is added without any action, then the default action associated is to exit from the current Menu.

Syntax

```
    [Menu: <Menu Name>]

        Add : Key Item: [Position] : <Display Item> : <Unique Key> :
              <Action Keyword> : <Action Parameter>
```

Where:

- The Action Keyword can be any Global Action
- The Action Parameter is decided by the Action Keyword. If the Action Keyword is Menu, then the Action Parameter necessarily has to be a Menu Name else it has to be a Report Name.

Example

```
[Menu: Commonly Used Reports]
      Add: Key Item: Trial Balance : T : Display : Trial Balance
      Add: Key Item: At Beginning : Outstandings : O : Menu   : Outstandings
```

In the above example, a Menu Commonly Used Reports is defined with 2 Items, viz.,

1. An Item Trial Balance is added displaying default report Trial Balance. Here, the Action is display, so the Action Value has to be a Report Name.
2. An Item Outstandings is added at the beginning to activate another Menu Outstandings. The Action here is a Menu, so the Action Value required is a Menu Name.

2.2 Action Association at Button/Key Definition

Action Association at Button/Key definition is done using the attribute Action followed by the Action Keyword with the parameters, if required.

Syntax

```
[Button: <Button Name>]
      Action: Action Keyword [: Action Parameters]
```

Where:

- The Action Keyword can be any Global or Object Specific Action
- The Action Parameter is decided by the Action Keyword. If the Action Keyword is Menu, then the Action Parameter necessarily has to be a Menu Name, else it has to be a Report Name.

Example: Actions with Parameters

```
[Button: Outstandings]
         Key    : F5
         Action: Menu: Outstandings
[Button: Trial Balance]
         Key    : F6
         Action: Display: Trial Balance
```

Action Menu requires a Menu Name as a Parameter and Actions Create, Display, Alter, etc requires a Report Name as a Parameter.

Example: Actions without Parameters

```
[Button: Printing Button]
         Action: Print Report
[Button: Exporting Button]
         Action: Export Report
```

Action Parameters for the Actions, Print Report and Export Report is not mandatory.If the Action Parameter is specified, then it prints the specified Report other than the Report associated with the current object else it prints the current Report.

2.3 Action Association at Field Definition

Action Association at Field definition is done using the Action Keyword with the parameters and the optional condition

Syntax

```
      [Field: <Field Name>]
            Action Keyword: <Action Parameters>[: Condition]
```

Where :

Action Keyword This can be both, Global or Object Specific Actions

<Action Parameters> can be the Value on which these Actions could be performed

Condition This is Optional. This restricts the action to perform only if the Condition returns True

Example

```
   [Field: My Trial Balance]
      Display     : Group     : $$IsGroup
      Display     : Ledger    : $$IsLedger
```

In the example above, the Field Trial Balance has 2 statements, viz.,

1. Displaying a Group if the current object in context is a Group
2. Displaying a Ledger if the current object in context is a Ledger

3. Components of Actions

Any Action is always executed with respect to two contexts:

- Originator
- Executer

The **Originator** is one that originates the Action viz., Menu, Form, Line or Field.

For example, a Down Arrow Key pressed. The event is passed from the current Report to the associated Form, Parts, Lines or Fields. Keys could be possibly associated either in a Menu, Form, Line or Field. If the activated Key is found in Form, it searches further for the Line Association, and then continues till Field. The Lowest Level Key Association gets the highest precedence. If the same Key is associated at a Form as well as a Field, the Key Association at the Field Level will get executed. In this case, the Field is the Originator.

The **Executer** is one on which the action is executed. For example, Form Accept Key though attached at Field Level is a Form Action. Hence, Form is an executer of the action. In case of execution, it searches from Report to the Field for the action to be executed. Line Down is a Part

Level Action though associated at the Form will be executed by the Part to move the current focus to the subsequent line. Hence, Part is an executer of the Action Line Down.

Originator	Executer
The Originator initiates the action by associating the Key or a Button	The Executer executes the action associated with the Key or a Button initiated by the originator
Global Actions can be originated by a Menu, Button/ Key or a Field whereas Object Specific Actions can be originated by a Menu, Form, Line or a Field	Global Actions can be executed by the one which has originated the Action. However, Object Specific Actions can be executed by the Objects that have not originated the Action
The sequence followed to gather all the Keys originating within a Report is Top to Bottom i.e., from a Report to a Field Definition. The lowest in the hierarchy gets the highest precedence. For example, if the same key is associated in both Form as well as Field Definition, the Key at Field Definition will be considered for execution	The sequence followed to consume the Keys originated is from Bottom to Top i.e., from a Field to a Report Definition. In other words, the lowest in the hierarchy get the highest preference. For example, if the same key is relevant for both Part as well as Line Definition, the Key will be executed in the context of the Line Definition
Example 1 [Key: Create Ledger] Key : Alt + C Action : Create : Ledger [Field: CST Supplier Ledger] Key : Create Ledger Associating the Key with the Field, Field is the originator as well as executer in this case	**Example 2** [Key: Part Display PgUp] Key : PgUp Action : Part PgUp [System: Form Keys] Keys : Part Display PgUp Key Part Display PgUp is originated by Form but its executer is the Part

TABLE 7:2 Components of Actions

4. Global Actions

As discussed above, Global Actions are Actions that are not specific to any UI Object. Global Action provides an indication to the TDL Interpreter as to which specific task should be executed to fulfill the user requirements. Global Actions are mainly performed on three principal definition types namely Report, Collection and Menu. Some of the frequently used Global Actions are discussed below:

4.1 Action — Menu

A Menu Action acts only on the Menu Definition and vice versa. The value of a Menu Action must be a Menu Name. This Menu has to be further defined to list the Items displaying another Menu or a Report. A Menu Definition continues until all the Items are used to display Reports and there are no further Menu Actions assigned to the final Menu Items.

Example 1

```
;; The following code demonstrates the usage of Action Menu along with further Menu Definitions
[#Menu: Gateway of Tally]

   Add : Key Item : Sample Final Accounts : F : Menu : Sample Final Accounts

[Menu: Sample Final Accounts]

;; Menu Definition to display when the above Item is activated
   Add : Key Item : Trial Balance : T  : Display : Trial Balance

   Add : Key Item : Profit & Loss : P  : Display : Profit and Loss

   Add : Key Item : Balance Sheet : B  : Display : Balance Sheet
```

In the example mentioned above:

- The Default Menu **Gateway of Tally** is altered to add a new Item Sample Item with Menu action displaying the **Sample Final Accounts** Sub Menu.
- The Sub Menu **Sample Final Accounts** is defined to display all the components of Final Accounts i.e.,
  - Trial Balance
  - Profit & Loss
  - Balance Sheet
- All the Items here, use the Display Action. Hence, no further Menu Definition is required.

As seen in the previous Topic Objects, Methods and Collections, Display Action takes the Report Name as its parameter and is used to display the Reports as is defined.

Example 2

```
;; The following code demonstrates the usage of Menu and Display Actions and also the
;; the relevance of their association in Menu and Reports (through Form)

[Button: Final Accounts]

;; Button Definition to activate a Menu
   Key          : F5

   Action       : Menu: Sample Final Accounts

/* Since the above Button activates a Menu, it can be acted only upon a Menu
 It cannot be associated to a Report */
[#Menu: Gateway of Tally]

   Buttons    : Final Accounts;; attaching a Button to the Menu
```

```
[#Form: Group Summary]
   Buttons    : Final Accounts
;; Above is an incorrect association as Buttons triggering Menu Action cannot
;; be attached to a Form.

[Button: Balance Sheet]
;; Button Definition to Display a Report
   Key        : F6
   Action     : Display: Balance Sheet

/* Since the above Button activates a Report, it can be associated both to a Menu as well as a Report */

[#Form: Group Summary]
   Button       : Balance Sheet;; attaching a Button to the Report
[#Menu: Display Menu]
   Button       : Balance Sheet;; attaching a Button to the Menu
```

In the Example mentioned above:

- A new Button **Final Accounts** is added to activate a Menu Sample Final Accounts which is attached to the default Menu Gateway of Tally.
- The Button **Final Accounts** cannot be attached to a Report since a Menu cannot be acted upon in a Report. In the above example, the Button **Final Accounts** is attached to the Form Group Summary which is incorrect since the Menu cannot be called from a Report.
- Another Button **Balance Sheet** is added to display a Report **Balance Sheet** which is enabled in all the Reports using the Form **Group Summary** and also in the Menu **Display Menu**.
- The Button **Balance Sheet** can be attached both to a Report as well as to a Menu since the Report can be acted upon by a Report as well as a Menu.

4.2 Action – Create and Alter

Create and Alter Action acts only upon the Report Definition. These actions activate the Report in Create or Alter Mode. In other words, the Report is started in the Edit Mode. In case of Create Action, the user enters the Report in order to add values whereas in case of Alter, the user enters the Report to modify the already created values.

These actions help the user to key in the relevant values. The values thus entered may or may not be stored. The treatment of values depend on the need. The values thus entered in the Report by the user if required to be retained can be stored as a part of Tally Database or Configuration File.

- As discussed in the Topic on Variables, all the persistent variable values can be stored in a Configuration File Tallysav.TSF for subsequent sessions.
- The values entered in the Report can also be stored as a part of the Tally Database

To store the values as a part of Tally Database, the Report must be associated to a Data Object. For example, Group, Ledger, Voucher, etc. are some of the Data Objects available in Tally.

For instance, in order to design an interface to create a Ledger:

- The Object Ledger must be associated to the Report using Report Attribute Object
- Values entered by the user in the Fields within the Report must be stored in relevant Methods using Field Attribute, Storage

Example

/ The following code demonstrates the usage of Action Create and Attribute Storage at Field. Definition to store the values entered within the relevant Object associated at Report Level*/*

```
[#Menu: Gateway of Tally]
   Add       : Key Item: Ledger Creation: L : Create: Create Ledger


[Report: Create Ledger]
   Form      : Create Ledger
   Object    : Ledger
```

;; Object Association done at Report Level

```
[Form: Create Ledger]
   Parts     : Create Ledger

[Part: Create Ledger]
   Lines     : Store LedgerName, Store LedgerGroup

   [Line: Store LedgerName]
      Fields: Short Prompt, Name Field
      Local : Field: Short Prompt: Info: "Name :"
      Local : Field: Name Field: Storage: Name
```

/ Storing the value entered by user in an Internal Method Name available within the Object associated at the Report Level*/*

```
   [Line: Store LedgerGroup]
      Fields       : Short Prompt, Name Field Local: Field:+
      Short Prompt: Info: "Under :"
      Local        : Field     : Name Field: Storage: Parent
      Local        : Field     : Name Field: Table: Group
```

/ Similarly, Parent Method is stored with the user entered value which is considered as the Group of the Ledger created. Also Group is a default Table/ Collection to display all the default as well as user defined Groups. Field Attribute Table helps to restrict the user input to a predefined list*/.*

In the example mentioned above:

- The Default Menu, Gateway of Tally have been altered to add a new Item Ledger Creation which allows the user to create a Ledger
- Report Create Ledger associates the Object Ledger to it which indicates that the Report is meant for creating an instance of the Object Ledger.
- The Name and Group of the Ledger are stored in the Internal Methods Name and Parent which stores.

Example

;; The following code demonstrates the usage of Alter Action at Button

```
[Button: My Reco Button]
```

;; Button meant to do Bank Reconciliation

```
   Key       : Alt + F5
   Action    : Alter: Bank Recon
```

;; Alter Action to trigger Bank Recon Report in Alter Mode

```
   Title     : "Reconcile"


[Form: My Bank Vouchers]
    Button   : My Reco Button
```

;; Associating the Button to the Report

In the example mentioned above:

- The Button **My Reco Button** is defined with an alter action to alter the Report Bank Recon on pressing the Alt + F5 Key. This button is used for entering dates in the Bank Reconciliation Report.
- The Button **My Reco Button** is associated to the Form **My Bank Vouchers**

Example

;; The following code demonstrates the usage of Alter Action at Field

```
[#Menu: Gateway of Tally]
   Add       : Key Item: Ledger Display : L : Display: My Ledger


[Report: My Ledger]
   Form      : My Ledger


[Form: My Ledger]
   Parts     : My Ledger
   Height    : 100% Page
   Width     : 100% Page

[Part: My Ledger]
   Lines     : My Ledger
   Repeat    : My Ledger: Ledger
;; Ledger is a default collection of Ledger Object
   Scroll    : Vertical

   [Line: My Ledger]
      Fields: My Ledger
      Key   : Line Object Enter Alter, Line Click Object Enter Alter
;;The above default Keys act upon Line Definition and the action Alter Object is associated with the Keys provided the current Report is in Display Mode
      [Field: My Ledger]
         Set As   : $Name
         Variable : Ledger Name
;;Variable Ledger Name retains the Ledger selected by the user for the subsequent report
         Alter: Create Ledger
;; Alter Action is used to activate the Report in Alter Mode
;; Create Ledger is user defined Report defined while Ledger Creation
```

In the example mentioned above:

- Two default Keys associated to a Line Definition that allows allows a selection of any of the lines is being to be repeated.
- Action associated with these Keys is Alter Object which means on hitting the Key, the Object associated with the current Line must be altered.
- Mode: Display is specified in these Keys signifies that the current report must be in Display Mode.
- Alter Action used at the Field definition prompts the report from being activated on the current field which must be in Alter Mode.

4.3 Action — Modify Object

The Action **Modify Object** alters the methods of an Object at any level in the Object Hierarchy. Modify Object action supports modifying multiple values of an Object by a comma separated by the specification of Method: Value pair.

Syntax

```
Action : Modify Object : <PrimaryObjectSpec>.<SubObjectPathSpec>.
MethodName : Value>[,Method Name: <Value> , …]
[,<SubObjectPathSpec>.MethodName :<Value>, …..]
```

The specifications given for <PrimaryObjectSpec>, <SubObjectPathSpec>, MethodName remain the same as described in the New Method syntax section in the topic Objects and Collection.

A single **Modify Object** Action cannot modify methods of multiple primary Objects, but can modify **multiple values of an Object**.

Modify Object is allowed to have Primary Object Specification only once i.e., for the first value. Further values permissible are optional in the Sub Object and Method Specification only.

From the second value onwards, the Sub Object specification is optional. If the Sub Object Specification is specified, the context is assumed to be the Primary Object specified for the first value. In absence of the Sub Object Specification, the previous value specification's Leaf Object is considered as the context.

Example 1

```
[Key: Alter My Object]

     Key : Ctrl + Z

     Action : Modify Object : (Ledger,"MyLedger").BillAllocations+

              [First, $Name="MyBill"].OpeningBalance : 100,+

              ..Address[Last].Address : "Bangalore"
```

The existing ledger **My Ledger** is being altered with new values for the **Opening Balance** for the existing bill and **Address**. The key **Alter My Object** can be attached to any Menu or Form.

Example 2

```
[Key: Alter My Object]

     Key : Ctrl + Z

     Action : Modify Object :(Ledger,"MyLedger").BillAllocations[1] +

              .OpeningBalance  :1000,Name: ”My New Bill”,..Address[First]+

              .Address :"Hongasandra Bangalore" , Opening Balance:5000
```

The existing ledger **My Ledger** is being altered with new values for the **Opening Balance** applicable on the existing bill, **Opening Balance** of the ledger and the first line of the **Address**. The key **Alter My Object** can be attached to any Menu or Form.

A button bearing the action Modify Object if associated at Menu Definition requires a primary object specification as Menu, which is not in context of any Data Object.

Example

```
[#Menu : Gateway of Tally]

   Add : Button : Alter My Object
```

The following points should be considered while associating a key with the action Modify Object:

- Since the Menu does not have any Info Objects in context, specifying Primary Object becomes mandatory.
- Since the Menu cannot work on scopes like Selected, Unselected, etc. the scopes specified are ignored.
- Any formula specified in the value and is evaluated assumes the Menu Object as the requestor.
- Even Method values pertaining to Company Objects can be modified.
- A button can be added in the Menu to specify the action Modify Object at the Menu level.

4.4 Action – Browse URL

The Action, Browse URL is used to provide a link to any web browser with an URL formula passed as a parameter.

Syntax

```
      Action: Browse URL: <URL Formula>
```

Example: Field acting as a hyperlink

```
[Key : Execute Hyperlink]
         Key       : Left Click
         Action    : Browse URL: "www.tallysolutions.com"

[Field: Hyperlink Company]
         Color     : Blue
         Border    : Thin Bottom
         Key       : Execute Hyperlink
         Set as    : "Tally Solutions Pvt. Ltd"
         Local     : Key : Execute Hyperlink :Action:Browse URL:+
                     http://www.tally.co.in
```

5. Actions — Create Collection, Display Collection and Alter Collection

5.1 Action — Create Collection

A Menu Item can be used to create Objects in a Collection with the action Create Collection. This action is generally used for creation of Masters such as Groups, Ledgers, Stock Items, Voucher Types, etc. Create Collection fetches a report through the defined Collection. A report displayed through this action, is done in Create mode.

Example

;; The following code demonstrates the usage of Create Collection Action

```
[#Menu: Gateway of Tally]
   Add      : Key Item: Ledger: L : Create Collection: Ledger
```

;; where a Ledger is a predefined Collection in DefTDL

One can also use the action Create in place of Create Collection, to create Objects in a collection. The only difference is that Create explicitly calls a Report and Create Collection requires a collection. Create Collection executes the same report through the defined Collection.

5.2 Action – Display Collection

A Menu Item or a Button can be used to display a popup of Object names in a Collection, which in turn, can trigger a Report. On choosing an Object from the popup, a report in display mode is triggered by the action, Display Collection. This action can be used for displaying the Masters or Reports pertaining to Groups, Ledgers, Stock Items, etc.

Example

;; The following code demonstrates the usage of Display Collection Action

```
[#Menu: Gateway of Tally]
   Add      : Key Item: Ledger :  L : Display Collection : Ledger
```

;; where Ledger is a predefined Collection in DefTDL

Though the Action name is Display Collection, Display is meant for the subsequent Report, which will be displayed on selection of an Object. Here, the Report is in display mode.

5.3 Action – Alter Collection

The Action, Alter Collection is similar to Display Collection, but it triggers the Report in Alter mode. This Action is generally used to alter the Masters such as Groups, Ledgers, Stock Items, Voucher Types, etc.

Example

;; The following code demonstrates the usage of Alter Collection Action

```
   [#Menu: Gateway of Tally]
      Add   : Key Item: Ledger: L : Alter Collection : Ledger
```

;; where Ledger is a predefined Collection in DefTDL

Though the Action is Alter Collection, Alter is meant for the subsequent Report which will be displayed on the selection of an Object.

Display Collection, Create Collection and Alter Collection routes the final report through a Collection. Let us understand, some critical Attributes require to achieve these actions.

5.4 Collection Attributes

The Collection attributes Trigger, Variable and Report, support the actions Create Collection, Display Collection and Alter Collection respectively.

```
[Collection: My Ledger]
   Type         : Ledger
   Trigger      : LedList Select
   Report       : Selected Ledger Display
   Variable     : Ledger Name
```

5.4.1 Trigger

The Collection attribute, Trigger is used to popup the Object names from a Collection. For example, a List of Items pop up when you choose the default Menu Item Stock Item.

Syntax

```
[Collection: <Collection Name>]
      Trigger: <Report Name>
```

The Report Name is the Interface used to display the Object names in a Collection.

5.4.2 Report

The Collection Attribute, Report displays a Report based on the Object selected. For example, Item Monthly Summary is a default Report being displayed when you choose a particular stock item.

Syntax

```
[Collection: <Collection Name>]
      Report: <Report Name>
```

The Report Name is the final report displayed, when an Object is selected from the Collection.

5.4.3 Variable

The Collection Attribute, Variable stores the name of the selected Object. This attribute is used with actions, Display Collection and Alter Collection.

Syntax

```
[Collection: <Collection Name>]
      Variable: <Variable Name>
```

The Variable Name is the variable which stores the Object name for the subsequent Report which is to be displayed.

Example

```
[Collection: Stock Items in Display Collection]
   Type         : Stock Item
   Trigger      : Stock Item Selection Interface
   Report       : Stock Item Final Report
   Variable     : Stock Item Name
```

6. Object Specific Actions

Some of the Object Specific Actions are discussed below:

6.1 Menu Actions – Menu Up, Menu Down, Menu Reject

Menu Actions - Menu Up, Menu Down, Menu Reject, etc. are acted upon on the Menu. These keys are associated to all the Menus (Default TDL codes as well as User Defined TDL codes) through the declaration **[System: Menu Keys]**

Example

```
[Key: Menu Up]
   Key      : Up
   Action   : Menu Up
[Key: Menu Down]
   Key      : Down
   Action   : Menu Down
[Key: Menu Reject]
   Key      : Esc
   Action   : Menu Reject

[System: Menu Keys]
   Key      : Menu Down, Menu Up, Menu Reject
```

The declaration **[System: Menu Keys]** declares a list of Keys that are commonly required for any Menu. Since all the common menu operations like Scroll Up, Scroll Down, Drill down, etc. are declared here; a new Menu added, does not require these keys to be associated since they are inherited from the above declaration.

6.2 Form Actions – Form Accept, Form Reject, Form End

Form Actions; Form Accept, Form Reject, Form End, etc. are acted upon Form. These keys are associated to all the Forms (Default TDL codes as well as User Defined TDL codes) through the declaration **[System: Form Keys]**

- Action Form Accept saves the current Form.
- Action Form Reject rejects the current Form i.e., the current form is quit without saving.

Example

```
[Key: Form Accept]
   Key       : Ctrl + A
   Action    : Form Accept
   Mode      : Edit

[Key: Form Display Reject]
   Key       : Esc
   Action    : Form Reject
   Mode      : Display

[Key: Form End]
   Key       : Ctrl + End
   Action    : Form End

[System: Form Keys]
   Key       : Form Accept, Form Display Reject, Form End
```

The declaration [System: Form Keys] declares a list of Keys that are commonly required for any Report. Since all the common Form operations like Save Form, Reject Form, Form End, etc. are declared here; a new Form added does not require these keys to be associated since they are inherited from the above declaration.

6.3 Part Actions – Part Home, Part End, Part Pg Up

Part Actions; Part Home, Part End, Part Pg Up, etc. are acted upon Part. These keys are associated with all the Forms (Default TDL codes as well as User Defined TDL codes) through the declaration **[System: Form Keys]**

- Action Part Home positions the cursor to the beginning of the current Part
- Action Part End positions the cursor to the end of the current Part
- Action Part PgUp is used to quickly scroll the page to view the previous page

Example

```
[Key: Part Display Home]
   Key       : Home
   Action    : Part Home
   Mode      : Display

[Key: Part Display End]
   Key       : End
   Action    : Part End
   Mode      : Display

[Key: Part Display PgUp]
   Key       : PgUp
   Action    : Part PgUp
   Mode      : Display

[System: Form Keys]
   Key       : Part Display Home, Part Display End, Part Display PgUp
```

The declaration **[System: Form Keys]** declares a list of Keys that are commonly required for any Part. Since all the common Part operations like Part Home, Part End, Part PgUp, etc. are declared here; a new Part added does not require these keys to be associated since they are inherited from the above declaration.

6.4 Line Actions – Explode, Display Object, Alter Object

Line Actions: Explode, Display Object, Alter Object, etc. are acted upon Line.

- Action Explode explodes a line further to display all the explode details specified in the Line Attribute Explode.
- Action Display Object is used to display the Object in context of the current line.
- Action Alter Object is used to alter the Object in context of the current line.

Example

```
[Key    : Line Explode]
   Key       : Shift + Enter
   Action    : Explode
[Key    : Line Object Display]
   Key       : Enter
   Action    : Display Object
   Mode      : Display
```

```
[Key: Line Object Alter]
   Key       : Ctrl + Enter
   Action    : Alter Object
   Mode      : Display
[System: Form Keys]
   Key       : Line Explode
   Key       : Line Object Display, Line Object Alter
```

The declaration **[System: Form Keys]** declares a list of Keys that are commonly required for any Line. Since all the common Line operations like Explode, Display Object, Alter Object, etc. are declared here; a new Line added does not require these keys to be associated since they are inherited from the above declaration.

6.5 Field Actions – Field Copy, Field Paste, Field Erase, Calculator

Field Actions: Field Copy, Field Paste, Field Erase, Calculator, etc. are acted on Fields.

- The Action Field Copy, copies the current field (Field where the cursor is positioned) contents in the OS clipboard which will be available later.
- The Action Field Paste, pastes the clipboard contents to the current Field.
- The Action Field Erase, is used to erase the contents of the current Field at a stretch without hitting the Backspace or Delete Key.
- The Action Calculator is used in case of Fields that require some computation, the result of which should be returned to the Field. Fields which take Amounts or Numbers as their value require this action.

Example

```
[Key: Field Copy]
   Key       : Ctrl + Alt + C
   Action    : Field Copy
[Key: Field Paste]
   Key       : Ctrl + Alt + V
   Action    : Field Paste
[Key: Field Erase]
   Key       : Esc
   Action    : Field Erase
   Mode      : Edit
[Key: Calculator]
   Key       : Alt + C
   Action    : Calculator
   Mode      : Edit
```

```
[Field: NumDecimals Field]
    Key     : Calculator
[System: Form Keys]
   Key      : Field Erase
   Key      : Field Copy, Field Paste
```

The declaration **[System: Form Keys]** declares a list of Keys that are commonly required for any Field. Since all the common Field operations like Field Copy, Field Paste, Field Erase, etc. are declared here; a new Field added does not require these keys to be associated since they are inherited from the above declaration. The Action Calculator is not required for all the Fields hence it has not been declared in Form Keys usage List. The Action Calculator has been associated to Fields where it is required. In the above example, NumDecimals Field is a numeric field which may require calculations. Therefore, the Calculator Key associating the Action Calculator is attached to the Field.

Learning Outcome

- Actions are activators of a specific task with a definite result. An Action always originates from a User Interface (UI) Objects Menu, Form, Line or Field.
- Global Actions and Object Specific Actions are the different types of actions used in TDL.
- Actions can be associated at various levels:
  - Action Association at Menu Definition
  - Action Association at Button/Key Definition
  - Action Association at Field Definition
- An Action is always executed with respect to two contexts:
  - Originator
  - Executer
- Some of the frequently used Global Actions are:
  - Menu
  - Modify object
  - Browse URL
  - Create and Alter
- Some of the Object Specific Actions are:
  - Menu Actions - Menu Up, Menu Down, Menu Reject
  - Form Actions – Form Accept, Form Reject, Form End
  - Part Actions – Part Home, Part End, Part Pg Up
  - Line Actions – Explode, Display Object, Alter Object
  - Field Actions – Field Copy, Field Paste, Field Erase, Calculator

User Defined Fields

Introduction

In Tally.ERP 9 the structure of an object, the data type and storages that are required in order to store the data are all pre-defined by the platform. All the data is stored in the Tally data base. By default, the data is always stored in the pre-defined storages only.

There may be instances when additional information needs to be stored in the existing objects. This need has given rise to the concept of User Defined Fields (UDF). A UDF is used to store the additional information which is part of the Tally database. In other words, UDFs store additional information in the existing objects.

1. What is UDF?

User Defined Fields have a storage component defined by the user. User Defined Fields are stored in the context of current object. It can be of any Tally data type such as String, Amount, Quantity, Rate, Number, Date, Rate of Exchange and Logical.

Defining UDFs does not serve the purpose unless it is associated with one or more internal object. When a UDF is created and used in an already existing report, the data is stored in the context of object, it is always attached to the object to which this report is associated i.e. the object in context.

1.1 Creating a UDF

Syntax

```
      [System: UDF]
            <Name of UDF>: <Data Type>: <Index Number>
```

UDFs should be defined under the section **[System: UDF]**.

<Name of UDF> Identifies the UDF and ideally it should describe the purpose for which it is created.

<Data Type> This deals with any of the Tally data type and Aggregate.

<Index Number> It can be any number between 1 and 65536.

The number falling between 1 to 9999 and 20001 to 65536 are opened for customisation and 10000 to 20000 is allotted for Common development in TSPL. The user can create 65536 UDFs of each data type.

The index numbers 1 to 29 is already used for default TDL.

Example

```
[System: UDF]

   MyUDF 1 : String : 20003

   MyUDF 2 : Date   : 20003
```

The advantage of a UDF in Tally is that they automatically get attached to the current object. There is no specific declaration which is required for the object association when the UDF is defined with the system definition.

1.2 To store the User Input in the UDF

The attribute Storage of the field definition is used to store the value entered in a field. The value is stored in the context of current object. The syntax of a Storage attribute is as follows:

Syntax

```
Storage : <Default Storage / Name of UDF>
```

<Name of UDF> It identifies the UDF and ideally describes the purpose for which it is created.

Example

```
[Field: NewField]

   Use         : NameField

   Storage     : MyUDF
```

1.3 To retrieve the value of UDF from an Object

In the context of the current object, the value of a UDF can be accessed by prefixing the $ to the UDF name.

Syntax

```
$<Name of UDF>
```

Example:

```
[Field: NewField]
   Use       : NameField
   Set As    : $MyUDF
```

2. Classification of UDF's

UDFs are classified into two types, which are as follows:

- Simple UDF
- Complex/Compound/Aggregate UDF

2.1 Simple UDF

A Simple UDF can store one or more values of single data type only. When a UDF is used for storage, it stores the value in the context of object associated at Line or Report Level, by default . Only one value is stored in this case.

2.1.1 UDF to store a single value

The following example code snippet demonstrates the usage of UDF to store a single value.

Example

```
[Report: CompanyVehicles]
   Object       : Company
             .
             .
             .
[Field: CVeh]
   Use          : Name Field
   Storage      : Vehicle
   Unique       : Yes

[System : UDF]
   Vehicle      : String : 700
```

The object is associated at the Report Level. The value stored in a UDF is in the context of a Company Object in this case. The UDF vehicle stores a single string value .

2.1.2 UDF to store multiple values

When multiple values of the same data type are to be stored, then the Repeat attribute of Part is used. The field of the line uses the same UDF name in the Storage attribute. The syntax of the Repeat attribute of Part in this case will be as follows:

Syntax

```
Repeat : <Line name > : < Name of UDF >
```

<Line Name > Name of the line to be repeated.

<Name of UDF> It identifies the name of the UDF to store multiple values

The example explained in the section “UDF to store single value” can be modified to store multiple values of string type.

Example

```
[Part: CompVeh]
   Line        : CompVeh
   Repeat      : CompVeh  : Vehicle
   Break On    : $$IsEmpty:$Vehicle
   Scroll      : Vertical
```

In this scenario, multiple values of type string can be stored under the object **Company**.

2.1.3 Creating collection of Values Stored in UDF

Multiple values stored in a UDF can be displayed as Table in a field. A collection has to be defined as shown :

Syntax

```
[Collection : <Collection Name>]
      Type        : <UDF Name> : <Object Name>
      Format      : $<UDF Name>, 20
```

Example

```
[Collection: CMP Vehicles]
   Type        : Vehicle : Company
   Childof     : ##SVCurrentCompany
   Format      : $Vehicle, 20
   Title       : "Company Vehicles"
```

The above code snippet creates a collection of values stored in the UDF of current Company object. Once the collection is defined it can be used in the Table attribute of field definition. So when the cursor is in the defined field, the values stored in the UDF will be displayed as a popup table.

Consider the following example:

```
[Field: EI Vehicles Det]
   Use         : Short Name Field
   Table       : CMP Vehicles, Not Applicable
   Show Table  : Always
```

A popup table is displayed when the cursor is placed in the field **'EI Vehicles Det'**. The Table contains values stored in the UDF which are Not Applicable as a list.

2.2 Aggregate UDF

A Simple UDF can only store values of a single data type so when multiple values of different data types are required to store as one entity, then in such cases an Aggregate UDF can be used.

Aggregate UDFs are very useful for storing multiple values and repeated values. An aggregate UDF is a combination of different types of UDFs. Aggregate UDFs can be used to store user data in a tabular format attached to any internal object and can be used as a collection of UDFs.

In other words, an Aggregate UDF comprises of a set of fields repeating itself more than once. The output can be stated in the form of a record consisting of fields of different types and sizes. When a line is repeated over an Aggregate UDF, it associates all its storage components (same or different data types) as a single unit.

2.2.1 Creating an Aggregate UDF

To create an Aggregate UDF the, the data aggregate is used while defining the UDF. The components are defined as simple UDFs.

Syntax

```
[System: UDF]
      <Name of UDF>: Aggregate : <Index Number>
```

Example

A Company wants to create and store multiple details of company vehicles.The details required are: Vehicle Number, Brand, Year of Mfg., Purchase Cost, Type of Vehicle, Currently in Service, Sold On date and Sold for Amount.

```
[System : UDF]
   Company Vehicles         : Aggregate : 1000
   VVehicle Number          : String    : 1000
   VBrand                   : String    : 1001
   VYear of Mfg             : Number    : 1000
   VPurchase Cost           : Amount    : 1000
   VType of Vehicle         : String    : 1002
   VCurrently in Service    : Logical   : 1000
   VSold On date            : Date      : 1000
   VSold for                : Amount    : 1001
```

To store the required details simple UDFs are defined and to store then them as one entity, one UDF of the type Aggregate is defined as shown in the example.

2.2.2 Using an Aggregate UDF

An Aggregate UDF defined does not associate each component field with it. The association will takes place only when you repeat a Line over an Aggregate UDF and within that Line you have fields which stores the value into the component UDFs.

Syntax

```
      Repeat : <Line name > : < Name of Aggregate UDF >
```

<Name of Aggregate UDF> This is the name of UDF defined with Aggregate data type.

Example

```
   [Part : Comp Vehicle]
      Line        : Comp VehLn
      Repeat      : Comp VehLn :  Company Vehicles
      BreakOn     : $$IsEmpty:$VBrand
                .
                .
                .

   [Field    : CMP VBrand]
      Use         : Short Name Field
      Storage     : VBrand
```

The Line is repeated over the Aggregate UDF and the Simple UDFs are entered in the fields.

2.2.3 Using Aggregate UDF in a Sub-Form

A Subform is an attribute that is used with a Field definition. It relates to a report (not Form) and can be invoked by a field. This attribute is useful to activate a report within a report, perform the necessary action and return to the report used to invoke the Subform. There is no limit on the number of Subforms that can be used at the field level.

Syntax

```
      [Field: Field Name]
            Sub Form : <Report Name> : <Condition>
```

Where:

<Report Name> This is the name of the Report to be displayed.

<Condition> This could be any expression, which evaluates to a logical value. The report will be displayed only when the condition is true.

A Sub Form is not associated to the Object at the Report level. An Object associated to the Field in which the Sub Form is defined, gets associated to the Sub Form. A Sub Form will inherit the info object from the Field which appears as a pop-up.

The Bill-wise Details is an example of a Subform attribute. This screen is displayed as soon as an amount is entered for a ledger whose Bill-wise Details feature has been activated.

Example

The following code snippet uses a Subform to enter the details of bills when the Bill Collection ledger is selected while entering a Voucher. The values entered in the Subform are stored in an Aggregate UDF. This UDF is attached to the object to which the field displaying the Subform is associated. Here, it is the Object of a Ledger Entries Collection.

The following code is used to associate a Subform to the default Field in a voucher.

```
[#Field: ACLSLed]
   Sub Form : BillDetail : ##SVVoucherType = "Receipt"  +
              and $LedgerName = "Bill Collection"
```

The **Name** Report for the Subform uses an Aggregate UDF to store the data. A Line is repeated over the Aggregate UDF at the Part level.

```
[Part : BillDetails]
   Scroll         : vertical
   Line           : BillDetailsH, BillDetailsD
   Repeat         : BillDetailsD : BAggre
   Break After    : $$Line=2
```

The Attibute **Storage** is used for all the fields.

```
   [Field : CustName1]
      Use         : Name Field
      Storage     : CustName
```

The UDF is defined as follows:

```
[System : UDF]
   CustName : String    : 1000
   BillNo   : String    : 1001
   BillAmt  : Amount    : 1001
   FPrint1  : String    : 1002
   BAggre   : Aggregate : 1000
```

Learning Outcome

- User Defined Fields are stored in the context of the current Object. These Fields can be of any Tally data type.
- UDFs should be defined under the section **[System: UDF]**.
- The attribute Storage in a Field definition is used to store the value entered in a Field. The value is stored in the context of a current Object.
- A Simple UDF can store one or more values of a single data type only.
- Aggregate UDFs are very useful for storing multiple values and repeated values.

Reports, Printing and Validation Controls

Introduction

In the previous lesson, the significance and usage of the User Defined Fields was explained. The classification and creation of UDF's was also discussed. This lesson is dedicated to Report creation and printing. The types of reports and the different ways of printing them will be explained in detail.

1. Reports

In Tally.ERP 9 the financial statements are generated as Reports based on the vouchers entered till date. The Balance Sheet, Profit & Loss A/c, Trial Balance etc are the some of the Reports which Tally.ERP 9 has by default.

Normally a business requires Reports in any of the following formats:

- Tabular Report: A Report with fixed number columns which can be configured
- Hierarchical Report : A Report designed in successive levels or layers
- Columnar Report : A Report with multiple columns

Tally.ERP 9 caters to generating all the types of Reports mentioned above.

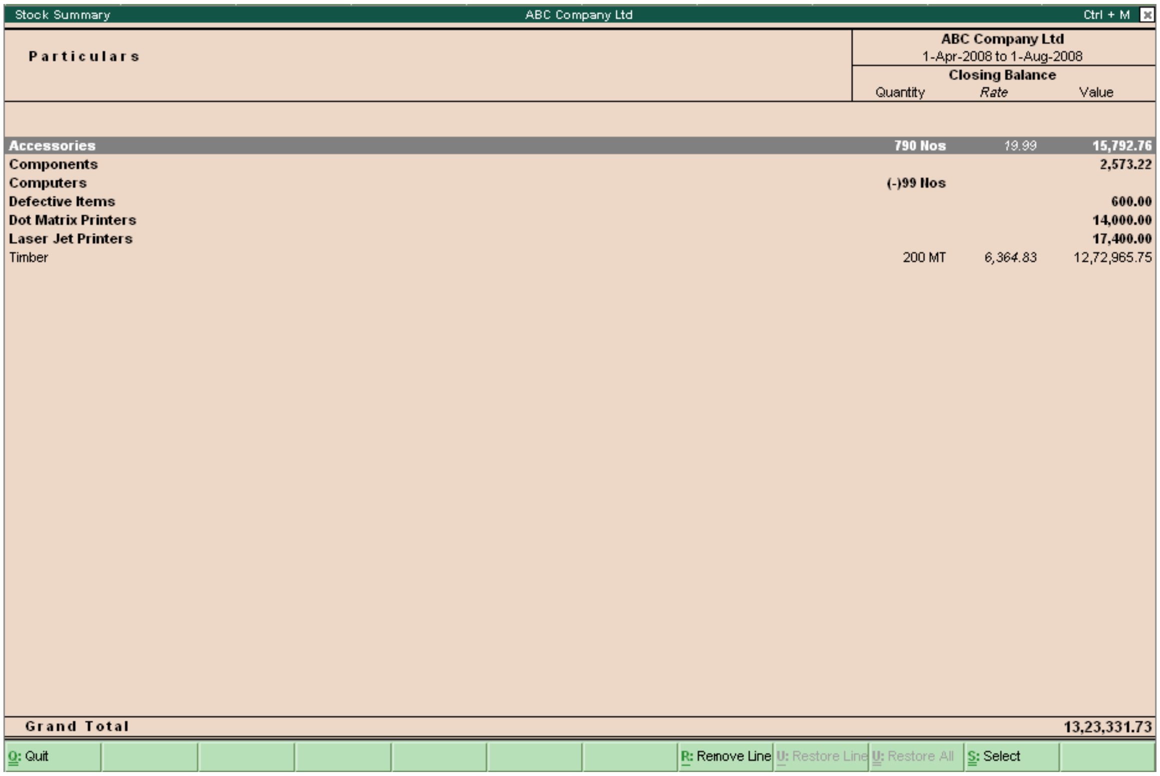
1.1 Tabular Reports

A Tabular Report has the simplest format of all the Report formats. A typical Tabular Report will have following components:

- Report Title : It contains the Name of the Report, the Title for each column, the Day/ Period for which a Report is generated, etc
- Report Details : It contains the actual information
- Report Total: It contains the Total of the respective columns

In a typical Tabular Report, the number of columns is fixed and is interactive i.e. an end user can change the appearance of the Report. The Day Book, Stock Summary, Trial Balance, Group Summary are the some of the default Tabular Reports in Tally.ERP 9.

The Tabular Report, Stock Summary is shown in Figure 9.1



Particulars	ABC Company Ltd 1-Apr-2008 to 1-Aug-2008		
	Closing Balance		
	Quantity	*Rate*	Value
Accessories	**790 Nos**	*19.99*	**15,792.76**
Components			**2,573.22**
Computers	**(-)99 Nos**		
Defective Items			**600.00**
Dot Matrix Printers			**14,000.00**
Laser Jet Printers			**17,400.00**
Timber	200 MT	*6,364.83*	12,72,965.75
Grand Total			**13,23,331.73**

Figure 1.1 Stock Summary

1.1.1 Designing a Tabular Report

A typical Tabular Report will have a Title Line, Details Line and an optional Total Line. The Details Line will be repeated over the objects of a Collection.

A Tabular Report can be made Interactive by adding the following features.

- Adding Buttons to change the period, to change the contents of the Report, etc (As discussed in the lesson 5: Variables, Buttons, Keys)
- Adding explosions to the lines

1.1.2 Displaying the Exploded Part

Tally.ERP 9 allows the user to display additional information about the current line object when the key combination SHIFT + Enter is pressed. This functionality is referred to as an explosion in Tally.ERP 9. The attribute Explode and Indent of Line definition, and the $$KeyExplode function is used.

The attribute Explode

The attribute ‘Explode’ refers to an attribute of the line, which is used to take the current data from the Line Object. A Part is displayed after the process of explosion is complete.

Syntax

```
Explode : <Part Name> : <Logical Condition>
```

<Part Name> This is the name of the Part which displays the additional information about the Line object.

<Condition> If the Condition is True, then it will result in an explosion.

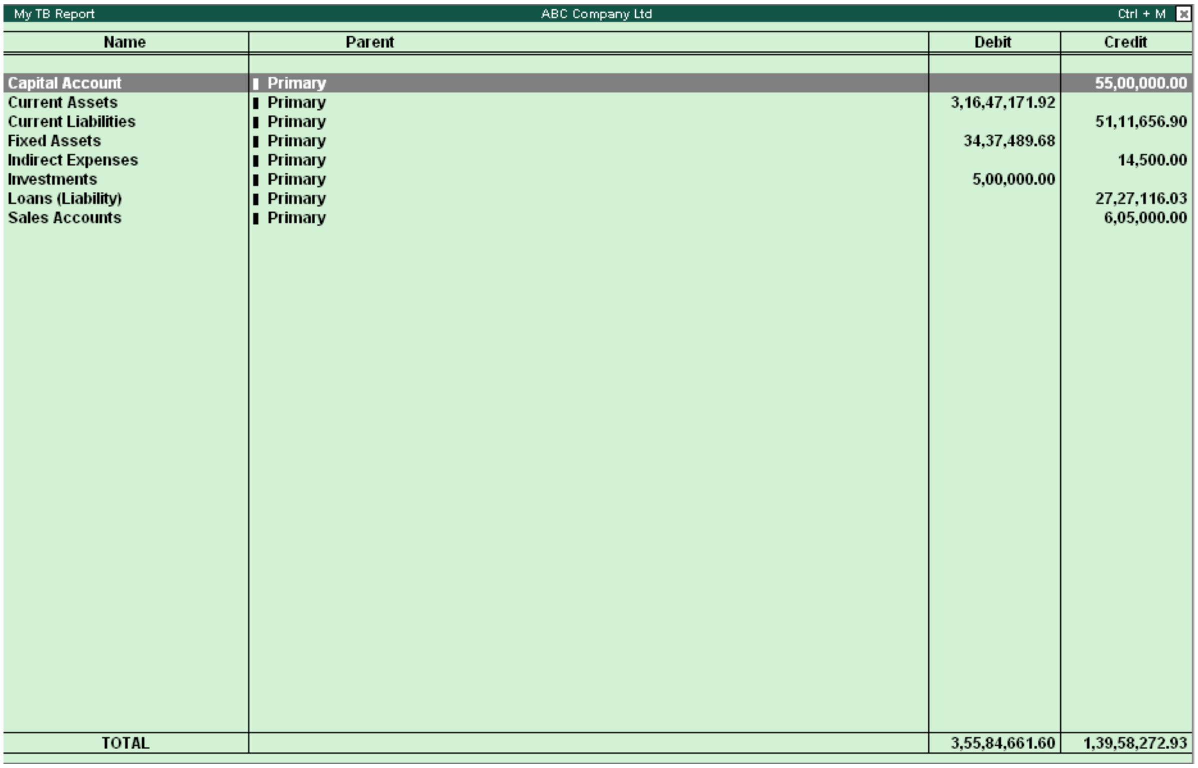
The Function of $$KeyExplode

$$KeyExplode function gives the current status of the keys Shift and Enter. This is used as a condition to check if the user has pressed the Shift+Enter Keys.

Example 1: Simple Tabular Report

Let us consider writing a simple Trial Balance.

```
[Part: My TB Part]
   Lines    : My TB Title, My TB Details
   Repeat   : My TB Details: My TB Groups
   Scroll   : Vertical
```



My TB Report ABC Company Ltd Ctrl + M

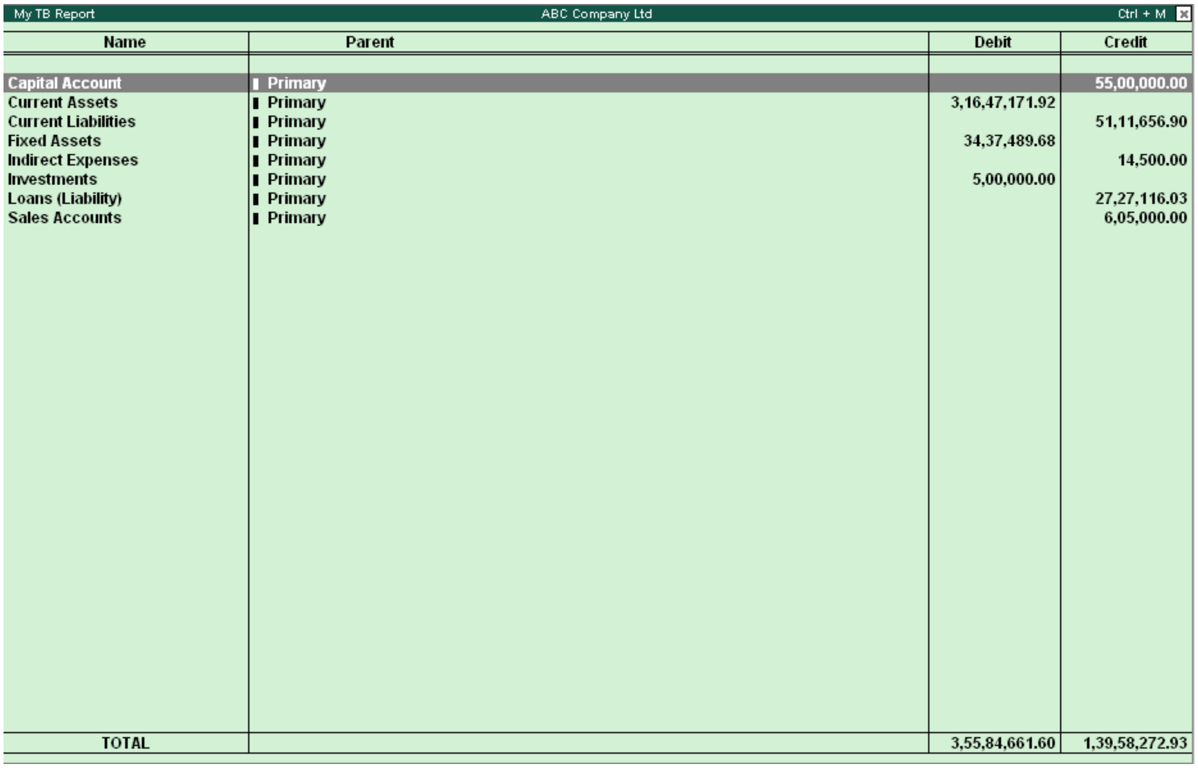
Name	Parent	Debit	Credit
Capital Account	Primary		55,00,000.00
Current Assets	Primary	3,16,47,171.92	
Current Liabilities	Primary		51,11,656.90
Fixed Assets	Primary	34,37,489.68	
Indirect Expenses	Primary		14,500.00
Investments	Primary	5,00,000.00	
Loans (Liability)	Primary		27,27,116.03
Sales Accounts	Primary		6,05,000.00
TOTAL		3,55,84,661.60	1,39,58,272.93

Figure 1.2 Simple Trial Balance Report

Example 2 : A Simple Interactive Tabular Report

A report showing all the Primary groups can be created and exploded by pressing Shift + Enter to view the sub groups. The ledgers can subsequently be viewed on the same screen with an indent for each level.

The report is as shown in Figure 9.3



My TB Report ABC Company Ltd Ctrl + M

Name	Parent	Debit	Credit
Capital Account	▮ Primary		55,00,000.00
Current Assets	▮ Primary	3,16,47,171.92	
Current Liabilities	▮ Primary		51,11,656.90
Fixed Assets	▮ Primary	34,37,489.68	
Indirect Expenses	▮ Primary		14,500.00
Investments	▮ Primary	5,00,000.00	
Loans (Liability)	▮ Primary		27,27,116.03
Sales Accounts	▮ Primary		6,05,000.00
TOTAL		3,55,84,661.60	1,39,58,272.93

Figure 1.3 Simple Interactive Tabular Report

The following code snippet displays the exploded part :

```
[Line: My TB Details]
   Fields      : My TB Name Field, My TB ParName Field
   Right Fields: My TB Dr Amt Field, My TB Cr Amt Field
   Explode     : My TB Group Explosion : $$IsGroup and $$KeyExplode

[Field: My TB Name Field]
   Use         : Name Field
   Set as      : $Name
   Variable    : MyGroupName1
```

The code for the exploded part is as shown below:

```
[Part: My TB Group Explosion]
   Lines        : My TB Details Explosion
   Repeat       : My TB Details Explosion   : My TB GroupsLedgers
   Scroll       : Vertical

[Line: My TB Details Explosion]
   Fields       : My TB Name Field, My TB ParName Field
   Right Fields: My TB Dr Amt Field, My TB Cr Amt Field
   Explode      : My TB Group Explosion : $$IsGroup and $$KeyExplode
   Indent       : 2  * $$ExplodeLevel
   Local        : Field  : Default  : Delete: Border
```

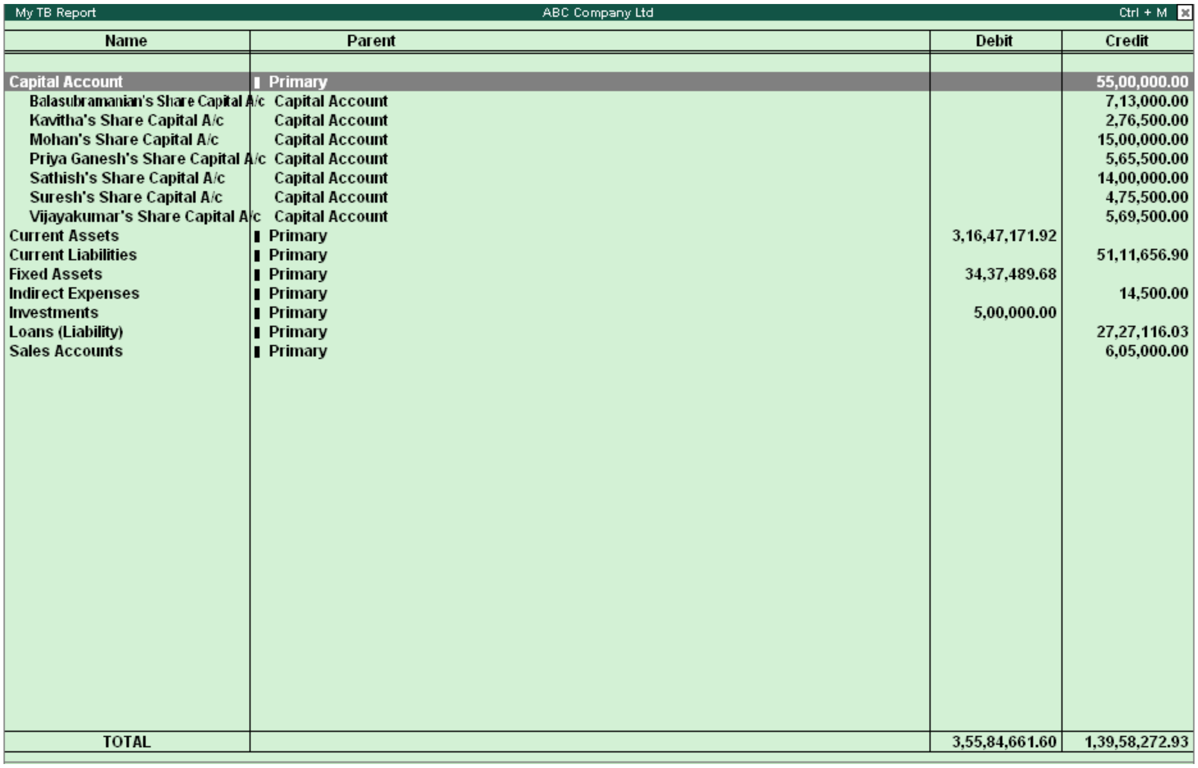
In the code snippet, the Collection **My TB GroupLedgers** is a union of collections of the Type Group and Ledgers respectively.

```
[Collection: My TB GroupsLedgers]
   Collection   : My TB Groups, My TB Ledgers
```

The variable **MygroupName1** is used in the attribute Child Of under the collections My TB Groups and My TB Ledgers.

```
[Collection: My TB Groups]
   Type         : Group
   Child Of     : #MyGroupName1

[Collection: My TB Ledgers]
   Type         : Ledger
   Child Of     : #MyGroupName1
```

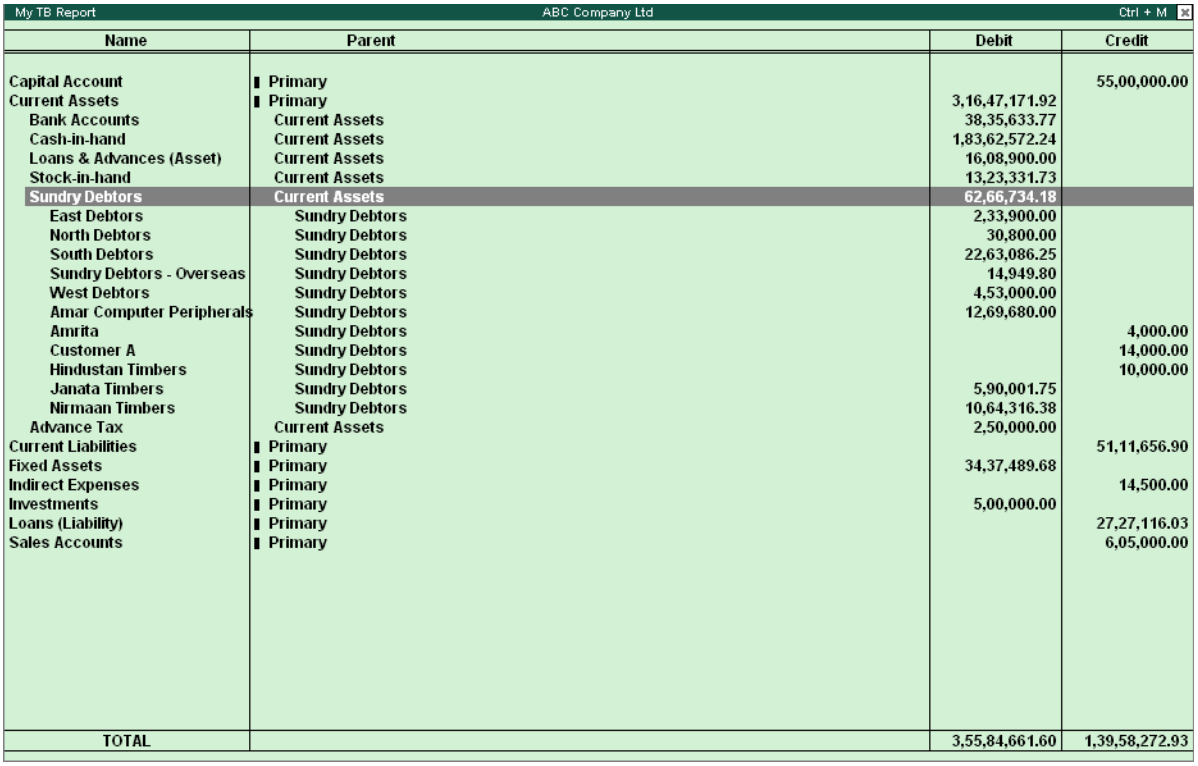
When the user presses the Shift + Enter keys, then the exploded part shows the Sub-groups under the group in the current line as shown in Figure 9.4.



Name	Parent	Debit	Credit
Capital Account	▮ Primary		55,00,000.00
Balasubramanian's Share Capital A/c	Capital Account		7,13,000.00
Kavitha's Share Capital A/c	Capital Account		2,76,500.00
Mohan's Share Capital A/c	Capital Account		15,00,000.00
Priya Ganesh's Share Capital A/c	Capital Account		5,65,500.00
Sathish's Share Capital A/c	Capital Account		14,00,000.00
Suresh's Share Capital A/c	Capital Account		4,75,500.00
Vijayakumar's Share Capital A/c	Capital Account		5,69,500.00
Current Assets	▮ Primary	3,16,47,171.92	
Current Liabilities	▮ Primary		51,11,656.90
Fixed Assets	▮ Primary	34,37,489.68	
Indirect Expenses	▮ Primary		14,500.00
Investments	▮ Primary	5,00,000.00	
Loans (Liability)	▮ Primary		27,27,116.03
Sales Accounts	▮ Primary		6,05,000.00
TOTAL		3,55,84,661.60	1,39,58,272.93

Figure 1.4 Interactive Tabular Report

When the keys Shift + Enter are pressed by the user, one more exploded part shows the Ledgers under the current Sub-group as shown in Figure 9.5.



My TB Report ABC Company Ltd Ctrl + M

Name	Parent	Debit	Credit
Capital Account	▮ Primary		55,00,000.00
Current Assets	▮ Primary	3,16,47,171.92	
Bank Accounts	Current Assets	38,35,633.77	
Cash-in-hand	Current Assets	1,83,62,572.24	
Loans & Advances (Asset)	Current Assets	16,08,900.00	
Stock-in-hand	Current Assets	13,23,331.73	
Sundry Debtors	Current Assets	62,66,734.18	
East Debtors	Sundry Debtors	2,33,900.00	
North Debtors	Sundry Debtors	30,800.00	
South Debtors	Sundry Debtors	22,63,086.25	
Sundry Debtors - Overseas	Sundry Debtors	14,949.80	
West Debtors	Sundry Debtors	4,53,000.00	
Amar Computer Peripherals	Sundry Debtors	12,69,680.00	
Amrita	Sundry Debtors		4,000.00
Customer A	Sundry Debtors		14,000.00
Hindustan Timbers	Sundry Debtors		10,000.00
Janata Timbers	Sundry Debtors	5,90,001.75	
Nirmaan Timbers	Sundry Debtors	10,64,316.38	
Advance Tax	Current Assets	2,50,000.00	
Current Liabilities	▮ Primary		51,11,656.90
Fixed Assets	▮ Primary	34,37,489.68	
Indirect Expenses	▮ Primary		14,500.00
Investments	▮ Primary	5,00,000.00	
Loans (Liability)	▮ Primary		27,27,116.03
Sales Accounts	▮ Primary		6,05,000.00
TOTAL		3,55,84,661.60	1,39,58,272.93

Figure 1.5 Interactive Tabular Reports - Sub Groups

1.2 Hierarchical Report (Drill down Report)

A Tally application provides a simple way of navigating from one report to another which is commonly referred to as a drill down. A Drill Down facility moves from one report to the other to give a detailed view based on the selection in the current report. A user can return to the first Report from the detailed view. A typical drill down in Tally.ERP 9 starts from the Report and reaches the Voucher Alteration screen.

1.2.1 Designing Hierarchical Reports

A Hierarchical Report can be designed by incorporating the following changes to a Tabular Report.

- Variable attribute of Report definition
- Child Of attribute of Collection definition
- Display and Variable attributes of Field definitions
- Variable Definition

Example

The following code snippet demonstrates the Drill down action, which is based on the Group Name displayed in the field. The Drill down action is achieved by specifying the two attributes Variable and Display at the field level.

```
[Field: MyTB Name]

   Width    : 120 mms

   Set as   : $Name

   Variable: GroupVar

   Display : My Trial Balance : $$IsGroup
```

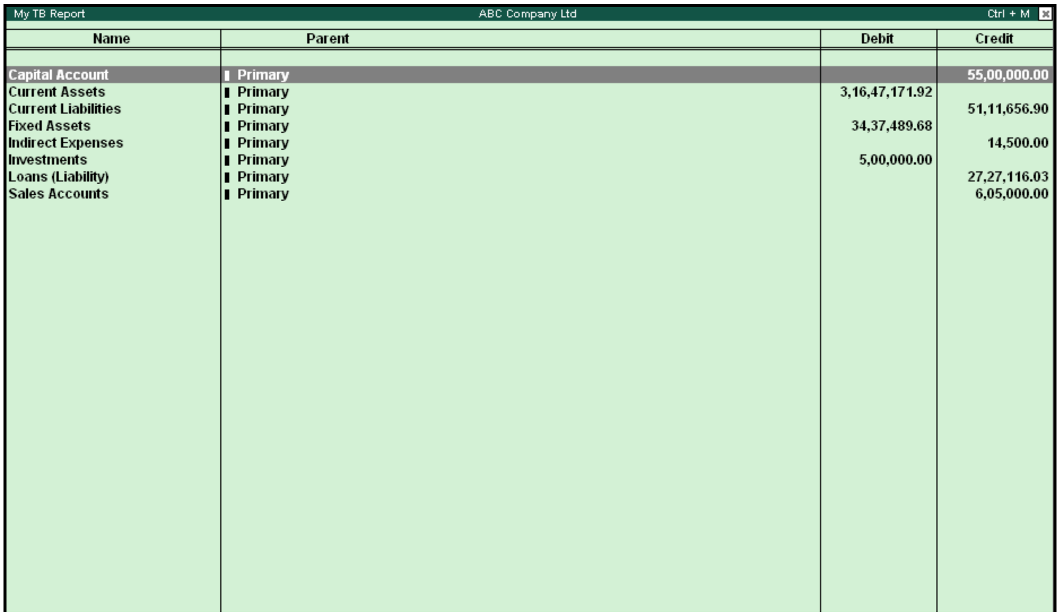
A Variable is defined as a Volatile and is associated at Report. The attribute Variable of Report definition is used to associate the Variable with the report.

```
[Variable: Group Var]

   Type        : String

   Default     : ""

   Volatile    : Yes


[Report: My Trial Balance]

   Form        : My Trial Balance

   Variable    : GroupVar
```

The same Variable is used in the Childof attribute of the Collection definition. When a line is repeated over this collection in the report when the user presses the Enter key the Report being displayed will have the objects whose Parent Name is stored in the variable.

```
[Collection: My Collection]

   Type        : Group

   Childof     : ## GroupVar
```

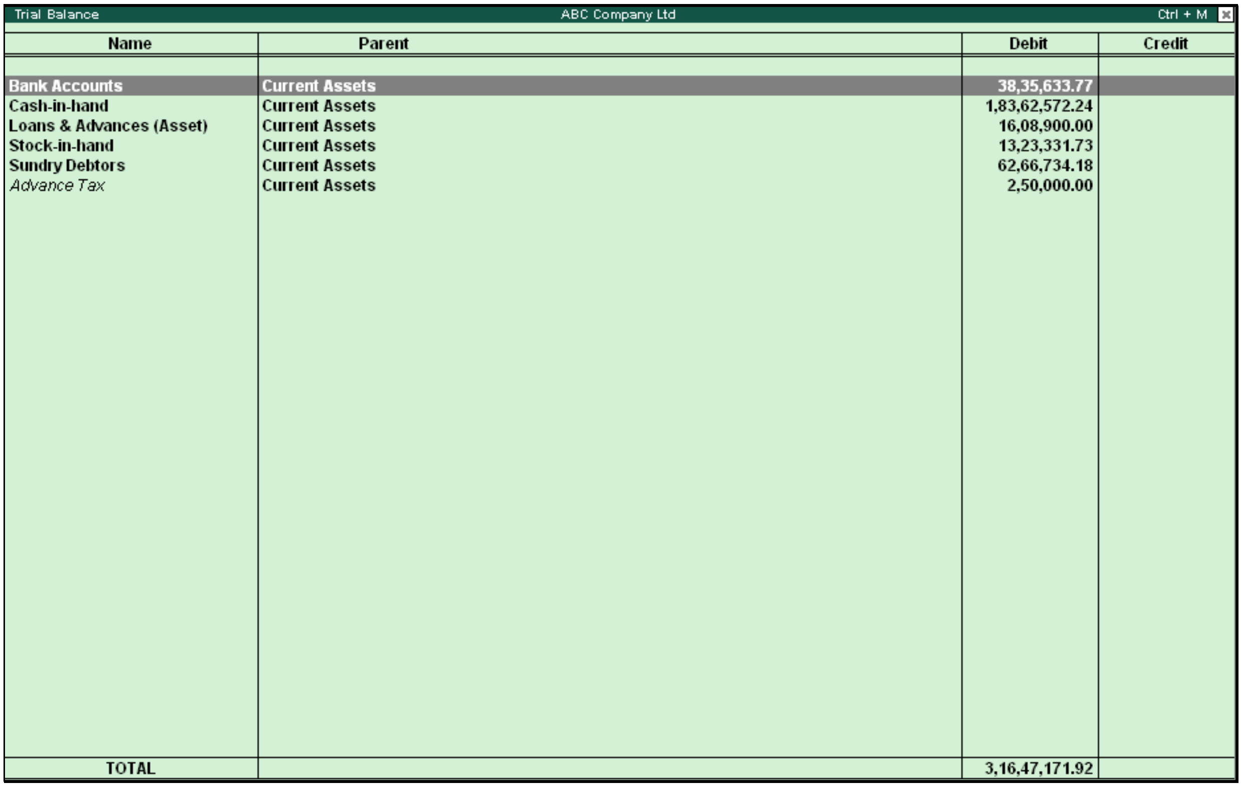
The following screen is displayed when the user selects the option from the Menu:



Name	Parent	Debit	Credit
Capital Account	▮ Primary		55,00,000.00
Current Assets	▮ Primary	3,16,47,171.92	
Current Liabilities	▮ Primary		51,11,656.90
Fixed Assets	▮ Primary	34,37,489.68	
Indirect Expenses	▮ Primary		14,500.00
Investments	▮ Primary	5,00,000.00	
Loans (Liability)	▮ Primary		27,27,116.03
Sales Accounts	▮ Primary		6,05,000.00

Figure 1.6 Trial Balance Report

When the key **Enter** is pressed by the user, the next screen displays the Sub Groups of the current Group as shown in Figure 9.7.



Name	Parent	Debit	Credit
Bank Accounts	Current Assets	38,35,633.77	
Cash-in-hand	Current Assets	1,83,62,572.24	
Loans & Advances (Asset)	Current Assets	16,08,900.00	
Stock-in-hand	Current Assets	13,23,331.73	
Sundry Debtors	Current Assets	62,66,734.18	
Advance Tax	Current Assets	2,50,000.00	
TOTAL		3,16,47,171.92	

Figure 1.7 Trial Balance - Sub group

1.3 Column Based Reports

The reports in which the number of columns added or deleted as per the user inputs are referred to as column based reports. There are four types of column based reports in Tally, namely Multi-Column Reports, Auto-Column Reports and Automatic Auto-Column Reports, Columnar Report. All these types are explained with examples in this section.

1.3.1 Multi-Column Reports

A Multi column Report is a report in which a column is repeated based on the criteria specified by user. Trial Balance, Balance Sheet, Stock Summary etc are the some of the default Reports in Tally.ERP 9 which has a Multi column feature. Normally this feature is used to compare the values across different periods.

1.3.2 Designing a Multi Column Report

In a Tabular Report Lines are repeated over a collection. But in a multi column Report, columns are repeated in addition to the repetition of the Lines over a Collection. Based on the user input columns are repeated. The column Report is used to capture the user inputs like Period, Company Name, Stock Valuation etc on which column is generated.

Following attributes are used at different components of a Report to incorporate the multi column feature.

The attribute Column Report

In TDL the attribute Column Report of the Report definition, facilitates the creation of multicolumn reports.

Syntax

```
ColumnReport: Report Name
```

The *Report Name* specified with this attribute is used to obtain the user input from the options displayed.

The attribute Repeat

The attribute Column Report is associated with a variable. The variable is specified in the Repeat attribute of Report definition. Both attributes must be specified in the Report definition to create a multi-column report.

Syntax

```
Repeat: Variable
```

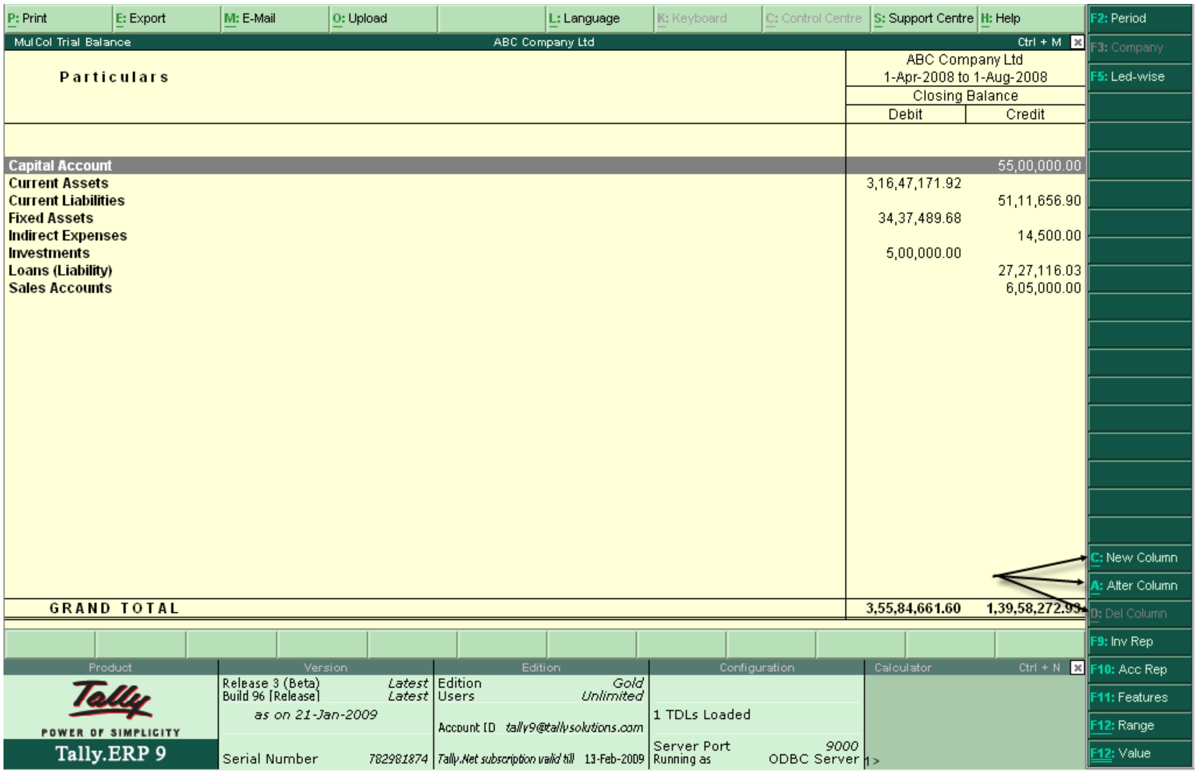
These two attributes automatically generates and displays three buttons on the Button Bar, namely “New Column”, “Alter Column” and “Delete Column” for further user interaction.

Example: Incorporating Multi Column Feature to Trial Balance report

Step 1 : Using Column Report & Repeat attribute at the Report

By using the Column Report & Repeat attribute at the Report, “New Column”, “Alter Column” and “Delete Column” buttons will be automatically added to ‘MulCol TrialBalance’ Report.

```
[Report: MulCol Trial Balance]
   ColumnReport: MyMultiColumns
   Repeat      : SVCurrentCompany, SVFromDate, SVToDate
```



Particulars	ABC Company Ltd 1-Apr-2008 to 1-Aug-2008 Closing Balance	
	Debit	Credit
Capital Account		55,00,000.00
Current Assets	3,16,47,171.92	
Current Liabilities		51,11,656.90
Fixed Assets	34,37,489.68	
Indirect Expenses		14,500.00
Investments	5,00,000.00	
Loans (Liability)		27,27,116.03
Sales Accounts		6,05,000.00
GRAND TOTAL	3,55,84,661.60	1,39,58,272.93

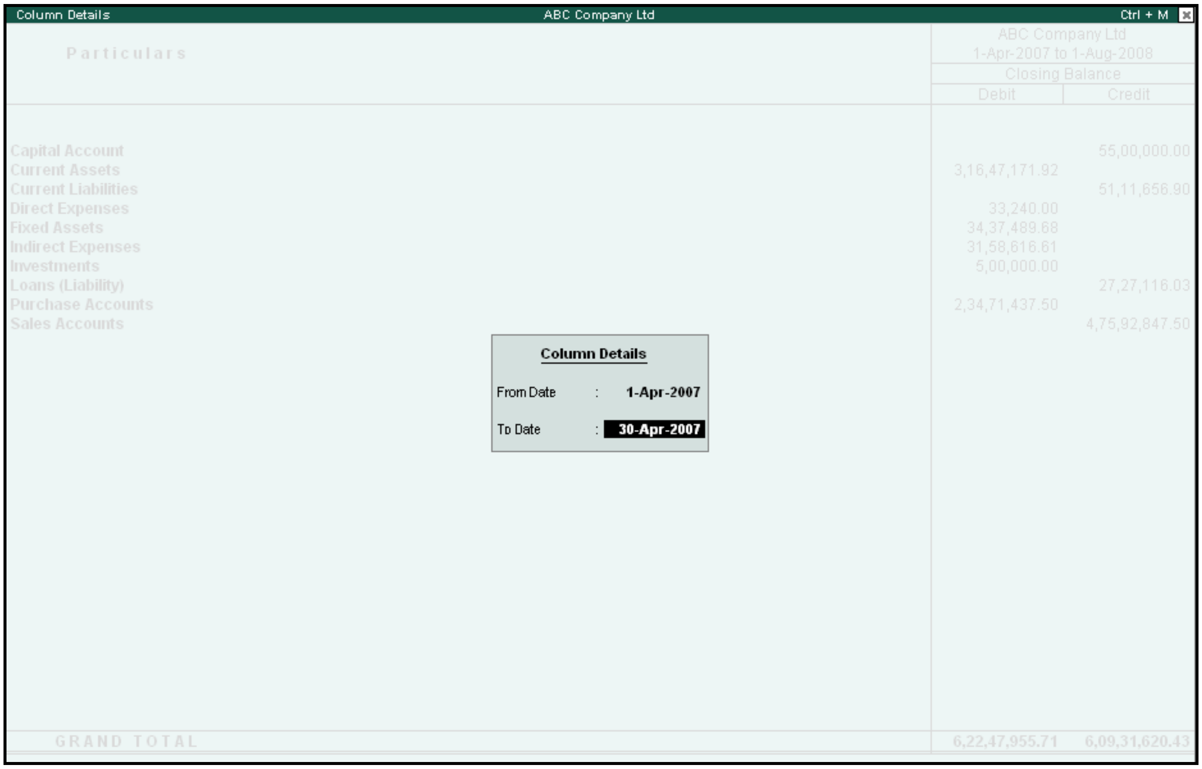
Release 3 (Beta) Latest
Build 96 [Release] Latest
as on 21-Jan-2009
Serial Number 782981874
Edition Gold
Users Unlimited
Account ID tally9@tallysolutions.com
Tally.Net subscription valid till 13-Feb-2009
1 TDLs Loaded
Server Port 9000
Running as ODBC Server

Figure 1.8 Multi Column Report

Step 2: Modifying the System Variables in a multi column Report

By clicking new column button, MyMultiColumns Report is displayed. In this Report, the user inputs are captured which will be reflected in the System Variables.

```
[Field: My MultiFromDate]
   Use          : Uni Date Field
   Modifies     : SVFromDate

[Field: My MultiToDate]
   Use          : Uni Date Field
   Modifies     : SVToDate

[Field: My MultiCompany]
   Use          : Name Field
   Modifies     : SVCurrentCompany
   Table        : Company
```



Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008 Closing Balance	
	Debit	Credit
Capital Account		55,00,000.00
Current Assets	3,16,47,171.92	
Current Liabilities		51,11,656.90
Direct Expenses	33,240.00	
Fixed Assets	34,37,489.68	
Indirect Expenses	31,58,616.61	
Investments	5,00,000.00	
Loans (Liability)		27,27,116.03
Purchase Accounts	2,34,71,437.50	
Sales Accounts		4,75,92,847.50
GRAND TOTAL	6,22,47,955.71	6,09,31,620.43

Column Details

From Date : 1-Apr-2007

To Date : 30-Apr-2007

Figure 1.9 Column Details for Multi Column Report

Step 3: Repeating Columns over a Variable and Lines over Objects of a Collection

To repeat columns over a Variable which is captured in MyMultiColumns Report following needs to be done at various components of the MulCol Trial Balance Report.

1. *Report Definition: Repeating over values of system variable which is captured in MyMulti-Columns Report*

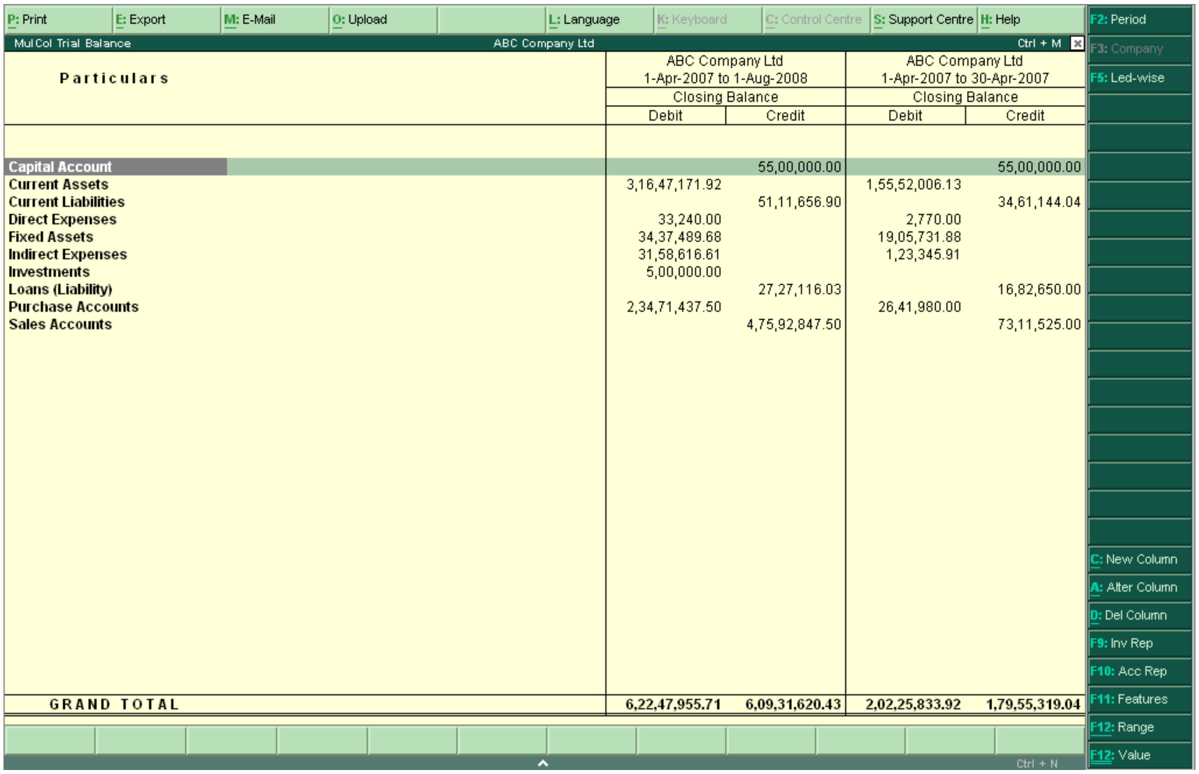
```
[Report: MulCol Trial Balance]
   Repeat      : SVCurrentCompany, SVFromDate, SVToDate
```

2. *Part Definition: Repeating Lines over objects of a Collection.*

```
[Part: MulCol TB Details]
   Lines       : MulCol TB Details
   BottomLines : MulCol TB Total
   Repeat      : MulCol TB Details: MulCol TB GroupLed
```

3. *Line Definition:- Repeating Field*

```
[Line: MulCol TB Details]
   Fields      : MulCol TB Name Field, MulCol TB Amount Field
   Repeat      : MulCol TB Amount Field
```



Mul Col Trial Balance — ABC Company Ltd

Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		ABC Company Ltd 1-Apr-2007 to 30-Apr-2007	
	Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		55,00,000.00
Current Assets	3,16,47,171.92		1,55,52,006.13	
Current Liabilities		51,11,656.90		34,61,144.04
Direct Expenses	33,240.00		2,770.00	
Fixed Assets	34,37,489.68		19,05,731.88	
Indirect Expenses	31,58,616.61		1,23,345.91	
Investments	5,00,000.00			
Loans (Liability)		27,27,116.03		16,82,650.00
Purchase Accounts	2,34,71,437.50		26,41,980.00	
Sales Accounts		4,75,92,847.50		73,11,525.00
GRAND TOTAL	6,22,47,955.71	6,09,31,620.43	2,02,25,833.92	1,79,55,319.04

Figure 1.10 Multi Column Report

1.4 Auto-Column Reports

An Auto column report is one in which multiple columns are repeated by just one click of a button. Trial Balance, Balance Sheet, Stock Summary etc. are some of the default Reports in Tally.ERP 9 which have an Auto column feature.

1.4.1 Designing an Auto Column Report

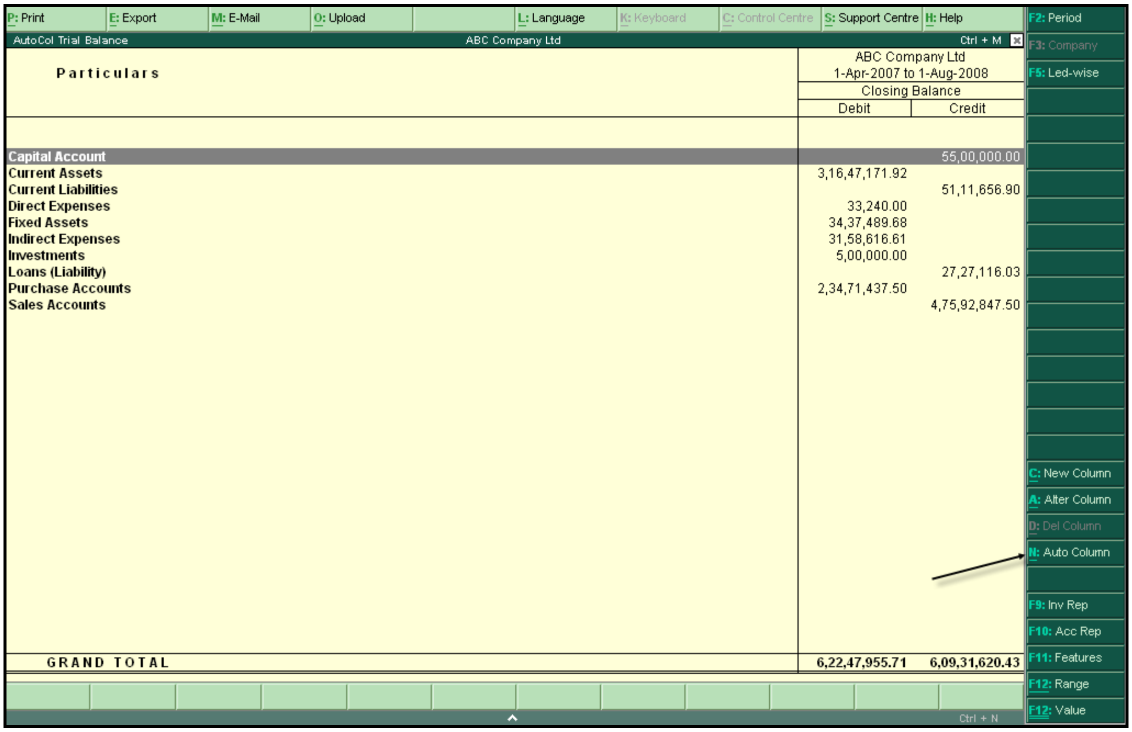
An Auto column Report is similar to a Multi column Report except that in an Auto column Report, a set of columns are repeated instead of only one column. The user input will decide the criteria on which these columns are repeated.

Example: Incorporating Auto Column Feature to Trial Balance report

Step 1 Adding the Configuration Screen to the Form

The Button MyAutoButton is added to Form. Through this Button, the configuration Report 'MyAutoColumns' is arrived at through the Auto columns mode.

```
[Form: MulCol Trial Balance]
   BottomButton: MyAutoButton,
[Button: MyAutoButton]
   Key          : Alt+N
   Action       : Auto Columns : MyAutoColumns
   Title        : $$LocaleString:"Auto Column"
```



AutoCol Trial Balance   ABC Company Ltd

Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008 Closing Balance Debit	 Credit
Capital Account		55,00,000.00
Current Assets	3,16,47,171.92	
Current Liabilities		51,11,656.90
Direct Expenses	33,240.00	
Fixed Assets	34,37,489.68	
Indirect Expenses	31,58,616.61	
Investments	5,00,000.00	
Loans (Liability)		27,27,116.03
Purchase Accounts	2,34,71,437.50	
Sales Accounts		4,75,92,847.50
GRAND TOTAL	6,22,47,955.71	6,09,31,620.43

Figure 1.11   Auto Column Reports

Step 2:The Configuration Report 'MultiAutoColumns'

In the configuration Report shown above, the user will be given with options like 'Days',' Monthly', Yearly' 'Company' etc based on which the columns are repeated. In TDL, these options are external objects.

```
[Collection: MyAuto Columns]
   Title       : $$LocaleString:"Column Details"
   Object      : MyCurrentCompany, MyQuarterly, MyMonthly, MyYearly, MyHalf-
   Yearly
   Filter      : Belongs
   Format      : $$Name, 15
```

;; Belongs is a system formula which filters the objects
;; based on the value of the Methods BelongsIf of all the objects
;; Function Name returns the Name of any given objects

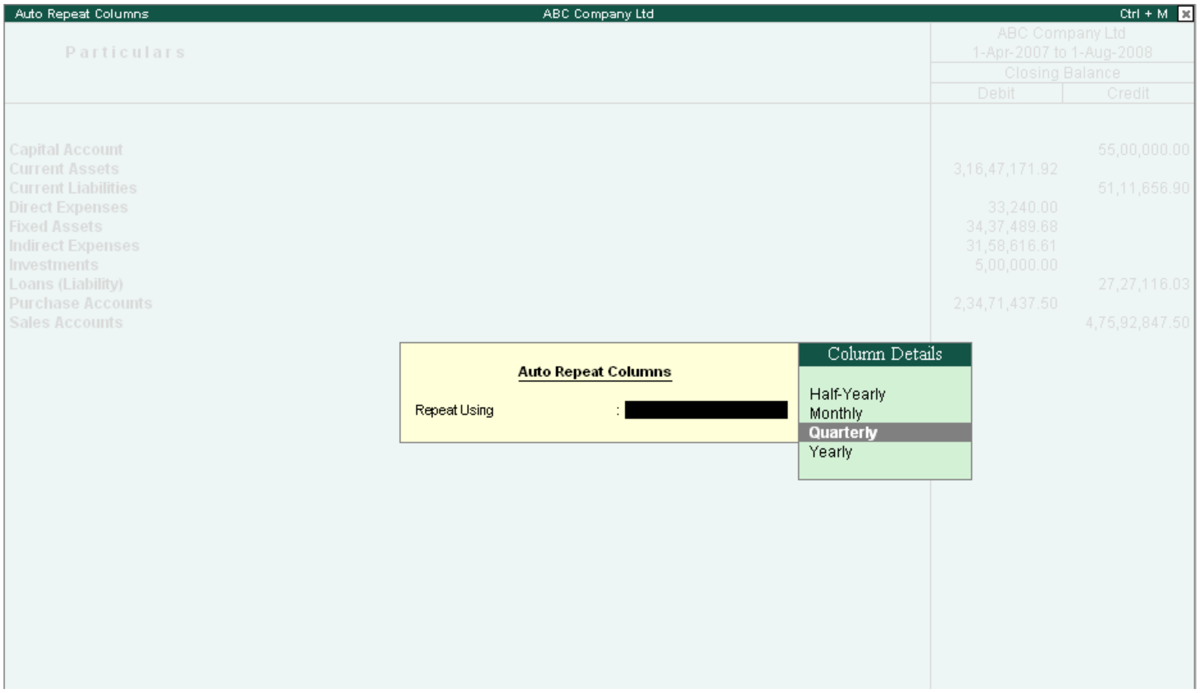
```
[Object: MyCurrentCompany]
   Name        : $$LocaleString:"Company"
   VarName     : "SVCurrentCompany"
   CollName    : "List of Primary Companies"
   BelongsIf   : $$NumItems:ListOfPrimaryCompanies > 1
   IsAgeWise   : No
   Periodicity : ""
```

;; Function NumItems returns the number of selected companies
;; BelongsIf is a method of object MyCurrentCompany, which
;; is used to control the display of the object in the collection

```
[Object: MyQuarterly]
   Name        : $$LocaleString:"Quarterly"
   VarName     : "SVFromDate, SVToDate"
   CollName    : "Period Collection"
   BelongsIf   : "Yes"
   IsAgeWise   : No
   Periodicity : "3 Month"

[Object: MyHalfYearly]
   Name        : $$LocaleString:"Half-Yearly"
   VarName     : "SVFromDate, SVToDate"
   CollName    : "Period Collection"
   BelongsIf   : "Yes"
   IsAgeWise   : No
   Periodicity : "6 Month"
[Object: MyMonthly]
   Name        : $$LocaleString:"Monthly"
   VarName     : "SVFromDate, SVToDate"
   CollName    : "Period Collection"
   BelongsIf   : "Yes"
   IsAgeWise   : No
   Periodicity : "Month"

[Object: MyYearly]
   Name        : $$LocaleString:"Yearly"
   VarName     : "SVFromDate, SVToDate"
   CollName    : "Period Collection"
   BelongsIf   : "Yes"
   IsAgeWise   : No
   Periodicity : "Year"
```

Notes

Columns can be repeated over any collection. It is not restricted only to a Period.



Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008 Closing Balance Debit	 Credit
Capital Account		55,00,000.00
Current Assets	3,16,47,171.92	
Current Liabilities		51,11,656.90
Direct Expenses	33,240.00	
Fixed Assets	34,37,489.68	
Indirect Expenses	31,58,616.61	
Investments	5,00,000.00	
Loans (Liability)		27,27,116.03
Purchase Accounts	2,34,71,437.50	
Sales Accounts		4,75,92,847.50

Auto Repeat Columns

Repeat Using :

Figure 1.12 Auto Repeat Columns

2. When the user selects any one of the options required, the system variables need to be modified so that, the columns can be generated in the parent Report on the basis of these values.

```
[Field: My SelectAuto]
   Use          : Short Name Field
   Table        : MyAutoColumns
   Show Table   : Always

[Field: My AutoColumns]
   Use          : Short Name Field
   Invisible    : Yes
   Set as       : $$Table:MySelectAuto:$VarName
   Set always   : Yes
   Skip         : Yes
```

;; Function Table selects the Object Name from the previous Field MySelectAuto
;; and displays the corresponding method value of VarName

```
   [Field: My CollName]
      Use          : Short Name Field
      Invisible    : Yes
      Set as       : $$Table:MySelectAuto:$CollName
      Modifies     : DSPRepeatCollection
      Set always   : Yes
      Skip         : Yes
```

;; We are modifying the value of the default variable DSPRepeatCollection
;; by the value of the Method CollName from the selected Object
;; DSPRepeatCollection is repeated in the Default Variables SVCurrentCompany,
;; SVFromDate and SVToDate, which gets new values for each column

```
   [Field: My StartPeriod]
      Use          : Short Date Field
      Invisible    : Yes
      Set as       : if $$IsEmpty:$$Table:MySelectAuto:$Periodicity then+
                     ##SVFromDate else if $$Table:MySelectAuto:+
                     $Periodicity = "Day" then ##SVFromDate else +
                     $$LowValue:SVFromDate
      Set always   : Yes
      Modifies     : SVFromDate
      Skip         : Yes
```

;; Value of Variable SVFromDate is set here based on the Periodicity Method.
;; LowValue is a Function that returns beginning date of the Current Period

```
[Field: My EndPeriod]
      Use          : Short Date Field
                         Invisible    : Yes
      Set as       : if $$IsEmpty:$$Table:MySelectAuto:$Periodicity then+
                     ##SVToDate else if $$Table:MySelectAuto:+
                     $Periodicity = "Day" then $$MonthEnd:#DSPStartPeriod+
                     else $$HighValue:SVToDate
      Set always   : Yes
      Modifies     : SVToDate
      Skip         : Yes
```

```
;; Value of Variable SVToDate is set here based on the Periodicity Method.
;; MonthEnd is a Function gives the last day for a given month

   [Field: My SetPeriodicity]
      Use          : Short Name Field
      Invisible    : Yes
      Set as       : if NOT $$IsEmpty:$$Table:MySelectAuto:+
                     $Periodicity then $$Table:MySelectAuto:+
                     $Periodicity else "Month"
      Set always   : Yes
      Modifies     : SVPeriodicity
```

3. The generated values are sent to the Parent Report by using the Form attribute ‘Output’.

```
   [Form: MyAutoColumns]
      No Confirm   : Yes
      Parts        : My AutoColumns
      Output       : My AutoColumns
```

Step 3: Repeating Columns over a Variable and Lines over Objects of a Collection

To repeat columns over a Variable which are captured in an Auto Columns Report, the following needs to be done at various components of the MulCol Trial Balance Report

1. Report Definition: This involves repeating the Values of a System Variable which is captured in MyMultiColumns Report.

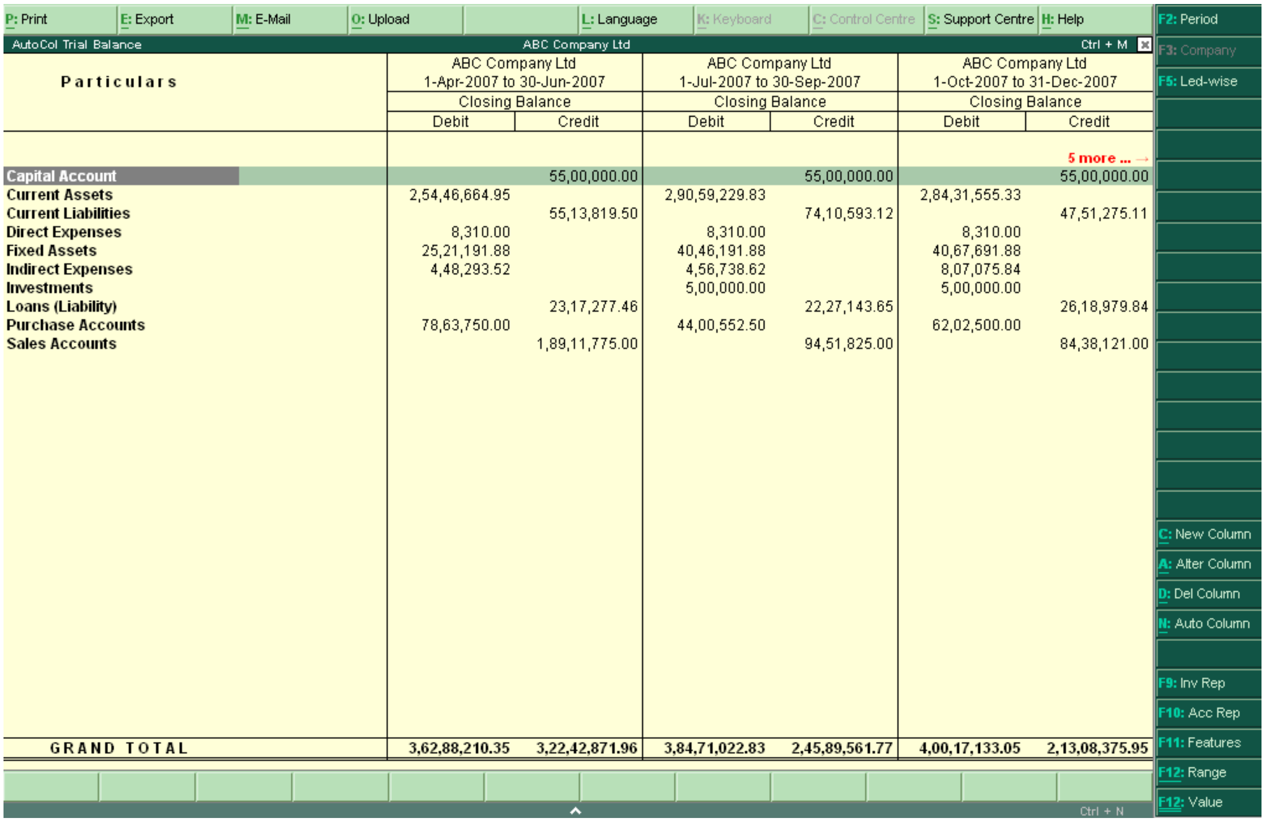
```
   [Report: MulCol Trial Balance]
      Repeat       : SVCurrentCompany, SVFromDate, SVToDate
```

2. Part Definition: This involves repeating Lines over the Objects of a Collection.

```
   [Part: MulCol TB Details]
      Lines        : MulCol TB Details
      BottomLines  : MulCol TB Total
      Repeat       : MulCol TB Details: MulCol TB GroupLed
```

3. Line Definition: This involves repeating a Field.

```
[Line: MulCol TB Details]
   Fields       : MulCol TB Name Field, MulCol TB Amount Field
   Repeat       : MulCol TB Amount Field
```



AutoCol Trial Balance — ABC Company Ltd

Particulars	ABC Company Ltd 1-Apr-2007 to 30-Jun-2007		ABC Company Ltd 1-Jul-2007 to 30-Sep-2007		ABC Company Ltd 1-Oct-2007 to 31-Dec-2007	
	Closing Balance		Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		55,00,000.00		55,00,000.00
Current Assets	2,54,46,664.95		2,90,59,229.83		2,84,31,555.33	
Current Liabilities		55,13,819.50		74,10,593.12		47,51,275.11
Direct Expenses	8,310.00		8,310.00		8,310.00	
Fixed Assets	25,21,191.88		40,46,191.88		40,67,691.88	
Indirect Expenses	4,48,293.52		4,56,738.62		8,07,075.84	
Investments			5,00,000.00		5,00,000.00	
Loans (Liability)		23,17,277.46		22,27,143.65		26,18,979.84
Purchase Accounts	78,63,750.00		44,00,552.50		62,02,500.00	
Sales Accounts		1,89,11,775.00		94,51,825.00		84,38,121.00
GRAND TOTAL	3,62,88,210.35	3,22,42,871.96	3,84,71,022.83	2,45,89,561.77	4,00,17,133.05	2,13,08,375.95

Figure 1.13 Auto Column Report

1.5 Automatic Auto-Column Reports

There may be situations when the columns are required automatically without the intervention of the user when the report is opened. The Attendance Sheet is an example of the Automatic auto-column Report in Tally.ERP 9.

1.5.1 Designing an Automatic Auto Column Report

In order to design an Automatic Autocolumn Report the function SetAutoColumns and the pre-defined variables DoSetAutocolumn and the DSPRepeatCollection are used.

The following points must be considered while creating the automatic auto-column reports:

- The value of the variable DoSetAutoColumn must be set to **Yes**.
- The variable DSPRepeatCollection stores the Collection Name to be repeated.

- The function SetAutoColumns accepts the name of a variable which is repeated over the value of variable DSPRepeatCollection.
- The columns are displayed based on the values in the collection provided by variable DSPRepeatCollection.

Example

Consider the example of creating an auto-column for a Trial Balance. The same report can be modified to have automatic Columns for Multiple selected companies. As mentioned earlier, the following should be resorted to:

The variable DoSetAutoColumn must be set to ***Yes****.*

```
[Report: MulCol Trial Balance]
   Set : DSPRepeatCollection: "List of Primary Companies"
```

The variable DSPRepeatCollection have to set "List of Primary Companies"

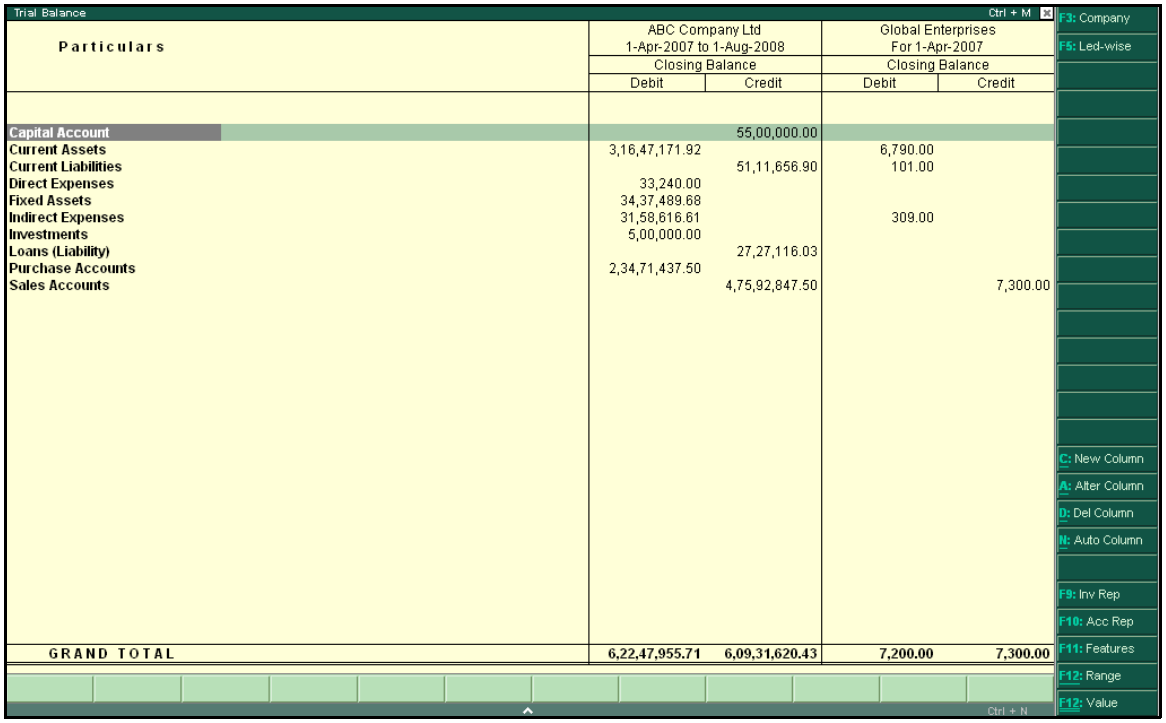
```
[Form: MulCol Trial Balance]
   Option : Set Auto Option : $$SetAutoColumns:SVCurrentCompany
```

Add a dummy option in the Form Definition such that the condition of the same is $$SetAutoColumns:SVCurrentCompany. The variable SVCurrentCompany will be repeated automatically as soon as you enter the report, provided multiple companies are loaded.

Also add the following lines to the Form Definition MultiCol Trial Balance

```
Option : Set Auto Option : $$SetAutoColumns:SVCurrentCompany
[!Form: Set Auto Option]
```

Multiple companies should be loaded for this program. Now when the user selects the Menu Item the following screen is displayed.



Trial Balance

Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		Global Enterprises For 1-Apr-2007	
	Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		
Current Assets	3,16,47,171.92		6,790.00	
Current Liabilities		51,11,656.90	101.00	
Direct Expenses	33,240.00			
Fixed Assets	34,37,489.68			
Indirect Expenses	31,58,616.61		309.00	
Investments	5,00,000.00			
Loans (Liability)		27,27,116.03		
Purchase Accounts	2,34,71,437.50			
Sales Accounts		4,75,92,847.50		7,300.00
GRAND TOTAL	6,22,47,955.71	6,09,31,620.43	7,200.00	7,300.00

Figure 1.14 Displaying Trial Balance for two different companies

1.6 Columnar Report

All the Voucher Reports containing Accounting Information (Ledger and/ or Group Info) available in a Voucher and can be displayed as Columns are categorized as Columnar Reports. For example; Sales Register, Purchase Register, Journal Register, Ledger, etc. where the Voucher Registers can display multiple columns and respective values for each column viz., the ledger, the parent of the ledger, etc. entered in the voucher, as opted by the user.

These types of Reports also use the Auto Column concept for achieving disparate columns.

Stock Registers and Sales Registers are a classic example of Columnar Reports.

2. Printing

In the previous section, we have understood the various types of reports and the techniques to generate the same. The most essential element of Reporting is printing. All the reports must be printable in one form or another.

2.1 Printing Techniques

The techniques used for Printing are as follows:

2.1.1 Menu Action – Print/ Print Collection

Menu Action, Print or Print Collection enters the final Report in Print mode.

Syntax

```
      [Menu: <Menu Name>]
            Add: Key Item:[Position]:<Display Item>:<Unique Key>:<Action Key
      word>:<Action Parameter>
```

;; where Action Keyword can be Print or Print Collection which triggers a list and displays the
;; final report based on user selection

Example

```
   [#Menu: Printing Menu]
      Add   : Key Item  : My Ledgers: L : Print Collection: Ledger Vouchers
      Add   : Key Item  : My Day Book: D : Print: Day Book
```

In the above example, we have added the Item My Ledgers, which has an action Print Collection associated to it. It displays a collection bearing the List of ledgers which on user selection, enters the final report in Print Mode. On accepting it directly goes to the printer.

2.1.2 Button Action – Print Report

Another method of printing reports is by way of associating a Button with an action Print Report at the Form definition. Action Print Report prints the current report by default. This action accepts Report Name as its parameter. If any report other than current needs to be printed, an additional parameter containing Report Name needs to be specified. The current report can pass the user selection to the printing report through a default collection called Parameter Collection.

Syntax

```
      [Button: <Button Name>]
            Action: <Print Report>[: Action Parameter]
```

Example

Consider a report displaying a list of employees, wherein the user selects the required employees for whom pay slips need to be printed. On clicking the Print Button, the current report bearing list of employees is not required. A new report printed for various pay slips allotted to selected employees is needed.

```
   [Button: Print Selected Pay slips]
```

;; Associate this button to the current report displaying list of employees

```
      Key   : Alt + F11 Title  : "Print Selected Pay slips"
 ;; Multiple Payslip Print Report will be printed on activation of this Button
;; The Report should be altered to include inbuilt Collection Parameter
;; Collection to print the user selection for list of employees

      Action : Print Report : Multi Pay Slip Print
      Scope  : Selected Lines

   [#Report: Multi Pay Slip Print]
      Collection: Parameter Collection
```

In the above example, the Button Print Selected Pay slips is defined with Action 'Print Report' which also has an action parameter i.e., the Report Name to be printed. The scope of the Button is Selected Lines, which means that the final Report 'Multi Pay Slip Print' must contain only the selected Objects from the current Report. The user selection is passed to the new Report through a Collection called 'Parameter Collection' which must be used in the destination Report 'Multi Pay Slip Print'. So, the Report 'Multi Pay Slip Print' can be modified and added to the collection 'Parameter Collection'.

2.2 Page Breaks

A Page Break is the point at which one page ends and another begins. Handling Page Breaks is very important, since the current page should indicate the continuation to the next page and the next page must indicate that the current page is continued from a previous page.This indicates that there has to be a closing identifier i.e., closing page break information and an opening identifier i.e., opening page break information.

In other words, Page Breaks specify the headers and footers for every page, and is printed across multiple pages. Closing Page Break starts printing from the first page and prints on every page except the last page. For e.g., Continued... to be printed at the bottom of each page. An opening Page Break starts printing from the second page till the last page. Closing Page Break is specified before Opening Page Break since in any circumstance, closing page break will be encountered first.

In TDL, Page Break can be handled both vertically as well as horizontally.

2.2.1 Types of Page Breaks

Vertical Page Breaks

In cases where a report containing data cannot be printed in a single page, one needs to use vertical page breaks.

Vertical Page Breaks can be specified at 2 levels; viz., Form and Part.

Form Level Page Break

Vertical Page Breaks can be specified at Form through the Form Attribute Page Break. It takes 2 parameters, viz., First Part for Closing Page Break and Second Part for Opening Page Break.

Syntax

```
      [Form: <Form Name>]
            Page Break : <Closing Part>, <Opening Part>
```

Example

Consider a Trial Balance report of a company, which requires the title and address of the Company in the first page and the grand total in the last page. In the pages between the first and last page, the text may be required to be continued at the end of each page and Company Name and Address at the beginning of each page.

```
   [Form: My Trial Balance]
   Page Break : Cl Page Break, Op Page Break
;; where both Cl Page Break and Op Page Break are Parts

   [Part: Cl Page Break]
      Lines : Cont Line

      [Line: Cont Line]
         Fields: Cont Field
         Border: Full Thin Top

         [Field: Cont Field]
            Set As       : "Continued…"
            Full width   : Yes
            Align        : Right
   [Part: Op Page Break]
      Parts : DSP OpCompanyName, DSP OpReportTitle
      Vertical: Yes
```

In the above example, Closing Page Break is defined to print Continued at the end of every continued page. Opening Page Break is defined to print the Company Name and Report Title at beginning of all the continuing pages. Since more than one part is used within a part definition, specify the alignment to Vertical, if required.

Part Level Page Breaks

Vertical Page Breaks can be specified at Part through the Part Attribute Page Break. This is generally used when the Page Totals are to be printed for each closing and opening pages.

It takes 2 parameters, viz., First Line for Closing Page Break and Second Line for an Opening Page Break.

Syntax

```
      [Part: <Part Name>]
            Page Break : <Closing Line>, <Opening Line>
```

Example

Consider a Trial Balance Report of a company, where we may require the running page totals to be printed at the end and beginning of each page.

```
   [Part: My Trial Balance]
      Page Break : Cl Page Break, Op Page Break
;; where both Cl Page Break and Op Page Break are Lines

   [Line: Cl Page Break]
      Use    : Detail Line
      Local  : Field: Particulars Fld: Set As: "Carried Forward"
      Local  : Field: DrAmt Fld: Set As: $$Total:DrAmtFld
      Local  : Field: CrAmt Fld: Set As: $$Total:CrAmtFld
      Local  : Field: NetAmt Fld: Set As: $$Total:NetAmtFld
      Border: Full Thin Top

   [Line: Op Page Break]
      Use    : Cl Page Break
      Local  : Field: Particulars Fld: Set As: "Brought Forward"
```

In the above example, the Line Cl Page Break is defined to use the pre-defined Detail Line and the relevant fields are modified locally to set the respective values. Similarly, Line Op Page Break is defined to use the above defined line Cl Page Break which locally modifies only the particulars field.

Horizontal Page Breaks

Horizontal Page Breaks are used if the number of columns run into multiple pages.

Line Level Page Breaks

Horizontal Page Breaks can be specified at Line through the Line Attribute Page Break. This is generally used to repeat a closing column at every closing page and opening columns at every opening page. It takes 2 parameters, viz., First Field for Closing Page Break and Second Field for Opening Page Break.

Syntax

```
        [Line: <Line Name>]
              Page Break : <Closing Field>, <Opening Field>
```

Example

Consider a Columnar Sales Register Report of a company, where multiple columns are printed across the pages. Some fixed columns are required in subsequent pages which makes it easy to map the columns in subsequent pages.

```
   [#Line: DSP ColVchDetail]
      Page Break : Cl Page Break, Op Page Break
;; where both Cl Page Break and Op Page Break are Fields

   [Field: Cl Page Break]
   [Field: Op Page Break]
      Fields      :  DBC Fixed, VCH No
```

In the example mentioned above, the Field Cl Page Break is defined as Empty since no Closing Column or Field is required and Field Op Page Break is defined with further fields DBC Fixed and VCH No which are available in default TDL.

Form Level Page Break	**Part Level Page Break**	**Line Level Page Break**
This is a Vertical Page Break	This is a Vertical Page Break	This is a Horizontal Page Break
Page Break attribute accepts Part Names as its value	Page Break attribute accepts Line Names as its value	Page Break attribute accepts Field Names as its value
Multiple Parts (parts within parts) can be printed both at closing and opening page breaks	Multiple lines (lines within lines) can be printed at both closing and opening page break	Multiple Fields (Fields within Fields) can be printed at both closing and opening page break
Form Level Page Breaks cannot handle running Page Totals	Running Page Totals can be handled with Part Level Page Break	Column Page Totals can be handled with Line Level Page Break

TABLE 9:1 Comparison between different pagebreaks

2.3 Frequently Used Attributes and Functions

2.3.1 Attributes

Line Level Attribute – Next Page

The Next Page attribute specifies the cut off line that gets printed in the subsequent page. It accepts a logical formula as its parameter.

Syntax

```
[Line: <Line Name>]
      Next Page: <Logical Formula>
```

Example

```
[Line: DSP Vch Explosion]
   Next Page   : (($$LineNumber = $$LastLineNumber) AND $$IsLastOfSet)
```

The attribute – Preprinted/ PrePrinted Border

The Attribute Preprinted or Preprinted Border can be specified at Part, Line and Field Definitions. These attributes work in conjunction with preprinted/ plain button in the Print Configuration screen. When the Preprinted attribute is set to Yes, the contents of the current Part, Line or Field will be left blank assuming the same to be pre-printed. When the Preprinted Border attribute is set to Yes, the borders used in the current Part, Line or Field will be assumed to be pre-printed.

Syntax

```
[Line: <Line Name>]
      Preprinted: <Logical Value>
```

Example

```
[Line: Company Name]
   Preprinted  : Yes
```

2.3.2 Functions

PageNo and PartNo

The PageNo function returns the current Page Number while the PartNo function returns the current Part Number of the page. These functions do not require any parameter and the return type for PageNo is Number and PartNo is String.

Syntax

```
$$PageNo
$$PartNo
```

Example

```
[Field: My PageNo]
   Set as: "Page " + $$String:$$PageNo  + " (" + $$PartNo + ")"
```

IsLastOfSet

This function is used to check if the current Form is the last Form being printed. This function doesn't require any parameter. It returns the logical value as True if the current Form is the last Form being printed, else returns False.

Syntax

```
$$IsLastOfSet
```

Example

```
   [Line: DSP Vch Explosion]
      Next Page   : (($$LineNumber = $$LastLineNumber) AND $$IsLastOfSet)
```

DoExplosionsfit

In the process of printing, if a line is exploded, then this function can be used to check whether the exploded part fits within the current page. This function also doesn't require any parameter and returns its logical value. It returns its logical value as Yes, if it is true.

Syntax

```
      $$DoExplosionsFit
```

Example

```
   [Line: EXPSMP InvDetails]
      NextPage    : NOT $$DoExplosionsFit OR (($$LineNumber = $$LastLineNumber)
```

BalanceLines

This function is used to check the balance number of lines in the repeated lines, including the exploded part-lines present, in a given part. **Scroll : Vertical** must be specified at the Part definition in order to use this function. This function too does not require any parameter and returns a Numerical value.

Syntax

```
      $$BalanceLines
```

Example

```
   [Line: AccType Detail]
      NextPage    : ($$BalanceLine > 0) AND (($$BalanceLines < 5)
```

In the example mentioned above, if the Balance number of lines is between 0 and 5, the remaining lines will be printed on the next page.

2.4 Validation and Controls

Data validation and controls in Tally can be done at two levels, either at the Platform level or at the TDL level. TDL Programmers do not have control over any of the Platform level validations. TDL Programmers can only add validation and controls at the TDL Level.

Let us understand some of the TDL Level validation and control mechanisms.

2.4.1 Field Level Attribute — Validate

This attribute checks if the given condition is satisfied. Unless the given condition is satisfied, the user cannot move further. In other words, if the given condition for Validate is not satisfied, the cursor remains placed on the current field without moving to the subsequent field. It does not display any error message.

Syntax

```
Validate : <Logical Formula>
```

Example

```
[Field: CMP Name]
   Use          : Name Field
   Validate     : NOT $$IsEmpty:$$Value
   Storage      : Name
   Style        : Large Bold
```

In this code snippet:

- The field CMP Name is a field in Default TDL which is used to create/ alter a Company.
- Validate stops the cursor from moving forward, unless some value is entered in the current field.
- The function, IsEmpty returns a logical value as True, only if the parameter passed to it contains NULL.
- The function, Value returns the value entered in the current field.

Thus, the Attribute Validate used in the current field, controls the user from leaving the field blank and forces a user input.

2.4.2 Field Level Attribute — Unique

This attribute takes a logical value. If it is set to **Yes**, then the values keyed in the field have to be unique. If the entries are duplicated, an error message, **Duplicate Entry** pops up. This attribute is useful when a Line is repeated over UDF/Collection, in order to avoid a repetition of values.

Syntax

```
Unique: [Yes / No]
```

Example

```
[!Field: VCHPHYSStockItem]
   Table : Unique Stock Item : $$Line = 1
   Table : Unique Stock Item, EndofList
   Unique: Yes
```

In this code snippet, the field, VCHPHYSStockItem is an optional field in DefTDL which is used in a Physical Stock Voucher. The attribute, Unique avoids the repetition of Stock Item names.

2.4.3 Field Level Attribute — Notify

This attribute is similar to the attribute Validate. The only difference here, is that it flashes a warning message and the cursor moves to the subsequent field. Here, a System Formula is added to display the warning message.

Syntax

```
Notify : <System Formula> : <Logical Condition>
```

Example

```
[!Field: VCH NrmlBilledQty]
   Set as            : if @@HasInvSubAlloc then $$CollQtyTotal: +
   BatchAllocations  : $BilledQty else @ResetVal
   Skip On           : @@HasInvSubAlloc
   Style             : if @@IsInvVch then "Normal" else "Normal Bold"
   Notify            : NegativeStock:##VCFGNegativeStock AND +
                       @@IsOutwardType AND$$InCreateMode AND +
                       $$IsNegative:@@FinalStockTotal
```

In this code snippet, VCH NrmlBilledQty is a default optional field in DefTDL used in a Voucher. 'Notify' pops up as a warning message, if the entered quantity for a stock item is more than the available stock and the cursor moves to the subsequent field.

2.4.4 Field Level Attribute — Control

The attribute Control is similar to Notify. The only difference is that it does not allow the user to proceed further after displaying a message. The cursor does not move to the subsequent field.

Syntax

```
Control : <System Formula : Logical Condition>
```

Example

```
[Field: VCH Number]
   Use               : Voucher Number Field
   Inactive          : @@NoVchNumbering
   Skip On           : @@AutoVchNumbering
   Control           : DuplicateNumber : @@NoDupVchNumbering AND +
                       (NOT $$InAlterMode OR NOT @SameVchTypeAndNum)AND +
                            $$IsDuplicateNumber:$$Value:
   SameVchTypeAndNum : $VoucherTypeName = ##ORIGVchType AND +
                           $$Value = ##ORIGVchNum
   Validate          : (@@NoDupVchNumbering AND NOT $$IsEmpty:$$Value) +
                                   OR NOT @@NoDupVchNumbering
      Keys                 : PrevVchNumber
```

In this code snippet, the field, VCH Number is a default field in DefTDL used in a Voucher. The duplication of voucher numbers for a particular voucher type is prevented by using the attribute, Control. The difference between the field Attributes, Validate, Notify and Control are:

Field Attributes	Displays Message	Cursor Movement
Validate	No	Restricted
Notify	Yes	Not Restricted
Control	Yes	Restricted

TABLE 9:2 Difference between the validation control attributes

2.4.5 Form Level Attribute — Control

This attribute achieves a higher level of control on the contents of a Form over other controls used at the Lower levels of the Form. If the condition specified with Control is not satisfied, then the Form displays an error message while trying to save. The Form cannot be saved until the condition in the attribute Control is fulfilled.

Syntax

```
      Control : <String Formula : Logical Formula>
```

Example

```
   [Form: Voucher]
      Control : DateBelowBooksFrom  : $Date < +
                $BooksFrom:Company  :##SVCurrentCompany
      Control : DateBelowFromDate   : $Date < $$SystemPeriodFrom
      Control : DateBeyondToDate    : $Date > $$SystemPeriodTo
```

In the example, Voucher is a default Form. While creating a voucher, the attribute, Control does not accept dates beyond the financial period or before beginning of the books.

2.4.6 Menu Level Attribute — Control

The attribute, Control restricts the display of Menu Items, based on the given condition.

Syntax

```
      Control : <Item Name> : <Logical condition>
```

Example

```
   [!Menu: Gateway of Tally]
      Key Item    : @@locAccountsInfo : A : Menu : Accounts Info. : NOT +
                    $$IsEmpty:$$SelectedCmps
      Control     : @@locAccountsInfo : $$Allow:Create:AccountsMasters OR +
                    $$Allow:Alter:AccountsMasters
```

In this code snippet, the Menu, Gateway of Tally is a default optional menu definition in DefTDL. The Menu Item, Account Info., will be displayed only if the given condition is satisfied. The function, **Allow** checks whether the current user has the rights to access the report displayed under the current Menu item. The value Accounts Masters has been derived from the attribute, Family at the report definition.

2.4.7 Report Level Attribute — Family

The value specified with the attribute, Family is automatically added to the security list as a pop-up while assigning the rights under Security Control Menu.

Syntax

```
[Report: <Report Name>]
      Family : <String Value>
```

Example

```
[Report: Ledger]
   Family : "Accounts Masters"
```

In this code snippet, the Accounts Masters will get added to the Security list. Without the user rights for Accounts Masters in the Security controls, this report neither be created, altered nor viewed.

Learning Outcome

- Tally.ERP 9 caters to 3 different types of Reports. These are:
  - Tabular Report: - Report with fixed number columns which can be configured
  - Hierarchical Report :- Report in successive levels or layers
  - Columnar Report :- Report with multiple columns
- The Explode attribute is an attribute of the line, which is used to take the current data from the Line Object.
- $$KeyExplode function gives the current status of the keys Shift and Enter. This is used as a condition to check if the user has pressed the Shift+Enter Keys.
- The Multi column report is a report in which a column is repeated based on the criteria specified by user.
- Page Break is the point at which one page ends and another begins.
- Data validation and controls in Tally can be done at two levels, either at the Platform level or at the TDL level.

Voucher and Invoice Customisation

Introduction

A voucher is a primary document that contains all the information regarding a transaction. To begin with, it is necessary to understand the classification of vouchers and their structure. Voucher and Invoice Customisation will be dealt with later in this Topic.

1. Classification of Vouchers

For every transaction in Tally, you can make use of an appropriate voucher to enter all the required details. Vouchers are broadly classified into three types:

- Accounting Vouchers
- Inventory Vouchers
- Accounting-cum-Inventory Vouchers

1.1 Accounting Vouchers

Accounting Vouchers imply recording transactions which require only the accounting details that do not have any impact on the inventory. Receipt, Payment, Contra and Journal Vouchers are all Accounting Vouchers.

Notes

These transactions affect only the Accounting Reports.

In cases where the option **Inventory Values are affected**? (which is used for Journal/ Payment/ Receipt entries) is set to **Yes** in the Ledger Master, the entries made will also accept the stock items. However, this is not a standard business practice. Entries of this sort, are usually reflected in the Inventory Reports.

1.2 Inventory Vouchers

Inventory Vouchers imply the recording of transactions which require details pertaining only to the inventory and do not have any impact on accounts. Stock Journal and Physical Stock Vouchers are both Inventory Vouchers.

These transactions do not affect the Accounting Reports except when the Closing Stock value is computed and the option if Integrate Accounts and Inventory option is set to Yes in the F11 Accounting/Inventory Features.

1.3 Accounting-cum-Inventory Vouchers

Accounting-cum-Inventory Vouchers are transactions which contain details pertaining to both Accounts as well as Inventory. Purchase Order, Receipt Note, Rejection In, Debit Note, Purchase, Sales Order, Delivery Note, Rejection Out, Credit Note, Sales are all Accounting-cum-Inventory Vouchers.

Purchase Orders, Receipt Notes, Rejection Ins, Sales Orders, Delivery Notes and Rejection Outs only affect the Inventory Reports whereas Debit Notes, Purchase Notes, Credit Notes and Sales affect the Accounting as well as Inventory Reports, if the Tracking Number is set to Not Applicable else it affects only the Accounting Reports.

2. The Structure of a Voucher Object

A Voucher Object store two types of information, Base Information and Actual Entries.

Base Information consists of base methods like Voucher Number, Date, Reference, Narration and so on, which are common to all the voucher types.

Actual Entries are the entries pertaining to Accounts and Inventory.

The following six collections have been introduced to handle transactions based on the three types of vouchers explained earlier. They are:

- Ledger Entries
- Inventory Entries
- All Ledger Entries
- All Inventory Entries
- Inventory Entries In
- Inventory Entries Out

The hierarchy of Voucher Objects is as shown below:

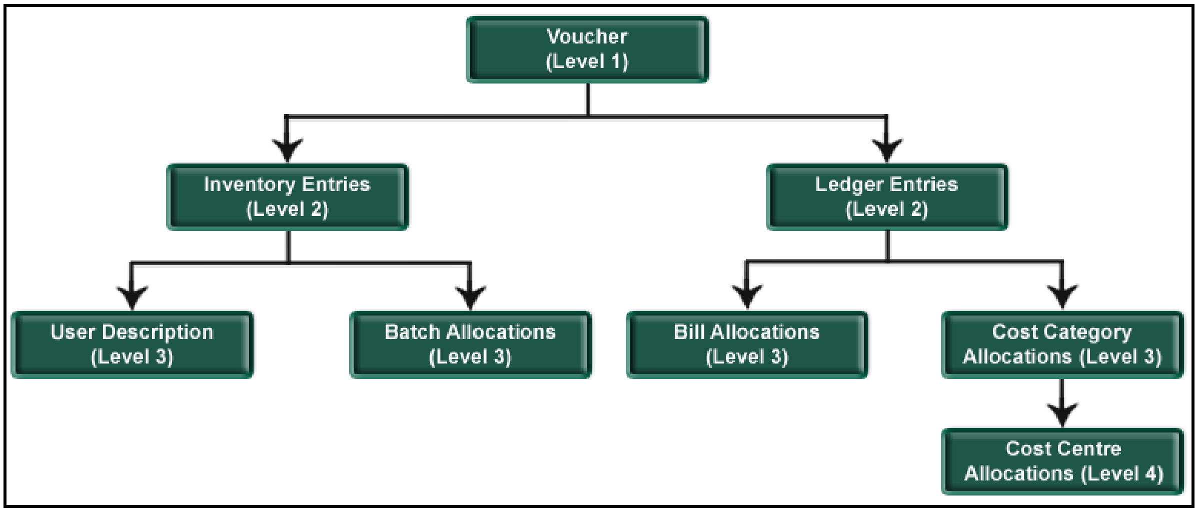

Voucher (Level 1)
Inventory Entries (Level 2)
Ledger Entries (Level 2)
User Description (Level 3)
Batch Allocations (Level 3)
Bill Allocations (Level 3)
Cost Category Allocations (Level 3)
Cost Centre Allocations (Level 4)

Figure 1.1 Hierarchy of Voucher objects

The base entries of a Voucher are Date, Voucher Type, Voucher Number, etc.

The first level consists of two basic collections namely, Ledger Entries and Inventory Entries. Each Ledger Entry Object has its own **Base Methods** like Ledger Name, Amount, Bill Allocation Collec¬tion and Cost Category Allocation Collection. Each Cost Category Allocation Object in turn, contains its own Methods, which are Name, Amount and a Cost Centre Allocation Collection.

Accounting Vouchers use collections of the following type:

- Ledger Entries
- All Ledger Entries

Inventory Vouchers use collections of the following type:

- Inventory Entries
- All Inventory Entries
- Inventory Entries In
- Inventory Entries Out

Accounting-cum-Inventory Vouchers use collections of the following type:

- Ledger Entries
- All Ledger Entries
- Inventory Entries
- All Inventory Entries

3. Customisation

A user usually enters transactions in a voucher and prints it in the default format provided. However, there may be instances, when the user would want to have it printed in a format other than the default one provided in Tally. In such circumstances, the user may have to get it customised according to the company needs.

Incases where, there is a requirement for customisation, adhere to the following steps:

1. Analyse the format required by the company to judge whether
   - The requirement can be met with the default format with some minor changes.

     OR

   - A new format needs to be designed
2. Check whether any additional input fields are required. If required, add the appropriate UDFs at relevant places.
3. Identify the definitions that need to be altered to suit the user requirements.

In this chapter we would be referring input screens as Vouchers and print screens as Invoice.

3.1 Voucher Customisation

Let us consider the following examples inorder to understand the concept of Voucher Customisation.

Case 1

Problem Statement

A Company named ‘ABC Company Ltd’ needs the Cheque No., Date and Bank Name printed on a Payment/ Receipt Voucher and Receipt. There should also be an option of whether the Cheque details are to be printed or not.

Solution

Step 1 : Add additional fields to capture the Bank Name, Cheque Number and Cheque Date

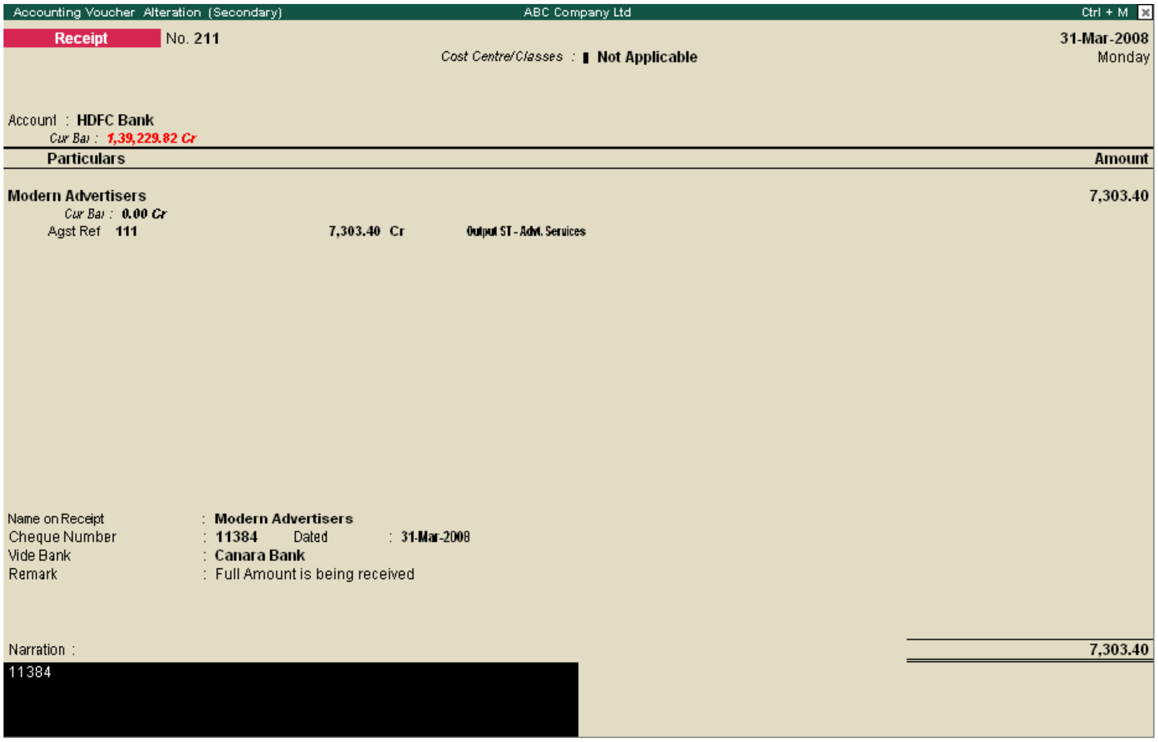
For this the following UDFs are created.

```
[System: UDF]
   BankName     : String     : 1000
   NarrWOCh     : String     : 1001
   ChequeNumber: Number      : 1000
   ChqDate      : Date       : 1000
```

The UDFs mentioned above are used in the existing Part VCH Narration.

```
[#Part: VCH Narration]
;; Modify the Narration Part to add the details
   Add       : Option    : BankDet VCH Narration : @@IsPayment OR @@IsReceipt
   Add       : Option    : BankDet VCH NarrationRcpt: @@ReceiptAfterSave
```

On entering the required details, the screen of the Receipt Voucher looks like this:



Accounting Voucher Alteration (Secondary) ABC Company Ltd Ctrl + M

Receipt No. 211 31-Mar-2008
Cost Centre/Classes : ▮ Not Applicable Monday

Account : HDFC Bank
Cur Bal : 1,39,229.82 Cr

Particulars				Amount
Modern Advertisers Cur Bal : 0.00 Cr				7,303.40
Agst Ref	111	7,303.40 Cr	Output ST - Advt. Services	

Name on Receipt : Modern Advertisers
Cheque Number : 11384 Dated : 31-Mar-2008
Vide Bank : Canara Bank
Remark : Full Amount is being received

Narration : 7,303.40
11384

Figure 1.2 Alteration screen of a Receipt Voucher

Step 2

The Configuration screen of Receipt and Payment Voucher is altered to add a new option. In this, the existing Parts Payment Print Config and Receipt Print Config have been altered.

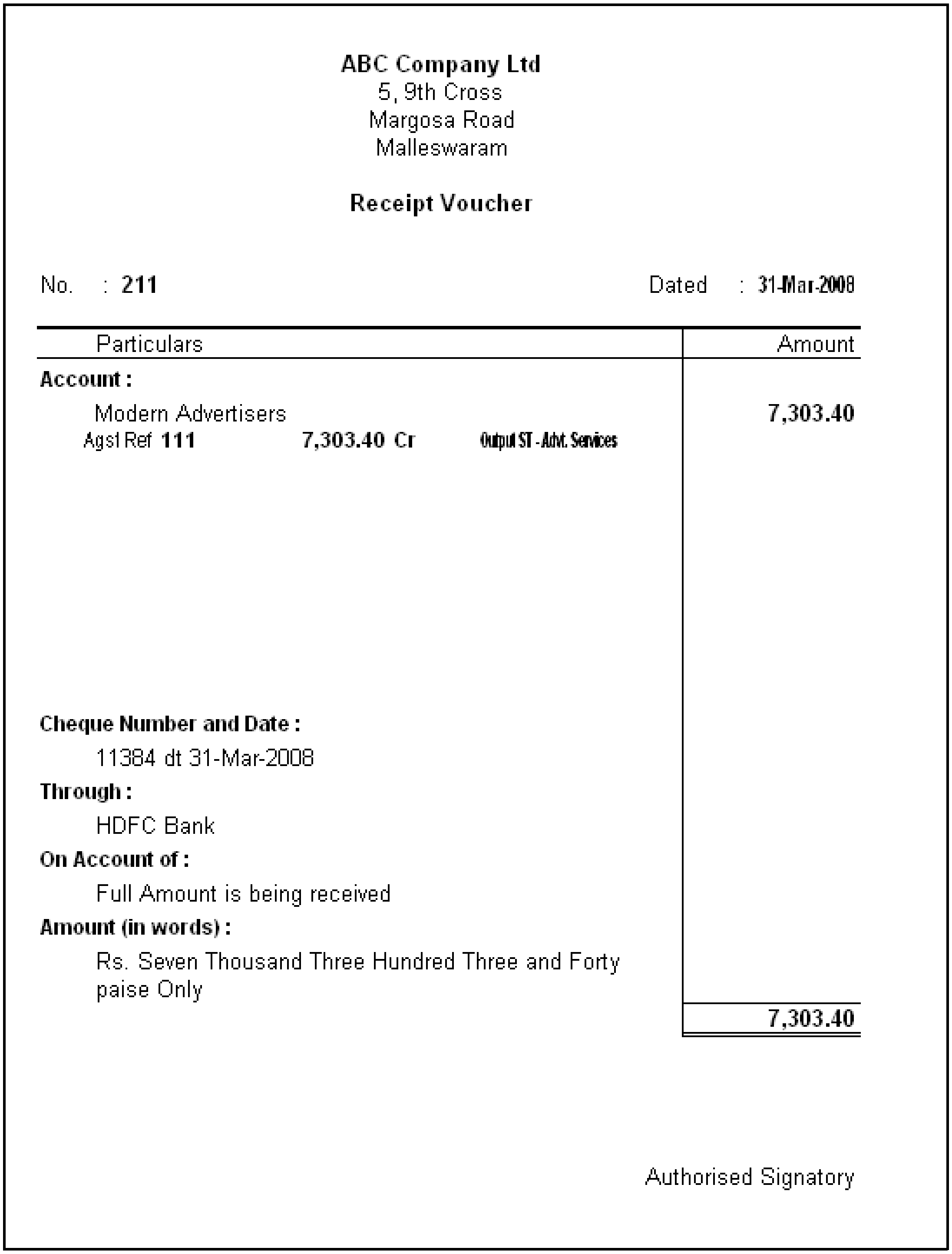
```
;; Payment Config Changes
[#Part: Payment Print Config]
   Add       : Lines    : Before: PPRVchNarr : PPR ChqDetails

;; Receipt Config Changes
[#Part: Receipt Print Config]
   Add       : Lines    : After: PPRWithCost: PPR ChqDetails
```

Step 3

The existing Field PPR Narr and Part PPRBottomDetails is altered to get the required Receipt / Payment Voucher.

```
[#Field: PPR Narr]
   Option         : PPR Narr Rct Pymt

[#Part: PPRBottomDetails]
   Option         : PPRBottomDetails Rct Pymt: (@@IsPayment OR @@IsReceipt) AND
   ##PPRChqInfo
```

The print out of a Customised Receipt Voucher is as shown:



ABC Company Ltd
5, 9th Cross
Margosa Road
Malleswaram

Receipt Voucher

No. : **211** Dated : **31-Mar-2008**

Particulars	Amount
Account :	
Modern Advertisers	**7,303.40**
Agst Ref **111** **7,303.40 Cr** **Output ST - Advt. Services**	
Cheque Number and Date :	
11384 dt 31-Mar-2008	
Through :	
HDFC Bank	
On Account of :	
Full Amount is being received	
Amount (in words) :	
Rs. Seven Thousand Three Hundred Three and Forty paise Only	
	7,303.40

Authorised Signatory

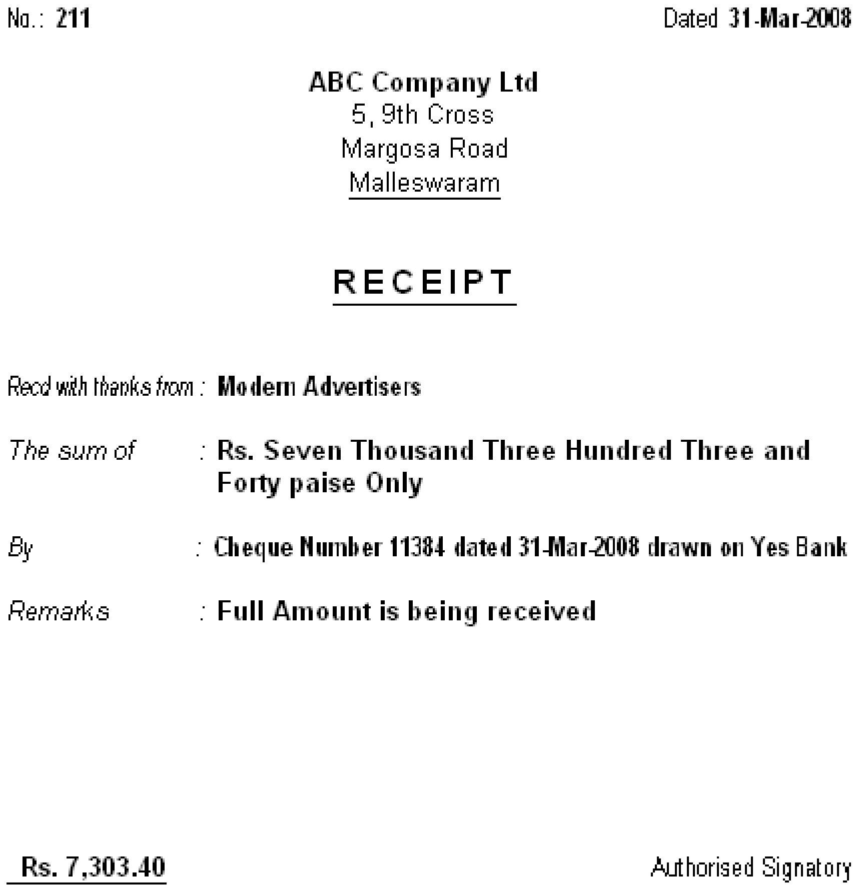
Figure 1.3 Print preview of a customised Receipt Voucher

Step 4

The existing Field PRCT Thru is altered to get the required Receipt/Payment Voucher.

```
[#Field: PRCT Thru]
   Option    : PRCT Thru Rct Pymt: @@IsReceipt
```

The print out of a Customised Receipt is as shown.



No.: 211 Dated 31-Mar-2008

ABC Company Ltd
5, 9th Cross
Margosa Road
Malleswaram

RECEIPT

Recd with thanks from : Modern Advertisers

The sum of : Rs. Seven Thousand Three Hundred Three and Forty paise Only

By : Cheque Number 11384 dated 31-Mar-2008 drawn on Yes Bank

Remarks : Full Amount is being received

Rs. 7,303.40 Authorised Signatory

Figure 1.4 Print Preview of a customised Receipt Voucher

Case 2

Problem Statement

Consider adding columns for Marks and **Number of Packages** to Sales Voucher, instead of lines which are already available by default in Tally.

Solution

To add a column in the Invoice screen, you should know:

- The position in which you have to add a field
- The number of lines to be altered to incorporate the new field
- The type of UDF required for the field (if required)

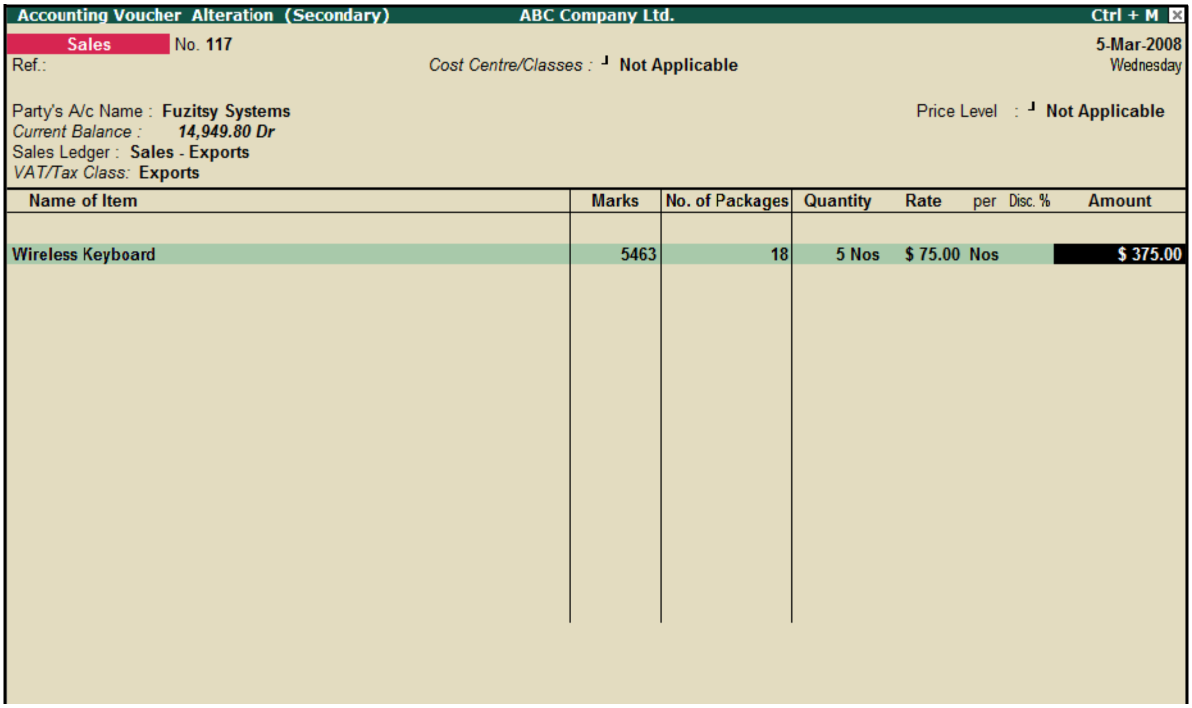
The steps to be followed are enlisted below:

- Firstly, you need to identify the lines that have to be altered to add the required fields.
- Check the field name in the column title and the details of lines in the inventory entries made. Similarly, check the ledger entries collection including the batch allocations, total and subtotal lines. Check all the lines that may be effected in the invoice portion.
- Add the field in all the lines found.

The following lines are to be altered to achieve the required modification.

```
;; Invoice Column Headings1 without class
[Line : EI ColumnOne]
;; Invoice Column Headings2 with class
[Line : EI ColumnTwo]
;; Invoice Inventory Entries without Class
[Line : EI InvInfo]
;; alternate quantity details line
[Line : STKVCH AltUnits]
 ;; Invoice Inventory Entries with Class
[Line : CI InvInfo]
 ;; are added at the form level
[Form : Export Invoice]
```

The following screen shows two input fields added or relocated in the Inventory Entries details:



Accounting Voucher Alteration (Secondary) ABC Company Ltd. Ctrl + M

Sales No. 117 5-Mar-2008 Wednesday

Ref.: Cost Centre/Classes : Not Applicable

Party's A/c Name : Fuzitsy Systems Price Level : Not Applicable

Current Balance : 14,949.80 Dr

Sales Ledger : Sales - Exports

VAT/Tax Class: Exports

Name of Item	Marks	No. of Packages	Quantity	Rate	per	Disc. %	Amount
Wireless Keyboard	5463	18	5 Nos	$ 75.00	Nos		$ 375.00

Figure 1.5 Voucher Alteration screen with new fields

Refer to the sample code for the same.

Case 3

Problem Statement

Consider adding a Subform for a stock item to enter the Height and Width. The dimension is calculated on the basis of the Height and Width entered, and the same is reflected in the Quantity field.

Solution

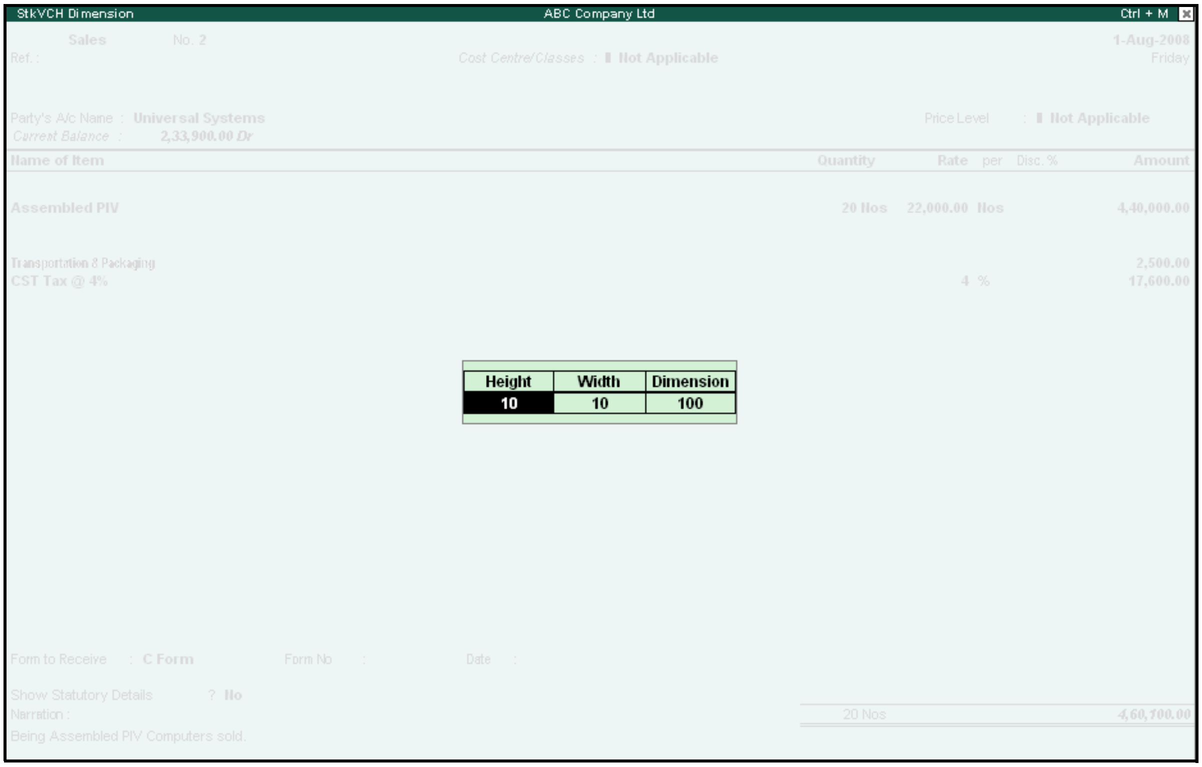
To add a Subform, one should know:

- The field at which a Subform needs to be called, with or without any condition.
- How to defin a Subform Report and its components.
- Whether the Subform would effect the main screen from which it was called, with any modifications.

Proper care should be taken to consider all the situations, while addressing similar requirements, such as with or without activation of Actual and Billed Quantity, with or without Batch wise screen.

The following lines are to be altered to achieve the required modification.

```
[#Field: VCHACC StockItem]
   Add : SubForm : At Beginning : StkVCH Dimension : NOT $$IsEnd:$StockItemName
```



StkVCH Dimension   ABC Company Ltd   Ctrl + M

Sales   No. 2   1-Aug-2008

Ref. :   Cost Centre/Classes : Not Applicable   Friday

Party's A/c Name : Universal Systems   Price Level : Not Applicable

Current Balance : 2,33,900.00 Dr

Name of Item	Quantity	Rate	per	Disc. %	Amount
Assembled PIV	20 Nos	22,000.00	Nos		4,40,000.00
Transportation 8 Packaging					2,500.00
CST Tax @ 4%		4	%		17,600.00

Height	Width	Dimension
10	10	100

Form to Receive : C Form   Form No :   Date :

Show Statutory Details ? No

Narration :   20 Nos   4,60,100.00

Being Assembled PIV Computers sold.

Figure 1.6 Sub Forms

Refer to the sample code for the same.

Case 4

Problem Statement

Altering an existing Discount column that would change the default working of Tally.

Solution

To achieve this, first change the default Discount column from Percentage to Amount. The changes that should be done in the default Tally screen are:

- Reformat Discount at Price level

```
[#Field: MPSDiscountTitle]
   Set as   : "Discount Amt"
```

- Reformat Discount at Inventory Entries not to show the Percent sign

```
[#Field: VCH Discount]
   Delete   : Format
   Add      : Format     : "NoPercent,NoZero"
```

- Reformat Discount at Batch Allocations not to show the Percent sign

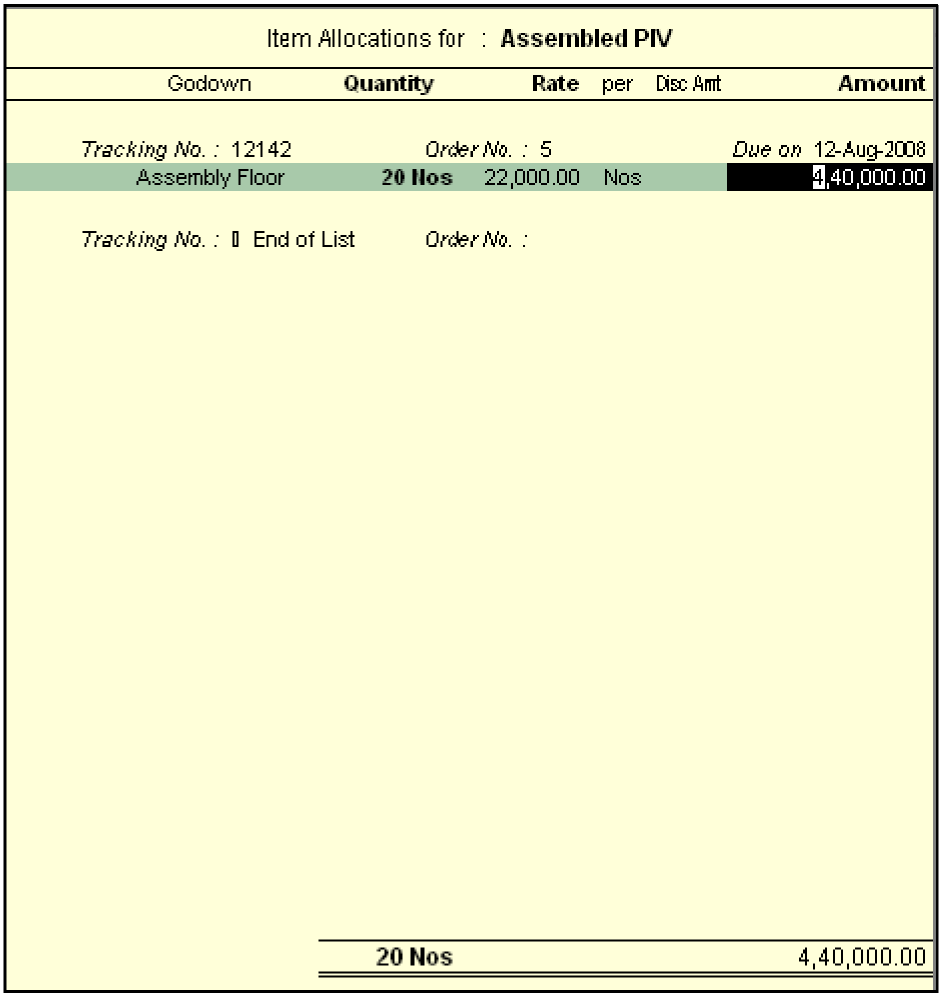
```
[#Field: VCHBATCH Discount] format
   Delete   : Format
   Add      : Format     : "NoPercent,NoZero"
```

- Change the valuation accordingly in VCH Value

```
;; To change the Invoice Value field when there are no Batch Allocations
[#Field : VCH Value]
   ResetVal: if (@@NoBaseUnits OR $$IsEmpty:$BilledQty) then $$Value +
             else (($Rate * $BilledQty) - $Discount)
```

- Change the formula by which the discount is calculated

```
;; Recalculate the following Formula rest will be taken care by Tally
[System: Formula]
   CalcedAmt    : ($Rate * $BilledQty) - $BatchDiscount
   NrmlAmount   : ($BilledQty * $Rate) - $BatchDiscount
```



Item Allocations for : **Assembled PIV**

Godown	Quantity	Rate	per	Disc Amt	Amount
Tracking No. : 12142	*Order No. :* 5				*Due on* 12-Aug-2008
Assembly Floor	**20 Nos**	22,000.00	Nos		4,40,000.00
Tracking No. : End of List	*Order No. :*				
	20 Nos				4,40,000.00

Figure 1.7 Stock Item Allocation Screen

3.2 Invoice Customisation

Invoice Customisation can broadly be classified into the following categories based on the requirement:

- Invoice Customization – User defined format
- Invoice Customization – Modifications to default format

3.2.1 Invoice Customization – User Defined Format

A totally new Invoice format needs to be developed in this category, after which it can be enabled in the following two different ways.

- Adding new format along with default format
- Replacing the existing format with new one

Adding a new format with a default format

To create a new format of invoice by modifying the existing Form Sales Color in addition to the default Print Report code.

This manner of Customisation begins with the following code snippet:

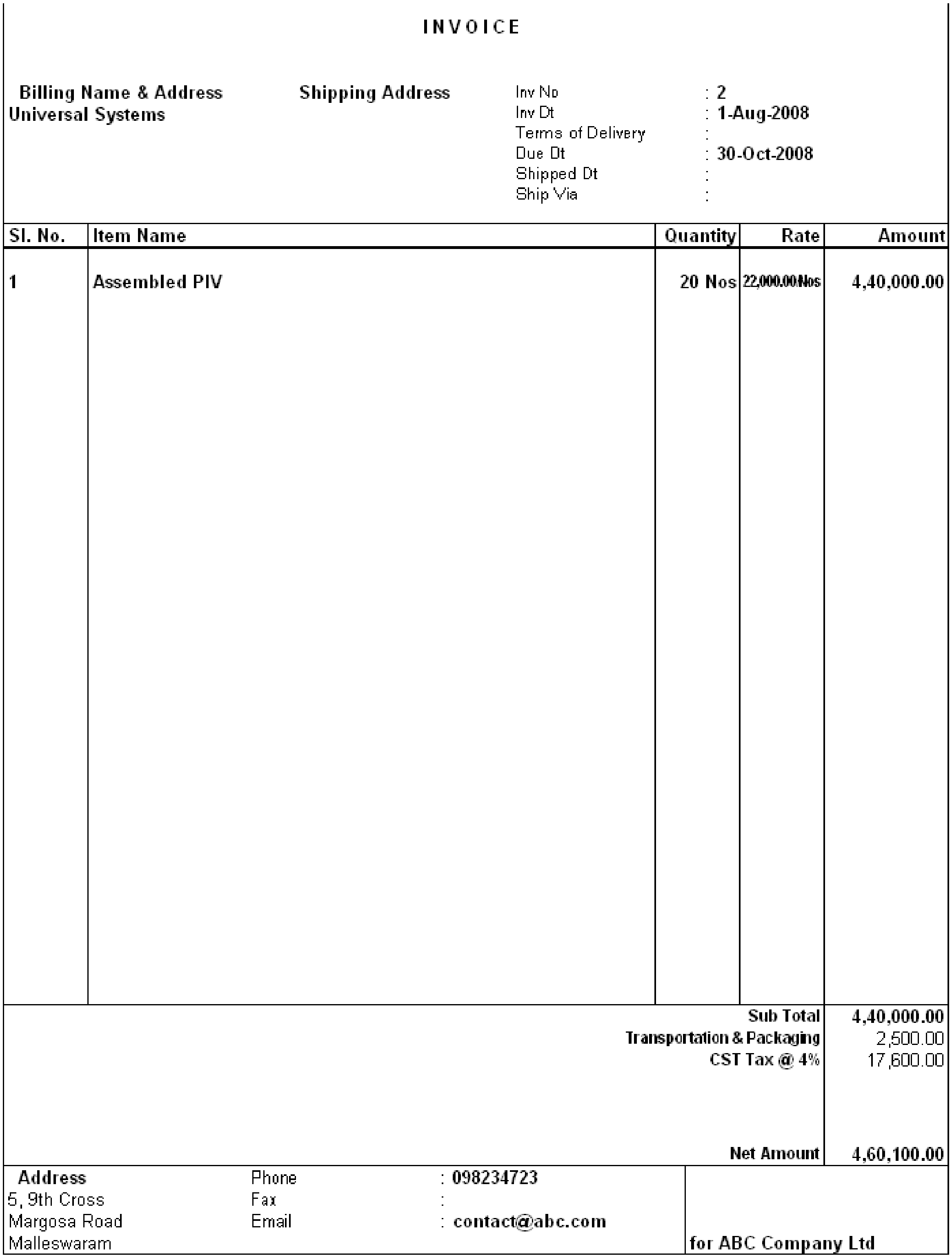
```
[#Form: Sales Color]
   Add         : Print     : Sales Invoice


[Report : Sales Invoice]
   Form        : Sales Invoice
   Object      : Voucher
```

In this code snippet, the default Print Report is deleted, the Report Sales Invoice is added and the Object Voucher is associated to it. However, in the previous example, it was not necessary to associate the Voucher Object, since it was already associated in the default Report, Printed Invoice.

Case 1

Problem Statement

ABC Company Ltd. requires a Sales Invoice which inturn requires the following format in addition to the default Sales Invoice.



INVOICE

Billing Name & Address
Universal Systems

Shipping Address

Inv No	: 2
Inv Dt	: 1-Aug-2008
Terms of Delivery	:
Due Dt	: 30-Oct-2008
Shipped Dt	:
Ship Via	:

Sl. No.	Item Name	Quantity	Rate	Amount
1	Assembled PIV	20 Nos	22,000.00 Nos	4,40,000.00

Sub Total	4,40,000.00
Transportation & Packaging	2,500.00
CST Tax @ 4%	17,600.00
Net Amount	4,60,100.00

Address
5, 9th Cross
Margosa Road
Malleswaram

Phone	: 098234723
Fax	:
Email	: contact@abc.com

for ABC Company Ltd

Figure 1.8 Invoice Customisation - Comprehensive

Replacing the existing format with the new one

By default, the basic formats provided for Commercial Invoice Printing are:

- Normal Invoice i.e. Comprehensive Invoice
- Simple Invoice i.e. Simple Printed Invoice

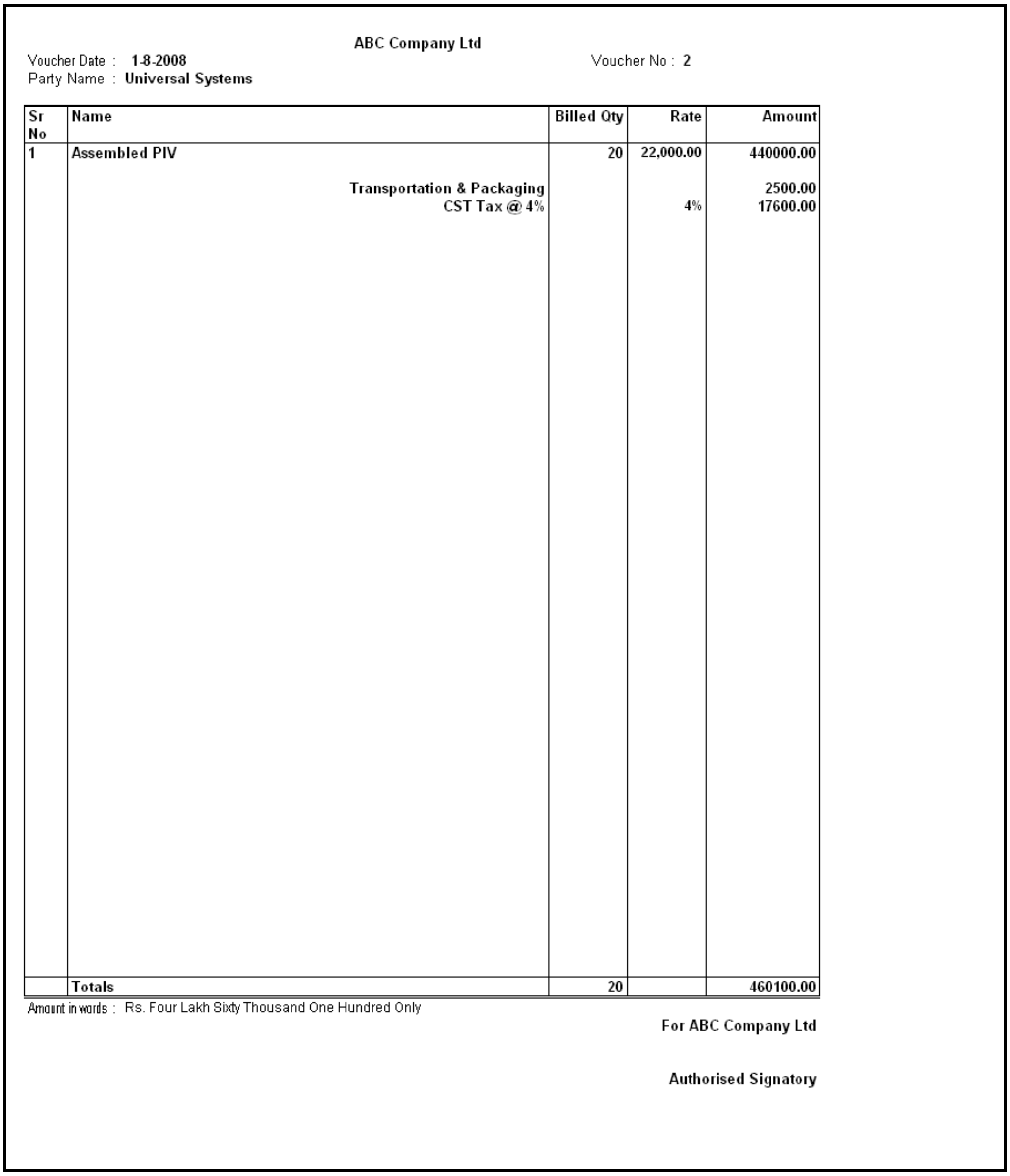
The Comprehensive Invoice and Simple Printed Invoice are two optional forms which are executed on the basis of satisfying a given condition. The default option available for print is the Comprehensive Invoice.

A Simple Invoice is printed if the option **Print in Simple Format** is set to **Yes** in F12 Configuration. On the other hand, a Comprenhensive Invoice is printed only if the user opts for a Neat Format mode of printing and the option mentioned above is set to **No**.

Case 1

Problem Statement

ABC Company Ltd. requires a Sales Invoice which inturn requires the following format for both a Normal Invoice as well as a Simple Invoice.



ABC Company Ltd

Voucher Date : 1-8-2008 Voucher No : 2

Party Name : Universal Systems

Sr No	Name	Billed Qty	Rate	Amount
1	Assembled PIV	20	22,000.00	440000.00
	Transportation & Packaging			2500.00
	CST Tax @ 4%		4%	17600.00
	Totals	20		460100.00

Amount in words : Rs. Four Lakh Sixty Thousand One Hundred Only

For ABC Company Ltd

Authorised Signatory

Figure 1.9 Invoice Customisation - Simple

Solution

Step 1

Default Forms for a Comprehensive Invoice and Simple Printed Invoice are modified with an optional Form.

```
[#Form: Comprehensive Invoice]
   Add: Option: My Invoice: @@IsSales

[#Form: Simple Printed Invoice]
   Add: Option: My Invoice: @@IsSales
```

Step 2

The Parts and Page Breaks of the default Form are deleted and new Parts are added.

To begin with, the Invoice is classified into three parts: Top Part, Body Part and Bottom Part

These Parts can be further divided into any number of Parts according to the user's requirement.

```
   [!Form: My Invoice]
      Delete       : Parts
      Delete       : Bottom Parts
      Delete       : PageBreak
      Space Top    : 0
      Space Bottom: 0
      Space Left   : 0
      Space Right : 0
      Add          : Part : My Invoice Top Part
      Add          : Part : My Invoice Body Part
      Add          : Bottom Part: My Invoice Bottom Part
```

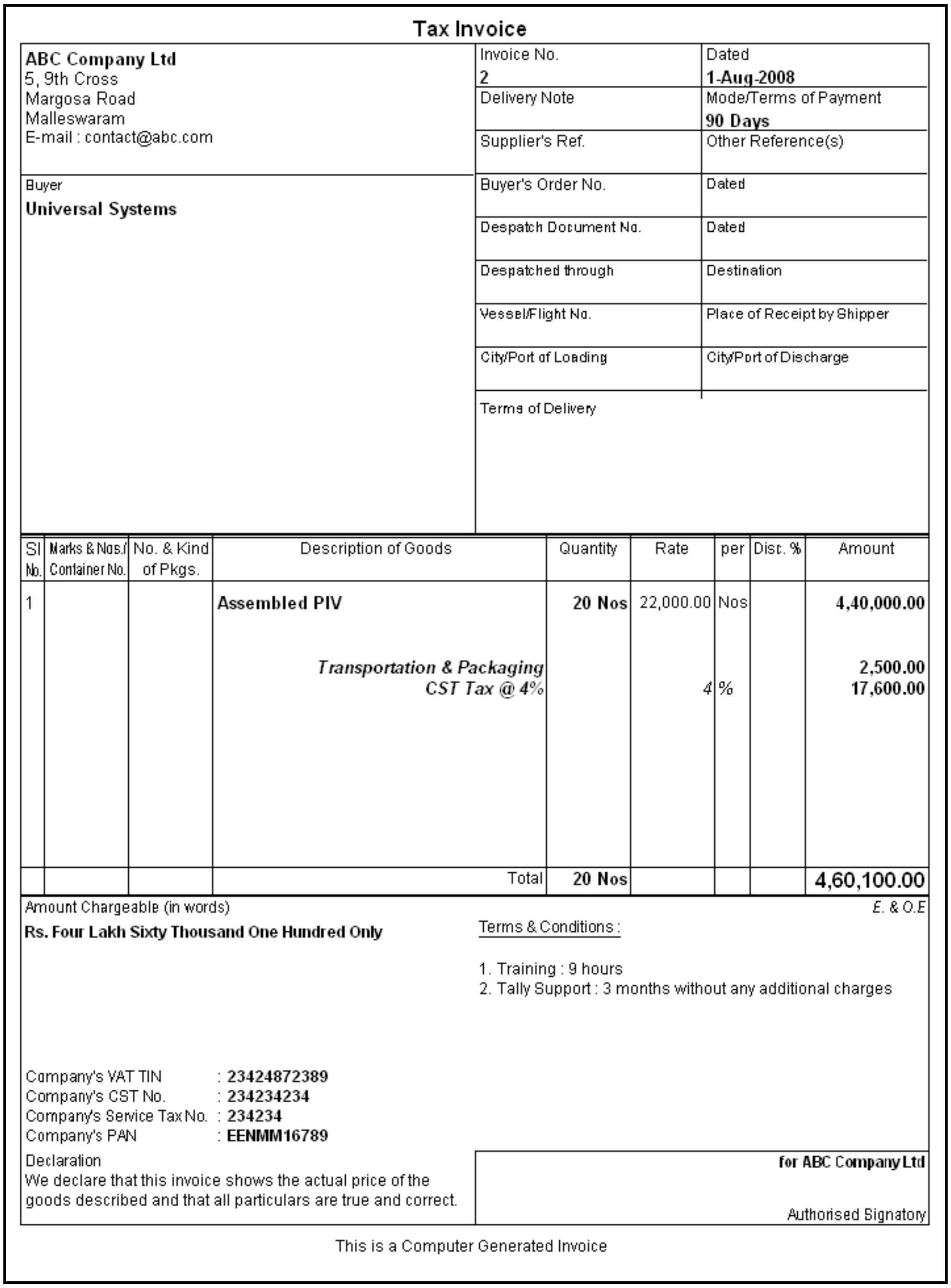
3.2.2 Invoice Customization – Modifications to default format

There may be a requirement in an Invoice customisation which is similar to the default Tally format with some minor changes. In such cases, one can just alter the default definitions as required.

Case 1

Problem Statement

A Company ABC Company Ltd. requires an Invoice with its Terms and Conditions as shown.



Tax Invoice

ABC Company Ltd
5, 9th Cross
Margosa Road
Malleswaram
E-mail : contact@abc.com

Buyer
Universal Systems

Invoice No. 2	Dated **1-Aug-2008**
Delivery Note	Mode/Terms of Payment **90 Days**
Supplier's Ref.	Other Reference(s)
Buyer's Order No.	Dated
Despatch Document No.	Dated
Despatched through	Destination
Vessel/Flight No.	Place of Receipt by Shipper
City/Port of Loading	City/Port of Discharge
Terms of Delivery	

Sl No.	Marks & Nos./ Container No.	No. & Kind of Pkgs.	Description of Goods	Quantity	Rate	per	Disc. %	Amount
1			**Assembled PIV**	**20 Nos**	22,000.00	Nos		**4,40,000.00**
			Transportation & Packaging					**2,500.00**
			CST Tax @ 4%		4	%		**17,600.00**
			Total	**20 Nos**				**4,60,100.00**

Amount Chargeable (in words)
Rs. Four Lakh Sixty Thousand One Hundred Only

E. & O.E

Terms & Conditions :

1. Training : 9 hours
2. Tally Support : 3 months without any additional charges

Company's VAT TIN	: **23424872389**
Company's CST No.	: **234234234**
Company's Service Tax No.	: **234234**
Company's PAN	: **EENMM16789**

Declaration
We declare that this invoice shows the actual price of the goods described and that all particulars are true and correct.

for ABC Company Ltd

Authorised Signatory

This is a Computer Generated Invoice

Figure 1.10 Invoice Customisation

Solution

Step 1

The default configuration Part IPCFG Right is altered to add the Line option.

```
[#Part: IPCFG Right]

   Add : Lines : GlobalWithTerms
```

Step 2

The default Part EXPINV ExciseDetails is altered to cater to the requirement.

```
[#Part: EXPINV ExciseDetails]

   Delete: Lines       : EXPINV ExciseRange, EXPINV ExciseRangeAddr, +
                         EXPINV ExciseDiv, EXPINV ExciseDivAddr, +
                         EXPINV ExciseSerial, EXPINV InvoiceTime, +
                         EXPINV RemovalTime
   Add   : Lines       : EXPINV SubTitle, EXPINV ExciseDetails
   Repeat: EXPINV ExciseDetails : Global Terms
   Local :Field: EXPINV SubTitle   : Info          : "Terms & Conditions :"
   Local :Field: EXPINV SubTitle   : Border        : Thin Bottom
   Local :Line : EXPINV SubTitle   : Space Bottom  : 1
   Invisible: NOT @@IsInvoice OR NOT ##ShowWithTerms
```

Case 2

Problem Statement

Sorting Inventory Entries as per user requirement.

Solution

The Inventory Entries of an invoice are printed in the order in which they are entered. This order can be changed as per user requirement. The sorting can be done in either the ascending or descending order in terms of the item name, stock group, stock category, units of measure, rate, value and so on. To denote the descending order, attach '—' sign to it.

To change the order of the default invoice:

- Define a Collection of inventory entries in the desired sorted order

```
[Collection : Sorted Inventory Entries]

   Type  : Inventory Entries : Voucher
   Sort  : Default : -$Parent:StockItem:$StockItemName, $StockItemName
```

- Note the Part in which the statement repeat 'Line of Inventory entries' mentioned in the DefTDL and Change this Part to 'repeat the Line with the new Collection defined'.

```
[#Part: EXPINV InvInfo]

   Repeat: EXPINV InvDetails : Sorted Inventory Entries
;; End-of-Code
```

Learning Outcome

- Vouchers are broadly classified into three types:
    - Accounting Vouchers
    - Inventory Vouchers
    - Accounting-cum-Inventory Vouchers
- Voucher Objects store two types of information, Base Information and Actual Entries.

Section II

TDL – Language Enhancements

General and Collection Enhancements

Introduction

In Tally.ERP 9, major changes have been provided by the platform to enhance the TDL capabilities which helps the programmer to develop and deploy quick and efficient solutions with ease. Major improvements have taken place in terms of language usage standardisation and performance improvements.

Breakthrough enhancements at the Collection level to provide capabilities of Remoting and Advanced Reporting. Collection is now a complete Data Processing Artifact in TDL.

This document provides in depth knowledge into the various enhancements at the attribute, modifier, method formula syntax and symbol prefixes. The foremost focus of the book is towards the enhancements at the collection level for providing the capabilities for aggregation, usage as tables, XML collection and dynamic object creation support for HTTP-XML based information interchange.

1. Attributes and Modifier Enhancements

In Tally.ERP 9 new attributes and modifiers are introduced to support the new capabilities. The behaviour of some of the exsisting attributes and modifiers is also changed.

1.1 New Attributes

New attributes that are introduced are explained in this section.

Field Attribute – Set By Condition

A new field attribute Set By Condition has been introduced. The attribute Set By Condition is similar to a conditional Set as at Field level. If multiple Set By Condition is mentioned under a Field, then the last satisfied Set By Condition will be executed.

Syntax

```
      Set By Condition : <Condition> : <Value>
```

Where,

<Condition> is any logical formula

<Value> is any string or formula.

Example:

```
[Field: Sample SetbyCondition]
   Set as          : "Default Value"
   SetbyCondition : ##Condition1 : "Set by Condition 1"
```

The Field *Sample SetbyCondition* will contain value *Set by Condition1* if the expression *Condition1* returns true else Field will contain value 'Default Value'.

Field Attribute – ToolTip

A new Field attribute ToolTip has been introduced. As the name suggests, the value specified with this attribute is displayed when the mouse pointer is placed on a particular field. This means that in addition to the static information displayed by Info or Set As attributes; we can communicate additional meaningful information with the help of this attribute. As against attributes Info or Set As, this attribute value is independent of the Field Width. In other words, when the user hovers the mouse pointer over the Field, a small hover box appears with supplementary information regarding the item being pointed over.

Syntax

```
      Tool Tip : <Value>
```

Where,

<Value> can be String or Formula

Example:

```
[Field: Led Name]
   Storage    : Name
   Tool Tip   : "Please Enter the Name of the Ledger"
```

Report Attribute – Full Screen

The new attribute Full Screen is introduced in Report definition. It helps to control the display of command window/calculator pane. It is a logical type of attribute.

Syntax

```
      Full Screen :  Yes /No
```

If this is set to Yes, command window will be hidden providing extra space when the report is displayed. The default value of this attribute is Yes. In case of the Sub-Report/AutoReport, if the value of this attribute is not specified, the default value is "No".

Example:

```
[Report : My Report]
   Full Screen : Yes
```

Part Attribute – Retain Focus

Attribute Retain Focus is added to part definition. It indicates that part should retain information about the line which is currently in focus even if the focus is moved to other part. This allows the part to make the same line as the current line when it gets back the focus.

Syntax

```
      Retain Focus : Yes / No
```

Example:

```
[Part : LedPart]
   Retain Focus : Yes
```

Part Attribute – Default Line

The attribute Default Line is used to highlight the appropriate line which satisfies the given condition. All the methods of the object associated with the line, can be used while specifying the condition.

Syntax

```
      Default Line : <Condition>
```

When the Report is invoked, the Line for which the condition is true is highlighted by default.

Example:

If the Line is repeated over collection of object Legers, then the following code will highlight the line of Cash Ledger.

```
[Part : The Main Part]
   Default Line : $Name = " Cash"
```

Collection Attribute – Sub Title

Along with the Table title, sub titles for the columns can also be given. The attribute Sub Title is introduced in Collection definition.

Syntax

```
      Sub Title : <List of Comma Separated Strings>
```

<List of Comma Separated Strings> are Strings separated by comma with respect to the number columns. Sub Title is a List type attribute.

Example:

```
[Collection: DebtorsLedTable]
   Type       : Ledger
   Child Of    : $$GroupSundryDebtors
   Format     : $Name, 15
   Format     : $OpeningBalance, 10
   Title          : $$LocaleString:"Table Sub-Titles"
   Sub Title : $$LocaleString:"Name"
   Sub Title : $$LocaleString:"Op.Balance"
```

The above code snippet displays a table with two columns. Column titles are also displayed with the help of attribute Sub Title.

Instead of specifying the Sub Title attribute multiple times a comma separated list can be given as shown.

```
   Sub Title : $$LocaleString:"Name", $$LocaleString:"Op.Balance"
```

1.2 Behavioral Changes of Attributes

Enhanced are done in the behaviour of following attributes:

Set as / Info Attributes

As of Release 2.x, the attributes Set as and Info were treated as the same attribute with aliases, when ‘Info’ is used, it had a special Skip and Prompt privilege. If both were specified the last specification would override the previous specification and would be the effective specification.

Tally.ERP 9 onwards, this behavior has been modified to treat both attributes as individual attributes. When both these attributes are specified in any field, ‘Info’ always takes the precedence and ‘Set as’ is ignored. In other words, Info carries more privilege than Set As.

The attribute Format

When the collection is a union of collections, the format object in this collection behaves as a place holder for the columns. It is mandatory to specify Format attribute in individual collection when a collection is union of collections.

Example:

```
[Collection: LedTable]
   Collection : DebtorsLedTable, CreditorsLedTable
   Format     : A, 20
   Format     : B, 25
```

Here the A, B act as dummy identifier and the second parameter is width. The collection DebtorsLedTable and CreditorsLedTable are defined as follows:

```
[Collection: DebtorsLedTable]
   Type       : Ledger
   Child Of   : $$GroupSundryDebtors
   Format     : $Name, 15
   Format     : $StateName, 15
[Collection: CreditorsLedTable]
   Type       : Ledger
   Child Of    : $$GroupSundryCreditors
   Format      : $Name, 15
   Format      : $StateName, 15
```

The above code snippet displays a table of two columns. The width of first column is 20 and second column is 25.

The attribute Sync

The behavior of the attribute Sync of Part definition is changed. The first line of next part is selected as the default of Sync attribute is now set to No. If the Part further contains parts then the value of Sync attribute specified at Parent level overrides the value specified at child level.

Example:

```
[Part : Main Part]
   Parts :SubPart1,SubPart 2
      Sync    : Yes
[Part : Sub Part 1]
   Sync    : No
[Part : Sub Part 2]
   Sync    : Yes
```

As a result of the default of Sync attribute is now set to No. In the above code snippet the Sync attribute finally has the value as Yes.

1.3 The Attribute – Child Of to support Voucher Type

Child Of attribute is enhanced further to support Voucher Type. Now with this enhancement a Collection of Vouchers of a particular Voucher Type can be constructed. Prior to this release, the same can be achieved by applying Filters to the Collection. But this approach will improve the performance.

Further the Collection attribute 'Belongs To' can be used in addition to 'Child of', to construct the Collection of Vouchers of a particular pre-defined Voucher Type including related user defined Voucher Types.

Syntax

```
      [Collection: <Coll Name>]
            Type      : Vouchers : Voucher Type
            Childof   : <String Formula>
            Belongs To: <Logical Value>
```

Where **<Coll Name>** is the name of Collection, **<String Formula>** can be a formula and should results to the name of the Voucher Type and Belongs To is an optional attribute and if used it takes **<Logical Value>** i.e. either YES or NO as value.

Example: 1

```
[Collection: Sales Vouchers]
        Type          : Voucher Type
        Child of  : $$VchTypeSales
```

The Collection ‘Sales Vouchers’ is a Collection of Vouchers whose Voucher Type is pre-defined voucher type ‘Sales’

Example: 2

```
[Collection: Sales Vouchers]
        Type                 : Voucher Type
        Child of      : $$VchTypeSales
        Belongs To    : Yes
```

The Collection ‘Sales Vouchers’ is a Collection of Vouchers whose Voucher Type is pre-defined voucher type ‘Sales’ or any other user defined Voucher Type whose ‘Type of Voucher’ is ‘Sales’.

1.4 Attribute Modifiers

In TDL, attribute modifiers are classified as Static/Load time or Dynamic/Run-Time modifiers. Use, Add, Delete, Replace/Change are Static/Load Time modifier. Option, Switch and Local are Run-Time modifiers. The sequence of evaluation is generalized accross all the definitions in TDL.

Sequence of Attribute Evaluation:

1. Use
2. Normal Attributes
3. Delayed Static/Load Time modifier
4. Dynamic/Run-Time modifier

1.5 A New Attribute Modifier – Switch

A new attribute modifier ‘Switch’ has been incorporated from Tally.ERP 9 onwards. This attribute is similar to the Option attribute but reduces code complexity and improves the performance.

The function Option compulsorily evaluates the conditions for all the options provided in the description code and applies all which satisfy the evaluation conditions. This means that it is not always easy to write the code where you just want one of the options to be applied, you have to make sure that other options are not applied using a negative condition. The new attribute provider Switch has been provided to support these types of scenarios where evaluation is carried out only up to the point where the first evaluation process has been cleared.

Apart from this, the Switch can be grouped using a label. Therefore, multiple switch groups can be created and zero or one of the switch cases would be applied from each such group.

Apart from this, the switch can be grouped using a label, as shown below:

Syntax

```
        Switch : <Label> : <Definition Name> : <Condition>
        Switch : <Label> : <Definition Name> : <Condition>
```

If multiple Switches are mentioned within a single definition, then evaluation will be carried out up to the point where first condition is satisfied for the given label.

Example: 1

```
[Field: Sample Switch]
   Set as : "Default Value"
   Switch : Case1 : Sample Switch1 : ##SampleSwitch1
   Switch : Case1 : Sample Switch2 : ##SampleSwitch2
```

In the above code snippet, multiple switch statements having the same label, zero or one statement will be executed.

Example: 2

```
[Field: Sample Switch]
   Set as : "Default Value"
;; If none of the condition is TRUE then Field will have Default Value

   Switch : Case1 : Sample Switch1 : ##SampleSwitch1
   Switch : Case1 : Sample Switch2 : ##SampleSwitch2
   Switch : Case2 : Sample Switch3 : ##SampleSwitch3
   Switch : Case2 : Sample Switch4 : ##SampleSwitch4
```

In the above code snippet, multiple switch groups are created and zero or one of the switch cases would be applied from each such group or label.

1.6 Behavioral Changes for Attribute Modifiers

The behavior of following attributes is enhanced.

Changed precedence of “Use”

The behavior of attribute USE which is used to inherit the properties from other definition has now changed. Irrespective of their order of specification within a definition, USE will be evaluated first. In other words, the order in which USE is specified is immaterial as in any case it will be evaluated first. If multiple USE attributes are specified in a single definition, they are evaluated in the order of their occurrence.

Example:

```
[Field: Attr Use1]
   Set as  : "This shows the changed behavior of 'Use' attribute"
   Style   : Large Bold
   Use     : Name Field
```

The Field *Attr Use1* uses existing Field *Name Field*. Since Use is having higher precedence over other attributes, Field *Attr Use1* will inherit all the attributes of *Name Field*. But the style *Large Bold* at the Field *Attr Use1* will override the inherited Style within the Field *Name Field*.

Changed behaviour of Delayed Attribute Modifiers “Add/Delete/Replace”

Static/Load Time modifiers like Add, Delete and Replace can be called as Delayed Attribute modifiers, as they are having least precedence among Delayed Static/Load Time modifiers.

Now these modifiers are generalized across all definitions. Earlier for definitions *Report, Key, Color*, *Style*, *Border* and *Variable*, the delayed attributes were applied as their sequence of appearance in the definition description. If more than one delayed attribute is used under any definition, then attributes will be applied as they appear. This has been done to bring consistency across the definitions.

Example:

```
[Report: Test Report]

   Form            : Form1

   Delete     : Form

   Form       : Form2
```

As a result of the above code snippet, the Report *Test Report* will not have any Form as Delete is evaluated last which deletes all the existing forms.

Example:

```
[Report: Test Report1]

   Form            : Form1

   Delete     : Form

   Add              : Form    : Form2
```

As a result of the above code snippet, the Report *Test Report1* will have one Form *Form2* since on deletion of all the Forms, Delayed attribute modifier *Add* is used to add a new Form *Form2*.

Enhanced Syntax of Delayed Attribute “Local”

Delayed attribute modifier Local which is used to locally modify the attributes of any child definition is now enhanced to accept nested Locals.

Syntax

```
      Local : <DefinitionType1> : <DefinitionName1> [: <DefinitionType2> +
              : <Definition Name2> : ... ] : <Attribute> : <Value>
```

Where,

<Definition Type> can be a Form, a Part, a Line or a Field,

<Definition name> is the name of the definition type,

<Attribute> is the attribute of the Definition of which value needs to be altered, and

<Value> is the value assigned to this attribute within the current Report or Form or Part or Line.

Example:

```
[Report: Custom Report]
   Local  : Line : TitleLine : Local : Field : AmtField : +
                                 Set as   : "Sales  Amount"
```

The Field *Amt Field* is localized at the Report *Custom Report*, by using nested locals.

1.7 Behavioral change in System Definitions

System Definitions overriding without '#' are treated as warnings now instead of errors. #, ! or * modifications to [System : MenuKeys], [System :Form Keys], [System :Formula] and [System :UDF] were shown as errors. They are now converted to warnings.

In Tally.ERP 9, overriding System Formula / Variable without prefixing a # have been treated as an Error.The usage of #, * and ! prefix to System Definitions like Menu Keys, Form Keys and UDF were not allowed and treated as errors.

Many existing Codes have stopped working due to this behavioral change. Hence in order to maintain backward compatibility, these have been enabled & treated as warnings and in some cases ignored so that all existing TDL Codes will still continue to work without any changes required for the same.

These warnings are thrown only by the compiler during the compilation using Tally Developer.9

However, it is recommended to use # for existing System Formula alteration and refrain from using # for System Menu Keys, Form Keys and UDF Definition or using ! for any system descriptions.

1.8 Partial Attribute Support

Prior to Tally.ERP 9, all descriptions supported partial search on their attribute words, for e.g., Set as could have been written as Set a, Set or Se, which would allow minimum number of characters to be present to an extent where another attribute does not start with those characters. This behavior is now removed as it is not practical to use partial words. But multiple aliases are now supported to allow meaningful attribute names.

Example:

- Set as can be written as Set
- Float Bottom Lines at Part Definition can be written as Float
- Top Part can be written as Part, Parts or Top Parts

Since these aliases have been introduced, most of the existing TDL will work without any changes. In case of Partial words/ Non-meaningful words used in any TDL, Tally would throw an error which needs to be corrected in TDL.

Change in usage of 'BLANK' Keyword in Menu Items

To insert empty line between Menu Items, BLANK keyword was used. Also Item Attribute without any "Value" used to be considered as BLANK prior to Tally.ERP 9. For consistency in TDL coding, the later is now disallowed. Only BLANK keyword can now be used to indicate an empty Menu Item.

2. Enhanced Special Symbols

In Tally.ERP 9 some new symbols are introduced and the behaviour of the definition modifier '#' is enhanced.

2.1 Multi – line commenting in TDL source code using /* and */

Multi-line commenting is a new feature in this release which renders the TDL code more user-friendly and easy to maintain. A simple Multi-line comment would look like:

```
/*
<Comment Line 1>
<Comment Line 2>
*/
```

2.2 Extension of modifying definitions using

The scope of modifying definitions using # is extended to System Formula Definition. Now, you can alter the value of the existing system formula using #. This helps to improve the performance with optimized formulae.

Example:

```
[#System: Formula]
   NameWidth        : 40
   MaxNameWidth     : 60
```

The above code alters the existing System Formula to change the values specified to Formulae *Name Width* and *Max Name Width* in DefTDL

2.3 '*' (Reinitialize) Definition modifier

The definition modifier "*" overwrites the existing content of definition. The "*" modifier is very useful when there is a need to completely replace the existing definition content with a new code.

Syntax

```
      [*<Definition Type> : <Definition Name>]
```

Example:

```
[Field: Sample ReInitialize]
   Info    : "Original Value"
   Style   : Large Bold
   Color   : Blue
```

```
[*Field: Sample ReInitialize]
   Info   :"ReInitialized-All the attribute values deleted +
            & newly  defined"
   Lines  : 1
```

3. Method Formula Syntax with Relative Object Specification

The '$' operator has been enhanced with a new capabilities. This allows direct access to any object method including its sub-collections to any level with a dotted notation framework. Using the new capability, there is no need to repeat a line over a sub-collection to access it. The values from any object anywhere can be accessed without making the object as the current object in context. Suffixing of PrimaryObjType:ObjNameFormula is still supported for backward compatibility. In cases where both are specified; the enhanced new primary object specification will be considered.

The earlier syntax to access a Method was:

```
      $MethodName OR $MethodName:PrimaryObjType:ObjNameFormula
```

The enhanced method formula Syntax is introduced to support access out of the scope Primary Object and to access Sub object at any level using (.) dotted notation with index and condition support.

The enhanced syntax is:

```
      $<PrimaryObjectSpec>.<SubObjectPathSpec>.MethodName
```

Where,

<PrimaryObjectSpec> can be (<Primary Object Type Keyword>, <Primary Object Identifier Formula>)

<SubObjectPathSpec> is given as CollectionName [<Index Formula>, [<Condition>]]

<MethodName> refers to the name of the method in the specified path.

<Index Formula> should return a number which acts as a position specifier in the Collection of Objects satisfying the given <condition>.

Example: Following are evaluated assuming Voucher as the current object

1. To get the Ledger Name of the first Ledger Entry from the current Voucher,

```
    Set as          : $LedgerEntries[1].LedgerName
```

2. To get the amount of the first Ledger Entry on the Ledger Sales from the current voucher,

```
    Set as          : $LedgerEntries[1,@LedgerCondition].Amount
    LedgerCondition : $LedgerName = "Sales"
```

3. To get the first Bill Name of the first Ledger entry on the Party Ledger from the current voucher,

```
   Set As           : $LedgerEntries[1,@@LedgerCondition]+
                      .BillAllocations[1].Name
   LedgerCondition  : $LedgerName = @@InvPartyName
```

4. To get the OpeningBalance of the first Bill for the Party, Acme Corp,

```
Set As           :  $(Ledger,@@PartyLedger).BillAllocations[1]+
                    .OpeningBalance
PartyLedger      : "Acme Corp"
```

Primary Object specification is optional. If not specified, current object will be considered as primary object. Sub-Collection specification is optional. If not specified, methods from the current or specified primary object will be available. Index specifies the position of the Sub-Object to be picked up from the Sub-Collection. Condition is the filter which is checked on the objects of the specified Sub-Collection.

<Primary Object Identifier Formula>, **<Index Formula>** and **Condition** can be a value or formula.<*Index Formula*> can be any formula evaluating to a number. Positive Number indicates a forward search and negative number indicates backward search. This can also be a keyword First or Last which is equivalent to specifying 1 or -1 respectively.

If both Index and Condition is specified, the index is applicable on the Object(s) which satisfy the condition so one gets the nth Object which clears the condition. Let's say for example, if the Index specified is 2 and Condition is Name = "Sales", then the second object which matches the name Sales will be picked up.

Primary Object Path Specification can either be relative or absolute. Relative Path is referred using empty parenthesis () or Dotted path to refer to the Parent object relatively. SINGLE DOT denotes the current object, DOUBLE DOT the Parent Object, TRIPLE DOT the Grand Parent Object and so on within an Internal Object. Absolute Path refers to the path in which the Primary Object is explicitly specified.

To access the Methods of Primary Object using Relative Path following syntax is used:

```
$().MethodName or $..MethodName or $… MethodName
```

Example:

Being in the context of LedgerEntries Object within Voucher Object, the following has to be written to access the Date from its Parent Object which is the Voucher Object.

```
$..Date
```

To access the Methods of Primary Object using Absolute Path :

```
$(Ledger, "Cash").OpeningBalance
```

4. Enhancements – Object Association

In TDL, Interface object exists in context of any data object. Every Interface object needs to be associated with some data object. In the absence of any explicit object association, Interface object will get associated with Anonymous object. TDL programmer can explicitly associate Interface objects like Report, Part, Line and Field with data object. In Tally.ERP 9 Object association become more natural and simpler.

4.1 Report Level Object Association

A Report normally will be associated with a data object, which it gets from the previous Report or will be associated with anonymous object. From Tally.ERP 9 onwards the syntax for association has been enhanced to override the default association as well. The Report attribute 'Object' has been enhanced to take an additional optional value 'ObjectIdentifierFormula'.

Syntax

```
      Object: <ObjectType> [: <ObjectIdentifierFormula>]
```

Where,

<ObjectType> is type of any Primary Object and

<ObjectIdentifierFormula> is an optional value and is any formula which evaluates to name of Primary Object.

Example: 1 – Prior to Tally.ERP 9

```
[#Form: Sales Color]
   Delete    : Print
   Add          : Print: New Sales Format

[Report: New Sales Format]
   Object    : Voucher
```

Default Sales color Form is modified to have new print format 'New Sales Format'. This Report gets the voucher object from previous Report.

Example: 2 – In Tally.ERP 9

```
[Report: Sample Report]
   Object    : Ledger: "Cash"
```

Ledger 'Cash' is associated to the Report 'Sample Report'. Now components of 'Sample Report' by default inherit this ledger object association.

4.2 Part Level Object Association

By default, Part inherits the Object from the Report/Part/Line. This can be overridden by two ways.

Using 'Object' attribute specification in Part definition.

Syntax: Prior to Tally.ERP 9

```
       Object : <SupplierCollection> : <SeekTypeKeyword> [: <SeekCondition>]
```

Where,

<SupplierCollection> is the name of the Collection of secondary Objects,

<SeekTypeKeyword> can be First or Last which denotes the position index and

<SeekCondition> is an optional value and is a filter condition to the supplier collection.

Example: Part in the context of Voucher Object

```
[Part: Sample Part]

   Line          : Sample Line

   Object           :InventoryEntries:First:@@StkNameFilter

   Scroll           : Vertical

[System: Formula]

   StkNameFilter: $StockItemName = "Tally Developer”
```

The first inventory entry which has stock Item “Tally Developer” is associated with Part ‘Sample Part’.

Using ‘Object Ex’ attribute specification in Part definition

From Tally.ERP 9 onwards, data object can be associated to Part by using new attribute Object Ex. Now even Primary Object can also be associated to a Part, which was not possible in the earlier Part level data object association. Also data Object associated to some other Interface Object can also be associated to a Part. This aspect will be elaborated in the section “Object Access via Interface Object”.

Syntax: In Tally.ERP 9

```
      Object Ex: <Method Formula Syntax>
```

Where,

<Method formula syntax> is, <Absolute Spec>.[<SubObjectSpec>]

Absolute Specification is

(<Object Type>, <Object Identifier Formula>), If only Absolute Spec is given then it should end with dot (‘.’).

Sub Object Specification is CollectionName[Index,<Condition>]

Example: 1

```
[Part: Sample Part]

   Object Ex : (Ledger, "Customer 1").
```

Ledger object “Customer 1” is associated to the Part ‘Sample Part’. Since only absolute specification used, the Object specification is ends with ‘.’.

Example: 2

```
[Part: Sample Part]

   Object Ex : (Ledger,"Customer").BillAllocations[1,@@Condition1]

[System: Formula]

   Condition1: $Name = "Bills 2"
```

Secondary Object ‘Bill Allocations’ is associated with Part ‘Sample Part’.

4.3 Line Level Object Association

An object can associated to a Line by Part attribute “Repeat’. Now Part attribute ‘Repeat’ is enhanced to support the following.

a. Extraction of collection from any Data object

Extraction of collection from Interface Object associated Data object. This aspect will be elaborated in the section “Object Access via Interface Object”.

Repeat Syntax: Prior to Tally.ERP 9

```
Repeat:<Line Name>: <Coll Name>: [<Supplier Coll>+
       :<SeekTypeKeyword>:<SeekCondition>]
```

Where,

<Coll Name> is the name of the Collection and if that Collection is present in the one level down of the object hierarchy then Supplier Collection needs to be mentioned.

<SupplierCollection> is the name of the Collection of secondary Objects,

<SeekTypeKeyword> can be First or Last which denotes the position index and

<SeekCondition> is an optional value and is a filter condition to the supplier collection.

Example: Part in the context of Voucher Object

```
[Part: Sample Part]
   Line       : Sample Line
   Repeat     :Sample Line:Bill Allocations:Ledger Entries:First:+
               @@LedFormula
[System: Formula]
   LedFormula : $LedgerName = “Customer”
```

The Line ‘Sample Line’ is repeated over Bill Allocations of first object ledger entries which satisfies the given condition

In Tally.ERP 9 – Repeat Syntax

```
Repeat: Line Name: MethodFormulaSyntax[:SupplierCollection: +
        SeekTypeKeyword: SeekCondition]
```

Where,

<MethodFormulaSyntax> is <Absolute Spec>.<SubObjectSpec>

<Absolute Spec> is (<Object Type>, <Object Identifier Formula>)

<Sub Object Spec> is CollectionName[Index,<Condition>]

and Supplier Collection syntax is provided just for the backward compatibility.

Example:

```
[Part: Sample Part]
   Line     : Sample Line
   Repeat   : Sample Line: (Ledger, “Customer”).BillAllocations
```

1.1 Field Level Object Association

By default inherits from the parent line or Field (if field inside a field). This cannot be overridden. However Field also allows Object specification syntax. This association if specified acts as the 'Secondary Context Object' for the Field. During any formula evaluation, if the formula / method fail in the context of primary object, secondary object is tried then.

2. Enhancements – Object Access via Interface Object

From Tally.ERP 9 onwards, data objects in association with Interface objects can be accessed using the new Interface object access syntax. Data object which is associated to Interface Object can be accessed with the following 2 step procedure.

1. Identifying Part and Line Interface object with 'Access Name'
2. Value/Collection Extraction

2.1 Identifying Part and Line Interface object with 'Access Name'

Part and Line can be identified by unique access name. For this purpose a new attribute 'Access Name' is introduced for Part and Line definition.

Syntax

```
      Access Name: Access Name Formula
```

Where,

<Access Name Formula> can be a formula and evaluated to string.

Example: 1 – Access Name at Part Definition

```
[Part: Sample Part]
   Line                  : Sample Line1
   Access Name    : "Sample Part"
```

Example: 2 – Access Name at Line Definition

```
[Line: Sample Line]
   Field          : Sample Fld1, Sample Fld2
   Access Name: "Repeated Line"  + $$String:$$Line
```

When Line 'Sample Line' is repeated over a collection, every Line is identified by unique Access Name.

2.2 Value Extraction

Once Part and Line Interface objects are able to uniquely identify by 'Access Name', then data object can be accessed by either new function $$ObjectOf or by 'New method formula syntax'.

Value Extraction by function $$ObjectOf

Methods of data object which is associated to Interface Object can be extracted by using the function $$ObjectOf.

Syntax

```
        $$ObjectOf : <DefinitionType> : <AccessNameFormula> : <EvaluationFormula>
```

Where,

<DefinitionType> may be Part or Line,

<AccessNameFormula> is a string through which a Part or Line can be uniquely identified, and

<EvaluationFormula> is a method that need to evaluated.

Example:

Line ‘Sample Line’ has Access Name as ‘Sample Acc Name’ and in association with Ledger Object.

```
[Field : Sample Field]
   Set as : $$Objectof:Line:”Sample Acc Name”:$Name
```

Field ‘Sample Field’ displays the name of the object Ledger which is associated with a Line whose access name is “Sample Line Acc Name”.

Value Extraction by using new method formula

Methods of data object which is associated to Interface Object can also be extracted by using new method formula. With this approach sub object’s methods can be extracted.

Example:

Line ‘Sample Line’ has Access Name as ‘Sample Acc Name’ and in association with Ledger Object.

```
[Field: Sample Field]
   Set as : $(Line,“MyLineAccessName”).BillAllocations[1].OpeningBalance
```

Field ‘Sample Field’ displays the name opening balance of a ledger which is associated with a Line whose access name is “Sample Line Acc Name”.

Repeat Syntax Using Access Name

Collection inside the data object which is associated to Interface Object can be extracted by using new method formula.

Syntax - Enhanced Repeat

```
        Repeat: Line Name: MethodFormulaSyntax[:SupplierCollection: +
                SeekTypeKeyword: SeekCondition]
```

Where,

<MethodFormulaSyntax> is <Absolute Spec>.<SubObjectSpec>

Absolute Spec is (<Object Type>, <Object Identifier Formula>)

Sub Object Spec is CollectionName[Index,<Condition>]

and Supplier Collection syntax is provided just for the backward compatibility.

Example:

Part having access name ‘MyPartAccessName’ is under the context of Voucher Object

We can repeat a line “Sample Line” over Inventory Enries of the Voucher Object which is associated with Part having the access name “MyPartAccessName”

```
[Part: Sample Part]

   Repeat   : Sample Line:(Part, “MyPartAccessName”).InventoryEntries
```

3. Bracket support in TDL

Prior to Tally.ERP 9, the usage of TDL language token bracket (‘(’ and ‘)’) was restricted as mathematical operator only. From this release onwards brackets can be used in the following scenarios.

1. During the function call to enclose the function parameter
2. In the language syntax for nesting formulas
3. As a Mathematical Operator

3.1 During the Function Call

Prior to Tally.ERP 9, when a parameter for function requires expression and that expression contains any language token then the TDL programmer is forced to replace the expression by a formula. This can now be achieved by enclosing the expression in a bracket. The expression inside the bracket is evaluated first and the result is used as the parameter for the function. Nesting can be performed up to any level.

Brackets can also be used in the places where function parameter expects identifier or constant value.

Example: 1

The Field ‘Sample Fld’ displays the first 5 characters of the currently loaded Company’s email address.

Prior to Tally.ERP 9

In this case, First parameter to the function ‘StringPart’ is a expression contains language token ‘:’. So a formula needs to be created.

```
[Field: Sample Fld]

   Set As                        : $$StringPart:@CmpEmailAddress:0:5

   CmpEmailAddress      : $Email:Company:##SVCurrentCompany
```

In Tally.ERP 9

```
[Field: Sample Fld]

   Set As         :$$StringPart:($Email:Company:##SVCurrentCompany):0:5
```

Example: 2

If the last object in the collection Ledger is a Sundry Creditor then the Field Sample Fld will have logical value Yes else No.

Prior to Tally.ERP 9

In this case, the condition contains language token ':' and constant value '-1'. So formulas needs to be created.

```
[Field: Sample Fld]
   Set as       :$$CollectionField:@@GroupCheck:@@IndexPosition:Ledger

[System: Formula]
   GroupCheck       : $Parent:Ledger:$Name = $$GroupSundryCreditors
   IndexPosition  : -1
```

In Tally.ERP 9

```
[Field: Sample Fld]
   Set As       : $$CollectionField:($Parent:Ledger:$Name = +
                           $$GroupSundryCreditors):(-1):(Ledger)
```

3.2 In the language syntax for nesting formulas

Prior to Tally.ERP 9, whenever an expression is a part of language syntax then language tokens are not permitted. This restriction leads to the necessity of additional formulas even when the formulas are not used more than once. With this enhancement, expressions can be used in language syntax by enclosing them in brackets.

Brackets can also be used in the places where attribute value expects identifier or constant value.

Example: 1

If the given condition is satisfied then the Field 'Sample Fld', will display "Cash Accounts"

Prior to Tally.ERP 9

In this case, the condition contains language token ':'. So a formula needs to be created.

```
[Field: Sample Fld]
    Set By Condition: @IsLedgerIsCash : "Cash Accounts"
    IsLedgerIsCash       : ($Name:Ledger:##SVLedger) = "Cash"
```

In Tally.ERP 9

```
[Field: Sample Fld]
    Set By Condition    :($Name:Ledger:##SVLedger)= "Cash" : "Cash Accounts"
```

Example: 2

First parameter for repeat attribute is using bracket for identifier.

```
[Part: Sample Part]
   Line          : Sample Line
   Repeat      : (Sample Line) : My Collection
```

As a Mathematical Operator

In TDL, brackets are used as mathematical operator to set the precedence of evaluation.

Example:

If parentheses are not used then Field 'Sample Fld' will display 26 otherwise 44.

```
[Field: Sample Fld]
   Set As      : 4 * (5 + 6)
```

4. Action Enhancements

Some of the existing actions are enhanced to support the multiline selection capabilities. Several new actions are also introduced in TDL.

4.1 Enhancements in Key Actions

Key action is enhanced to perform various operations on multiple lines. For example, multiple vouchers can be selected/ unselected and various actions such as deletion, modification, etc. can be performed on the selected vouchers only. To achieve this, two attributes Scope and Selectable are introduced. Scope attribute is introduced in Key definition and Selectable attribute is available at Part and Line definition.

The attribute Scope

In Key definition, a new attribute 'Scope' is introduced, through which scope for the Action(s) can be specified.

Syntax

```
      Scope : <Scope Keyword>
```

<Scope Keyword> can have any of the following possible values: - *Current Line/ Line, All Lines, Selected Lines, Unselected Lines and Report*

The attribute Selectable

Selectable attribute can be applied to Part and Line definition.

Part Definition

At Part level, the attribute 'Selectable' indicates that the lines owned by this Part are selectable or not and the default value for the same is Yes.

Syntax

```
      Selectable : <Logical Formula>
```

Line Definition

At Line level, the attribute ‘Selectable’ indicates that the line (or lines within this line) is selectable or not, The default value of attribute Selectable for repeated lines is ‘Yes’ and for non-repeated lines it is ‘No’. The value is also inherited from Parent Part/Line and the same can be overridden at Line level.

Syntax

```
      Selectable : <Logical Formula>
```

<Logical Formula> must return the value as Yes or No

Following actions have been introduced/ changed.

- Toggle Select – Selects / deselects a line
- Select All – Selects all the lines within a part
- Unselect All – Deselects all the lines within a part
- Invert Selection – Selects all the unselected lines within a part
- Modify Object – Modifies the values stored in the methods of an Object

The behaviour of existing actions Cancel Object, Delete Object, Remove Line and Multi Field Set have been modified to obey the scope specified in the Key description.

The actions Print Report, Upload Report, Email Report and Export Report can be executed now on the Selected Line scope. In the resultant report, the selected lines will be available as objects in the collection “Parameter Collection”. This collection can be used in the called report for displaying data.

Actions like Cancel Object, Audit Object and Delete Object are enhanced to work with Report scope.

4.2 New Actions

Following new actions are introduced in the language:

Modify Object

Modify Object is enhanced to alter a method of an object at any level. Modify Object action also supports modifying multiple values of an object. Multiple values can be specified as a comma separated list of <Method Name> : <Value> pairs.

Syntax

```
      Action : Modify Object : <PrimaryObjectSpec>.<SubObjectPathSpec> +
               .MethodName : value>[,Method Name: <value> , …]+
               [,<SubObjectPathSpec>.MethodName :<value>, …..]
```

The specifications given for <PrimaryObjectSpec>, <SubObjectPathSpec>, MethodName remain same as described in the **New Method syntax section**.

A single Modify Object action cannot modify Multiple Objects, but can modify multiple values of an Object.

Modify Object is allowed to have Primary Object Specification only once i.e., for the first value. Further values permissible are optional Sub Objects and Method Specification only.

Sub Object Specification is optional for second value and onwards. If Sub Object Specification is specified, the context is assumed to be the Primary Object specified for the first value. In absence of Sub Object Specification, the previous value specification's Leaf Object is considered as the context.

Example: 1

```
[Key: Alter My Object]
   Key                  : Ctrl + Z
   Action         : Modify Object:(Ledger,"MyLedger").BillAllocations +
                    [First,$Name="MyBill"].OpeningBalance +
                    : 100,..Address[Last].Address : "Bangalore"
```

Existing ledger *My Ledger* is being altered with new value for opening balance of existing bill bearing Name as *MyBill* and last line of Address. Key *Alter My Object* can be attached to any Menu or Form clicking which the above will be altered.

Example: 2

```
[Key: Alter My Object]
   Key      : Ctrl + Z
   Action   :     Modify Object :(Ledger,"MyLedger").BillAllocations[1]+
            .OpeningBalance : 1000, Name:"My New Bill",..Address[First]+
            .Address : "Hongasandra Bangalore", +
            Opening Balance : 5000
```

Existing ledger *My Ledger* is altered with new values for opening balance for existing bill, opening balance of ledger and address. Key *Alter My Object* can be attached to any Menu or Form.

Example: 3

```
[Key: Alter My Object]
   Key       : Ctrl + Z
   Action    : ModifyObject : LedgerEntries[1].BillAllocations[1].Name : +
            "Test1", Amount :"1000.00", ..BillAllocations[2].+
            Name : "Test2", Amount : "2000.00", ().Date : "1.4.08"
```

In a Voucher context, Key *Alter My Object*, alters Name, Amount and Date methods of Sub object Bill Allocation in one line.

Action Modify Object in a Menu Definition

In Menu definition, a button which has action Modify Object can be added.

Example:

```
[#Menu: Gateway of Tally]

   Add        : Button : Alter My Object
```

While associating a key with action Modify Object, following points should be considered:

- Since menu does not have any Data Objects in context, specifying Primary Object becomes mandatory.
- Since Menu cannot work on scopes like Selected Lines, Unselected Lines, etc., scopes specified are ignored.
- Any formula specified in the value is evaluated assumes Menu Object as requestor.
- Even Method values pertaining to Company Objects can be modified.
- A button can be added at the menu to specify the action Modify Object at the Menu level

Action – Set Object Values

The new action introduced is similar to the action modify object. The action Set Object values work only in the Edit mode of a Report as it uses current context. This action changes the values of the object from current context as specified.

Syntax

```
      Action: Set Object Values: <SubObjectPathSpec>.<Method Name> +
              : <Method Value>
```

Where,

<SubObjectPathSpec> is given as CollectionName [<Index Formula>, [<Condition>]]

<MethodName> refers to the name of the method in the specified path and

<Method Value> is the value to be set for the <Method Name>.

This action alters the current object in memory. When the Primary object is saved the changes will be updated in Tally database.

Example:

```
[Key : My Key]

   Action :Set Object Values : Opening Balance : ($$AsAmount : 10)
```

Action – Backup Company

Backup Company action allows to take the backup of multiple companies.

Syntax

```
      Backup Company : <parameter sep char> : <String Formula>
```

Where

<Parameter Sep Char> is a character used to separate parameter.

<String Formula> must evaluate to the value in the following order seperated by the **<Parameter Sep Char>**:

```
<Destination> <Source> <Company Name> <Company Number>
```

<Destination> is the path where the backup file is to be stored.

<Source> is the path from where the company data is to be taken for backup

<Company Name> is name of the Company

<Company Number> is number of the company

These four values must be specified for each company. These can be repeated for multiple companies

.

Example: Single Company

```
[Button : My Cmp Bk Button]
   Title     : BackUp Cmp
   Action    : BackUp Company: "," :"C:\,C:\Tally.ERP 9\Data,Global +
               Enterprises,10037"
   Key       : Alt + G
```

Example: Multiple Company

```
[Button : My Cmp Bk Button]
   Title     : BackUp Cmp
   Action    : BackUp Company : "," : "C:\,C:\Tally.ERP 9\Data,+
               Global Enterprises,10037,C:\,C:\Tally.ERP 9\Data,+
               TDL Demo,10027"
```

OR

```
[Button : My Cmp Bk Button]
   Title     : BackUp Cmp
   Action    : BackUp Company : "," : @@MyCmpFor
[System :Formula]
   MyCmpFor  : "C:\,C:\Tally.ERP 9\Data,Global Enterprises,10037, +
               C:\,C:\Tally.ERP 9\Data,TDL Demo,10027"
```

Action – Restore Company

This action allows to restore multiple companies in one go.

Syntax

```
Restore Company : <parameter sep char> : <String Formula>
```

Where

<Parameter Sep Char> is a character used to separate parameter.

<String Formula> must evaluate to the value in the following order seperated by the <**Parameter Sep Char>**:

```
<Destination> <Source> <Company Name> <Company Number>
```

<Destination> is the path where the backup file is to be stored.

<Source> is the path where the backup file is available

<Company Name> is name of the Company

<Company Number> is number of the company

These four values must be specified for each company.These can be repeated for multiple companies.

Example: Single Company

```
[Button : My Cmp Res Button]
   Title    : Restore Cmp
   Action   : Restore Company : “,” : “C:\Tally.ERP 9\Data,C:\,+
              Global Enterprises,10037”
```

Example: Multiple Company

```
[Button : My Cmp Res Button]
   Title    : Restore Cmp
   Action   : Restore Company : “,” : “C:\Tally.ERP 9\Data,C:\,+
              Global Enterprises,10037,C:\Tally.ERP 9\Data,C:\,+
              TDL Demo,10027”
```

OR

```
[Button : My Cmp Res Button]
   Title    : Restore Cmp
   Action   : Restore Company : “,” : @@MyCmpFor
[System :Formula]
   MyCmpFor:“C:\,C:\Tally.ERP 9\Data,Global Enterprises,10037, +
              C:\,C:\Tally.ERP 9\Data,TDL Demo,10027”
```

Action – ChangeCrypt Company

This action allows to change the TallyVault Passward of multiple companies in one click.

Syntax

```
      ChangeCrypt Company : <parameter sep char> : <String Formula>
```

Where

<Parameter Sep Char> is a character used to separate parameter.

<String Formula> must evaluate to the value in the following order seperated by the **<Parameter Sep Char>** :

```
      <Company Data Folder> <New Tally Vault Key> <Old Tally Vault Key> +
      <Company Name> <Company Number>
```

<Company Data Folder> is the path of the company data folder

<New Tally Vault Key> is the new password of the company

<Old Tally Vault Key>is the old password of the company

<Company Name> is name of the Company

<Company Number> is number of the company

These five values must be specified for each company. These can be repeated for multiple companies.

Example:

```
[Button : Chg Pwd]
   Title      : Change Pwd
   Key        : Alt + b
   Action     : ChangeCrypt Company: “,” : ”C:\Tally.ERP 9\Data\10037, +
                NewPwd,OldPwd,Global Enterprises,10037”
```

Action – Browse URL

A New action **Browse URL** has been introduced. It is used to open web browser with any URL formula passed as a parameter. A list of parameters separated by space can be specified, if the application accepts command line parameters. **Exec Command** is an alias for action Browse URL.

Syntax

```
      Action : Browse URL : <URL Formula> [: <command line parms>]
```

Where,

<URL formula> is an expression which evaluates to any link to a web site

<command line parms> List of command line parameters separated by space

Browse URL Key Action can be used to open web browser with any URL Formula.

Example:

```
[Button : Open Notepad]
   Title      : $$LocaleString:"Notepad"
   Key       : ALT + N
   Action     : Exec Command : Notepad : “Browse URL.Txt”
[Form : Hyperlink]
   Parts       : Hyperlink
   Button     : Open Notepad
```

Example: Field acting as a hyperlink

```
[Key: Execute Hyperlink]
   Key       : Left Click
   Action      : Browse URL : “www.tallysolutions.com”
```

```
[Field: Hyperlink Company]
   Color     : Blue
   Border    : Thin Bottom
   Key       : Execute Hyperlink
   Set as    : "Tally Solutions Pvt. Ltd"
   Local     : Key : Execute Hyperlink : Action : Browse URL: +
                        http://www.tally.co.in
```

*Existing Action **Register Tally** has been removed for generalization which is now replaced with Action **Browse URL**.*

Action – HTTP Post

A new Key/ Button Action **HTTP Post** has been introduced which will help in exchanging data with external applications using web services. In other words, HTTP Post Action can be used to submit data to a server over HTTP and gather the response. This will enable a TDL Report to perform a HTTP Post to a remote location.

This Action will be discussed in detail under the topic **HTTP XML Collection**.

Action – Refresh TDL

A new Key/ Button Action **Refresh TDL** has been introduced which allows the TDL programmer to reload the active TDL Files without having to restart Tally.

Syntax

```
      Action: Refresh TDL
```

Example: Field acting as a hyperlink

```
[Key: Refresh TDLs]
;; Any Key can be assigned if Report already have F5 assigned
   Key       : F5
   Action      : Refresh TDL
;; Refresh TDL will work from any Report
[#Form: Default]
   Key       : Refresh TDLs
```

5. Events introduced

Tally.ERP 9 Series A Release 1.0 onwards, actions can also be carried out based on certain events. On encountering these events, the given action or list of actions will be executed.

Currently, two events have been introduced in Tally.ERP 9:

- On: Form Accept
- On: Focus

5.1 Event – On Form Accept

A new event **On:Form Accept** is introduced that can be specified within **Form** Definition. A list of actions can be executed when the form is accepted which can also be based on some condition.

Syntax

```
        On: Form Accept: <Condition>:  Action: Action parameters
```

Where,

<Condition> should return a logical value.

<Action> any one of the actions

<Action Parameters> parameters of the actions specified.

Example:

```
   [Form : TestForm]
      On    : FormAccept:Yes:HttpPost:@@SCURL:ASCII:SCPostNewIssue:+
            SC NewIssueResp
```

5.2 Event – On Focus

A new Event **On: Focus** is introduced which can be specified within definitions **Part**, **Line** and **Field**. When Part, Line or Field receives focus, a list of actions get executed which can also be conditionally controlled.

Syntax

```
      On : Focus : Condition : Action : Action parameters
```

<Condition> should return a logical value.

<Action> any one of the actions.

<Action Parameters> parameters of the actions specified.

Since On : Focus is a list type attribute as many actions can be specified which will be executed sequentially.

Example:

```
   [Part: TestPart1]
      On    : FOCUS : Yes  : HTTP Post : @@MyUrl : ASCII : ReqRep, RespRep
   [Part: TestPart2]
      On    : FOCUS : Yes  : CALL : SCSetVariables  : $$Line
```

6. User Defined Function

This is one of the breakthrough changes which has taken place at the platform level. We all know that TDL is a definition language which provides capability for rapid development. But now TDL is procedural as well. With the introduction of Functions/Procedures as a part of Tally.ERP 9 family the TDL capabilities have reached a new dimension.

This will help the application programmers to develop their own functions for achieving business functionality. Their will be a decrease in platform dependency for particular business function. The resultant would be faster development cycles for business modules.

The creation and usage of functions is discussed in detail in the section III **"User Defined Functions for Tally.ERP 9"**.

7. New Functions

Following functions are introduced in the Language:

7.1 $$IsObjectBelongsTo

Existing function **IsBelongsTo**, will only check if the current object belongs to a specified object. The new function **IsObjectBelongsTo** has been introduced to provide more explicit control in the hands of the programmer by allowing him to specify the object type and name in addition to parentage against which it needs to be checked. This function is very useful in the context of summarized objects as they are not of any native type and are just aggregation of objects. This function allows an easy link back into the native object type and walk up the chain. It is very useful when creating hierarchical reports on summarized collections.

Syntax

```
$$IsObjectBelongsTo:ObjType:ObjName:BelongsToName
```

Where,

<ObjType> denotes the Type of the Object,

<ObjName> denotes Name of the Object and

<BelongsToName> denotes the name of the object type.

Example:

Whether Group *North Debtors* belongs to Group *Sundry Debtors* or not directly or indirectly can checked using the following statement.

```
$$IsObjectBelongsTo:Group:"North Debtors":$$GroupSundryDebtors
```

7.2 $$NumLinesInScope

Tally.ERP 9 onwards, various operations can be performed on multiple lines. To know how many lines were considered for any operation, Function NumLinesInScope has been introduced.

Syntax

```
        $$NumLinesInScope:<ScopeKeyword>
```

where Scope Keyword can be *All Lines*, *Selected Lines*, *UnSelected Lines*, *Current Line/Lines*.

Example:

```
[Field: Sample Fld]

    Set As    : $$NumLinesInScope:SelectedLines
```

Field *Sample Fld*, displays total number of selected lines in the Part to which it belongs to.

7.3 $$DateRange

A new Built-in function $$DateRange is introduced to convert data types of the value from one form to due date format. Prior to this only through Field's format specification conversion was possible. Now the new function can be used inside User Defined Functions also.

Syntax

```
        $$DateRange:<Due Date Expression>:<Base Date Expression>:<Flag>
```

Where ,

<Base Date Expression> is a String Expression and evaluates to DueDate

<Due Date Expression> is a String Expression and evaluates to Date

<Flag> is a logical expression decides whether to include date given in second parameter.

Example:

In the below code snippet the method 'Order Due date' will have value as "10 Days" from $Date and the $Date is also inclusive.

```
 SET VALUE : OrderDueDate : $$DateRange:"10 Days":$Date:True
```

All the extract values now can be achieved using GroupBy, hence the $$Extract functions has been removed in Tally.ERP 9 . Eg: $$ExtractGrpVal, $$ExtractLedVal etc.

8. Enhanced Collection Capabilities

Collection, the data processing artifact of TDL provides extensive capabilities to gather data not only from Tally database but also from external sources using ODBC, DLLs, and HTTP and so on. A set of new capabilities have been added to Collection which provides far more flexibility and power in the hands of the TDL programmer. This will allow writing significantly complex reports with ease and still delivering enhanced performance with high volume of data.

8.1 Aggregation and Reporting

Tally.ERP 9 onwards, Collection has been enriched with the following capabilities.

- Data Roll up/ Summarization
- Collection re-use, extraction and chaining
- Indexed or Searchable Collection on TDL defined keys

Following attributes under Collection have been introduced to achieve the above.

Source Collection

In the context of the summary collection i.e. to achieve Data roll up, this attribute is mandatory. Source Collection specifies the collections to be used for source data. Multiple Source Collections can be used which can either be specified as a comma separated list or can be listed in several lines.

Syntax

```
      Source Collection : <Collection name>, <Collection Name> …
```

Where,

<Collection Name> is any predefined collection, the methods and sub objects of which are available to the current collection for further processing

Example:

```
[Collection: Vouchers Collection]
   Type              : Voucher
[Collection: Summary Collection]
   Source Collection : Vouchers Collection
```

The 'Summary Collection' uses 'Vouchers Collection' as source data.

Walk

Attribute Walk allows specifying further elements to walk on the source. Walk is optional and if not specified, the methods pertaining to source object only are available to be used. Walk can be specified to any depth for within the source object. This gives enormous flexibility and power. The Walk list has to be specified in the order in which they occur in the source object.

Syntax

```
      Walk : <Sub-Object Type/ Sub-Collection>[, <Sub-Object Type/ +
             Sub-Collection> …]
```

Where,

<Sub–Object Type/Sub-Collection> is the name of the Sub Objects/ Sub Collection.

Example:

```
[Collection: Vouchers Collection]
   Type                    : Voucher


[Collection: Summary Collection]
   Source   Collection : Vouchers Collection
   Walk               : Inventory Entries
```

In the *Summary Collection*, by saying *Walk : Inventory Entries*, only methods within Inventory Entries Object are available to the current collection. In case, Objects pertaining to Batch Allocations are required, then Walk can be written as

```
   Walk : Inventory Entries, Batch Allocations
```

wherein all the methods within Batch Allocations will be available to the current collection.

By

Attribute By is mandatory and it allows to specify the criteria based on which the aggregation is done. In other words, it works like **GROUP – BY**. Aggregation criteria can be one or more.

Syntax

```
      By : <Method-Name>: <Method-Formula>
```

Example:

```
[Collection: Vouchers Collection]
   Type                : Voucher


[Collection: Summary Collection]
   Source Collection   : Vouchers Collection
   Walk                : Inventory Entries
   By                  : PartyLedgerName    : $LedgerName
   By                  : StockItemName      : $StockItemName
```

In ‘Summary Collection’, Partywise Stock Items are clubbed on which Aggregation i.e., Sum/Min/Max operations would be performed.

Aggr Compute

Aggr Compute attribute is used for aggregation purpose based on the criteria(s) specified with attribute By. Aggregation can be done to find Sum, Minimum or Maximum of the Method within the Grouped Method. The Method on which Aggregation has to be performed can be of Data Type Number, Quantity, Rate or Amount.

Syntax

```
      Aggr Compute  :  <Method-Name> : <Aggr-Type> : <Method-Formula>
```

Where

<Method–Name> refers to the method where the result can be stored and referred to later.

<Aggr–Type> takes operation to be performed on the given method within the given criteria i.e., *Sum*, *Max* or *Min*.

<Method–Formula> should be evaluated to method names on which Aggregation operation needs to be performed.

Example:

```
[Collection: Vouchers Collection]
    Type : Voucher
[Collection: Summary Collection]
    Source   Collection  : Vouchers Collection
    Walk                 : Inventory Entries
    By                   : PartyLedgerName   : $LedgerName
    By                   : StockItemName     : $StockItemName
    Aggr Compute         : BilledQty         : Sum   : $BilledQty
```

BilledQty method retains the result of Aggregation i.e., Summation of method BilledQty for a StockItem within a particular Party.

Compute

Apart from the ones used in By and Aggr Compute attributes, none of the other methods can be accessed unless they are declared explicitly. One of the ways of declaring the required methods is by listing them using attribute Compute

Syntax

```
      Compute : <Method-Name> : <Method-Formula>
```

Example:

```
      Compute : Date : $Date
```

Method *Date* is being declared and made available for subsequent use.

Fetch

Another way of declaring required methods is by listing them in Fetch attribute. The only difference here is that the method names of the Objects within this collection has to be referred by the same name as in the Object.

Syntax

```
      Fetch : <Existing-Method-Name-in-Source> …
```

Where,

<Existing – Method Name in source> refers to the methods of the source collection.

Example:

```
   Fetch      : Date, Narration
```

is equivalent to writing

```
   Compute     : Date      : $Date
   Compute    : Narration : $Narration
```

Fetch using wildcard characters:

The two wild characters can be used in Fetch attribute * and ?.

- * is used To fetch all the methods and collections of the current object in context.
- **?** is used To fetch all the methods of current object in context.

Example:

- To fetch all methods of current Object within Walk.

  ```
  Fetch  : ?
  ```

- To fetch all methods and collection of current Object within Walk.

  ```
  Fetch  : *
  ```

- To fetch the methods StockItemName,BilledQty,Amount and all the method of collection Batch Allocation

  ```
  Fetch  : StockItemName, BilledQty, Amount, BatchAllocations.*
  ```

Keep Source

The attribute Keep Source is used to store the source data in main memory. The default value of this attribute is No.

When the Source Collection from which the Summary Collection is being prepared has a large number of objects and Keep Source is set to Yes, then the system goes out of memory since holding those objects in memory in one shot is not possible.

When Keep Source is set to No, the source objects are not retained in memory and they are processed as they are collected.

Syntax

```
      Keep Source : Yes/No/...
```

Where,

Each dot specifies parent one level up

. - Single dot retains the data of the source collection in current object.

.. - Double Dot will retain the data of the Source Collection in current object's parent.

... - Triple Dot will retain the data of the Source Collection in the current object's parent's parent and so on.

Example:

- Keep the source collection in the current owner

```
Keep Source : Yes
```

OR

```
Keep Source : .
```

- Don’t keep the source collection data

```
Keep Source : No
```

- Keep the current source collections data in the current object’s parent

```
Keep Source : ..
```

- Keep the current source collections data in current object’s grand parent

```
Keep Source :...
```

Please note that using the current object as a source-collection means Keep Source is N/A as there is no actual source collection created.

Search Key

This attribute is used to create index dynamically where the TDL programmer can define the key and the Collection is indexed in the memory using the Key. Once the index is created, any object in the collection can be instantly searched without needing a scan as in the case of a filter. Search Key is **Case Sensitive**.

This attribute has to be used in conjunction with function **CollectionFieldByKey**. This function basically maps the Objects at the run time with the Search Keys defined at the Collection.

Syntax

- ***Attribute – Search Key***

```
Search Key : < Combination of Method name/s >
```

- ***Function – CollectionFieldByKey***

```
$$CollectionFieldByKey:Method-Name:Key-Formula:CollectionName
```

Where

<*Method-Name*> is the name of the method,

<Key-Formula> is a formula that maps to the methods defined in the search key exactly in the same order.

Data Source

Attribute data source allows to specify XML file as data source. The collection can be created directly from the specified XML file and the data in the XML file can be displayed in a report.

Syntax

```
        DataSource : <Type> : <file path> : <Encoding>
```

Where,

<Type> specifies the type of data source. File Xml or HTTP XML

<File Path> data source file path

<Encoding> ASCII or UNICODE. This is Optional .The default value is UNICODE.

Example:

```
[Collection : My XML Coll]
   DataSource : File Xml : "C:\MyFile.xml"
```

In the above code snippet the type of file is 'File XML ' as the data source is XMl file. The encoding is Unicode by default as it is not specified.

Example:

```
[Collection : My XML Coll]
   DataSource : HTTP Xml : "http:\\localhost\MyFile.xml": ASCII
```

In the above code snippet the type of file is 'HTTP XML' as the data source is obtained through HTTP. The encoding of the file 'MyFile.XML' is ASCII.

Support for other data sources like ODBC, Dll will also be available in future releases.

Data Roll up/summarization capability in TDL Collection

Data roll up/ summarization capability facilitates the creation of large summary collections of aggregations in a single scan using the new attributes of the Collection definition as discussed above.

Prior to Tally.ERP 9, all the totals were generated using functions like **CollAmtTotal** or **FilterAmtTotal** via collections. These have certain advantages and disadvantages. While they provide excellent granularity and control, each call is largely an independent activity to gather the data set and then aggregate it. This can make the code very complex and may not scale up if a large number of totals need to be generated as in the case of most business summary reports or a large underlying data set being used. Considering the object oriented nature of Tally data and existence of sub-objects up to any level, the task becomes even more complex. These functions require multiple data scans to produce a summary report with multiple rows and columns.

This methodology has now been complemented with a single scan to get all the totals including those based on User Defined Fields (UDFs). Native aggregation capability has now been added to a collection itself. The overall effect is a reduction in TDL code complexity and resource requirement, enhanced performance by orders of magnitude especially concerning reports generation.

Example: 1

```
[Collection: My Source Collection]
   Type              : Voucher

[Collection: My Summary Collection]
   Source Collection : My Source Collection
   Walk               : Ledger Entries
   By                : MyLedgeName : $LedgerName
   Aggr Compute      : My Total    : Sum  : $Amount
```

In the above code snippet, *My Summary Collection* is created out of source collection *My Source Collection* where traversing is done to *Ledger Entries* using *Walk* and *Ledger Name* is the method on which aggregation is performed to find the sum of the ledger amount.

Example: 2

In some scenario, current object itself is required as a Source and aggregation is performed on collection obtained from it or its sub-collections. In such circumstances, if we use Source Collection as Voucher, then the entire vouchers within the company will be scanned unnecessarily to find the current one which is a time consuming process. To avoid this, we can use *Source Collection: Default*, which will assume the current voucher as a Source.

```
[Collection : LedgerInAccAllocations]
   Source Collection : Default
   Walk              : InventoryEntries, AccountingAllocations
   By                : LedgerName : $LedgerName
   Compute           : RateOfVA : $RateOfVAT:TaxClassification:+
                       $TaxClassificationName
   Aggr Compute      : Amount   : Sum   : $Amount
   Filter               : IsVATLedgerinAcc
[System: Formula]
   IsVATLedgerinAcc  : $$IsSysNameEqual:VAT:$TaxType:Ledger:$LedgerName
```

While printing a voucher as an invoice, if an aggregation has to be done on its tax ledgers to show as summary within the invoice, this has to be collected from the accounting allocations of the same voucher.

Collection re–use, extraction and chaining support in TDL Collection

A collection can extract information from other collections including its sub-objects with the choice of method(s), filter(s) and sort-order. Source Collection within a collection, collection(s) can be chained. In other words, Summary Collection can be used as Source Collection for some other Collection and so on.

Example:

```
[Collection: My Source Collection]
   Type              : Voucher


[Collection: My Summary Collection]
   Source Collection : My Source Collection
   Walk              : Ledger Entries
   By                : MyLedgerName : $LedgerName
   Aggr Compute      : MyTotal      : Sum : $Amount
[Collection: My Parent Summary Collection]
   Source Collection : My Summary Collection
   By                          : MyParent     : $Parent:Ledger:$LedgerName
   Aggr Compute      : MyParentTotal: Sum : $MyTotal
```

In the above code snippet, *My Parent Summary Collection* extracts a sub-set of information from a collection to an already summarized collection *My Summary Collection*.

Indexed or Searchable Collection on TDL defined keys

The capabilities discussed above extend the data gathering capabilities of TDL. However business reporting in general and in Tally uses hierarchical presentation or columnar presentation rather than simple table representation. This creates a unique and natural experience of working with the product and business data.

In case, one can simply repeat the summarized collection and get the desired report, everything works fine with the existing capabilities. However, if Report is having two or more dimensions like Ledger and Cost Center and so on, a simple repeat on the summarized collection will not suffice.

Let us understand the same with the help of an example.

Example:

When a Report to be designed with ledgers as rows and cost centers as columns, the following options are available:-

- Use function(s) like **CollectionField** or **FilterValue** in each column.
- Create **Summary Collection** for each column.

The first one will scan through the whole collection for every value required. The second one will scan the whole source data as many times as number of columns. Both of them will take a significant hit on the scale and volume that it can handle and affect the resultant performance.

To provide presentation capabilities beyond simple tables, a new capability has been added to the Collection definition. A search key can be defined in the collection using the Search Key attribute. This implies that a unique key is created for every object which can be used to instantly access the corresponding objects and its values without needing to scan or re-collect. The corresponding function created to access the same is **$$CollectionFieldByKey**.

Example:

```
[Collection: LedCC]
   Use          : Voucher Collection
   Walk         : LedgerEntries, Category Allocations, Cost Centre +
                 Allocations
   By           : PartyLedgerName : $PartyLedgerName
   By           : Cost Centre Name: $Name
   Aggr Compute : Amount  : $Amount
   Search Key   : $PartyLedgerName + $CostCentreName

[Field: My Rep Field]
   Set as       : $$CollectionFieldByKey:$Amount:@MySearchKey:LedCC
   MySearchKey  : #LedName + #CCName
```

In Collection *LedCC*, a search key is created for every object with the help of Ledger Name and Cost Center.

Now on any row/column in the report, combination total is accessed using

```
     $$CollectionFieldByKey:$Amount:@MySearchKey:LedCC
```

Where *MySearchKey* is the formula to get the *Ledger Name* + *Cost Center name* at a particular point, LedName is the Field having LedgerName in current context and CCName is the variable storing the Cost Centre Name in current context.

8.2 The Summary Collection is available through Tally ODBC Interface

Now Objects of the Summary Collection can be exposed to Tally ODBC Interface through Collection attribute ‘Is ODBC Table’. The values of the Collection attributes “Fetch’, ‘Compute, ‘By’ and Aggr Compute’ are available through Tally ODBC Interface.

Syntax

```
      [Collection: <Name of Summ Coll>]
            Is ODBC Table : <Logical value>
```

Where **<Name of Summ Coll>** is the name of the Summary Collection and **<Logical value>** can be either Yes or No.

Example:

```
   [Collection: Source Collection]
            Type              : Voucher
   [Collection: Summary Collection]
            Source Collection : My Source Collection
            Walk              : Ledger Entries
            By                               : LedgerName: $LedgerName
            Aggr Compute      : Total            : Sum : $Amount
            Compute                       : Parent: $Parent:Ledger:$LedgerName
            Is ODBC Table          : Yes
```

The values of methods of 'Summary Collection' 'LedgerName', 'Total' and 'Parent' are exposed to Tally ODBC interface.

8.3 HTTP XML Collection (GET and POST with and without Object Specification)

Collection capability has been enhanced to gather live data from **HTTP/web-service** delivering **XML**. The entire XML is now automatically converted to TDL objects and is available natively in TDL reports as $ based methods. There is no need to access the data via specialized functions like **$$XMLValue**. Reports can be shown live from an HTTP server. Coupled with the new [OBJECT:] extensions and POST action you can also submit data back to the server almost operating Tally as a client to HTTP-XML web-services.

HTTP – XML Collection

Consider the following XML data stored in the file *TestXML.xml* which is available at server *Remote Server*.

```
<CUSTOMER>
  <NAME>Sapna Awasthi</NAME>
  <EMPID>1000</EMPID>
  <PHONE>
        <OFFICENO>080-66282559</OFFICENO>
        <HOMENO>011-22222222</HOMENO>
        <MOBILE>990201234</MOBILE>
  </PHONE>
  <ADDRESS>
        <ADDRLINE>C/o. Info Solutions</ADDRLINE>
        <ADDRLINE>Technology Street</ADDRLINE>
        <ADDRLINE>Tech Info Park</ADDRLINE>
  </ADDRESS>
</CUSTOMER>
```

This capability allows us to retrieve and store this data as objects in Collection. The attributes in collection for gathering XML based data from a remote server over HTTP are RemoteURL, RemoteRequest, XMLObjectPath, and XMLObject. Whenever the collection is referred the data is fetched from the remote server and is populated in the collection.

Syntax

```
[Collection: <Collection Name>]
   RemoteURL      : http-url
   RemoteRequest :<request-report-name>,<pre-request-display-report> : +
                  <encoding type>
   XMLObjectPath : <Start-node> : <Path-to-start-node>
   XMLObject      : <TDL-Object-Name>
```

Where

Remote–URL attribute is used to specify the URL of the HTTP server delivering the XML data

RemoteRequest attribute is used to specify the Report name which is to be sent to the HTTP server as an XML Request. If the report requires user inputs then it has to be accepted before the request is sent. Pre-request display report specifies the name of the report which accepts the user-input.

XMLObjectPath attribute is used when only a specific fragment of the response XML is required and converts the same to TDL Objects in Collection. By default, it takes the root node.

<Start-Node> allows you to specify the name and position of the XML node from which the data should be extracted. It takes two parameter as follows:

```
         <Node Name> : <Position>
```

<Path-to-Start-Node> is used to specify the path to reach the <*start node*> from the root node.

The path specification is :

```
      <Root-node> : <Child Node> : <Start Pos> : <Child Node>: <Start Pos> …
```

XMLObject attribute is used to specify the TDL Object specification.

The following syntax is used for object specification

```
      [Object: <Object Name>]
          Storage      : <Name>     : Type
          Collection   : <Name>     : Type
```

/ The second Parameter in the Collection Type can be a Object type in case of a complex collection or a simple data type in case of simple collection */*

All these attributes cater to specific requirements based on the GET request or POST request and whether the obtained data is stored in Tally.

Prerequisites for data transfer over HTTP

In order to retrieve the data available in *TestXML.xml* file from a remote server (Pre-defined IP Address) ensure that web service is running on the machine. Check for IIS Server Installation. The file *TestXML.xml* can be copied to the directory *C:\Inetpub\wwwroot* to be accessible at the root and then the URL can be specified as follows *http://localhost/TestXML.xml*.

If the XML request needs to be processed at the remote server by a file (.asp, .php, etc.), at least one web server (e.g., IIS, Apache etc) and PHP/ASP must be installed on the system.

Simple GET Request

If it is required to access the data (XML format) from remote server in a collection it is sufficient to specify the URL of the server only. The attribute RemoteURL is used. The data thus obtained is available in the collection as objects and can be accessed as native methods.

The collection to populate XML Data available at the URL *http://Remoteserver/TestXML.xml* is created as follows:

Example:

```
[Collection: XML Get Collection]
   Remote URL      : "http://RemoteServer/TestXML.xml"
```

This collection can be used in a TDL Report to display the data retrieved. The method names will be same as the XML Tag names.

By default, all the data from XML file is made available in the collection. If only a specific data fragment is required it can be obtained using the collection attribute *XML Object Path*.

Example:

From the XML file, if only address is required then the collection is defined as follows:

```
[Collection : XML Get CollObjPath]
   Remote URL      : "http://Remoteserver/TestXML.xml"
   XML Object Path : ADDRESS:1:CUSTOMER
```

Consider that the XML file on the remote server contains multiple customer objects with the hierarchy mentioned earlier. The file *"TestXML.xml"* has the following structure :

```
<CUSTOMERS>
         <CUSTOMER>
             .
             .
         </CUSTOMER>
         <CUSTOMER>
             .
             .
         </CUSTOMER>
         <CUSTOMER>
             .
```

```
              .
         </CUSTOMER>
</CUSTOMERS>
```

If the address of second Customer is required then the collection is defined as shown:

```
[Collection : XML Get CollObjPath]
   Remote URL      : "http://Remoteserver/TestXML.xml"
   XML Object Path : ADDRESS:1:CUSTOMERS:CUSTOMER:2
```

Consider that the Address further contains data as shown:

```
            <CUSTOMER>
               .
               .
                <ADDRESS>
                       <PHONE> 9902012345 </PHONE>
                       <PHONE> 9902099020 </PHONE>
                </ADDRESS>
               .
             </CUSTOMER>
```

In this case to retrieve the second phone number of third customer, the collection is defined as follows:

```
   [Collection : XML Get CollObjPath]
      Remote URL      : "http://Remoteserver/TestXML.xml"
      XML Object Path : PHONE:2:CUSTOMERS:CUSTOMER:3:ADDRESS:1
```

Simple GET Request and mapping the response to TDL Object

The data available in XML format is at the URL “http://Remoteserver/TestXML.xml”. The data is required to be mapped as TDL Objects. The collection attribute *XML Object* is used to specify the object name to which the obtained data is mapped.

Example:

```
[Collection: XML Get Collection]
    Remote URL : "http://Remoteserver/TestXML.xml"
    XML Object : Customer Data
```

The Object specification for “Customer Data” is as follows:

```
[Object: Customer Data]
   Storage     : Name      : String
   Storage      : EmpId      : String
   Collection   : Phone      : XML Phone Coll      ;; Complex Collection
   Collection   : ADDRESS    : XML AddressColl     ;; Complex Collection

[Object: XML Phone Coll]
   Storage      : OfficeNo    : String
   Storage      : HomeNo      : String
   Storage      : Mobile      : String
[Object: XML AddressColl]
   Collection   : AddrLine : String       ;; Simple collection
```

A Simple POST

If a TDL report is to be sent to the HTTP server as an XML request and the XML response is to be obtained in the collection, then the collection attribute “Remote Request” is used. The attribute “Remote Request” takes a Report name as a parameter which sends the request in XML format to the web page on the remote server. The response data received from the server is then available in the collection.

Example:

The Test.php page on the remote server accepts the data in the following XML format.

```
<ENVELOPE>
    <REQUEST>
        <NAME>Tally</NAME>
        <EMPID>00000</EMPID>
    </REQUEST>
</ENVELOPE>
```

Following collection sends request in the above XML format with the help of a TDL report XML-PostReqRep. The encoding scheme selected is ASCII.

```
[Collection: XML Post Collection]
    Remote URL    : "http://Remoteserver/test.php"
    RemoteRequest : XMLPostReqRep : ASCII
    XMLObjectPath : CUSTOMER
```

The report *XMLPostReqRep* is automatically executed when the collection is referred.

In the Report, the *XMLTAG attribute* is used at Part and Field Definitions.

```
[Part: XMLPostReqRep]
   XML Tag  : "REQUEST"
   Scroll     : Vertical
```

```
[Field: XMLPostReqRepName]
   XML Tag  : "NAME"
   Set As   : " Tally "
[Field: XMLPostReqRepPwd]
   XML Tag  : " EMPID "
   Set As   : " 00000 "
```

The XML Tag *<Envelope>* is added by Tally while sending the XML request.

The response received from *http://Remoteserver/test.php* page is the same XML given previously.

The data now available in the collection can be displayed in a report.

Post Request with Pre-request Report

A Pre-Request report is required when some inputs are to be accepted from the user and the XML Request is to be generated out of those inputs. In that case, a TDL report is used which has to be accepted first. If the data captured through pre request report has to be sent to remote server for processing then it has to be made available in the Request report. The input report name is specified as Pre-Request report.

```
[Collection: XML Post Collection]
   Remote URL    : "http://localhost/test.php"
   RemoteRequest : XMLPostReqRep, XML PreReqRep : ASCII
   XMLObjectPath : CUSTOMER
```

The Report *XMLPostReqRep* sends the XML request to the page *Test.php* in the format described earlier. Before sending the XML request to the page the data entered in the report *XML PreReqRep* must be accepted. The data entered in the Pre-Request report can also be sent to the remote server in the XML request. Both the reports are triggered when the collection is referred.

Action – HTTP POST

A new Key/ Button Action **HTTP Post** has been introduced which will help in exchanging data with external applications using web services. In other words, HTTP Post Action can be used to submit data to a server over HTTP and gather the response. This will enable a TDL Report to perform a HTTP Post to a remote location.

Syntax

```
      [Key: <Key Name>]
          Key    : <Key Combination>
          Action : HTTP Post : <URL Formula> : <Encoding> : +
                   <Request Report>: <Error Report> : <Success Report >
```

Where,

<URL Formula> can be any string formula which resolves as an URL and is defined under System Definition.

<Encoding> is the encoding scheme ASCII or UNICODE .

<Request Report> is the name of the TDL Report which will be used for generating XML Request to be sent.

<Error Report > is displayed in case of failure.

<Success Report> is displayed when the post is successful.

The details pertaining to URL (at the receiving end), Encoding Format, Request Report, Error Report and Success Report should be specified along with HTTP Post Action. Universal Resource Locator (URL) for which information is intended has to be specified through a System Formula.

Encoding Format specifies the encoding to be used while transmitting information to the receiving end. The valid encoding formats are ASCII and UNICODE. UNICODE is set by default.

Request Report is the name of the TDL Report which will be used for generating XML Request to be sent. Error Report and Succcess Reports are optional and will enable the programmer to display a Report with the details of the XML response received.

Success and failure is determined by <STATUS> tag in the standard message format. If it is 1, it is a success other wise failure. Based on the value of the <STATUS> tag 0/1, the error report and success report are executed respectively. It will not close or accept the form if status is not equal to 1. Both the Request / Response are exchanged in XML format.

Example:

```
   [Key: XMLReqResp]

       Key    : Ctrl + R

       Action : HTTP Post : @@MyUrl : ASCII : ReqRep: ERRRespRep: SuccRep

       Scope  : Selected Lines
;;URL Specification must be done as a system formula
   [System: Formula]

        MyUrl : http://127.0.0.1:9000
```

The defined Key XMLReqResp in the snippet above must be attached to an initial Report. When the report is activated and this Key is pressed, the Action HTTP Post activates a defined report ReqRep which generates the request XML. The response data is made available in collection called Parameter Collection. The reports ERRRespRep and SuccRep can use the Parameter Collection to display the error message/data in the Report.

Notes

The XML response received for the action HTTP POST must be in the Tally compatible XML format. The file "XML for HTTP POST" shows the format received as a reponse from the PHP application file "CXMLResponse as per Tally".

8.4 Usage As Tables

A Collection in TDL as we all understand can populate the data from a wide range of sources which are available as Objects in the Collection.

The various sources of Objects in Collection are:

- External Objects i.e. Objects created by the TDL programmer
- Internal Objects i.e. All Internal Objects provided by platform and stored in Tally DB. For example Ledger, Group, Voucher, Cost Centre, Stock Item etc
- Objects populated in the collection from an external database using ODBC referred to as ODBC Collection
- Objects populated in collection from an XML file present on the remote server over HTTP. This collection is referred to as an XML Collection.
- Objects obtained after aggregation of data from lower level in the hierarchy of internal objects.

Tables are based on Collections. Prior to this release, not all collection types as given above could be used as tables. Not all internal objects were available in the Table. Only the masters i.e. Groups, Ledgers, Stock Items, etc could be displayed in the Table. Using Vouchers in table was not possible. Data from ODBC Collection was also not possible.

From this Release onwards, all limitations pertaining to usage of Collections as Tables have been completely eliminated. Any Collection which can be created in TDL can be displayed as a table now. Collection with Aggregation and XML Collections can also be used as Tables.

Prior to this release, the following types of Collections could not be used as Tables:

- Voucher Collection As Table
- Collections with Aggregation As Table
- Displaying information at lower levels in Object hierarchy in a Table
- Displaying aggregate methods in Table
- Displaying ODBC Collection As Table
- Displaying XML Collection As Table

Let us consider the following examples to understand the capability in a better way

Voucher Collection As Table

Now the Vouchers can be displayed as table in a field.

Example: Voucher Collection as Table

```
[Collection: Vch Collection]

   Type           : Voucher

   Filter         : PurcFilter

   Format         : $VoucherNumber, 10

   Format         : $VoucherTypeName, 25

   Format             : $PartyLedgerName, 25

   Format         : $Amount, 15

[System: Formula]

   PurcFilter  : $$IsPurchase:$VoucherTypeName
```

```
;;Field displaying Table
[#Field: EI OrderRef]
   Table        : Vch Collection
   Show Table   : Always
```

Collection with Aggregation As Table

Example: 1 – Displaying Inventory Entries (lower level information in Voucher) As Table

```
[Collection: Vch Collection]
   Type              : Voucher
   Filter            : PurcFilter
[Collection: Summ Collection]
   Source Collection : Vch Collection
   Walk                       : Inventory Entries
   By                : Name   : $StockItemName
[System: Formula]
   PurcFilter        : $$IsPurchase:$VoucherTypeName
;; Field displaying Table
[#Field: EI OrderRef]
   Table             : Summ Collection
   Show Table        : Always
```

Example: 2 – Displaying Collections with aggregate methods As Table

```
[Collection: Vch Collection]
   Type                : Voucher
   Filter            : PurcFilter
[Collection: Summ Collection]
   Source Collection : Vch Collection
   Walk              : Inventory Entries
   By                : Name      : $StockItemName
   Aggr Compute      : BilledQty : Sum : $BilledQty
   Aggr Compute      : Amount    : Sum : $Amount
   Format            : $Name, 25
   Format            : $BilledQty, 25
   Format            : $Amount, 25

[System: Formula]
   PurcFilter        : $$IsPurchase:$VoucherTypeName
```

```
;;Field displaying table
[#Field: EI OrderRef]
   Table              : Summ Collection
   Show Table         : Always
```

ODBC Collection As Table

Example: Data fetched from Excel file in Collection displayed as Table

In the sample given below the excel file *"Sample Data.xls"* containing the data is present in the path *"C:\Sample Data.xls*". If the complete path is not specified, it locates the Excel file in the Tally application folder.

```
[Collection: ODBC Excel Collection]
   ODBC        : "Driver= {Microsoft Excel Driver (*.xls)};DBQ= C: +
                 \Sample Data.xls"
   SQL         : "Select * From [Ledgers$]"
   Format      : $_1, 25
   Format      : $_2, 20
   Format      : $_3, 15
   Format      : $_4, 25

;; Field displaying table
[#Field: EI OrderRef]
   Table        : ODBC Excel Collection
   Show Table : Always
```

XML Collection As Table

XML Data fetched from Remote URL in Collection as Table .Below is the XML Data Sample to be retrieved from the Remote URL

```
<CUSTOMER>
    <NAME>Keshav</NAME>
    <ADDRESS>
       <ADDRLINE>Line1</ADDRLINE>
       <ADDRLINE>Line2</ADDRLINE>
       <ADDRLINE>Line3</ADDRLINE>
    </ADDRESS>
    <ADDRESS>
        <ADDRLINE>Line1</ADDRLINE>
        <ADDRLINE>Line2</ADDRLINE>
        <ADDRLINE>Line3</ADDRLINE>
    </ADDRESS>
</CUSTOMER>
```

In the example, the complete URL of the file is *http://localhost/XMLData.xml*.Here the file *XMLData.xml* is located in the local machine. Instead of local host, the IP Address of the machine or 127.0.0.1 can be specified. The web service should be installed in the machine.

Example:

```
[Collection: XML Table]
   RemoteURL     : "http://localhost/XMLData.xml"
   XMLObjectPath : CUSTOMER
   Format        : $NAME, 25
;; Field displaying Table
[#Field: EI OrderRef]
   Table         : XML Table
   Show Table    : Always
```

8.5 Dynamic Object support for HTTP–XML Information Interchange

When a Collection is used for editing (alter/create), objects are dynamically added to the collection when a new line is repeated over the same. The type of object which is added depends on the specification in the TYPE attribute. In case the TYPE attribute is not specified it defaults to adding a standard empty object. So if the TYPE is ledger, a ledger object would be added and so on.

However, the following holds true for a COLLECTION keeping in mind the latest enhancements

- It can be made up of multiple types of objects (say Ledgers and Groups)
- It can have TDL defined objects which are retrieved from XML file .They are specified using XML Object.
- It can have aggregated objects

Depending solely on the TYPE attribute for deciding the object type is a constraint with respect to the above facts. This is now being removed with the introduction of a new attribute which will independently govern the type of object to be added to the collection on-the-fly. The following is now supported in collection

```
NEWOBJECT: type-of-object: condition
```

Whenever a new object is to be added at the collection level, it will walk through the NEW

OBJECT attribute specification and validate the condition specified. The first one which is satisfied decides the type of object to be added. The object can be a schema defined internal object or a TDL defined object [OBJECT: MYOBJECT]

The capability to use objects defined in TDL is being separately enhanced and shown here for completeness of the NEW OBJECT attribute. As of now, these TDL defined objects can be used only for HTTP-XML based exchange with other systems to take input and send requests or receive XML and operate them like TDL objects. They cannot be persisted or saved into the Tally company database.

Please refer the following code snippet for Object specification.

Example: This collection can be used in a Report opened in Alter Mode.

```
[Collection: Coll Customer]
   New Object    : Customer Data                      ;; New TDL Object Defined

[Object: Customer Data]
   Storage       : Name         : String
   Storage       : CustId       : String
   Collection    : PhoneColl    : Phone               ;; Complex Collection
   Collection    : AddressColl : Address              ;; Complex Collection

[Object: Phone]
   Storage       : OfficeNo     : String
   Storage       : HomeNo       : String
   Storage       : Mobile       : String

[Object: Address]
   Storage       : AddrLine1    : String
   Storage       : AddrLine2    : String
```

In case there is no NEW OBJECT specified, the existing behavior will continue for backward compatibility. In case of Sub-Objects like LedgerEntries, the same behavior continues since they are added by their parent objects and not by the Collection.

9. Collection Capabilities for Remoting

Enabling access to your organizational data in an 'any-time, any-where' and yet being truly usable is what Tally.ERP 9 delivers. With Tally.NET enabled remote access, it will be possible for any authorized user to access Tally.ERP 9 from anywhere.

Major Enhancements have taken place at the collection level to achieve remoting capabilities. The attributes Fetch, Compute and AggrCompute provided at the collection level and FetchObject and FetchCollection at the report level significantly help in above functionality.

The remoting capabilities are discussed in detail in the next section II **"Writing Remote Compliant TDL for Tally.ERP 9 "**.

Writing Remote Compliant TDL Reports

Introduction

Enabling access to your organizational data in an ‘any-time, any-where’ and yet being truly usable is what Tally.ERP 9 delivers.With Tally.NET enabled remote access, it will be possible for the owner or any authorized user to access Tally.ERP 9 data from anywhere.With this capability they will be able to access all the reports and information from a remote location.

All these has been made possible by adopting Client/Server architecture in the product. The underlying principle of any client/server environment is the communication between client and server in a request/response fashion.The request/response is in the form of XML.Client sends request and the server responds.

Starting from Tally.ERP 9 family the default product delivers the capability to access any TDL reports from anywhere.There have been significant enhancements in Tally platform at the Collection,Report and Function Level for delivering this capability.The way TDL Reports have been changed in default TDL to optimize the performance and seamlessly work without clogging the network is the focus of this document.The idea is to reduce the server calls for accessing the data.The same concepts can be followed for creating the customized Reports Remote Compliant.

Given below is the overall enabled enviroment using Tally.NET

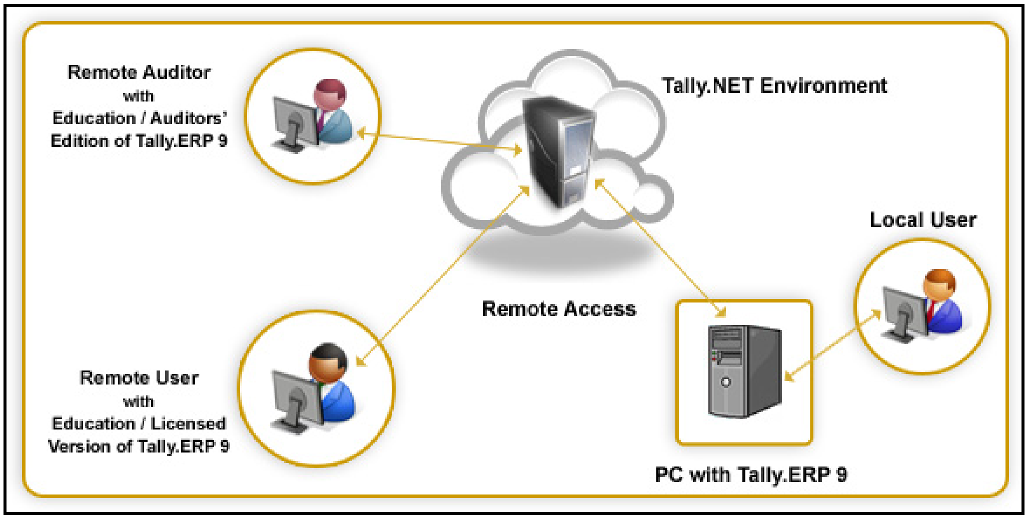


Figure 1.1 Overview of Tally.NET

We will begin our discussion with an overview of client/server environment in general and then moving on to Tally Client/Server, the role of Tally.NET server in such a scenario.The topics covered henceforth will focus on on understanding the execution of TDL reports and optimizing the code for executing in this environment.

1. Client/Server Architecture – An Overview

Clients and Servers are separate logical entities that work together over a network to accomplish a task. A client is defined as a requester of services and a server is defined as the provider of services.

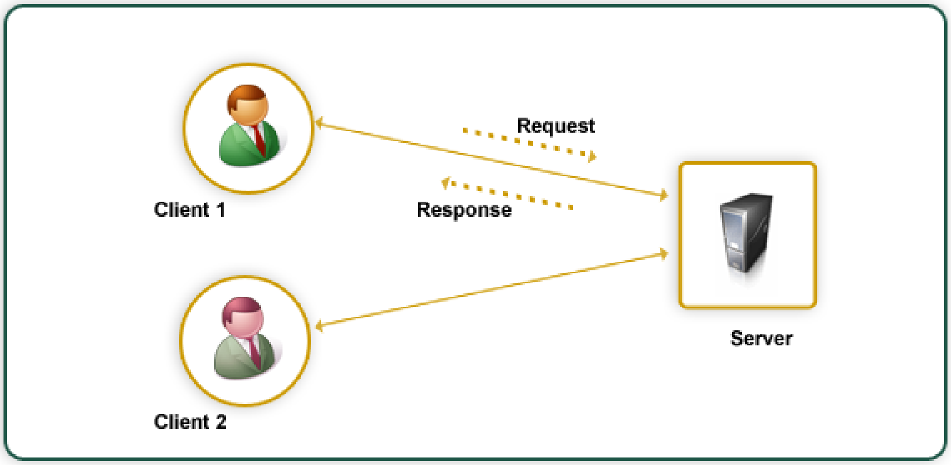


Figure 1.2 Block diagram of client/server architecture

Some of the advantages of client/server architecture are as follows

- Centralization - Resources, and data security are controlled through the server
- Scalability – Entire system can be scaled horizontally or vertically.
- Accessibility - Server can be accessed remotely

2. Tally Client/Server Architecture using Tally.NET

Tally.NET is a framework which provides number of services to Tally.ERP 9 users. The Tally.NET architecture is derived from client/server architecture. In this architecture Tally.ERP 9 Client is connected to Tally.ERP 9 server via middleware ie Tally.NET Server. Following are the major components of the Tally.NET architecture.

- Tally.NET Server
- Tally.ERP 9 Server
- Tally.ERP 9 Client

2.1 Tally.NET Server

Tally.NET Server is a middleware in Tally.NET architecture. The communication between Tally.ERP 9 Server and Tally.ERP 9 client is being handled by Tally.NET Server. It provides the Routing services for Tally. It is through Tally.NET server that we are able to provide an entire range of services which we commonly refer to as Tally.NET features. The user can utilize Tally.NET to Synchronize data, access online help and support, manage licenses across locations and the auditor can use it to scrutinise the client's data from a remote location all this can be done in a secured environment.

The system administrator can create users with the rights to access or audit data from a remote location and assign controls based on their security level for the required company only. The remote users accessing the company data behave as clients on Tally.NET. Tally.NET takes care of the user authentication when a remote user tries to connect to the Tally.ERP 9 Server.

Tally.NET Features

- Register and Connect companies from Tally.ERP 9
- Create and maintain Remote Users
- Remote availability of Auditor's License
- Synchronization of data (via Tally.NET)
- Remote access of data by any user (including BAP users)
- Use online help and support capability from within Tally or browser
- Application Management (across Multi-serial, Multi-Location) via Tally or browser

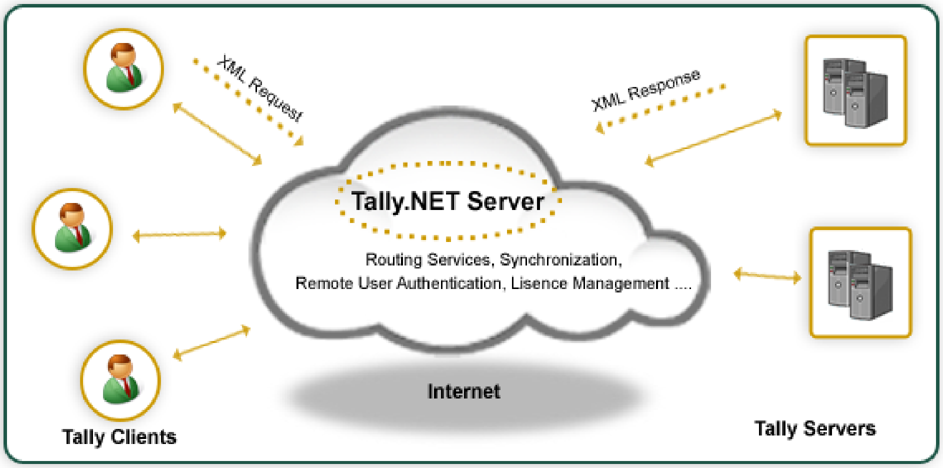

XML Request
XML Response
Tally.NET Server
Routing Services, Synchronization,
Remote User Authentication, Lisence Management ....
Internet
Tally Clients
Tally Servers

Figure 1.3 Tally.NET Architecture

2.2 Tally.ERP 9 Server

Tally.ERP 9 Server is a typical Tally application which hosts the Tally Company and is always connected to the Tally.NET server. User creation, authorization, connecting the company to Tally.NET server is handled at this end.

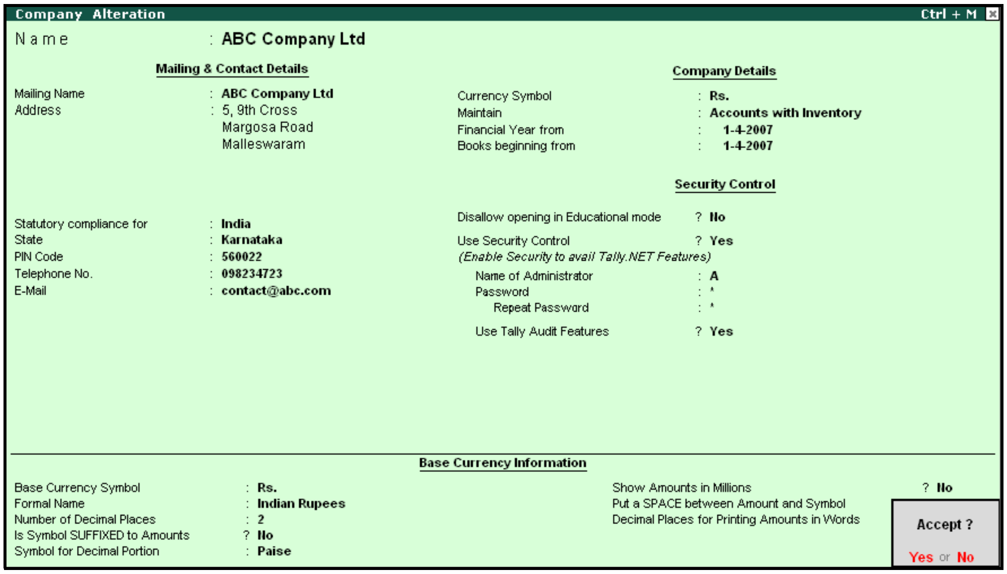
2.3 Tally.ERP 9 Client

Tally.ERP 9 Client is a typical Tally application. Tally client can remotely access the Tally Company which is hosted by Tally server. Authenticated users connect to enabled companies from this end.

3. Setting up Server Tally for Remote Access

Following are the steps needs to be executed to setup the Tally server.

Step 1:- Enable Security control to avail Tally.NET features.
Go to **Gateway of Tally** ,click **F3 :Company Info.** > **Alter**



Name : ABC Company Ltd

Mailing & Contact Details

Mailing Name : ABC Company Ltd
Address : 5, 9th Cross
Margosa Road
Malleswaram

Statutory compliance for : India
State : Karnataka
PIN Code : 560022
Telephone No. : 098234723
E-Mail : contact@abc.com

Company Details

Currency Symbol : Rs.
Maintain : Accounts with Inventory
Financial Year from : 1-4-2007
Books beginning from : 1-4-2007

Security Control

Disallow opening in Educational mode ? No
Use Security Control ? Yes
(Enable Security to avail Tally.NET Features)
Name of Administrator : A
Password : *
Repeat Password : *
Use Tally Audit Features ? Yes

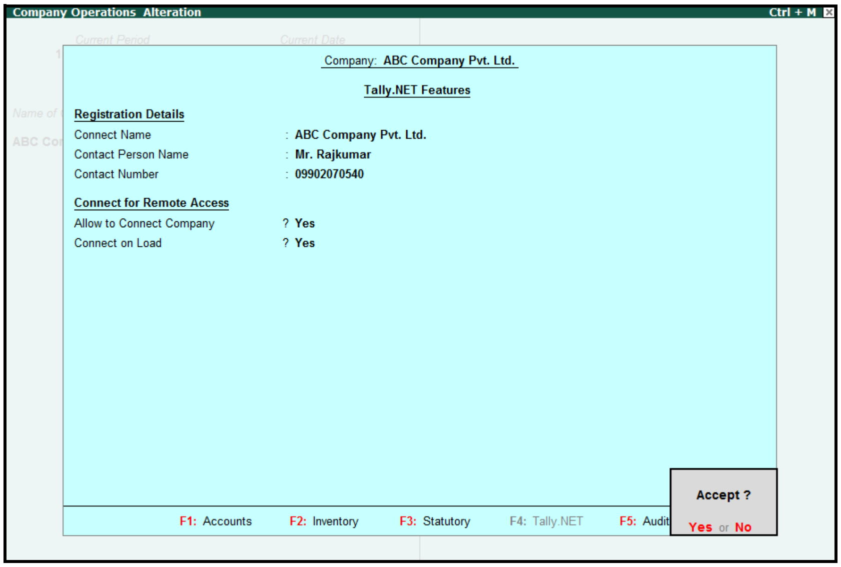
Base Currency Information

Base Currency Symbol : Rs.
Formal Name : Indian Rupees
Number of Decimal Places : 2
Is Symbol SUFFIXED to Amounts ? No
Symbol for Decimal Portion : Paise

Show Amounts in Millions ? No
Put a SPACE between Amount and Symbol
Decimal Places for Printing Amounts in Words

Figure 1.4 Enabling Security control to avail Tally.NET features

Step 2:- Configuring Tally.NET features
Go to **Gateway of Tally,** click **F11:Features** > **Tally.NET Features**



Company: ABC Company Pvt. Ltd.

Tally.NET Features

Registration Details

Connect Name : ABC Company Pvt. Ltd.
Contact Person Name : Mr. Rajkumar
Contact Number : 09902070540

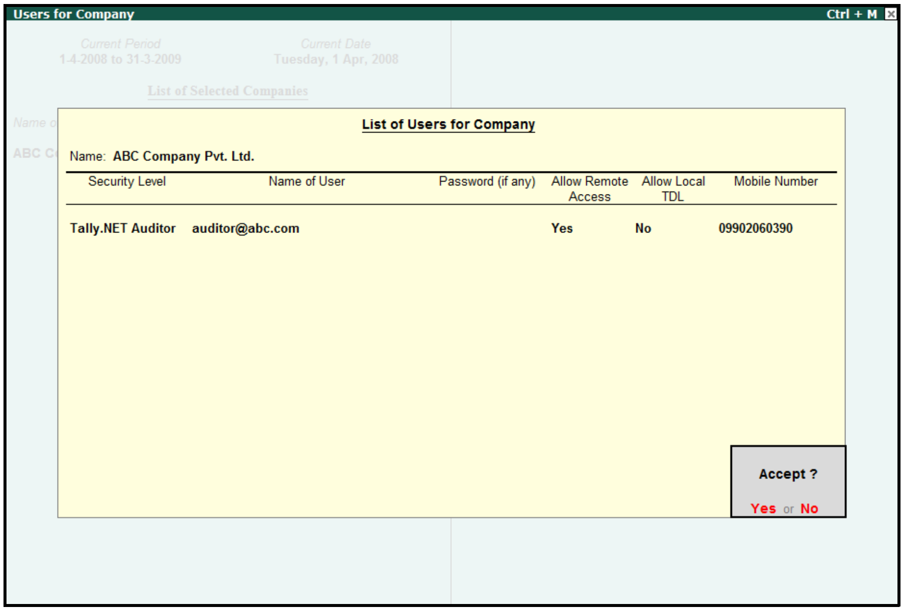
Connect for Remote Access

Allow to Connect Company ? Yes
Connect on Load ? Yes

Figure 1.5 Configuring Tally.NET features

Step 3:- Authorizing the Remote Users

Go to **Gateway of Tally** ,click **F3 :Company Info** > **Security Control > Users & Passwords**



List of Users for Company

Name: **ABC Company Pvt. Ltd.**

Security Level	Name of User	Password (if any)	Allow Remote Access	Allow Local TDL	Mobile Number
Tally.NET Auditor	auditor@abc.com		Yes	No	09902060390

Accept ?

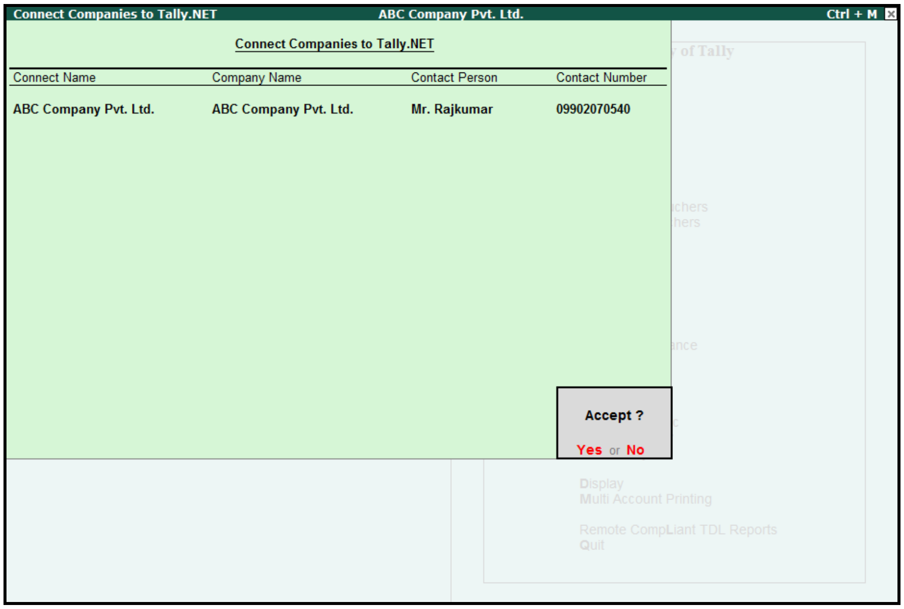
Yes or No

Figure 1.6 Authorizing the Remote Users

- *Users classified under the security level Tally.NET User and Tally.NET Auditor should be created individually by the system administrator.*
- *Allow Remote Access should be set to Yes only if client wants his Tally.NET Auditor/ Tally.NET User to access data remotely.*
- *If Allow Local TDL is set to Yes, then client can load Local TDLs in addition to remote TDLs.*
- *If Allow Local TDL is set to No, then client can not load Local TDLs.*

Step 4:- Connecting Companies to Tally.NET

Go to **Gateway of Tally** ,click **F4:Connect Company**


Connect Companies to Tally.NET ABC Company Pvt. Ltd. Ctrl + M

Connect Companies to Tally.NET

Connect Name	Company Name	Contact Person	Contact Number
ABC Company Pvt. Ltd.	ABC Company Pvt. Ltd.	Mr. Rajkumar	09902070540

Accept ?

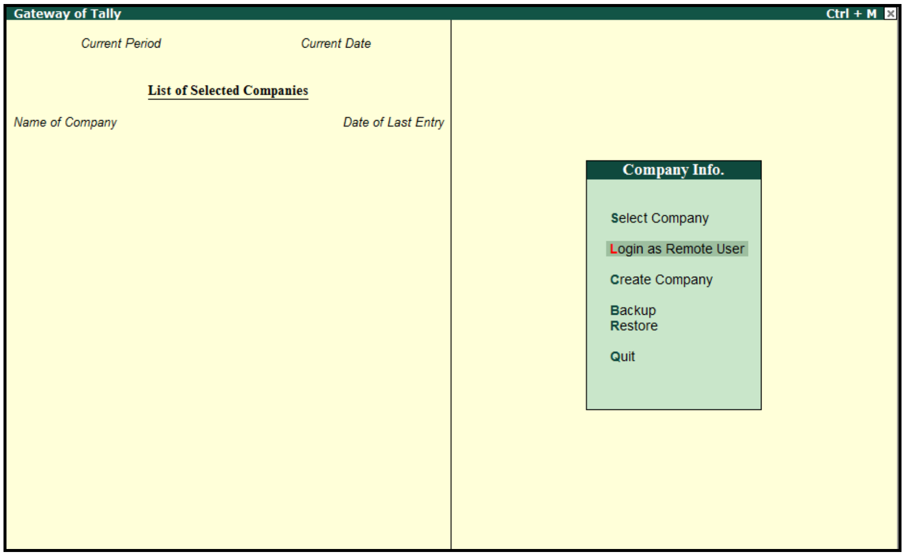
Yes or No

Figure 1.7 Connectiong companies to Tally.NET

4. Setting up the Client Tally

The users classified under Tally.Net User or Tally.NET Auditor can access data by logging in from a remote location. The user has to execute the following steps to login as a remote user.

Step 1:- Get connected to the Tally.NET


Gateway of Tally Ctrl + M

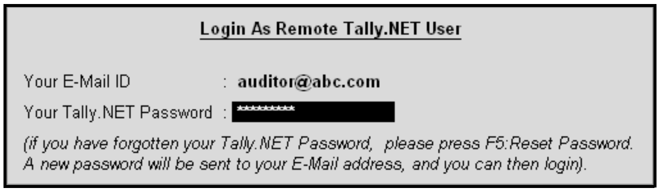
Current Period Current Date

List of Selected Companies

Name of Company Date of Last Entry

Figure 1.8 Connectiong Tally.NET

Step 2:- Provide the User name and password



Login As Remote Tally.NET User

Your E-Mail ID : auditor@abc.com

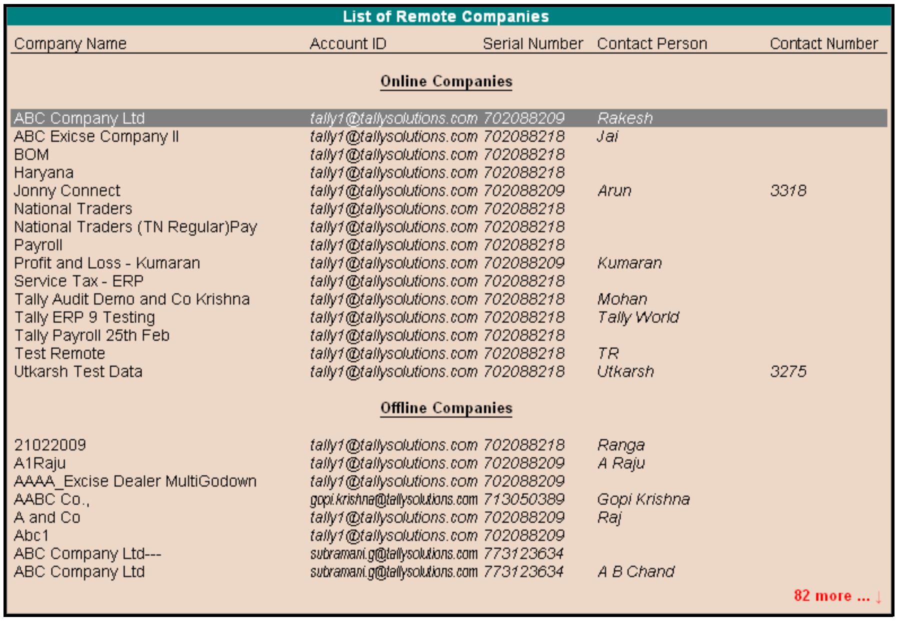
Your Tally.NET Password : *********

(if you have forgotten your Tally.NET Password, please press F5:Reset Password.
A new password will be sent to your E-Mail address, and you can then login).

Figure 1.9 Providing User Name and Password

After entering the valid username and password Tally displays the screen to select the remote company.

Step 3:- Load the Remote company



List of Remote Companies

Company Name	Account ID	Serial Number	Contact Person	Contact Number
Online Companies				
ABC Company Ltd	tally1@tallysolutions.com	702088209	Rakesh	
ABC Exicse Company II	tally1@tallysolutions.com	702088218	Jai	
BOM	tally1@tallysolutions.com	702088218		
Haryana	tally1@tallysolutions.com	702088218		
Jonny Connect	tally1@tallysolutions.com	702088209	Arun	3318
National Traders	tally1@tallysolutions.com	702088218		
National Traders (TN Regular)Pay	tally1@tallysolutions.com	702088218		
Payroll	tally1@tallysolutions.com	702088218		
Profit and Loss - Kumaran	tally1@tallysolutions.com	702088209	Kumaran	
Service Tax - ERP	tally1@tallysolutions.com	702088218		
Tally Audit Demo and Co Krishna	tally1@tallysolutions.com	702088218	Mohan	
Tally ERP 9 Testing	tally1@tallysolutions.com	702088218	Tally World	
Tally Payroll 25th Feb	tally1@tallysolutions.com	702088218		
Test Remote	tally1@tallysolutions.com	702088218	TR	
Utkarsh Test Data	tally1@tallysolutions.com	702088218	Utkarsh	3275
Offline Companies				
21022009	tally1@tallysolutions.com	702088218	Ranga	
A1Raju	tally1@tallysolutions.com	702088209	A Raju	
AAAA_Excise Dealer MultiGodown	tally1@tallysolutions.com	702088209		
AABC Co.,	gopi.krishna@tallysolutions.com	713050389	Gopi Krishna	
A and Co	tally1@tallysolutions.com	702088209	Raj	
Abc1	tally1@tallysolutions.com	702088209		
ABC Company Ltd---	subramani.g@tallysolutions.com	773123634		
ABC Company Ltd	subramani.g@tallysolutions.com	773123634	A B Chand	

82 more ... ↓

Figure 1.10 Loading Remote Company

The above screen displays the list of companies to which the remote user has access. First all the Online companies are listed followed by the list of offline companies.

5. TDL – In a Client/Server Environment

In client/ server environment, data resides in the server. A typical client will have only user interface. Whenever client requires data, it has to send request to the server with credentials and server will respond with the data.

In Tally.NET environment, server and client exchange the request/response in encrypted XML format. When a client is Tally application, Tally client will have only user interface and needs to get data from the server on demand. A typical Tally application is developed in TDL.In TDL language, definitions are broadly classified as Data Objects & Interface Objects. Interface object define the user interface and Data objects store the value in Tally primary or secondary database. Tally client will have only Interface Objects locally and Data Objects needs to be fetched from the server on request.

It is TDL Programmer's responsibility to fetch the required data from the Tally server to Tally Client.

6. TDL Enhancements for Remote

TDL language is enhanced with the client/server capability. Collection and Report definitions are enhanced to make server calls.Enhancements have taken place in the platform for the execution of the Functions and Actions.

6.1 Collection Enhancements

In TDL, Collection definition is a data repository which contains the data objects. Whenever Tally Client uses a Collection, it has to fetch the objects from the Remote server. But Tally Client need not require the all the methods of an Object. Also fetching the entire Object may be costly in terms of network bandwidth.

The required methods of an object(s) at the Tally Client are fetched using the Collection attribute 'Fetch'. In addition to 'Fetch', methods which are doing aggregation or computation using 'Aggr Compute' & 'Compute' are also brought to Tally Client.

Internally fetching a method will generate a XML fragment and will be sent to Tally Server as a request.

1. **Fetch**

Syntax

```
Fetch : Existing-Method-Name-in-Source, …
```

Where,

<Existing-Method-Name-in-Source> are the internal methods of the Object which needs to be fetched to Client.

2. **Compute**

```
Compute : Method-Name : Method-Formula
```

Where,

<Method-Formula> is any computational method and Method-Name denotes the name of the method.

Please refer 'TDL Enhancements for Tally.ERP 9.pdf' for further information on Collection attributes 'Aggr Compute' , 'Compute' and 'Fetch'

Example: Fetching Name & Closing Balance of Ledger Object

Step 1:- Fetching Name and Closing Balance method of Ledger object

```
[Collection: Ledgers]
   Type                      : Ledger
   Fetch      : Name, Closing Balance, Parent
   Compute   : PClosingBalance: $ClosingBalance:Group:$Parent
   Format      : $Name, 15
```

Step 2: - Utilizing the fetched methods
a) As a Table

```
[Field: Sample Field]
   Table               : Ledgers
   Show Table     : Always
```

b) In Repeat at Part Level

```
[Part: Sample Part]
   Line  : Sample Line
   Repeat: Sample Line : Ledgers

   [Line: Sample Line]
      Fields            : Sample Fld1, Sample Fld2, Sample Fld3

      [Field: Sample Fld1]
         Use          : Name Field
         Set as : $Name

      [Field: Sample Fld2]
         Use    : Amount Field
         Set as : $ClosingBalance

      [Field: Sample Fld3]
         Use    : Amount Field
         Set as : $PClosingBalance
```

Sample Request Format XML file to fetch the internal methods and Compute method

```
<ENVELOPE>
     <HEADER>
          <VERSION>1</VERSION>
          <TALLYREQUEST>EXPORT</TALLYREQUEST>
          <TYPE>COLLECTION</TYPE>
          <ID>Ledger</ID>
     </HEADER>
     <BODY>
         <DESC>
            <STATICVARIABLES>
             <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
             <SVCURRENTCOMPANYTYPE="String">DemoCompany</SVCURRENTCOMPANY>
            <SVCURRENTDATE TYPE="Date">20-Dec-2008</SVCURRENTDATE>
             <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>
             <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>
             <SVCURRENTKBLANGUAGEID TYPE="Number">1033
           </SVCURRENTKBLANGUAGEID>
         </STATICVARIABLES>
         <TDL>
         <TDLMESSAGE>
            <COLLECTION NAME="Ledger" ISMODIFY="No" ISFIXED="No"
                      ISINITIALIZE="Yes" ISOPTION="No" ISINTERNAL="No">
            <TYPE>Ledger</TYPE>
            <METHOD>PClosingBalance:$ClosingBalance:Group:$Paren </METHOD>
            <NATIVEMETHOD>Name</NATIVEMETHOD>
            <NATIVEMETHOD>Parent</NATIVEMETHOD>
            <NATIVEMETHOD>ClosingBalance</NATIVEMETHOD>
           </COLLECTION>
         </TDLMESSAGE>
      </TDL>
   </DESC>
</BODY>
<ENVELOPE>
```

6.2 Report Level Enhancements

Fetching the Object

When a multiple methods of a single Object is required for a Report, then that Object can be fetched at Report level. For this purpose new Report attribute ‘Fetch Object’ is introduced. Internally fetching an object will generate a XML fragment and will be sent to Tally Server as a request.

Syntax

```
      Fetch Object: <Object Type> :<Object Name>:<Method Name1 +
                    [,<Method Name 2>…]
```

Where,

<Object Type> denotes the type of the Object,

<Object Name> denotes the name of the object and

<Method Name 1> denote the method to be fetched.

Example: Pre fetching Ledger Object with methods Name & Closing Balance

```
[Report: Simple Report]
   Fetch Object : Ledger : Ledger Name : Name, Parent,Closing Balance
```

In the above code snippet, Ledger Name is the variable which stores the name of the Ledger Object whose methods needs to be fetched at the Report.

Sample Request Format XML file to fetch the object

```
<ENVELOPE>
   <HEADER>
      <VERSION>1</VERSION>
      <TALLYREQUEST>EXPORT</TALLYREQUEST>
      <TYPE>OBJECT</TYPE>
      <SUBTYPE>Ledger</SUBTYPE>
      <ID TYPE="Name">Cash  </ID>
   </HEADER>
   <BODY>
      <DESC>
         <STATICVARIABLES>
            <SVCURRENTCOMPANY>Demo Company</SVCURRENTCOMPANY>
            <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
            <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>
            <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>
            <SVCURRENTDATE TYPE="Date">1-May-2008</SVCURRENTDATE>
            <SVVALUATIONMETHOD TYPE="String"></SVVALUATIONMETHOD>
            <SVBUDGET TYPE="String"> </SVBUDGET>
            <SVCURRENTKBLANGUAGEIDTYPE="Number">1033
            </SVCURRENTKBLANGUAGEID>
            <SVCURRENTUILANGUAGEIDTYPE="Number">1033
            </SVCURRENTUILANGUAGEID>
         </STATICVARIABLES>
         <FETCHLIST>
            <FETCH>Name</FETCH>
            <FETCH>Parent</FETCH>
            <FETCH>Closing Balance</FETCH>
         </FETCHLIST>
      </DESC>
   </BODY>
</ENVELOPE>
```

Pre Fetching the Object

There are some scenarios in which it is required to set the value of variables according to the data fetched along with the object.At the report level the Set attribute for changing variable value takes precedence and Fetch Object is evaluated later.In those cases fetching the object first becomes mandatory.For this purpose a new attribute “Pre Fetch Object” is introduced which will be evaluated before the Set attribute.

Syntax

```
      Pre Fetch Object: <Object Type> :<Object Name>:<Method Name1 +
                        [,<Method Name 2>…]
```

Where,

<Object Type> denotes the type of the Object,

<Object Name> denotes the name of the object and

<Method Name 1> denote the method to be pre fetched.

Example:

```
[Report: Simple Report]
   Set                    : LedgerName: “Cash”
   Pre Fetch Object       : Ledger : LedgerName : LastVoucherDate
   Set                                : SVFromDate: $LastVoucher-
   Date:Ledger:##LedgerName
```

In the above code snippet, variables are set once and the PreFetchObject is done and again the variables are set to make sure that the values of the variables which were depend on the object will set now

Pre fetching the Collection

When the same collection is used in the Report either for repeating the line over its objects or multiple functions using the same,then a Collection of those objects can be pre fetched at the Report level. A new Report attribute ‘Fetch Collection’ is introduced to pre fetch a Collection.

Syntax

```
      Fetch Collection:<Collection 1>[,<Collection 2>..]
```

Where,

<Collection 1> is the collection whose objects need to be pre fetched at Report

Example: Pre fetching Ledger collection

```
[Report: Sample Report]
   Fetch Collection      : Ledger
   Local                 : Collection: Fetch : Ledger
```

In the above code snippet Ledger Collection is pre fetched.

Sample Request Format XML file to fetch the object

```
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <TALLYREQUEST>EXPORT</TALLYREQUEST>
    <TYPE>COLLECTION</TYPE>
    <ID>All Party</ID>
  </HEADER>
  <BODY>
    <DESC>
      <STATICVARIABLES>
        <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
        <SVUSEPARMLIST>No</SVUSEPARMLIST>
        <SVFORTABLE>No</SVFORTABLE>
        <SVCURRENTCOMPANY TYPE="String">Remote Vivek</SVCURRENTCOMPANY>
        <SVCURRENTDATE TYPE="Date">2-May-2008</SVCURRENTDATE>
        <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>
        <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>
        <SVVALUATIONMETHOD TYPE="String"></SVVALUATIONMETHOD>
        <SVBUDGET TYPE="String"></SVBUDGET>
      </STATICVARIABLES>
      <TDL>
```

```
          <TDLMESSAGE>
            <COLLECTION NAME="All Party" ISMODIFY="No" ISFIXED="No"
                 ISINITIALIZE="Yes" ISOPTION="No" ISINTERNAL="No">
              <TYPE>Ledger</TYPE>
              <BELONGSTO>Yes</BELONGSTO>
              <CHILDOF>$$GroupSundryDebtors</CHILDOF>
              <NATIVEMETHOD>OpeningBalance</NATIVEMETHOD>
              <NATIVEMETHOD>ClosingBalance</NATIVEMETHOD>
            </COLLECTION>
          </TDLMESSAGE>
        </TDL>
      </DESC>
    </BODY>
</ENVELOPE>
```

6.3 Function on Request

Functions in TDL are defined and provided by the platform. TDL programmer can only call a function. Now in client/server environment functions can be evaluated by either sever or client or both client and server. Based on this information functions can be classified as follows.

1. Evaluated at client side
2. Evaluated at server side
3. Hybrid

Evaluated at client side

These are the functions which will be evaluated at the client side. For this no server request is required from the client. If these functions require any parameter as data, then required data needs to be fetched from the server before the function is called.

Example:

$$KeyExplode, $$ExplodeLevel ,$$Line etc are the functions which does not require any parameter from the Tally server and is executed at the Tally client.

Evaluated at server side

These are the functions which will be evaluated at the server side. For each call of a function, a request will be sent to server along with the parameters.

Example:

$$NumStockItems, $$NumLedgers etc. are the functions which will be executed at the server side.

Sample Request Format XML file for Function Call

```
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <TALLYREQUEST>EXPORT</TALLYREQUEST>
    <TYPE>FUNCTION</TYPE>
    <ID>$$NumLedgers</ID>
  </HEADER>
  <BODY>
    <DESC>
      <STATICVARIABLES>
       <SVCURRENTCOMPANY>Demo Company</SVCURRENTCOMPANY>
       <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
       <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>
       <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>
       <SVCURRENTDATE TYPE="Date">1-May-2008</SVCURRENTDATE>
       <SVCURRENTKBLANGUAGEID TYPE="Number">1033</SVCURRENTKBLANGUAGEID>
       <SVCURRENTUILANGUAGEID TYPE="Number">1033</SVCURRENTUILANGUAGEID>
      </STATICVARIABLES>
    </DESC>
  </BODY>
</ENVELOPE>
```

Hybrid

These are the functions which will be executed on either client or server side based on the availability of the data.

Example:

$$IsSales, $$CollAmtTotal, $$FilterAmtTotal etc are the functions which will be executed at the server or client side based on the availability of data.

Server side execution: -

```
$$FilterAmtTotal:$OpeningBalance:Ledgers:MyFilter
```

Since Ledger Collection is available on sever, so the function FilterAmtTotal will be executed at the Server end.

Client side execution: -

```
$$FilterAmtTotal:$Amount:LedgerEntries:MyFilter
```

‘Ledger Entries’ collection is available inside the Voucher Object. So the required Voucher Object needs to be fetched to the Client before the function is executed. Once the Voucher is brought to the client, function will be executed on the client side since it is assumed to be executed in Voucher context.

6.4 Action Enhancements

The Action “Modify Object” is executed in the Display mode of any report. This action can be executed at the client’s end to modify any object present on the Server Company.For details on usage of this action please refer ‘TDL Enhancements for Tally.ERP 9”

Syntax

```
      Action : Modify Object : <PrimaryObjectSpec>.<SubObjectPathSpec> +
               .Method-Name : value>[,Method Name: <value> , …] +
               [,<SubObjectPathSpec>.MethodName:<value>, …..]
```

where,

<PrimaryObjectSpec> can be (<Primary Object Type Keyword>, <Primary Object Identifier Formula>)

<SubObjectPathSpec> is given as CollectionName [<Index Formula>, [<Condition>]]

<MethodName> refers to the name of the method in the specified path.

<Index Formula> should return a number which acts as a position specifier in the Collection of Objects satisfying the given **<condition>**.

Sample Request Format XML file for Modifying Ledger Object

```
<ENVELOPE>
   <HEADER>
      <VERSION>1</VERSION>
      <TALLYREQUEST>IMPORT</TALLYREQUEST>
      <TYPE>DATA</TYPE>
      <SUBTYPE>Ledger</SUBTYPE>
      <ID>All Masters</ID>
   </HEADER>
   <BODY>
      <DESC>
         <STATICVARIABLES>
            <SVCURRENTCOMPANY>Demo Company</SVCURRENTCOMPANY>
         </STATICVARIABLES>
      </DESC>
      <TALLYMESSAGE>
         <LEDGER NAME="Customer 1" RESERVEDNAME="">
```

```
            <ADDRESS.LIST TYPE="String">
            <ADDRESS>Abc</ADDRESS>
            <ADDRESS>Def</ADDRESS>
            </ADDRESS.LIST>
            <MAILINGNAME.LIST TYPE="String">
            <MAILINGNAME>Customer 1</MAILINGNAME>
            </MAILINGNAME.LIST>
            <ALTEREDON>20090112</ALTEREDON>
            <NAME TYPE="String">Customer 1</NAME>
            <CURRENCYNAME>Rs.</CURRENCYNAME>
            <PINCODE>560001</PINCODE>
            <PARENT>Sundry Creditors</PARENT>
            <ISDEEMEDPOSITIVE TYPE="Logical">Yes</ISDEEMEDPOSITIVE>
            <SORTPOSITION> 1000</SORTPOSITION>
            <OPENINGBALANCE>1.00</OPENINGBALANCE>
            <LANGUAGENAME.LIST>
            <NAME.LIST TYPE="String">
            <NAME>Customer 1</NAME>
            <NAME>Alias</NAME>
            </NAME.LIST>
            <LANGUAGEID> 1033</LANGUAGEID>
            </LANGUAGENAME.LIST>
         </LEDGER>
      </TALLYMESSAGE>
   </BODY>
</ENVELOPE>
```

7. Writing Remote Compliant TDL Reports

TDL programmer can optimize the performance of the Remote compliant TDL by minimizing the server request call. Below mentioned are the guidelines to optimize the Remote Compliant TDL Reports.

7.1 Fetching the single Object

When an entire Report requires multiple methods of a single Object, then Object can be pre fetched with required methods. In this approach only one server call is made to fetch the all the required methods.

Example:

```
[Report: Final Led Report]
   Form                   : Final Led Report
   Fetch Object    : Ledger : LedgerName: Name,Ledger Contact,+
                                           Ledger Phone,TBalOpening, TBalClosing
```

7.2 Repeating Lines over a Collection

Following techniques are used to optimize the performance when a line is repeated over a collection in a report to be displayed on the client.

Fetching the Methods

Whenever a collection is reffered to in a Report,the required methods needs to be explicitly fetched from the server. It is mandatory to specify fetch in the Collection for all methods which are used in the fields.If fetch is not used then the data will not be displayed in the field.

```
[Part: LedReport]
   Line       : LedReportDetails
   Repeat   : LedReportDetails : Ledger
   Scroll   : Vertical

                  [Line: LedReportDetails]
      Fields                        : Led Name
      Right Field : LedClosingBalance

                  [Field: Led Name]
         Use            : Name Field
         Set as    : $Name

                  [Field: LedClosingBalance]
         Use       : Amount Forex Field
         Set as    : $ClosingBalance

[#Collection: Ledger]
   Fetch     : Name,Closing Balance
```

Function inside the Repeat

When Lines are repeated over a Collection and a function is used at the field level,then each repeat will trigger an additional server request for function call.In this scenario entire function call logic can be moved to ‘Compute’ of the repeated Collection. The later approach will do only one server request. Hence performance is drastically improved.

```
[Part: LedReport]
   Lines      : LedReportDetails
   Repeat    : LedReportDetails : Ledger
   Scroll    : Vertical

         [Line: LedReportDetails]
      Fields       : Led Name
      Right Fields: LedClosingBalance, LedSalesTotal

               [Field: Led Name]
         Use       : Name Field
         Set as    : $Name

               [Field: LedClosingBalance]
         Use       : Amount Forex Field
         Set as    : $ClosingBalance

                           [Field: LedSalesTotal]
         Use       : Amount Forex Field
         Set as    : $LedgerSalesTotal

[#Collection: Ledger]
   Fetch          : Name, Closing Balance
   Compute      : LedgerSalesTotal:+
                  $$AsReqObj:$$FilterAmtTotal:LedVouchers:MyParty:$Amount
```

Repeating over Period Collection

In Reports where lines are repeated over Period Collection and values of the each column is calculated over a period for the required object. For example in Sales Register value of each column is calculated based on a period and object Voucher Type. In this scenario, an additional computational method needs to be added to Period Collection to fetch the values for each column.

```
[#Collection: Period Collection]
   Compute : TBalDebits  : $TBalDebits:VoucherType:#VoucherTypeName
   Compute : TBalCredits: $TBalCredits:VoucherType:#VoucherTypeName
   Compute : TBalClosing: $TBalClosing:VoucherType:#VoucherTypeName
```

7.3 Using the same Collection in more than one Report

When more than one Report requires different methods of the Objects of the same Collection then using the same collection with all the methods fetched in it reduces the performance.This can be improved in the following ways

Fetching the required methods locally at Report

In the following code snippet, Sample Report1 requires Opening Balance of a Ledger where as Sample Report2 requires Closing Balance. Instead of modifying the Collection to fetch both Opening Balance and Closing Balance, same is localized in respective Reports.

```
[Report: Sample Report1]
   Local    : Collection : Ledger : Fetch : Opening Balance

[Report: Sample Report2]
   Local    : Collection : Ledger : Fetch : Closing Balance
```

Seperate Collections for fetching different methods

In the following code snippet two Collections are created for fetching opening balance and closing balance. Later first Collection can be utilized in the 'Sample Report1' and second one in the 'Sample Report2'

```
[Collection: Fetch Opening Balance]
   Type      : Ledger
   Fetch    : Opening Balance

[Collection: Fetch Closing Balance]
   Type      : Ledger
   Fetch    : Closing Balance
```

User Defined Functions

Introduction

TDL is a comprehensive 4GL language which gives tremendous power in the hands of the programmer by providing data management, complex report generation and screen design capabilities using only a few lines of code, leading to rapid development. Till now TDL had very few aspects of the procedural programming.

To mention a few

- Value calculations were achieved using System Formula or by writing external methods at object level.
- Repetitive execution of certain lines of code was possible using certain platform defined functions like $$CollectionField, $$CollectionAmtTotal etc. The functions used to take care of these implicitly.
- Sequential execution of certain segments of code was achieved by using Action List.

Now with the introduction of "User Defined Functions", a path breaking development in the history of TDL, procedural programming aspects are introduced into the language along with preserving the basic nature of a definition language.

1. Functions – In General

In procedural languages Functions are called as Sub routine or Procedures.

If it is required to execute a certain set of statements repeatedly to achieve a certain functionality it is not a good programming practice to write the same set of statements in the program again and again.

For example:

n separate set of statements in a computer program requires the sum of two numbers for some complex computation. Each statement will repeatedly compute a+b with different set of a and b. To avoid this a function is created which will accept a and b and return the result ie the sum to the calling program. This reduces the no of lines in the code along with improving code readability.

A function accepts certain values processes the values in a certain manner and finally returns a value to the calling program. The values which a function accepts or the calling program passes to the function are called parameters and the result which is passed by the function to the calling program is called the return value.

A function is mainly used for some of the following purposes:

1. Repeating a block of code
2. Perform some calculations
3. To execute set of statements

2. Functions – In TDL

In TDL prior to Tally.ERP 9 Functions were defined by Platform and TDL programmer could only call the function to achieve a certain functionality. From Tally.ERP 9 onwards functions can be defined in TDL layer. User Defined Function in TDL has been provided as a Definition which allows user to specify a set of actions/statements to be executed in the order as specified.

Traditionally TDL is a non procedural language, action driven language. The sequence of execution was not in the hands of programmer. But with this development in a Function Definition Conditional evaluation of statements and looping is made possible. User defined Functions basically can be used for performing complex calculations or executing a set of actions serially. Function can accept parameter(s) as input and return a 'Value' to the caller.

Functions give following benefits to the TDL programmer:

- Allows conditional execution/evaluation of statements
- Execution of a set of statements repeatedly generally referred to as loops
- To define variables and store values from intermediate calculation / process
- To accept parameters from the calling segment of code
- To work on data elements like, getting an object from the calling context, defining the function execution context, looping on the objects of a collection etc.
- Return a 'Value' to the caller of the function
- Perform a set of actions sequentially/conditionally or repeatedly without returning a value.

With this development the programmers can write business functions with complex computations by themselves without platform dependency.

3. Function – Building Blocks

In TDL Function is also a definition. It has two blocks

1. Definition Block
2. Procedural Block

A glimpse into the function .

```
[Function : Function Name]

      ;; Definition Block
      ;; Parameter Specification
             Parameter: Parameter 1  : Datatype

             Parameter: Parameter 2  : Datatype
```

```
;; Variable Declarations
        Variable : Var 1  : Number
        Variable : Var 2  : String
;; Explicit Object Association
        Object   : ObjName: ObjectType
;;Return Value
        Returns  :Datatype
;;Definition Block Ends here
;;Procedural Block
        Label 1  : Statement 1
        Label 2  : Statement 2
              |
              |
        Label n  : Statement n
 ;; Procedural Block Ends here
```

3.1 Definition Block

The definition Block is utilized for following purpose

3.2 Parameter specification

This is used to specify the list of parameters which passed by the calling code. The values thus obtained are referred to in the function with these variable names. The syntax for specifying the same is given below

Syntax

```
      PARAMETER: <Variable Name> :<Data Type of the Variable>
```

Where,

<Variable Name> is the name of the Variable which holds the parameter send by the caller of the Function.

<Data Type of the Variable> is the Data type of the Variable send by the caller of the Function

Example:

The Function ‘FactorialOf’ receives number as parameter from the Caller.

```
[Function : FactorialOf]
   Parameter : InputNumber : Number
```

Variable declaration

If a Function requires some Variable(s) for intermediate calculation, then those Variable(s) needs to be defined. The scope of these Variable(s) will be within the Function and looses its value after exiting the Function.

Syntax

```
      VARIABLE : <Variable Name> [:<Data Type of the Variable>]
```

Where,

<Variable Name> is the name of the Variable

<Data Type of the Variable> is the Data type of the Variable.

Datatype is optional. If datatype is specified a separate Variable definition is not required (these are considered as inline variables). If data type is not specified the interpreter will look for a variable definition with the name specified.

Example:

The Function 'FactorialOf' requires intermediate Variables 'Counter' and 'Factorial' for calculation within the Function Definition.

```
[Function : FactorialOf]
   Parameter : InputNumber     : Number
   Variable  : Counter         : Number
   Variable  : Factorial       : Number
```

Static Variable declartion

Static Variable is a Variable, whose value persists between successive calls to a Function.

The scope of the static variable is limited to the Function where in which it is declared and exists for the entire session.

Syntax

```
      STATIC VARIABLE : <Variable Name> [:<Data Type of the Variable>]
```

Where,

<Variable Name> is the name of the Static Variable

<Data Type of the Variable> is the Data type of the Static Variable.

Datatype is optional. If datatype is specified a separate Variable definition is not required (these are considered as inline variables). If data type is not specified the interpreter will look for a variable definition with the name specified.

Example:

The static variable 'Sample Static Var' retains the value between successive calls to Function 'Sample Function'

```
[Function : Sample Function]
   Static Variable : Sample Static Var: Number
```

Return value specification

If a Function returns a value to the caller, then its data type is specified by using 'RETURNS' statement.

Syntax

```
      RETURNS: <Data Type of the Return Value>
```

Where,

<Data Type of the Return Value > is the Data type of the return value of the Function

Example:

The Function 'FactorialOf' returns value of type 'Number' to the caller of the Function

```
[Function : FactorialOf]
   Parameter : InputNumber: Number
   Returns     : Number
   Variable  : Factorial : Number
```

Object specification

Function will inherit the Object context of the caller. This can be overridden by using the attribute Object for function definition. This now becomes the current object for the function.

Syntax

```
      Object:  <ObjType>:<ObjIdValue>
```

Where,

<ObjType> is the type of the object and **<ObjIdValue>** is the unique identifier of the object.

Example:

The Function 'Sample Function' will be in the context of the Ledger 'Party'

```
[Function : Sample Function]
   Object : Ledger : "Party"
```

3.3 Procedural Block

This block contains a set of statements. These statements can either be a programming construct or can be an Action specification. Every statement inside the procedural block has to be uniquely identified by a label specification

Syntax

```
      LABEL SPECIFICATION : Programming Construct
             Or
      LABEL SPECIFICATION : Action: Action Parameter
```

Example:

The Function ‘DispStockSummary’ is having two Actions with Label.

```
[Function : DispStockSummary]
   01: Display: Stock Summary
   02: Display: Stock Category Summary
```

4. Programming Constructs-In Function

4.1 Conditional Constructs

IF–ENDIF

The ‘IF–ENDIF’ statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically two-way decision statement and is used in conjuncti with an expression. Initially expression will be evaluated and based on the whether expression is true or false, it transfers the execution flow to a particular statement.

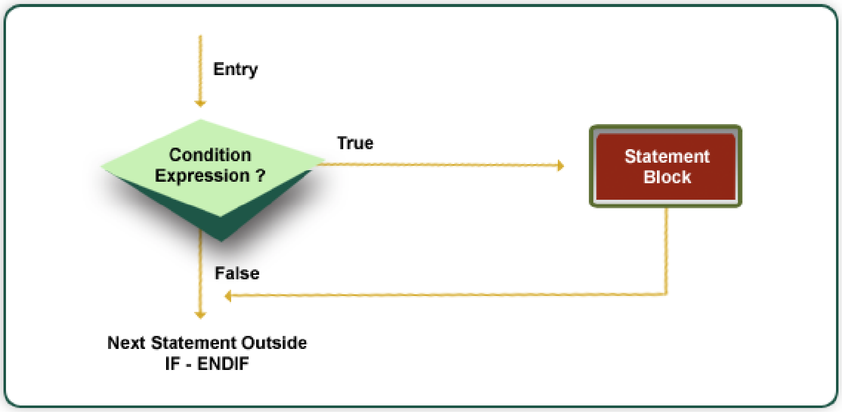


Figure 1.1 Flow Chart for IF – ENDIF

Syntax

```
       IF : <Condition Expression>
            STATEMENT 1
               …
            STATEMENT N
            ENDIF
```

Example:

If Function parameter sent to Function ‘FactorialOf’ is less than zero then it is multiplied by -1 to find the absolute value.

```
[Function : FactorialOf]
   Parameter      : InputNumber : Number
   Returns         : Number
   Variable        : Counter   : Number
   Variable        : Factorial : Number
   1 : SET   : Counter   : 1
   2 : SET   : Factorial: 1
   3 : IF ##InputNumber < 0
   4 :      SET   : InputNumber: ##InputNumber * -1
   5 : END IF
   6 : WHILE: ##Counter <= ##InputNumber
   7 :      SET    : Factorial: ##Factorial * ##Counter
   8 :      SET    : Counter : ##Counter + 1
   9 : END WHILE
   10: RETURN ##Factorial
```

DO–IF

When a IF-ENDIF statement block contains only one statement, then the same can be written in single line by using DO-IF statement.

Syntax

```
      DO IF: <Condition Expression> :STATEMENT
```

Example:

If Function parameter sent to Function ‘FactorialOf’ is less than zero then it is multiplied by -1 to find the absolute value. IF - END IF statement is re written using DO - IF statement.

```
[Function : FactorialOf]
   Parameter : InputNumber : Number
   Returns   : Number
   Variable  : Counter   : Number
   Variable  : Factorial : Number
   1 : SET    : Counter   : 1
   2 : SET    : Factorial: 1
   3 : DO IF : ##InputNumber < 0 : ##InputNumber * -1
   4 : WHILE : ##Counter <= ##InputNumber
   5 :      SET    : Factorial: ##Factorial * ##Counter
   6 :      SET    : Counter  : ##Counter + 1
   7 : END WHILE
   8 : RETURN ##Factorial
```

IF–ELSE–ENDIF

The IF–ELSE–ENDIF statement is an extension of the simple IF-ENDIF statement. If condition expression is true, then true block statement(s) are executed; otherwise the false block statement(s) are executed. In either case, either true block or false block will be executed, not both.

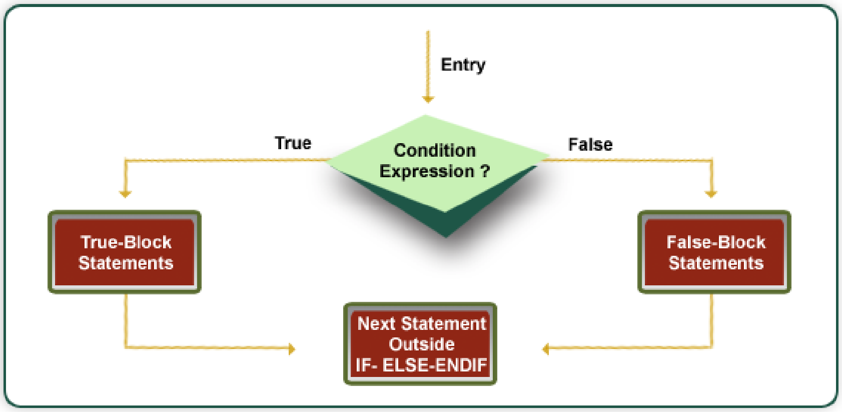


Figure 1.2 Flow Chart for IF – ELSE - ENDIF

Syntax

```
     IF : <Condition Expression>
          STATEMENT 1
              …
          STATEMENT N
          ELSE
          STATEMENT 1
              …
          STATEMENT N
          ENDIF
```

Example:

Finding greatest of three numbers

```
[Function : FindGreatestNumbers]
   Parameter   : A : Number
   Parameter   : B : Number
   Parameter   : C : Number
   RETURNS             : Number
   01 : IF     : ##A  >   ##B
   02 :     IF     :   ##A > ##
   03 :           RETURN : ##A
   04 :     ELSE
```

```
          RETURN : ##C
    END IF
ELSE
    IF     : ##B >  ##C
RETURN : ##B
    ELSE
          RETURN : ##C
    END IF
END IF
```

4.2 Looping Constructs

WHILE – ENDWHILE

In looping, a sequence of statements is executed until some conditions for termination of the loop are satisfied. A typical loop consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement checks condition and then directs the repeated execution of the statements contained in the body of the loop.

The WHILE –ENDWHILE is an entry controlled loop statement. The condition expression is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the statements within the body, the condition expression is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the condition expression finally becomes False and the control is transferred out of the loop.

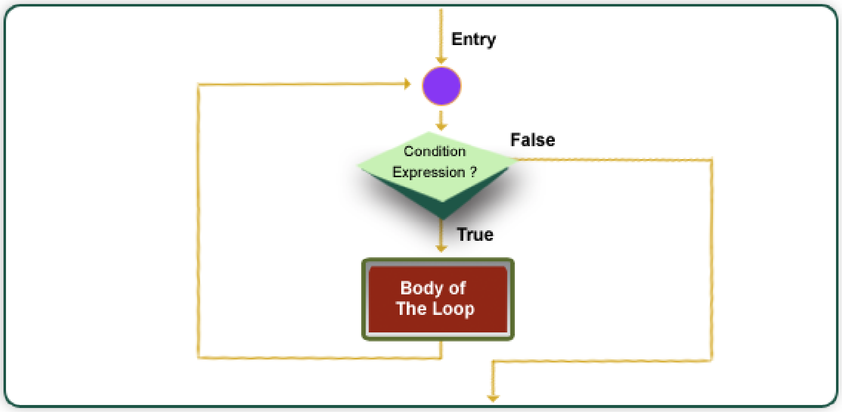


Figure 1.3 Flow Chart for WHILE – ENDWHILE

Syntax

```
WHILE : <Condition Expression>
        STATEMENT 1
         …
        STATEMENT N
ENDWHILE
```

Example:

The Function ‘FactorialOf’ repeats statements 4 and 5 till given condition is satisfied.

```
[Function : FactorialOf]
   Parameter  : InputNumber : Number
   Returns    : Number
   Variable   : Counter : Number
   Variable   : Factorial : Number
   1 : SET    : Counter : 1
   2 : SET    : Factorial: 1
   3 : WHILE  : ##Counter <= ##InputNumber
   4 :      SET   : Factorial: ##Factorial * ##Counter
   5 :      SET   : Counter : ##Counter + 1
   6 : END WHILE
   7 : RETURN ##Factorial
```

WALK COLLECTION – END WALK

If a Collection has ‘n’ Objects then WALK COLLECTION – ENDWALK will be repeated for ‘n’ times. Body of the loop is executed for each object in the collection, making it the current context.

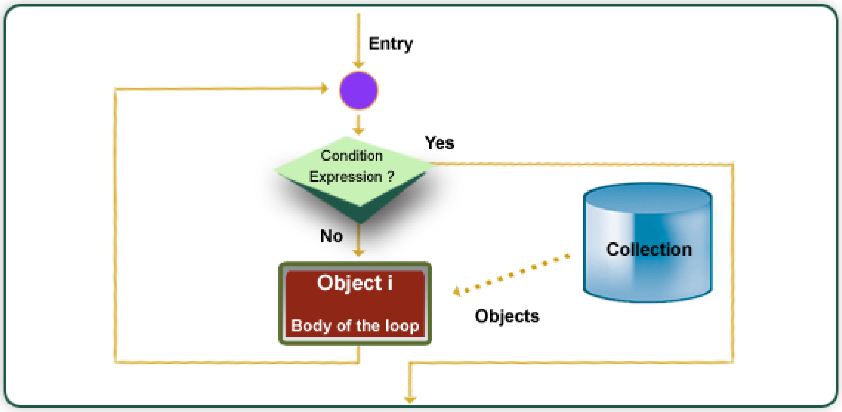


Figure 1.4 Flow Chart for WALK COLLECTION – ENDWALK

Syntax

```
      WALK COLLECTION : <Collection Name>
                        STATEMENT 1
                        …
                        STATEMENT N

      ENDWALK
```

Example:
Walking over all the Vouchers and counting the same

```
[Collection : Vouchers Coll]
   Type                    : Voucher
[Function : CountVouchers]
   Returns                 : Number
   Variable                : Count : Number
   001 : SET               : Count : 0
   002 : WALK COLLECTION : Vouchers Coll
   003 :     SET   : Count       : ##Count + 1
   004 : END WALK
   005 : RETURN            : ##Count
```

4.3 Control Constructs

Loops perform a set of operations repeatedly until the condition expression satisfies given condition or Collection is exhausted. Sometimes, when executing the loop, it becomes desirable to skip the part of the loop or to exit the loop as a certain condition occurs or to save the current state and return back to the current state later.

BREAK

When a Break statement is encountered inside the loop, the loop is immediately exited and control is transferred to the statement immediately following the loop. BREAK statement can be used inside the WHILE – END WHILE and WALK COLLECION – END WALK. When loops are nested, the Break would only exit from the loop containing it.

Syntax

```
      BREAK
```

Example:
In Function 'PrintNumbers' loop is running from 1 to 10. But because of BREAK statement loop will be terminated as counter reaches the 6.

```
[Function : PrintNumbers]
   Variable   : Counter : Number
   1 : SET    : Counter : 1
   2 : WHILE  : ##Counter < = 10
   3 :      LOG    : ##Counter
   4 :      IF     : ##Counter > 5
   5 :            BREAK
```

```
6 :       END IF
7 :       SET      : Counter : ##Counter + 1
8 : ENDWHILE
9 : RETURN
```

CONTINUE

In some scenarios instead of terminating the loop, loop needs to be continued with next iteration after skipping any statements in between. For this purpose CONTINUE statement can be used. CONTINUE statement can be used inside the WHILE – END WHILE and WALK COLLECION – END WALK.

Syntax

```
      CONTINUE
```

Example:

Function to Count total number of Journal Vouchers

```
[Collection : Vouchers Coll]
   Type                          : Voucher
[Function : CountJournal]
   Returns                : Number
   Variable               : Count : Number
   01 : SET              : Count : 0
   02 : WALK COLLECTION: Vouchers Coll
   03 :     IF : NOT $$IsJournal:$VoucherTypeName
   04 :           CONTINUE
   05 :     ENDIF
   06 :     Count            : ##Count + 1
   07 : END WALK
   08 : RETURN           : ##Count
```

START BLOCK – END BLOCK

START BLOCK –- END BLOCK has been introduced to save the current state and execute some actions within the block and return back to the original state. This is handy in cases where the Object context needs to be temporarily switched for the purpose of executing some actions. Current source and target object contexts are saved and the further statements within the START and END BLOCK section gets executed and once END BLOCK is encountered, the Object context is restored back to the original state.

Syntax

```
START BLOCK
     Block Statements
END BLOCK
```

Example:

```
10 : WALK COLLECTION         : EDCollDetailsExtract
11 :   INSERT COLLECTION OBJECT: InventoryEntries
12 :   SET : QtyVar : $$String:$Qty + " Nos"
13 :   SET : AmtVar : $$String:$Amt
14 :   START BLOCK
15 :        SET OBJECT
16 :        SET VALUE : ActualQty : $$AsQty:##QtyVar
17 :        SET VALUE : BilledQty : $$AsQty:##QtyVar
18 :        SET VALUE : Amount: $$AsAmount:##AmtVar
18A:   END BLOCK
19 :   SET TARGET : ..
20 :   SET VALUE  : StockItemName: $Item
21 : END WALK
```

;; For Explanation on Object context, Source Object and Target Object,
;; Set Target, Set Object, please refer Topic Function Execution - Object Context

In the above code snippet, `EDCollDetailsExtract` collection is being walked over and the values for Objects within Voucher Entry are being set.

RETURN

This statement is used to return the flow of control to the calling program with or without returning a value. When return is used the execution of the function is terminated and the calling program continues from where it had called the function.

Syntax

```
RETURN : <value expression>
```

Where,

<value expression> is optional ie it can either return a value or return void.

Example:

The Function 'FactorialOf' returns factorial of number

```
[Function : FactorialOf]
   Parameter  : InputNumber : Number
   Returns      : Number
   Variable   : Counter   : Number
   Variable   : Factorial : Number
   1 : SET    : Counter : 1
   2 : SET     : Factorial: 1
   3 : WHILE  : ##Counter <= ##InputNumber
   4 :      SET   : Factorial : ##Factorial * ##Counter
   5 :      SET   : Counter : ##Counter + 1
   6 : ENDWHILE
   7 : RETURN : ##Factorial
```

5. Calling a Function

A Function can be invoked in two ways.

1. By using a "CALL" action –This is mainly used when the function does not return a value. It only performs a certain functionality.
2. By Using the prefix $$ with the function name within a value expression-This is used when return value is expected from the function execution. This value is used in the value expression of the calling program.

5.1 Using Action – CALL

Action CALL can be used to call a function. It can be invoked a Key, Menu Item or Button.

Syntax

```
      CALL  : <Function Name> [: <Parameter List>]
```

Where,

<Function Name> is the name of a user defined function.

<Parameter List> is the parameters accepted by the function.

Example:

Calling the Function as a procedure with CALL action

```
[#Menu : Gateway of Tally]
   Button : Call Function

[Button : Call Function]
   Key     : Alt + F9
   Title   : "Call Function"
   Call    : DispStaturoryRpts
```

5.2 Using – Symbol Prefix $$

Function can be executed by prefixing it with symbol ‘$$’. This can be used inside a value expression or as a value for Set As attribute of the field. The value returned from the function is used.

Syntax

```
      $$FunctionName :<Parameter List>
```

Where,

<Function Name> is the name of a user defined function.

<Parameter List> is the parameters accepted by the function.

- *During Tally Startup Tally executes a function with the name “TallyMAIN”*
- *Internal functions always override if both exists in same name.*

Example:

Calling the User Defined Function at Field using $$

```
[Field : Call Function]
   Use          : Number Field
   Set as     : $$FactorialOf:#InputFld
```

6. Function Execution – Object Context

We all are aware that in TDL any function, method or formula gets evaluated in the current object context. All platform defined functions will be executed with current object and requestor context.

With the introduction of User Defined Functions another type of context is introduced. This is known as the target context.

6.1 Target Object Context

Target Context mainly refers to the object context which can be set inside the function which allows the function to perform manipulation operations for that object ie alteration and creation. The object context of the caller and target object context can be different. It will now be possible to obtain the values from caller object context and alter the values of the target object with those values.

User has option to override the context within the function later or use the same context being passed. He can change the current and target object at any point and also switch target and current object to each other.

The fuction $$TgtObject is used to evaluate the value of expression in the context of target object.

6.2 Parameter Evaluation Context

It is important to note that the parameter values which are passed to the functions are always evaluated in context of the caller. Parameter specification within the functions is just to determine the datatype, order and no of parameters. These are basically the placeholders for values passed from caller object context. The target object context or context switch within the function does not affect the initial values of the parameters. Later within the function these values can be altered just like ordinary variables.

6.3 Return Value Evaluation

We have already discussed above that function can return a value. This can be specified by the function by indicating the data type of the value it returns, no specification assumed as a void function (a function which does not return a value). Each statement (Actions discussed in the next section) used can return a value. This depends on the individual action. Some actions may not return value. The data type of return value is also predefined by the action. Return value of the last action executed can be extracted using an internal function '$$LastResult'. Any error messages generated from the system can be obtained with $$LastError. This can only be used to determine the result of the intermediate statements used within the function.The final value which is to be returned from the function has to be explicitly given using the RETURN construct discussed in previous section.

7. Valid Statements inside a Function

The statements used inside the procedural block of a function can either be a

- Programming Construct as discussed in the previous sections
- It can be a TDL action

There have been major changes in some actions to work especially with functions. Some new actions have been introduced as well. Let us now discuss the various Actions used inside functions

7.1 Actions for Variable Manipulation

TDL provides various new action that can be used inside User Defined Functions.

SET

This action is used to assign a value for a variable.

Syntax

```
SET : <VariableName> : <Value Expression>
```

Where,

<Variable Name> is the variable for which value needs to be assigned

<Value Expression> is the formula evaluating to value

Example:

```
[Function : FactorialOf]
   Parameter : InputNumber : Number
   Returns     : Number
   Variable  : Counter   : Number
   Variable  : Factorial : Number
   1 : SET    : Counter   : 1
   2 : SET    : Factorial: 1
   3 : WHILE : ##Counter <= ##InputNumber
   4 :      SET     : Factorial : ##Factorial * ##Counter
   5 :      SET     : Counter : ##Counter + 1
   6 : ENDWHILE
   7 : RETURN: ##Factorial
```

EXCHANGE

This Action is used exchange (swaps) the value of two variables. But only Variables of the same Data type can be exchanged.

Syntax

```
       EXCHANGE : <First Variable Name> : <Second Variable Name>
```

Where,

<First Variable Name> and <Second Variable Name> are the Variables whose values needs to be swapped.

Example:

```
      01: EXCHANGE:Var1:Var2
```

In the above statement both variables are of type Number and their values are swapped.

INCREMENT

This Action is used to increment the value of a Variable by 1. INCREMENT is used inside the loop to increment value of the control variable by one.

Syntax

```
       INCREMENT : <Variable Name>
```

Where,

<variable Name> is the name of the Variable whose value need to be incremented by 1.

Example:

```
[Function : FactorialOf]
   Parameter    : InputNumber : Number
   Returns       : Number
   Variable      : Counter : Number
   Variable      : Factorial : Number
   1 : SET       : Counter : 1
   2 : SET       : Factorial: 1
   3 : WHILE     : ##Counter <= ##InputNumber
   4 :      SET       : Factorial : ##Factorial * ##Counter
   5 :      INCREMENT: Counter
   6 : ENDWHILE
   7 : RETURN    : ##Factorial
```

DECREMENT

This Action is used to decrement the value of a Variable by 1. DECREMENT is used inside the loop to decrement the value of the control variable by one.

Syntax

```
       DECREMENT : <Variable Name>
```

Where,

<Variable Name> is the name of the Variable whose value need to be decremented by 1.

Example:

In Function 'PrintNumbers' loop is running from 10 to 1.

```
[Function : PrintNumbers]
   Variable       : Counter : Number
   1 : SET         : Counter : 10
   2 : WHILE      : ##Counter > 0
   3 :      LOG        : ##Counter
   4 :      DECREMENT : Counter
   5 : ENDWHILE
   6 : RETURN
```

7.2 Action Enhancements and New Actions

The global actions are enhanced, so that they can be called from the User Defined Functions. Some new actions are also introduced.

Global Actions- Alter / Execute / Create

These are the global actions meant for the purpose of opening a report in the modes specified. The return value of these actions is FALSE if user rejects the report, else TRUE if he accepts it.

Syntax

```
      Display/Alter/Execute/Create: Report Name
```

Example:

The Function ‘CreateReport’ opens Leadger Creation screen

```
[Function : CreateReport]

    01 : Create : Ledger
```

Global Actions – MENU, DISPLAY

These global actions are used to invoke a menu or a report in display mode. The return value of these actions is TRUE if Ctrl+Q is used to reject the report (i.e. via Form Reject To Menu action). It returns FALSE when user uses Esc to reject the report (i.e. via Form Accept action). For menu this is only applicable if it is the first in the stack.

Syntax

```
      Menu    : <Menu name>
      Display : <ReportName>
```

Example:

The Function ‘DispPaySheet’ opens Pay Sheet and by pressing Escape it will pop up the ‘Statements of Payroll’ Menu.

```
[Function : DispPaySheet]

   01 : Menu    : Statements of Payroll

   02 : Display : Pay Sheet
```

MSG BOX

This action is used to Display a message box to user. It comes back to the original screen on the press of a key. This is can be used by the programmers to display intermediate values of variables during calculations thus helping in error diagnosis.

Syntax

```
       MSG BOX: <Title Expression>:<Message Expression>:<GreyBack Flag>
```

Where,

<Title Expression> is the value is displayed on the title bar of the message window.

<Message Expression> is the actual message which is displayed in the box. This can be an expression as well i.e. the variable values can be concatenated and displayed along with the message in the display area of the box.

<GreyBack Flag> indicates if the background window to be greyed out during message display. It takes two values ie YES/NO

Example:

```
01: MSGBOX:"Return Value":##Factorial
```

QUERY BOX

This action is used to Display a confirmation box to user and ask for an yes/no response.

Syntax

```
QUERY BOX: <Message Expression>:<GreyBack Flag>:<EscAsYes>
```

Where,

<Message Expression> is the message which is displayed inside the box. This can be an expression.

<GreyBack Flag> Same as in msg box .

<Escape as Yes>This is a flag which indicates the response when the user presses ESC key. This can be specified as YES/NO. A YES value for this flag indicates that the response should be treated as YES on press of an ESC key.

Progress Bar Actions

Sometimes a Function may take some time to complete the task. It is always better to indicate the user that the task is occurring, how long the task might take and how much work has already been done. One way of indicating the amount of progress, is to use an animated image. This can be achieved by using following Actions.

- START PROGRESS
- SHOW PROGRESS
- END PROGRESS

START PROGRESS

This Action setup the Progress Bar by mentioning total number of steps involved in the task. In addition to this, Title, Sub Title and Subject of the Progress Bar also can be given as parameter.

Syntax

```
START PROGRESS : <Number of steps> :< Title> [:< Sub Title> :< Subject>]
```

Where,

<Number of steps> denotes the whole task quantified as a number,

<Title>,**<Sub Title>** and **<Subject>** Shows the Title, Sub Title and Subject of Progress Bar respectively.

Example:

```
START PROGRESS: ##TotalSteps:"TDS Migration":+
                  @@CmpMailName:"MigrationgVouchers.."
```

SHOW PROGRESS

This Action shows the current status of the task to the user.

Syntax

```
      SHOW PROGRESS : <Number of Steps Completed>
```

Where,

<Number of Steps Completed> is a number denotes the amount of work completed.

Example:

Progress Bar showing the progress of the task

```
      SHOW PROGRESS:##Counter
```

END PROGRESS

When a task is completed, Progress Bar can be stopped by using Action END PROGRESS. This Action does not take any parameter.

Syntax

```
      END PROGRESS
```

LOG

During expression evaluation, intermediated values of the expression can be passed to calculator window and a log file 'tdlfunc.log' inside the application directory. This is very much helpful for debugging the expression. By default logging is enabled inside the function.

Syntax

```
       LOG : < Expression>
```

Where,

<Expression> whose value need to be passed to the calculator window.

Example:

While finding the factorial of a number, intermediated values are outputted to Calculator window using LOG action

```
[Function : FactorialOf]
   Parameter  : InputNumber : Number
   Returns    : Number
   Variable   : Counter : Number
   Variable   : Factorial : Number
   1 : SET    : Counter : 1
   2 : SET    : Factorial: 1
   3 : WHILE  : ##Counter <= ##InputNumber
   4 :      SET     : Factorial : ##Factorial * ##Counter
```

```
5 :       SET     : Counter : ##Counter + 1
5a:       LOG    : ##Factorial
6 : ENDWHILE
7 : RETURN : ##Factorial
```

SET LOG ON

While debugging a Function, some times it is required to conditionally Log the values of an expression. If logging is stopped, then logging can be re-started based on the condition Action SET LOG ON. This Action does not require any parameter.

Syntax

```
      SET LOG ON
```

SET LOG OFF

This Action is used in conjunction with SET LOG ON. Log can be stopped by Action SET LOG OFF. This Action does not require any parameter.

Syntax

```
      SET LOG OFF
```

SET FILE LOG ON

This Action is similar to SET LOG ON. SET FILE LOG ON is used to conditionally Log the values of an expression to log file ‘tdlfunc.log’. This Action does not require any parameter.

Syntax

```
      SET FILE LOG ON
```

SET FILE LOG OFF

This Action is used in conjunction with SET FILE LOG ON. Logging the file ‘tdlfunc.log’ can be stopped by Action SET LOG OFF. This Action does not require any parameter.

Syntax

```
      SET FILE LOG OFF
```

Actions – For Object and Context Manipulation

As we have already seen in the previous sections functions can operate on 3 object contexts. ie Requestor, Current Object and Target object context. When a function is invoked the target object context will be same as the current object context of the caller, ie the target object will be set to the current object.

Here we will discuss the various actions for manipulation of Object and Context.

NEW OBJECT

Creates a New object from object specification and sets it as target object. This Action takes only Primary Object as Parameter.

Syntax

```
NEW OBJECT: <ObjType>:<ObjIdValue>
```

Where

<ObjType> is the type of the object to be created and

<ObjIdValue> is the unique identifier of the object. If this is an existing object in DB then the further manipulations are performed on that object else it creates a new object altogether.

Example:

```
01: NEW OBJECT:Stock Item :”My Stock Item”
```

This creates a new object in memory for Stock Item and sets it as the target object. Later by using other methods of this target object can be set and saved to the Tally DB.

INSERT COLLECTION OBJECT

Inserts the new object of the type specified in collection and makes it as current target object. This object is inserted into the collection at the end. This Action will take only Secondary Collection as parameter.

Syntax

```
INSERT COLLECTION OBJECT : <CollectionName>
```

Where,

<CollectionName> is the name of the Secondary Collection

Example:

```
01: INSERT COLLECTION OBJECT: Ledger Entries
```

Insets a new object Ledger Entries in memory under Voucher and sets it as the target object. Later by using other methods of this target object can be set and saved to the Tally DB.

SET VALUE

Sets value of a method for the target object. The value formula is evaluated with respect to the current object context. This can use the new method formula syntax. Using this it is possible to access any value from the current object.

Syntax

```
SET VALUE: <Method Name>[: <Value Formula>]
```

Where,

<Method Name> is the name of the method and

<Value Formula> is the value which needs to be set to the method. It is optional. So that if the second parameter is not specified, it searches for the same method in the context object and the value is set based on it. If the source method name is same as in Target Object, then the Source Object method name is optional.

Example:

```
      01: SET VALUE : Ledger Name : $LedgerName
```

Or

```
            01: SET VALUE : Ledger Name
```

The above statements setting the values of Ledger Entries Object from the current Object context.

Example:

```
      02     : WALK COLLECTION: Vouchers of My Objects
            03     :     NEW OBJECT: Voucher
```

;; Since the methods Date, VoucherTypeName are same in the source object and target object, they are not specified again as SET VALUE : DATE : $Date.

```
            04     :     SET VALUE: Date
            05     :     SET VALUE: VoucherTypeName
```

Example:

Party 1 is a ledger under Group North Debtor and Party 2 is a Ledger under Group South Debtor. After executing the following function Party 2 will also come under Group South Debtor.

```
[Function : Sample Function]
   Object          : Ledger : "Party 1"
   01 : NEW OBJECT : Ledger : "Party 2"
```

;; absence of Value expression will assume that same method to be copied from source

```
   02 : SET VALUE  : Parent
   03 : ACCEPT ALTER
```

RESET VALUE

Sets the value of the method using the Value Formula. If Value Formula is not specified it sets the existing value to null.

Syntax

```
      RESET VALUE : MethodName [: Value Formula]
```

Where,

<Method Name> is the name of the method and

<Value Formula> is an optional parameter and if it is used, it will reset the value of the method.

Example:

```
      01     : SET VALUE   : Ledger Name : $LedgerName
      02     : RESET VALUE : Ledger Name : “New Value”
```

In the above code snippet RESET VALUE resets the value of the method ‘Ledger Name’

CREATE TARGET / ACCEPT CREATE

Accepts the Target Object to Company Data base. That is it saves the target object to the Database. This creates a new object in the database if it does not exist else results in an error.

Syntax

```
        CREATE TARGET / ACCEPT CREATE
```

SAVE TARGET / ACCEPT

Accepts the Target Object to Company Tally DB. If another object exists in Tally DB with same identifier then the object is altered else a new object is created.

Syntax

```
        SAVE TARGET / ACCEPT
```

ALTER TARGET / ACCEPT ALTER

Accepts the Target Object to Company DB. Alters an exiting object in DB. If the object does not exists it results in error.

Syntax

```
      ALTER TARGET / ACCEPT ALTER
```

SET OBJECT

Sets the current object with the Object Specification. If no object specification is given the target object will be set as the current object. Only Secondary Object can be used along with this Action.

Syntax

```
      SET OBJECT [: <Object Spec>]
```

Where,

<Object Spec> is the name of the Secondary Object.

Example:

```
[Function : Sample Function]
   Object           : Ledger : "My Test Ledger"
   01 : LOG         : $Name
   02 : SET OBJECT  : BillAllocations[1]
   03 : LOG         : $Name
   04 : SET OBJECT  : ..
   05 : LOG         : $Name
```

Initially the context object is Ledger so $Name give the name of Ledger. By Using ‘SET OBJECT’ current Object is changed to first Bill allocation. So second $Name is giving the Bill name. Fourth line changes current Object back to Ledger using doted notation.

SET TARGET

Sets the target object with the Object Specification. If no object specification is given the current object will be set as the target object.

Syntax

```
        SET TARGET: <Object Spec>
```

Where,

<Object Spec> is the name of the Object

Example:

```
        01 : SET TARGET :  Group
```

This sets the target object as Group Object. Later by using other methods of this target object can be set and saved to the Tally DB.

Usage of Object manipulation Actions:

Duplicating all payment Vouchers

```
[Function : DuplicatePaymentVouchers]
;;Process for each Payment Voucher
   01 : WALK COLLECTION : My Vouchers
;; Create new Voucher Object as Target Object
   02 : NEW OBJECT : Voucher
;;For New Object set methods from the First Object of the Walk Collection i.e from Current  Object
   03 : SET VALUE : Date : $Date
   04 :     SET VALUE : VoucherTypeName : $VoucherTypeName
   05 : S   ET VALUE : Narration : $Narration + " Duplicated"
;; Walk over Ledger Entries of the current Object
   05a:     WALK COLLECTION : LedgerEntries
;;Insert Collection Object to the Target Object and become present Taget Object
   06 :           INSERT COLLECTION OBJECT : Ledger Entries
;;Set the Values of the Taget Object's Method from Current Objects Methods
   07 :           SET VALUE : Ledger Name : $LedgerName
   08 : SET VALUE : IsDeemedPositive : $IsDeemedPositive
   09 :           SET VALUE : Amount : $Amount
;;Set the Voucher Object as Target, (which is 1 level up in the hierarchy) as Voucher is already having
;;Object spec
   10 :     SET TARGET : ..
   11 : END WALK
;;Save the Duplicated Voucher to the DB.
   15 : CREATE TARGET
   16 : ENDWALK
17 : RETURN
```

What's New in TDL for Tally.ERP 9 Release 3.61

Emerging needs from the development partners drives the TDL language enhancements from time to time.

In Release 3.61, following language enhancements have been incorporated:

1. Action Enhancements

1.1 TDL Action Browse URL Ex

As we are already aware that there is an action **Browse URL/ Execute Command,** which is used to open a web, page or invoke an external application. At times, we may come across situations where subsequent actions are dependent on the completion of the previous action i.e., the closure of external application. For example, the current action is used to execute an external file, which unzips/ extracts various other files. The subsequent actions use these extracted files to process further. In such circumstances, Browse URL when used, will trigger the requested application and continue executing the subsequent actions.

Hence, in order to bring sync between the calling and the called application, we have introduced a new action B**rowse URL Ex** or **Execute Command Ex**. Action Browse URL Ex, when triggered waits till the external application is closed and then allows the application to resume with the subsequent action.

Similar to Action Browse URL, Action Browse URL Ex can be used to:

- Open an external file with its associated application
- Open a folder in explorer
- Open a Web Page in the browser
- Run an executable application

Notes

BrowseURLEx *is useful for URL, folder and executable without extension (e.g. tally instead of tally.exe) and it has same behaviour as Browse URL.*

BrowseURLEx is useful for URL, folder and executables without extension (e.g. tally instead of tally.exe) and it has same behavior as Browse URL.

Syntax

```
    BrowseURL Ex: <URL/File path/executable file path/folder path> +
                                        [:<Command line Parameters>]
```

Where,

<URL/File path/Executable file path> can be:

- a URL which can be opened in a browser
- a file which is to be opened in its associated application,
- an executable file which is to be run/execute
- folder which is to be opened in the explorer

<Command line Parameters> this is an optional parameter. For example: Zip application may need parameter “d” to decompress, “c” to compress etc.

Example:1

To open a URL in browser

```
  Action: BrowseURL Ex: “http://google.com”
```

Example:2

To open an executable and wait for it to complete

In the given example, 7zip is to be opened and wait until it finishes i.e., the running application first wait for 7zip.exe to finish decompressing of software.7z and then it will proceed further.

```
    Action: BrowseURL Ex: “C:\7zip.exe”: “d software.7z”
```

Example:3

To open a folder

```
      Action: BrowseURL Ex : “C:\abc”
```

Example:4

To open a pdf file in pdf reader

```
    Action: BrowseURL Ex : “C:report.pdf\”
```

In this example, report.pdf will be opened in default PDF reader of the system. This can be useful while reading a report, which is exported in PDF format.

2. Function Enhancements

2.1 Function FileReadRaw

We had a function FileRead to read the contents from a text file which was designed to ignore quotes, comments, spaces, etc. while reading the entire line.

Now, a new function **FileReadRaw** is introduced similar to Function FileRead except that the Function FileReadRaw can read lines with

- Quotes
- Comment characters (; /* */)
- Spaces & Tabs

Syntax

```
      $$FileReadRaw[:<Number>]
```

Where,
<Number> denotes number of characters to be read.

Example:

```
[Function: Test FileReadRaw]

   Variable: GetPath: String
   Variable: Get_DownLoad_FileLine  : String

     000 : Set : GetPath : "C:\TextSource.txt"
     010 :   Open File: ##GetPath: Text : Read
     020 :   While: (TRUE)
     030 :    Set: Get_DownLoad_FileLine: $$FileReadRaw
```

By using this, we can read line with quotes, comment characters, line continuation, spaces, tabs, etc.

If **$$FileReadRaw** is specified with the parameter, the behavior is same as Function FileRead.
If **$$FileReadRaw** is specified without parameters, the entire line is being read without ignoring quotes, spaces, etc.

3. Enhancements in Release 3.6

Introduction

The TDL language is enriched with new capabilities based on the emerging needs from time to time. In Release 3.6, following language enhancements have been incorporated:

3.1 Collection Enhancements

New methods for File Properties – LastModifiedDate and LastModifiedTime

In Collection definition, Directory as a Data Source was supported in Release 3.0 where the properties of the file i.e., Name, FileSize, IsDirectory, IsReadOnly and IsHidden were supported as the Methods. In Release 3.6, we have introduced two new methods, to extract additional file properties in Tally, called **LastModifiedDate** and **LastModifiedTime**.

This can be very useful in Integration scenarios with Tally using external Files. Prior to importing a file the last imported date and time can be validated against Last Modified Date and Time of the File prior to importing from a file.

LastModifiedDate – This returns the date on which the file was altered last. The format supported is *dd-mmm-yyyy*.

Syntax

```
      $LastModifiedDate
```

Example:

```
[Collection: ListofFiles]
   Data Source  : Directory : "C:\"
   Format       : $Name, 25
   Format       : $FileSize, 15
   Format       : $IsReadOnly, 15
   Format       : $LastModifiedDate, 15
```

$LastModifiedDate, here would return the date on which the file was modified last for e.g., **27-May-2012**

LastModifiedTime – This returns the time at which the file is altered last. The format supported is *hh:mm:ss (24 hours)*

Syntax

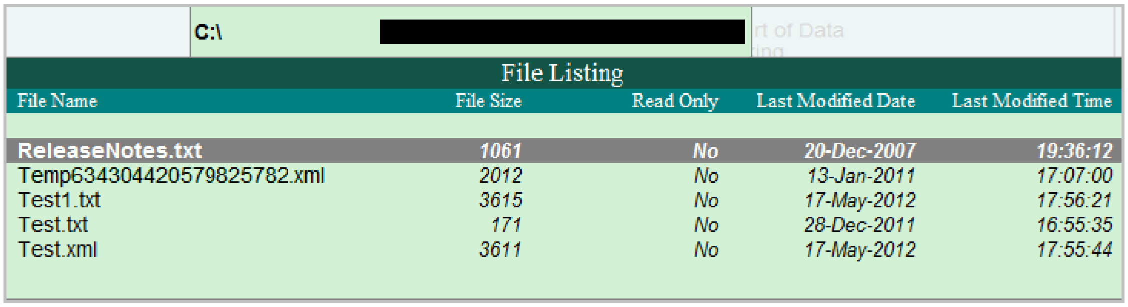
```
      $LastModifiedTime
```

Example:

```
[Collection: ListofFiles]
   Data Source  : Directory : "C:\"
   Format       : $Name, 25
   Format        : $FileSize, 15
   Format        : $IsReadOnly, 15
   Format        : $LastModifiedDate, 15
   Format        : $LastModifiedTime, 15
```

$LastModifiedTime, here would return the time at which the file was modified last for **e.g., 16:28:18**

The following screen capture displays the last modified date and last modified time in the table:



C:\ [redacted]

File Listing

File Name	File Size	Read Only	Last Modified Date	Last Modified Time
ReleaseNotes.txt	1061	No	20-Dec-2007	19:36:12
Temp634304420579825782.xml	2012	No	13-Jan-2011	17:07:00
Test1.txt	3615	No	17-May-2012	17:56:21
Test.txt	171	No	28-Dec-2011	16:55:35
Test.xml	3611	No	17-May-2012	17:55:44

3.2 Action Enhancements

TDL Action Execute TDL

TDL action **Execute TDL** is introduced to programmatically load TDL, perform some actions and subsequently, unload the TDL or Keep the TDL loaded for the current Tally session.

This can prove to be very useful in cases, where user needs to programmatically associate a TDL on the fly for performing some operations.

Syntax

```
     EXECUTE TDL  : <TDL/TCP File Path>[ : <Keep TDL Loaded Flag> : +
                                          <Action> : <Action Parameters>]
```

Where,

- **TDL/TCP File Path** specifies the path of the TDL/TCP File to be loaded dynamically
- **Keep TDL Loaded Flag** is the Logical Flag to determine if the TDL should be kept loaded after the said action is executed. If the value is set to Yes, then the TDL will continue to be loaded else the Executed TDL will be unloaded after the execution of the specified action.
- **Action** specifies the global Action to be performed i.e., Display, Alter, Call, etc.
- **Action Parameters** are the parameters on which the Action is performed, for e.g.,
  - If the action is Call, the Action Parameters contain the Function name along with Function Parameters, if any.
  - If the Action is Display, Alter, etc. then the Parameter should be the Report Name.

Example:

```
[Function: TDL Execution with Keep TDL Loaded enabled]
   00 : Execute TDL : "C:\Tally.ERP9\BSTitleChange.TDL":+
                               Yes: Display: Balance Sheet
   10 : Display     : Balance Sheet
```

File **BSTitleChange.TDL** contains the following lines of code:

```
[#Report: Balance Sheet]
   Title  : "TDL Executed Programmatically"
```

In the example, the TDL **BSTitleChange.TDL** which is used to change the Title of the report Balance Sheet is loaded dynamically by executing the action “Execute TDL”. The **Keep TDL Loaded Flag** is set to **Yes** in the above snippet. Based on the subsequent action Display: Balance Sheet the Balance Sheet report will be shown with a new Title. The next statement also displays Balance sheet. The title for this will again be the same i.e., the changed title, as the dynamic TDL is still loaded even after the action “Execute TDL“ has completed its execution.

```
[Function: TDL Execution with Keep TDL Loaded enabled]
   00 : Execute TDL : "C:\Tally.ERP9\BSTitleChange.TDL" : No +
                                            : Display : Balance Sheet
   10 : Display     : Balance Sheet
```

In this example, the report Balance sheet would be displayed twice and the title of the first report is “TDL Executed Programmatically” i.e., the changed title as per **BSTitleChange.TDL** whereas in the second report the title is Balance sheet since the attribute **Keep TDL Loaded Flag** is set to **No**.

3.3 Platform Functions and Variables

Function PrinterInfo

Function **PrinterInfo** is introduced to extract the settings information for any installed printer.

This function is very useful to get the information of the printer, based on which, we can determine dimensions for pre-printed invoice, etc.

Syntax

```
      $$PrinterInfo:<Printer Name>:<Information Type>
```

Where,

- **Printer Name** refers to the name of the printer for which the information is required
- **Information Types** are permissible information types like PrintSizeinInches, LeftMarginInMMs, etc.

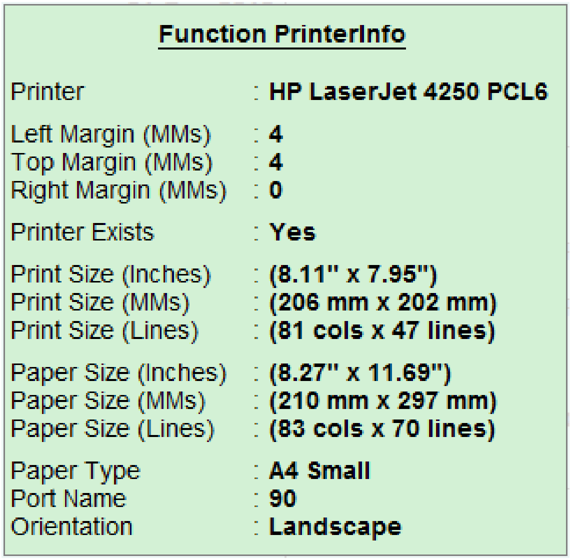
Example:

```
$$PrinterInfo:HPLaserJet4250PCL6:PrintSizeInInches
```

List of permissible Information Types are:

- **LeftMarginInMMs** - Returns Number which denotes the space to be left in the Left side of the page in Millimeters
- **TopMarginInMMs** - Returns Number which denotes the space to be left on the Top of the page in Millimeters
- **RightMarginInMMs** - Returns Number which denotes the space to be left in the Right side of the page in Millimeters
- **PrinterExists** - Returns Logical value (Yes/No) indicating whether the Printer Exists or not
- **PrintSizeInInches** - Returns the dimensions which denotes the Print Area in Inches i.e., excluding the Margins
- **PrintSizeInMMs** - Returns the dimensions which denotes the Print Area in Millimeters i.e., excluding the Margins
- **PrintSizeInLines** - Returns the dimensions which denotes the Print Area in Lines i.e., excluding the Margins
- **PaperSizeInInches** - Returns the dimensions which denotes the Paper Size in Inches which includes the Margins
- **PaperSizeInMMs** - Returns the dimensions which denotes the Paper Size in Millimeters which includes the Margins
- **PaperSizeInLines** - Returns the dimensions which denotes the Paper Size in Lines which includes the Margins
- **PaperType** - Returns the selected Type of the Paper, for e.g., A4, A5 Small, etc.
- **PortName** - Returns the Port Name configured for the Printer
- **Orientation** - Returns the Orientation Type of Paper i.e., Landscape or Portrait

The following screen capture displays the selected Printer details for all the information types:



Function PrinterInfo

Printer	: HP LaserJet 4250 PCL6
Left Margin (MMs)	: 4
Top Margin (MMs)	: 4
Right Margin (MMs)	: 0
Printer Exists	: Yes
Print Size (Inches)	: (8.11" x 7.95")
Print Size (MMs)	: (206 mm x 202 mm)
Print Size (Lines)	: (81 cols x 47 lines)
Paper Size (Inches)	: (8.27" x 11.69")
Paper Size (MMs)	: (210 mm x 297 mm)
Paper Size (Lines)	: (83 cols x 70 lines)
Paper Type	: A4 Small
Port Name	: 90
Orientation	: Landscape

Function IsInternetActive

IsInternetActive is a function which helps us determine if the Internet is currently active. This will return True If the Internet is accessible else returns False. This can be used to perform conditional operations i.e., based on the Internet Connectivity, certain actions can be triggered.

This checks if the internet is active such that the operations pertaining to connecting to the web pages, emailing, uploading files to FTP, etc. can be performed.

Syntax

```
      $$IsInternetActive
```

Example:

```
[Function: EmailifConnected]
   00: IF : $$IsInternetActive
;; Function called to Email O/s Stmts
   10:   Call : Email Outstanding Statements
   20: ENDIF
```

The objective of this example is to Email the Outstanding Statements if the Internet connectivity is available.

Function CaseConvert

Prior to this release, the function $$Upper is used to convert the string expression to upper case but there is no functions were available for other conversions like Lower case, Title Case etc. To overcome the difficulty of converting the string to Lowercase, Title case, etc. we have introduced a common new function called **$$CaseConvert**, to convert the case of given expression to the specified case format. This function will return a string expression, i.e., in the converted format.

This function is very useful when one needs to follow the case rules to display their Name of the company, Name of the bank etc.

Syntax

```
      $$CaseConvert:<CaseKeyword>:<Expression>
```

Where,

CaseKeyword can be **All Capital, Upper Case, All Lower, Lower Case, Small Case, First Upper Case, Title Case, TitleCaseExact, Normal, Proper Case** etc.

- **All Capital/UpperCase** – To convert the input expression to upper case
- **All Lower/LowerCase/SmallCase** – To convert the input expression to lower case
- **First Upper Case** – To convert the first letter of the first word in a sentence. Other characters will remain as it is.
- **TitleCase** – To convert the input expression to Title case i.e., the principal words should start with capital letters.
  - It will not convert the prepositions, articles or conjunctions unless one is the first word.
  - It will ignore a subset of words from capitalization like the, an, and, at, by, to, in, of, for, on, cm, cms, mm, inch, inches, ft, x, dt, eis, dss, with.
  - For above subset of words original strings' cases will be preserved
- **TitleCaseExact** – To convert the input expression to Title case i.e., the principal words should start with capital letters.
  - It will not convert the prepositions, articles or conjunctions unless one is the first word.
  - It will ignore a subset of words from capitalization like the, an, and, at, by, to, in, of, for, on, cm, cms, mm, inch, inches, ft, x, dt, eis, dss, with.
  - For above subset of words will be converted to small case
- **Proper Case** – To convert the input expression to Title case i.e., all the words in a sentence should start with capital letters.
- **Normal** – To preserve the input expression as it is.

Expression is any expression of type String

Example:1

To convert the expression to Upper case:

```
[Field: String Convert]

   Set as : $$CaseConvert:UpperCase:"Tally solutions Pvt. Ltd."
```

In this example, the function returns "TALLY SOLUTIONS PVT. LTD." in the field String Convert.

Example:2

To convert the expression to Lower case:

```
[Field: String Convert]
   Set as :$$CaseConvert:LowerCase:"Tally solutions Pvt. Ltd."
```

In this example, the function returns, “tally solutions pvt. ltd.” in the field String Convert.

Example:3

To convert the expression to Title Case:

```
[Field: String Convert]
   Set as :$$CaseConvert:TitleCase:"To convert the striNg to Title case"
```

In this example, the function returns “To Convert the StriNg to Title Case” in the field String Convert.

Example:4

To convert the expression to Title Case Exact:

```
[Field: String Convert]
   Set as :$$CaseConvert:TitleCaseExact:"To convert the striNg to Title
case"
```

In this example, the function returns “To Convert the String to Title Case” in the field String Convert.

Example:5

To convert the expression to First upper case:

```
[Field: String Convert]
   Set as : $$CaseConvert:FirstUpperCase:"Tally solutions pvt. ltd."
```

In this example, the function returns “Tally solutions pvt. ltd.” in the field String Convert.

Function RandomNumber

A random number is a number generated by a process, whose outcome is unpredictable, and which cannot be sub sequentially reliably reproduced. i.e., Random numbers are numbers that occur in a sequence such that, the values are uniformly distributed over a defined interval and it is impossible to predict future values based on past or present ones.

In this release a new TDL function is introduced to generate Random Numbers called **$$RandomNumber**.

In case of auditing, this can be useful for auditors who would like to pick up some vouchers randomly for the authentication.

Syntax

```
$$RandomNumber[:<MinRange>[:<MaxRange>]]
```

Where,
Min Range and **Max Range** are optional. In the absence of Max Range, Long Max is considered (i.e., (2^31) -1 i.e., 2147483647. In the absence of Min Range, ZERO is considered

We can generate random numbers in different ways:

1. **No Parameters**: Don't pass any parameters i.e., just invoke $$RandomNumber default values are assumed
2. **Only with the MinRange Parameter**: Here no need of passing Max range. In this scenario Random number were generated from given Min Range
3. **Both MaxRange and MinRange as Parameter**: In this scenario random numbers were generated for given range

Example:1

With no Parameters:

```
Set As: $$RandomNumber
```

The above code will return the Random Number between 0 and 2147483647.

Example:2

With MinRange Parameter only:

```
Set As: $$RandomNumber:9999
```

The above code returns the Random Numbers between 9999 and 2147483647, the random number is greater than or equal to 9999. Here the value of MinRange is 9999 and MaxRange is 2147483647.

Example:3

With both Parameters(MinRange and MaxRange):

```
Set As: $$RandomNumber:9:9999
```

The above code returns the Random Numbers between 9 and 9999.

Variable SVPrintOrientation

Variable **SVPrintOrientation** is introduced to set the required Printer Orientation i.e., Portrait or Landscape within a Report. It is recommended to declare a local variable within the function or report and set the variable value to avoid the system Printer Configuration changes to be effected globally.

This is useful where a Report needs to printed in a different orientation i.e., Landscape. For e.g., in Cheque Printing, one needs to print the cheques in Landscape mode whereas other reports in portrait then the user need not keep on switching the printer settings from portrait to landscape and vice versa based on report getting printing. For Cheque Printing Report, one can default Landscape Orientation.

Example:

```
[#Report: Balance Sheet]
;; Local Variable Declaration
   Variable: SVPrintOrientation  : String
   Set     : SVPrintOrientation  : "Landscape"
```

Since the variable is locally declared and updated within the Report Balance Sheet, the same will not affect the global printer settings.

4. Enhancements in Release 3.0

Introduction

TDL language is expanding day by day. With every release new capabilities are introduced in the language.

In this release, the programmable configuration support is extended to the actions 'Print Report', 'Export Report', 'Upload Report' and 'Email Report'.

A new attribute Plain XML is introduced at report level to export the report in plain XML format.

New functions $$StrByCharCode, $$InPreviewMode whereas the functions $$Inwords and $$ContextKeywords are enhanced.

New events Start Import, Import Object and End Import are introduced for Import file definition.

Enhancements in the Tally.ERP9 Release 3.0 are explained in depth in this document.

4.1 Collection Enhancements

Collection Attribute (WalkEx)

With every new release, our focus is more oriented towards bringing in performance improvements in our product. The performance improvement greatly depends on the data gathering and processing artifact Collection, which used to gather and deliver data to the presentation layers. The performance is enhanced drastically if the collection gathering happens judiciously as per its specific usage.

Sometime back, we had introduced a Collection attribute Keep Source for performance enhancement. This was used to retain the source collection gathered once with the specified Interface Object i.e., either with the current Object or its parents/owners. This drastically improved the performance in the scenarios where the same source collection is referred multiple times within the same Object hierarchy chain.

Similarly, there are scenarios where we have Union of multiple collections using the same source collection and each collection walks over its sub objects across different paths and computes/aggregates the values from sub level. In such cases, significant CPU cycles will be utilized in gathering and walking over the same Source Object along different paths more than once.
A new attribute WalkEx is introduced which when specified in the resultant collection, allows us to specify a collection list. The source collection specification is available in the resultant collection.

The collections referred in WalkEx do not have any source collection specified and contain attributes only to walk the source collection and aggregate over Sub Objects of an already gathered collection. The advantage of using WalkEx is that, all walk paths specified in the collection list are traversed in a single pass for each object in the source collection. This results in improvements in performance drastically.

Syntax

```
[Collection: <Collection Name>]
     WalkEx: <Collection Name1>, <Collection Name2>, …..
```

Where,

Collection Name1, **Collection Name2** are the collection names specifying Walk and aggregation/computation attributes.

Example:

The requirement here is to generate a Report displaying Item Wise and Ledger Wise amount totals for all Vouchers

Using the Union Approach

;;The source collection “Voucher Source” is a Collection of Vouchers

```
[Collection: VoucherSource]
   Type : Voucher
```

;;The collection using “Voucher Source” as a source collection and walking over Ledger Entries Sub Object aggregating Amount by grouping over Ledger Name

```
[Collection: Ledger Details]

   Source Collection : VoucherSource
   Walk              : AllLedgerEntries
   By                : Particulars: $LedgerName
   Aggr Compute      : Tot Amount : Sum: $Amount
   Keep Source       : ().
```

;;The collection using Voucher Source as a source collection and walking over Inventory Entries Sub Object aggregating Amount by grouping over Stock Item Name

```
[Collection: StockItem Details]
   Source Collection : VoucherSource
   Walk              : AllInventoryEntries
   By                : Particulars: $StockItemName
   Aggr Compute      : Tot Amount : Sum: $Amount
   Keep Source       : ().
```

;;The Resultant Collection which is a union of objects from the above two collections “Ledger Details” and “StockItem Details”

```
[Collection: Union LedStk Vouchers]
   Collection: Ledger Details, StockItem Details
```

As seen from the example above both the collections “Ledger Details” and “StockItem Details” are using the same Collection “Voucher Source”. We can observe that while gathering, summarizing values from the Source Collection, each object of the collection “Voucher Source” is traversed twice for aggregating objects over two different paths i.e., once for “Ledger Entries” and then again for “Inventory Entries”.The Report finally uses the Union Collection “Union LedStk Vouchers” to render the same.

Let us now move on to the new approach using “WalkEx” to achieve the same

Using the WalkEx Approach

;;The source collection “Voucher Source” is a Collection of Vouchers

```
[Collection: VoucherSource]
   Type    : Voucher
```

/ The collection “UnionLedStkVouchers” is the resultant collection which will contain all the Objects obtained out of walks and multiple walks over the same Source Collection. The Report finally uses this Collection. The attribute WalkEx is specified which has values as collection names “Ledger Details” and “StockItem Details”*/*

```
[Collection: Union LedStk Vouchers]
   Source Collection: VoucherSource
   WalkEx           : Ledger Details, StockItem Details
   Keep Source      : ().
```

*/*The Collection “Ledger Details” is walked over “AllLedgerEntries” SubObjects and aggregates the amount by grouping over Ledger Name. Note that the absence of source collection */*

```
[Collection: Ledger Details]
   Walk             : AllLedgerEntries
   By               : VchStockItem          : $LedgerName
   Aggr Compute     : VchLedAmount   : Sum    : $Amount
```

*/*The Collection “StockItem Details” is walked over “AllInventoryEntries” SubObjects and aggregates the amount by grouping over Stock Item Name. Note the absence of Source Collection in this */*

```
[Collection: StockItem Details]

   Walk              : AllInventoryEntries
   By                : VchStockItem              : $StockItemName
   Aggr Compute      : VchLedAmount      : Sum   : $Amount
```

The Collections used within the WalkEx uses the same Source Collection. Each Object of "Voucher Source" is walked across "Ledger Entries" and "Inventory Entries" in a single pass. Thus, there is an exponential improvement in performance as it traverses each object only once to gather the values for the resultant collection. In the case of Union Collection, for every Collection using different walk path, the Source Collection Object is being traversed again and again.

Walk Ex– Attribute Usage Implications

Let us consider the following code design, to understand the implication on various other collection attribute's in cases when Walk Ex is used.

```
;; Source Collection
[Collection: Src Coll]

          ...

 ;;Resultant Collection "Res Coll" using the Source Collection "Src Coll" and specifying Walk Ex Collections "Walk Ex Coll 1" and "Walk Ex Coll 2"
[Collection: Res Coll]

   Source Collection: Src Coll

   WalkEx: Walk Ex Coll 1, Walk Ex Coll 2

;;Walk Ex Coll 1
[Collection: Walk Ex Coll 1]

   Walk  : Path1, SubPath1, SubSubPath1

   By    : GroupName1 : $Method1

;;Walk Ex Coll 1
[Collection: Walk Ex Coll 2]

   Walk  : Path2, SubPath2, SubSubPath2

   By    : GroupName2 : $Method2
```

The following table shows the attributes of collection definition and their applicability in the Resultant collection as well as WalkEx collections.

Attributes	**Resultant Collection**	**Walk Ex-Collections**
Source Collection	Specified and applicable	Ignored
Keep Source	Specified and applicable	No significance
Is Client Only	Specified and Considered	Ignored
Sorting	Specified and applicable	Ignored
Filtering	Specified and applicable	Ignored
Max	Specified and applicable	No significance
Parm Var	Specified and Considered	Ignored but to be inherited from the resultant collection
Source Var	Specified and Considered	Specified and applicable
Compute\Filter Var	Specified and applicable	Specified and applicable
Fetch	Specified and applicable	Specified and applicable
Compute	Specified and applicable	Specified and applicable
Aggr Compute	No significance	Specified and applicable
Walk\By	If Specified these two attributes WalkEx will be ignored	Specified and applicable

Directory as a Data Source

As we are aware, a collection can be populated dynamically using the data available from a variety of external data sources. A common attribute Data Source is used to specify the Type and identity of the source from where the data is to be retrieved. Thereafter the data is available as objects and the associated information can be extracted from them using the corresponding methods. For e.g: If the data is populated from an XML file, the tag names are referred to as the method names. In case, the data is populated from a compound variable the corresponding member variable names is referred to as method names.

Prior to this release, the Data Sources supported were

- XML available over HTTP/HTTPS using Post / Get methods
- XML File available within the local disk or over a network
- Output XML from an External DLL
- Specific Objects from Current/Parent Report
- Variable

Syntax

```
Data Source: <Type> : <Identity> [: <Encoding> ]
```

Where,

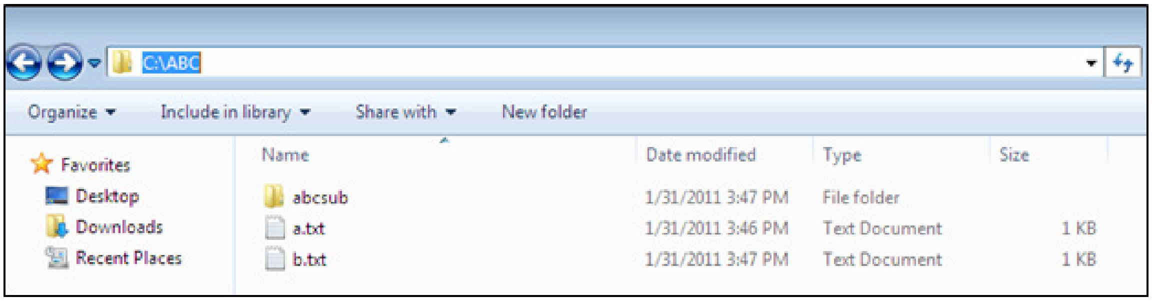
<Type> specifies the type of data source, File Xml, HTTP XML, Report, Parent Report, DLL or a Variable

<Identity> can be file path or scope etc. depending on the type specification

<Encoding> can be ASCII or UNICODE. This is Optional. The default value is UNICODE.

From this release 3.0 onwards, the collection attribute **Data Source** is enhanced to support **"Directory"** as data source type. This will enable to gather all information pertaining to the contents of the disk directory/folder. Each folder constituent ie either File or Directory along with its corresponding details are available as an object in the collection.

Let us consider the directory/folder "ABC" is as shown below:



Name	Date modified	Type	Size
abcsub	1/31/2011 3:47 PM	File folder	
a.txt	1/31/2011 3:46 PM	Text Document	1 KB
b.txt	1/31/2011 3:47 PM	Text Document	1 KB

Figure 1.1 Folder path

The folder contains two files "a.txt" and "b.txt" and the folder "abcsub". In order to retrieve the item details along with the corresponding information like type, size and date modified within a collection, the attribute Data Source can be specified with the enhanced syntax as

Syntax

```
[Collection: <Collection Name>]
     Data Source : <Type> : <Identity> [:< Encoding>]
```

Where,

<Collection Name> is the name of the collection where the data is populated as object
<Type> specifies the type of data source. As per new enhancement it is "Directory"
<Identity> Directory/folder Path when the type specified is directory
<Encoding> can be ignored for Type "Directory"

The specification given below will populate the collection "ABC Contents" with the folder contents from the path "C:\ABC". In this case each of the item i.e., a.txt, b.txt and "abcsub" will be available as separate objects of the collection. The related information pertaining to each object will be available as methods $name, $filesize, $IsDirectory, $IsReadOnly, $IsHidden

```
[Collection: ABC Contents]
   Data Source: Directory: "C:\ABC"
```

$filesize is applicable only if the object is of type FILE

4.2 Image Printing Capabilities

Over the years, there has been a major requirement from the user community to enable Image Printing in Tally. Earlier, we used to achieve this capability by creating a new Font Type by associating an Image with a particular character and further using the Style for the field. This had a few limitations in terms of image size, resolution and color. Also, it was not a clean way of incorporating the feature.

From this release onwards, we are supporting Image Printing using the latest enhancements mentioned as below

1. Graph Type attribute of Part allowing the specification of BMP, enabled for Print Mode
2. A new Definition Type “Resource” introduced in TDL

The configuration settings allow the user to specify the location of Image file and the same is printed as a logo in the top left of the following default Reports.

- Sales Invoice both Normal and Simple formats.
- Delivery Note/Challan
- Debit Note
- Credit Note
- Outstanding Receivables
- Remainder Letters
- Pay Slips
- Purchase Order
- Receipt voucher
- Confirmation of Accounts

Part Attribute – Graph Type

Prior to Tally.ERP 9 release 3.0, specification of Image (as BMP only) within a part was supported. The attribute **Graph Type** of the part is used for the same.

Syntax

```
      [Part: PartName]
                     …
          Graph Type        : Yes / Bitmap Image Path
```

Where,

Graph Type accepts **‘Yes’** or a path of bmp file. If the value of Graph Type is **“Yes”** then it will be treated as a graph. If the value is not **“Yes”** then system will look for a Bitmap file with the given expression. However bitmap image was only supported in the Display Mode.

From this Release onwards, this capability has been extended in the Print mode also.

Few points to be considered

- The attribute Graph Type supports Bitmap and JPG/JPEG.If the image type is specified as a JPEG/JPG, it will consume the same. If it is any other type, this will be considered as a BMP and locate the same from the path specified. If the file is unavailable or is not a valid image file, then the area allocated for image will be blank.

- The Part containing Graph Type cannot display or print any contents and any contents specified within the fields will be ignored. That's why it requires the specification of dummy lines and fields within the part.
- Part Height and Width must be apportioned appropriately as per the Image size to be printed. [If the height and/or width is not given then system will take the actual image size and use this same for display] If the user has specified Height and/or width which is different actual image size then system will do proportionate resizing of the image to fit into the given area] For example let's say area allocated for image is 300 X 300 and actual image size is 150X 75. Then system will display image in 300 X 150]

New Definition Type – Resource

A new definition type "Resource" has been introduced in TDL. This will allow accessing and using the resources (images/icons/cursors etc.) from a local disk, HTTP/FTP or from a DLL/EXE. The image formats supported at present are BMP/JPEG/ICON/CUR.

The resource thus created can be used in a part definition using the attribute "Image". This is applicable in both Print and Display mode.

However, when the same report is exported in PDF only BMP and JPEG is supported. Other file formats will be ignored.

Syntax

```
    [Resource: Name]
         Source              : <Path to Image file>
         Resource            : <NameOfResource>[:<DLL/EXE Name with path>]
         Resource Type       : Bitmap / Icon / Jpeg / Cursor
```

Attribute – Source

This is a Single attribute, hence accepts only one parameter. This attribute allows us to specify the image file path. This file can be a local disk file or a file available over an HTTP/FTP path.

Syntax

```
    Source: <Path to Image file>
```

Where,

<Path to Image file> can be a string expression which evaluates to the file path along with Filename and extension

Example 1:

```
[Resource: CmpImage]
   Source : "C:\Tally.bmp"
```

;; where the image "Tally.bmp" is available in local disk

Example 2:

```
[Resource: CmpImage]
   Source : “Http://www.tallysolutions.com/images/tallyHTTP.JPEG”
```

;; where the image tallyHTTP.JPEG is available over an HTTP Path

Attribute – Resource

This is a Dual attribute and accepts two parameters. The first parameter refers to the resource name present in an Exe / DLL. The second parameter is used to specify the path and the name of Exe / DLL. However this is optional. Incase not specified then system will look for the resource within tally.exe itself.

Syntax

```
      Resource: <NameOfResource>[:<DLL/EXE Name with path>]
```

Where,

<Name of Resource> is a string expression which evaluates to name of the resource present in specified DLL / EXE. When resources are added to DLL/EXE user can give a separate name for the resource)

<DLL/EXE Name with path> can be a string which evaluates to the complete DLL / EXE path

Example 1:

```
[Resource: CmpImage]
   Resource: “TITLEICON”
```

;; This uses the resource “TITLEICON” present in the Tally exe, as we have not specified the EXE path

Example 2:

```
[Resource: CmpImage]
   Resource      : “60040” : C:\ProgramFiles\WindowsNT +
                   \Accessories\wordpad.exe
   Resource Type :BMP
```

;; This uses the resource "60040" present in the wordpad.exe, as we have specified the EXE path as second parameter

The attribute “Source” and “Resource” are mutually exclusive i.e . either of them can be used. We cannot use both together. If both are specified in TDL, then system will use SOURCE and ignores RESOURCE attribute.

Attribute – Resource Type

This is a single attribute and hence accepts only one value as a parameter. This allows the specification of type of the resource. Type can be one of the standard windows image resources like - Bitmap, Icon, Cursor or JPEG. The type specified in the resource type will be used for loading the image appropriately. Resource Type is a mandatory attribute and must be specified for all sources. If not specified the type would be defaulted to Bitmap.

For the Icon resources - the nearest sized Icon will be take. For example if we have two Icons 16X16 and 32X32 and the part size is 2020 then 16X16 icon will be used for displaying.

Syntax

```
      Resource Type  : BMP / Icon / Jpeg / Cursor
```

Example:

```
[Resource: CmpImage]
   Source         : "C:\Tally.bmp"
   Resource Type : BMP
```

Part Attribute – Image

The resource thus created by using the Definition "Resource" can be used in the Part with the introduction of a new attribute "Image".

Syntax

```
      [Part: PartName]
                     …
          Image   : <Resource Name>
```

Where,

<Resource Name> is the name of the resource definition.

Example:

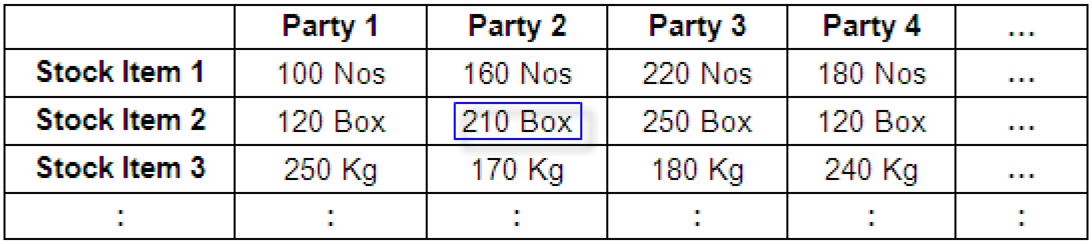
```
[Part: Part ABC]
   Image : CmpImage
```

4.3 Enhanced Columnar Capability

Columnar Reports in general

A matrix report looks like a grid. It contains a row of labels, a column of labels, and information in a grid format that is related to both the row and column labels. In Tally, two dimensional matrix reports can be designed using the auto column report approach (using Repeat Variables). Traditionally these types of Reports are referred as columnar Reports. In particular, Matrix report is a variant of automatic auto column reports where the columns are repeated over a variable associated at the Report. The collection repeated with this variable is used to populate the repeated values into the variable. The method value in the detail line is extracted from a different collection based on corresponding row and column indexes.

Following is a typical two dimensional matrix report showing the total number of stock items sold for each party.



	Party 1	Party 2	Party 3	Party 4	...
Stock Item 1	100 Nos	160 Nos	220 Nos	180 Nos	...
Stock Item 2	120 Box	210 Box	250 Box	120 Box	...
Stock Item 3	250 Kg	170 Kg	180 Kg	240 Kg	...
:	:	:	:	:	:

Enhanced Capabilities for Columnar Reporting

The latest enhancements in the area of Columnar Reporting enables us to design the reports using a new approach altogether. A field within a line can display method values from multiple objects of the collection. Context Free repeat within the part and line enable repetition on simple/ list variable values also.

These features give a better control in the hands of the programmer in designing such reports.

If we consider the above report layout, the labels in columns can now repeat over a collection of Party's. The data in the cells can be populated based on the combination of the row and column label values across the dimensions. Like in the above example, the highlighted cell contains the value of total sales quantity corresponding to the party “Party 2” for the Stock Item “Stock Item 2”

The following enhancements have enabled to achieve this functionality

- Repeat Attribute for Part and Line over a Collection
- Context Free Repeat for Part and Line together with SET/Break On
- Usage of function $$LineObject

Attribute “Repeat” Enhancements – Part and Line

The Repeat Attribute has been enhanced consistently across Part and Line Definition to support “Context Based” as well as “Context Free Repeat”

Attribute Repeat – Part Definition

The common syntax allows the repetition of a contained line with or without a collection.

Syntax

```
[Part: <Part Name>]
      Repeat : <Line Name> [: <Collection>]
      Set    : <Count>
```

Where,
<Part Name> is the name of the part
<Line Name> is the name of the line to be repeated

<Collection> is the name of the collection on which the line is repeated. This is an optional parameter.
<Count> denotes the number of times the line is to be repeated if Collection Name is not specified.

Context based Repeat – The Repeat attribute of the part can repeat the contained line over a collection. Each line in this case is associated with each object of the collection. This was the earlier capability even before Tally.ERP 9

Context Free Repeat – From Release 1.8, the collection parameter in the above syntax has been made optional. This allows the repetition of a contained line without a collection. Since the no of times the line has to be repeated is not known, the usage of attribute SET to specify the count becomes mandatory. In case of Edit mode, attribute Break on can be used to specify the terminating condition for repletion.

Attribute Repeat – Line Definition

So far, the repeat attribute at Line definition has been accepting only a field name which internally uses the repeat behavior of the Report and Variable for determining the no of times it can be repeated.
This attribute is now enhanced to support the consistent syntax to enable “Context Based” and “Context Free” repetition of the same field horizontally.

Syntax

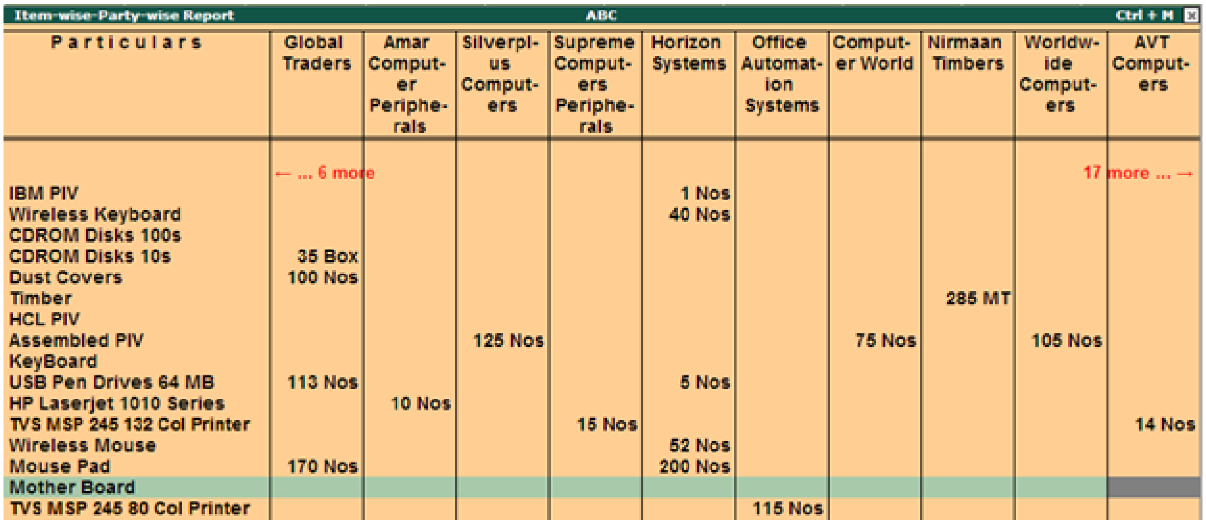
```
    [Line: <Line Name>]

        Repeat: <Field Name> [: <Collection Name>]
                Set   : <Count>
```

Where,
<Line Name> is the name of the Line.
<Field Name> is the name of the Field to be repeated.
<Collection Name> is the name of the collection on which the Field is repeated which is optional.
<Count> denotes the number of times the Field is required to be repeated if Collection Name is not specified.

Context based Repeat – The Repeat attribute of the line can repeat the contained field over a collection. Each field in this case is associated with each object of the collection
Context Free Repeat – The collection parameter in the above syntax is optional. This allows the repetition of a contained field without a collection. Since the no of times the field to be repeated is not known, the usage of attribute SET to specify the count becomes mandatory. In case the SET is not specified the Field will be repeated as per the existing Columnar Behavior.

Example 1: Item Wise Party Wise sales quantity report using Context Based Repeat of Field

The following screen shows the Item-wise-Party-wise Report using the enhanced columnar capability.



Item-wise-Party-wise Report ABC Ctrl + M

Particulars	Global Traders	Amar Computer Peripherals	Silverplus Computers	Supreme Computers Peripherals	Horizon Systems	Office Automation Systems	Computer World	Nirmaan Timbers	Worldwide Computers	AVT Computers
	← ... 6 more									17 more ... →
IBM PIV					1 Nos					
Wireless Keyboard					40 Nos					
CDROM Disks 100s										
CDROM Disks 10s	35 Box									
Dust Covers	100 Nos									
Timber								285 MT		
HCL PIV										
Assembled PIV			125 Nos				75 Nos		105 Nos	
KeyBoard										
USB Pen Drives 64 MB	113 Nos				5 Nos					
HP Laserjet 1010 Series		10 Nos								
TVS MSP 245 132 Col Printer				15 Nos						14 Nos
Wireless Mouse					52 Nos					
Mouse Pad	170 Nos				200 Nos					
Mother Board										
TVS MSP 245 80 Col Printer						115 Nos				

Figure 1.2 Item-wise-Party-wise Report

Below is the code snippet used to design the above report with the enhanced columnar capability.

1. Collection definitions used, i.e. for Stock Item, Party and the collection for getting the values as shown below:

```
;; Collection Definition
[Collection: Smp CFBK Voucher]
   Type   : Voucher
   Filter: Smp IsSalesVT

[Collection: Smp Stock Item]
   Source Collection: Smp CFBK Voucher
   Walk              : Inventory Entries
   By                : IName: $StockItemName
   Aggr Compute      : BilledQty: SUM: $BilledQty
   Keep Source       : ().
   Filter            : SmpNonEmptyQty

[Collection: Smp CFBK Party]
   Source Collection: Smp CFBK Voucher
   Walk              : Inventory Entries
   By                : PName : $PartyLedgerName
   Aggr Compute      : BilledQty: SUM: $BilledQty
   Keep Source       : ().
   Filter            : Smp NonEmptyQty

[Collection: Smp CFBK Summ Voucher]
   Source Collection: Smp CFBK Voucher
   Walk              : Inventory Entries
   By                : PName: $PartyLedgerName
   By                : IName: $StockItemName
   Aggr Compute      : BilledQty: SUM: $BilledQty
   Keep Source       : ().
   Search Key        : $PName + $IName

;; System Formula
[System: Formula]
   Smp IsSalesVT  : $$IsSales    : $VoucherTypeName
   Smp NonEmptyQty: NOT $$IsEmpty: $BilledQty
```

From the above Collection, following can be observed:

- The Rows i.e., Stock Items are repeated over the Collection Smp Stock Item.
- The Columns i.e., Party Names are repeated over the Collection Smp CFBK Party.
- The Intersection values between these Rows and Columns i.e., Item wise Party wise Sales Quantity is set using the Collection Smp CFBK Summ Voucher. This Collection is indexed on Methods $PName + $IName using Collection Attribute Search Key. Thus the Collection is indexed on Party Name and Stock Item Name which makes it unique across all the Objects within the Collection Smp CFBK Summ Voucher.

2. The Lines Title and Detail are repeated for the Party Names as shown below

```
[Line: Smp CFBK Rep Title]
   Use   : Smp CFBK Rep Details
   Local : Field  : Default             : Type       : String
   Local : Field  : Default             : Align      : Center
   Local : Field  : Smp CFBK Rep Name   : Set as     : "Particulars"
   Local : Field  : Smp CFBK Rep Name   : Widespaced : Yes
   Local : Field  : Smp CFBK Rep Party  : Set as     : $PName
   Local : Field  : Smp CFBK Rep Party  : Lines      : 0
   Local : Field  : Smp CFBK Rep ColTotal: Set as    : "Total"
```

```
[Line: Smp CFBK Rep Details]
   Fields: Smp CFBK Rep Name, Smp CFBK Rep Party, Smp CFBK Rep Col Total
   Repeat: Smp CFBK Rep Party: Smp CFBK Party
```

Title Line uses the detail line where the Field “Smp CFBK Rep Party” is repeated over the Collection “Smp CFBK Party”. In the Title Line, the Field “Smp CFBK Rep Party” is set with the value “$PName” which sets the Party Names from the Collection “Smp CFBK Party”

3. Retrieving the values in cells based on Party name available from context and stock item name available in field as shown below:

```
[Field: Smp CFBK Rep Name]
   Use        : Name Field
   Set as     : $IName
   Display    : Stock Vouchers
   Variable   : Stock Item Name

[Field: Smp CFBK Rep Party]
   Use        : Qty Primary Field
   Set as     : $$ReportObject:$$CollectionFieldByKey:$BilledQty:+
                @SKFormula:SmpCFBKSummVoucher
   SKFormula  : $PName + #SmpCFBKRepName
   Format     : “NoZero”
   Border     : Thin Left
```

In the above code snippet, we can observe that Field “Smp CFBK Rep Party” is the intersection between the rows and columns. The value is gathered from the Collection “Smp CFBK Summ Voucher” using the function CollectionFieldByKey where the Index Key in the current context is passed as a parameter. “$PName” in the current object context returns the Party Name. Similarly, the Field Value “#SmpCFBKRepName” in the current context returns the Stock Item Name. Hence, the Search Key Index, “Party Name + Stock Item Name” for every Intersection point is passed to this function which extracts and returns the corresponding Quantity from the Collection.

4. Calculating Field Level Totals i.e., Stock Item Totals across all Parties are calculated using Line Attribute Total and Function Total as shown below:

```
[Line: Smp CFBK Rep Details]
   Total    : Smp CFBK Rep Party

[Field: Smp CFBK Rep Col Total]
   Use     : Qty Primary Field
```

```
    Set as : $$Total:SmpCFBKRepParty
```

Line “Smp CFBK Rep Details” contains an Attribute Total with accepts Field Names as its value. In other words, we declare at the Line, the Fields to be summed for later use. This sum gets accumulated and rendered in the Field “Smp CFBK Rep Col Total” where Total Function returns the accumulated Total for the given Field Name as the Parameter to this Function.

New Built–in Function $$LineObject

Since the Line Attribute Field can now be repeated over a Collection wherein the Object context inherited from Line is overridden in Field. Hence, to switch back to the parent i.e., Line’s object context and extract the required method value from the Line's Object context, a New Function LineObject is introduced.

Syntax

```
        $$LineObject:<String Formula>
```

Where,

String Formula can be an expression that gets evaluated in the Object context associated at the current field's parent line in the Interface Object hierarchy.

Interactive Reporting capabilities using Aggregated or External objects

The Actions “Remove Line” and “Show Last Removed Line” and “Show Removed Lines” work on the concept of Object Identifier. Whenever the collection of internal objects are rendered as a report, the default buttons "Remove Line", “Restore Line” using the above actions work on them as they are uniquely identifiable.

In cases, where the Collection used contain aggregated Objects, or objects from an external data sources like XML etc, the objects available do not contain an unique identifier. When such collections are rendered the Actions mentioned above do not work.

In order to overcome the problem the attribute Search Key behavior has been enhanced to assign a unique key for such Object Types. It takes a single or a combination of methods which will serve as a unique identifier to each object of the aggregated or an external collection. It has to be ensured that each object in the collection must contain unique values for the method which is assigned as the key.

Syntax

```
        [Collection: <Collection Name>]
                  Search Key :  <Expression>
```

Where,

<Expression> which evaluates to an unique identifier for each object of the collection. It is usually a combination method names separated by ‘+’ which must make unique combination for each object of the Collection.

Example 1:

Please observe the previous sample report Item Wise Party Wise report, wherein Alt + R Key combination does not work for Removal of Line as there is no unique identifier for the Line Object. Each line in the example is repeating over the objects of the collection “Smp Stock Item”. To specify the unique Object identifier, alter this Collection by specifying Search Key attribute with a unique combination of Methods as value. In this case, it is the method name $IName i.e. the Stock Item Name over on which the objects are grouped.

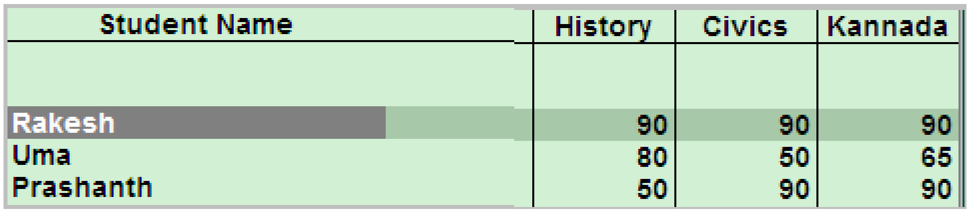
```
[#Collection: Smp Stock Item]
   Search Key: $IName
```

Example 2:

Following is another example using an external data objects as per available in the following XML file containing the data for Students and corresponding marks in various subjects.

```
<StudData>
  <Student>
     <Name>Rakesh</Name>
     <Subject>
          <Name>History</Name>
       <Mark>90</Mark>
         </Subject>
      <Subject>
          <Name>Civics</Name>
       <Mark>90</Mark>
         </Subject>
      <Subject>
          <Name>Kannada</Name>
       <Mark>90</Mark>
         </Subject>
      </Student>
     <Student>
       <Name>Uma</Name>
         <Subject>
        <Name>History</Name>
           <Mark>80</Mark>
         </Subject>
      <Subject>
         <Name>Civics</Name>
           <Mark>50</Mark>
        </Subject>
      <Subject>
         <Name>Kannada</Name>
           <Mark>65</Mark>
       </Subject>
     </Student>
    <Student>
        <Name>Prashanth</Name>
        <Subject>
           <Name>History</Name>
         <Mark>50</Mark>
           </Subject>
     <Subject>
           <Name>Civics</Name>
         <Mark>90</Mark>
     </Subject>
        <Subject>
           <Name>Kannada</Name>
         <Mark>90</Mark>
     </Subject>
  </Student>
</StudData>
```

The data populated from the above XML is displayed as a columnar report as shown below



Student Name	History	Civics	Kannada
Rakesh	90	90	90
Uma	80	50	65
Prashanth	50	90	90

Figure 1.3 Student-wise-Subject-wise Marks Report

Student wise Subject wise Marks information is listed in tabular information as shown above. Now, on removing the selected line(s), the required lines must be removed. Since, this report is constructed out of an external source i.e., XML Data, the same requires an unique identifier for each object in the repeated line. In this case it is the Student Name, hence the Search Key should contain this as an identifier.

Below is the sample code required to display the above report in a columnar fashion with the Remove/Restore Line behavior incorporated:

```
[Report: Ext XML Data Stud]
   Form   : Ext XML Data Stud
```

```
[Form: Ext XML Data Stud]
   Parts : Ext XML Data Stud
   Bottom ToolBar Buttons: BottomToolbarBtn8, BottomToolbarBtn9,+
                           BottomToolbarBtn10

[Part: Ext XML Data Stud]
   Lines        : Ext XML Data Stud Heading, Ext XML Data Stud Info
   Repeat       : Ext XML Data Stud Info: Ext XML Data Students
   Scroll       : Vertical
   CommonBorder : Yes

      [Line: Ext XML Data Stud Heading]
        Fields: Ext XML Data Stud Name, Ext XML Data Stud Mark
        Repeat: Ext XML Data Stud Mark : Ext XML Data Stud Subj Summary
        Local : Field: Default: Type   : String
        Local : Field: Default: Style  : Normal Bold
        Local : Field: Default: Align  : Centre

        Local : Field: Ext XML Data Stud Name: Set As: “Student Name”
        Local : Field: Ext XML Data Stud Mark: Set As: $SubjectName
        Local : Collection: Ext XML Data Stud SubjSummary: Delete: Filter
        Local : Collection: Ext XML Data Stud SubjSummary: +
        Delete: By: StudentName
        Border: Thin Top Bottom

      [Line: Ext XML Data Stud Info]
        Fields: Ext XML Data Stud Name, Ext XML Data Stud Mark
        Repeat: Ext XML Data Stud Mark: Ext XML Data Stud Subj Summary
        [Field: Ext XML Data Stud Name]
          Use   : Name Field
          Set As: $Name

        [Field: Ext XML Data Stud Mark]
          Use   : Number Field
          Set As: $$Number:$SubjectTotal
```

```
         Align : Right
         Border: Thin Left

[Collection: Ext XML Data Students]
   Data Source     : File XML : "D:\StudData.xml": Unicode
   XML Object Path: Student  :1  :StudData
   Search Key      : $Name

[Collection: Ext XML Data Stud Subj Summary]
   Source Collection: Ext XML Data Students
   Walk               : Subject
   By                 : StudentName  : $..Name
   By                 : SubjectName  : $Name
   Aggr Compute       : SubjectTotal : SUM  : ($$Number:$Mark)
   Keep Source        : ().
   Filter             : ForThisStudent

[System: Formula]
   ForThisStudent    : $StudentName = $$ReqObject:$Name
```

In the above code, Line Ext XML Data Stud Info is repeated over the Collection Ext XML Data Students where Search Key is specified to be $Name. Hence, the Remove/Restore Line behavior will work.

4.4 Persisting Variables at System Scope in a User Specified File

As announced in Release 2.0, we are aware that the variables at the report scope can be persisted in a user specified file using the action SAVE VARIABLE. This can be re-loaded as required using the action LOAD VARIABLE.

The latest enhancements in variable persistence allow the user to persist and re-load the variables at System Scope (in a User Specified File) as well.

Action – SAVE VARIABLE

The action SAVE VARIABLE which is used to persist the Report Scope Variables in a user specified file now allows us to persist the System Scope Variables also. Syntax of this action remains same. The desired behavior is achieved with changes in the variable list specification.

Syntax

```
      SAVE VARIABLE : <FileName> [:<Variable List>]
```

Where,

<File Name> is the name of file in which the report scope/ system scope variables are persisted. The extension. PVF will be taken by default if the file extension is not specified.

Variable List specification changes

1. It is the list of comma separated variables that need to be saved in the file.
2. **Now * also can be used to specify the variable list which means all at 'current scope'**.
   - The current scope can either be System or Report.
   - Specifying *, will ignore the persist flag and save all variables in the scope irrespec tive of "Persist: Yes" at the Variable definition level.
3. If Variable list is not provided, it will persist all the variables which are set "Persist: Yes" at variable definition level.
4. Dotted notation syntax is also supported in the variable list specification for scope specification. However this cannot be used for SUB levels. This can be used only for accessing parent scope variables.
   - Single Dot "." refers to current scope. Double Dot " . ." to parent scope. Triple Dot " . . . " to grandparent scope.
   - " (). " refers to the System Scope.

Action – LOAD VARIABLE

The action LOAD VARIABLE which is used to load the Report Scope Variables in a user specified file now allows us to load the System Scope Variables also. Syntax of this action remains same. The desired behavior is achieved with changes in the variable list specification.

Syntax

```
LOAD VARIABLE : <FileName> [:<Variable List>]
```

Where,

<File Name> is the name of file in which the report scope/ system scope variables are persisted. Specifying file extension is mandatory while loading variable values.

Variable List specification changes

1. It is the list of comma separated variables that need to be loaded from the file.
2. While loading * is not relevant and will be ignored
3. While loading 'Persist' flag of the variable is ignored. It is assumed that the variable must have a persist flag OR it is saved forcefully and hence to be loaded.

Example 1:

There is a requirement to persist values of all system scope variables in a user specified file and load the values from the file whenever required. Refer to the below code snippet:-

```
[#Menu: Gateway of Tally]
   Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

*Buttons **SLSystemScopeSave** & **SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions S**AVE VARIABLE** & **LOAD VARIABLE**.*

```
 [Button: SLSystemScopeSave]
   Key           :  Alt+F
   Action        :  SAVE VARIABLE : SLSystemScope.pvf : *
   Title         : "Save Sys Var"
```

*Values of all system scope variables will be persisted in the file **SLSystemScope.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button: SLSystemScopeLoad]
   Key           :  Alt + L
   Action        :  LOAD VARIABLE : SLSystemScope.pvf
   Title         : "Load Sys Var"
```

*Values of all system scope variables will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.*

Example 2:

There is a requirement to persist values of all system scope variables which are set **"Persist : Yes"** at variable definition level in a user specified file and load the values from the file whenever required. Refer to the below code snippet:-

```
[#Menu: Gateway of Tally]
   Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

*Buttons **SLSystemScopeSave** & **SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions **SAVE VARIABLE** & **LOAD VARIABLE**.*

```
 [Button: SLSystemScopeSave]
   Key           :  Alt+F
   Action        :  SAVE VARIABLE : SLSystemScope.pvf
   Title         :  "Save Sys Var"
```

*Values of all variables at system scope which are set **"Persist : Yes"** at variable definition level will be persisted in the file **SLSystemScope.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button: SLSystemScopeLoad]
   Key           :  Alt + L
   Action        :  LOAD VARIABLE : SLSystemScope.pvf
   Title         :  "Load Sys Var"
```

*Values of all variables will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.*

Example 3:

There is a requirement to persist the system scope variables **SVSymbolInSign** & **SVInMillions** in a user specified file and load values of these variables from the file whenever required. Refer to the below code snippet:-

```
[#Menu: Gateway of Tally]
   Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

*Buttons **SLSystemScopeSave** & **SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions **SAVE VARIABLE & LOAD VARIABLE**.*

```
[Button: SLSystemScopeSave]
   Key       :  Alt+F
   Action    :  SAVE VARIABLE : SLSystemScope.pvf : +
                SVSymbolInSign,SVInMillions
   Title     :  "Save Sys Var"
```

Values of the system scope variables SVSymbolInSign & SVInMillions will be persisted in the file SLSystemScope.pvf on execution of the action SAVE VARIABLE.

```
[Button: SLSystemScopeLoad]
   Key       :  Alt + L
   Action    :  LOAD VARIABLE : SLSystemScope.pvf  : +
                SVSymbolInSign,SVInMillions
   Title     :  "Load Sys Var"
```

*Values of the system scope variables **SVSymbolInSign** & **SVInMillions** will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.*

Example 4:

Let us suppose the following report is displayed in Create mode from a menu item.

```
[Report: Smp SLReport]
   Form     : Smp SLForm
   Variable : SaveLoadVar1,  SaveLoadVar2
```

The variables **SaveLoadVar1 & SaveLoadVar2** are declared at Report Scope.

```
[Form: Smp SLForm]
   Parts    : Form SubTitle, Smp SL Part
   Button   : Smp SaveVar, Smp LoadVar
```

Buttons **SmpSaveVar & SmpLoadVar** are added at Form Level to execute the actions **SAVE VARIABLE & LOAD VARIABLE**.

Let us look into the below scenarios to persist & load System Scope as well as Report Scope Variable values:-

I. Persist & Load all Report Scope Variables & a specific System Scope Variable

```
[Button: Smp SaveVar]
   Key    : Alt + S
   Action : SAVE VARIABLE: SLReportCfg.pvf: *,().SVInMillions
   Title  : "Save Variable"
```

*Values of all variables declared at report scope and the value of system scope variable **SVInMillions** will be persisted in the file **SLReportCfg.pvf** on execution of the action **SAVE VARIABLE**. (The variable SVInMillions is prefixed with (). to denote the same as System Scope Variable).*

```
 [Button: Smp LoadVar]
   Key     : Alt + L
   Action  : LOAD VARIABLE : SLReportCfg.pvf : *,().SVInMillions
   Title   : "Load Variable"
```

Variable list specification will be ignored. Values of all report scope variables and the value of system scope variable **SVInMillions** will be loaded from the file **SLReportCfg.pvf** on execution of the action **LOAD VARIABLE**.*

II. Persist & Load a specific Report Scope variable & a specific System Scope variable

```
[Button: Smp SaveVar]
   Key      : Alt + S
   Action   : SAVE VARIABLE: SLReportCfg.pvf : SaveLoadVar1,+
              ().SVInMillions
   Title    : "Save Variable"
```

*Value of Report scope variable **SaveLoadVar1** and value of system scope variable **SVInMillions** will be persisted in the file **SLReportCfg.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button: Smp LoadVar]
   Key       : Alt + L
   Action    : LOAD VARIABLE : SLReportCfg.pvf :SaveLoadVar1,+
                ().SVInMillions
   Title     : "Load Variable"
```

*Value of Report scope variable **SaveLoadVar1** and value of system scope variable **SVInMillions** will be loaded from the file **SLReportCfg.pvf** on execution of the action **LOAD VARIABLE**.*

4.5 New Events Introduced

As a part of the Language enhancements, in recent past there have been significant enhancements as a part of the **Event Framework**. Before this release events introduced were mostly related to handling application start up and close, company loading and unloading. The Object specific events were mainly focused around trapping events while rendering the data on screen and print.

In this Release, we have introduced events to handle user specific requirements on data manipulation to be handled during Export and Import of data. With the introduction of Events, **Start Import**, **Import Object** and **End Import**, the programmers have got complete control to manipulate the data prior to importing the same into the company. This can be useful in scenarios like data transfers between Inter Branch where Delivery Note in a branch gets transformed into Receipt Note in the second branch; Sales transaction in a Branch gets transformed into Purchase transaction in the second branch and so on. Also, an action **Import Object** is introduced to begin the Import process.

While exporting Full objects to XML and SDF formats with the introduction of Export Events, **Before Export**, **Export Object** and **After Export**, the user will be able to trap these events and get an access to the object being exported which can be altered as required before export. This can be useful in scenarios like changing required information during export, not displaying price/ amount of the stock item while synchronizing Delivery Note to the branch offices, creating a consolidated sales entry from all the sales transactions of the day etc.

Import Events

The following events can be used within **Import File** Definition

Syntax

```
        On: Start Import: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event **Start Import** executes the actions before beginning the import process. At this stage, the data objects will not be available since it is prior to gathering the data from the file. This event can be used to communicate any messages to the user like starting the import process, etc.

Syntax

```
        On: Import Object: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event Import Object executes the actions post gathering the Objects from the File before importing the same in the current company. At this stage, the data objects are available since it is post gathering the data from the file. This event is actually useful to manipulate & transform the data from one form to another, i.e., from Receipt Note to Delivery Note, etc.

Syntax

```
      On: Import Object: <Logical Condition>: Import Object
```

If the Event On Import Object is used, it overrides the default Import Object behavior as a result of which we need to explicitly specify to being importing the objects. Post performing the necessary actions prior to importing the objects, Action Import Object must be specified to instruct the system to continue the import process.

Syntax

```
      On: End Import: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event End Import executes the actions after importing the objects. At this stage, the data objects will not be available since it is post importing the objects within the current company. This event can be used to communicate any messages to the user like ending the import process, Import Successful, etc.

Example:

```
[#Import File: Vouchers]
   On : Start Import      : Yes    : Call   : Start Import
   On : Import Object     : Yes    : Call   : Change Values
   On : Import Object     : Yes    : Import Object
   On : End Import        : Yes    : Call   : End Import

[Function: Start Import]
   00     : MSGBOX             : "Status": "Starting Import Process"

[Function: Change Values]
   00     : SET VALUE          : Narration : $Narration + " - Updated by +
                                 Import Object Event"
   10     : SET TARGET         : LedgerEntries[1]
   20     : SET VALUE          : LedgerName: "Branch Ledger"



[Function: End Import]
   00     : MSGBOX: "Status" : "Imported data successfully"
```

In the above example, before importing the data, **Narration** Method is being altered and first **Ledger Name** is being altered to the **Branch Ledger**. Before starting and after ending the import process, appropriate messages are being displayed to the user.

Export Events

The following events can be used within **Form** Definition

Syntax

```
On: Before Export: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event **Before Export** executes the action before beginning the export. This event can be used to communicate any message to the user.

Syntax

```
On: Export Object: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event **Export Object** executes the action before the object is exported. The user will get the object being exported which can be altered as required before export. The form level Export Object is used to get an access to the object associated at the Report Level and manipulate the same before exporting.

Syntax

```
On: After Export: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event **After Export** executes the action at the end of Form Export. This event can be used to communicate any message to the user.

The following events can be used within **Line** Definition

Syntax

```
On: Export Object: <Logical Condition>: <Action>: <Action Parameters>
```

If the logical condition specified returns TRUE, Event **Export Object** executes the action before every object is exported. The line level Export Object is used to get an access to the each object associated at the line level and manipulate the same before exporting.

Example:

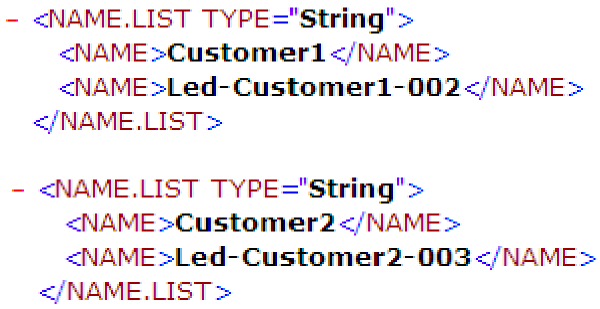
```
[Form: ExpEvtForm]
   On          : Before Export : Yes   : Call  : Export Start
   On          : After Export  : Yes   : Call  : Export End
   Part        : ExpEvtPart
   Button      : Export Button
   Full Object : Yes
```

```
[Part: ExpEvtPart]
   Line     : ExpEvtLine
   Repeat   : ExpEvtLine: ExpLedger
   Scroll   : Vertical

[Line: ExpEvtLine]
   On          : Export Object  : Yes   : Call  : ExportObject:$$Line
   Fields      : ExpEvtFld1, ExpEvtFld2
   Full Object : Yes

[Collection: ExpLedger]
   Type     : Ledger
   Fetch    : Name, Parent

[Function: ExportStart]
   00 : MSGBOX: “Status”: “Starting Export”

[Function: ExportObject]
   Parameter : LineNo : Number
   01 : INSERT COLLECTION OBJECT : Name
   02 : SET VALUE : Name : “Led”+“-”+$name+“-”+“00”+$$String:##LineNo

[Function: ExportEnd]
   00 : MSGBOX: “Status”: “Ending Export”
```

In the above example, the line is repeated over the Collection **ExpLedger** which is type **ledger**. The event **Export Object** at the line level will be triggered before exporting every ledger object. The function “Export Object” which is called on occurrence of the event inserts a new object for the collection "Name" and method **Name** (alias name) will be set with the new value by concatenating the strings “Led”, Name of the ledger and the line no prefixed with “00”.

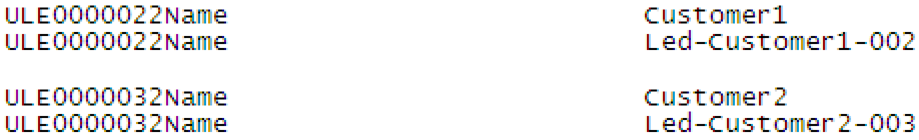
Before starting and after ending the export, appropriate messages are being displayed to the user through the events Before Export and After Export at Form Level

The exported fragments of XML and SDF outputs are given below in which we can observe that an alias name is created with the value as set inside the function



```
- <NAME.LIST TYPE="String">
    <NAME>Customer1</NAME>
    <NAME>Led-Customer1-002</NAME>
  </NAME.LIST>

- <NAME.LIST TYPE="String">
    <NAME>Customer2</NAME>
    <NAME>Led-Customer2-003</NAME>
  </NAME.LIST>
```

Figure 1.4 XML Format



```
ULE0000022Name                              Customer1
ULE0000022Name                              Led-Customer1-002

ULE0000032Name                              Customer2
ULE0000032Name                              Led-Customer2-003
```

Figure 1.5 SDF Format

4.6 Enhancement – Programmable Configuration

Prior to Tally.ERP 9 release 1.52 when multiple reports were printed or mass mailing was being done in a sequence, prior to each **Action**, a configuration report is displayed for user input. This would interrupt the flow, thereby requiring a dedicated person to monitor the process which is time consuming. This had been addressed in Tally.ERP 9 release 1.6, by providing an optional logical parameter to suppress the repeated display for the configuration screen before the invocation of global actions 'Print', 'Export', 'Upload' and 'Email'.

Actions enabled for Programmable Configurations

In order to print, export, upload and email the current report in context the actions 'Print Report', 'Export Report', 'Upload Report' and 'Email Report' are used. Prior to this release the programmable configuration was not supported for these actions. With the latest enhancement the display of configuration screen can be suppressed for these actions also.

The syntax of these actions supporting programmable configurations is:

Syntax

```
<Action Name> [ : <Report Name> [:<Logical Value>]]
```

Where,

<Action Name> can be any Print, Export, Mail and Upload. Enabled for 'Print Report', 'Export Report', 'Upload Report' or 'Email Report' in the latest release..

<Report Name> is name of the report or a dot (.). Since 'Print Report', 'Export Report', 'Upload Report' and 'Email Report' takes the current report in context and the subsequent parameter is the logical parameter for suppressing configuration, dot (.) signifies the specification of current Report Name. This is an optional parameter. However this is mandatory in case suppress configuration is to be enabled.

<Logical Value> can be TRUE, FALSE, and YES or NO. This is an optional parameter. By default value is NO. If it is set to YES then configuration screen would not be displayed.

The variables to be set as per requirement of each Action is done in the same way as discussed in prior releases. Refer to the topic on "Programmable Configuration for Actions" in Release 1.6 document for more details.

Example:

To export current report without displaying configuration screen

```
          |

          |

  40: EXPORT REPORT:    .     : TRUE

         |

         |
```

4.7 Optional Default TDL Loading

Many Third Party Applications use Tally's rapid application development environment to render various complex reports using Tally Definition Language (TDL). Tally.ERP 9 acts as a front end application for various external databases to retrieve and manipulate information as and when required. Tally, being a comprehensive business application loads all the TDL's required as per the functional aspects of the Application. In cases where the third party applications require using Tally purely as a development platform, loading of complete application TDLs may prove to be expensive in terms of startup time.

This release onwards the application TDLs are segregated as

- **Base TDL Files** – This contains the commonly required templates like styles, variables, buttons which can be used by any report which is rendered.
- **Default TDL Files** –This contain the TDLs which are specifically meant for functional requirements of the Tally.ERP 9 application.

This has enabled us to launch Tally using the minimal Base TDL files avoiding the overhead of loading the Default TDL files. This can be achieved by using the command line parameter **/NODEF**.

Syntax

```
<Tally Application> /NODEF
```

Example:

```
D:\Tally.ERP9\Tally.Exe/NODEF/NOINITDL/TDL:"D:
              \Party\CustomReports.TDL"
```

The above Tally.ERP 9 application would start only with Base TDLs without loading default TDL Files which means that Tally Application would start rapidly. None of the INI TDLs will be loaded due to the parameter **/NOINITDL**, only the TDL file passed with the parameter **/TDL**, D:\Party\CustomReports.TDL will be loaded.

In line with the above enhancement, the product Tally.Developer 9 Release 3.0 will also support the command line parameter /NODEF. In case, the application needs to be started with only the Base TDLs then the option /NODEF will be used.

4.8 Refresh Issues in context of User Defined Function Evaluation

As we are already aware, the TDL Procedural artifact “Function” is used in two scenarios

1. **Evaluation** – where the function is expected to perform some computation and return the result to the expression within which it is called. The usage is similar to a Predefined function. In evaluation mode, the function is called using a “$$”

Example:

```
[Field: My Field]
   Set as   : $$MyUserFunction:Parameter1: Parameter2
```

2. **Execution** – where the function is expected to perform certain set of tasks which changes the state of the application or the data. The usage is similar to a Predefined Action. In execution mode, the function is called using the keyword “Call” and can be invoked from a Key/Button, Menu item, an Event or from within another function.

Example:

```
[Key: My Key]
     Action : CALL : MyUserActionFpunction : Parameter1 : Parameter2
```

In case of a predefined Function, whenever the function accessed and manipulated certain UI elements like a variable, field value, or data elements like method values of objects the link between the element and the calling UI is established. Each time these get manipulated again, the function gets reevaluated, new values get calculated and the corresponding UI is refreshed with new values.

Let us look at the example below, to articulate this better

```
   [Variable: My Variable]
     Type : String

        [Field: My Field]
          Type    : String
          Set as  : ##MyVariable
```

When report is started, the ‘Set as’ attribute of the My Field is evaluated. During this evaluation a link between the Field My Field and the Variable My Variable (which is accessed for its value).

Now during a scenario, say a F12 configuration of the report changes the MyVariable value. Now system would automatically determine that, Field My Field was depending on the value of the variable which has changed now, and hence RE-EVALUATE the field's Set as attribute to get its new value.

In case of a TDL procedural “Function” we faced certain issues, where the fields calling the function for some evaluation, were not refreshed with new values when the accessed elements get modified elsewhere and the function does not get reevaluated.

To articulate this better, let’s extend the above example by using a user defined function.

```
[Variable: My Variable]
  Type: String

    [Field: My Field]
      Type   : String
      Set as : $$MyUserFunction

[Function: My User Function]
  Returns     : String
  01 : RETURN : ##MyVariable
```

In the above case, the system would not establish any relation between the Field and the variable as it is processed via a function and hence, when the Value of the variable is changed elsewhere, Field's Set as will not get re-evaluated automatically to get its new value.

This issue has been resolved in this Release. The related refresh problems which we may have faced in context of using "Function" in evaluation scenario, has been resolved. However, in negligible cases we may hit with performance issues due to repeated refresh. This mainly happens when the modification of values of UI / data elements like objects, variables etc causes the regeneration of linked UI elements. To overcome the same, we have established and implemented certain rules at the platform level itself. In very few cases you may require a slight change in design of the function using the new actions and functions may be useful.

Function $$ExclEvaluate

This function when prefixed to an expression helps in evaluating it without establishing the link with the UI elements.There are very few cases, where the programmer would not want the system to establish the relationship between the caller and the object that is being accessed to refresh the value in the subsequent modification.In such cases, prefixing $$ExclEvaluate would indicate the system the same.

Example:

```
[Variable: My Variable]
  Type: String
    [Field: My Field]
      Type  : String
```

```
          Set as : $$ExclEvaluate:##MyVariable
```

OR

```
          Set as : $$ExclEvaluate:$$MyUserFunction

   [Function: My User Function]
     Returns     : String
     01 : LOG    : ##MyVariable
     02 : RETURN : “Constant String”
```

Action START SUB ... END SUB

In evaluation mode the dependent regenerations of UI elements are deferred till the function exit. In cases where we would like to trigger regenerations based on the set of statements as and when they occur, we can enclose those within the block START SUB and END SUB Actions

To articulate this,

Let's take the above example, where the Variable is being accessed by a field.

Following function on a button press changes the value of the Variable two times.

```
[Function: My User function]

   01 : SET : My Variable : “First Value”
   02 : SET : My Variable : ##MyVariable + “, Second Value”
```

In normal scenario, as both SET actions are modifying the value of the variable, the field (dependent on this variable) would get re-valuated twice. However, the platform has the ability to do it only once during the end of the function by default, when the function is called in EVALUATION mode. To change this behavior to refresh the field twice, we can cover these two SET actions inside a START SUB and END SUB as below.

```
[Function: My User function]

   01 : START SUB
   02 : SET : My Variable : “First Value”
   03 : SET : My Variable : ##MyVariable + “, Second Value”
   04 : END SUB
```

Action SUB ACTION

The purpose of the above action is same as START SUB and END SUB. The only difference is that this action takes an Action to be executed as parameter. The former one encloses a set of Actions inside the block.

Following is the alternative of above code by using SUB ACTION rather than using the SUB ACTION START and END BLOCK.

```
[Function: My User function]

   01 : SUB ACTION : SET : My Variable : “First Value”
   02 : SUB ACTION : SET : My Variable : ##MyVariable + “, Second Value”
```

Action START XSUB ... END XSUB

In execution mode the dependent regenerations are handled as an when they occur. In cases, where we would like to defer regenerations based on the set of statements, we can enclose those within the block X SUB START ... X SUB END

Let's take an example below to demonstrate this

```
[Field: My Field]

   Set as  : $Value1 + $Value2
;; field depends on the Value1 and  Value2 of the current object

[Function: ModifyCurrentObj]

   01 : SET VALUE : Value1 : “Something else”
   02 : SET VALUE : Value2 : “Another value”
```

The above code would normally cause the field to be re-evaluated twice during the execution of the function..However enclosing it in a XSUB block would convert it into a single re-evaluation as below

```
[Function: ModifyCurrentObj]

   01 : XSUB START
   02 : SET VALUE : Value1 : “Something else”
   03 : SET VALUE : Value2 : “Another value”
   04 : XSUB END
```

Action–XSUB ACTION

The purpose of the above action is same as XSUB START and XSUB END. The only difference is that this action takes an Action to be executed as parameter. The former one encloses a set of Actions inside the block.

Following is the alternative of above code by using XSUB ACTION rather than using the XSUB START and XSUB END BLOCK.

```
[Function: ModifyCurrentObj]

   02 : XSUB ACTION :SET VALUE : Value1 : "Something else"
   03 : XSUB ACTION: SET VALUE : Value2 : "Another value"
```

4.9 Functions and Attribute Enhancements

Attribute – Plain XML

Tally provides the capability to export any report in XML format. The XML generated is in standard format for better readability. i.e. line ending characters after each closing tag, indentation for each sub tag etc. Most of the applications can directly consume the data available in standard format. However there are some legacy and non-standard applications which require an XML without formatting and applied styles. They consume the entire unformatted XML available as a single string without even a new line character.

A new attribute Plain XML is introduced in Report definition. This attributes generates the XML without applying any formats and styles.

Syntax

```
      Plain XML : <Logical Expression>
```

Where,

<Logical Expression> can be any expression which evaluates to logical value Yes/No

Example:

```
[Report: Simple Trial balance]
   Form        : Simple Trial balance
   Title       : "Trial Balance"
   Plain XML   : YES
```

Attribute – Format for Quantity Datatype

In Tally, the quantity of a Stock item can be expressed using a Simple or a Compound Unit of Measure.

Simple Unit – The unit of measure which is used to express the quantity of an Item. Examples of Simple units are kgs, nos, pcs etc.

Compound Unit – The unit of measure which is a combination of Simple units related to each other by a conversion factor is termed as a Compound Unit. Examples of compound units are kg of 1000 gms, dozen of 12 nos,. In case of a compound unit the highest unit is referred to as the Base/Primary unit and the sub units thereafter are referred to as the Tail units. The quantity is always expressed in terms of the Primary unit. A compound unit can be nested further to contain another compound unit as a Tail unit upto any levels. Eg Bag of 10kgs of 1000 gms.

For eg: If the unit of measure used for a Stock item “Grains” is Bag of 10kgs of 1000 gms and the closing balance is 12-5-250 bags. This means that the quantity of the items is 12 bags 5 kgs 250 gms. Whenever the tail unit quantity crosses the conversion factor it adds up to the bigger unit. If the gms exceeds 1000 in above example and value is 12-5-1250 bags then this will be converted to 12-6-250 bags.

In TDL, the datatype to support the representation and storage of the data of above type is “Quantity”. This datatype comprises of the subtypes Number, Base/Primary units, Alternate/Secondary units and unit symbol.

As we know when a method of type quantity is retrieved in a report, it is always expressed in terms of the primary unit. In case of the Unit of Measure is a nested compound unit, the user may require the quantity in terms of any of the units in the entire Compound unit chain.

The format attribute of the field has been enhanced to specify the Tail unit in which the quantity value needs to be extracted.

Syntax

```
[Field: <Field Name>]
      Type         : Quantity
      Set As       : $<Method Name>
      Format       : "Tail Units:" + <String Expression>
```

Where,

String Expression must evaluate to any Tail Unit Name used in the Item.

Example:

As per example taken previously, the unit of measure used for a Stock item “Grains” is Bag of 10kgs of 1000 gms and the closing balance is 12-5-250 bags. In a field, we may require to retrieve the value in kgs or gms instead on bags, the following specification can be used.

```
[Field: Qty Format Enhancement]
   Use       : Qty Primary Field
   Set As    : $ClosingBalance
   Format    : “Tail Units:” + “kgs”
```

- If the Format is “Tail Units:kgs” the value returned would be 125 kgs 250 gms =12 X10 kgs+5kgs and 250 gms .
- If the Format is “Tail Units:gms” the value returned would be 125250 gms = 12X10X1000 gms+ 5X1000 gms +250 gms.

Approprite conversions take place as per the conversion factors set in the nested Compound unit chain

Attribute – Cell Write

When the data is exported from an external application to Excel Format, especially in the following scenarios Excel faces refresh issues. Here we are considering the scenarios when Tally exports the data into Excel.

1. If a cell in an Excel Template is having a formula which depends on multiple cells which are being written from Tally. If, one out of these cells is having drop-down list then the excel formula is not refreshed after the Export.
2. If the design of Excel template is depending on one of the Excel cell and this cell is written by Excel Export from Tally then the template using the contents of this cell will not take these changes into effect.

This problem can be addressed at the TDL level by writing those data corresponding to cells prior to on which rest of the cells containing the formula/template are dependent. The rest of the data can be written as a chunk only.

For this purpose new attribute '**Cell Write**' is introduced at **Field**. This attribute enables writing of the specific field value in the Excel file, before the entire information gets written.

Notes

This attribute has to be used judiciously and strictly as per the above scenarios, since this will increase the export time by multifold.

Syntax

```
Cell Write  :<Logical Value>
```

Where,

<Logical Value> can be YES or NO

Example:

```
[Field: VAT acc Rate Fld]
   Cell Write : YES
```

Function – $$StrByCharCode

Everyone is aware that Indian government recently launched a symbol to represent Indian currency. To display the same in Tally.ERP9 a function $$StrByCharCode is introduced in TDL. The function $$StrByCharCode accepts the ASCII code or Unicode and displays the corresponding special symbol. This function can be used in scenarios where the special symbols are to be displayed in Tally.ERP9 e.g. foreign currency symbol .

Syntax

```
$$StrByCharCode:<ASCII code/ Unicode>
```

Where,

<ASCII code/Unicode> can be any expression which returns a valid ASCII or Unicode number.(This number must be in decimal system).

For example the ASCII code for the new rupee symbol is 8377, for Carriage Return is 13 etc.

Example:

```
[Field: StrByCharCode Report]
   Set AS    : $$StrByCharCode:@@CodeChar

[System: Formula]
   Code Char: 8377
```

The new Rupee symbol is displayed in the field ‘StrByCharCode Report’.

Function – $$InPreviewMode

In the scenarios, where the printing events Before Print and After Print are used to trigger an Action, the action was called even if the report is in preview mode. To overcome this problem the function $$InPreviewMode is introduced, using which events can be triggered conditionally as required.

The function $$InPreviewMode checks if the report is in preview mode or not. This function is useful in the scenarios where some specific controls are to be applied related to actual Printing.

For example, a document can be printed only once, after printing an Invoice the voucher cannot be altered or deleted etc.

Syntax

```
      $$InPreviewMode
```

Example:

```
[#Report: Printed Invoice]
   On : Print : NOT $$InPreviewMode : CALL : UpdateDocSetPrintedFlag
```

In this case the Action created using function “UpdateDocSetPrintedFlag” is triggered only in Print mode and not in preview mode.

Function – $$RemoteUserId

In a remote environment multiple users connect to the same company and access the data therein. All the TDLs available at the server are enabled for the Remote user. There may be scenarios where some restrictions need to be applied to the data access based on the user identity. This can be achieved at the TDL level by using a new function $$RemoteUserId which will return the user name of the remote user accessing the TDL.

Syntax

```
      $$RemoteUserId
```

This function when called in the TDL will return the user name of the remote user at the Server end.

Function – $$InWords Enhancements

Till now the function $$InWords accepted only amount data type and displayed the amount in words. Now the function $$InWords is extended to accept Number data type as well and display it in words.

Syntax

```
      $$InWords:<Expression> : <Format String>
```

Where,

<Expression> can be any expression which evaluates to an Amount or Number

<Format String> is any string expression which is used to specify the format e.g. Forex, No Symbol etc.

Example:

```
[Field: InWords]
   Set as : $$InWords:100000
```

The function displays '"ONE LAKH" in the field 'Inwords'.

Function – $$ContextKeyword Enhancements

Till now the function $$ContextKeyword was used to return the Title of the Report or Menu. In the scenarios, like adding a report in the list of favorites, where the definition name of the current report was required instead of the report Title. The function $$ContextKeyword is enhanced to return Report name or Definition name.

Now the function $$ContextKeyword accepts two logical parameters as follows:

```
      $$ContextKeyword: [:<1st Logical Expression>]+
                        [:<2nd Logical Expression>]
```

Where,

<1st Logical Expression> can be any expression which evaluates to Yes/No. The default value of the first parameter is No and it returns the Title of the current report. If the value is specified as YES, then the title of the parent report is returned. If no report is active then the parameter is ignored. If the attribute Title is not specified in report definition, then by default it returns the name of report definition.

<2nd Logical Expression> can be any expression which evaluates to Yes/No. It specifies that the name of the report definition should be returned instead of the Title of the report.

Example:

```
[Field: Context Keyword Rep]
   Set As: $$ContextKeyword:No:Yes
```

The function returns the name of the current report definition.

Example:

```
[Field: Context Keyword Parent]
   Set As: $$ContextKeyword:Yes:Yes
```

The function returns the name of the parent Interface definition i.e., either a Menu Definition Name or Parent Report Definition Name.

5. Enhancements in Release 2.0

5.1 TDL Procedural Enhancements

With every Release the TDL Procedural Capabilities are getting strengthened at a commendable pace. The latest along this path is the File Input Output Capability.

TDL Procedural File Input/Output Capabilities

With every new Release of Tally.ERP 9 we are enriching our language to incorporate procedural capabilities in the most unique way. The latest in this is the Procedural File Input Output capability.

As we are aware, any High level programming language will support Reading and Writing From/To multiple hardware devices. It will have predefined constructs in form of functions to Read and Write from a File as well. This file can reside on the hard disk or a network which can be accessed via various protocols HTTP or FTP.

This capability introduced in TDL now will pave the way for supporting import/export operations from the Tally DataBase in the most amazing way. It will now be possible to export almost every piece of information in any Data Format which we can think of. We support Text and Excel Format which allow data storage in SDF-Fixed Width, CSV-comma separated etc sufficing the generic data analysis requirements of any business.

The TDL Artifacts used for supporting various Read/Write operations are Actions and Functions. These are made to work only inside the TDL Procedural Block. Write operations are mostly handled using Actions and all file Access and Read operations are performed using Functions. These give tremendous control in the hands of the programmer for performing the data manipulations To/From the file. And that too on a file present on a network accessible using the protocols FTP and HTTP. Since these artifacts operate on characters and not bytes the file encoding ASCII/UNICODE does not have any effect on the operations.

The entire procedural Read/Write artifacts basically operate on two file contexts.

- **Source file Context**

When a file is opened for Reading purpose, the context available for all read operations is the Source File Context. All the subsequent read operations a performed on the Source File Context.

- **Target file Context**

When a file is opened for Writing purpose, the context available for all write operations is the Target File Context. All the subsequent Write operations are performed on the Target File Context.

It is important to understand that these contexts are available only inside the procedural block(User Defined Function) where the files are opened for use. The file context concept is different from the concept of Object Context where the Object context is carried over to all its child Objects. File Contexts are only available to the functions and subsequent functions called from within the parent Function. The caller function can override the inherited context by opening a new file within its block.

The files contexts created by opening a file is available only till the execution of the last statement. Once the control is out of the function the file is automatically closed. However, it is highly recommended as a good programming practice to close a file on exit.

Both the file contexts ie Source and Target file are available simultaneously. This makes it possible to read from one file and write to another file simultaneously.

Various File Operations

A programming language which supports File Read/Write typically support following fundamental operations

- Open - This is an operation which identifies the file which needs to be opened for Read/ Write purpose
- Close - This is an operation which closes the opened file after Read/Write
- Read - This is an operation to read the data from an opened File
- Write - This is an operation to write the data to an opened File
- Seek - This is an operation to the character position in an already opened file
- Truncate - This is an operation which will delete the particular no of characters/entire contents of the file

The TDL Artifacts used for supporting various Read/Write operations are Actions and Functions. These are made to work only inside the TDL Procedural Block. Write operations are handled using Actions and all file Access and Read operations are performed using Functions.

General File Operations

As discussed above the entire procedural Read/Write concepts basically operate on two file contexts. A source file context and a target file context. Source context is used to read the contents from a file which is opened for reading purpose, where as the target context is used to write the data to a file which is opened for writing purpose. Since both these file contexts are available simultaneously it is possible to read from one file and write to another file.

- Action – OPEN FILE

This action is used open a text/excel file for read/write operations. The File can reside in Hard Disk, can be in main memory or can be in FTP/HTTP site. Also this Action is used to open a file for read/write operations.

If no parameters are used then a memory buffer will be created and will act as a file. This file will be in both read / write mode and will act as both source as well as target context.
Based on the mode specified (read / write) the file automatically becomes the source file or target file respectively.

Syntax

```
OPEN FILE [: File Name [: File Format [: Open Mode [: Encoding ]]]]
```

Where,
<File Name> File name can be a expression which evaluates to a regular disk file name like C:\Output.txt or to a HTTP/FTP site like “ftp://ftp.tallysolutions.com/Output.txt”

<File Format> This can be one of Text or Excel. By default text mode will be considered if not specified. Also during opening an existing file, if the mode does not match the Action will fail.

<Open Mode> This can be read / write. By default it is read. If a file is opened for read purpose then it must exist. If write mode is specified, and the file exists this will be open it for updating or if file does not exist a new file is created. If the file is opened in the Write mode, it is also possible to read from the file as well.

<Encoding> This can be ASCII or Unicode. If not specified it will consider Unicode as default value for Write Mode. In read mode this parameter will ignored and considered based on the encoding of the file being read.

Example 1:

The below Action opens a Text File 'Output.text' in Write mode under Tally application Folder and if it is already exists the same file will be opened for appending purpose.

```
    10 : OPEN FILE : “Output.txt” : Text : Write   : ASCII
```

Example 2:

The below Action opens a Text File 'Output.text' in Write mode at the HTTP URL specified . If the file already exists the same file will be opened for appending purpose.

```
    10: OPEN FILE: “http://www.tallysolutions.com/Output.txt” : Text
```

Example 3:

The below Action opens a Excel File 'Output.xls' in Read mode under C drive and if the file does not exist the Action will fail.

```
    10 : OPEN FILE : “C:\Output.xls” : Excel : Read
```

Please refer to the functions like $$MakeFTPName and $$MakeHTTPName for constructing the FTP and HTTP URLs using the various parameters like servername,username,password etc.Refer to Tally.Developer 9 Function Browser for help on usage.

- Actions – CLOSE FILE and CLOSE TARGET FILE

A file which is opened for Read/Write operation needs to be closed once all the read/write operation is completed. However if the files are not closed explicitly by the programmer, these are closed by default when the function is returned. But it is always recommended to close the file after the operations are completed.

Files which are opened in the current function can only be closed by it. If a function inherits file contexts from the caller function then it cannot close these files, however it can open its own instances of the files, in such cases the caller context files will not be accessible.

- Action – CLOSE FILE

This action is used to close an opened source file

Syntax

```
        CLOSE FILE
```

Example:

In the example below Excel file 'Output.xls' which is opened for reading purpose is closed.

```
   10   : OPEN FILE : “Output.xls” : Excel : Read
          .
          .
          .
   30   : CLOSE FILE
```

- Action – CLOSE TARGET FILE

This action is used to close an opened target file

Syntax

```
       CLOSE TARGET FILE
```

Example:

In below example the text file ‘Output.txt’ which is opened for writing purpose is closed.

```
   10 : OPEN FILE: “Output.txt” : Text : Write
          .
          .
          .
   30 : CLOSE TARGET FILE
```

General Functions

- $$TgtFile

All file accessing functions for both text and excel files, operates on the source file context. The function $$TgtFile can be used to switch to target file context temporarily. This function which evaluates the expression passed in the context of a target file.

Syntax

```
      $$TgtFile:Expression
```

Example:

In the example below, the objective is to Read the content of a cell in Sheet 1 and copy it to a cell in the Sheet 2 of the same file. The function opens the File “ABC.xls” in Write mode.

```
[Function: Sample Func]

   Variable : Temp : String
```

```
10  : SET        : Temp : ""
20  : OPEN FILE  : "Output.xls" : Excel : Write
30  : ADD SHEET  : "Sheet 1"
40  : WRITE CELL : 1 : 1 : "Item A"
50  : SET        : Temp : $$TgtFile : $$FileReadCell:1:1
60  : ADD SHEET  : "Sheet 2"
70  : WRITE CELL : 1:1: ##Temp
80  : CLOSE TARGET FILE
```

In this example there is no file present in the source file context as the file is opened in the Write mode. In that case, for reading a value from Sheet 1, the expression $$FileReadCell:1:1 will return no value. Prefixing the expression with $$Tgtfile will temporarily change the context to TargetFile for the evaluation of expression and will fetch the value from cell 1 of Sheet 1 existing in the TargetFile context.

- $$FileSize

This function will return the size of the file specified in bytes. This function takes optional parameter if the parameter is not given then it works on the current context file and returns the size.

Syntax

```
$$FileSize[:FileName]
```

Where,
FileName is an expression which evaluates to the file name along with the path.

Example:

Below example gives the size of Excel file 'output.xls' in terms of bytes

```
10 : Log : $$FileSize: "Output.xls"
```

Read/Write Operation on Text Files

- Writing to a File

Various Actions are introduced in order to write to a text file. These Actions operate on the Target File context. The scope of these Actions is within the TDL procedural block(User Defined Functions) where the file is opened and the context is available.

- Action – WRITE FILE

This Action is used to append a file with the text specified. The write always starts from the end of the file. This Action always works on the target file context.

Syntax

```
WRITE FILE : <TextToWrite>
```

Where,

<TextToWrite> can be any expression which evaluates to data that need to be written to the file.

Example:

In the example below a txt file ‘Output.txt’ is opened in write mode and the content ‘Krishna’ is appended at the end of the file.

```
10   : OPEN FILE   : “Output.txt” : Text : Write
20   : WRITE FILE : “Krishna”
30   : CLOSE TARGET FILE
```

- Action – WRITE FILE LINE

This Action is similar to WRITE FILE but it also places a new line character (New Line/Carriage Return) after the text. All the subsequent writes begin from the next line.This Action always works on the target context.

Syntax

```
WRITE FILE LINE: <TextToWrite>
```

Where,
<TextToWrite> can be any expression which evaluates to data that need to be written to the file.

Example:

In the example below a txt file 'Output.txt' is opened in write mode and the two more lines were appended to the end of the file.

```
10   : OPEN FILE       : “Output.txt” : Text : Write
20   : WRITE FILE LINE: “Line 1”
30   : WRITE FILE LINE: “Line 2”
40   : CLOSE TARGET FILE
```

- Action – TRUNCATE FILE

This action removes the content of the file by setting the file pointer to the beginning of the file and inserting an end of file marker. This can be used to erase all characters from an existing file before writing any new content to it.

Syntax

```
TRUNCATE FILE
```

Example:

In the Example below the entire contents of the existing txt file 'Output.txt' is removed and 'New Value' is inserted subsequently.

```
10 : OPEN FILE   : “Output.txt” : Text : Write
```

```
20 : TRUNCATE FILE
30 : WRITE FILE : "New Value"
40 : CLOSE TARGET FILE
```

- Action – SEEK FILE

This Action operates on the Target File Context. This Action is used to move the file pointer to a location as specified by the no of characters. As we know that it is possible to Read and Write from the Target File context, all the subsequent Reads and Writes will be starting from this position. By Default, if the file position is not specified Read pointer will be always be from the beginning of file and write pointer will be from the end of the file.

We have already covered how to Read from Target File Context by using the function $$TgtFile

Syntax

```
SEEK FILE: <File Position>
```

Where,

<File Position> can be any expression which evaluates to number which considered as number of characters.

Reading a File

Some functions and Actions are introduced which can operate on the Source File context to read from the file or access some information from them. The scope of these functions is within the TDL procedural block (User Defined Functions) where the file is opened and the context is available. It is also possible to read from the Target File Context by using the function $$TgtFile.

- $$FileRead

This function is used to read data from a text file. This takes an optional parameter. If this is not specified or parameter is having value as 0 this will read one line and ignore the end of line character. However the file pointer is positioned after the end of line character so that next read operation starts on the next line.

If number of characters are mentioned this function will read that many number of characters.

Syntax

```
$$FileRead [:<CharsToRead>]
```

Where,

<CharsToRead> can be any expression which evaluates to number of characters to read

Example:

Below example Read the first line of the text file 'Output.txt'

```
10 : OPEN FILE : "Output.txt" : Text : Read
```

```
20 : LOG       : $$FileRead
30 : CLOSE  FILE
```

- $$FileIsEOF

This function is used to check if the current character being read is the End of file character.

Syntax

```
$$FileIsEOF
```

- Action – SEEK SOURCE FILE

This Action works on a source file context.This action sets the current file pointer to the position specified. Zero indicates the beginning of the file and -1 indicates the end of the file. The file position is determined in terms of the characters. All the subsequent reads begin from this position onwards.

Syntax

```
SEEK SOURCE FILE: <File Position>
```

Where,

<File Position> can be any expression which evaluates to number which is no of characters.

Example:

In the example below the first line of the file 'Output.txt' is read starting from the 3 character.

```
10  : OPEN FILE        : "Output.txt" : Text : Read
20  : SEEK SOURCE FILE : 2
30  : LOG              : $$FileRead
40  : CLOSE FILE
```

Read/Write Operation on Excel Files

For an Excel file all Read and Write operations will be performed on an Active Sheet.

- Action – SET ACTIVE SHEET

This Action is used to change the Active Sheet during read and write operations.

Syntax

```
SET ACTIVE SHEET: <Sheet Name>
```

Where,

<Sheet Name> can be an expression which evaluates to the string and will be considered as the name of the sheet.

Example:

The example opens an Excel sheet Output.xls in Read mode and makes 'Sheet 2' as active and reads the content from the first cell.

```
10 : OPEN FILE        : "Output.xls" : Excel : Read
20 : SET ACTIVE SHEET : "Sheet 2"
30 : Log              : $$FileReadCell:1:1
40 : CLOSE FILE
```

- Writing to a File

Various Actions are introduced in order to write to a excel file. These Actions operate on the Target File context. The scope of these Actions is within the TDL procedural block (User Defined Functions) where the file is opened and the context is available.

- Action – ADD SHEET

This Action adds a sheet in the current workbook which is opened for writing. Sheet will always be inserted at the end .If a sheet with the same name already exists the sheet will be made as active.

Syntax

```
ADD SHEET : <Sheet Name>
```

Where,

<Sheet Name> can be an expression which evaluates to the string and will be considered as the name of the sheet.

Example:

The example below opens an existing Excel sheet 'Output.xls' in write mode and new sheet 'New Sheet' will be inserted at the end of the workbook.

```
10  : OPEN FILE : "Output.xls" : Excel : Write
20  : ADD SHEET : "New Sheet"
```

- Action – REMOVE SHEET

This Action removes the specified sheet from current workbook. The entire content in the sheet will be removed. This Action will fail if the workbook has only one sheet or if the specified sheet name does not exist in the workbook.

Syntax

```
REMOVE SHEET : <Sheet Name>
```

Where,

<Sheet Name> can be an expression which evaluates to the string and will be considered as the name of the sheet.

Example:

The example below creates a work book with a sheet 'New Sheet'

```
10  : OPEN FILE     : "Output.xls" : Excel : Write
20  : ADD SHEET     : "New Sheet"
30  : REMOVE SHEET  : "Sheet1"
```

- Action – RENAME SHEET

This action renames a work sheet.

Syntax

```
RENAME SHEET: <Old Sheet Name> : <New Sheet Name>
```

Where,

<Old Sheet Name> and **<New Sheet Name>** can be an expression which evaluates to the string and will be considered as the name of the sheet.

Example:

The example below renames the existing sheet with new sheet name

```
01 : OPEN FILE     : "Output.xls" : Excel : Write
02 : RENAME SHEET :@@OlSheetName : @@NewSheetName
04 : CLOSE TARGET FILE
```

- Action – WRITE CELL

This Action Writes the specified content at the cell address specified by row and column number of the currently active sheet.

Syntax

```
WRITE CELL: <Row No>: <Column No> : <Content To be Written>
```

Where,

<Row No> and **<Column No>** can be any expression which evaluates the number which can be used to identify the cell 'Content To be Written' can be any expression which evaluates to data which needs to be filled for the identified cell.

Example:

The example opens an Excel File 'Output.xls', adds a new sheet and in that sheet first cell will have content as 'Krishna'

```
10  : OPEN FILE   : "Output.xls" : Excel : Read  : ASCII
15  : ADD SHEET   : "New Sheet"
20  : WRITE CELL : 1 : 1 : "Krishana"
30  : CLOSE FILE
```

- Action – WRITE ROW

This Action writes multiple cell values at a specified row in the Active sheet. The no of values separated by commas are written starting from the initial column no specified for the row specified.

Syntax

```
WRITE ROW:<Row No> :<Initial Column No> : <Comma Separated Values>
```

Where,

<Row No> and **<Initial Column No>** can be any expression which evaluates the number which can be used to identify the cell.

‘Comma Separated Values’ can be expressions separated with comma which evaluates to data that needs to be filled starting from cell as mentioned by 'Row Number' and 'Initial Column Number'

Example:

The examples below fills cell (1,A) and cell (1,B) with the values from expressions 'Val1' and 'Val2'

```
10   : OPEN FILE : “Output.xls” : Excel : Write
20   : ADD SHEET : “New Sheet”
30   : WRITE ROW : 1 : 1 : @@Val1, @@Val2
40   : CLOSE TARGET FILE
```

- Action – WRITE COLUMN

This Action writes multiple cell values at a specified column in the Active sheet. The no of values separated by commas are written starting from the initial row no specified for the column.

Syntax

```
WRITE COLUMN: <Initial Row No>: <Column No>: <Comma Separated Values>
```

Where,

<Initial Row No> and **<Column No>** can be any expression which evaluates the number which can be used to identify the cell.

‘Comma Separated Values’ can be expressions separated with comma which evaluates to data that needs to be filled starting from cell as mentioned by 'Initial Row Number' and 'Column Number'

Example:

The example below fills cell (5, E) and cell (6,E) with the values from expressions ‘Val3’ and ‘Val4’

```
10   : OPEN FILE     : “Output.xls” : Excel : Write
20   : ADD SHEET     : “New Sheet”
30   : WRITE Column : 5 : 5 : @@Val3, @@Va4
```

```
40   : CLOSE TARGET FILE
```

- Reading a File

Some functions and Actions are introduced which can operate on the Source File context to read from the file or access some information from them. The scope of these functions is within the TDL procedural block(User Defined Functions) where the file is opened and the context is available. It is also possible to read from the Target File Context by using the function $$TgtFile.

- $$FileReadCell

This function returns the content of the cell identified by using row and column number of the active sheet

Syntax

```
$$FileReadCell: <Row No>: <Column No>
```

Where,

<Row No> and **<Column No>** can be expression which evaluates to the number to identify row number and column number

Example:

The Function $$FileReadCell Logs the contents of the first cell of the excel sheet ‘Sheet 1’

```
10 : OPEN FILE         : “Output.xls” : Excel : Read
20 : SET ACTIVE SHEET : “Sheet 1”
20 : Log               : $$FileReadCell : 1 : 1
```

- $$ FileGetSheetCount

This function returns number of sheets in the current workbook.

Syntax

```
$$GetSheetCount
```

Example:

The Function $$FileGetSheetCount returns the total number of sheets in an Excel sheet 'Output.xls'

```
10 : OPEN FILE : “Output.xls” : Excel : Read
20 : Log       : $$FileGetSheetCount
```

- $$ FileGetActiveSheetName

This function returns name of the active sheet

Syntax

```
$$FileGetActiveSheetName
```

Example:

The Function $$ FileGetActiveSheetName returns the name of the Active sheet after opening the Excel file ‘Output.xls’

```
10 : OPEN FILE: “Output.xls” : Excel : Read
20 : Log      : $$FileGetActiveSheetName
```

- $$FileGetSheetName

This Function returns the name of the sheet at a specified index.

Syntax

```
$$FileGetSheetName : <Sheet Index>
```

Where,
<Sheet Index> can be any expression which evaluates to number as Sheet Index

Example:

The Function :$$FileGetSheetName gives the name of the sheet at a given index

```
10  : OPEN FILE: “Output.xls” : Excel : Read
Log : $$FileGetSheetName:1
```

- $$ FileGetSheetIdx

This Function Returns the Index of the sheet for a specified sheet name.

Syntax

```
$$FileGetSheetIdx : <Sheet Name>
```

Where,
<Sheet Name> can be any expression which evaluates to the name of the Excel Sheet

Example:

The Function $$FileGetSheetIdx gives the index number of the sheet name

```
10 : OPEN FILE: “Output.xls” : Excel : Read
20 : Log      :  $$FileGetSheetIdx: “Ledgers”
```

- $$FileGetColumnName

This Function gives column name in terms of alphabets for given index

Syntax

```
$$FileGetColumnName:Index
```

Where,

<Index> can be any expression which evaluates to the Index number

Example:

The Function $$ FileGetColumnName returns value J

```
10   : OPEN FILE: "Output.xls" : Excel : Read
20   : Log       : $$FileGetColumnName:10
```

- $$FileGetColumnIdx

This function returns the index of the column for a given alphabetical name

Syntax

```
$$FileGetColumnIdx: <Name>
```

Where,

<Name> can be any expression which evaluates to name of the column in alphabets.

Example:

The Function $$FileGetColumnIdx returns value as 27

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       :$$FileGetColumnIdx:AA
```

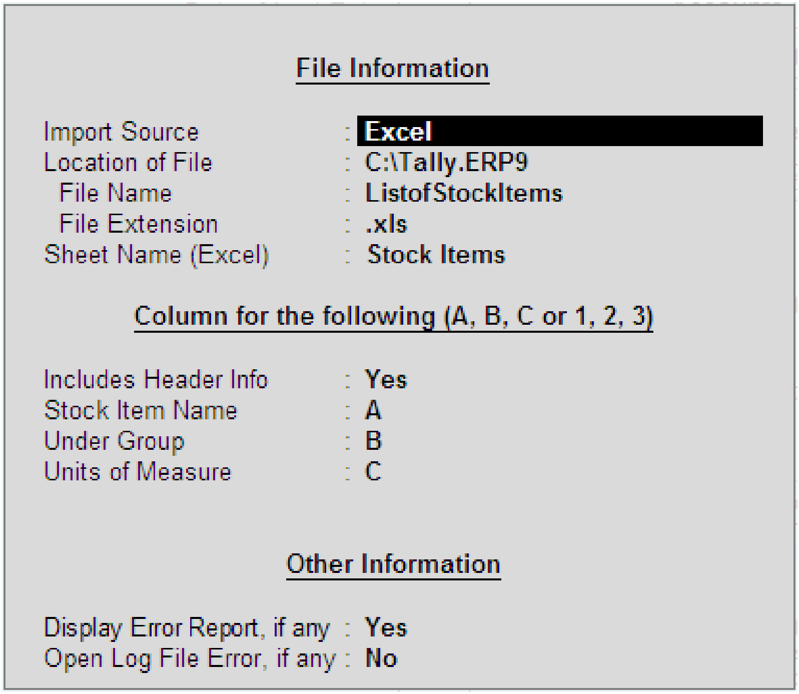
Use Case – Import from Excel

Scenario

ABC Company Limited, who is in to trading business is using Tally.ERP 9. It deals with purchase and sale of computers, printers etc. The company management wants to import the stock items from the Excel sheet or a text file in to Tally.ERP9.

Functional Demo

A configuration report is added in Tally.ERP9 to accept the file location, work sheet name, column details etc. An option to display the error report can also be specified.



File Information

Import Source : Excel
Location of File : C:\Tally.ERP9
File Name : ListofStockItems
File Extension : .xls
Sheet Name (Excel) : Stock Items

Column for the following (A, B, C or 1, 2, 3)

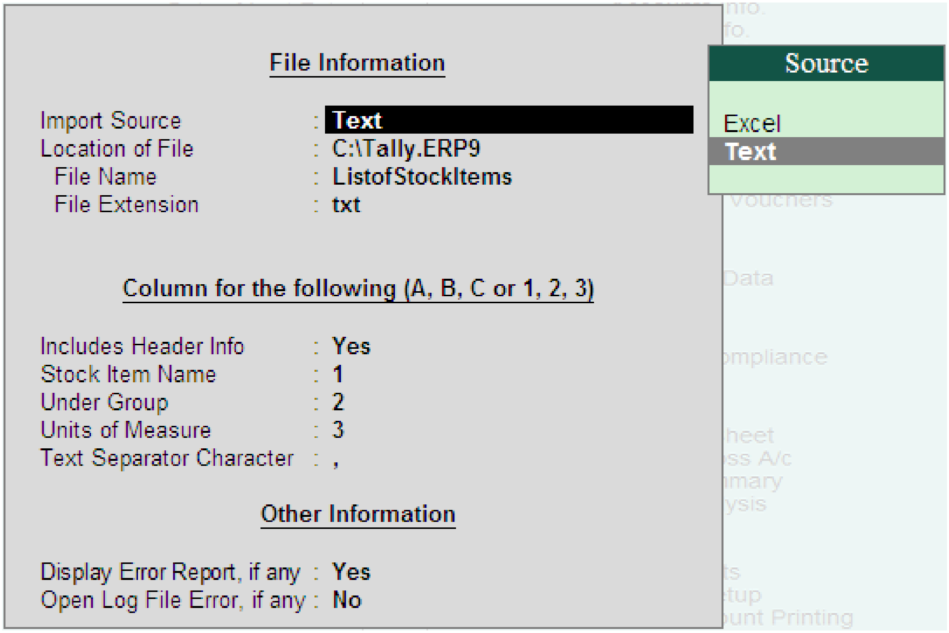
Includes Header Info : Yes
Stock Item Name : A
Under Group : B
Units of Measure : C

Other Information

Display Error Report, if any : Yes
Open Log File Error, if any : No

Figure 1.6 The Configuration Report

By default Excel format is selected. But the user can also select the Import source format as Text and specify the file details. The text separator character should be specified as well in addition to the column details.



File Information

Import Source : Text
Location of File : C:\Tally.ERP9
File Name : ListofStockItems
File Extension : txt

Column for the following (A, B, C or 1, 2, 3)

Includes Header Info : Yes
Stock Item Name : 1
Under Group : 2
Units of Measure : 3
Text Separator Character : ,

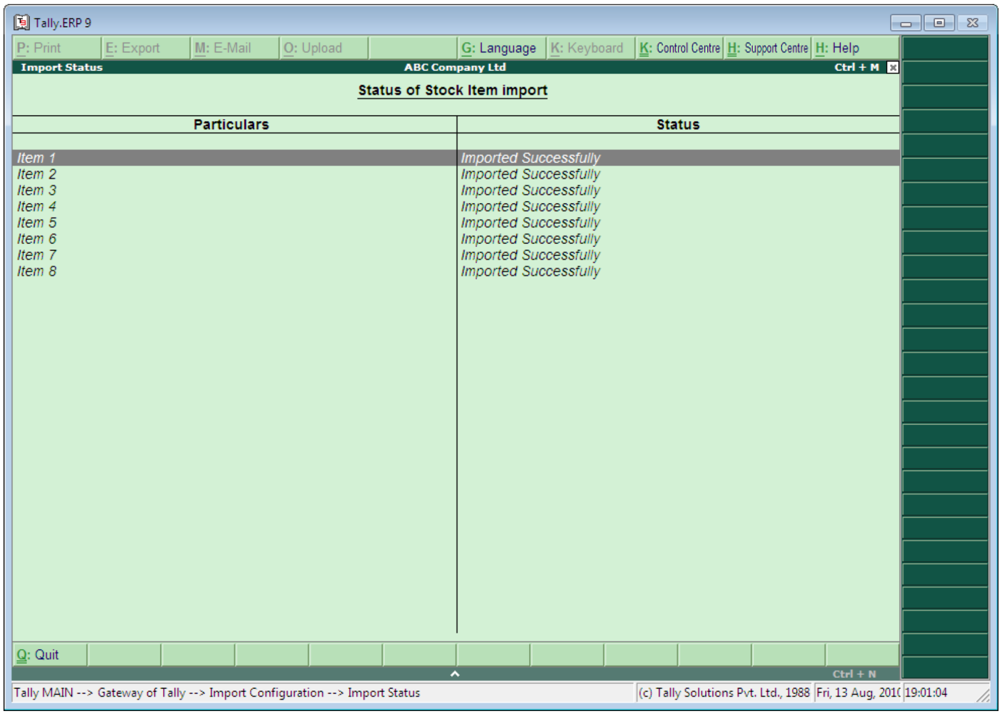
Other Information

Display Error Report, if any : Yes
Open Log File Error, if any : No

Figure 1.7 The Configuration Report

Once the details are entered a confirmation message is displayed to start the import process.

If the user has selected the option to display the error report the after the successful import, the report is shown with the imported stock items and status as “Imported successfully” as follows:

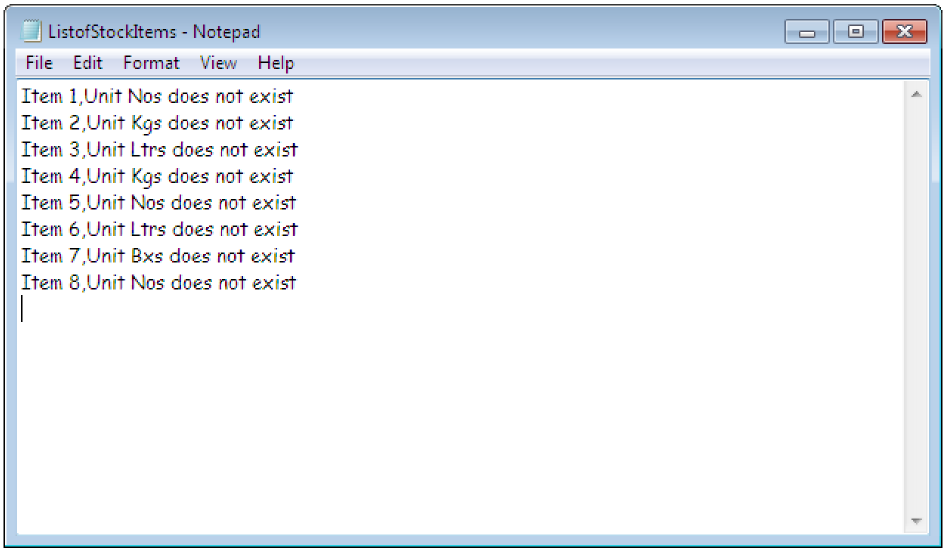


Status of Stock Item import

Particulars	Status
Item 1	Imported Successfully
Item 2	Imported Successfully
Item 3	Imported Successfully
Item 4	Imported Successfully
Item 5	Imported Successfully
Item 6	Imported Successfully
Item 7	Imported Successfully
Item 8	Imported Successfully

Figure 1.8 Success Report

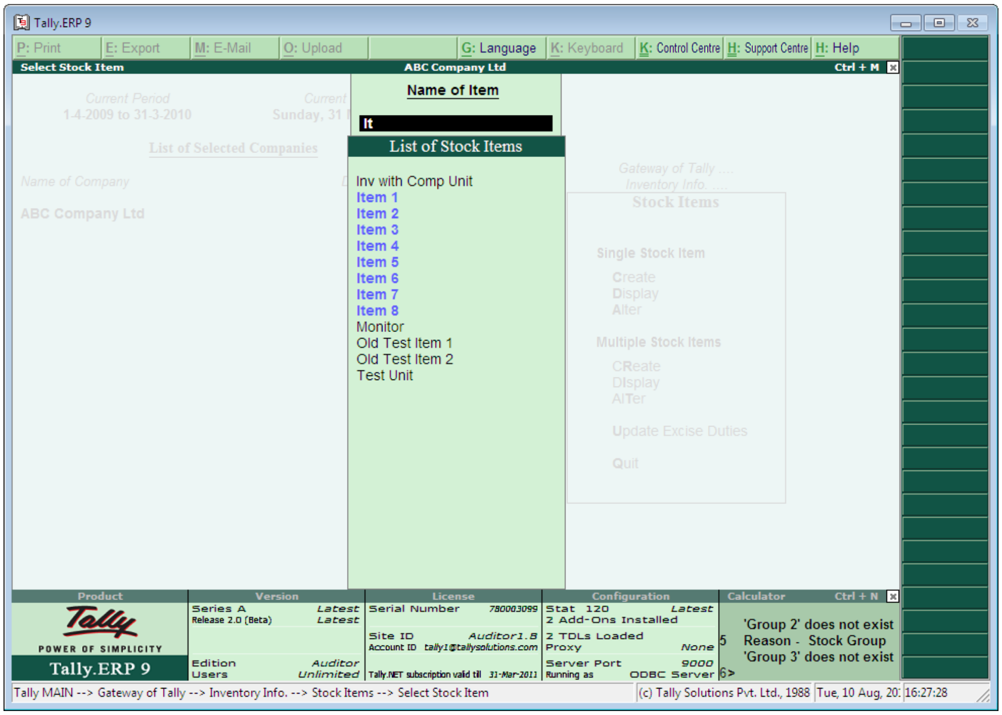
If the user has selected the option to display the Log file, then after the import the log file is displayed as shown:



```
Item 1,Unit Nos does not exist
Item 2,Unit Kgs does not exist
Item 3,Unit Ltrs does not exist
Item 4,Unit Kgs does not exist
Item 5,Unit Nos does not exist
Item 6,Unit Ltrs does not exist
Item 7,Unit Bxs does not exist
Item 8,Unit Nos does not exist
```

Figure 1.9 Log File

The imported items are now available in the Stock Item list as shown:



Name of Item

It

List of Stock Items

Inv with Comp Unit
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Monitor
Old Test Item 1
Old Test Item 2
Test Unit

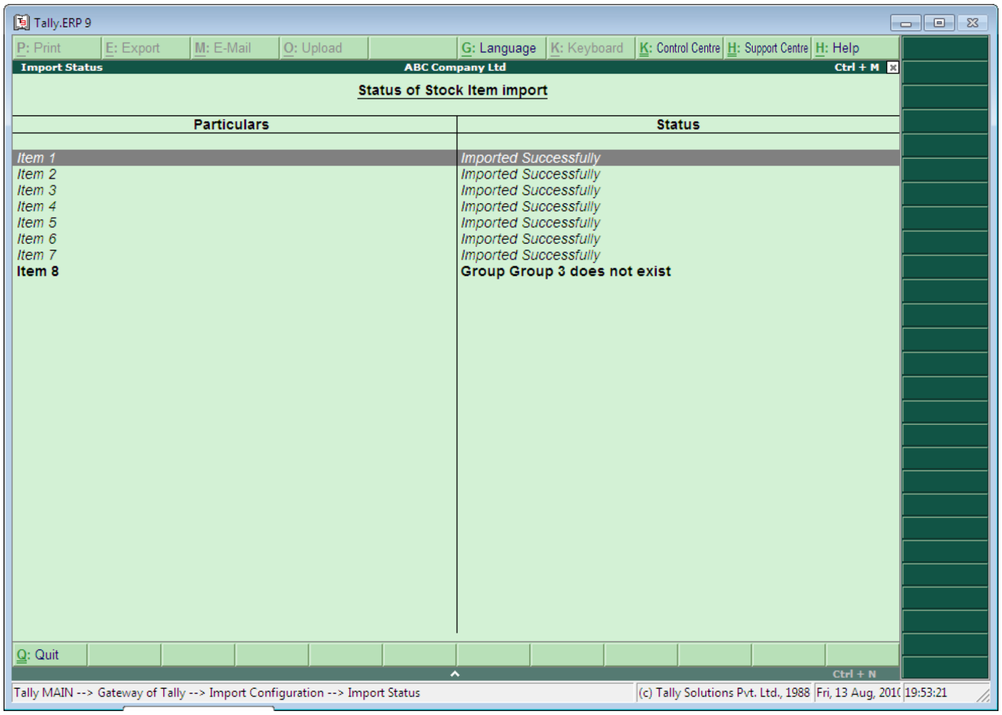
Series A Latest
Release 2.0 (Beta) Latest
Edition Auditor
Users Unlimited
Serial Number 780003099
Site ID Auditor1.8
Account ID tally1@tallysolutions.com
Tally.NET subscription valid til 31-Mar-2011
Stat 120 Latest
2 Add-Ons Installed
2 TDLs Loaded
Proxy None
Server Port 9000
Running as ODBC Server

5 'Group 2' does not exist
Reason - Stock Group
'Group 3' does not exist
6>

Tally MAIN --> Gateway of Tally --> Inventory Info. --> Stock Items --> Select Stock Item
(c) Tally Solutions Pvt. Ltd., 1988 Tue, 10 Aug, 20: 16:27:28

Figure 1.10 List of Stock Items

In case the import is unsuccessful and then the error report is displayed with the reason to failure is displayed as follows:



Import Status

ABC Company Ltd

Status of Stock Item import

Particulars	Status
Item 1	Imported Successfully
Item 2	Imported Successfully
Item 3	Imported Successfully
Item 4	Imported Successfully
Item 5	Imported Successfully
Item 6	Imported Successfully
Item 7	Imported Successfully
Item 8	**Group Group 3 does not exist**

Tally MAIN --> Gateway of Tally --> Import Configuration --> Import Status (c) Tally Solutions Pvt. Ltd., 1988 Fri, 13 Aug, 201(19:53:21

Figure 1.11 Error Report

Solution Development

The import from the excel file is simplified as the user can specify the import details. The file I/O capabilities are used to develop the solution.

The steps followed to achieve the requirement are:

3. A report is designed to display the configuration report. The value entered by the user is stored in system variable.

```
Local: Field: Name Field: Modifies: SIC Source: Yes
Local: Field: Name Field: Variable: SIC Source
         |
         |
Local: Field: Name Field: Modifies: SIC DirPath: Yes
Local: Field: Name Field: Variable: SIC DirPath

[System: Variable]
  SIC Source : “Excel”
  SIC DirPath: “C:\Tally.ERP9”
```

4. On form accept the function to import the stock item is called.

```
On: Form Accept: Yes: Form Accept
On: Form Accept: Yes: Call: Smp Import Stock Items
```

5. A function “Smp Import Stock Items” is defined.

a. In this function first of all the format of the source file is checked and then the action Open File is used to open the file in read mode accordingly.

```
20 :  IF : ##SICSource = “Excel”
30 :     OPEN FILE : @@TSPLSMPTotFilePath: Excel: READ
40 :  ELSE  :
50 :     OPEN FILE : @@TSPLSMPTotFilePath: Text: READ
60 :  ENDIF
```

b. The data from the Excel cells are read and added as an item in the list variable.

```
120: WHILE: NOT $$IsEmpty:($$FileReadCell: ##Row : +
           ##ItemColumns.ItemName)
130:         LIST ADD EX: Item Details
140:        SET : ItemDetails[$$LoopIndex].ItemName: $$FileReadCell: +
                  ##Row:##ItemColumns.ItemName
150:         SET : ItemDetails[$$LoopIndex].ItemGrp: $$FileReadCell: +
                  ##Row: ##ItemColumns.ItemGrp
160:         SET : ItemDetails[$$LoopIndex].ItemUOM: $$FileReadCell: +
                  ##Row: ##ItemColumns.ItemUOM
170:         INCREMENT: Row
180: END WHILE
```

c. If the source format is Text the text file is read line by line and added as an item to the list variable.

```
210: WHILE: NOT $$FileIsEOF
220:    SET: Temp Var: $$FileRead
230:    IF : NOT $$IsEmpty:##TempVar AND (NOT ##SICIncHeader OR +
                          (##SICIncHeader AND $$LoopIndex > 1))
240:       LIST ADD EX: Item Details
250:       SET : ItemDetails[##Counter].ItemName: $$SICExtractDet: +
                  ##TempVar:##ItemColumns.ItemName
260:        SET : ItemDetails[##Counter].ItemGrp: $$SICExtractDet : +
                  ##Temp Var:##ItemColumns.ItemGrp
270:        SET : ItemDetails[##Counter].ItemUOM: $$SICExtractDet: +
                  ##TempVar:##ItemColumns.ItemUOM
280:        INCREMENT: Counter
290:     ENDIF
300: END WHILE
```

d. A collection is populated using the List variable as data source.

```
[Collection: TSPL SMP Imp StockItem]
   Data Source: Variable: Item Details

[Collection: TSPL SMP Imp StockItem Summ]
   Source Collection    : TSPL SMP Imp StockItem
   By    : SICStockItem : $ItemName
   By    : SICStockGroup: $ItemGrp
   By    : SICStockUOM  : $ItemUOM
   Filter: TSPL SMP NonEmpty Item
```

e. Now the Stock Item objects are created. If the item cannot be imported then the item details are written in the error file or compound variable based on the format selected for displaying i.e. Report or Log.

```
380: WALK COLLECTION : TSPL SMP Imp StockItem Summ
390:    SET : Last Status: “”
400:   IF : $$IsEmpty:$Name:StockItem:$SICStockItem
410:         NEW OBJECT: Stock Item
420:         SET VALUE: Name: $SICStockItem
430:         IF : NOT $$IsEmpty:$Name:StockGroup:$SICStockGroup
440:            SET VALUE: Parent: $SICStockGroup
450:         ELSE:
460:           SET:LastStatus:“Group”+$SICStockGroup+“does not exist”
470:         ENDIF

480:         IF : NOT $$IsEmpty:$Symbol:Unit:$SICStockUOM
490:            SET VALUE: Base Units: $SICStockUOM
```

```
500:          ELSE :
510:             SET: LastStatus:"Unit"+$SICStockUOM+"does not exist"
520:          ENDIF
530:          IF : $$IsEmpty:##LastStatus
540:             SAVE TARGET
550:             SET: Last Status: "Imported Successfully"
560:          ENDIF
570:    ENDIF
;; Writing Import Status to the LOG File if LOG File is to be displayed at the end
580:    IF : ##SICOpenLogFile
590:      WRITE FILE LINE: $SICStockItem + ##SICTextSep + ##LastStatus
600:    ENDIF
;; Updating List of Compound Variables is Status is to be displayed in a Report
610:    IF: ##SICDisplayReport
620:      LIST ADD EX: ItemImportStatus
630:      SET: ItemImportStatus[##Counter].ItemName: $SICStockItem
640:      SET: ItemImportStatus[##Counter].Status: ##LastStatus
650:      INCREMENT: Counter
660:    ENDIF
670: END WALK
```

f. If the format selected is Report then the stock item name and the status is updated in a compound variable. Where as if the format selected is Log file then the action write file is used to write in the file.

```
WRITE FILE LINE : $SICStockItem + ##SICTextSep + ##LastStatus
```

g. After the import, if the user has selected to display the error report, the function is called to display the same.

```
690: IF : ##SICDisplayReport
700:       DISPLAY : TSPL Smp SIC Error Report
710: ENDIF
```

h. After the import, if the user has selected to display the log file then the log file is displayed.

```
720 : IF : ##SICOpenLogFile
730 :       EXEC COMMAND : @@TSPLSmpErrorFilePath
740 : ENDIF
```

4. The Error Report displays the reason of failure if the Stock Item cannot be imported. In the error report the line is repeated over the collection populated using the list variable as the data source.

Function Parameter Changes – Optional Parameters

Prior to this Release, while invoking a user defined function, it was mandatory to pass values to all the parameters declared within the function.We have now introduced the capability to have optional parameters. The function will execute smoothly even if the caller of the function does not pass the value to these optional parameters. However, caller of the function must pass all the mandatory parameters. Only rightmost parameters can be optional. i.e., any parameter from the left or middle cannot be optional.

If the Parameter value is supplied by the calling function, the same is used else the default Parameter value provided within the Parameter Attribute is used as the Parameter value.

For this enhancement, Function attribute Parameter is modified to accept parameter value.

Syntax

```
[Function: <Function Name>]
   Parameter: <Parameter Name1> : <Data Type>
   Parameter: <Parameter Name2> : <Data Type>
   Parameter: <Parameter Name3> : <Data Type> [: Parameter Value]
   Parameter: <Parameter Name4> : <Data Type> [: Parameter Value]
```

Where,

<Parameter Name1> and **<Parameter Name2>** are mandatory parameters for which values must be passed while calling the function.

<Parameter Name3> and **<Parameter Name4>** are optional parameters for which values may or may not be passed while calling the function. If values for these parameters are passed, it ignores the parameter value specified within the Parameter Attribute and in absence of these values, specified parameter value is taken into consideration.

Notes

Parameter Value *indicates* ***Optional Parameters*** *and all the* ***Optional Parameters*** *should be the rightmost elements of the function.*

Example:

```
[Function: Split VchNo]
;; this Function returns number part of voucher number from a string
;; For e.g., Voucher Number is Pymt/110/2010-11.  This Function will return only 110.

   Parameter: pVchNo: String
   Parameter: pSplitChar: String: “/”;; usual separator
   00: FOR TOKEN : TokenVar: ##pVchNo: ##pSplitChar
```

```
10:      IF : $$LoopIndex = 2
20:           RETURN: ##TokenVar
30:      ENDIF
40: END FOR
```

While invoking the function **SplitVchNo**, only the Voucher No is passé. 2nd Parameter is optional and default value is “/” which can be passed only if the separator character is other than “/”.

Optional Parameters can be very useful where Parameter values remain constant in most of the cases and rarely require some change.

A small change has been done in a way function parameters are tokenized. The last parameter passed to the function is not broken up into subparts now. This is particularly useful in cases where we require the result of one function to be treated as a parameter to another function. In other words, if a function requires 4 parameters, it tokenizes only till 3 parameters and all the subsequent values are considered as the 4th parameter(last parameter)

5.2 Variable Framework Enhancements

In the prior releases we have experienced major changes to the Variable Framework in form of introduction to Compound Variables and List Variables. We are continuously enhancing and making changes to ensure consistency and uniformity across the TDL framework. The following enhancements have taken place in variable framework recently.

Variable Persistence at Report Scope

Variables at report scope can now be persisted in to a user specified file. This is stored in a standard variable format and also allows reloading the report scope variables from the specified file. The Actions SAVE VARIABLE and LOAD VARIABLE are introduced for this purpose.

SAVE VARIABLE

The action SAVE VARIABLE is used to persist the Report Scope Variables in a user specified file.

Syntax

```
SAVE VARIABLE : <FileName> [:<Variable List>]
```

Where,

<FileName> is the name of file in which the report scope variables are persisted. The extension .PVF will be taken by default if the the file extension is not specified.

<Variable List> is the list of comma separated variables that need to be persisted in the file. Specifying the variable list is optional.

If the Variable List is not specified, all variables at the report scope which have Persist attribute set to YES will be persisted in the specified file.

You need not declare the variable at System level unless it is required to persist the same in the default configuration file tallycfg.tsf.

Example:

Let us assume that the variables **EmpNameVar** and **EmpIDVar** are declared at the Report Scope and the same needs to be persisted in a user specified file. We can achieve this using the newly introduced actions **SAVE VARIABLE** and **LOAD VARIABLE**. The buttons SAVEVAR and LOADVAR are added at the Form Level.

```
[Button: SaveVar]
   Key    : Alt + S
   Action:  Save Variable : SmpVar.pvf : EmpNameVar, EmpIDVar
```

The action SAVE VARIABLE will persist the values of the variables EmpNameVar and EmpIDVar in the file SmpVar.pvf

LOAD VARIABLE

The action LOAD VARIABLE is used to reload the report scope variables from the specified file.

Syntax

```
      LOAD VARIABLE : <FileName> [:<Variable List>]
```

Where,

<FileName> is the name of file in which the report scope variables are persisted. The extension .PVF will be taken by default if the the file extension is not specified.

<Variable List> is the list of comma separated variables that need to be loaded from the file. It is optional to specify the variable list.In case it is not specified all variables saved in the file will be loaded.

Example:

In the previous example, we have persisted values of the Report Scope Variables **EmpNameVar** and **EmpIDVar** in the file **SmpVar.pvf**. Now let us see how to re-load these report scope variables from the file.

```
[Button: LoadVar]
   Key     : Alt + L
   Action  : LOAD VARIABLE : SmpVar.pvf : EmpNameVar, EmpIDVar
```

The action **LOAD VARIABLE** will load the report scope variables **EmpNameVar** and **EmpIDVar** from the file **SmpVar.pvf**.

Member Variable Specification or Dotted Notation Specification is not allowed for specifying Variable list for both the actions SAVE VARIABLE and LOAD VARIABLE. It has to be a variable name identifier at the current report scope.

Variable Copy

The content of variable can now be entirely copied from one instance to another instance.

COPY VARIABLE

The action COPY VARIABLE is used to copy the content from one variable (Source) to another variable (Destination). This action is supported for all types of variables (Simple / Compound / List Variables).

Syntax

```
      COPY VARIABLE: <Destination Variable> :< Source Variable>
```

Where,

<Destination Variable> is the name of Simple/Compound/List Variable

<Source Variable> is the name of Simple/Compound/List Variable from which the content has to be copied

Example: Copying from Simple Variable to Simple Variable

```
[Function: SimpleVar Copy Function]

   VARIABLE : SimpleVar1 : String : "Employee1"
   VARIABLE : SimpleVar2 : String :

   10 : COPY VARIABLE : SimpleVar2 : SimpleVar1
   20 : LOG : "Source"+##SimpleVar1
   30 : LOG : "Destination"+##SimpleVar2
```

In this example, the variables **SimpleVar1** and **SimpleVar2** are declared at the Function level. After execution of the action **COPY VARIABLE** the content of the variable copied from **SimpleVar1** to **SimpleVar2**.

Copying from Compound Variable to Compound Variable

Let us suppose, the below compound variables are defined:-

```
[Variable: Employee1]
   Variable : EmpName    : String : "Praveen"
   Variable : Designation: String : "Manager"

[Variable: Employee2]
   Variable : EmpName    : String
   Variable : Designation: String
```

In the below function, we are copying the contents from the Compound Variable Employee1 to Employee2

```
[Function: Compound Var Copy Function]

   VARIABLE : Employee1
   VARIABLE : Employee1
   10 : COPY VARIABLE : Employee2 : Employee1
   20 : LOG : “Source”+## Employee1.EmpName
   30 : LOG : “Source”+## Employee1. Designation
   40 : LOG : “Destination”+## Employee2.EmpName
   50 : LOG : “Destination”+## Employee2. Designation
```

The content will be copied from a member variable of Compound Variable (Source) to another member variable of compound variable (Destination) based on the member variable names since more than one member variable may have the same data type.

Copying from List Variable to List Variable

Let us suppose, the below compound variables are defined:-

```
[Variable: Employee1]

   Variable : EmpName   : String
   Variable : Designation: String

[Variable: Employee2]

   Variable : EmpName   : String
   Variable : Designation: String
```

In the below function, the compound variables **Employee1** and **Employee2** are declared as **List Variable**. We are copying all the elements from the compound list variable Employee1 to the compound list variable **Employee2**.

```
[Function: ListVar Copy Function]

   LIST VARIABLE :Employee1, Employee2

   10  : LIST FILL : Employee1 : Employees : $Name : $Name
```

```
20  : LIST FILL : Employee1 : Employees : $Name : $Designation + :
      Designation
30  : COPY VARIABLE : Employee2 : Employee1
40  : LOG : "Source Variable - Employee"
50  : FOR IN : KEY VAR : Employee1
60  :     LOG : $$LISTVALUE:Employee1:  ##KEYVAR:EmpName
70  :     LOG : $$LISTVALUE:Employee1:  ##KEYVAR:Designation
80  : END FOR
90  : LOG : "Destination Variable - Employee"
100 : FOR IN : KEY VAR : Employee2
110 :     LOG : $$LISTVALUE:Employee2:  ##KEYVAR:EmpName
120 :     LOG : $$LISTVALUE:Employee2:  ##KEYVAR:Designation
130 : END FOR
```

Scope specification in Variable Dotted Syntax

The Dotted Notation Syntax for Variables (##) is now enhanced to allow specification of scope / relative scope etc.

Syntax

```
        ..    (DOUBLE DOT) denotes owner scope
        …     (TRIPPLE DOT) denotes owner's owner scope and so on
        ().    denotes a system scope
```

Where,

<Definition Type> is the name of the definition such as Report, Function etc. in the current execution chain.

<Definition Name Expression> can be an expression which evaluates to a Definition Name.

The Definition Name Expression is optional.

(**<Definition Type>, <Definition Name Expression>**). Can be used to specify an absolute scope specification. The element (<Definition Type>, <Definition Name Expression>) has to be in the current execution chain else one will not be able to refer the same.

Example:

Let us suppose the Variable **TSPLSMPScopeVar** is declared at System Scope.

```
[Variable: TSPLSMPScopeVar]
   Type : String

[System: Variable]
   TSPLSMPScopeVar: "System Scope"
```

The below function **TSPLSMP ScopeSpec** is called from a Menu. We have declared the variable **TSPLSMPScopeVar** in the function scope also.

```
[Function: TSPLSMP ScopeSpec]
   VARIABLE    : TSPLSMPScopeVar
   01 : SET    : TSPLSMPScopeVar  : “Function Level”
   02 : Display: TSPLSMP ScopeSpec
```

The below report is displayed from the function **TSPLSMP ScopeSpec**. We have declared the variable **TSPLSMPScopeVar** in the Report Level also.

```
[Report: TSPLSMP ScopeSpec]
   Form    : TSPLSMP ScopeSpec
   Variable: TSPLSMPScopeVar
   Set     : TSPLSMPScopeVar : “Report Level”
```

Below are the field definitions of the report **TSPLSMP ScopeSpec**. Let us see the variable values at the field level by specifying the scope in Variable Dotted Syntax

```
[Field: TSPLSMP ScopeSpecCurrent]
   Use   : Name Field
   Set As: ##TSPLSMPScopeVar
;;Variable value in this field will be "Report Level" (Current Scope)

[Field: TSPLSMP ScopeSpecOwner]
   Use   : Name Field
   Set As: ##..TSPLSMPScopeVar
   Border: Thin Left Right
;;Variable value in this field will be "Function Level" (Owner's Scope)

[Field: TSPLSMP ScopeSpecSystem]
   Use    : Name Field
   Set As : ##().TSPLSMPScopeVar
   Border : Thin Left
;;Variable value in this field will be "System Level" (System Scope)

[Field: TSPLSMP ScopeSpecAbsolute]
   Use    : Name Field
   Set as : ##(Function,“TSPLSMP ScopeSpec”).TSPLSMPScopeVar
   Border : Thin Left
;;Variable Value in this field will be "Function Level"  (Absolute Specification)
```

Definition Name and Instance Name of Variable can be different now

A variable can be declared in a scope in two ways i.e. either specifying the name of the variable (in this case a separate variable definition is required) or specifying the name of the variable and a data type (in this case a separate variable definition is not required which is called as inline declaration).

In this chapter, we are going to discuss about the Report Scope variable decalartion syntax and examples. It is applicable for other scopes also.

Let us look in to the variable declaration syntax of Report Scope

```
[Report: <Report Name>]

    ;;This syntax expects a separate variable definition in the same name
   Variable      : <Variable Names>
```

OR

```
    ;;Inline declaration
   Variable      : <Variable Names> [:<Data Type>[:<Value>]]
```

OR

```
   List Variable : <Variable Names> [:<Data Type>[:<Value>]]
```

Example:

```
[Report:  SMP Report]
   Variable     : Emp Name
   Variable     : Emp Relation : String
   List Variable: Employee1
   List Variable: Employee2 : String :  "Prem"

[Variable: Emp Name]
   Type : String

[Variable: Employee1]
   Variable :  EmpName : String
   Variable :  EmpID   : String
```

Now the Data Type parameter can be pointing to a variable definition in which case it will allow you to have a variable which has an instance name and definition name different. This allows flexibility to create two instance of a compound structure in the same scope with different instance names without requiring duplicate the definition. This capability is available at all the scopes where variable declaration is allowed.

Existing Syntax

```
[Report: <Report Name>]

    Variable      : <Variable Names>
              OR
    Variable      : <Variable Names> [:<Data Type>[:<Value>]]
              OR
    List Variable : <Variable Names> [:<Data Type>[:<Value>]]
```

Enhanced Syntax

```
[Report: <Report Name>]

    Variable      : <Variable Names>
              OR
    Variable      : <Variable Names> [:<Data Type>[:<Value>]]
              OR
    List Variable : <Variable Names> [:<Data Type>[:<Value>]]
              OR
    Variable      : <Instance Names>:[<Variable Name>]
              OR
    List Variable : <Instance Names>:[<Variable Name>]
```

Where,

<Instance Names> is the list of Simple / Compound / List Variables separated by comma (instance variables).

<Variable Name> is the Simple or Compound variable name. A separate variable definition is required. It should not be an inline variable.

Example 1:

Given below is the definition of a Compound Variable “Employee”

```
[Variable: Employee]

   Variable : EmpName   : String

   Variable : Designation: String
```

Now we can create a variable instance using the definition of another variable. Let us understand with the help of below report definition.

```
[Report: Employee Report]
;;An instance is declared with the name as 'Prem' and definition name as 'EMPLOYEE'. The variable instance 'Prem'  will inherit the entire structure of variable definition  'EMPLOYEE'.
   Variable : Prem : Employee
;;An instance is declared with the name as 'Ramesh' and definition name as 'Employee'.
   Variable : Ramesh: Employee
;;Locally the instance "Ramesh" is modified to add a member variable.
   Local    : Variable : Ramesh : Add : Variable : EmpID : String
;; Two instances are declared with the names "Kamal" & "Vimal" and the definition name as "Employee"
   Variable : Kamal, Vimal: Employee
;; A List Variable instance is declared with the name "EmployeeList" and the definition name as "Employee"
   List Variable : EmployeeList : Employee
```

Example 2:

```
[Report: TSPL SMP Variable Instance]
   Variable : Employee  : String  : "Suresh"
   Variable : New Employee  :  Employee
```

In this example, we are trying to declare a variable instance 'New Employee' which is of type of another variable 'Employee'. This will NOT work because the variable 'Employee' is declared as inline and an explicit Definition does not exist for the same.

Hence, inline variables can not be used to declare another variable instance.

Use Case – Multiple Email Configurations

Scenario

ABC Company Ltd a manufacturing company is having Head Office in Bangalore and branch offices in Delhi, Mumbai, Kolkata and Chennai. The company is using Tally.ERP 9 in all the locations.

The Head Office and Branch Offices are using the e-mail capability of Tally extensively to send remainder letters/outstanding statements to the customers.

The System Administrator at the Head office will be facilitating the Brach office staff for email configurations in Tally. The company is using its own mail server and also using another mail server "SIFY". If there is a change in mail server, the system admin needs to communicate the information to branch staff and they will be updating the email configurations in Tally.ERP 9.

Now the company wants to set the email configurations centrally for all the branches so that branch staff need not struggle for email configurations particularly when there is change in mail server. This solution provides the facility of saving multiple configurations in multiple file names and later loading it from the file based on user selection.

Requirement Statement

At present in Tally.ERP 9, the users need to set the email configurations locally and update required details.

Now the configurations can be created centrally and shared among the locations. So that the user need not set the email configuration every time. They have to simply load the configuration from the file. This can be achieved using the newly introduced actions SAVE VARIABLE & LOAD VARIABLE

Functional Demo

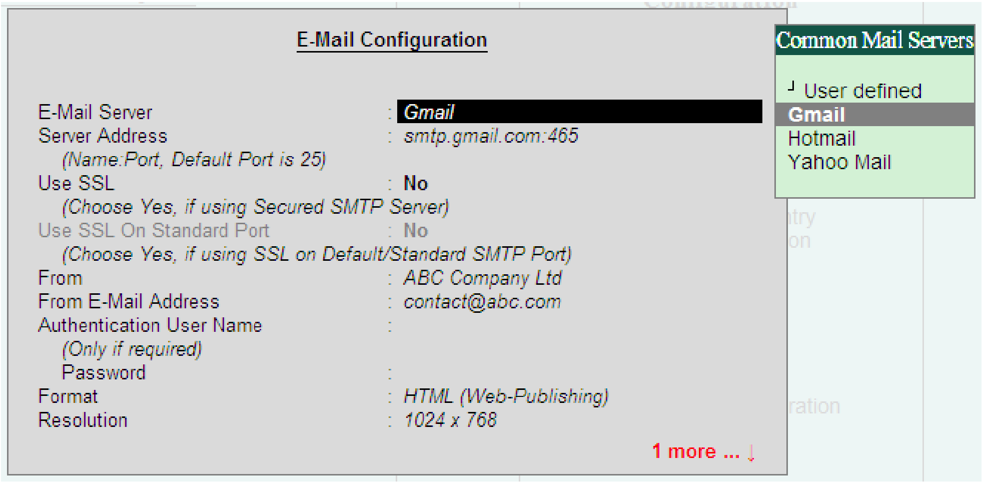
Before looking into the design logic, we will have a functional demo.

Let us suppose ABC Company Ltd is using its own mail server and another mail server Sify in Head Office and its branch offices.

Saving Email Configurations

Let us suppose the System Administrator in Head Office wants to save the required email configurations in Tally.ERP 9 for HO and Branches

Gateway of Tally –> F12 (Configure) –> E-Mailing The email configuration screen will appear as shown below:

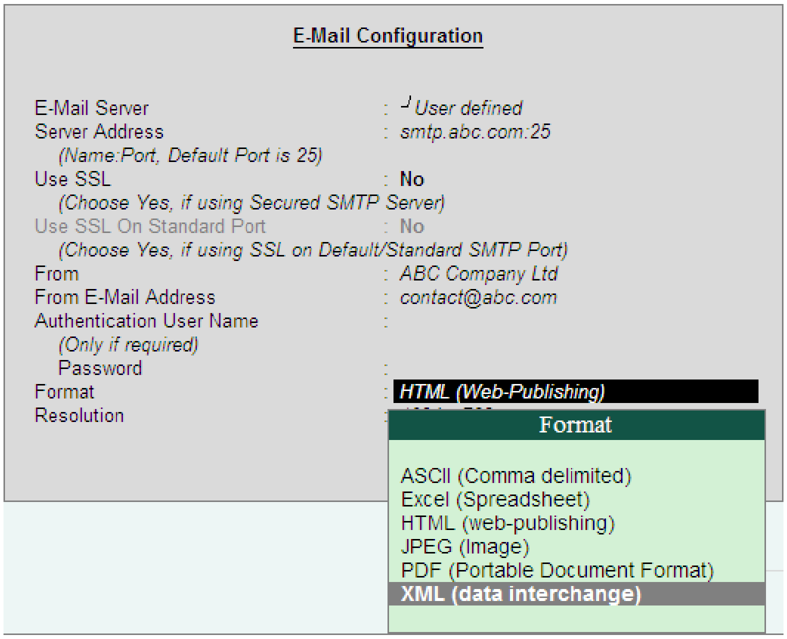


E-Mail Configuration

```
E-Mail Server                      : Gmail
Server Address                     : smtp.gmail.com:465
   (Name:Port, Default Port is 25)
Use SSL                            : No
   (Choose Yes, if using Secured SMTP Server)
Use SSL On Standard Port           : No
   (Choose Yes, if using SSL on Default/Standard SMTP Port)
From                               : ABC Company Ltd
From E-Mail Address                : contact@abc.com
Authentication User Name           :
   (Only if required)
   Password                        :
Format                             : HTML (Web-Publishing)
Resolution                         : 1024 x 768
                                                   1 more ... ↓
```

Figure 1.12 Email Configuration Screen

The System Admin needs to save the Configurations for the mail servers abc and Sify. Hence, he has to specify the Email server as "User Defined" and enter the required configuration settings as shown in the below screen shot:



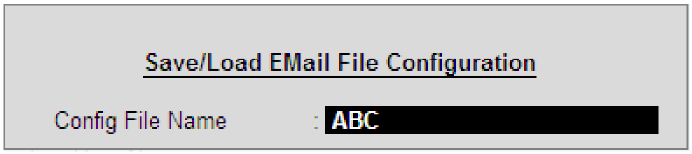
E-Mail Configuration

E-Mail Server : ┘User defined
Server Address : smtp.abc.com:25
(Name:Port, Default Port is 25)
Use SSL : No
(Choose Yes, if using Secured SMTP Server)
Use SSL On Standard Port : No
(Choose Yes, if using SSL on Default/Standard SMTP Port)
From : ABC Company Ltd
From E-Mail Address : contact@abc.com
Authentication User Name :
(Only if required)
Password :
Format : HTML (Web-Publishing)
Resolution :

Format
ASCII (Comma delimited)
Excel (Spreadsheet)
HTML (web-publishing)
JPEG (Image)
PDF (Portable Document Format)
XML (data interchange)

Figure 1.13 User Defined Configuration

Now the System Admin has to press **Alt+S** or click on the Button **Save Config**. The below screen will appear and he has to enter the configuration file name:



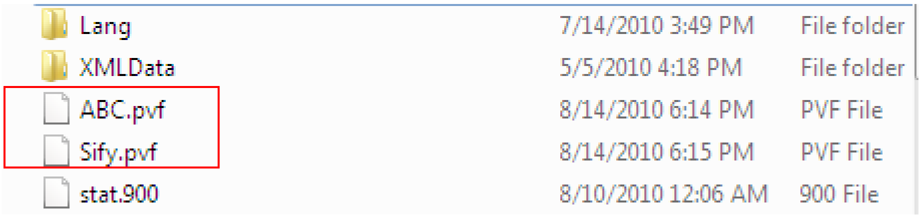
Save/Load EMail File Configuration

Config File Name : ABC

Figure 1.14 Save Configuration Screen

Once the System Admin accepts this screen, the configuration details will be saved in the file “abc.pvf”. Similarly he has to create the Configuration for the mail server “Sify”.

The files will be created in the Tally.ERP 9 application folder as shown in the below screen shot:



Lang	7/14/2010 3:49 PM	File folder
XMLData	5/5/2010 4:18 PM	File folder
ABC.pvf	8/14/2010 6:14 PM	PVF File
Sify.pvf	8/14/2010 6:15 PM	PVF File
stat.900	8/10/2010 12:06 AM	900 File

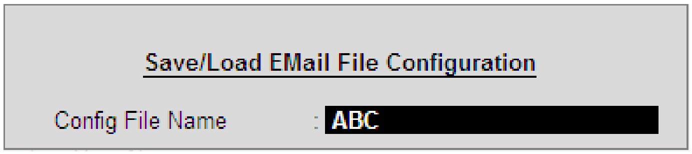
Figure 1.15 Files in Application folder

The admin can share these two files to the staff in HO and Branch Offices and they should place the file in the respective Tally.ERP 9Application folders.

Loading Configurations

Gateway of Tally –> F12(Configure) –> E-Mailing The Email configuration screen will be displayed with the previously set configurations.

Now the user at the HO/Branch wants to load the configurations for the email server “abc”. He has to press **Alt+L** or Click on the Button “Load Config” and enter the file name as shown in the below screen shot:



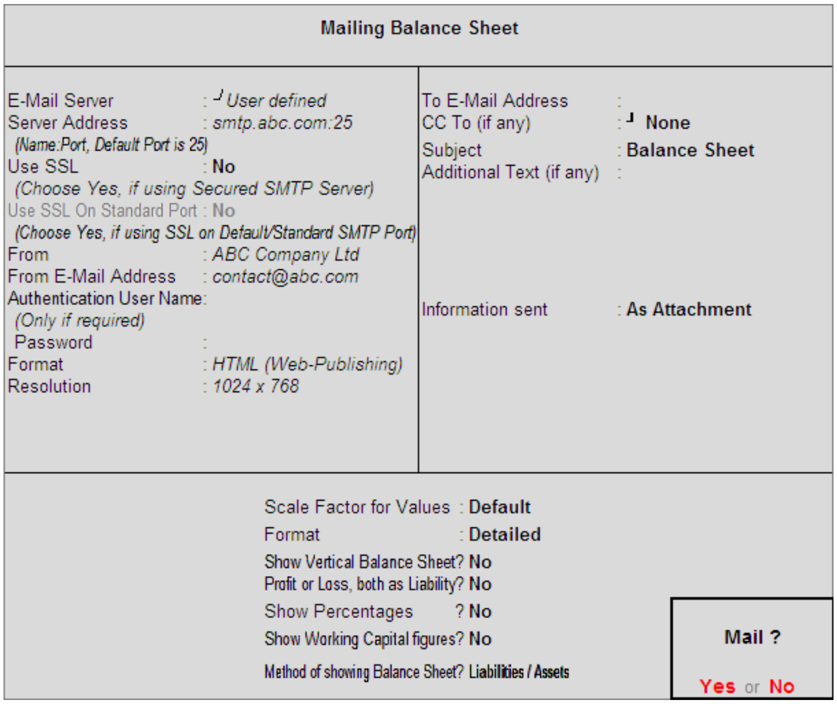
Save/Load EMail File Configuration

Config File Name : ABC

Figure 1.16 Load Configuration Screen

Accept the screen then the Email Configuration Report will display the configuration details loaded from the file “abc”. Accept the email configuration screen and the settings will be applicable to all the reports.

Suppose, the User now wants to mail the report Balance Sheet. He selects Balance sheet and press **Alt+M,** the below configuration report will appear:



Mailing Balance Sheet

E-Mail Server : User defined
Server Address : smtp.abc.com:25
(Name:Port, Default Port is 25)
Use SSL : No
(Choose Yes, if using Secured SMTP Server)
Use SSL On Standard Port : No
(Choose Yes, if using SSL on Default/Standard SMTP Port)
From : ABC Company Ltd
From E-Mail Address : contact@abc.com
Authentication User Name:
(Only if required)
Password :
Format : HTML (Web-Publishing)
Resolution : 1024 x 768

To E-Mail Address :
CC To (if any) : None
Subject : Balance Sheet
Additional Text (if any) :

Information sent : As Attachment

Scale Factor for Values : Default
Format : Detailed
Show Vertical Balance Sheet? No
Profit or Loss, both as Liability? No
Show Percentages ? No
Show Working Capital figures? No
Method of showing Balance Sheet? Liabilities / Assets

Figure 1.17 Email Configuration Screen

Note that the configuration details are changed as per the selected configuration.

Now the user wants to change the email server as “Sify”.

Gateway of Tally –> F12(Configure) –> E-Mailing –> Press Alt+L –> Enter the file name as “Sify” and press enter.
The email configuration screen will have new configurations loaded from the file “Sify”.

Similarly we can save/load multiple configurations.

Solution Development

The steps followed to achieve Saving Multiple Email Configuration are:-

1. **Declaring variables at Report Level**

The variables SVMailServerName, SVMailServer, SVMailFormat, SVMailUseSsl etc. are declared at the Report Level. All these variables are having the attribute Persistent:Yes set at the Definition level.

```
[#Report: EMail Configuration]

   Variable : SVMailServerName, SVMailServer, SVMailFormat, SVMailUseSsl

   Variable : SVMailUseSSLOnStdPort, SVMailAuthUserName, SVExportFormat
```

2. **Saving Configuration**

A Button is added to the Form and the action will call a User Defined Function.

In User Defined Function, we are executing a report to accept a File Name from the user. We are persisting all the report scope variables in the specified file through the Action SAVE VARIABLE.

3. **Loading Configurations**

A Button is added to the Form and on click of the same, the action will call a User Defined Function.

In the User Defined Function, we are executing a report to accept the File name from the user. We are reloading the report scope variables from the file through the Action LOAD VARIABLE. Please refer to the below code snippet for Save and Load configurations.

```
[Function: TSPL Smp SaveLoadVar]

   Parameter: IsSaveVar         : Logical: Yes

   Variable : ConfigNamewithExt: String: Yes


   00: EXECUTE : TSPL Smp SaveLoadConfig
;; Correcting the file name entered with or without extension by user
   06:    IF   : ##SaveLoadConfigName CONTAINS ".Pvf"

   10:      SET: ConfigNamewithExt: ##SaveLoadConfigName

   20:    ELSE :

   30:      SET: ConfigNamewithExt: ##SaveLoadConfigName + ".pvf"

   40:    ENDIF
```

```
;; Saving or Loading the variables based on parameter value
   50:    IF: NOT $$IsEmpty:##SaveLoadConfigName
   60:        IF: ##IsSaveVar
   70:            SAVE VARIABLE: ##ConfigNamewithExt
   80:        ELSE:
   90:            LOAD VARIABLE: ##ConfigNamewithExt
   100:       ENDIF
   110:  ENDIF
```

The corresponding field values need to be reflect the values of the variables loaded from the file. This is handled by using the following code

```
Local: Field: DSPMailServer : Set as : +
       If #DSPMailServerName Contains $$SysName:UserDefined +
       Then ##SVMailServer Else  +
           If #DSPMailServerName NOT Contains $$SysName:UserDefined +
             Then $$GetMailServerAddr:#DSPMailServerName +
           Else ##SVMailServer

Local: Field: DSPMailServerName    : Set As : ##SVMailServerName
Local: Field: DSPMailFormat        : Set As : ##SVMailFormat
Local: Field: DSPMailUseSsl        : Set As : ##SVMailUseSsl
Local:Field:DSPMailUseSSLOnStdPort : Set As : ##SVMailUseSSLOnStdPort
Local: Field: DSPMailAuthUserName  : Set As : ##SVMailAuthUserName
Local: Field: DSPFinalExportFormat : Set As : ##SVExportFormat
```

Also if the field values are changed, the Report level variables need to be modified with those values . This is handled using the following code

```
Local: Field : DSP MailServerName: Modifies : SVMailServerName  : Yes
Local: Field : DSP MailServer: Modifies     : SVMailServer : Yes
Local: Field : DSP MailFormat: Modifies     : SVMailFormat : Yes
Local: Field : DSP MailUseSsl: Modifies     : SVMailUseSsl : Yes
Local: Field : DSP MailUseSSLOnStdPort      : Modifies :   +
               SVMailUseSSLOnStdPort: Yes
Local: Field : DSP MailAuthUserName         : Modifies : +
               SVMailAuthUserName: Yes
```

```
   Local: Field : DSP FinalExportFormat        : Modifies: +
                  SVExportFormat: Yes
```

On Accept of the Form EMail Configuration, we are calling a User Defined Function to set the System Variable values. So that, the changed configuration details will be available for all the reports. Please refer to the below Code Snippet

```
[Function: TSPL Smp Update System Variables]

   10: SET: ().SVMailServerName      : ##SVMailServerName
   20: SET: ().SVMailServer          : ##SVMailServer
   30: SET: ().SVMailFormat          : ##SVMailFormat
   40: SET: ().SVMailUseSsl          : ##SVMailUseSsl
   50: SET: ().SVMailUseSSLOnStdPort : ##SVMailUseSSLOnStdPort
   60: SET: ().SVMailAuthUserName    : ##SVMailAuthUserName
   70: SET: ().SVExportFormat        : ##SVExportFormat
```

5.3 Event Framework Enhancements

This is a path breaking enhancement in Tally which will enable scheduled execution of any Action. This has been supported with the introduction of a System Event called Timer. We can have a set of timer events of specified durations and trigger an Action on the same. For eg: if we require Synchronization to be triggered every one hour we can define a Timer event which triggers the action Sync. Actions for Starting and Stopping the timer have been provided.

Timer Event

As we are already aware that Events like **System Start**, **System End**, **Load Company**, **Close Company**, **On Form Accept** introduced earlier as a part of the Event Framework etc. require user intervention. Automated events which can be used to take timely backups, display automated messages, etc. were not possible earlier.

With the breakthrough introduction of **Timer Event**, performing Timer based automated events are possible now. System Event **Timer** is introduced to perform the required set of operations automatically at periodic intervals.

Syntax

```
      [System: Event]
            <Timer Name>: TIMER : <Condition> : <Action> : <Action Parameters>
```

Where,
<Timer Name> is user defined name for the timer event.
TIMER keyword indicates that it is a Time based event.
<ConditionExpr> should return a logical value.
<ActionKeyword> any one of the actions.
<Action Parameters> parameters of the actions specified.

We can have multiple Event Handlers with unique names which can invoke specific Actions at different intervals.

In order to specify the interval for the various Timers and to begin and end the Timers the associated Actions Introduced are Start Timer and Stop Timer

Start Timer

This Action starts the specified timer and this action accepts the Timer Name and Duration in seconds as action parameters.

Syntax

```
      START TIMER: <Timer Name>: Duration in seconds
```

Where,
<Timer Name> is user defined name for the timer event.

Stop Timer

Action stops the specified timer and this action accepts the Timer Name as its parameter.

Syntax

```
      STOP TIMER: <Timer Name>
```

Where,
<Timer Name> is user defined name for the timer event.

Following is an example scheduling automatic backups every hour:

Example:

```
[System: Event]
;; Setting up timer event to call a function
   Auto Backup     : TIMER: TRUE: CALL: Take Backup Function
;; Starting the Timer when Tally Application Starts
   Schedule Backup: System Start: TRUE: START TIMER: Auto Backup: 3600

;; Adding Keys to Company Info. Menu
[#Menu: Company Info.]
   Add: Keys: Stop Backup Timer

;; Declaring a Key to Stop the Timer
[Key: Stop Backup Timer]
   Key    : Alt + S
   Action : Stop Timer: Auto Backup
   Title  : “Stop Backup”
```

In the above example, following is done:

- **Auto Backup**, a **Timer Event** is declared under **System Event** to invoke the Function **Take Backup Function** at periodic intervals as specified within the Action **Start Timer**.
- **Schedule Backup**, a **System Start** event is declared under **System Event** to **Start** the above **Timer**, **Auto Backup** and execute the specified action every 3600 Seconds i.e., every hour.
- A corresponding Key to **Stop** the **Timer** is associated to Menu **Company Info**. which is defined to **Stop** the **Timer**. User can stop the timer if he chooses not to continue taking automatic backups any further.

Timer Events can be very useful in many cases like displaying Exception Reports, Negative Balances intimation, Inventory Status below Minimum or Reorder Level, Outstanding Reminders, Auto Sync at regular intervals and many more.

5.4 Action Enhancements

New actions are introduced in this release viz. Refresh Data, Copy File and Sleep.

Refresh Data

In Tally, whenever any report is being viewed, it contains the most recent updates till the last entry. If any report is left open and subsequently viewed later, possibly few more entries would have gone in the system entered by various other users on the Network. Hence, the report which is currently being viewed is older. To view the updated report, user has to exit the report and once again enter the Report. To solve this problem, a new action Refresh Data is introduced which refreshes the data in memory automatically, as and when required.

Syntax

```
REFRESH DATA
```

Refresh Data can be used along with **Timer Event** and every few seconds the Report can be refreshed automatically to display the u information.

Example:

```
[System: Event]
   Refresh Timer: TIMER: TRUE: Refresh Data

[#Form: Balance Sheet]
   Add: Keys: Enable Refresh

[Key: Enable Refresh]
   Key    : Alt + R
   Action: Start Timer: Refresh Timer: 300
```

In the above example,
Refresh Timer, a **Timer Event** is declared under **System Event** to invoke the Action **Refresh Data** at periodic intervals.
A key **Enable Refresh** is added in the Balance Sheet Report which will be used to Start the Timer **Refresh Timer** every 5 min.

The Action Refresh Data is a Company -Report Specific Action. It will always require a Report in memory to Refresh the Data.

SLEEP

Action SLEEP is introduced to specify time delays during execution of the code. For few seconds, the system will be dormant or in suspended mode.

Syntax

```
SLEEP : <Duration in Seconds>
```

Sleep Action halts the functioning of the Application for few seconds as specified in the **Duration**.

Example:

```
[#Menu: Gateway of Tally]
  Add: Item: Trial Balance after 10 secs: CALL: TBafterSleep

[Function: TBafterSleep]
   00: SLEEP  : 10
   10: DISPLAY: Trial Balance
```

In the above example, system will halt for 10 seconds and display Trial Balance subsequently.

Copy File

We have introduced a new Action Copy File. This allows us to perform Copy from

- One location to another within the same System
- Uploading of Files from a given Path to a FTP Site
- Downloading of File from FTP Site to specified location/folder

Syntax

```
Copy File : <Destination File Path>:<Source File Path>
```

Where,
<Destination File Path> Can be an expression evaluating to a valid local/FTP path
<Source File Path> Can be an expression evaluating to a valid local/FTP path

Example:

```
CopyFile:##MyDstFilePath: ##MySourceFilePath
```

If any of the File path is an FTP path then the same can be constructed using the functions like $$MakeFTPName. It accepts the various parameters like servername,username,password etc. The code snippet below sets the value of the variable MyDstFilePath using the function MakeFTPName

```
SET : MyDstFilePath : $$MakeFTPName :##SVFTPServer : ##SVFTPUser: +
        ##SVFTPPassword:@SCFilePathName
```

The function $$MakeFTPName uses the various parameters which are System Variables captured from the default configuration reports.

5.5 TDL Enhancements for Remoting

There have been various enhancements at the TDL level to enable Remote Edit Capability in the product. The enhancements are listed as below

- **Fetch Object Attribute Changes**

The attribute Fetch Object is supported at Report, Form, Field and Function level as well. The Object Name specification in the syntax allows expressions now. It is also possible to specify multiple Object Names separated by the Fetch Separator Character. A new function $$FetchSeperator is introduced to return this character.

- **Fetch Values Atribute Introduced**

The evaluation of External Methods of an Object requires Object Context to be available at the Client End. A new Attribute Fetch Values is provided at the Report level to specify the list of External Methods.

- **Multi Objects Attribute Introduced**

Whenever multiple Objects of the same collection is getting modified at the Client End, a new attribute called MultiObjects is introduced at the Report Level to enable the same.

- **Modifies Attribute Changes**

The Modifies attribute of the field is changed to accept a third parameter(optional) which is an expression. This allows the variable to be modified with the value of the expression rather than the field value.

- **Collection Attribute – Parm Var**

As we already know the Collection Artifact evaluates the various attributes either during initialization or at the time of gathering the collection. It may require various inputs from the Requestor context for the same.

The direct reference of values/expressions from the report elements and objects in the collection at various points creates various issues like code complexity, performance lapses and non availability of these values on Server in Remote Environment.

In order to overcome the above we have introduced a new Collection attribute Parm Var.

Parm Var in collection is a context free structure available within the collection. The requestors Object context is available for the evaluation of its value. This is evaluated only once in the context of the caller/requestor. This happens at collection initialization and the expression provided to it is evaluated and stored as a variable which can be referred within any of the attributes of the collection at anytime and is made available at the Server end as well.

Lets understand each of these in detail.

Fetch Object Attribute Changes

When multiple methods of a Single/Multiple Objects of the same type are required, then that Object can be fetched at Report, Form, Field and Function

Report Level

Fetch Object attribute has been enhanced at report level to take an expression instead of a variable name that evaluates to name of an object.

The existing syntax of the Fetch Object attribute at report level is as follows:

Syntax: Prior to 2.0

```
      Fetch Object : <Object Type> : <Variable Identifier > : <List of methods>
```

Example:

```
   Fetch Object: Ledger: LedgerName: Name, Parent, ClosingBalance
```

Syntax: 2.0 Onwards

The enhanced Syntax:

```
      Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
   Fetch Object: Ledger: ##LedgerName: Name, Parent, ClosingBalance
```

In the example above since the Object name is an expression we need to prefix the variable name with ##.

Form Level

The attribute Fetch object is now introduced at Form Level. In scenarios where multiple forms are available at a report and for each form we require to fetch methods pertaining to different objects

Syntax

```
      Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[!Form: AccoutingViewVoucher]
   Switch: AccVoucherView : NormalAccoutingViewVoucher:+
           NOT$$IsAttendance:##SVVoucherType
   Switch: AccVoucherView: AttdAccoutingViewVoucher : +
           $$IsAttendance:##SVVoucherType

[!Form: AttdAccoutingViewVoucher]
   Fetch Object: AttendanceType: @@AttdEntryList :  +
                 AttendanceProductionType, AttendancePeriod, BaseUnits

[!Form: NormalAccoutingViewVoucher]
   FetchObject : Ledger: @@AllLedEntryList : Name, Parent, ReserveName
```

Field Level

There may be scenarios where we may need to Fetch Object values dynamically based on current field values. For eg : The field may be associated with a Table of ledgers. Based on the ledger selected the corresponding methods of the Object require to be fetched. In that case this attribute will be useful.

Syntax

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[Field: LED VAT Class]
   Fetch Object: TaxClassification : $$Value : FirstAlias,RateofVAT, +
                 TaxType
```

Function Level

There may be scenarios where the method values need to be fetched based on the Object name passed as a parameter to the function. In such cases Fetch Object at the function level is required. If we have already fetched the object methods at the Report or field level the same will be propogated to the called function. In case it is not fetched earlier, the same can be fetched at the function level as well. This enables dynamic fetch of Objects

Syntax

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[Function: FillUsingTrackingObj]
   Parameter    : pTrackKey        :  String
   Fetch Object: Tracking Number :  ##pTrackKey: *.*
```

In case same set of methods for multiple objects need to be fetched then multiple Object Names need to be specified in the Fetch Object Syntax separated by the Fetch separator character .

- Function – $$FetchSeparator

This function returns C_FETCH_SEPARATOR character that is used for separating multiple object names in FETCH OBJECT attribute.There may be scenarios where the same set of methods needs to be fetched from multiple objects . In that case it is possible to specify multiple object names in the fetch object syntax separated by the character which is returned from the function $$FetchSeparator

Example:

```
Fetch Object: Ledger: "Debtor North" +$$FetchSeparator + "Debtor South":
               Name, Parent, ClosingBalance
```

Fetch Values

This is a report level attribute which allows computation of values for user defined(external methods) based on the current Object context available.

Syntax

```
      Fetch Values: <List of methods>
```

Example:

```
[Report: VAT Classification]
    Object      : Tax Classification
    Fetch Values: MasterID, CanDelete
```

Multi Objects

This is a Report level attribute which is required to be specified in case Multiple Objects of the same collection is being added/modified in a Report.Required specifically multi master creation or alteration

Syntax

```
      MultiObjects: <Edit Collection>
```

Where,

<Edit Collection> is the Collection name that you would make modifications to.

Example:

```
[Report: Multi Ledger]
   Multi Objects: Ledger Under MGroup
```

Modifies

This is a field level attribute enhanced further to take a third optional parameter. Prior to Tally.ERP9 Release 2.0 if a field has modifies, the field value will be set to variable. And based on this variable value we require some calculations or concatenation to be performed we required an invisible field for the same. With this enhancement we can modify the variable value at the same field itself using an expression i.e. the field and variable may have different values

Syntax: Prior 2.0

```
      Modifies:<Variable Name>: <Logical Value>
```

Where,
<Variable Name> is name of the variable
<Logical Value> an expression which evaluates to logical value

Example:

```
[Field: BatchesInGodown]
   Modifies: DSPGodownName : Yes
```

Syntax: 2.0 onwards

```
      Modifies:<Variable Name>: <Logical Value>:<expression>
```

Example:

```
[Field: BatchesInGodown]
   Modifies: DSPGodownName : Yes: ##DSPGodownName + " - Godown"
```

Say field value is 'Main location' Output of the above would be Main location - Godown

Collection Attribute – Parm Var

As we already know the Collection Artifact evaluates the various attributes either during initialization or at the time of gathering the collection. It may require various inputs from the Requestor context for the same. For eg the evaluation of Child of and Filter attribute happens at the time of gathering the collection. It requires certain values from Requestors context like $name. In filter attribute if $name of each object is to be compared with $name of the Requestors context then we have to refer it as $ReqObject:$name. The direct reference of values/expressions from the report elements and objects in the collection at various points creates a few issues as follows:

- Increases the code complexity as observed in the Filter example above
- The performance is impacted as there is are repeated references in case of Filters
- In a Remote Environment where the Requestor Context is not available within the collection at the Server side

In order to overcome the above we have introduced a new Collection attribute Parm Var.

We already have the capability of declaring inline variables at the collection level using the Attributes Source Var, Compute Var and Filter Var. These are the context free structures available within the collection for various evaluations. For storing values in these, the various object contexts available are Source Objects, Target Objects etc. We have introduced one more attribute called Parm Var in collection which is a context free structure available within the collection. The requestors Object context is available for the evaluation of its value. This is evaluated only once in the context of the caller/requestor. This happens at collection initialization and the expression provided to it is evaluated as stored as a variable which can be referred within any of the attributes of the collection at anytime and is made available at the Server end by passing it with the XML Request.

Syntax

The attribute ParmVar evaluates the value of the variable based on the requestor object's context.

```
      Parm Var : <Variable Name> : <Data Type> : <Formula>
```

Where,
<Variable Name> is the name of variable.
<Data Type> is the data type of the variable
<Formula> can be an expression which evaluates to value of the variable data type.

Example:

In the code given below the Line Groups and Ledgers is repeating on a Group collection. From within that line a part is exploded which displays the List of Ledgers belonging to that Group. The line List of Ledgers within this part Repeats on the collection Smp List of Ledgers

```
[Part: Groups and Ledgers]
   Lines  : Groups and Ledgers
   Repeat : Groups and Ledgers : List of Groups
   Scroll : Vertical

   [Line: Groups and Ledgers]
     Fields       : GAL Particulars
     Right Fields : GAL ClosBal
     Explode      : List of Ledgers : ##ExplodeFlag

[Part: List of Ledgers]
     Lines  : List of Ledgers
     Repeat : List of Ledgers : Smp List of Ledgers

[Collection: Smp List of Ledgers]
   Type     : Led ger
   Child Of : $Name
```

In the collection **Smp List of Ledgers** the **Child** of attribute is evaluated based on the method **$Name** which is available from the Group Object in context. The line **Groups and Ledgers (Requestor Object)** is associated with a Group Object.

In a Remote environment when such a Report is being displayed at the Clients end, the request for the collection gathering goes to the Server End. At the server end the Requestor Context is not available. So the evaluation of $Name will fail.

To overcome such a situation, we have introduced a new attribute called "Parm Var" which is a context free structure available within the collection, which evaluates the expression based on the Requestors Context, thereby available at the Server Side also.

The Collection is Redefined as per below using attribute ParmVar

```
[Collection: Smp List of Ledgers]

   Type     : Ledger
   Child Of : ##ParmLedName
   Parm Var : ParmLedName : String  : $Name
```

The value of variable **"ParmLedName"** is evaluated at the Client side based on method $name available from Group Object Context and sent to the Server. While gathering the objects at the server side, the attribute ChildOf is evaluated which uses the variable **ParmLedName** instead of $Name, available at the Server.

5.6 Default TDL Changes

In the release 2.0 many new features like Remote Edit, SMS support etc are introduced. The TDL language is also enriched with new capabilities to support these features. Using the new language capabilities the source code of Tally.ERP9 Release 2.0 is also enhanced. The changes are made in many definitions for e.g. the values of some of the attributes are changed, new attributes are added and formulas are rewritten.

Although we have tried to ensure maximum backward compatibility, still there may be cases where application developer may require to validate/rewrite the existing TDL codes to make them compatible with Tally.ERP9 Release 2.0.In the section below we are trying to summarize the changes in terms of listing the definitions. Although,our sincere efforts have been in the direction of providing a comprehensive listing of definitions, still you may come across a few cases where changes have been made. If any of these definitions are being used in your customizations please refer to the source code changes available to you with the latest Release of TDE.

Mandatory Fetch at the Collection Level

This release onwards Fetch is mandatory in every collection.All the methods which are required to be used in a Report are to be fetched at the Collection level.

Voucher Creation

Whenever a new Voucher is being created,it is important to take care of the following

- The variable name “SVViewName” has to be set to System Names
  - AcctgVchView – For all accounting vouchers
  - InvVchView – For all inventory vouchers except Stock Journal voucher
  - PaySlipVchView – For payroll vouchers
  - ConsVchView – For Stock Journal voucher
- The method name PersistedView has to be set to the value of the above variable “SVViewName”

Refer to the Example below as used inside the Function Block

```
ds : Set : SVViewName : $$SysName:AcctgVchView
10 : NEW OBJECT : Voucher
                  |
                  |
                  |
Aa : Set Value : PersistedView : ##SVViewName
30 : CREATE TARGET
```

Extract Collections List and Usage as Tables

Many existing collection definitions have been converted as Extract Collections. So if these collections are used in any of the user TDLs then the code needs to be rewritten for Tally.ERP9 Release 2.0. Many fields which were using the old collections as Tables have been modified to use the Extract Collections now. The Table Attribute has been changed for those fields.

Following Table shows the fields in which extract collections are used in the Table attribute:

Field Name	Table Name OLD	Extract Colletion / Table Name
EI AccAllocName	Inv SalesLedgersAlloc Inv Purch Ledgers Inv Sales Income Ledgers Inv Purch Expense Ledgers NonInv Purch Support Ledgers NonInv Sales Support Ledgers	Inv SalesLedgersAlloc Extract Inv Purch Ledgers VchExtract Inv Sales Income Ledgers Extract Inv Purch Expense Ledgers Extract NonInv Purch Support Ledgers - VchExtract NonInv Sales Support Ledgers - VchExtract
EI Consignee	Party Ledgers, Cash Ledgers Invoice Ledgers	Party Cash Ledgers Extract Invoice Ledgers Extract
EICommonLED	Inv Sales Ledgers Inv Purch Ledgers Inv Sales Income Ledgers Inv Purch Expense Ledgers	Inv Sales Ledgers Extract Inv Purch Ledgers Extract Inv Sales Income Ledgers Extract Inv Purch Expense Ledgers Extract

VCH VATClass	VCH VAT Sales ClassificationVCH VCH VAT Purc ClassificationVCH	VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract
VCH AccAllocVATClass	VCH VAT Sales ClassificationVCH VCH VAT Purc ClassificationVCH	VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract
VCH POS PartyContact	Party Ledgers	Party Ledgers Extract
VCHACC StockItem	Vch Stock Item	Vch Stock Item Extract
VCHJRNLStockItem	Vch Stock Item	Vch Stock Item Extract
ACGLLed	GainLoss Ledgers	GainLoss Ledgers Extract
ACLSFixedLed	Cash Class Ledgers	Cash Class Ledgers Extract
ACLSLed	Cash Ledgers Normal Ledgers Normal Ledgers, Cash Ledgers Non CENVAT Ledgers Non CENVAT Ledgers, Cash Ledgers,	Cash Ledgers VchExtract Normal Ledgers Extract Normal Cash Ledgers Extract Non CENVAT Ledgers Extract Non CENVAT Cash Ledgers Extract
EI AccDesc	Sales Support Ledgers Purchase Support Ledgers	Sales Support Ledgers VchExtract Purchase Support Ledgers VchExtract
VCH AccVATClass	SD Sales Classification Etc…	SD Sales Classification Extract Etc …
VCHIndentStockItem	Vch Stock Item	Vch Stock Item Extract
VCHBATCH Godown	Stockable Godown JOB Stockable Godown	Stockable Godown VchExtract JOB Stockable Godown VchExtract
VCHBATCH OrdrName	Active Batches	Active Batches VchExtract
VCHBATCH NrmlName	Active Batches	Active Batches VchExtract
VCHBATCH JrnlName	Active Batches	Active Batches VchExtract

EI VATClass	SD Sales Classification VCH VAT Sales ClassificationVCH VCH VAT Purc ClassificationVCH VAT Sales With Rate ClassificationVCH VAT Purc With Rate ClassificationVCH Addl VAT Sales With Rate ClassificationVCH Addl VAT Purc With Rate ClassificationVCH CessOn VAT Sales With Rate ClassificationVCH CessOn VAT Purc With Rate ClassificationVCH CST Sales With Rate Classification CST Purc With Rate Classification	SD Sales Classification Extract VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract VAT Sales With Rate ClassificationVCH Extract VAT Purc With Rate ClassificationVCH Extract Addl VAT Sales With Rate ClassificationVCH Extract Addl VAT Purc With Rate ClassificationVCH Extract CessOn VAT Sales With Rate ClassificationVCH Extract CessOn VAT Purc With Rate ClassificationVCH Extract CST Sales With Rate Classification Extract CST Purc With Rate Classification Extract
VCHBATCH GRNName	Active Batches	Active Batches VchExtract
POS BatchName	Active Batches	Active Batches VchExtract
VCHBATCH DealerGodown	Stockable DealerGodown	Stockable DealerGodown VchExtract
VCHBATCH ExciseMfgrGodown	Stockable ExciseMfgrGodown	Stockable ExciseMfgrGodown VchExtract
VCHBILL TDSLedger	TDS Ledger Table	TDS Ledger Table VchExtract
VCHBILL STaxLedger	Service Tax Ledger Table	Service Tax Ledger Table VchExtract
VCHCSTCAT Name	Voucher Cost Category	Voucher Cost Category VchExtract
VCHCST Name	Cost Centre All Cost Centre	Cost Centre VchExtract All Cost Centre VchExtract
STKVCH Ledger	Party Ledgers, Cash Ledgers	Party Cash Ledgers Extract
PF CashBank Ledger	Cash Ledgers	Cash Ledgers VchExtract
VCH AttdEmpName	Payroll DeactvationEmployees	Payroll DeactvationEmployeesExtract
VCH AttdType	List of Attendance Types	List of AttdTypesExtract
VCH AutoAttdType	List of Attendance Types	List of AttdTypesExtract
VCH EmpName	Payroll Cost Centres Manual Vch Employees Under Category	Payroll CostCentres AsVCHExtract Manual AsVchEmployees Under CategoryExtract
PAYROLLFixedLed	Payroll Liab Ledgers	Payroll Liability LedgersExtract
Payroll VCH EmpCatParticulars	List of CostCategories	List of CostCategories Extract
Payroll VCH EmpParticulars	Payroll Cost Centres Manual Vch Employees Under Category	Payroll CostCentres VCHExtract Manual Vch Employees Under CategoryExtract

PayrollVCHPayhead Name	Vch Pay Heads	Vch Pay Heads Extract
Payroll FunctionAutoCategoryName	Payroll Vch Categories	Payroll Vch Categories Extract
Payroll FunctionAutoCstTables	Payroll Cost Centres AutoVch Employees Under Category	AutoVch PyrlCostCentres VCHExtract AutoVch Employees Under CategoryExtract
Payroll FunctionAutoPayheadName	Payroll Ledgers	AutoVch PayrollLedgersExtract
TDSAutoLedger	Normal Ledgers	Normal Ledgers Extract
TDSFilter Bank	Cash Class Ledgers	Cash Class Ledgers Extract
EI TrackOrder	InvSalesOrders InvPurcOrders	InvSalesOrders, Not Applicable, EndOfList, NewNumber InvPurcOrders, Not Applicable, EndOfList, NewNumber
EI SalesOrder	InvSalesOrders InvPurcOrders	InvSalesOrders, Not Applicable, EndOfList, NewNumber InvPurcOrders, Not Applicable, EndOfList, NewNumber
SRVTPartyName	Service Party Ledgers	Service Party Ledgers Extract
SRVTPartyBillName	Pending Party Bills	Pending Party Bills Extract

Modified Definition List and corresponding Changes

Changes in Set As

Definition Type	Definition Name
Part	VCH Excise SubCat Tax Rate
	Trader PurcTypeofDuty
	VCH Excise SubCat Tax Rate
Filed	VCH Excise SubCat Tax Rate
	Trader PurcTypeofDuty
	TDS TaxPartyLedger
	VCH TaxPymtDetails
	Trader PLARG23SlNo
	Trader SupplierRG23No
	Trader OriginalRefNo
	Trader MfgrImprName
	Trader DN SupplierInvNo
	Trader DN SupplierInvDate

	Trader DN NatureofPurc
	Trader DN QtyPurchased
	Trader DN QtyReturn
	Trader DN AssessableValue
	Trader CN SupplierInvNo
	Trader CN SalesInvDate
	Trader CN SalesInvNo
	Trader CN QtySold
	Trader CN QtyReturn
	Trader CN SplAEDOfCVDNotPassOn
	DealerInv AmtofDuty
	DealerInv DutyPerUnit

Options Added–In Alter Mode

In the definition **[Form: Voucher]** the option for alter mode is added and it is used to list all the fetches.

Definition Type	Definition Name
Form	Voucher

Fetch Object Added

Definition Type	Definition Name
Field	VCH StockItem

Compute Var and Fetch Attribute Added

Definition Type	Definition Name
Collection	Vouchers of FBT Category Calc
	Memo Vouchers of FBT Category Calc
	FBTCategoryCalc
	Vouchers of Regular FBT Category Calc
	Vouchers of Recovered FBT Category Calc
	VCHInTNo
	VCHInTNoG
	VCHInTNoB
	VCHInTNoBG
	VCHOutTNo
	VCH OutTNoG

	VCH OutTNoB
	VCH OutTNoBG
	TaxBill Details
	PayFunctionCaterotyCollection
	PayFunctionEmployeeCollection
	AllStatLedgersSlabSummary
	AllPFStatLedgers
	Admin AutoFil JrnlEmployees
	AutoFil PF Ledgers
	Admin AutoFil Employees
	AdminAutoPayableColl
	AdminAutoPayableColl PayrollSrc
	AdminAutoPayableCollJrnl
	AdminAutoPayableColl JrnlSrc
	PFESI EmployeeFilter Summary
	PFESI EmployeeFilter Vouchers
	Excise RG23DNoColl
	Trader ListOfPurcCleared
	Trader ListOfPurcNonCleared
	Trader ListOfMultiPurcCleared
	Trader ListOfMultiPurcNonCleared
	ExciseDealer Inventory Entries
	ExciseDealerInvoice InventoryEntries
	TDS DeductSameVoucher
	TDS TaxObjPartyBills
	TDS ITIgnoreNOP
	TDSDuty LedTable
	TaxObj AgstTableDebitNote
	TaxObj AgstTable
	SRCTaxObj AgstTable
	TDS CommonPartyLedger
	TaxObj AdvAgstTable
	TaxObj DedTable
	Pending TCS Bills
	PndgTaxBillsTillCurVchDate
	TaxBillColl

	TCS Vouchers of Party
	Pending Tax Bills
	BankColl
	InvSalesOrders
	ExciseInvSalesOrders
	InvPurcOrders
	InvOutTrackNumbers
	InvInTrackNumbers
	Pending Sales Orders
	VCHSo
	Pending Purc Orders
	VCHPo
	VCH OutTNo Src
	InPending Tracking Numbers
	OutPending Tracking Numbers
	Pending Bills
	STX SalePending TaxObj
	Pending STaxParty Bills
	STX CrDrNotePending TaxObj
	STX CategorywisePending TaxObj
	STX RcptPending TaxObj
	STXSource
	STX SalePending TaxObj
	STX JV SalePending TaxObj
	STXTaxObjOutput GAR7PymtAlloc
	STXTaxObjInput GAR7PymtAlloc

Few Attributes Added and DebugExec Action Used

Definition Type	**Definition Name**
Function	ESIDeductionPayFunction
	ESIEligibilityPayFunction
	ESIContributionPayFunction
	PFNormalPayFunction
	PTNormalPayFunction
	PTMonthlyPayFunction
	FirstEPF12PayHeadAbsVal Function

	FirstEPF833PayHeadAbsVal Function
	PFNormalVchPayFunction
	PTMonthlyVCHPayFunction
	PTNormalVchPayFunction
	ESIDeductionVchPayFunction
	ESIEligibilityVCHPayFunction
	ESIEligibilityOnCurrentEarnVch
	ESIEligibilityOnSpecifiedFrmlVch
	ESIContributionVchPayFunction
	FirstEPF12PayHeadAbsVchVal Function
	FirstEPF833PayHeadAbsVchVal Function
	IsExciseRG23DNoExistsFunc
	IsSpecialAEDOfCVDExistsInStkItem
	SetTDSPymtDetails
	VoucherFill
	OrderObjExists
	TrackingObjExists
	FillUsingVoucher
	CopyBatchAllocationsValues
	STCatCheck
	STCatRate
	STCatCessRate
	STCatSecondaryCessRate
	STCatAbatmentNo
	STCatAbatPer
	STCatCheck

6. Enhancements in Release 1.8

6.1 Invoking Actions on Event Occurrence with System & Printing Events Introduced

In any language, event handling is one of the powerful feature as it allows the developer to perform some operation based on some implicit action. In order to detect the events and to perform some action based on the event a proper Event Frame work is required.

Prior to this release we had introduced the Events “Form Accept” and “Focus”. In this release there has been a major enhancements in the Event Framework as a whole. This document will explain in detail about the events that are supported in TDL. Let us start with an brief overview of the Event Framework and the type of events.

Event Framework Overview

When the user does something an event takes place. Events are action which are detected by a program and can change the state of system or the execution flow. Events can occur based on user actions or can be system generated. In TDL, the Key Framework is mainly used to handle user actions like keyboard and mouse events. This can be considered to be a part of Event Framework.

As we know that TDL is a definition language which does not have any explicit control on the flow of execution, the programmer has no control over what will happen when a particular event occurs. We have certain attributes like SET/PRINTSET which are used to initiate some action on occurrence of event/change of state (like report construction etc). In this scenario there is a need of generic Event Framework, which allows the programmer to trap the events and initiate actions/ set of actions at a state when the event has occurred.

The event framework allows the specification of Event Handler where it is possible to specify Event Keyword, a condition to control event handling and the Action to be performed. The process to detect an event and then execute the specified action is called as event handling.

Types of Events

When user operates the application different types of events are generated. The events are classified as System Events or Object Specific Events based on their origin.

System events are events for which no object context is available when they occur.
Example: Tally application launch

The events that are performed only if the specific object context is available are referred as Object Specific events.
Example: Form Accept event is Form specific event

System Events

In TDL a new type "Events" is introduced in System definition. All the system events are defined under this definition. As of now TDL event framework supports following four system events viz. System Start, System End, Load Company, Close Company.

Syntax

```
[System: Events]
    Label : <EventKeyword> : <ConditionExpr> : <ActionKeyword> : +
            <Action Parameters>
```

Where,
<Label> is a name assigned to the event handler. This has to be unique for each event handler.
<EventKeyword> can be any one of System Start, System End, Load Company, Close Company.
<ConditionExpr> should return a logical value.
<ActionKeyword> any one of the actions.
<Action Parameters> parameters of the actions specified.

The events System Start, System End are executed when the user launches or quits Tally application respectively.
The events Load Company, Close Company are executed when the user loads or closes a company respectively.

Example:

```
[System: Events]

     AppStart1: System Start : TRUE : CALL : MyAppStart
```

The function *'MyAppstart'* is called as soon as Tally application is launched.

Object Specific Events

Objects specific events can be specified for the associated object only.

Example: Before Print event is specific to report object

The attribute ON is used to specify the object specific events as follows:

Syntax

```
      ON : EventKeyword : <ConditionExpr>: <ActionKeyword>: <Action Parameters>
```

Where,
<EventKeyword> can be any one of Focus, Form Accept, Before Print and After Print.
<ConditionExpr> should return a logical value.
<ActionKeyword> any one of the actions.
<Action Parameters> parameters of the actions specified.

ON is a list type attribute, so list of actions can be executed sequentially when the specific event occurs.

- Event – Form Accept

The event Form Accept is specific to Form object hence can be specified only within Form definition. A list of actions can be executed when the form is accepted which can also be based on some condition. After executing the action Form Accept, the current object context is retained. So all the actions that are executed further, will have the same object context.

The event Form Accept when specified by the user overrides the default action Form Accept. So when Form Accept event is triggered then the Form will not be accepted until the user explicitly calls the action Form Accept.

Example:

```
[Form: TestForm]

   On : FormAccept : Yes :HttpPost : @@SCURL : ASCII : SCPostNewIssue:+
          SC NewIssueResp
```

The action Http Post is executed when the event Form Accept is encountered. But the form will not be accepted until the user explicitly calls the action Form Accept on event Form Accept as follows:

```
   On : FormAccept : Yes : Form Accept
```

Now after executing the action Http Post, Tally will execute the action Form Accept as well.

- Event – Focus

The event Focus can be specified within definitions Part, Line and Field. When Part, Line or Field receives focus, a list of actions get executed which can also be conditionally controlled.

Example:

```
[Part: TestPart2]
    On : FOCUS : Yes  : CALL  : SCSetVariables  : $$Line
```

- Event – Before Print

The event Before Print is specific to Report object so it can be specified only within Report definition. The event Before Print is triggered when the user executes Print action. The action associated with the event is executed first and then the report is printed.
A list of actions can be executed before printing the report based on some condition.

Example:

```
[Report: Test Report]
   On : BEFORE PRINT : Yes :CALL: BeforeRepPrint
```

The function *'BeforeRepPrint'* is executed first and then the report *'Test Report'* is printed.

- Event – After Print

The event After Print can be specified for Report, Form, Part and Line definition. The event After Print, first prints the current interface object and then executes the specified actions for this event.

A list of actions can be executed after printing the report based on some condition. Print is an alias for After Print.

Example:

```
[Line: LV AccTitle]
   On : After Print: Yes: CALL : SetIndexLV:#LedgerName
```

The function *'SetIndexLV'* is called after printing the line *'LV AccTitle'*. So if there are 10 lines to be printed, the function will be called ten times.

6.2 Collection Enhancements

Using External PlugIns as a Data Source for Collections

A Dynamic Link Library takes the idea of an ordinary library one step further. The idea with a static library is for a set of functions to be collected together so that a number of different programs could use them. This means that the programmers only have to write code to do a particular task once, and then, they can use the same function in lots of other programs that do similar things.

A Dynamic Link Library similar to a program, but instead of being run by the user to do one thing it has a lot of functions "exported" so that other programs can call them. There are several advantages to this. First, since there is only one copy of the DLL on any computer used by all the applications that need the .library code, each application can be smaller and save disk space. Also, if there is a bug in the DLL a new DLL can be created and the bug will be fixed in all the programs that use the DLL just by replacing the old DLL file. DLLs can also be loaded dynamically by the program itself, allowing the program to install extra functions without being recompiled.

What is DLL?

A **Dynamic Link Library** (DLL) is a library that can be called from other executable code, either from an application or from another DLL. It can be shared by several applications running under Windows. A DLL can contain any number of routines and variables.

Dynamic Link Library has the following advantages:

- **Saves memory and reduces swapping**: Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in memory. In contrast, Windows must load a copy of the library code into memory for each application that is built with a static link library.
- **Saves disk space**: Many applications can share a single copy of the DLL on disk. In contrast, each application built with a static link library has the library code linked into its executable image as a separate copy.
- **Upgrades to the DLL are easier**: When the functions in a DLL change, the applications that use them do not need to be recompiled or re-linked as long as the function arguments and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.

A potential disadvantage of using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module.

Differences between Applications and DLLs

Even though DLLs and applications are both executable program modules, they differ in several ways. To the end user, the most obvious difference is that DLLs are not programs that can be directly executed. From the system's point of view, there are two fundamental differences between applications and DLLs:

- An application can have multiple instances of itself running in the system simultaneously, whereas a DLL can have only one instance.
- An application can own things such as a Stack, Global memory, File handles, and a message queue, but a DLL cannot.

Types of DLL

When you load a DLL in an application, there are two methods of linking i.e., Load-time Dynamic Linking and Run-time Dynamic Linking.
Static Linking happens during program development time where as dynamic linking happens at run time.

- Load time Dynamic Linking /Static Linking

In Load-time Dynamic Linking, an application makes explicit calls to exported DLL functions like local functions. To use load-time dynamic linking, provide a header (.h) file and an import library (.lib) file when you compile and link the application. When you do this, the linker will provide the system with the information that is required to load the DLL and resolve the exported DLL function locations at load time.

- Run-time Dynamic Linking /Dynamic Linking

Dynamic linking means the process that Windows uses to link a function call of one module to the actual function in the DLL.

In Run-time Dynamic Linking, an application calls either the LoadLibrary function or the LoadLibraryEx function to load the DLL at run time. After the DLL is successfully loaded, you use the GetProcAddress function to obtain the address of the exported DLL function that you want to call. When you use run-time dynamic linking, you do not need an import library file.

Please note that Tally will not support for Static Linking, only Dynamic Linking is possible.

DLL Approach in Tally

As we discussed, **Dynamic Link Library (DLL)** is a file that can contain many functions. We can compare it with the library functions provided with many programming languages like C, C++. In Tally, we have the provision to access the external functions by uploading the DLLs. In general, the DLLs can be generated using VC++, VB and .Net framework etc and can be invoked from TDL. Hence using TDL, the functions of DLL can be invoked to perform the necessary operations.

Why it is required in Tally?

In Tally all the functions are not required for all the customers. Only generalized features are included to make the functionality of Tally Simple. But for some customers, simple Tally may not cater the need, for that we need to extend the functionality of Tally by writing the programs in TDL. TDL is designed to handle the functions in built in Tally. For the functions that are not available in Tally we use DLL wherein we can include many functions and use it in Tally by calling those functions.

How to use DLL in Tally?

Loading the DLL’s

1. Copy the DLL file to Tally folder, say C:\Tally.ERP9.

DLL points to the external functions that are to be loaded during the startup of Tally application. Tally loads DLLs from the source to the memory, and DLL functions are available with Tally for usage.

OR

2. Register the DLL file using setup program or Command prompt.

In TDL, DLL can be invoked by using CallDLLFunction and DLL Collection

The CallDLLFunction is a platform function which was already available earlier and is NOT a part of Collection Enhancement. We have discussed it here as an additional information for your knowledge. DLL Collection is an enhancement which has been emphasized in the subsequent sections.

The CallDLLFunction: The internal Function "CallDllFunction" can be used to call an external DLL containing multiple Functions.

DLL Collection: DLL collection can be used to obtain datasets in collection from external Plug-Ins. These Plug-Ins are written as DLL's which can be used to fetch external data (i.e. either from Internet, from external Database etc).

The CallDLLFunction

The internal Function “CallDLLFunction” can be used to call an external DLL containing multiple Functions

Example:

If a DLL “TestDll” contains two functions FuncA and FuncB

- FuncA takes one parameter of String DataType and returns a String
- FuncB takes a parameter of String DataType and Executes the Function. It only returns the status of the function execution (boolean value)

The syntax of invoking the DLL from TDL is as given below:

Syntax

```
[Field : <Field Name>]
      Set As: $$CallDllFunction:<DllName>:<FunctionName>: +
              <Param 1>:<Param 2> ….
```

Where,
<DLLName> The name of the DLL.
<Function Name> Name of the function.
<Param1> and **<Param2>**….. Parameters for the function.

The value returned from the function will be available in the field

To call FuncA

```
[Field: Field2]
   Use   : NameField
;; Assuming Field1 is of Type String
   Set As: $$CallDllFunction: TestDll: FuncA: #Field1
```

To call FuncB

```
[Key: Key1]
   Key    : Ctrl+A
   Action: Set: VarStatus:$$CallDllFunction:TestDll:FuncB:#Field1
```

This key can be associated to a Form or a Menu. The function FuncB in TestDll can be used to return the status of the execution i.e. either Success/Failure (1/0). This value can be obtained in a variable in TDL and used to display appropriate message to the user.

CallDllFunction can be used to call any function which can return single values. If the function returns an array of values then it is advisable to use a DLLCollection.

Let us have an overview on using DLLCollection.

DLL Collection

Tally now provides a TDL interface to obtain datasets in Collection from external Plug-Ins. These Plug-Ins are written as DLL's which can be used to fetch external data (i.e. either from Internet or from external Database etc). These DLL's should return a valid XML which can be easily mapped into TDL Collection. In other words, TDL developer can provide simple string value and/or XML to the DLL function. The DLL gives XML data as an output. Collection takes this data and converts into objects and object methods which can be accessed in TDL like other objects.

DLL collection will be very useful in the following scenarios:

1. Display stock quotes from the internet
2. Get data from different formats like CSV, HTML
3. External device interfaces
4. RFID Barcode scanner
5. Petrol Pump device interface
6. Foot fall count
7. External application interfaces
8. GAS distributor application
9. To get attendance details in Pay Roll through swipe

In DLL collection we are supporting Plug-Ins and ActiveX-Plug-Ins.

- **Plug-Ins**: DLLs created by using C++ or VC++. These DLLs need not be registered separately.
- **ActiveX Plug-Ins**: DLLs created by using VB6, VB.Net, C#.Net etc. These DLLs have to be registered. Registration process is explained in detail later.

At present the Collection definition allows us to work with a C++ DLL, VB DLL, .Net DLL which has a function defined by the name "TDLCollection" (The function has to be created by this name ONLY). This function delivers an XML which is available as objects in the Collection.

Attributes of DLL Collection

The attributes of DLL Collection can be categorized as follows:

- For specifying the source
  - Data Source
- For sending inputs to DLL
  - Input Parameter
  - Input XML
- For validating/formatting the data received from DLL
  - Break On
  - XSLT
- For selective conversion of XML
  - XML Object
  - XML Object Path

1. Data Source

The attribute Data Source is used to set the source of the collection. By using this attribute we are invoking the actual DLL to TDL for further process.

Syntax

```
[Collection: <Collection Name>]
      Data Source : <Type> : <Identity> [:< Encoding>]
```

Where,

<Type> specifies the type of data source. File XML, HTTP XML, Report, Parent Report, Variable, PlugIn XML, AxPlugIn XML.

<Identity> can be file path / the source of DLL.

<Encoding> can be ASCII or UNICODE. It is applicable only for the data sources File XML and HTTP XML.

a. For Plug-in DLL

<Type> is “PlugIn XML”.

<Identity> It identifies the source of DLL ie; the path of DLL

Syntax

```
Data Source: PlugIn XML: <Path to dll>
```

b. For ActiveX DLL

<Type> is “AxPlugin XML”.

<Identity> It identifies the source of DLL ie; < Project Name>.<Class Name>

Syntax

```
Data Source: AxPlugin XML : < Project Name>.<Class Name>
```

Example:

a. For Plugin DLL

```
Data Source: PlugIn XML: mydll.dll
```

b. For ActiveX Plugin DLL

```
Data Source: AxPlugin XML: DLLEg1.MyClass
```

For C#.Net, which has concept of namespaces, the source identifier is “Namespace.ClassName”

Syntax

```
Datasource : AxPlugin XML : <namespace>.<classname>
```

Example:

```
Datasource : AxPlugin XML : testcsharpdll.Class1
```

2. Input Parameter

The attribute Input Parameter is used to pass single string value to the DLL function.

Syntax

```
Input Parameter : <Expression>
```

Where,

<Expression> It returns a string value which is used to pass to the DLL function.

Example:

```
Input Parameter: Test string
```

In this example ‘Test String’ is the string value which is used to pass to the specified DLL.

3. Input XML

The attribute Input XML is used to pass XML format data to the DLL function.

Syntax

```
Input XML : <Post Request Report Name>, <Pre-Request Report Name>
```

Where,

<Post Request Report Name> This is the name of the TDL report. It will be responsible for generating XML data which is passed to the DLL function as input.

<Pre-Request Report Name> This is optional. It is used to get any input from end user.

Example:

```
Input XML: DLLRequestReport
```

4. Break On

Break on attribute used to validate the XML data received from DLL function. If XML data contains the string specified in this attribute which is referred as error string then the validation fails and collection will not be constructed.

Syntax

```
Break On : <String Expression1>, <String Expression 2> …..
```

Where,

<String Expression 1>, **<String Expression 2>**… gives the string value which act as an error string to validate the XML data.

Example:

```
Break On: My Error String
```

If XML data received from DLL function contains “My Error String” then collection will not be constructed, same as in XML collection.

5. XSLT

The attribute XSLT is used to transforming XML document received from DLL function to another XML document. It will be applied before constructing the collection. This attribute is same like in XML collection.

Syntax

```
XSLT: < XSLT File name>
```

Where,

<XSLT File name> Name of the XSLT file name.

Example:

```
XSLT : "C:\\Myfile.xslt"
```

6. XML Object Path

The attribute XML Object Path is used to set the starting XML node from where the objects construction starts. If only a specific data fragment is required it can be obtained using the collection attribute XML Object Path. This attribute is same like in XML collection.

Syntax

```
XML Object Path : <StartNode>: <StartNodePosition>: <Path to start node>
```

Where,

<StartNode> It gives the name of the starting XML Node.
<StartNodePosition> It gives the position of the starting XML Node.
<Path to Start Node> It gives the path of the starting XML Node.
<Path to start node> can be extended like .
<root node>:<child node>:<start position>:<child node>:<start position>:....

Example:

```
XML Object Path: MyNode: 1: Root
```

7. XML Object

The attribute XML Object is used to represent the structure of DLL collection object to which the obtained data is mapped. This is an optional attribute and it is same like in XML collection.

Syntax

```
XML Object : < Object Name>
```

i. In DLL collection all the attributes except Datasource are optional.
ii. All error messages related to DLL collection is stored in dllcollection.log file.

Following examples demonstrates the usage of DLL collection attributes.

Example 1: Data Source Ax PlugInXml

The XML data received from the ActiveXDLL “testdll.class1” is to be displayed in a Report. For this a DLL XML collection is constructed and only a fragment of XML data is to be populated in the collection.

Consider the following input XML fragment:

```
<EmpCollection>
       <Emp>
           <Name>Emp1</Name>
           <EmpId>101</EmpId>
           <Designation>Manager</ Designation >
       </Emp>
       <Emp>
           <Name>Emp2</Name>
           <EmpId>102</EmpId>
           <Designation >Senior Manager</ Designation>
       </Emp>
</EmpCollection>
```

The same XML is used to explain all further examples

The TDL code snippet for generating the report as follows:

```
[Part: DLL Coll Part]
   Lines : DLL Coll Line 1, DLL Coll Line2
   Repeat: DLL Coll Line2: My DLL Collection
   Scroll: Vertical

[Line: DLL Coll Line1]
   Fields: DLL Coll Field 1

   [Field: DLL Coll Field 1]
     Set As : "Retrive fragment EMP List from XML data"

[Line: DLL Coll Line 2]
   Fields: SL No, Emp Name, Emp ID, Emp Desig

     [Field: SL No]
          Use    : Name Field
          Set As : $$Line

     [Field: Emp Name]
          Use    : Name Field
          Set As : $Name


     [Field: Emp ID]
          Use    : Name Field
          Set As : $EmpId

     [Field: Emp Desig]
          Use    : Name Field
          Set As : $Designation
```

```
[Collection: My DLL Collection]
   Datasource     : AxPlugin XML : testdll.class1
   XML Object Path: Emp          : 1 : EmpCollection
```

In the above example, the attribute Datasource is used to set the source of DLL i.e. the class name from the DLL "testdll.class1".

The attribute XMLObjectPath retrieves the XML fragment starting from the first <EMP> tag under the XML tag <EmpCollection> from the specified DLL. The XML data thus fetched from the DLL is then displayed in a Report.

In this example Emp is the name of the starting XML Node, 1 is the position of the starting XML Node and EmpCollection is the path of the starting XML node.

In this case DLL has to be registered. The registration process is explained in detail in the section "Implementation and Deployment of DLL".

Example 2: Data Source PlugInXML

Consider the previous example, instead of ActiveX Plugin DLL, now the data source is simple PlugIn DLL.
The source keyword PluginXML is used in the attribute Data source. In this case only the DLL name must be specified.

The collection definition is as follows:

```
[Collection: My DLL Collection]
   Datasource     : Plugin XML: testdll.dll
   XML Object Path: Emp    : 1: EmpCollection
```

Only one difference from the above example is that, here the DLL registration is not required. Copy the DLL to Tally.ERP 9 folder and execute the program.

Example 3: InputXML

There are scenarios where the DLL expects some input as in XML fragment before sending the required XML output. The DLL XML collection attribute InputXML allows sending the Input XML to the source DLL in XML format.

The collection is defined as follows:

```
[Collection: InputXMLCollection]
   Data Source     : AxPlugin XML : TestDLL.Class1
   XML Object Path: Emp           : 1 : EmpCollection
   Input XML       : PostReqRep, PreReqRep
```

As explained earlier, the attribute InputXML accepts two parameters i.e., PostReqReport and PreReqReport

In this example, the report PreReqRep accepts the user input and the report PostReqRep generates the input XML which is sent to the DLL. The response received from the DLL is populated in the collection InputXMLCollection.

The reports PostReqRep and PreReqRep are defined as follows:

```
[Report: PostReqRep]
   Form  : PostReqReport
   Export: Yes

[Form: PostReqReport]
          .
          .
          .
   [Line : PostReqReport]

     Fields : Short Name Field, PostReqReportName, Name Field, +
              PostReqRepID, Simple Field, PostReqRepDesig
     Local : Field: Short Name Field : Set As : “Name:”
     Local : Field: Name Field       : Set As : “Emp ID:”
     Local : Field: Simple Field     : Set As : “Designation:”

   [Field: PostReqReportName]
     Set As : ##PreReqNameVar
               XMLTag: “Name”
             .
             .
             .
```

```
;; Pre Request Report accepting User Inputs
[Report: PreReqRep]
   Form   : PreReqReport
          .
          .

[Part: PreReqReport]
   Lines:  PreReqReport Name, PreReqReportID, PreReqReportDesig
   [Line: PreReqReport Name]
     Fields : Short Name Field, PreReqReport Name
     Local  : Field : Short Name Field : Info : "Enter Employee Name:"

     [Field: PreReqReport Name]
       Use      : Name Field
       Set As   : "Enter your Name"
       Width    : 50
       Modifies: DLLPreReqNameVar
                 .
                 .
[System: Variable]
   DLLPreReqNameVar : ""
```

Example 4: InputParameter

In scenarios where only one value is to be sent as an input to the source DLL the attribute Input Parameter can be used as follows:

```
[Collection: InpParameterColl]
   Data Source     : AxPlugin XML: TestDLL.Class1
   XML Object Path: result
   Input Parameter: ##InputParameterVar
```

The value of variable "InputParameterVar" is sent as an input to the DLL "TestDLL.Class1". The response received is available in the collection "InpParameterColl".

Example 5: BreakOn

Following code snippet validates the XML received from the DLL "tesdll.class1".

```
[Collection: DLL XML Get CollObjPath]
   Datasource      : AxPlugin XML: testdll.class1
   XML Object Path: Emp         : 1: EmpCollection
   Breakon         : Manager
```

Break On attribute is used to check whether the error string “Manager” exists in the output xml. If the error string exists the XML is considered as an invalid XML and empty collection is created. Otherwise the XML is considered as valid and the collection is populated from the received XML fragment.

Signature of function “TDL Collection” in the DLL

The DLL created using any programming languages when called from Tally must contain a main function named as "TDL Collection". The signature of this function is specific to each programming language.

The detailed signature of function "TDL Collection" in different languages is as follows:

For VC++ DLL

Consider the following example for VC++ DLL to generate an XML fragment for Employee details. This DLL accepts the input from the TDL and it returns an XML file as output from DLL. Using this XML fragment it constructs a collection.

```
extern “C” HRESULT __declspec(dllexport)

TDLCollection (const wchar_t * pInputParam,
                const wchar_t * pInputXML,
                      wchar_t ** pXMLCollection,
                         long * pCollectionSize)
{
   *pCollectionSize = 1024;
   if ((*pXMLCollection = (wchar_t *)
      (CoTaskMemAlloc (*pCollectionSize * sizeof  (wchar_t))))== NULL)
   {
        return -1;
   }
   wcscpy (*pXMLCollection,  L"<EmpCollection>\
                              <Emp>\
                                 <Name>Emp1</Name>\
                                 <EmpId>101</EmpId>\
                                 <Designation>Manager</ Designation >\
                              </Emp>\
                              <Emp>\
                                 <Name>Emp2</Name>\
                                 <EmpId>102</EmpId>\
                         < Designation >Senior Manager</ Designation >\
                              </Emp>\
                                 </EmpCollection>"\
);
```

In the above example there are different inputs giving as a parameter to TDL collection function.

pInputParam: It is an input value to the DLL and it is a string value of collection attribute “Input Parameter”. The TDL passes an input parameter to the DLL.

pInputXML: This is an input value to the DLL and the XML data constructed using collection attribute "Input Xml"

Output values from TDLCollection function:

pXMLCollection: Output buffer containing resultant data, based on this data collection will be constructed.

pCollectionSize: Number of wide characters including the terminating NULL character.

For VB 6 DLL

Consider the example for displaying the values in XML format using VB6. Here also two parameters are passing to TDL Collection.

```
Public Function TDLCollection(pInputParam As String,
                              pInputXML As String) As String

      TDLCollection = “<Root>
                           <Name>Amazing</Name>
                           <Name>Brilliant</Name>
                           </Root>”
End Function
```

In the above example two attributes are giving as parameters to TDL collection function.

- **pInputParam**: Simple string value to the function, as specified in collection definition using attribute “Input Parameter”.
- **pInputXML**: Input value in Xml format, as specified in collection definition using attribute “Input XML”.
- The function must return an output String value in XML format.

For C#.Net DLL

Consider the following example for .Net DLL to convert the input string to upper case. Here TDL collection passes two parameters.

```
public string TDLCollection (string pInputParam, string pInputXml)
{
   string resultxml;  // to contain xml to be sent back to Tally
   if (!String.IsNullOrEmpty(pInputXml))
   {
     resultxml = pInputXml.ToUpper();
   }
   else
   {
     resultxml = null;
   }
        return resultxml;
}
```

In the above example we are passing XML data to TDL collection function. All the data is in various tags are converted to upper case. The Input XML will be as follows:

```
<Root>
   <Name>
        <fname>fname 1</fname>
        <lname>lname 1</lname>
   </Name>
   <Name>
        <fname>fname 2</fname>
        <lname>lname 2</lname>
   </Name>
</Root>
```

The output XML will be like

```
<Root>
   <Name>
       <fname>FNAME 1</fname>
       <lname>LNAME 1</lname>
```

```
            </Name>
            <Name>
                <fname>FNAME 2</fname>
                <lname>LNAME 2</lname>
            </Name>
         </Root>
```

Inputs to TDLCollection function:

pInputParam: It is an input value to the DLL and it is a string value of collection attribute "Input Parameter". The TDL passes an input parameter to the DLL.

pInputXML: This is an input value to the DLL and the XML data constructed using collection attribute "Input Xml".

For VB.Net DLL

In VB.Net the signature for the function TDL Collection is as follows:

```
Public Function TDLCollection(ByVal pInputParam As String,
                              ByVal pInputXML As String) As String
```

Inputs to TDLCollection function:

pInputParam: It is an input value to the DLL and it is a string value of collection attribute "Input Parameter". The TDL passes an input parameter to the DLL.

pInputXML: This is an input value to the DLL and the XML data constructed using collection attribute "Input Xml".

Implementation and Deployment of DLL

Once the DLL is ready for Deployment the following needs to be taken care for implementation of the same

1. The dependency for the particular DLL needs to be checked based on the environment in which it is developed. The necessary environment needs to be installed for the same.
2. The DLL needs to be registered in the system where it is to be deployed. This can be done in two ways
   - Registering the DLL manually
   - By running the setup program which is created for deployment

Dependencies with respect to DLLs created using various Environments

- Created using NET framework: For DLLs created using VB .NET and C# .NET etc we require Microsoft .Net Framework. For example the DLL is created using Visual Studio 2005 then Microsoft .Net Framework 2.0 or above should be installed on the system.
- Created using Visual Basic 6.0: For DLLs created using VB 6 we require service pack 6 to be installed on the system

References

Net Framework can be downloaded and installed from the following link:

http://download.microsoft.com/download/6/0/f/60fc5854-3cb8-4892-b6db-bd4f42510f28/dotnetfx35.exe

Service Pack 6 for Visual Basic 6.0 can be downloaded from the following links:

http://www.microsoft.com/downloads/details.aspx?FamilyId=9EF9BF70-DFE1-42A1-A4C8-39718C7E381D&displaylang=en

Multi part - http://www.microsoft.com/downloads/details.aspx?familyid=83BF08E6-012D-4DB2-8109-20C8D7D5C1FC&displaylang=en

How to register DLLs?

After downloading the necessary environment the DLL needs to be registered before it is being used and called by Tally program. As we already discussed, there are two ways in which a DLL can be registered.

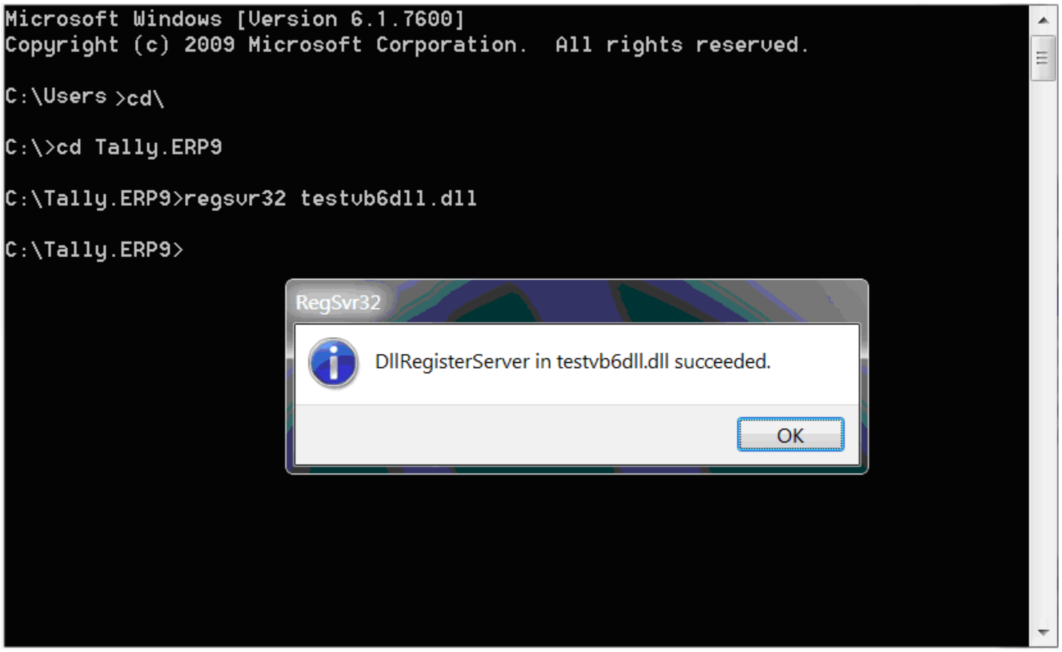
We will take you through the process of Registering a DLL (Manual & Set Up)

Manual Registration

- For VB6 DLLs

To register the DLL manually, do the following steps:

1. Copy the DLL file to the specific folder say C:\Tally.ERP9
2. Open Command Prompt and change the current directory to the folder where DLL is copied, i.e., :\Tally.ERP9
3. Type the command RegSvr32 <DLL Name>
4. After the command is entered in the command prompt displays a message box as shown:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users >cd\

C:\>cd Tally.ERP9

C:\Tally.ERP9>regsvr32 testvb6dll.dll

C:\Tally.ERP9>
```

DllRegisterServer in testvb6dll.dll succeeded.

Figure 1.18 Registering DLL using command prompt

Now you can use the DLL for calling from Tally.

- For .NET DLLs

If you are trying to register .Net DLL we need to use RegAsm command instead of RegSvr32.

Notes

Manual registration does not automatically take care of missing dependencies .So it is always advisable to use Set Up Programs for Registration.

Registering DLLs by using Setup Program

By using this facility double click on the setup program and proceed with installation. It automatically registers the required DLL into the selected folder.

Creating a Set Up Program

The creation of Set Up program varies from one language to another. Please refer to any learning material for Set Up creation specific to your Development Environment. As an example and common usage we will just discuss creating Set Up using VB 6.

Deployment of VB6 DLLs using Package and Deployment Wizard

1. Open the VB project you want to create a setup program.
2. Select Package and Deployment wizard from Add-In menu.
3. If the option is not visible then choose Add-In Manager and double click package and Deployment wizard.
4. Proceed with the Wizard options.

For more details please refer the following links:
http://www.configure-all.com/deployment.php

http://www.developerfusion.com/article/27/package-deployment-wizard/2/

Dynamic Table Support using Unique Attribute

The Unique attribute of Collection definition is used to control the display of unique values in the table for a specified method based on values selected from the table previously in a field. The display of values is changed dynamically based on the field value.

The existing syntax of unique attribute is

Syntax

```
      Unique : <Table Object Method> [,<Field Object Method>]
```

Where,
<Table Object Method> is a method whose value is uniquely displayed in the table.
<Field Object Method> is the storage/method which is associated with the field which is used to control the display of Table values dynamically. If a particular table object method value from the Table is selected in the field, then that value is removed from the table based on the value of **<Field Object Method>**. This parameter is optional.

Example:

```
  [Part: StkBat]
     Repeat : GrpLedLn: StkItemColl

     [Line : GrpLedNm]
       Field  : StkIt, StkBatNm

       [Field : StkIt]
         Use    : Name Field
         Storage: ItemName

       [Field : StkBatNm]
         Use        : Name Field
         Table      : BatList
         Storage    : BtName
         Show Table : Always
         Dynamic    : Yes

[Collection: BatList]
   Title   : "List of Batches"
   Type     : Batch
   Format   : $BatchName,20
   Child of: #StkIt
   Unique   : $BatchName,$BtName

[Collection:StkItemColl]
   Type     : StockItem
   Fetch    : Name

[System : UDF]
   BtName    : String :2010
   ItemName : String :2010
```

The table “Bat List” is used to display batch names in a Table attached to the field “StkBatNm”. The storage associated with the field is “BtName”. Once the Batch name is selected in the field “StkBatNm”, in the next line, the table will populated with batches which are not selected previously in the field.

Assuming, some stock items belong to more than one batch then also the table wont display the common batches since it may have been already selected in the field for a different stock item. To provide this flexibility for controlling the uniqueness of data, the attribute unique is enhanced.

The enhanced syntax is:

Syntax

```
      Unique : <Table Object Method>[,<Field Object Method> +
                [,<Extended method>]]
```

Where ,

<Table Object Method> is a method whose value is uniquely displayed in the table.

<Field Object Method> is the storage/method which is associated with the field which is used to control the display of Table values dynamically. If a particular table object method value from the Table is selected in the field, then that value is removed from the table based on the value of <Field Object Method> This parameter is mandatory if extended method is specified else its optional.

<Extended Method> is a storage/method whose value specifies whether the previous value of field object method should be used to control unique values display in the table. If the current value of the value of <Extended Method> is same as that of previous values then <Field Object Method> value is considered while populating unique value in the table. Other wise the <Field Object Method> value is ignored to set the unique values in the table. This parameter is optional.

The collection and definition is modified as follows so that while populating unique values of Batch names in the table, StockItem name is also considered apart from the value of the field storage/method “BtName”. i.e. if the same stock item is selected in the field which has been selected previously then the field storage/method value “BtName” is considered for controlling display of Batches else it is ignored.

Example:

```
[Collection: BatList]
   Title    : “List of Batches”
   Type     : Batch
   Format   : $BatchName,20
   Child of: #StkIt
   Unique   : $BatchName,$BtName,$ItemName
```

Here the method $Itemname used in the unique attribute is the storage defined in the field ‘*StkIt*’.

Use Case

Consider the following Scenario:

Stock Item	Batch Name
Item 1	Batch A
	Batch B
	Batch C
Item 2	Batch A
	Batch C
Item 3	Batch A
	Batch B
	Batch C

There are two fields in the line one which displays stock item name and the other displays batches. The selected batch is stored in the UDF say BtName.

The following table displays the values in each field and the unique values in the tables based on the selection.

Line No	Value in Field 1	Values in Table	Selected Value in Field 2
1	Item 1	Batch A Batch B Batch C Primary Batch	Batch A
2	Item 2	Batch A Batch C Primary Batch	Batch C
3	Item 1	Batch B Batch C Primary Batch	Batch B

Using Variable as a Data Source for Collections

The collection attribute Data Source is enhanced to support 'Variable' as data source. Now variable element(s) can be gathered as objects in the collection and their respective simple member variables are available as methods. Member List Variables will be treated as sub-collections.

Syntax

```
    Data Source  : <Type> : <Identity> [:<Encoding>]
```

Where,

<Type> is the type of data source. File XML, HTTP XML, Report, Parent Report, Variable.

<Identity> can be file path / scope key words / variable specification based on the type of data source.

<Encoding> can be ASCII or UNICODE. It is applicable for the data source types File XML & HTTP XML.

*Please refer to the topic **"Using Variable as a Data Source for Collections"** under Variable Framework for more clarity with examples.*

6.3 Evaluating expressions by Changing the Object Context with $$ReqOwner Introduced

In a programming language, an expression is a combination of values, variables, operators and functions that are evaluated according to the rules of their precedence and association. Similarly, expressions in TDL can be a combination of Method/Variable/Field/Constant Values, and Function/Formula evaluation results.

Example: For TDL Expression

```
  $Name + ##VarTest + $$MachineDate + @@FormABC + 90 + #FieldXYZ
```

Where,
Name is a Method, **VarTest** is a Variable, **MachineDate** is a Function,
FormABC is a System Formula, **90** is a constant value, and **FieldXYZ** is a Field.

Methods, Variables and Fields are **Leaf components** in an expression as other components like Formulae or Functions finally evaluate into either one of these or result into constants.

A TDL Expression always gets evaluated in the context of an Interface (Requestor) and Data Object. Whenever a report is constructed a Interface object hierarchy is created ie Report contains a Form, Form contains a Part and so on. Every Interface Object is associated with a Data Object. In the absence of explicit data object association, an implicit anonymous object gets associated. A method value is evaluated in context of the data object and Variable and Field value is evaluated in context of Interface object. There may be cases where we would require evaluating these in a context different from the implicit context (Interface and Data). TDL provides many functions which provide the facility to either change the Data or Interface Object Context. A change in Data Object context does not change the current Interface (Requestor) Object context and a change in Interface Object Context does not change the current Data Object Context.

We can mainly categorize these functions into two categories

- Data Object Context Switching Functions
- Interface Object Context Switching Functions

Data Object Context Switching Functions

The Tally database is hierarchical in nature in which the objects are stored in a tree like structure. Each node in the tree can be a tree in itself. An object in Tally is composed of methods and collection. Method is used to retrieve data from the database. A collection is a group of objects. Each object in the collection can further have methods and collection. The Internal Object hierarchy is predefined in Tally and cannot be altered. These can only be persisted in Tally Database.

Every Interface Object exists in context of a Data Object which is associated to it. As discussed above, an expression (specifically method value) gets evaluated in context of the Data Object associated with the Requestor (Interface Object). By using the functions as given below we can change the Data Object Context for expression evaluation.

Switching the Data Object Context does not imply a change in current Requestor (Interface Object)

In all the subsequent examples we will be using the Voucher Data Object hierarchy to demonstrate the various scenarios for Context Change. Hierarchy is as shown in the diagram below.

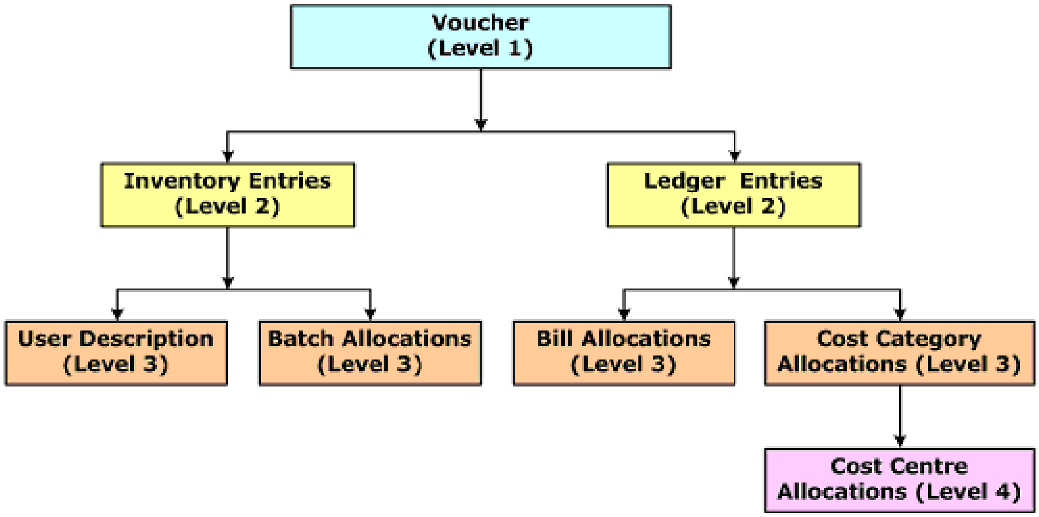


Figure 1.19 Data Object Hierarchy of Voucher

1. Owner

Function Owner evaluates the given expression in the context of parent data object in the Data Object hierarchy chain i.e., $$Owner will change the Data Object to the parent of the current Data Object in context.

For example, if the current object context is Batch Allocations, to access the value from **Inventory Entries** which is its parent data object, Owner function can be used.

Syntax

```
$$Owner:<Expression>
```

Example 1:

In the example given below, let us assume that the field “Bill Allocations Amount” field (Requestor) exists in context of Bill Allocations Data Object. In order to evaluate the method “Amount” from Ledger Entries Object Context we need to use the function $$Owner.

```
[Field: Bill Allocations Amount]
   Set As   : $$Owner:$Amount
```

In the above Field, the **Amount** Method from parent Object **LedgerEntries** is set by using **Owner** Function.

Example 2:

Similarly, let’s assume that the current data object context for the field “Bill Allocations Remarks” is Bill Allocations and we need to evaluate the Method Narration from Voucher Object.

```
[Field: Bill Allocations Remarks]
   Set As  : $$Owner:$$Owner:$Narration
```

In the above Field, the Narration from Object **Voucher** which is 2 levels above the hierarchy is set using Owner Function twice. In other words, we are requesting the **Narration** Method from Owner of Owner.

Alternatively, in the above examples, we can use the dotted method formula syntax.

```
[Field: Bill Allocations Amount]
   Set As  : $..Amount

[Field: Bill Allocations Remarks]
   Set As  : $...Narration
```

In the above examples **..** denotes the parent and **...** denotes the Grand Parent.

2. BaseOwner

Function BaseOwner evaluates the given expression in the context of the base/ primary data object in the Data Object hierarchy chain available with the Report Object (in memory).

*Since the entire Data Object hierarchy is cached with the Object associated at the Report, **Baseowner** Function changes the Data Object context to the Object associated at the Report.*

For example, if the current data object context is Batch Allocations, to access the method value from **Voucher**, **BaseOwner** function can be used.

Syntax

```
      $$BaseOwner:<Expression>
```

Example:

As per the Voucher hierarchy, let’s assume that our current data object context for the field “Bill Allocations Remarks” is **Bill Allocations**. In order to access the value of Method **Narration** from Voucher Object which is the base/primary object in the object hierarchy chain, we can use the function **$$BaseOwner**.

```
[Field: Bill Allocations Remarks]
   Set As  : $$BaseOwner:$Narration
```

In the above Field, the Method Narration from the base Object **Voucher** is set by using **BaseOwner** Function.

Alternatively, in the above example, we can use the dotted method syntax.

```
[Field: Bill Allocations Remarks]
   Set As   : $().Narration
```

In the above example **().** navigates to the Primary/Base Data Object

3. PrevObj

Function PrevObj evaluates the given expression in the context of previous data object of the collection which contains the current data object in context.

Syntax

```
$$PrevObj:<Expression>
```

Example:

Let’s assume that a line is being repeated over a collection of Outstanding Bills which is sorted on PartyName. After every party Info, a Total Line is needed to print the subtotal for current Party.

```
[Line: Outstanding Bills]
   Option : Partywise Totals  : $$PrevObj:$PartyName != $PartyName


   [!Line: Partywise Totals]
     Add : Lines : At Beginning : Party SubTotal Line
```

In the above example, an optional line will be included only if Previous Object's Method PartyName is not equal to current Object's Method PartyName.

4. NextObj

Function NextObj evaluates the given expression in the context of next data object of the collection which contains the current data object in context.

Syntax

```
$$NextObj:<Expression>
```

Example:

Let's assume that a line is being repeated over a collection of Outstanding Bills which is sorted on PartyName. After every party Info, a Total Line is needed to print the subtotal for current Party.

```
[Line: Outstanding Bills]
    Explode : Partywise Totals  : $$NextObj:$PartyName != $PartyName
```

In the above example, a part is exploded provided next object's method PartyName is different from current object's method PartyName. This will enable explosion for each party only once and thereby we can easily achieve the subtotal line as desired.

5. FirstObj

Function FirstObj evaluates the given expression in the context of first data object of the collection which contains the current data object in context.

Syntax

```
$$FirstObj:<Expression>
```

Example:

Let’s assume a line is being repeated over the ledger collection wherein a field, we require the first object's name to be set.

```
[Field: First Name]
   Set As: $$FirstObj:$Name
```

In the above example, a Field First Name is set to **First Object** of Name Method in a Collection.

6. LastObj

Function LastObj evaluates the given expression in the context of last data object of the collection which contains the current data object in context.

Syntax

```
$$LastObj:<Expression>
```

Example:

Let's assume a line is being repeated over the ledger collection wherein a field, we require the last object's name to be set.

```
[Field: Last Name]
   Set As: $$LastObj:$Name
```

In the above example, a Field Last Name is set to Last Object of Name Method in a Collection.

7. TgtObject

As we already know that apart from Interface (Requestor) and current Data Object Context, there is one more context available with reference to User Defined Functions and Aggregate Collections i.e, the Target Object Context. In case of functions, the object being manipulated is the Target Object. In case of aggregate Collection the object being populated in the resultant collection is the Target Object.

There are scenarios where the expression needs to be evaluated in the context of Target object, in such cases the $$TgtObject can be used. Using the $$TgtObject values can be fetched from the target object without setting the target object as the current context object.

Syntax

```
$$TGTObject:<Expression>
```

Example 1:

Consider writing a Function to import a Voucher from Excel wherein the Source Object is Collection gathered out of Objects in Excel Worksheet and the Target Object being the Voucher and its sub objects. While setting value to Inventory Entries sub object, the Target Object is changed to Inventory Entries and the Source Object continues to be Excel Objects. In order to set values to methods, Quantity and Rate, Stock Item context is required since Unit Information is available for Item. Hence, TGTObject Function is prefixed to the Expression @BillQty and @BillRate in order to evaluate these Methods in the context of Target Object which is the Inventory Entries Object.

```
[Function: Import Voucher]

   Local Formula: BillQty : $$AsQty :$ExcelBilledQty

   Local Formula: BillRate: $$AsRate:$ExcelItemRate


   90 : INSERT COLLECTION OBJECT : Inventory Entries

   100: SET VALUE: BilledQty     : $$TgtObject:@BillQty

   110: SET VALUE: Rate          : $$TgtObject:@BillRate

   120: SET TARGET ..

   130: SAVE TARGET
```

Example 2:

Consider another example where while populating a summary collection of Sales Voucher, we need to track the maximum sales amount for each Item with the date on which the maximum sales triggered.

In the following example, while populating the Summary Collection, Method **MaxItemAmt** is being computed for Maximum Amount. Subsequently, Date is also computed by validating if current object's Amount is greater than previous computed Amount. Since, Maximum Amount so far is computed and accumulated in the Target Object being populated, we need to access it using function **TGTObject**. Hence, **$$TgtObject:$MaxItemAmt** evaluates the Method **MaxItemAmt** in the context of computed Target Object **MaxItemAmt**.

```
[Collection: Src Voucher]

     Type    : Vouchers   : VoucherType

     ChildOf: $$VchTypeSales

[Collection: Summ Voucher]

     Source Collection: Src Voucher

     Walk             : Inventory Entries

     By               : ItemName   : $StockItemName
;; The following returns the Date and Amount for an Item on which Maximum sales has happened
     Aggr Compute: MaxDate: SUM: IF  $$IsEmpty:$$TgtObject:$MaxItemAmt +

                 OR $$TgtObject:$MaxItemAmt <$Amount THEN $Date ELSE +
                        $$TgtObject:$MaxDate
;; MaxItemAmt is the method in Target Object hence TgtObject function is used to evaluate the Method
;; MaxItemAmt in Target Object Context
     Aggr Compute: MaxItemAmt: MAX: $Amount
```

8. LoopCollObj

As we are aware that it is now possible to gather Data Collection in context of each object of another collection which is referred to as a Loop Collection. To access the methods of **Loop Collection** Objects from within Data Collection, LoopCollObj is used with which the expression is evaluated in the context of the Loop Collection Objects.

Syntax

```
      $$LoopCollObj:<Expression>
```

Example:

To see a consolidated list of vouchers across all the loaded companies.

```
[Collection: Company Collection]
   Type    : Company
   Fetch   : Name

[Collection: Vouchers of Multiple Companies]
   Collection : MultiCmpDB VchCollection: Company Collection
   Sort       : Default                 : $Date, $LedgerName

[Collection: MultiCmpDB VchCollection]
   Type     : Voucher
   Fetch    : Date, Vouchernumber, VoucherTypeName, Amount, MasterID,+
                LedgerName
   Compute : Owner Company: $$LoopCollObj:$Name
```

In the above example, **LoopCollObj** function, changes the context to the Loop Collection Objects which is the Company Collection and hence, returns company name.

9. ReportObject

Function ReportObject evaluates the given expression in the context of the Data Object associated with the Report Interface Object.

One of the important Use Case of Report Object is its usage in purview of in memory Collection gathering. Whenever a collection is gathered it is retained in memory with the Data Object of the current Interface (Requestor) Object. If the same collection is being used in expressions again and again then it’s beneficial from the performance point of view to attach it to the Report Object and evaluate it in context of Report Object n number of times. This eliminates the need to re-gather the collection every time in context of other Data Objects.

Syntax

```
$$ReportObject:<Expression>
```

Example 1:

From a **Bill Allocations Data Object** context, Voucher Number of Report Object Voucher is required.

```
[Field: Bill No]
   Set As: $$ReportObject:$VoucherNumber
```

Example 2:

In a Report, Sales of each Item against corresponding Parties required.

```
[Collection: CFBK Voucher]
   Type   : Voucher
   Filter: IsSalesVT

[Collection: CFBK Summ Voucher]
   Source Collection: CFBK Voucher
   Walk             : Inventory Entries
   By               : PName     : $PartyLedgerName
   By               : IName     : $StockItemName
   Aggr Compute     : BilledQty : SUM: $BilledQty
   Search Key       : $PName + $IName

[Field: CFBK Rep Party]
   Use      : Qty Primary Field
   Set as   : $$ReportObject:$$CollectionFieldByKey:$BilledQty: +
              @MyFormula:CFBKSummVoucher
   MyFormula: ##PName + #CFBKRepName
```

In the above example, **ReportObject** Function during its first execution retains the collection within the Voucher Object (which is the Data Object associated with the Report Object). During the subsequent calls, the method values are fetched from the Objects available in the Report Data Object instead of re-gathering the entire Collection again. This helps in performance improvement drastically.

10. ReqObject

The function ReqObject evaluates the given expression in context of the Data Object associated with the Interface (Requestor) Object. There may be scenarios where during the expression evaluation the Data Object context changes automatically and all the methods referred to are evaluated in context of the changed Data Object Context. The Data Object associated with the Interface (Requestor) Object is lost. Specifically in those cases, where we need to evaluate methods in context of the data object associated with the Interface (Requestor) Object we will use the function $$ReqObject.

Syntax

```
      $$ReqObject:<Expression>
```

.

Example:

A Report is required to display Ledgerwise Sales Totals

```
[Field :Fld LedSalesTotal]
   Set As   : $LedgerSalesTotal

[#Collection : Ledger]
   Compute :LedgerSalesTotal :$$FilterAmtTotal :LedVouchrs: +
             MyParty: $Amount

[Collection: Led Vouchers]
   Type     : Voucher
   Filter   : OnlySales

[System: Formula]
   My Party   : $PartyLedgerName = $$ReqObject:$Name
   Only Sales: $$IsSales:$VoucherTypeName
```

In the above example, a new method **LedgerSalesTotal** is added in the Ledger Object to compute the Sales Total from all the Vouchers filtered for the current Party Ledger Object. The Interface Object (Requestor) for this method is the field “FldLedSalesTotal”. In the Formula **My Party**, current Ledger Name must be checked with the Party Ledger Name of the Voucher Object which is the current Data Object context. The Data Object associated with the Requestor is the “Ledger Object”. So, in order to evaluate the method $name from the Interface (Requestor) Object's Data Object context the function $$Reqobject must be used.

11. ObjectOf

As we are already aware, that we have the capability to identify a Part and Line Interface Object using a unique Access Name. A Form/Report can be identified from any level using the Definition Type. The function ObjectOf is used to evaluate the expression in context of the Data Object associated with the Interface Object identified by the Access Name.

The Interface Object being referred to, should be assigned a unique AccessName via **Access Name** attribute.

Syntax

```
$$ObjectOf:<DefinitionType>:<AccessNameFormula>:<Evaluation Formula>
```

Example:

The Part “Cust Object Association” is associated with the Ledger Object “Customer”. It is identified by the Access Name “CustLedger”.

```
[Part: Cust Object Association]
   Lines      : Cust Object Association
;; Object associated at Part
   Object Ex  : (Ledger, "Customer").
;; Access Name specified so that this part can be accessible
   Access Name: "CustLedger"
;; In some other field across parts, a field can access the methods of Ledger Object associated with part "CustObjectAssociation" we can use ObjectOf Funtion

[Field: Ledger Parent]
   Set as : $$ObjectOf:Part:"CustLedger":$Parent
```

In the above example, Field Ledger Parent from a different Part accesses the method “$Parent” from the Ledger object “Customer” as it is the Object associated with the part “Cust Object Association” identified by Access Name “CustLedger”.

12. Table

The function Table evaluates the **expression** in the context of the table object which is selected in the given **Field**.

Syntax

```
$$Table:<Field Name>:<expression>
```

Example:

In the code snippet given below, the table is displayed in the field “Vehicle Number”. In the other fields Vehicle Type, Vehicle YOP $$Table is used to evaluate the methods $VehType,$VehYOP in context of the Data Object selected in the field “Vehicle Number”.

```
[Field: Vehicle Number]
   Table     : List of Vehicles
   Show Table: Always
```

```
[Field: Vehicle Type]
   Set as : $$Table:VehicleNumber:$VehType

[Field: Vehicle YOP]
   Set as : $$Table:VehicleNumber:$VehYOP

[Collection: List of Vehicles]
   Type    : Veh AggUDF: Company
   ChildOf: ##SVCurrentCompany
   Format : $VehNo, 20
   Format : $VehType, 40
   Format : $VehYOP, 4
   Fetch  : VehNo, VehType, VehYOP
;; For Remote Client End
```

13. TableObj

The function TableObj is similar to the function **Table**. The expression is evaluated in context of the Data Object selected in the Table in the field specified. The difference of this with Table function is that, in case no object is selected in the Table or expression evaluation fails $$Table returns a blank string. In such a case $$TableObj returns a logical value (FALSE) as the result.

Syntax

```
      $$TableObj:<Field Name>:<expression>
```

Example:

A Field needs to be skipped based on the selection of the table in a field.

```
[!Field: VBOrdDueDRNote]
      Skip  : $$TableObj:VCHBATCHOrder:$$IsOrder
```

In the above example, if the Object selected in the Field **VchBatchOrder** is an Object Order, then the current field needs to be skipped.

Interface Object Context switching functions

Objects used for designing the User Interface are referred to as Interface objects. Report, Form, Menu etc. are interface objects. Interface objects like Report and Menu are independent items and can exist on their own. The objects Form, Part, Line and Field can't exist independently. They must follow the containment hierarchy as given below

A Report can have more than one Form, Part, Line and Field definitions but at least one has to be there. The hierarchy of these definitions is as follows:

- Report uses a Form
- Form uses a Part
- Part uses a Line
- Line uses a Field
- A Field is where the contents are displayed or entered

We can take an example of a Simple Customized Invoice Report (as given in the diagram below) in order to understand the containment hierarchy of Interface Objects.

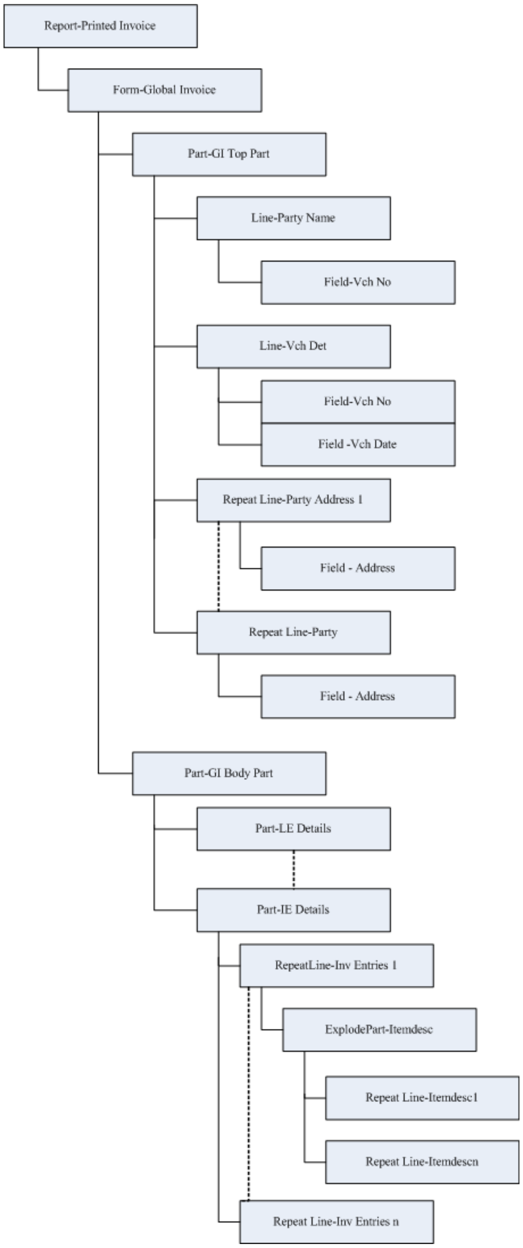


Figure 1.20 Interface Object Hierarchy

A set of available attributes of interface objects are predefined by the platform. A new attribute can not be created for an interface object. Interface objects are always associated with a Data Object and essentially add, retrieve or manipulate the information in Data Objects.

At the run time when a report is constructed after the evaluation of all of the above, a complete hierarchy of Interface Objects is created. As we have already discussed, an expression is evaluated in context of the current Interface Object which is referred to as the Requestor and the Data Object associated to it. We will now discuss the switching functions which will change the Interface Object Context for expression evaluation.

Switching the Interface (Requestor) Object Context does not imply a change in current Data Object.

1. AsReqObj

Function AsReqObj is used to save the Interface (Requestor) context to the current object for the evaluation of the expression. All the future references done using ReqObject will point to the saved Interface Object context. The actual requestor is overridden using the function AsReqObject.

Syntax

```
$$AsReqObj:<Expression>
```

Example:

In the example below, a Table of Company Vehicles is displayed in a Field “Select Vehicle” which exists in context of the Voucher Object. The table is filtered on the basis of Unused Vehicles.

```
[Field: Select Vehicle]
;; In Voucher Entry
   Table  : CMP Vehicles
   Storage: VCHVehicle

[Collection: CMP Vehicles]
   Type      : Company Vehicles : Company
   Childof   : ##SVCurrentCompany
   Format    : $VehicleNumber, 20
   Format    : $VBrand, 10
   Title     : “Company Vehicles”
   Filter    : Unused Veh
```

```
[System: Formula]
   Unused  Veh   : $$AsReqObj:$$FilterCount:PrevSalesVchs: +
                   UsedVehicle = 0
   Used Vehicle : $$ReqObject:$VehicleNumber = $VCHVehicle
   Only Sales    : $$IsSales  :$VoucherTypeName

[Collection: PrevSalesVchs]
   Type   : Voucher
   Filter: Only Sales
```

In the above example,

- Field **Select Vehicle** is the Interface (Requestor) Object, which is associated with Data Object **Voucher**.
- **Table/ Collection** of Company Vehicles is displayed in the Field.
- Table is filtered for **Unused vehicles**.
- This collection contains the list of Vehicle Numbers which needs to be compared with the ones used in the previous sales vouchers. Since Requestor is the Field with the data object Voucher, Function **ReqObject** will get evaluated in the context of Voucher Object which is not expected. Hence to make the current collection i.e., **CMP Vehicles** as **requestor object** for future reference, **AsReqObj** Function is used.
- In the **FilterCount** Function, when the object context changes to the list of sales vouchers, **ReqObject** Function evaluates the parameter **$VehicleNumber** in the context of requestor Collection **CMP Vehicles** set using **AsReqObj** Function earlier and compares the same with the Voucher UDF **VchVehicle** stored in the respective vouchers.

2. ReqOwner

Function ReqOwner evaluates the given expression in context of the Interface (Requestor) objects Owner in the current Interface object hierarchy. For instance, Report is a owner requestor for Form; Form is a owner requestor for Part and so on. From Line, when **ReqOwner** Function is used, the expression gets evaluated in the context of the Part containing the current line.

Syntax

```
$$ReqOwner:<Expression>
```

Example:

```
[#Menu: Gateway of Tally]
   Add: Key Item: ReqOwner Sample: W : Alter: ICCF ReqOwner

[Report: ICCF ReqOwner]
   Form       : ICCF ReqOwner
   Variable   : VarReqOwner: String: "Keshava"
```

```
[Form: ICCF ReqOwner]
   Parts: ICCF ReqOwner

   [Part: ICCF ReqOwner]
     Lines: ICCF ReqOwner

     [Line: ICCF ReqOwner]
       Fields: ICCF ReqOwner

       [Field: ICCF ReqOwner]
         Set As     : $$FunctoreturnReqOwner
         Set Always: Yes

[Function: FunctoreturnReqOwner]
   Variable: VarReqOwner: String: "Madhava"
   Variable: Temp       : String: $$ReqOwner:##VarReqOwner


   01: MSGBOX: ##VarReqOwner : ##Temp
   10: RETURN: $$ReqOwner    : ##VarReqOwner
```

In the above example, Variable **VarReqOwner** is declared & initialized in a Report as well as a function. From the Field, this function **ReqOwnerFunc** is referred to perform some computation and return the result. Since, **ReqOwner** is used in the Function and Field is the Requestor Owner for Function, Field walks back the Interface (Requestor) Object hierarchy to fetch the Variable value. Hence, the Variable value **Keshava** of the nearest Interface Object i.e., the Report is returned.

3. AccessObj

As we are already aware, that we have the capability to identify a Part and Line Interface Object using a unique Access Name. The function AccessObj changes the Interface Object context to the one identified by the Access name to evaluate the expression

The Interface Object being referred to should be assigned a unique **AccessName** via Access Name attribute.

Syntax

```
$$AccessObj:<DefinitionType>:<AccessNameFormula>:<Evaluation Formula>
```

Example:

```
[Line: ABC]
   Access Name : "AccABC"
```

```
[Field: XYZ]
   Set As: $$AccessObj:Line:“AccABC”:##VarABC
```

In the example above the function $$AccessObj changes the Interface Object context from the field “XYZ” to the line “ABC” which is identified by Access Name “AccABC”. The variable value is evaluated in context of the line “ABC”.

4. ExplodeOwner

Function ExplodeOwner changes the Interface (Requestor) Object to the Line owning the current exploded Part and evaluates the given expression i.e., Field and Variable Values in the context of Interface Object

Syntax

```
      $$ExplodeOwner:<Expression>
```

Example:

In the following example, Field **NameField** is being evaluated in the context of Line **Smp InvEntries** which owns the current exploded part **Smp Expl Part**

```
[Line: Smp InvEntries]
   Fields : Name Field
   Local  : Field: Name Field: Set As: $StockItemName
   Explode: Smp Expl Part

[Part: Smp Expl Part]
   Lines  : Smp Batch Allocations
   Repeat : Smp Batch Allocations: Batch Allocations
   Scroll : Vertical

   [Line: Smp Batch Allocations]
     Fields: Name Field
     Local : Field: Name Field: Set As:  $$ExplodeOwner:#NameField
```

5. PrevLine

When the line is repeating, we may require to evaluate an expression in context of the previous line for example, we might require to fetch the field values stored in previous line for an expression in the current line. The function PrevLine is used to change the Requestor to the Previous Line for expression evaluation.

Syntax

```
      $$PrevLine:<Expression>
```

The PrevLine function not only changes the Interface (Requestor) Object context but also changes the Data Object context to the Object associated with the Requestor.

Example:

In the following example, in case of repeated lines where subtotals are required to be displayed or printed for same party, we can explode a subtotal line after comparing previous line's Ledger and current line's Ledger. If the field values are not the same, then subtotal line is exploded.

```
[Line: PrevParticulars]
   Explode : PrevParticulars ExpPart : $$PrevLine:+
             #PartyParticulars != #PartyParticulars
```

6.4 Variable Framework with Compound Variables Introduced

Variables in TDL (Tally Definition Language) are entities which can hold values during the execution of a program. The values of these variables are initialized when it is created and can change during the entire execution of program. Program can change the variable value by specifying expressions which are evaluated and the value of the variables which are set.

Variables are context free structures which do not require any specific object context for manipulation. Variables are declared by name and can be operated using the same name. It is also possible to access and operate variables declared at the parent scope also. Variables are light weight data structures which are very simple to operate and provide the capability of storing multiple values of same type and different types as well. It is also possible to perform the various manipulation operations like insert/update/delete/sort and find. These are mainly used to perform complex computations.

Variable can hold a single value, or more than one value of same type or even different types. It can be declared at various scopes such as Report, Function and System Level. The various types of variables available in TDL are explained as below.

1. **Simple Variable**

Simple variables allow storage of a single value of the specified data type.

2. **Simple Repeat Variables**

The Simple Variable can hold method values of multiple objects of a collection based on an implicit index. This concept is used in Columnar Reports only where the lines should be repeated vertically and the fields should be repeated horizontally.

3. **Compound Variable**

Compound Variables allows us to store values of different data types. This is achieved by making the variable itself compound, by allowing variable declaration inside itself. These sub variables are called member variable of the main variable.

A member variable can be a single instance or a list variable. A member variable can be a compound variable and can have members again, and therefore any hierarchy can be created.

Compound variables help grouping of related information together into one specification.
In another terms, we can think about compound variables as an 'object'. Following table shows the similarities between an object and a compound variable.

Object	Compound Variable
Can have methods	Can have Simple Variable as its member
Can have repeated methods (simple collections)	Can have a Simple List Variable as its member
Can have collections (compound collections)	Can have Compound List Variable as its member
Cannot have objects under it directly	Can have a Compound Variable as its member

We can have a comparison between the internal Data Object **'Voucher**' and a Compound Variable **'CLV Emp'** to understand the similarities between an Object and Compound Variable. For instance, the Compound Variable **'CLV Emp'** is defined as below:-

```
[Variable: CLV Emp]
   Variable     : Name        : String
   Variable     : Designation : String
   Variable     : Age         : Number
   Variable     : Salary      : Amount
   List Variable: Contact Nos : String
   List Variable: Relatives
   Variable     : Contact Address

;; Defining Compound Variable
[Variable: Relatives]
   Variable: Name    : String
   Variable: Age     : Number
   Variable: Relation: String
   Variable: Salary  : Amount

;;Defining another compound variable
[Variable: Contact Address]
   Variable:  Street Name   : String
   Variable:  City Name     : String
```

Object : Voucher	Compound Variable : CLV Emp
Object “Voucher” is having methods directly under it such as Date, Voucher Number, Narration etc.,	Compound Variable “CLV Emp” is having Simple Member Variables such as Name, Age, Salary etc.,
Voucher is having the repeated method BasicBuyerAddress (Simple Collection)	CLV Emp is having the Simple List Member Variable ‘Contact Nos’
Voucher is having the collection “Inventory Entries” (Compound Collection).	CLV Emp is having the Compound List Member Variable ‘Relatives’
Voucher object is not having another voucher (primary object) under it directly.	CLV Emp is having the another Compound Member Variable ‘Contact Address’

4. List Variable

A variable at the declaration time can be either declared as a single instance or as a list. List variable is a container (data structure) variable and hence they are not defined. Variables can be declared as list.

List Variable can hold one or more variables which can be either a simple or compound variables. Each of these is called Element Variables. Element Variable holds value as well as key if specified. The key is optional, and hence without a key also elements can be added to list variables. The value of key specified for each of the element variables must be unique.

□ Simple List Variable

Simple Variable can be declared as a list. Simple List Variables can hold multiple values of single data type.

□ Compound List Variable

Compound Variable can be declared as a list. Compound List Variables can hold multiple values of different data types.

Usage

Variable Definition and Attributes

A Variable definition is similar to any other definition. The behavior of the variable is specified by the programmer via Variable definition.

Syntax

```
      [Variable: <Variable Name>]
          Attribute: Value
```

A meaningful name which determines its purpose can be given as a variable name.

Let us discuss the attributes of Variable definition in detail.

Type

This attribute determines the Type of value that will be held by the variable. All the data types supported by TDL such as String, Number, Date etc., can be used to specify the variable data type. In the absence of this attribute, a variable assumes to be of the Type String by default.

Syntax

```
      [Variable: <Variable Name>]
            Type        : <Data Type>
```

Example:

```
[Variable: GroupNameVar]
   Type: String
```

In this example, a variable which holds the data type of string is defined.

Default

Default value of the variables can be specified during definition using **DEFAULT** attribute. This is the initial value which is assigned to the variable when it is instantiated / declared. We can also specify the default value during declaration / instantiation. The difference is that the default value specified using this attribute at the definition time will be applicable to all instances of the variable declared (at any scope). Default value specified while declaration will apply only to the specific instance.

Declaration and scope will be covered in detail subsequent topics. The above explanation will be more clear after that.

Syntax

```
      [Variable: <Variable Name>]
         Default:  <Default Value>
```

Example:

```
[Variable: GroupNameVar]
   Type    : String
   Default: $$LocaleString:"SundryDebtors"
```

In this example, the default value for the variable is set as “Sundry Debtors”.

Volatile

If the **Volatile** attribute in variable definition is set to **Yes**, then the variable is capable of retaining previous values from the caller scope. The default value of this attribute is **Yes**. ie., if the variable by the same name is declared in the called Report/Function and the volatile attribute is set to "Yes", then in the called Report it will assume the last value from the caller Report. The default value of the attribute Volatile is always **"Yes"**

In order to understand it better, let us elaborate the above further. When a variable is declared / instantiated it assumes a default value. The default value which it assumes is controlled by the following factors.

1. If volatile is set to "Yes" for a variable in its definition and is instantiated / declared inside function/report, and the variable by the same name exists in the parent scope, then it will take its default value from the parent scope. If no variable by the same name exists in the parent scope, it will take the default value specified within the definition.

2. If default value is specified within the declaration itself, it will assume that value.

If a new report **Report2** is initiated, using a volatile variable **GroupNameVar,** from the current report **Report1**, the same variable in Report 2 will have the default value as last value saved in Report 1. Within Report 2 the variable can assume a new value. Once the previous report **Report1** is returned back from **Report2**, then the previous value of the variable will be restored. A classic example of this is a drill down **Trial Balance**.

Syntax

```
[Variable: <Variable Name>]
      Volatile: <Logical Value>
```

Example:

```
[Variable: GroupNameVar]
   Type     : String
   Volatile: Yes
```

Volatile Attribute of **GroupNameVar** Variable is set to **Yes**, which means that Group NameVar can inherit values from one Report to another.

Variables defined at the function level are Non Volatile by default. They do not inherit the values from the caller scope.

Notes

Scope will be discussed in detail in the subsequent topics

Persistent

This Attribute decides the retention periodicity of the variable. i.e till when will it retain the value, i) till application termination or ii) after application termination as well. Setting the attribute **Persistent** to **Yes**, means that the value saved during the last application session will be retained permanently in the system. When the next session of Tally is started it will take its initial value from the value saved in the previous session. i.e., the latest value of the variable will be retained across the sessions. Please note that Variables declared at the system scope can only be persisted.

A List variable at a system scope can also be persisted by specifying the persistent attribute for its element variable (whether it is simple/compound) within the definition. Inline variables even at system scope cannot be persisted. We will discuss about inline variable declaration in the further topics.

Syntax

```
      [Variable: <Variable Name>]
            Persistent: <Logical Value>
```

Example:

```
[Variable: SV Backup Path]
   Type       : String
   Persistent: Yes
```

The Attribute **Persistent** of the variable **SV Backup Path** has been set to **Yes** which means that it retains the latest path given by the user even during the **subsequent sessions of Tally**.

Notes

All the Persistent Variable Values are stored in a File Named TallySav.Cfg in the folder path specified for Tally Configuration file in F12 -> Data Configuration. Each time Tally is restarted, these variable values are accessed from this file.

Repeat

The attribute Repeat for a variable is used for its usage in Columnar Reports. It accepts Collection name and optional method name as a parameter. Multiple values are stored in the variable based on an implicit Index. Method value of each object of the collection will be to be picked up and stored in the variable based on implicit index. In case method name is not specified the variable name is considered as the method name and picked up from the collection.

Syntax

```
      [Variable: <Variable Name>]
           Repeat: <Collection Name>[:<Method Name>]
```

Where,
<Variable Name> is the name of the variable.
<Collection Name> can be an expression which evaluates to a collection name.

<Method name> is the name of the method whose value needs to be picked up from each object of the collection. If not specified, variable name is considered as the method name.

Example:

```
[Variable: SVCurrentCompany]
   Volatile     : Yes
   Repeat       : ##DSPRepeatCollection
```

Let us suppose the variable ‘DSPRepeatCollection’ holds the value “List of Primary Companies”. Method value ‘SVCurrentCompany’ will be gathered from the each object of the collection and stored in the index 1, index2 and so on.

*We will elaborate further about Repeat Attribute under the topic **“Implication of Repeat Variables in Columnar Report”**.*

Variable

The attribute **Variable** is used to define the member variable (Simple/Compound) for a Compound Variable.

Syntax

```
      [Variable: <Variable Name>]
            Variable: <Variable Names> [:<Data Type>[:<Value>]]
```

Where,

<Variable Names> is the list of Simple or Compound Variables separated by comma.

<Data Type> is used to specify the data type of Simple Variable. In case of Compound Variable, data type cannot be specified here as it consists of members belonging to various data types. If the data type is not mentioned, the primary variable definition is mandatory.

<Value> is the default/initial value provided for the variable.

Specifying both **<Data Type>** and **<Value>** are optional.

If the data type is specified, then it is called inline declaration of variable. [We will learn about inline declarations and Compound Variables further].

Example:

```
[Variable: CLV Emp]
   Variable: Name   : String
   Variable: Age    : Number : 25
   Variable: Salary: Amount
   Variable: Relatives
```

In this example, the simple variables Name, Age, Salary and the compound variable ‘Relatives’ are defined as members for the Compound Variable **CLV Emp**.

List Variable

The attribute **List Variable** is used to specify a list of either a Simple or Compound Variable.

Syntax

```
      [Variable: <Variable Name>]
          List Variable: <Variable Names> [:< Data Type>[:<Value>]]
```

Where,

<Variable Names> is the list of Simple or Compound Variables separated by comma.

<Data Type> is the data type of Simple Variable. In case of Compound Variable, data type cannot be specified here as it consists of members belonging to various data types.

<Value> it denotes the no of elements in the list.

Specifying both **<Data Type>** and **<Value>** are optional.

Example:

```
[Variable: CLV Emp]
   Variable      : Name   : String
   Variable      : Age    : Number
   Variable      : Salary: Amount
   List Variable: City   : String : 3
   List Variable: Relatives

[Variable: Relatives]
   Variable: Name     : String
   Variable: Age      : Number
   Variable: Relation: String
   Variable: Salary   : Amount
```

In this example, in addition to simple variables, a simple list variable **City** and a compound list variable **Relatives** are defined as members using the attribute **List Variable**. A separate definition is required for the compound list variable **Relatives** as it holds the multiple values of different data types. [We will learn about List Variables in the forthcoming topics]

Variable Declaration and Scope

Variables can be declared at various scopes. The availability of the variable within the definition under which it is declared is called as the scope. The lifetime of the variable will be within the scope. For example, if the scope of a particular variable is within a function, then the variable will last till the function is executing and then is destroyed.

Variables can be declared at System, Report and Function scopes. Let us have a detailed look on the variable scopes.

System

Variables declared at system level will starts its life when the application starts, and will be alive till the application termination.

System variables are declared using a special [System: Variable] definition. The variables declared at system scope are accessible everywhere in the system.

Syntax

```
[System: Variable]
   Variable Name : <Initial Type Based Value>
                   Or
   Variable: <Variable Names> [:<Data Type>[:<Value>]]
   List Variable: <Variable Names> [:<Data Type>[:<Value>]]
```

Where,
<Initial Type Based Value> is the initial value specified to the variable.

The variables can be declared at the system scope by using the above. The attribute Variable and List Variable usage is same as described above in the "Variable Definition".

Example:

```
[System: Variable]
   BSVerticalFlag : No
```

:
The BSVerticalFlag Variable is declared in System Scope. Hence, this variable value being modified in a Report is retained even after we quit and re-enter the Report.

Report

Variables declared at report definition are termed having Report Scope. These variables will exist till the life of the report.

The variables declared at report scope are accessible from the report itself and all TDL elements which are executed from within this report such as another report, function etc.

Report variables will get its default value from the definition specification or from the declaration specification or values will be inherited from the owner scope if variable is marked as volatile.

Report allows two special attributes SET and PRINT SET to set / override the values of the variable during the startup of the report either in display / print mode respectively.

Form definition also has a SET attribute which overrides the variable's value during startup creation and subsequent re-creation of the form during any refresh / regeneration.

We will study about these value specification attributes in detail under the topic "Manipulating Simple and Compound List Variables".

Syntax

```
[Report: <Report Name>]
    Variable      : <Variable Names>
          Or
    Variable      : <Variable Names> [:<Data Type>[:<Value>]]
          Or
    List Variable : <Variable Names> [:<Data Type>[:<Value>]]
```

The variables can be declared at Report scope by using the above. The attribute Variable and List Variable usage is same as described above in the “Variable definition”.

Example:

```
[#Report: Balance Sheet]
   Variable:  Explode Flag
```

Explode Flag Variable is made local to Report Balance Sheet by associating it using the Report attribute ‘Variable’. This variable retains its value as long as we work with this Report. On exiting the Report, the variable is destroyed and the values are lost.

Function

Function (User Defined Function) also allows the variables to be declared at its scope. Function variables has lifetime till the end of the execution of the function.

Function variables can also be declared with default value. Function variables will never inherit the value from the parent context. This means that a volatile attribute on function variables has no effect. Function allows actions to change the value of the variables.

Function allows a special scope as STATIC. A static variables declared in a function are equivalent to a system variables but can be accessed only within the defined function. Their initial values are set only during the first call to the function, and later it retains the value for further calls.

Only simple or compound variables can be declared as static. List variables are not currently supported at static scope.

Syntax

```
Variable : <Variable Names>
      Or
Variable        : <Variable Names> [:<Data Type>[:<Value>]]
      Or
List Variable   : <Variable Names> [:<Data Type>[:<Value>]]
      Or
Static Variable: <Variable Names>[:<Data Type>[:<Value>]]
```

The variables can be declared at Function scope by using the above. The attribute Variable and List Variable usage is same as described above in the “Variable definition”.

Example 1:

```
[Function : FactorialOf]
   Variable : Factorial
```

The Function ‘FactorialOf’ requires variable ‘Factorial’ for calculation within the Function.

Example 2:

```
[Function: Sample Function]
   Static Variable : Sample Static Var: Number
```

The static variable ‘Sample Static Var’ retains the value between successive calls to Function ‘Sample Function’.

Inline Declaration

Variables can also be defined (with limited behavior) during declaration itself, so a separate definition would not be mandatory. These are called inline variable specification (i.e during declaration itself the variables are defined inline)

Only the DATA TYPE and the DEFAULT VALUE can be specified as the behavior for inline variables. If the DATA TYPE is specified as a variable name (i.e., not an implicit data type key words such as String, Amount etc.,) or is left blank it is treated as pre-defined variables.

- Persistence

Inline variables even at system scope cannot be persisted.

- **Declaring Inline Simple Variable**

Variable attribute allows declaring inline Simple Variable by specifying data type. Initial value to the variable can also be specified optionally.

Syntax

```
      Variable: <Variable Names> [:<Data Type>[:<Value>]]
```

Where,

<Variable Names> is a list of Simple Variables separated by comma.

<Data Type> is the data type of Simple Variable.

<Value> is the default/initial value provided for the variables and this value specification is optional.

Example:

```
[Report:  Cust Group Report]
   Variable: VarGroupName1, VarGroupName2 : String : “Sundry Debtors”
```

In this example, the Simple Variables VarGroupName1, VarGroupName2 of type string are declared in a Report hence the following separate variable definitions are not required which will help to reduce the coding complexity.

```
[Variable: VarGroupName1]
   Type : String
[Variable: VarGroupName2]
   Type : String
```

- **Declaring Inline Simple List Variable**

List Variable attribute allows declaring inline Simple List Variable by specifying the Data Type. If the default value is specified, it is treated as the count to initialize the list with the specified elements by default.

Syntax

```
      List Variable : <Variable Names> [: <Data Type> [: <Value>]]
```

Where,
<Variable Names> is a list of Simple Variables separated by comma.
<Data Type> is the data type of Simple Variable.
<Value> it is treated as the count to initialize the list with the specified elements by default.
The no of elements can be specified only for an index based list.

Example:

```
[System: Variable]
   List Variable: VarGroupName1, VarGroupName2 : String : 10
```

In this example, the variables VarGroupName1 and VarGroupName2 of string data type are declared as inline simple list variables at system level and each variable will have 10 elements by default.

- **Declaring Inline Compound List Variable**

For Compound List Variables, ***definition is mandatory***. They cannot be declared inline.

Using Modifiers with Variables

Variable allows static modifiers such as Add/Delete/Change and Dynamic modifier ‘Local’.

Static Modification

Add / Delete / change modifiers can be used on variables to change the behavior.

Example:

```
[#Variable: SV From Date]
   Delete: Default
```

Locally modifying variables

When different reports require the same Compound Variable and some modifications are required specific to respective reports like adding additional members (local to the report) are possible through the Dynamic Modifier 'Local'.

Example:

In our example, we have defined a Compound Variable CLVEMP as below.

```
[Variable: CLV Emp]
   Variable     : Name          : String
   Variable     : Designation   : String
   Variable     : Age           : Number
   Variable     : Salary        : Amount
   List Variable: Contact Nos   : String
   List Variable: Relatives
   Variable: Contact Address

;; Defining Compound List Variable
[Variable: Relatives]
   Variable: Name    : String
   Variable: Age      : Number
   Variable: Relation: String
   Variable: Salary   : Amount

;;Defining another compound variable
[Variable: Contact Address]
   Variable:  Street Name:  String
   Variable:  City Name   : String
```

In 'Employee Report1', the variable is declared and no modifications are required locally.

```
[Report: Employee Report1]
   Variable : CLV EMP
```

In 'Employee Report2', the same variable is declared but locally one member variable is added and one existing member variable is deleted.

```
[Report: Employee Report2]
   Variable: CLV EMP
   Local : Variable : CLV EMP : Add   : Variable : Qualification : String
   Local : Variable : CLV EMP : Delete: Variable : Age
```

Also member variables can be localized within a compound variable. This provides the ability to re-use a compound structure defined earlier and do any local modifications as required.

Example:

```
[Variable: CLVEMP]
   Variable: Contact Address
   Local   : Variable : Contact Address : Add : Variable : State : String
```

List Variable Manipulations

Simple and Compound List variables support various data manipulation operations such as Adding / Deleting / Expanding List elements, Value Specifications, Retrieving values from the list elements, Searching and Sorting, Populating List Variable from a Collection, etc. New Actions and Functions specific to List Variables are introduced for these manipulations. Before look in to these manipulations, let us understand the concept of Key, Index and Variable Path Specificaton using Dotted Notation Syntax.

Concept

1. **Key**

List variables can hold multiple values of the variable type using a string based 'Key' specification. 'Key' is of type String by default. We can specify a different datatype for a key only in scenarios where we require key based sorting. It is optional to specify the key value while adding values to the list variable. The TDL Programmer has to explicitly specify the key value. Key is unique for all elements in the list. If an element is added with duplicate key, then the existing element is over written.

It is advisable to use a key only when we require frequent access to the elements of the list based on a key.

2. **Index**

An element of the list can be accessed via 'Index'. Index of an element is the location/position of the variable from the first element in the current sorting order. Even if we have specified keys for elements of a list, index is generated internally. It is always possible to access each element in the list by specifying the index within square brackets [ ] in the dotted notation syntax. This is explained below. Index can be negative as well. In that case it is possible to access the elements in the reverse order of entry.

Variable Path Specification using Dotted Notation Syntax

We aware that in Tally.ERP 9, method value of any object including its sub-collections to any level can be accessed or modified with dotted notation syntax. The behavior of symbol prefix $ was enhanced to access the method value of any object and an action MODIFY OBJECT was introduced to modify multiple values of any object.

Compound Variables allows us to store values of different data types. A member variable can be a single instance or a list variable. A member variable can be a compound variable and can have members again, and therefore any hierarchy can be created. In short, it is similar to a Data Object.

Hence all the attributes and actions which operate Variable are now enhanced to take extended variable path syntax. i.e.,variable path can be specified using dotted notation syntax. The syntax can be used to fetch any value from any member within the hierarchy. This syntax is applicable wherever we need to specify either the variable identifier or access the value of the variable. In case of value access the operator ## is used. Value access using operator ## is discussed in detail in the topic **Index Based Retrieval using ## Operator**.

Syntax

```
<Element Variable Specification>.<Member Variable Specification>.+
<Simple Member Value specification>
```

Where,
<Element Variable Specification> can be a Compound Variable or Compound List Variable [Index Expression].
<Member Variable Specification> can be a Compound Variable Member or Compound List Member Variable [Index Expression].
<Simple Member Value Specification> refers to the name of the simple member in the specified path.
<Index Expression> can be an expression that evaluates to number. Suffixing a variable with index refers to an Element Variable. This can be positive or negative. Negative index denotes reverse access.

Example:

The below compound variable is defined:-

```
[Variable: CLV Emp]
   Variable      : Name   : String
   Variable      : Age    : Number
   Variable      : Salary: Amount
   List Variable: Relatives


[Variable: Relatives]
   Variable: Name     : String
   Variable: Age      : Number
   Variable: Relation: String
   Variable: Salary   : Amount
```

The same is declared at the System Scope, hence the same can be accessed anywhere in the system.

```
[System: Variable]
   List Variable: CLV Emp
```

Example 1:

Suppose, we want to set the value of a simple variable ‘Employee Name’ which is declared at Report Level

```
[Report: Employee Report]
   Variable: Employee Name   : String
   SET     : Employee Name   : ##CLVEMP[1].Name
```

The variable Employee Name will be set with the value of member “Name” of the first element of the Compound List Variable “CLVEMP”.

Example 2:

In case I need to display the age of first relative of the second employee I would use the following statement in a field in a report.

```
[Field: RelAge]
   Set As : ##CLVEMP[2].Relatives[1].age
```

We will discuss in detail regarding the value specification attributes and actions with the enhanced variable path specification in the further topics.

List Variable Manipulations – A Detailed Look

Let us have a detailed look on List Variable manipulations with examples:-

1. **Adding / Deleting / Expanding Elements**

- **Adding Elements to the List Variable**
  - Action LIST ADD

The Action LIST ADD is used on a list variable to add an element to the list variable based on KEY. This is mandatory before we set value into the element. KEY is compulsory in this case.

Key is unique for all elements in the list. If an element is added with duplicate key, then the existing element is over written.

Syntax

```
      LIST ADD : <List Variable Specification> : <Key Formula> [:<Value For +
                 mula>[:<Member Specification>]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be an expression which evaluates to unique string value.

<Value Formula> can be an expression which returns a value. It sets the initial value of the element variable and it is optional.

<Member Specification> this specification is required only if the value needs to be added to a specific member of a Compound List Variable. If member specification is not provided the first member variable is considered for the value.

The actions LIST APPEND & LIST SET are alias for the action LIST ADD.

To add multiple values dynamically to the List variable, we can use LIST ADD within a looping construct like While…Walk Collection etc.

Example:

Adding elements to Simple List Variable

1. Adding an element to Simple List Variable SLV Emp with a Key

```
LIST ADD: SLV Emp: "E001"
```

2. Adding an element to Simple List Variable SLV Emp with a Key and a value

```
LIST ADD : SLV Emp : "E001" : "Kumar"
```

3. Adding an element to Simple List Variable SLV Emp with a Key and a value and subsequently overriding a value corresponding to a particular key

```
LIST ADD : SLV Emp : "E001" : "Kumar"
LIST SET : SLV Emp : "E001" : "Keshav"
```

Where the value corresponding to a Key 'E001' is changed to Keshav

Adding Elements to Compound List Variable

A Compound Variable CLV Emp is defined below which stores the employee details such as Name, Age, Salary and the details of the Relatives.

```
[Variable: CLV Emp]
;;simple member variable
   Variable: Name   : String
;;simple member variable
   Variable: Age    : Number
;;simple member variable
   Variable: Salary: Amount
;;compound list member variable
   List Variable: Relatives
```

```
;;Compound Variable is defined here
[Variable: Relatives]
   Variable: Name     : String
   Variable: Age      : Number
   Variable: Relation: String
   Variable: Salary   : Amount
```

The same is declared at the System Scope, hence the same can be accessed anywhere in the system.

```
[System: Variable]
   List Variable: CLV Emp
```

1. Adding an element to Compound List Variable CLV Emp with a Key

```
   LIST ADD    : CLVEmp: "E001"
```

2. Adding an element to Compound List Variable CLV Emp with a Key and a Value

```
   LIST ADD : CLVEmp: "E001":  "Kumar"
```

In this example, since the member specification is not provided the first member variable is considered for the value.

3. Adding an element to Compound List Variable CLV Emp with a Key and a value with member specification

```
   LIST ADD : CLVEmp: "E001": 25 : Age
```

In this example, we have provided the member specification, hence the member variable Age is considered for the value.

4. Adding an element to the Compound List Member of a Compound List Variable with a Key and a value with member specification

```
   LIST ADD: CLVEmp[1].Relatives: "R001": "Prem" : Name
```

In this example, we are adding an element to the Compound List Variable "Relatives" and the member variable 'Name' is considered for the value. 'Relatives' is a Compound List Member variable of the Compound List Variable CLVEMP.

The values are hard coded in the examples for explanation purpose. The above Simple & Compound List Variable examples are used to explain further list variable manipulations.

- Action LIST ADD EX

The action LIST ADD EX is used on a list variable to add an element to the list variable without **KEY**.

Syntax

```
      LIST ADD EX : <List Variable Specification> [:<Value Formula> +
                     [:<Member Specification>]]
```

Where,

< List Variable Specification> is the Simple List / Compound List Variable specification.

<Key Formula> can be an expression which evaluates to a unique string value.

<Value Formula> can be an expression which returns a value. It sets the initial value of the element variable and it is optional.

<Member Specification> this specification is required only if the value needs to be added to a specific member of a Compound List Variable. If member specification is not provided the first member variable is considered for the value.

*The action **LIST APPENDEX** is an alias for the action **LIST ADDEX**.*

Adding elements to Simple List Variable

1. Adding an element to Simple List Variable SLV Emp

```
LIST ADD EX: SLV Emp
```

2. Adding an element to Simple List Variable SLV Emp with Value

```
LIST ADD EX: SLV Emp : “Kumar”
```

Adding elements to Compound List Variable

1. Adding an element to Compound List Variable CLV Emp

```
LIST ADD EX: CLV Emp
```

2. Adding an element to Compound List Variable CLV Emp with value

```
LIST ADD EX: CLV Emp : “Kumar”
```

In this example, since the member specification is not provided the first member variable is considered for the value.

3. Adding an element to Compound List Variable CLV Emp with value and member specification

```
LIST ADDEX: CLV Emp: 25: Age
```

In this example, we have provided the member specification, hence the member variable ‘Age’ is considered for the value.

4. Adding an element to the Compound List Member variable of a Compound List Variable with value and member specification

```
LIST ADDEX: CLVEmp[1].Relatives: “Prem” : Name
```

In this example, we are adding an element to the Compound List Variable “Relatives” and the member variable ‘Name’ is considered for the value. ‘Relatives’ is a Compound List Member variable of the Compound List Variable CLVEMP.

- **Deleting Elements from the List Variable**
  - Action LIST DELETE

The Action LIST DELETE is used to delete an element from the list based on Key. Key formula is optional, if not specified the all the elements in the list are deleted.

Syntax

```
    LIST DELETE : < List Variable Specification > [ : <Key Formula>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be an expression which evaluates to unique string value. It is optional.

Notes

*The Action **LIST REMOVE** is an alias for the action **LIST DELETE**.*

Example:

Deleting elements from Simple List Variable

Deleting a single element from a Simple List Variable

```
LIST DELETE : SLV Emp:  “E001”
```

The element identified by key ‘E001’ will be deleted from the Simple List Variable “SLV Emp” on execution of the action.

Deleting all elements from a Simple List Variable

```
LIST DELETE : SLV Emp
```

Since the key formula is not specified, all the elements from a Simple List Variable “SLV Emp” will be deleted on execution of the action.

Deleting elements from a Compound List Variable

Deleting elements from a Compound List Variable is similar to deleting elements from Simple List Variable.

Deleting an element from a Compound List Variable

```
    LIST DELETE : CLV Emp : "E001"
```

The element identified by key **'E001'** will be deleted from the Compound List Variable "CLV Emp" on execution of the action.

Deleting all elements from a Compound List Variable

```
    LIST DELETE : CLV Emp
```

Since the key formula is not specified, all the elements from the Compound List Variable "CLV Emp" will be deleted on execution of the action.

- Action LIST DELETE EX

The Action LIST DELETE is used to delete an element from the list based on index. INDEX formula is optional, if not specified then all the elements in the list are deleted.

A negative index denotes reverse position.

Syntax

```
      LIST DELETE EX : < List Variable Specification > [ : <Index Formula>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Index Formula> can be an expression which evaluates to an index number. It is optional.

Notes

LIST REMOVE EX *is the alias for the action* ***LIST DELETE EX****.*

Example:

Deleting Elements from Simple List Variable

Deleting a single element from a Simple List Variable

```
    LIST DELETE EX  : SLVEmp:  2
```

The element identified by the index number '**2**' will be deleted from the Simple List Variable "SLV Emp" on execution of the action.

Deleting all elements from a Simple List Variable

```
    LIST DELETE EX : SLVEmp
```

Since the index formula is not specified, all the elements from a Simple List Variable "SLV Emp" will be deleted on execution of the action.

Deleting elements from a Compound List Variable

Deleting elements from a Compound List Variable is similar to deleting elements from Simple List Variable.

Deleting an element from a Compound List Variable

```
   LIST DELETE EX : CLVEmp : 10
```

The element identified by the index '10' will be deleted from the Compound List Variable "CLV Emp" on execution of the action.

Deleting all elements from a Compound List Variable

```
   LIST DELETE EX : CLVEMP
```

Since the index formula is not specified, all the elements from the Compound List Variable "CLV EMP" will be deleted on execution of the action.

- **Expanding Elements in the List Variable**
  - Action LIST EXPAND

The Action LIST EXPAND is used to create specified number of blank elements and insert into the list. All these elements are created without a key. If key specification is required for each element, then either a LIST FILL or a loop can be used to add elements individually.

Syntax

```
      LIST EXPAND : <List Variable Specification> : <Count Formula>
```

Where,
<List Variable Specification> is the Simple List or Compound List variable specification.
<Count Formula> can be an expression which evaluates to a number.

Example:

Expanding Simple List Variable

```
   LIST EXPAND : SLVEMP : 10
```

In this example, the count formula is 10. Hence, 10 blank elements are added to the Simple List Variable 'SLVEMP'.

Expanding Compound List Variable

```
   LIST EXPAND : CLVEMP : 5
```

In this example, the count formula is 5. Hence, 5 blank elements are added to the Compound List Variable 'CLVEMP'.

```
LIST EXPAND : CLVEMP[1].Relatives : 10
```

In this example, the count formula is 10. Hence, 10 blank elements are added to the Compound List Variable 'Relatives'. 'Relatives' is a Compound List Member variable of the Compound List Variable 'CLVEMP'.

2. Value Specifications

The value for the Simple/List Variables (Simple & Compound) can be specified using the **Attributes** at Report and Form Level and using **Actions** at User Defined Functions.

Report Level

The attributes SET and PRINTSET are used to specify the variable values at Report Level.

- Attribute "SET"

The Report attribute SET can be used to specify a variable name and its value, which will be set during the startup of the report.

Syntax

```
SET : <Variable Specification> :<Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be an expression which evaluates to a value to the variable of the specified data type.

Example:

```
;; Setting value to Simple Variable
   SET: Var: "ABC"
;; Setting value to Simple List Variable element
   SET: ListVar[1]: "XYZ"
;; Setting value to Compound List Variable element's member
   SET: CLVEMP[1].Name: "Kumar"
```

- Attribute "PRINT SET"

The Report attribute **Print Set** is similar to SET attribute but sets the value of the variables to specified value when the report is started in print mode.

Syntax

```
PRINT SET : <Variable Specification> : <Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be an expression which evaluates to a value to the variable of the specified data type.

Example:

```
;; Setting value to Simple Variable
   PRINTSET: Var: "ABC"
```

;; Setting value to Simple List Variable element

```
   PRINTSET: ListVar[1]: "XYZ"
```

;; Setting value to Compound List Variable element's member

```
   PRINTSET: CLVEMP[1].Name: "Kumar"
```

- **Form Level**
  - Attribute "SET"

The Form attribute 'SET' is similar to Report SET attribute, while report sets the value once in its life time, the form SET is executed during every regeneration / refresh of the report.

Syntax

```
      SET : <Variable Specification> :<Value Expression>
```

Where,

<Variable Specification> is the variable path specification

<Value Expression> can be an expression which evaluates to a value to the variable of the specified data type.

Example:

;; Setting value to Simple Variable

```
   SET: Var: "ABC"
```

;; Setting value to Simple List Variable element

```
   SET: ListVar[1]: "XYZ"
```

;; Setting value to Compound List Variable element's member

```
   SET: CLVEMP[1].Name: "Kumar"
```

- **Function Level**

The actions SET, MULTISET, INCREMENT,DECREMENT and EXCHANGE are used to specify the variable values.

  - Action SET

Values of variables can be set / updated via the SET action. This action is available as a global action, and can be used within a function also.

List variables and compound variables cannot have values; they can have only element / member variables inside it respectively.

If SET action is used on compound variables the value will be set to the FIRST member variable. If the first member variable is again compound program would search first non-compound leaf member and set the value.

For list variables the value is treated as the count, and the list is expanded by the number of elements provided in the expression.

Syntax

```
      SET : <Variable Specification> :<Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be an expression which evaluates to a value to the variable of the specified data type.

Example:

```
;; Setting value to Simple Variable
   SET: Var: "ABC"
;; Setting value to Simple List Variable element
   SET: SLVEMP[1]: "XYZ"
;; Setting value to Compound List Variable element's member
   SET: CLVEMP[1].Name: "Kumar"
```

- Action MULTISET

The action MULTI SET is used to set values of compound member variables in one call.
All member specifications are relative from the compound variable specification given.

Syntax

```
      MULTI SET : <CompoundVariable Specification> + : <Member Specification :
                  Value> [, <Member Specification : Value>, …]
```

Where,
<Compound Variable Specification> is the Compound Variable specification.
<Member Specification : Value> list of name-value pair for setting member values .

Example:

```
   MULTISET : CLVEMP[1]:Name:"Vimal",Age:26,Salary:($$AsAmount:10000)
```

In this example, all the member variables for the first element of the Compound List Variable **CLVEMP** are set using the MULTISET Action.

```
   MULTISET : CLVEMP[1].Relatives[1]:Name:"Hari", Age:20, +
              Relation:"Brother"
```

In this example, all the member variables for the first element of the Compound List Variable Relatives are set using the MULTISET Action. **Relatives** is a Compound List Member variable of the Compound List Variable **CLVEMP**.

- Action EXCHANGE

The Action Exchange can be used to swap the values between two variables provided both belong to the same data type.
The same cannot be done for the Simple List or Compound List as a whole. However, value of an element of Simple List and Compound List Member Variable belonging to same data type can be exchanged

Syntax

```
      EXCHANGE : <First Variable Specification> : <Second Variable +
                 Specification>
```

Where,
<First Variable Specification> is the simple variable specification.
<Second Variable Specification> is the simple variable specification.

Example:

Exchanging value of a Simple Variable to another Simple Variable

```
EXCHANGE : EmpVarOld: EmpVarNew
```

Both the variables are of String data type. The value of the variable **EmpVarOld** is exchanged to the variable **EmpVarNew** on execution of the action.

Exchanging value of an element of Simple List Variable to the element of another Simple List Variable

```
EXCHANGE: SlvEmpOld[1] : SlvEmpNew[1]
```

The value of the first element of **SlvEmpOld** is exchanged to the first element of the **SlvEmpNew**. Both the Simple List Variables are of String data type

Exchanging value of a Simple Variable to a member variable of a Compound List Variable

```
EXCHANGE  : EMP Salary : CLVEmp[1].Salary
```

In this example, the value of a variable **Emp Salary** is exchanged to the member variable Salary of the Compound List Variable **CLVEmp**. Both the simple variables are of string data type.

- Action INCREMENT

Increment is a special action provided in function scope to increment values of the variable. This is supported only on simple variables of type number.

Syntax

```
INCREMENT : <Simple Variable Specification> [: <NumIncrement Expression>]
```

Where,
<Simple Variable Specification> is the simple variable specification.
<NumIncrement Expression> can be an expression which evalues to a number. Based on this, variable value will be incremented. It is optional. If not specified, the variable value will be incremented by 1.

*The action **INCR** is an alias for the action **INCREMENT**.*

Example:

```
;; Incrementing the variable value by 1
   INCREMENT :  Counter
;;  Incrementing the variable value by 2
   INCR      :  Counter : 2
```

  - Action DECREMENT

Decrement is a special action provided in function scope to decrement values of the variable. This is supported only on simple variables of type number.

Syntax

```
      DECREMENT :< Simple Variable Specification> [:< NumIncrementExpression>]
```

Where,

<SimpleVarSpecification> is the simple variable specification.

<NumIncrementExpr> can be an expression which evalues to a number. Based on this, variable value will be decremented. It is optional. If not specified, the variable value will be decremented by 1.

*The action **DECR** is an alias for the action **DECREMENT**.*

Example:

```
;; Decrementing the variable value by 1
   DECREMENT :  Counter
;; Decrementing the variable value by 2
   DECR      :  Counter : 2
```

- **Field Level**
  - Attribute MODIFIES

The Field attribute Modifies is used to modify the value of the variable.

Syntax

```
      Modifies : <Variable Specification>[:<Logical Flag>]
```

Where,

<Variable Specification> is the variable path specification.

<Logical Flag> can be a logical value TRUE/FALSE. TRUE would set the value after the field's acceptance, and FALSE will set during the acceptance of the report having the field.

Example:

```
[Field: EMP Age]
   Modifies : EMPAgeVar : Yes
```

In this example, the value of the variable EMPAgeVar will be modified with the value stored/keyed in the field EMP Age after field's acceptance.

3. Retrieving value from List

- **$$ListValue**

The function ListValue is used to retrieve value of an element in the list for a given key. If the list is of compound variables an optional member specification can be provided to extract value of a specific member.

Syntax

```
$$ListValue:<List Variable Specification>:<Key Formula> +
                [:<Member Specification>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be an expression which evaluates to a string value.

<Member Specification> member specification is required only if the value needs to be extracted from a specific member of a Compound List Variable.

Example:

Retrieving value from Simple List Variable

```
$$ListValue:SLVEMP: "E001"
```

In this example, the function returns the value of the element identified by the key ‘E001’ from the simple list variable ‘SLV Emp’.

```
$$ListValue:SLVEMP:##KeyVar
```

In this example, the variable ‘KeyVar’ holds the key value. The function returns the value of the element identified by the key from the simple list variable ‘SLV Emp’.

Retrieving value from Compound List Variable

```
$$ListValue:CLVEmp:##KeyVar:Age
```

In this example, the variable ‘KeyVar’ holds the key value. The function returns the identified Compound List Variable element's member variable value. In our case, we have specified the member specification as ‘Age’.

- **$$ListValueEx**

The Function $$ListValueEx returns the value of an element at the specified index in the list.

Syntax

```
$$ListValueEx : <List Variable Specification>:<Index Formula> +
                    [:< Member Specification>]
```

Where,

<List Variable Specification> is the Simple or Compound List Variable specification.

<Index Formula> can be an expression which evaluates to an index number.

<Member Specification> this specification is required only if the value needs to be extracted from a specific member of a Compound List Variable.

Example:

Retrieving value from Simple List Variable

```
$$ListValueEx:SLVEmp:##IndexVar
```

In this example, the variable 'IndexVar' holds the index value. The function returns the value of the element identified by the index from the simple list variable 'SLV Emp'.

Retrieving value from Compound List Variable

```
$$ListValueEx:CLVEmp:##IndexVar:Age
```

In this example, the variable 'KeyVar' holds the index value. The function returns the identified Compound List Variable element's member variable value. In our case, we have specified the member specification as 'Age'.

◘ Index Based Retrieval using ## Operator

The operator ## is used to access the value of the variable. This operator is extended to allow dotted notation syntax to access variables / member variables / element variables of a list at any level

When this operator is used on a compound variable(without path specification), it returns the value of the first member variable by default. Similarly, on a list variable this returns the number of items in the list.

Syntax

```
##<Element Variable Specification>.<Member Variable Specification>.+
        < Simple Member Value specification>
```

Where,

<Element Variable Specification> can be a Compound Variable or Compound List Variable [Index Expression].

<Member Variable Specification> can be a Compound Variable Member or Compound List Member Variable [Index Expression].

<Simple Member Value Specification> refers to the name of the simple member in the specified path.

<Index Expression> can be an expression that evaluates to number. Suffixing a variable with index refers to an Element Variable. This can be positive or negative. Negative index denotes reverse access.

Example:

Retrieving Value from Simple List Variable

```
SET : TempVar :  ##SLVEMP[3]
```

The value of the element identified by the index '3' from the Simple List Variable 'SLVEMP' will be set to the variable 'TempVar'.

Retrieving Value from Compound List Variable

```
LOG: ##CLVEmp[2].Relatives[1].Name
```

In this example, we are retrieving value of the identified Compound List Variable (Relatives) element's member variable value. 'Relatives' is a member variable of the Compound List Variable CLVEMP.

4. Looping Construct – For In/For Each

The FOR IN loop is used to iterate over the values in the list variable. The number of iterations depends on the number items in the list variable.

Syntax

```
FOR IN : <Iterator Variable>: <List Variable Name >
        .
        .
END FOR
```

Where,

<Iterator Variable> is the name of the variable which holds the Key value in every occurrence of the iteration.

<List Variable Name> is the name of Simple List or Compound List Variable.

This construct will walk only the elements in the list which are having key. Since, the iterator variable is filled with a key for each element; all elements which do not have key are ignored.

This is useful to walk keyed list variable elements in the current sorting order. If the element does not have key then other loops like WHILE, FOR, etc can be used and these elements can be operated via index.

Example:

Iterating the Simple List Variable Values

```
FOR IN : KeyVar : SLV Emp
  LOG : $$ListValue:SLVEmp:##KeyVar
END FOR
```

In this example, the iterator variable "KeyVar" holds the Key value in every occurrence of the iteration. In every iteration, the value of the element identified by the key is logged using the function $$ListValue.

Iterating the Compound List Variable Values

```
FOR IN : KeyVar: CLV Emp
  LOG : $$ListValue:CLVEmp:##KeyVar:Age
END FOR
```

In this example, the iterator variable "KeyVar" holds the Key value in every occurrence of the iteration. In every iteration, the value of member "Age" of the element of the Compound List Variable "CLVEMP" identified by key is logged using the function $$ListValue.

*The looping construct **FOR EACH** is an alias for the looping construct **FOR IN**.*

5. List Variable Specific Functions

- **$$ListKey**

The function $$ListKey returns the corresponding key for the given index.

Syntax

```
$$ListKey: <List Variable Specification>: <Index Specification>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Index Specification> can be an expression which evaluates to a number.

Example:

Retrieving key from a Simple List Variable

```
01 : LOG : $$ListKey:SLVEMP:2
```

In this example, Key of the second element of Simple List Variable ‘SLVEMP’ is retrieved.

Retrieving key from a Compound List Variable

```
02 : LOG : $$ListKey:CLVEmp[1].Relatives:1
```

In this example, Key of the first element of Compound List Variable ‘Relatives’ is retrieved. ‘Relatives’ is a member of Compound List Variable 'CLVEMP'.

- **$$ListIndex**

The function **$$ListIndex** returns the Corresponding index for the given Key

Syntax

```
$$ListIndex: < List Variable Specification >: < Key Specification >
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Specification> can be an expression which evaluates to string value.

Example:

Retrieving index from a Simple List Variable

```
01 : LOG : $$ListIndex:SLVEMP:E001
```

In this example, the index of the element identified by the key value ‘E001’ is retrieved from the Simple List Variable ‘SLVEMP’.

Retrieving index from a Compound List Variable

```
02: LOG : $$ListIndex:CLVEmp:E001
```

In this example, the index value of the element identified by the key value ‘E001’ is retrieved from the Compound List Variable ‘CLVEMP’.

- ### $$ListCount

The function $$ListCount retrieves the number of items in the list.

Syntax

```
$$ListCount:<List Variable Specification>
```

Where,

<ListVariable Specification> is the Simple List or Compound List Variable specification.

Example:

```
01 : LOG : $$ListCount:SLVEMP
02 : LOG : $$ListCount:CLVEMP
```

- ### $$ListFind

This function is used to check if a given key exists in the list or not. It returns a logical flag indicating the existence of the key.

Syntax

```
$$ListFind:<List Variable Specification>:<Key Formula>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be an expression which evaluates to a string value.

Example:

```
01 : LOG : $$ListFind :SLVEMP :E001
02 : LOG : $$ListFind :CLVEMP :E001
```

- ### $$ListValueFind

This function can be used to check if a given value exists in the list. If a given list has more than one same value, index can be used to retrieve the n'th matching value.

Syntax

```
$$ListValueFind:<List Variable Specification>:< Occurance +
                Specification>:<Value Formula>[:<Member Specification>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Occurance Specification> can be an expression which evaluates to a number.

<Value Formula> can be an expression which evaluates to a value.

<Member Specification> can be specified if the list element is compound. It is optional.

Example:

```
;; Finding value from the Simple List Variable
   01 : LOG : $$ListValueFind :SLVEMP:1:RAMESH
;;Finding value from the Compound List Variable with member specification.
   03 : LOG : $$ListValueFind :CLVEmp:1:PRIYA:Name
```

The function will return Yes if the value exists in the list else it will return No.

6. Some Common Functions Used

- **$$IsSysNameVar**

This function checks if the variable has a value which is a SysName like Not Applicable, End of List, etc. In case of repeated variables if any one value is a non-sysname returns FALSE.

Syntax

```
$$IsSysNameVar:<Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
   $$IsSysNameVar:EmpVar
```

In this example, the function $$IsSysNameVar returns logical value YES if the variable **EmpVar** has Sysname as value else it returns NO.

- **$$IsDefaultVar**

This function determines that the content of the variable has a “Default” or blank as the value. This function is applicable only for Simple variables. In case of simple repeated variable, if any one value is non-default, then this is not a default variable.

Syntax

```
$$IsDefaultVar:<Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Field: DefaultVar]
   Set as : $$IsDefaultVar:SVValuationMethod
```

In this Example the function $$IsDefaultVar returns logical value YES if value of variable “SVValuationMethod” is blank or “Default” else it returns NO.

- $$IsActualsVar

This function checks if the content of the variable is blank or sysname or “ACTUALS”.

Syntax

```
$$IsActualsVar:<Variable Specification>
```

Where,
<Variable Specification> is the simple variable path specification.

Example:

```
$$IsActualsVar:SVBudget
```

In this example, the function $$IsActualsVar returns logical value YES if the value of variable SVBudget is blank or sysname or “ACTUALS” else it returns NO.

$$IsCurrentVar

This function checks if the content of the variable is Blank or Sysname or “Stock in hand”.

Syntax

```
$$IsCurrentVar : <Variable Specification>
```

Where,
<Variable Specification> is the simple variable path specification.

Example:

```
$$IsCurrentVar:DSPOrderCombo
```

In this Example, the function $$IsCurrentVar returns logical value “YES” if value of variable “DSPOrderCombo” is blank or sysname or “Stock-In-Hand” Else it returns “No”.

$$ExecVar

This function returns the value of a variable in the parent report chain.

Syntax

```
$$ExecVar:<Variable Specification>
```

Where,
<Variable Specification> is the simple variable path specification.

Example:

```
$$ExecVar:DSPShowMonthly
```

In this example, the function $$ExecVar returns the value of the variable DSPShowMonthly from the parent report.

$$FieldVar

This function returns the value of the field which is acting as a variable with the specified name.

Syntax

```
$$FieldVar:<Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Collection: GodownChildOfGodownName]
   Type        : Godown
   Child of    : $$FieldVar:DSPGodownName
```

In this example, $$FieldVar is used to fetch the value of the variable DSPGodownName whose value is modified in a field. This value becomes the value for the ChildOf attribute.

- **$$ParentFieldVar**

This function gets the field variable value from its parent report.

Syntax

```
      $$ParentFieldVar:<Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Field: ParentFieldVar]
   Set as     : $$ParentFieldVar:SVStockItem
```

In the above Example the function returns field variable value from its parent report for the variable "SVStockItem".

7. Populating List from a Collection

The action LIST FILL is used to fill a list from a collection instead of using the looping constructs. The specified collection is walked and the key formula and value formula is evaluated in the context of each object to create list elements.

Syntax

```
      LIST FILL : <List Variable Specification> : <CollectionName> +
                  [:<Key Formula> [:<Value Formula> [:<Member Specification>]]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Collection Name> is the name of collection from which the values needs to be fetched to fill the list variable.

<Key Formula> can be an expression which evaluates to string value. It is optional.

<Value Formula> can be an expression which returns a value. The data type of the value must be same as that of List variable. Value formula is optional if not specified only KEY is set for each added element.

<Member Specification> Member specification can be given if the list contains a compound variable.

If both key and value are not specified blank elements are added to the list.

Example:

Populating Simple List Variable from a Collection

```
LIST FILL : SLV Emp : Employees :$Name : $Name
```

All the employee names from the collection ‘Employees’ will be added to the Simple List Variable once the action LIST Fill is executed.

Populating Compound List Variable from a Collection

```
LIST FILL : CLV Emp : Employees :$Name : $Name
```

In this example, all the employee names from the collection ‘Employees’ will be added to the first member variable as there is no member specification.

```
LIST FILL : CLV Emp : Employees :$Name: $Designation:Designation
```

In this example, Designations of all the employees from the collection ‘Employees’ will be added to the member variable ‘Designation’.

```
LIST FILL: CLV EMP[1].Relatives:Employees:$Name:$SpouseName:Name
```

In this example, spouse name of all the employees from the collection ‘Employees’ will be added to the member variable ‘Name’ of a Compound List Variable ‘Relatives’. ‘Relatives’ is a member variable of the Compound List Variable ‘CLVEMP’.

8. Sorting of List Elements

Initially when the list variable is created, it is sorted on the order of insertion. TDL provides the facility to sort the values in the list variable either on key or value. Following actions allows changing the sort order:-

- List Key Sort
- List Value Sort
- List Reset Sort

Action LIST KEY SORT

The action LIST KEY SORT allows the user to sort the elements of the list based on the key.

Keys are by default of type string, so absence of key data type specification will consider key data type as string while sorting. The user can override this by specifying a key data type.

Keys are optional for elements. All elements in the list may not have a key. In such cases, comparisons of elements would be done on the insertion order.

Syntax

```
LIST KEY SORT : <List Variable Specification>[:<Ascending/DescendingFlag>
                [:<Key Datatype>]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Ascending/DescendingFlag> can be YES/NO. YES is used to sort the list in ascending order and NO for descending. If the flag is not specified then the default order is ascending.

<Key Data Type> can be String, Number etc. Its optional.

*The action **LIST SORT** is an alias for the action **LIST KEY SORT***

Example:

Sorting Simple List based on Key

```
;;Ascending order
   LIST KEY SORT: SLVEmp: Yes : String
;;Decending Order
   LIST KEY SORT: SLVEmp: No  : String
```

Sorting Compound List based on Key

```
;;Ascending order
   LIST KEY SORT: CLVEmp: Yes : String
;;Decending order
   LIST KEY SORT: CLVEmp[1].Relatives : No : String
```

◻ Action LIST VALUE SORT

The action LIST VALUE SORT allows the user to sort the elements of the list based on the value.

The data are sorted as per the data type specified for the list variable in case of simple list, and the member specification data type in case of compound list. If a compound list is chosen and member specification is not specified, then the list is sorted by the value of the first member variable.

If duplicate values are in the list, the key data type passed is considered to sort by key, and then in absence of key, insertion order is used.

Syntax

```
      LIST VALUE SORT :<List Variable Specification>[:<Ascending/Descending +
                       Flag> [:<Key Datatype> [:<Member Specification>]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Ascending/DescendingFlag> can be YES/NO. YES is used to sort the list in ascending order and NO for descending. If the flag is not specified then the default order is ascending.

<Key Data Type> can be String, Number etc. It’s optional.

<Member Specification> is the member specification incase of compound list. If not specified then the list is sorted by the value of first member variable.

Example:

Sorting Simple List based on Value

```
;;Ascending Order
   LIST VALUE SORT: SLVEmp: Yes : String
;;Decending Order
   LIST VALUE SORT: SLVEmp: No  : String
```

Sorting Compound List based on Value

```
;;Ascending Order
   LIST VALUE SORT: CLVEmp: Yes : String
;;Decending order
   LIST VALUE SORT: CLVEmp[1].Relatives: No : String
```

- **Action LIST RESET SORT**

The action **LIST RESET SORT** resets the sorting method of the list and brings it back to the insertion order.

Syntax

```
LIST RESET SORT: <List Variable Specification>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

Example:

```
   LIST RESET SORT: SLVEMP
   LIST RESET SORT: CLVEMP
```

Field Acting as a Variable

The Variable attribute in a Field Definition is used to make the field behave as a variable with the specified name. The variable need not be defined as it inherits data type from the field itself. Field can act as simple variable only since it can hold only simple value.

Syntax

```
[Field : <Field Name>]
    Variable   :  <Variable Name>
```

Where,

<Field Name> is the name of the field.

<Variable Name> is the name of variable.

Example:

```
[Field: EmployeeName]
   Variable : EmpNameVar
```

Implication of Repeat Variables in Columnar Report

The report in which the number of columns added or deleted as per the user inputs is referred to as Columnar Report. In a Columnar Report, Lines are repeated vertically and Fields are repeated horizontally. The Columnar Report can be a

- **MultiColumn Report** - Column can be repeated based on the user input
- **AutoColumn Report** - Multiple columns can be repeated based on the user input on a single click of a button
- **Automatic Auto Columns** - Report can be started with predefined multiple columns without user intervention

The Attribute Repeat – Variable, Report and Line

Let us see the implications of Repeat Attribute of Variable / Report / Line Definitions in context of Columnar Reports.

1. Repeat Attribute of Variable Definition

Please refer to the topic “Variable Definition and Attributes”.

2. Repeat Attribute of Report Definition

The Repeat Attribute of Report Definition is used specifically in Columnar Reports. When we specify Repeat attribute with a variable name, the reports becomes a Columnar Report and the number of columns depend upon the values stored in the variable. Only simple variables can be repeated. Also, a report can have more than one variables repeated. In such cases, the number of columns in the report depends on the maximum value a Repeat Variable holds.

The Repeat attribute of the report is declaration cum repeat specification, so a separate declaration is not required. Even if a declaration is done using a Variable attribute Repeat is considered as a repeat specification.

Syntax

```
[Report: <Report Name>]
    Repeat : <Variable Names>
```

Where,

<Report Name> is the name of the Report.

<Variable Names> is the list of comma separated variables.

3. Repeat Attribute of Line Definition

The Repeat Attribute of Line Definition is used to repeat the Field horizontally in columns.

Syntax

```
[Line: <Line Name>]
    Repeat : <Field Name>
```

Where,

<Line Name> is the name of the line.

<Field Name> is the name of the field which needs to be repeated.

Example:

Let us look in to the usage of Repeat Attribute at Variable / Report / Line Definitions in designing the Columnar Stock Item wise Customer wise Sales Report

In this report, Stock Item names should be repeated vertically and Customer/Party names should be repeated horizontally. The columns should be automatically available when the report is started.

Repeat Attribute of Variable Definition

```
[Variable: PName]

   Type   : String

   Repeat: ##DSPRepeatCollection
```

The variable 'DSPRepeatCollection' holds the collection name 'CFBK Party'. The collection definition is as below. This collection contains a method name 'PName'. In this case, the variable 'PName' would be filled with the method value from each object of the collection "CFBK Party".

```
[Collection: CFBK Party]

   Source Collection   : CFBK Voucher

   Walk                : Inventory Entries

   By                  : PName      : $PartyLedgerName

   Aggr Compute        : BilledQty : SUM: $BilledQty

   Filter              : NonEmptyQty
```

The variable 'PName' holds the multiple values based on an implicit index. Method value of the each object of the collection **'CFBK Party'** will be picked up and stored in to the variable's first index, second index and so on.

Repeat Attribute of Report Definition

```
[Report: CFBK Rep]

   Use          : DSP Template

   Form         : CFBK Rep

   Variable     : DoSetAutoColumn, PName

   Repeat       : PName

   Set          : DoSetAutoColumn        : Yes

   Set          : DSPRepeatCollection    : "CFBK Party"

   Set          : SVFromDate             : $$MonthStart:##SVCurrentDate

   Set          : SVToDate               : $$MonthEnd:##SVCurrentDate
```

The attribute "Repeat" determines that it is a Columnar Report. The number of columns depends on the number of values available in the variable "PName".

Repeat Attribute of Line Definition

```
[Line: CFBK Rep Details]
   Fields : CFBK Rep Name, CFBK Rep Party, CFBK Rep Col Total
   Repeat : CFBK Rep Party
   Total  : CFBK Rep Party
```

The Field **'CFBK Rep Party'** is repeated based on the number of values of variable (NumSets). So those many numbers of instance of the field are created. Each field will have an implicit index number (starting from 1). This implicit index is used to evaluate expressions in the context of the field.

Common Functions used with Columnar Reports

1. **$$NumSets**

This function returns the number of columns in the report. It does not take any parameter. If the report is an auto report or sub report, it returns the number of columns in the parent of the auto/ sub report.

Number of set is the maximum number of values a repeated variable can hold in that report.

Syntax

```
$$NumSets
```

Example:

```
[Field: CFBK Rep Col Total]
   Use       : Qty Primary Field
   Set as    : $$Total:CFBKRepParty
   Border    : Thin Left
   Invisible: $$Numsets=1
```

In this example, the total column will be invisible if there is only one column in the report.

2. **$$LowValue**

This function can be used to get the lowest value in a set of values in the repeated variable.

Syntax

```
$$LowValue:<Variable Specification>
```

Where,
<Variable Specification> is a simple variable specification.

Example:

Let us suppose, the Repeat Variables in a Columnar Report are SVFromDate and SVToDate

Consider the below Field Definition in the same report:-

```
[Field: VariableLowValue]
   Use      : Name Field
   Set as   : $$LowValue:SVFromDate
```

The function $$Lowvalue returns the lowest value in a set of values in the repeat variable SVFromDate

3. $$HighValue

This function can be used to get the highest value in a set of values in the repeated variable.

Syntax

```
$$HighValue:<Variable Specification>
```

Where,
<Variable Specification> is a simple variable specification.

Example:

Let us suppose, the Repeat Variables in a Columnar Report are SVFromDate and SVToDate
Consider the below Field Definition in the same report:-

```
[Field: VariableHighValue]
      Use      : Name Field
      Set as   : $$HighValue:SVToDate
```

The function $$HighValue returns the highest value in a set of values in the repeat variable SVToDate

4. $$IsCommon

This function is used with repeated variable to check if all the values in the repeat set are same.

Syntax

```
$$IsCommon:<Variable Specification>
```

Where,
<Variable Specification> is a simple variable specification.

Example:

Let us suppose, the Repeat Variable in a columnar report is SVCurrentCompany

Consider the below Field Definition in the same report

```
[Field: VariableIsCommon]
   Use      : Logical Field
   Set as   : $$IsCommon:SVCurrentCompany
```

The function $$IsCommon returns a logical value 'Yes' if all the values in the SVCurrentCompany are same otherwise it returns 'No'

5. $$VarRangeValue

This function gets a list of variable values separated by specified separator character. If no separator character is specified, by default comma (,) is taken as separator character.

Syntax

```
$$VarRangeValue:<Variable Specification>[:<Separator Character> [:<Start
                Position> [:<End Position>]]]
```

Where,

<Variable Specification> is the simple variable specification.

<Separator Character> is the seperator character.

<Start Position> is the index which denotes the starting position.

<End Position> is the index which denotes the ending position.

Notes *Specifying Start and End Positions are optional. If not specified, the function will return all the values of the specified Repeat variable separated by comma(,)*

If Start and End Positions are specified, the function will return the values of repeat variable with in the Specified index Range. Again, specifying End Position is optional. If End Position is not specified, the function will return the entire values from the starting position.

Example:

```
$$VarRangeValue:SVFromDate
```

In this example, the function returns the entire set of values of the Repeat Variable SVFromDate.

```
$$VarRangeValue:SVFromDate:",":1:5
```

In this example, the function returns the value of the specified index range (1 to 5) of the Repeat Variable SVFromDate

```
$$VarRangeValue:SVFromDate:",":3
```

In this example, the function returns the entire set of values from the Starting Index position of the Repeat Variable SVFromDate

Variables usage and behaviour in Auto Report

A report can be marked as an auto report via **AUTO** attribute; this indicates the system that, this report cannot instantiate its own variables. These reports inherit variables from the parent scope. This is mainly used for configuration reports which require modifying the configuration variables of parent report.

Syntax

```
      [Report: <Report Name>]
          Auto : <Logical Value>
```

Where,

<Report Name> is the name of the report.

<Logical Value> can be YES / NO. Default value is No.

Example:

```
[Report: Voucher Configuration]
   Auto          : Yes
   Title         : $$LocaleString:"Voucher Configuration"
```

The above is a default configuration report which marked as an Auto Report is used to modify the variables of parent report.

A report can be launched in 'Auto' mode through the Actions **Modify Variable** and **Modify System**

Action MODIFY VARIABLE

The action MODFY VARIABLE launches the given report in 'auto' mode. Since the launched report is in auto mode, this cannot have its own instance of variables and any modification would affect the parent context.

Syntax

```
      MODIFY VARIABLE : <Report Name>
```

Where,

<Report Name> is the name of the report which is to be launched in 'Auto Mode'.

Example:

Let us look in to the below code snippet

```
[Button: F2 Change Period]
   Key           : F2
   Action        : Modify Variables : Change Period
   Title         : $$LocaleString:"Period"
```

The Action Modify Variable is launched the report Change Period in Auto Mode. The report is having two fields SVFromDate and SVToDate

```
[Field: SVFromDate]
   Use           : Short Date Field
   Modifies      : SVFromDate
   Variable      : SVFromDate
```

```
[Field: SVToDate]
   Use          : Short Date Field
   Format       : Short Date, End:#SVFromDate
   Modifies     : SVToDate
   Variable     : SVToDate
```

The varible value changes would affect the parent report context only (ie. It will affect values of the variables **SVFromDate** and **SVTodate** which are associated to the report from which the report **Change Period** is launched in Auto Mode).

Action MODIFY SYSTEM

The action **MODIFY SYSTEM** launches the given report in ‘auto’ mode. Even if this report is called under some other report context, this action makes the new report to get the system context and thereby modify system scope variables.

Syntax

```
      MODIFY SYSTEM : <Report Name>
```

Where,
<Report Name> is the name of the report which is to be launched in 'Auto Mode'.

Example:

Let us look in to the default code snippet

```
[Button: Change System Period]
   Key         : Alt+F2
   Action      : Modify System    : Change Menu Period
   Title       : $$LocaleString:“Period”
```

The Action Modify System has launched the report Change Period in Auto Mode. The report is having two fields SVFromDate and SVToDate

```
[Field: SVFromDate]
   Use          : Short Date Field
   Modifies     : SVFromDate
   Variable     : SVFromDate

[Field: SVToDate]
   Use          : Short Date Field
   Format       : Short Date, End:#SVFromDate
   Modifies     : SVToDate
   Variable     : SVToDate
```

The value changes would affect the variables at system scope as the report is launched using the Action Modify System.

Repeat Line with Optional Collection

We are already aware that the Repeat Attribute of a Part is used to Repeat a line over a Collection.

Syntax:

```
[Part: <Part Name>]
        Repeat : <Line Name> : <Collection>
```

Where,

<Part Name> is the name of the part.

<Line Name> is the name of the line to be repeated.

<Collection> is the name of the collection on which the line is repeated. This was mandatory

In this case the same line will be repeated for each object of the collection. Each line will be associated with an Object of the collection. Report created in Create/Alter/Display mode will either store method values into the object or fetch method values from the Object. Any expression evaluation within this line will happen with an object in context.

With the introduction of List Variable (Simple/Compound) there will be a requirement to store values into the Variable by accepting user inputs and also to display or use it for expression evaluation. Since Variables are Context free structures there is no need to associate element variables with the line. For this purpose the Repeat Attribute of the part is enhanced to have the collection as Optional. Now it is possible to Repeat a Line with or without a Collection. In cases where collection is not specified then the number of lines to be repeated is unknown. Hence, specifying SET attribute is mandatory. In case of Edit, SET can be optional if Break on is specified.

Syntax

```
[Part: <Part Name>]
       Repeat : <Line Name> [: <Collection>]
```

Where,

<Part Name> is the name of the part.

<Line Name> is the name of the line to be repeated.

<Collection> is the name of the collection on which the line is repeated. It is now optional to specify the collection name.

Storing Values into List Variables

With this enhancement, values can be added to the List Variable (Simple/Compound) dynamically by accepting the user inputs by repeating a line without a Collection. Multiple lines can be added dynamically or fixed number of lines can be added as per user requirement while repeating the line.

Example:

To accept the values from a user to the Simple List Variables SLVEMP, a report is opened in Create Mode. Let us look in to the Part Definition

```
[Part: SLV List Values]
   Lines  : SLV List Title, SLV List Values
   Repeat : SLV List Values
   BreakOn: $$IsEmpty:#SLVAlias
   Scroll : Vertical
```

In this example, the line is repeated without a collection and it will break if the fields value ‘SLV Alias’ is empty.

Let us look in to the Field Definitions:-

```
[Line: SLV List Values]
   Fields: SLV Alias, SLV Name

   [Field: SLV Alias]
     Use : Name Field

   [Field: SLV Name]
     Use      : Name Field
     Delete   : Key
     Add      : Key      :SLV List Key
     Inactive: $$IsEmpty:#SLVAlias

[Key:  SLV List Key]
   Key        : Enter
   Action List: Field Accept, SLV List Add

[Key: SLV List Add]
   Key    : Enter
   Action: LIST ADD:SLVEMP:#SLVAlias:#SLVName
```

Values are added to the List Variable “SLVEMP” using the Action “LIST ADD”.

Similar way, user inputs can be added / altered dynamically to the Compound List Variable also.

Retrieving Values from List Variables

In the above example, we had stored the values in to a Simple List Variable “SLVEMP”. Let us suppose, the values needs to be retrieved from the Simple List Variable “SLVEMP” and displayed in a Report.

This report “SLV List Values with Key Display” is opened in Display mode. Let us look in to the code snippet of the part definition:-

```
[Part: SLVList ValuesDisplay]
   Lines        : SLV List DisplayTitle, SLV List DisplayValues
   Repeat       : SLV List DisplayValues
   Set          : $$ListCount:SLVEmp
   Scroll       : Vertical
   CommonBorder: Yes
```

In part level, the number of lines is fixed using the Attribute ‘SET’ based on the number of elements in the Simple List Variable “SLVEmp”.

```
[Line: SLV List DisplayValues]
   Fields: SLV Alias, SLV Name

   [Field: SLV Alias]
     Use    : Name Field
     Set as : $$ListKey:SLVEMP:$$Line

   [Field: SLV Name]
     Use    : Name Field
     Set as : $$ListValue:SLVEMP:#SLVAlias
```

Key and Value from the Simple List Variable “SLVEMP” are retrieved using the functions $$ListKey and $$ListValue at the field level.

Similar way, the values can be retrieved from a Compound List Variable also.

Variables in Collection

The inline variables can be declared at the collection using the Attributes Source Var, Compute Var and Filter Var.

Incase of Simple Collection, during the evaluation only current objects are available. Where as incase of Aggregate/Summary collection during the evaluation following three sets of objects are available:

Source Objects : Objects of the collection specified in the Source Collection attribute
Current Objects : Objects of the last collection specified in the Walk path
Aggregate Objects : Objects obtained after performing the grouping and aggregation

There are scenarios where some calculation is to be evaluated based on the source object or the current object value and the filtration is done based on the value evaluated with respect to final

objects before populating the collection. In these cases to evaluate value based on the changing object context is tiresome and some times impossible as well.
The collection level variables provide Object-Context Free processing. The values of these inline variables are evaluated before populating the collection.

The sequence of evaluation of the collection attributes is changed to support the attributes ComputeVar, Source Var and Filter Var. The variables defined using attributes Source Var and ComputeVar can be referred in the collection attributes By, Aggr Compute and Compute. The variable defined by Filter Var can be referred in the collection attribute Filter.

The value of these variables can be accessed from anywhere while evaluating the current collection objects.

Source Var

The attribute Source Var evaluates the value of based on the source object.

Syntax

```
      Source Var : <Variable Name> : <Data Type> : <Formula>
```

Where,
<Variable Name> is the name of variable.
<Data Type> is the data type of the variable.
<Formula> can be an expression which evaluates to value of the variable data type.

Example:

```
   Source Var : Log Var: Logical : No
```

The value of the LogVar variable is set to NO

Compute Var

The attribute Compute Var evaluates the value of based on the current objects.

Syntax

```
      Compute Var : <Variable Name> : <Data Type> : <Formula>
```

Where,
<Variable Name> is the name of variable.
<Data Type> is the data type of the variable.
<Formula> can be an expression which evaluates to value of the variable data type.

Example:

```
   Compute Var: IName : String : if ##LogVar then $StockItemName else +
                 ##LogVar
```

Filter Var

The attribute Filter Var evaluates the value of based on the objects available in collection after the evaluation of attributes Fetch and Compute.

Syntax

```
      Filter Var : <Variable Name> : <Data Type> : <Formula>
```

Where,

<Variable Name> is the name of variable.

<Data Type> is the data type of the variable.

<Formula> can be an expression which evaluates to value of the variable data type.

Example:

```
   Filter Var : Fin Obj Var : Logical : $$Number:$BilledQty > 100
```

Using Variable as a Data Source for Collections

The collection attribute Data Source is enhanced to support ‘Variable’ as data source. Now variable element(s) can be gathered as objects in the collection and their respective simple member variables are available as methods. Member List Variables will be treated as sub-collections.

Syntax

```
      Data Source  : <Type> : <Identity> [:<Encoding>]
```

Where,

<Type> is the type of data source. File XML, HTTP XML, Report, Parent Report, Variable.

<Identity> can be file path / scope key words / variable specification based on the type of data source.

<Encoding> can be ASCII or UNICODE. It is applicable for the data source types File XML and HTTP XML.

Example:

Simple List Variable as Data Source

```
[Collection: LV List Collection]

   Data Source: Variable: SLVEmp
```

The elements of the Simple List Variable ‘SLVEmp’ will be available as objects in the collection ‘LV List Collection’.

Let us suppose if a Line is repeated over the collection ‘LV List Collection’ and the value can be retrieved in the field level as shown below:-

```
[Field: SLVEmp Field]

   Use      :  Name Field

   Set as   :  $SLVEmp
```

Compound List Variable as Data Source

```
[Collection: CV List Collection]
   Data Source: Variable: CLVEmp
```

The elements of the Compound List Variable CLVEmp will be available as objects in the collection CV List Collection. It is used as a Source Collection in the below Summary Collection

```
[Collection: CV List SummaryCollection1]
   Source Collection: CV List Collection
   Walk              : Relatives
   By                : Relation: $Relation
   Aggr Compute      : MaxAge: Max: $Age
   Aggr Compute      : MinAge: Min: $Age
   Aggr Compute      : TotSal: Sum: $Salary
```

In this example, we are walking to the sub-collection "Relatives" and then performing grouping and aggregation.

Variables in Remoting

In a Tally.NET Environment where Tally at the remote end sends request to the Tally Company at the Server, all client requests must contain the dependencies based on which data is gathered. In other words, any request sent to the server must accompany the values configured at the client to extract data from the server. For e.g., A Collection of Ledgers falling under user selected group must accompany with the request sent to the server. Hence, the request to the server must contain the Variable value which signifies the Group name.

Only the configuration information which is relevant to the data fetching from the Server needs to be sent to the Server and not the ones which are User Interface related like Show Vertical Balance Sheet, Show Percentages, etc.

When a Collection is sent to the Server, all the dependencies i.e., variable values are enclosed within the requests **automatically**.

Consider the following examples

Example 1:

```
[Collection:  User Ledger Coll]
   Type      : Ledger
   Child of  : ##UserSelectedGroup
```

While sending this collection to the server, automatically the value for the variable **UserSelectedGroup** is also passed to the server and server responds accordingly.

Example 2:

```
[Collection:  Emp Coll]
   Type    : Cost Centre
   Filter :  EmpSpouseName

[System: Formula]
   EmpSpouseName: $SpouseName= ##CLVEMP[1].Relatives[1].Name
```

In the above example, variable value of CLVEMP[1].Relatives[1]. Name will be enclosed within the request to the server

In some cases, variable values will not be remoted automatically like Child Of : $FuncName which in turn returns variable value through the Function. Such variables need to be remoted using an adhoc Compute within the collection. This Compute is required to set a manual dependency on the variable and hence consider this while sending request to Server.

Let us consider the below example

```
[Collection: User Ledger Coll]
   Type     : Ledger
   Child of : $$UserFunc

[Function: UserFunc]
   00 : RETURN   : ##FuncVar
```

In this example, the function **UserFunc** returns the value through the variable ‘FuncVar’. Hence, the variable ‘FuncVar’ need to be remoted using an adhoc Compute like below

```
[Collection: User Ledger Coll]
   Type     : Ledger
   Child of : $$UserFunc
   Compute  : FuncVar   : ##FuncVar
```

Please refer to the Sample Codes which are available in the Tally.Developer9 Folder related to Simple, Simple Repeat Variables and Static Variables.

Use Case – Report Configuration

Scenario

ABC Company Limited, who is in to trading business is using Tally.ERP 9. It deals with purchase and sale of computers, printers etc. The company management likes to view the Stock Summary Report in various dimensions. Hence, every time they need to set the configurations for the report and view it. They want to have multiple configurations for the same report and they want to set it at one time.

Requirement Statement

By default, in Tally.ERP 9, the user has to set the configurations in Stock Summary Report as per the requirement every time. The requirement can be customized using the Compound List Variable.

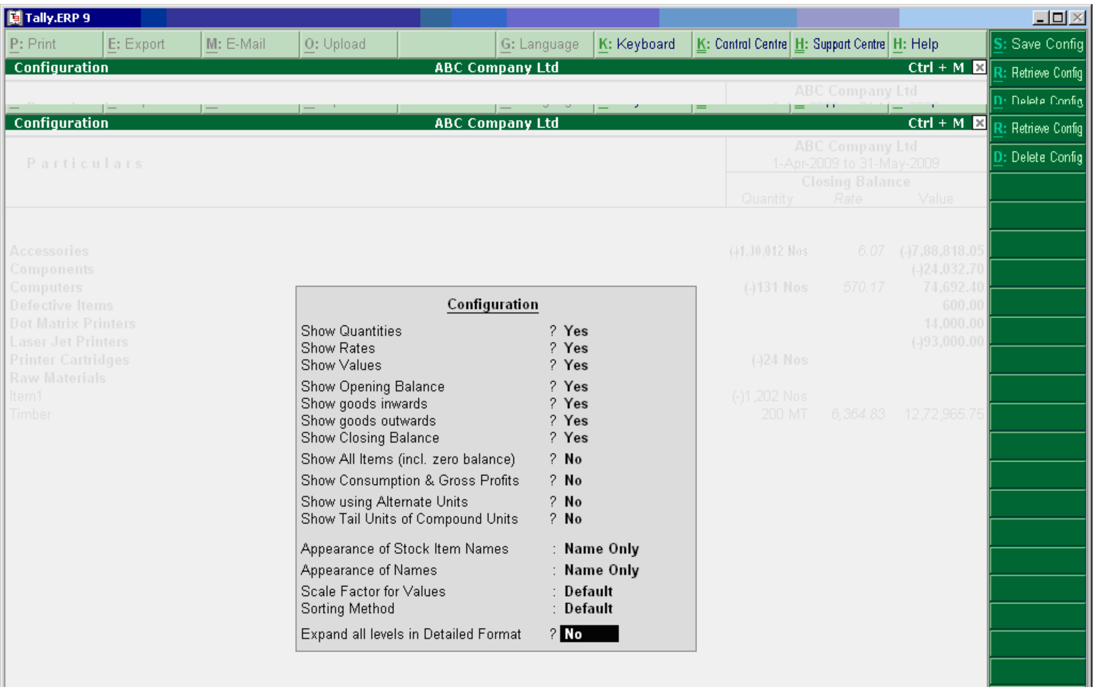
Functional Demo

The solution has been developed using Compound Variables and User Defined Functions. Before looking into the design logic, we will have a functional demo.

A new stock summary report is created for demonstration purpose and the same is available as a part of “TDL Language Enhancements” sample TDLs. Enable the TDL in Tally.ERP 9.

Saving Multiple Configurations

Gateway of Tally –> TDL Language Enhancements –> What's New –> Release 1.8 –> Variable Framework –> Stock Summary –> F12 –> Set the required configuration.

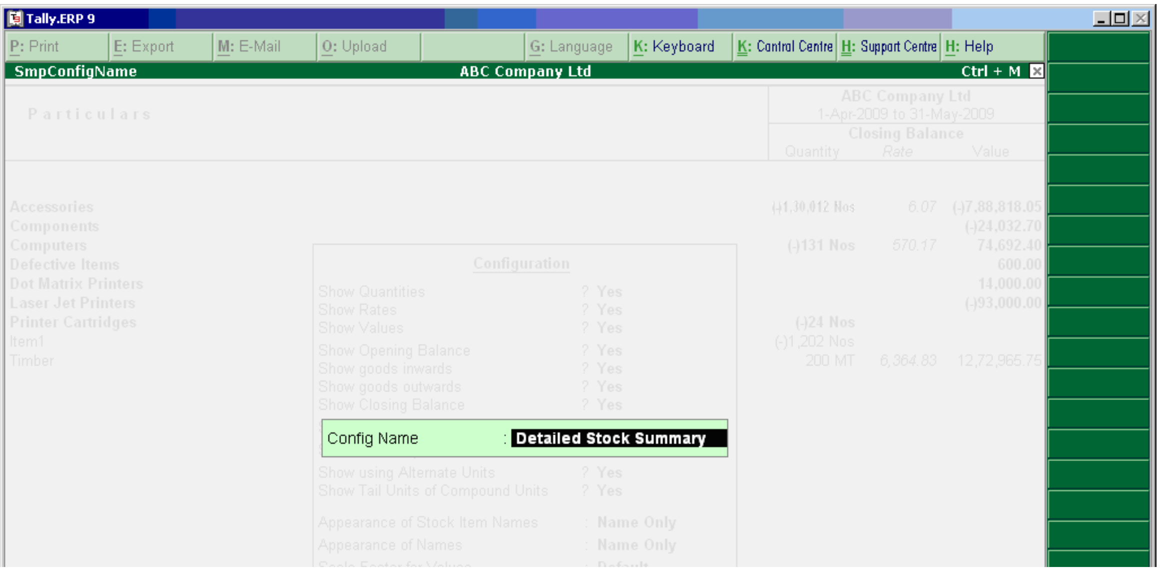


Configuration

Show Quantities ? Yes
Show Rates ? Yes
Show Values ? Yes
Show Opening Balance ? Yes
Show goods inwards ? Yes
Show goods outwards ? Yes
Show Closing Balance ? Yes
Show All Items (incl. zero balance) ? No
Show Consumption & Gross Profits ? No
Show using Alternate Units ? No
Show Tail Units of Compound Units ? No
Appearance of Stock Item Names : Name Only
Appearance of Names : Name Only
Scale Factor for Values : Default
Sorting Method : Default
Expand all levels in Detailed Format ? No

Figure 1.21 Setting required Configurations

The above configuration has to be saved using the button **Alt+S** (Save Config). Enter the Configuration Name and accept it as shown below.

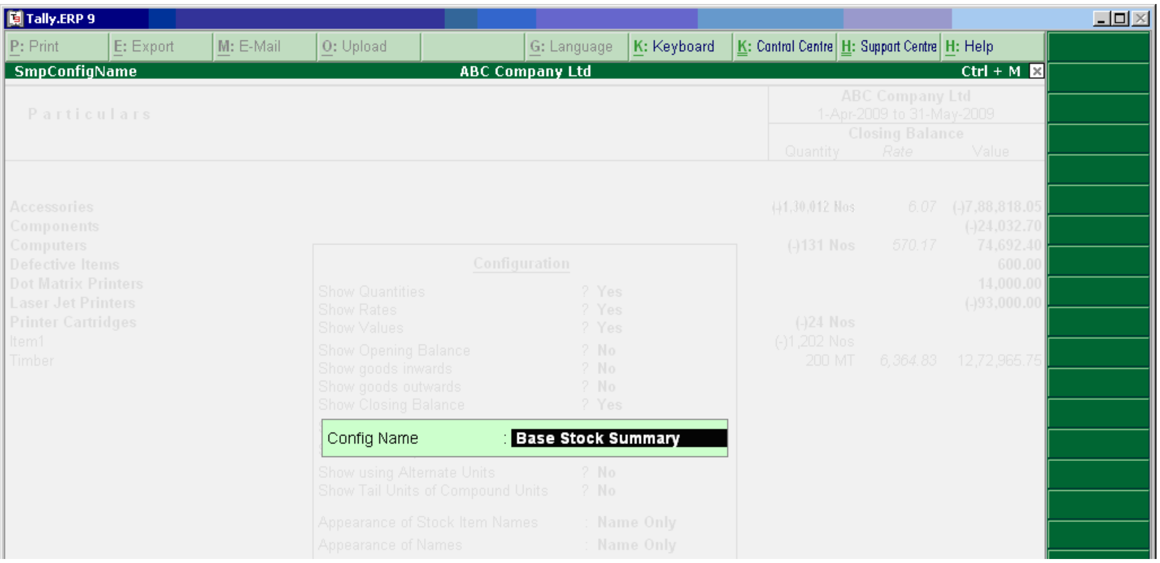


```
Particulars                          ABC Company Ltd
                                     1-Apr-2009 to 31-May-2009
                                     Closing Balance
                                     Quantity       Rate        Value

Accessories                          (-)1,30,012 Nos  6.07   (-)7,88,818.05
Components                                                   (-)24,032.70
Computers                            (-)131 Nos     570.17      74,692.40
Defective Items                                                    600.00
Dot Matrix Printers                                             14,000.00
Laser Jet Printers                                           (-)93,000.00
Printer Cartridges                   (-)24 Nos
Item1                                (-)1,202 Nos
Timber                               200 MT       6,364.83   12,72,965.75
```

```
Configuration

Show Quantities                    ? Yes
Show Rates                         ? Yes
Show Values                        ? Yes
Show Opening Balance               ? Yes
Show goods inwards                 ? Yes
Show goods outwards                ? Yes
Show Closing Balance               ? Yes

Config Name                        : Detailed Stock Summary

Show using Alternate Units         ? Yes
Show Tail Units of Compound Units  ? Yes

Appearance of Stock Item Names     : Name Only
Appearance of Names                : Name Only
```

Figure 1.22 Saving the Configuration with a suitable name

Similarly, we can save another configuration for the same report as shown below.



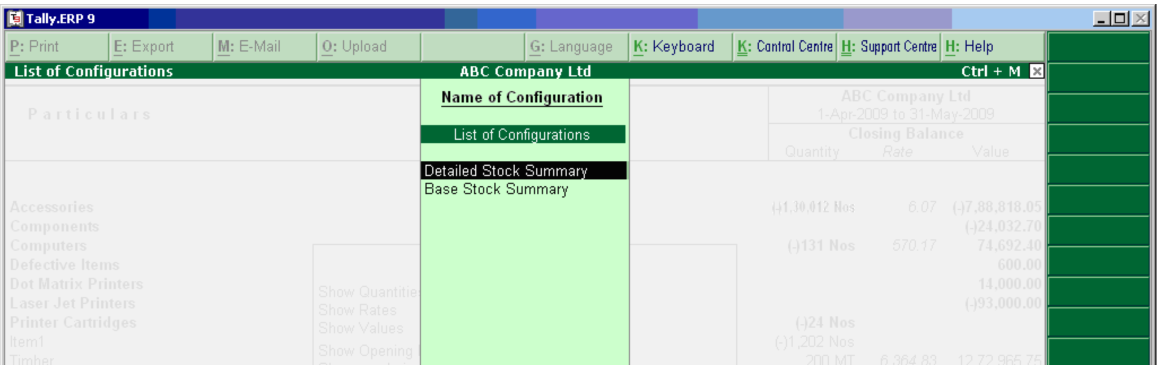
```
Particulars                          ABC Company Ltd
                                     1-Apr-2009 to 31-May-2009
                                     Closing Balance
                                     Quantity       Rate        Value

Accessories                          (-)1,30,012 Nos  6.07   (-)7,88,818.05
Components                                                   (-)24,032.70
Computers                            (-)131 Nos     570.17      74,692.40
Defective Items                                                    600.00
Dot Matrix Printers                                             14,000.00
Laser Jet Printers                                           (-)93,000.00
Printer Cartridges                   (-)24 Nos
Item1                                (-)1,202 Nos
Timber                               200 MT       6,364.83   12,72,965.75
```

```
Configuration

Show Quantities                    ? Yes
Show Rates                         ? Yes
Show Values                        ? Yes
Show Opening Balance               ? No
Show goods inwards                 ? No
Show goods outwards                ? No
Show Closing Balance               ? Yes

Config Name                        : Base Stock Summary

Show using Alternate Units         ? No
Show Tail Units of Compound Units  ? No

Appearance of Stock Item Names     : Name Only
Appearance of Names                : Name Only
```

Figure 1.23 Saving another Configuration

Retrieving Configuration to view the Report in Different Dimensions

Gateway of Tally –> TDL Language Enhancements –> What's New –> Release 1.8 –> Variable Framework –> Stock Summary –> F12 –> Alt+R (Retrieve Config) Select the Required Configuration and Enter.



List of Configurations — ABC Company Ltd

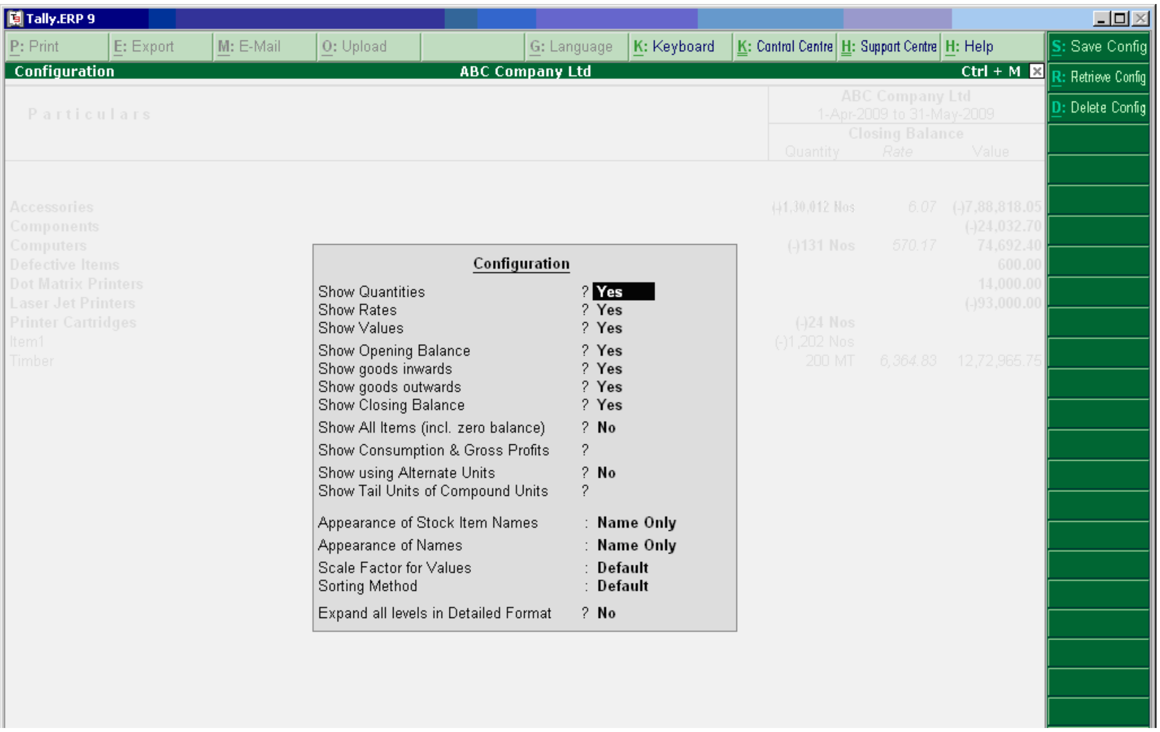
Name of Configuration

List of Configurations

Detailed Stock Summary
Base Stock Summary

Figure 1.24 Retrieving and Selecting the Required Configuration

The configuration will be set automatically as shown below.



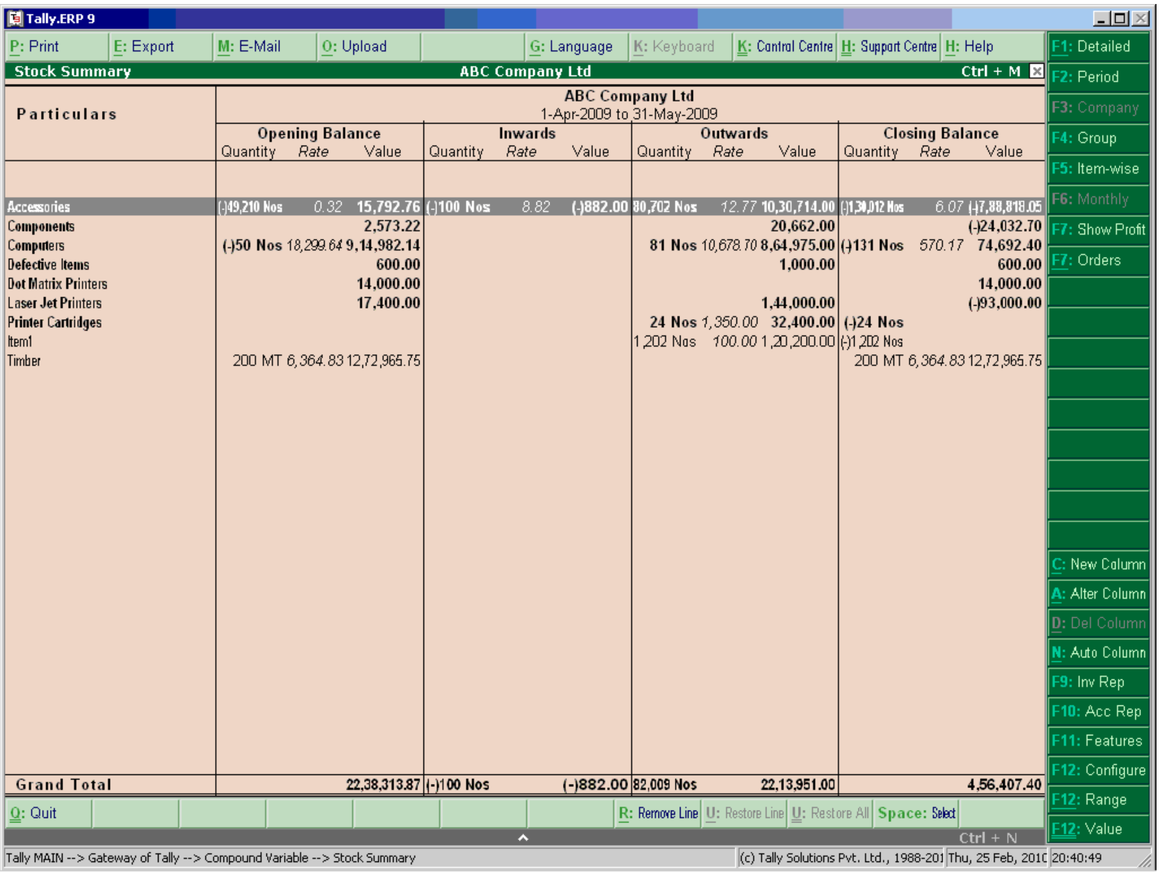
Configuration — ABC Company Ltd

Configuration

```
Show Quantities                         ? Yes
Show Rates                              ? Yes
Show Values                             ? Yes
Show Opening Balance                    ? Yes
Show goods inwards                      ? Yes
Show goods outwards                     ? Yes
Show Closing Balance                    ? Yes
Show All Items (incl. zero balance)     ? No
Show Consumption & Gross Profits        ?
Show using Alternate Units              ? No
Show Tail Units of Compound Units       ?

Appearance of Stock Item Names          : Name Only
Appearance of Names                     : Name Only
Scale Factor for Values                 : Default
Sorting Method                          : Default

Expand all levels in Detailed Format    ? No
```

Figure 1.25 Applying the selected Configuration

Accept the screen to view the Report.



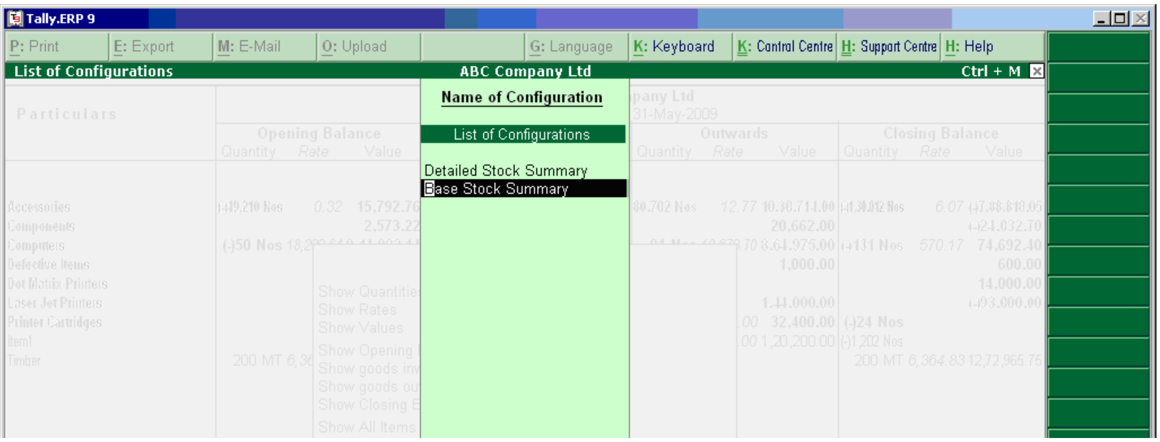
Stock Summary — ABC Company Ltd

ABC Company Ltd
1-Apr-2009 to 31-May-2009

Particulars	Opening Balance			Inwards			Outwards			Closing Balance		
	Quantity	Rate	Value	Quantity	Rate	Value	Quantity	Rate	Value	Quantity	Rate	Value
Accessories	(-)49,210 Nos	0.32	15,792.76	(-)100 Nos	8.82	(-)882.00	80,702 Nos	12.77	10,30,714.00	(-)1,30,012 Nos	6.07	(-)7,88,818.05
Components			2,573.22						20,662.00			(-)24,032.70
Computers	(-)50 Nos	18,299.64	9,14,982.14				81 Nos	10,678.70	8,64,975.00	(-)131 Nos	570.17	74,692.40
Defective Items			600.00						1,000.00			600.00
Dot Matrix Printers			14,000.00									14,000.00
Laser Jet Printers			17,400.00						1,44,000.00			(-)93,000.00
Printer Cartridges							24 Nos	1,350.00	32,400.00	(-)24 Nos		
Item1							1,202 Nos	100.00	1,20,200.00	(-)1,202 Nos		
Timber	200 MT	6,364.83	12,72,965.75							200 MT	6,364.83	12,72,965.75
Grand Total			22,38,313.87	(-)100 Nos		(-)882.00	82,009 Nos		22,13,951.00			4,56,407.40

Figure 1.26 Report configured as per the selection

To view the same Report with another configuration, **Press F12 –> Alt+R –> Select the required configuration and press Enter**.



List of Configurations — ABC Company Ltd

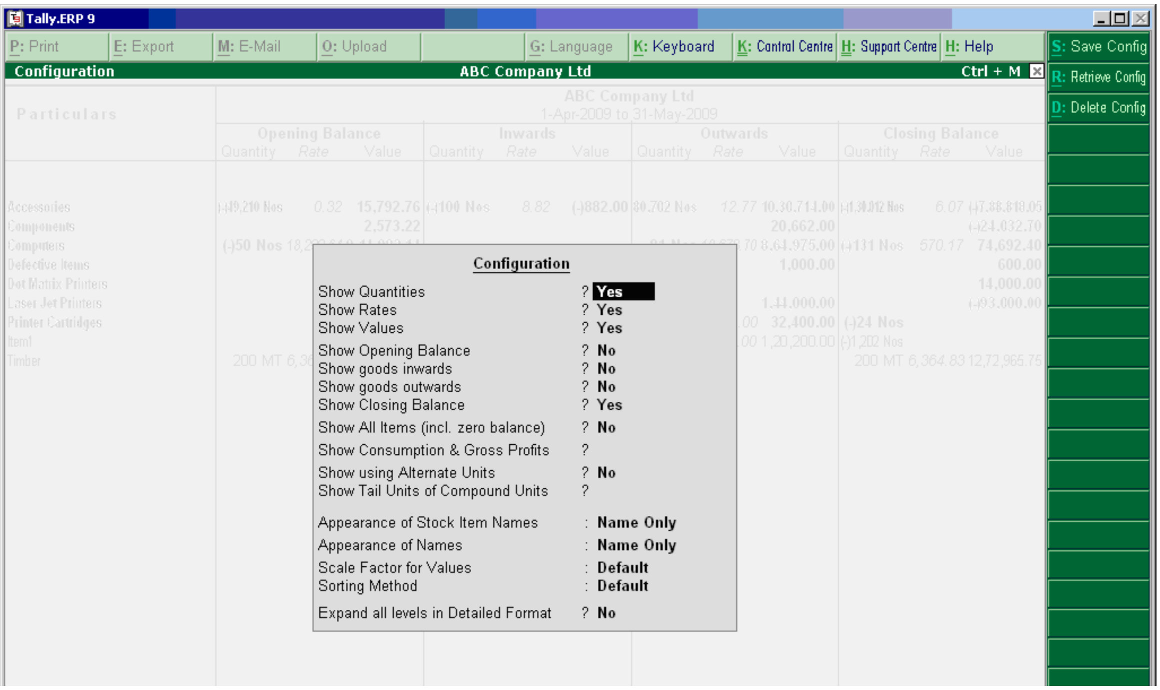
Name of Configuration

List of Configurations

Detailed Stock Summary
Base Stock Summary

Figure 1.27 Retrieving and Selecting another Configuration

The configuration will be set automatically as shown below.

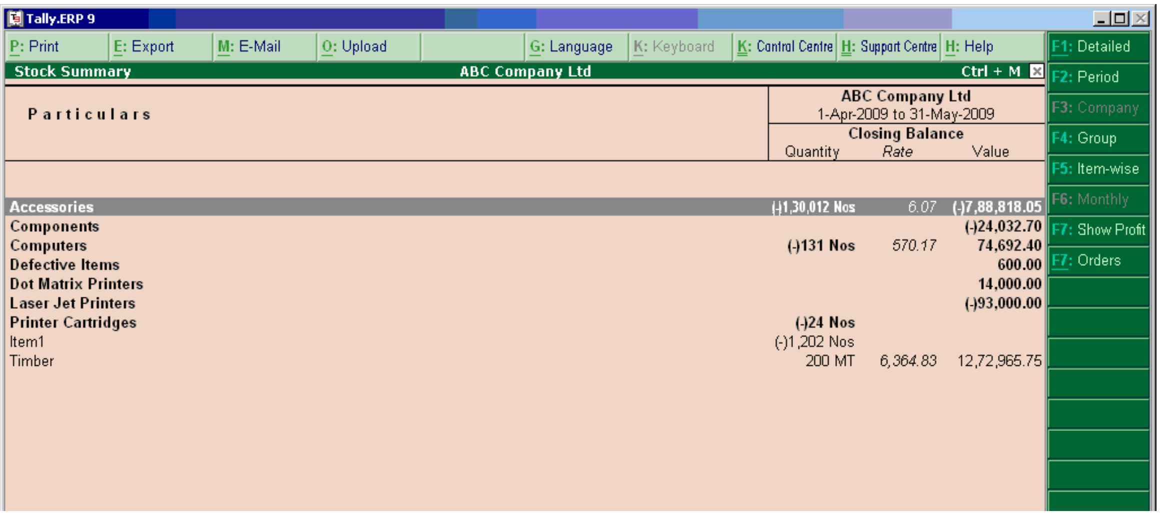


Configuration

Show Quantities	?	Yes
Show Rates	?	Yes
Show Values	?	Yes
Show Opening Balance	?	No
Show goods inwards	?	No
Show goods outwards	?	No
Show Closing Balance	?	Yes
Show All Items (incl. zero balance)	?	No
Show Consumption & Gross Profits	?	
Show using Alternate Units	?	No
Show Tail Units of Compound Units	?	
Appearance of Stock Item Names	:	Name Only
Appearance of Names	:	Name Only
Scale Factor for Values	:	Default
Sorting Method	:	Default
Expand all levels in Detailed Format	?	No

Figure 1.28 Applying the selected Configurations

Accept the Screen to view the Report.



Particulars	ABC Company Ltd 1-Apr-2009 to 31-May-2009 Closing Balance		
	Quantity	Rate	Value
Accessories	(-)1,30,012 Nos	6.07	(-)7,88,818.05
Components			(-)24,032.70
Computers	(-)131 Nos	570.17	74,692.40
Defective Items			600.00
Dot Matrix Printers			14,000.00
Laser Jet Printers			(-)93,000.00
Printer Cartridges	(-)24 Nos		
Item1	(-)1,202 Nos		
Timber	200 MT	6,364.83	12,72,965.75

Figure 1.29 Report configured as per the selection

Deleting the Configuration

Gateway of Tally –> TDL Language Enhancements –> What's New –> Release 1.8–> Variable Framework –> Stock Summary –> F12 –> Alt+D (Delete Config).

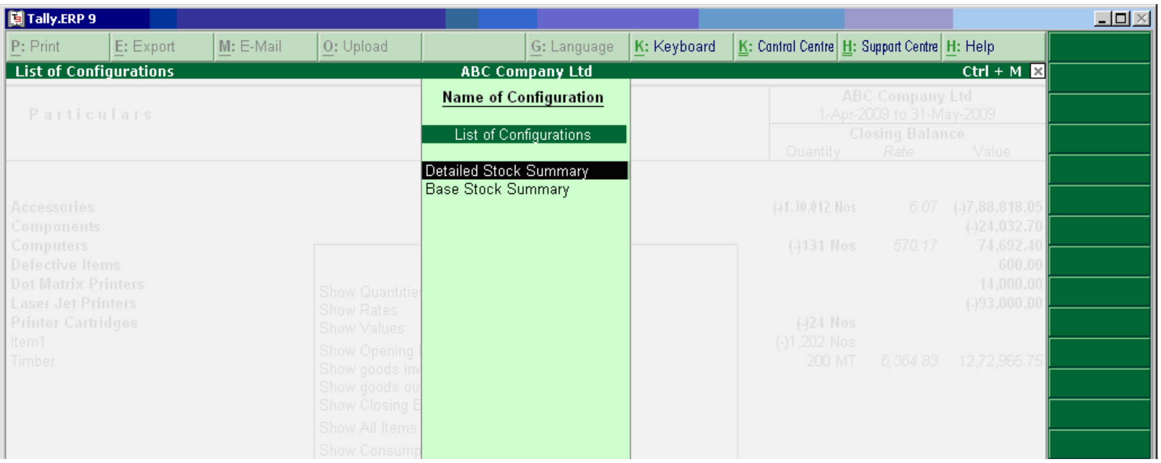


Figure 1.30 Selecting the configuration name for deletion

Select the Configuration to be deleted and press Enter.

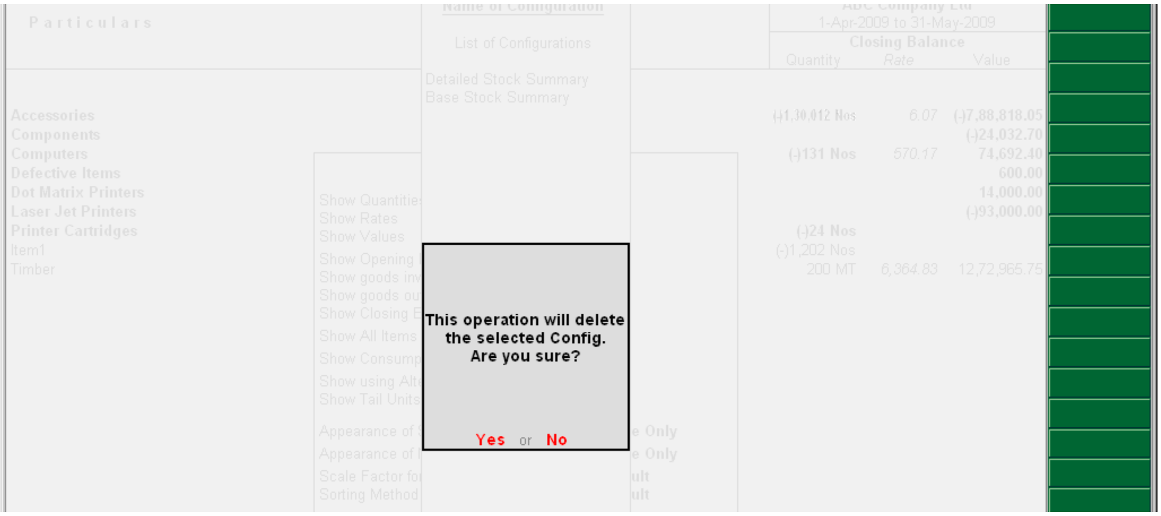
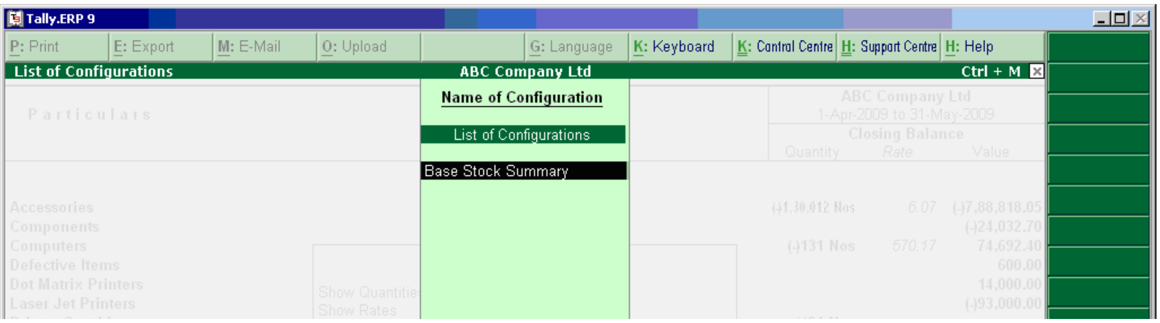

This operation will delete the selected Config. Are you sure?

Figure 1.31 Confirmation before deleting the selected configuration

Accept it to delete the configuration. Press **Alt+R** –> The report is not displaying the configuration “Detailed Stock Summary” (as shown below).



List of Configurations ABC Company Ltd

Name of Configuration

List of Configurations

Base Stock Summary

Particulars

ABC Company Ltd
1-Apr-2009 to 31-May-2009
Closing Balance

Particulars	Quantity	Rate	Value
Accessories	(-)1,30,012 Nos	6.07	(-)7,88,818.05
Components			(-)24,032.70
Computers	(-)131 Nos	570.17	74,692.40
Defective Items			600.00
Dot Matrix Printers			14,000.00
Laser Jet Printers			(-)93,000.00

Show Quantitie
Show Rates

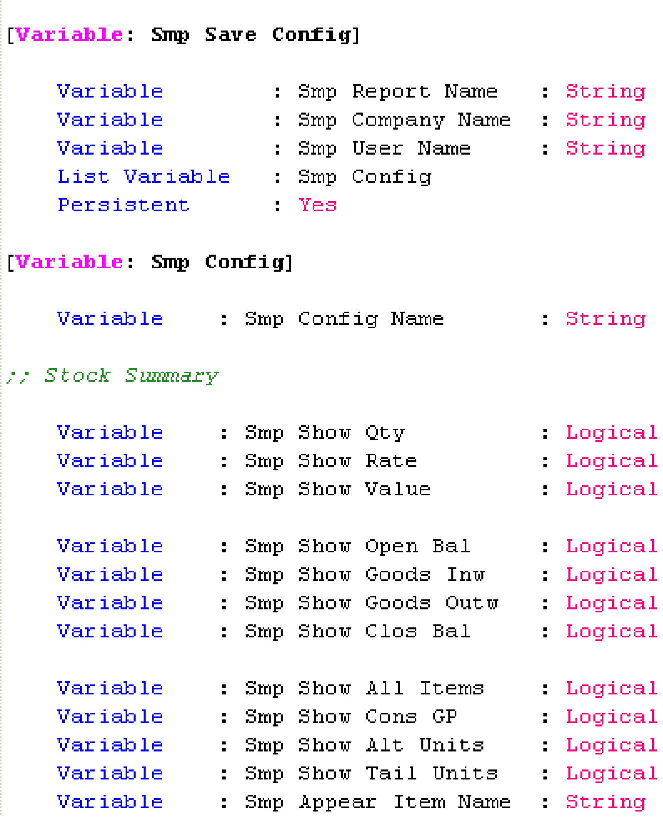
Figure 1.32 Report of Configuration List post deletion

Solution Development

The solution (Setting different configurations for the same report, saving and retrieving them as and when required) was possible by using Compound List Variable.

The steps followed to achieve the requirement are:-

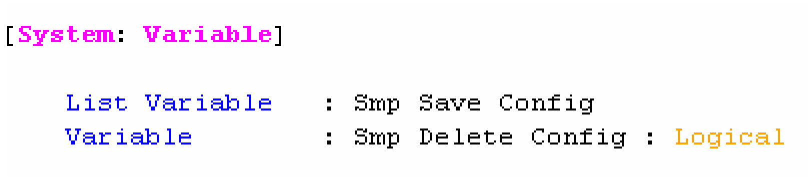
1. Defining Compound Variables with required members with Persistence behaviour



```
[Variable: Smp Save Config]

    Variable        : Smp Report Name    : String
    Variable        : Smp Company Name   : String
    Variable        : Smp User Name      : String
    List Variable   : Smp Config
    Persistent      : Yes

[Variable: Smp Config]

    Variable    : Smp Config Name       : String

;; Stock Summary

    Variable    : Smp Show Qty          : Logical
    Variable    : Smp Show Rate         : Logical
    Variable    : Smp Show Value        : Logical

    Variable    : Smp Show Open Bal     : Logical
    Variable    : Smp Show Goods Inw    : Logical
    Variable    : Smp Show Goods Outw   : Logical
    Variable    : Smp Show Clos Bal     : Logical

    Variable    : Smp Show All Items    : Logical
    Variable    : Smp Show Cons GP      : Logical
    Variable    : Smp Show Alt Units    : Logical
    Variable    : Smp Show Tail Units   : Logical
    Variable    : Smp Appear Item Name  : String
```

Figure 1.33 Defining Compound Variables

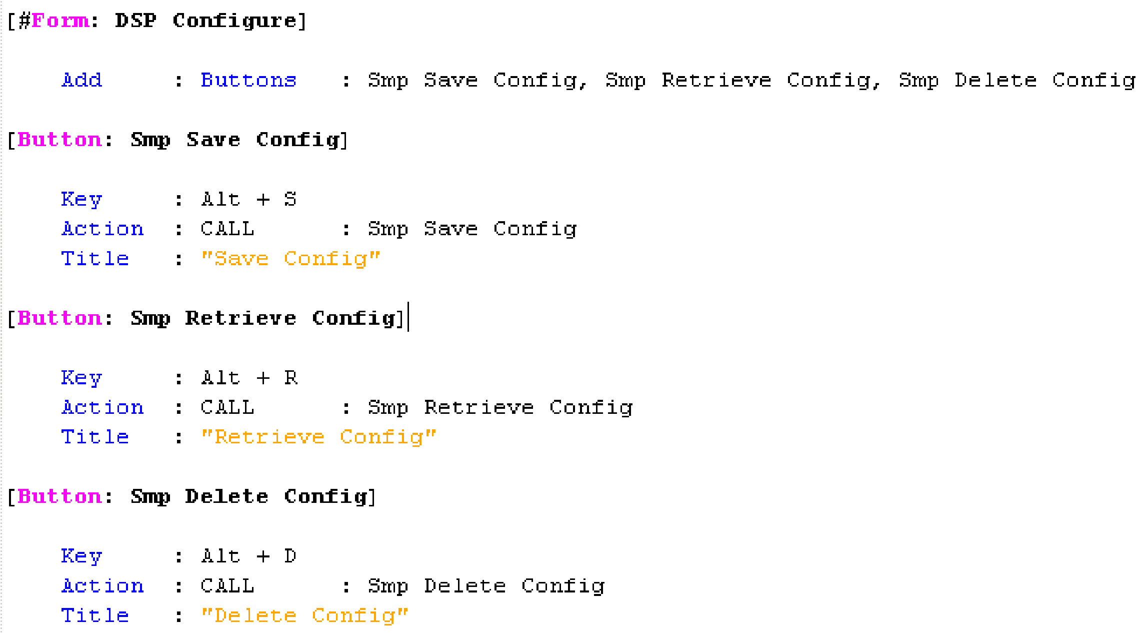
2. Declaring Compound List in System Scope



```
[System: Variable]

    List Variable    : Smp Save Config
    Variable         : Smp Delete Config : Logical
```

Figure 1.34 Declaring Compound List Variable in System Scope

3. Adding Relevant Buttons

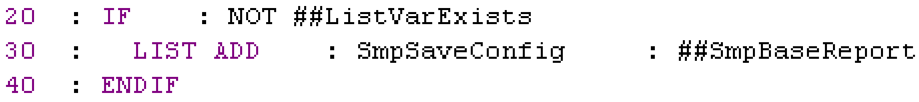


```
[#Form: DSP Configure]

    Add      : Buttons    : Smp Save Config, Smp Retrieve Config, Smp Delete Config

[Button: Smp Save Config]

    Key      : Alt + S
    Action   : CALL       : Smp Save Config
    Title    : "Save Config"

[Button: Smp Retrieve Config]

    Key      : Alt + R
    Action   : CALL       : Smp Retrieve Config
    Title    : "Retrieve Config"

[Button: Smp Delete Config]

    Key      : Alt + D
    Action   : CALL       : Smp Delete Config
    Title    : "Delete Config"
```

Figure 1.35 Adding relevant buttons

4. When the user chooses to save a configuration,

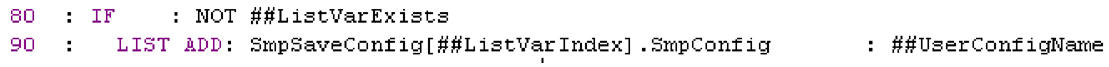
- Add an element with current Report Name as Key



```
20  : IF     : NOT ##ListVarExists
30  :   LIST ADD    : SmpSaveConfig      : ##SmpBaseReport
40  : ENDIF
```

Figure 1.36 Adding an element to the list variable

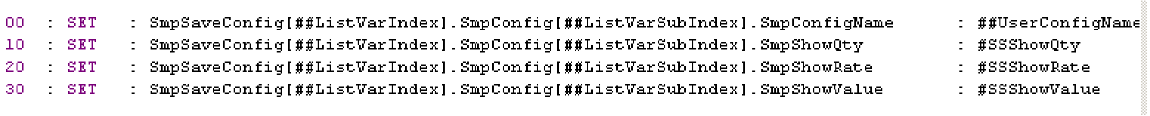
- Add an element within the above element with Config Name (specified by user) as Key.



```
80  : IF     : NOT ##ListVarExists
90  :   LIST ADD: SmpSaveConfig[##ListVarIndex].SmpConfig       : ##UserConfigName
```

Figure 1.37 Adding a sub-element

- Set the **variable values** in the current Configuration Screen to the respective **Members** within the above sub element.

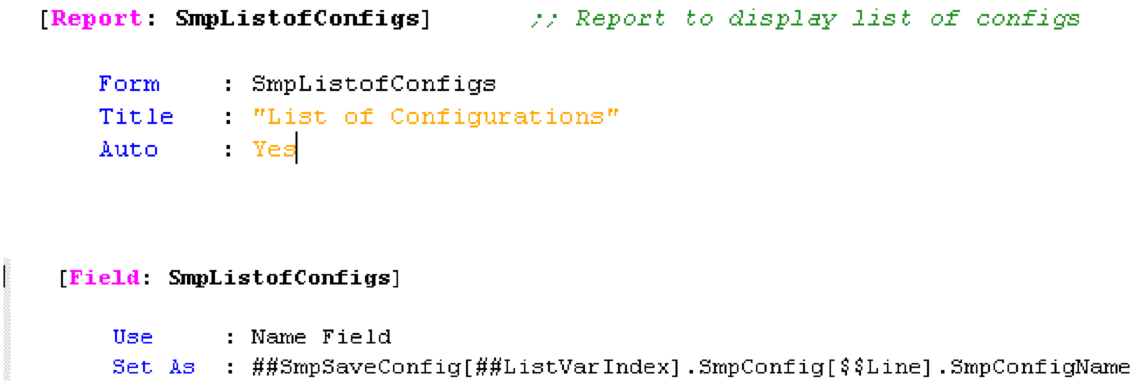


```
00  : SET   : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpConfigName      : ##UserConfigName
10  : SET   : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowQty         : #SSShowQty
20  : SET   : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowRate        : #SSShowRate
30  : SET   : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowValue       : #SSShowValue
```

Figure 1.38 Setting the member variable values

5. When the user chooses to retrieve a configuration,

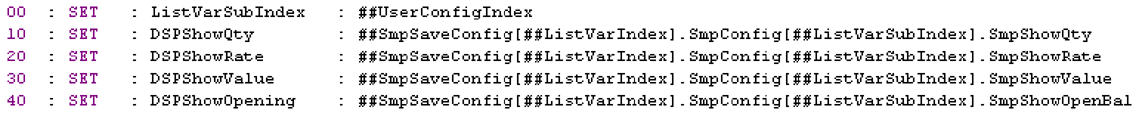
- Display a Report showing list of available Configurations in a Table.



```
[Report: SmpListofConfigs]       ;; Report to display list of configs

    Form     : SmpListofConfigs
    Title    : "List of Configurations"
    Auto     : Yes


 [Field: SmpListofConfigs]

     Use      : Name Field
     Set As   : ##SmpSaveConfig[##ListVarIndex].SmpConfig[$$Line].SmpConfigName
```

Figure 1.39 Report to display list of configurations

- On selecting the desired configuration, retrieve the saved values from the compound variable and set the values to the respective configuration variables.

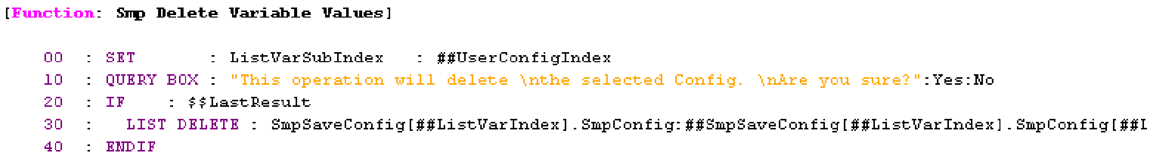


```
00  : SET   : ListVarSubIndex    : ##UserConfigIndex
10  : SET   : DSPShowQty         : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowQty
20  : SET   : DSPShowRate        : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowRate
30  : SET   : DSPShowValue       : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowValue
40  : SET   : DSPShowOpening     : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowOpenBal
```

Figure 1.40 Applying selected configuration values to the Report variables

6. When the user chooses to delete a configuration,

- Display a Report showing list of available Configurations in a Table.
- On selecting the desired configuration, delete the saved values from the compound variable.



```
[Function: Smp Delete Variable Values]

    00  : SET        : ListVarSubIndex   : ##UserConfigIndex
    10  : QUERY BOX : "This operation will delete \nthe selected Config. \nAre you sure?":Yes:No
    20  : IF     : $$LastResult
    30  :   LIST DELETE : SmpSaveConfig[##ListVarIndex].SmpConfig:##SmpSaveConfig[##ListVarIndex].SmpConfig[##I
    40  : ENDIF
```

Figure 1.41 Deleting the selected configuration from the List Variable

TDL Capabilities Used

- User Defined Functions
- Compound List Variable

Code Snippets are extracted from the working solution provided with the Samples.

6.5 Licensing Binding Mechanism

Nowadays, it is a common practice to have multiple applications for various business operations at different branches/ locations and then integrate their data and/or reports, as and when required. Tally, being the most common and popular product across all industries, many Third Party Applications look forward to integrate their applications with Tally.

To ensure secure environment, Third Party Applications need to build a robust licensing mechanism in order to validate the users of their application which may be time consuming and costly. Alternatively, they can opt to use the robust licensing mechanism already built in Tally and stitch it together with Tally Application.

License Information like Tally Serial Number, Account Email ID, etc can be retrieved from Tally and validated with the current instance of an external application. In order to use the Tally licensing mechanism, Third Party Applications need to send various XML Requests to Tally running at a predefined IP Address and a Port. On receiving the XML Request in Tally understandable format, Tally responds with the required information, data or a Report requested.

For any further information on XML Formats, please refer the documents and samples available at www.tallysolutions.com in the path Developer Network -> Tally Technology -> Integration Capabilities.

We have broadly classified various approaches for retrieving License Information from Tally that can be followed by Third Party Applications based on the desired level of security ranging from simple to the most complex one.

The Approaches that can be used by Third Party Applications to retrieve License Information from Tally based on the level of Security desired are as under:

License Info Retrieval using Open XML

This approach is one of the simplest approaches with minimal security wherein the Third Party Applications will be able to send an XML Request to invoke platform functions in Tally to retrieve the required License Information. This is a less secured environment as the license data returned will be available as an Open XML

In Tally, a platform function **LicenseInfo** is available which accepts a parameter to determine the type of License details required and returns the value accordingly.

For e.g., **$$LicenseInfo:SerialNumber** returns the Serial Number of the running copy of Tally.

Following are the list of parameters allowed for Function **LicenseInfo**:

Parameters permissible for LicenseInfo	**Return Type**	**Description**
SerialNumber	Number	Serial Number
RemoteSerialNumber	Number	Remote Serial Number in case of Remote Login
AccountID	String	Account ID
SiteID	String	Site ID
AdminEmailID	String	Admin Email ID
IsAdmin	Logical	Whether System logged in user is Administrator or not
IsIndian	Logical	Whether country is India or not
IsSilver	Logical	Whether Product flavour is Silver
IsGold	Logical	Whether Product flavour is Gold
IsEducationalMode	Logical	Whether Product is running in Educational mode
IsLicensedMode	Logical	Whether Product is running in Licensed mode
LicServerDate	Date	License Server Date
LicServerTime	String	License Server Time
LicServerDateTime	String	License Server Date & Time

The following XML Request is required to fetch Tally Serial Number:

```
<!-- XML Request -->
<ENVELOPE>
   <HEADER>
     <VERSION>1</VERSION>
     <TALLYREQUEST>EXPORT</TALLYREQUEST>
     <TYPE>FUNCTION</TYPE>
      <!-- Platform Function Name in Tally.ERP 9 -->
     <ID>$$LicenseInfo</ID>
   </HEADER>
   <BODY>
     <DESC>
       <FUNCPARAMLIST>
           <!-- Parameter for the function LicenseInfo -->
         <PARAM>Serial Number</PARAM>
       </FUNCPARAMLIST>
     </DESC>
   </BODY>
</ENVELOPE>
```

The previous XML Request fetches the following XML Response:

```
<!-- XML Response -->
<ENVELOPE>
    <HEADER>
        <VERSION>1</VERSION>
        <STATUS>1</STATUS>
    </HEADER>
    <BODY>
        <DESC>
        </DESC>
        <DATA>
          <RESULT TYPE="Long">790003089</RESULT>
        </DATA>
      </BODY>
</ENVELOPE>
```

In the above response received from Tally, Serial Number is retrieved within the **RESULT** Tag.

Similarly, to fetch Account ID of the current Tally Application, replace the Parameter **Serial Number** with **Account ID** within Param Tag in the XML Request.

License Info Retrieval using Encoding Procedure built in a TCP

This approach is a slightly better approach than the previous one since Response received here is encoded using some encoding mechanism built within TDL. Third Party Application will send a Validation String within the XML Request. At Tally's End the validation string and the required License Info will be encoded using the encoding mechanism built within TDL. The converted Strings will then be sent back within the XML Response to the Third Party Applications which will decode the strings at their end.

Following needs to be made available for this approach to be executed:

At Tally End

A TDL needs to be written containing the encryption mechanism to encrypt a string.

Following is an example of String encryption in Tally using TDL Function:

```
;; TDL Function to Encrypt an input String by reversing it
[Function: StrEnc]

   Parameter: pStringtoReverse: String
   Variable : ReverseString   : String

   00: FOR RANGE : IteratorVar : Number : ($$StringLength: +
       ##pStringtoReverse - 1): 0: 1
   10: SET: ReverseString : ##ReverseString + $$StringPart:+
       ##pStringToReverse : ##IteratorVar:1
   20: END FOR
   30: RETURN: ##ReverseString
```

Above is a simple example of String encryption in Tally. Similarly, much robust encryption mechanisms can be built in TDL and used in Third Party Applications.

- Report having a string variable and triggering the encrypt function with string variable as a parameter returning the encrypted value within required XML Tags.

```
;; TDL Report to invoke above Function
[Report: Sec XML Request2]
   Form    : Sec XML Request
;; Variable for received String
   Variable: EncString: String
```

```
[Form: Sec XML Request]
   Parts: Sec XML Request

[Part: Sec XML Request]
   Lines : Sec XML Req SerialNo, Sec XML Req EncString
   Scroll: Vertical
   XMLTAG: "TALLYLICENSEINFO"

;; Serial Number of Tally
   [Line: Sec XML Req SerialNo]
     Fields: Name Field
     Local : Field: Name Field: Set As : $$StrEnc:@@LicSlNo
     Local : Field: Name Field: XMLTAG : "SerialNumber"

;; To Encrypt the received String
   [Line: Sec XML Req EncString]
     Fields: Name Field
     Local : Field: Name Field: Set As : $$StrEnc:##EncString
     Local : Field: Name Field: XMLTAG : "EncryptedString"
```

On receiving XML Request, the report is executed and both Serial Number as well as String received within XML Request are encrypted and sent back to Third Party Applications.

At Third Party Application End

An XML Request to trigger the Tally Report with request String to be encrypted.
Following XML Request triggers the previous Report associated with Tally:

```
<!-- XML Request -->
<ENVELOPE>
   <HEADER>
     <VERSION>1</VERSION>
     <TALLYREQUEST>Export</TALLYREQUEST>
     <TYPE>Data</TYPE>
     <ID>Sec XML Request2</ID>
   </HEADER>
   <BODY>
     <DESC>
       <STATICVARIABLES>
```

```
                  <SVEXPORTFORMAT>$$SysName:XML</SVEXPORTFORMAT>
                  <EncString>Keshav</EncString>
         </STATICVARIABLES>
      </DESC>
   </BODY>
</ENVELOPE>
```

Sec XML Request2 is the TDL Report which is requested and variables **SVExportFormat** (format in which response is required) and **EncString** (Variable Name specified in TDL Report for string to be encrypted) are enclosed within the XML Request.

The following response is received from Tally on sending the above request

```
<!-- XML Response -->
<ENVELOPE>
   <TALLYLICENSEINFO>
   <SERIALNUMBER>980300097</SERIALNUMBER>
   <ENCRYPTEDSTRING>vahsek</ENCRYPTEDSTRING>
   </TALLYLICENSEINFO>
</ENVELOPE>
```

In the above response, Serial Number and String sent as request are returned encrypted i.e., reversed from Tally.

On receiving the response, Third Party Application needs to decrypt the Serial Number as well as String and validate the current instance. This is a much secure environment as the response is in encrypted form.

License Info Retrieval using Encryption Functions provided within Tally

This Approach is similar to the previous approach except that it uses an inbuilt Platform Function to encrypt the string. In Tally, the validation string and the required License Info can be encrypted using the function **$$EncryptStr** provided within the platform. The encrypted Strings will be sent back within the XML response to the Third Party Application. The Third Party Application will decrypt the Strings at their end using the standard DLL shipped by us for decryption.

XML Request is similar to the Request in the previous approach except that

- An additional variable value Password must be specified with the XMLTag **Password** (Variable Name used in TDL Report for Password) and
- The requested Report triggers the platform function **EncryptStr** for encryption mechanism.

A supporting DLL File **EncryptDecrypt.DLL** is provided along with Sample Files to decrypt the Encrypted String in Tally using Function **DecryptStr** available in DLL. This Function accepts 4 parameters viz.,

- Input String to be decrypted
- Password specified while encoding in XML request
- Output String Variable to hold the decrypted return Value
- Output String Buffer Length

The above DLL can be copied either to the local path of Third Party Application or Windows System Directory.

On decrypting the above string, Third Party Application can validate the returned String and Serial Number and continue if validation is successful.

License Info Retrieval using Encryption Algorithms built using Third Party DLLs

This Approach is the most secured approach wherein an external DLL is written to encrypt the given string. Third Party Application will send a Validation String within the XML Request. At Tally's End the validation string and the required License Info will be encrypted using an External DLL which can have its own Encryption Routines. Tally uses the **CallDllFunction** to trigger the DLL written for encryption and returns the encrypted strings to Third party Application within XML Response. At Third Party Application End decryption algorithms will be required which can again be provided inside the same DLL used for encryption.

Function CallDLLFunction

The Platform Function **CallDLLFunction** is used to trigger the function enclosed within an external DLL *(written in C++/VC++)*

Syntax

```
$$CallDLLFunction:<DLL Name>:<Function Name>:<Param 1>:+
                  <Param 2>:…<Param N>
```

where *DLL Name* is any DLL written in C++/VC++, *Function Name* is Function available in the DLL, *Param 1* to N are arguments which depend upon the number of parameters needed by the Function designed.

DLL must exist in the Tally Application folder or Windows System Folder.

XML Request for this approach is similar to the Request in the previous approach except that

- The requested Report triggers the function written within DLL for encryption mechanism using **CallDLLFunction**.

Subsequently, Third Party Application can decrypt the encrypted String and Serial Number using the decrypt function within the same DLL or any other DLL.

7. Enhancements in Release 1.61

7.1 General Enhancements

Reducing Table Search Enhancement

The current search capability on a Table allows the user to highlight a particular set of items based on the search text entered in the field. The text is searched from the beginning of the item names in the list and is applicable to the first column only.

In a scenario where there are large number of items in the list/table, it is impossible for the user to remember the starting characters of the item names. He may remember only a part of the item name which he requires to search. Even after the relevant items are searched and highlighted, all the items are displayed which is not required.

The latest enhancement in TDL allows the user to search a text from any part of the item name which appears in the list. The table keeps on narrowing down and displaying only those items which fulfill the search criteria. It is also possible now to specify whether the search criteria should be applicable on first column or all columns of the table.

Field Attribute – Table Search

We have introduced a new field attribute called ‘Table Search’ to achieve the above capability.

Syntax

```
      [Field: <Field name>]
          Table Search : <Enable reducing table search>:<Apply search to all +
                         columns>
```

Where,

<Enable reducing table search> – A logical value(yes/no) to specify whether we want to enable the reducing search or not.

<Apply search to all columns> – A logical value(yes/no) to specify whether the search criteria should apply to all columns of the table or not.

We can also use expressions in attribute values which evaluate to logical values

Function $$TableNumItems

A new function $$TableNumItems is introduced which returns the number of items in the list/table.

Example:

```
[Collection: RTS Ledger]
   Type        : Ledger
   Format      : $Name
   Format      : $Parent
   Format      : $ClosingBalance
```

```
[Field: Reducing Table Search GT 100]
   Use          : Name Field
   Table        : RTS Ledger
   Show Table   : Always
   Table Search : $$TableNumItems > 100  : Yes
```

In the above example the field ‘Reducing Table Search GT 100’ is displaying the table ‘RTS Ledger’ which has three columns ‘Name’,’Parent’ and ‘Closing Balance’. The attribute Table Search evaluates the first value to yes only when the number of items in the table exceeds 100 ie reducing search will be enabled if this criteria is met. The second attribute value is set to ‘Yes’ ie the search criteria will apply to all columns in the table.

Functionality Achieved

Using the above capability we have been able to deliver the functionality of applying the above search technique to all the tables available in the default product. This will be ofcourse based on the configuration settings selected by the user.

Use Cases

1. Company search based on the Company ID.
2. Ledger search based on parent Group name available in other column in a table.
3. While selecting the Stock item Name in a voucher the user can now narrow the search, based on the UOM and make his selection of item based on the closing balance available for that UOM.

8. Enhancements in Release 1.6

In this release there have been enhancements at the User Defined Functions, Collections and Actions.

This document takes us into depth of the changes for the Actions-Print,Upload,Export,Mail.It is now possible to program the configurations for these Actions. This breakthrough capability has enabled us to deliver the mass mailing feature in our product.

The collection attribute Keep Source is enhanced to accept a new value i.e 'Keep Source : ().This is done with an aim to improve the performance.The Loop Collection capability has paved the way for displaying and operating on Multi Company Data along with ease of programming.

We are enriching the TDL language with more and more procedural capabilities by introducing $$LoopIndex and Looping construct FOR RANGE.There have been some changes in the Action NEW OBJECT as well.

With the introduction on the function SysInfo,it is now possible to retrieve all system related information consistently using a single function.

8.1 General Enhancements

Programmable Configuration for Actions - “Print, Export, Upload, Mail”

In Tally.ERP9, Actions Print, Export, Mail and Upload depend upon various parameters like Printer Name, File Name, Email To, etc. Prior to execution of these actions, the relevant parameters are captured in a Configuration Report. These parameters are persisted as system variable so that next time these can be considered as default settings.

There are situations when multiple reports are being printed or mass mailing is being done in a sequence. Subsequent to each Print or Email Action, if a configuration report is popped up for user input, this interrupts the flow, thereby requiring a dedicated person to monitor the process which is time consuming too.

This has been addressed in the recent enhancement in Tally.ERP 9, where the configuration report can be suppressed by specifying a logical parameter. Also, the variables can be set prior to invoking the desired action.

Before exploring the new enhancements, let us see the existing behavior of the actions Print, Email, Export and Upload.

Existing behavior of Actions - Print, Export, Mail, Upload

Presently whenever any of these actions is invoked, a configuration report is displayed to accept the user input.

In Tally.ERP 9 whenever the actions Print, Export, Mail or Upload are executed, a common configuration report 'SVPrintConfiguration' is displayed. The user provides the details in the configuration screen based on the action being executed. The action gets executed based on the values provided in the configuration.

The current syntax of these action is:

Syntax

```
      <Action Name> :  <Report Name>
```

Where,

<Action Name> can be any of the Mail, Upload, Print or Export.

<Report Name> is name of the report.

For successful execution of these actions along with the report name, additional action specific parameters are also required. These action specific parameters are passed by setting the value of variables through configuration report - 'SVPrintConfiguration'.

The default configuration report 'SVPrintConfiguration' is invoked only when the Report specified does not contain the Print attribute in its own definition. The Print attribute allows the user to specify his own configuration settings whenever any of these Actions are invoked.

Example:

```
   Mail : Balance Sheet
```

The action Mail needs information regarding the To email ID, From email ID, CC email ID, Email server name etc which are provided through the configuration report.

For example a report needs to be mailed to multiple emails at one go. Currently for every mail the configuration screen is displayed and every time the user has to manually provide the To email ID, From email ID, Email server name etc.

So whenever any of the above mentioned actions are executed, a configuration report is displayed which requires user inputs. In some scenarios this behavior is not desirable. The configuration setting can be specified once and user should be able to use it multiple times.

The action syntax is enhanced to avoid the display of configuration screen repeatedly.

Actions Changes for Programming Configurations

The global actions Print, Export, Mail and Upload are enhanced to suppress the configuration screen. These actions now, accept an additional logical parameter. Based on the value of the logical parameter the configuration report is suppressed.

The new syntax of these actions is:

Syntax

```
    <Action Name> :  <Report Name> : <Logical Value>
```

Where,
<Action Name> can be any of the Mail, Upload, Print or Export.
<Report Name> is name of the report.
<Logical Value> can be TRUE, FALSE, YES or NO.
With the new syntax it is possible to configure the values of the report only once and then mail it, to the specified email addresses without repeated display of the configuration report.

Example:

```
   10 : MAIL : Ledger Outstandings: TRUE
```

As the configuration report is not displayed, the values of the mail action specific variables like 'SVPrintFileName', 'SVOutputName' etc must be specified for the successful execution of these actions.
Following section explains the action specific variables and their acceptable values.

The Configuration Variables – Action Specific

The action specific Variables can be classified in four categories based on their usage.

Common Variables
SVOutputType - The value of this variable is one of the predefined button type keywords like 'Print Button', 'Export Button', 'Upload Button', 'Mail Button'. The variables value is used by the functions $$InMailAction, $$InPrintAction, $$InUploadAction and $$InExportAction to determine the execution of correct option in the form 'SVPrintConfiguration'.

For eg, if the value of 'SVOutputType' is 'Print Button' then the optional form 'SV PrintConfig' in the report 'SVPrintConfiguration' is executed.

SVPrintFileName - This variable accepts the output location as value. The value of this variable is specific to each action. The usage is explained for each action in detail along with each Action.

SVExportFormat - The value of this variable is the name of the format to be used with actions Mail, Upload and Export actions. The values SDF, ASCII, HTML, EXCEL, XML, AnsiSDF, AnsiASCII, AnsiXML, AnsiHTML and AnsiExcel which are set using the $$SysName.

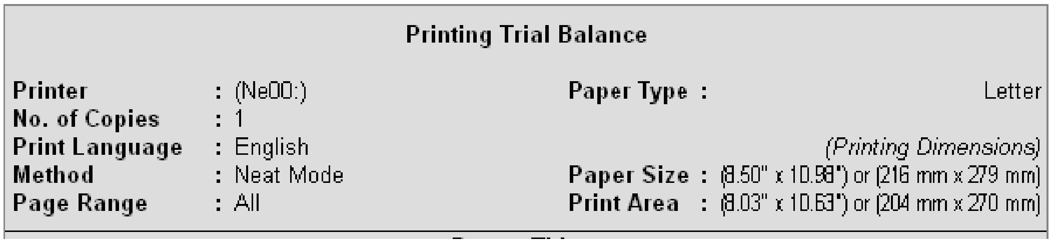
Example:

```
SET: SVExportFormat: $$SysName:Excel
```

SVExcelExportUpdateBook - This is a logical value and can be used only if the 'Excel' format is used. If the value is set to YES then the existing file is overwritten other wise it will ask for "Overwrite" confirmation screen.

SVBrowserWidth, SVBrowserHeight - These variables are used to set the width and height of the page when the format is HTML.

Variables Specific to action – Print
As soon as the user executes a Print action following screen is displayed:



Printing Trial Balance

Printer : (Ne00:)
No. of Copies : 1
Print Language : English
Method : Neat Mode
Page Range : All

Paper Type : Letter
(Printing Dimensions)
Paper Size : (8.50" x 10.98") or (216 mm x 279 mm)
Print Area : (8.03" x 10.63") or (204 mm x 270 mm)

Figure 1.42 Print Screen

This screen captures the user input such as the "Printer Name", "No. of Copies" to be printed etc. The various action specific variables required by the Print action are modified based on the user input. Following variables are used by Print action:

SVPrintMode - This variable is used by Print action and accepts printing mode as value. The Print mode can be "Neat", "DMP" or "Draft" which are system names. The default mode is Neat mode.

SVPrintFileName - This variable is applicable for Print action, only if "Print To File" option is selected by the user while printing in DOT Matrix or Quick / Draft format. In this case the variable 'SVPrintFileName' accepts the filename by using the function $$MakeExportName function to add right extensions.

SVPrintToFile - This is a logical value which determines if the print output should be saved in a file. If the value is TRUE the output is saved in the file specified in the variable 'SVPrintFileName'. If the vale is FALSE then the variable 'SVPrinterName' must contain a valid printer name.

SVPrinterName - It accepts a printer name as a value for printing. The default value is taken from the system settings available in Control Panel for Printers and Faxes.

SVPreview - This is a logical variable and is applicable only for Print action with neat mode format. If the value is set to Yes then preview of the report is displayed. Otherwise the report is printed without displaying the preview.

SVPrintCopies - The variable is applicable only for Print action. It accepts a number to print multiple copies.

SVPrePrinted - The variable is applicable only for Print action and it specifies whether a pre-printed stationary or plain paper is to be used for printing.

SVPrintRange - The variable is applicable only for Print action. This variable determines the range of pages to be printed.

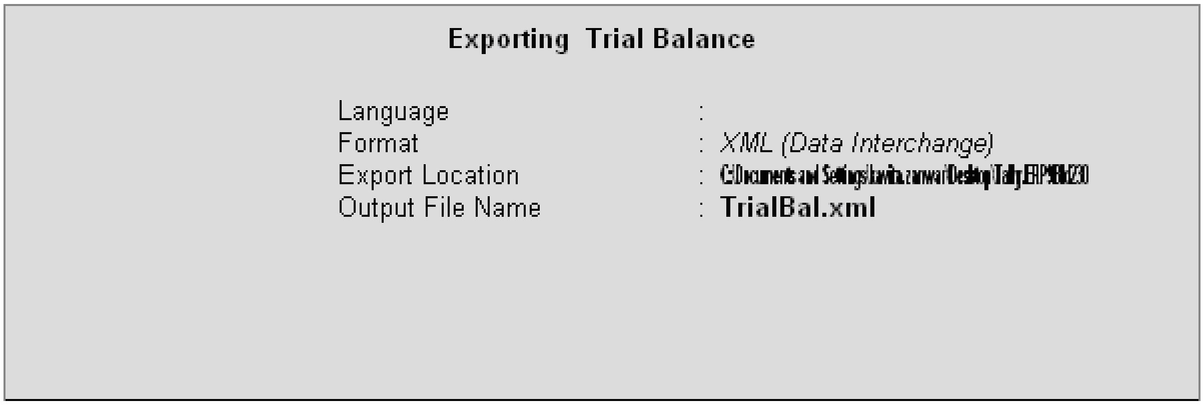
SV Draft Split Names - It accepts a logical value to determine if the long names should be split into multiple lines.

SV Draft Split Numbers - It accepts a logical value to determine if the long numbers should be split into multiple lines.

SVPrintStartPageNo - The variable is applicable only for Print action. This variable allows to specify the starting page number.

SVPosMode - It determines if POS mode to be used. The default value of this variable is No.

Variables Specific to action – Export
Following screen is displayed when the user executes Export action:



Exporting Trial Balance

Language :
Format : *XML (Data Interchange)*
Export Location : C:\Documents and Settings\[illegible]\Desktop\Tally.ERP9\[illegible]
Output File Name : **TrialBal.xml**

Figure 1.43 Export Screen

This screen captures the user input such as the "Export Format", "Output File Name" etc. The action Export uses the variables 'SVExportFormat', 'SVPrintFileName' etc.

SVPrintFileName - For the Export action, the value of this variable is the output file name. The path can be specified directly or the function $$MakeExportName can be used to create the output path.

The function $$MakeExportName suffixes the extension based on the export format if only the file name is passed as a parameter.

Syntax

```
$$MakeExportName:<String Formula>:<Export format>
```

Where,

<String Formula> is a string formula which evaluates to the path\filename.

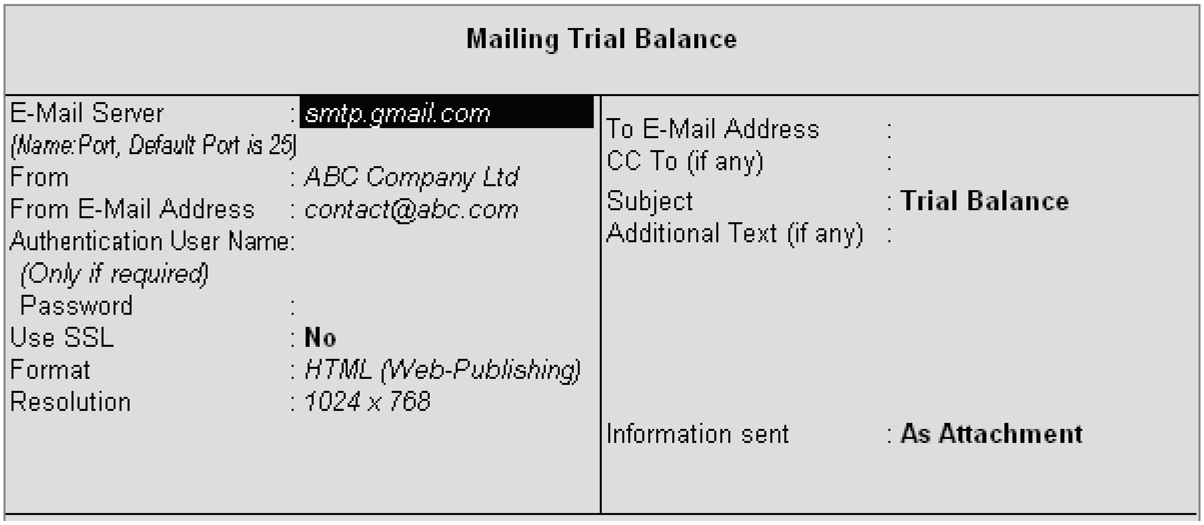
<Export format> is the name of format which has to be used while exporting.

Example:

```
$$MakeExportName:"C:\Tally.ERP\abc.xls":Excel
```

Variable Specific to action – Mail

When the user executes Mail action following screen is displayed to capture the mailing details:



Mailing Trial Balance

E-Mail Server : smtp.gmail.com
(Name:Port, Default Port is 25)
From : ABC Company Ltd
From E-Mail Address : contact@abc.com
Authentication User Name:
(Only if required)
Password :
Use SSL : **No**
Format : HTML (Web-Publishing)
Resolution : 1024 x 768

To E-Mail Address :
CC To (if any) :
Subject : **Trial Balance**
Additional Text (if any) :

Information sent : **As Attachment**

Figure 1.44 Mail Screen

For successful execution on mail action, the user has to enter above details. The URL is then created using the function $$MakeMailName and the value is stored in the variable 'SVPrintFileName'.

SVPrintFileName - This variable accepts the URL location as value. The function $$MakeMailName is used to construct the URL. The mail is sent to specified mail addresses using the given server.

Syntax

```
$$MakeMailName:<ToAddress>:<SMTP Sever name>:<From Address>:<CC +
               Address>:<Subject>:<Username>:<Password>:<Use SSL flag>
```

Where,

<To Address> is the email id of the receiver.

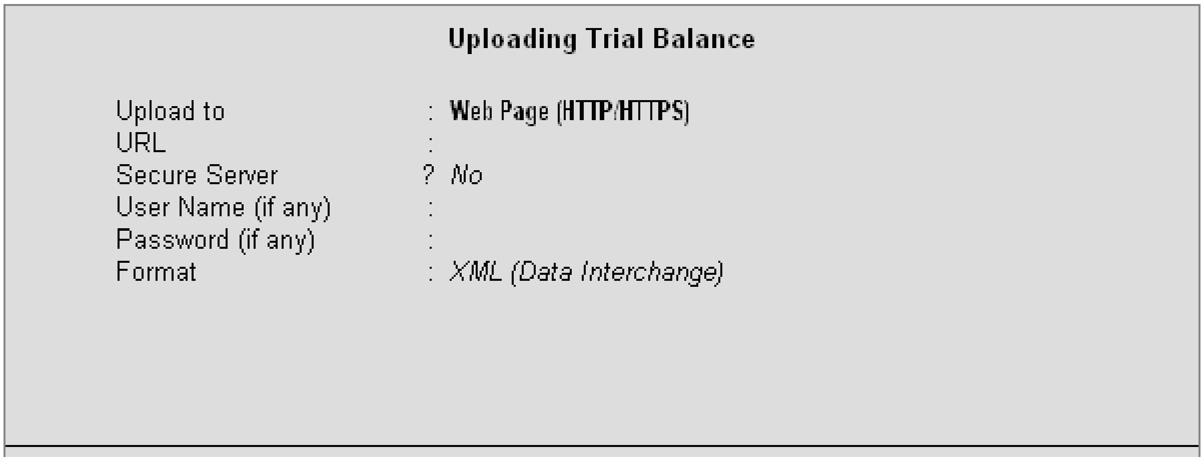
<SMTP Server Name> is name of server from which the mail is sent.
<From Address> is sender's email id.
<CC Address> is the email id where the copy of the mail is to be sent.
<Subject> is the subject of the mail.
<User Name> is the user id on the secured server.
<Password> is password for user id on the secured server.
<Use SSL Flag> can be TRUE, FALSE, YES or NO If the Use SSL flag is set to TRUE then, Username and Password must be specified it can't be empty.

Example:

```
   $$MakeMailName : "abc"+ "<" +" abc@abc.com" + ">" smtp.gmail.com":+
                    "abc@gmail.com":"":"Your outstanding  payment":+
                    abc@gmail.com:abc123:True
```

Variables Specific to actions – Upload

Following screen is displayed to capture the details of the folder where the report is to be uploaded:



Uploading Trial Balance

Upload to : Web Page (HTTP/HTTPS)
URL :
Secure Server ? No
User Name (if any) :
Password (if any) :
Format : XML (Data Interchange)

Figure 1.45 Upload Screen

Based on the information entered by the user the URL of the upload site is created using the function $$MakeHTTPName or $$MakeFTPName and the value is stored in the variable 'SVPrintFileName'.

SVPrintFileName - This variable accepts the URL of upload site. The URL is constructed using the functions $$MakeHTTPName or $$MakeFTPName depending on the protocol selected by the user for upload.

The function $$MakeFTPName is used for creating the file transfer protocol based on the specifications.

Syntax

```
        $$MakeFTPName:<FtpServer>:<FtpUser>:<FtpPassword>:<FtpPath>
```

Where,
<FtpServer> is the FTP server name.
<FtpUser> is the FTP user name.
<FtpPassword> is the FTP password.
<FtpPath> is the full path of the folder on the FTP server.

Example:

```
   $$MakeFTPName:"ftp://ftp.microsoft.com":"": "":"dbook.xml"
```

The function $$MakeHTTPName is used for creating the Hyper Text Transfer Protocol for the specified security features.

Syntax

```
        $$MakeHTTPName:<HttpUrl>:<HttpIsSecure>:<HttpUserName>:+
                       <HttpPassword>:<CompanyName>
```

Where,
<HttpUrl> is the HTTP URL name
<HttpIsSecure> is a logical attribute which checks whether the HTTP is secure or not.
<HttpUserName> is the HTTP user name.
<HttpPassword> is the HTTP password.
<CompanyName> is the name of Company.

Example:

```
   $$MakeHTTPName:"https://www.abc.com":Yes:"guestuser":"pswd99":+
                                             "ABC Company Ltd"
```

Use Case

Scenario:
The report "Bill-wise details" is to be mailed to each party with their respective Bill Details; however mails to all parties should be sent at one key stroke without e-mail configuration screen popping-up multiple times.

As of now, a user has to manually change the email IDs for each ledger.

Solution : Following steps needs to be implemented

Step 1: Create function

```
[Function   : FuncEmailingOutstanding]

   Variable: LedgerName : String
```

Step 2: Creating Local Formulae for enhanced readability of code

```
Local Formula: FromAddress : "abc" + "<abc@abc.com >"
Local Formula: ToAddress : if $$IsEmpty:$Email then " abc@abc.com "+
               else   $Email
Local Formula: Subject  : ##LedgerName + "(Bill-wise Details)"
```

Step 3: Setting the values of common variables used in mail action

```
03a: SET: SVExportFormat: $$SysName:HTML
```

Step 4: Walking the Ledger Collection to retrieve email-id of the ledger

```
01: WALK COLLECTION  : SDLedger
```

Step 5: Set the values of variables used in mail action

```
02 : SET: LedgerName : $Name
03b: SET: SVMailEmbedImage: @@AsAttach
03c: SET: ExplodeFlag    : "Detailed"
03d: SET: SVPrintFileName: $$MakeMailName:@ToAddress:+
     "smtp,gmail.com":@FromAddress: "admin@tallysolutions.com":+
     @Subject:"":"":FALSE
```

Step 6: Calling the action

```
04: MAIL: Ledger Outstandings: TRUE
```

;; TRUE to suppress the configuration report

```
06: END WALK
|
|
08: RETURN
```

8.2 Collection Enhancements

In the Collection definition the attribute Keep Source and Collection are enhanced. The collection attribute Keep Source is enhanced to accept a new value i.e. Keep Source : ‘().’ to make the data available at Primary Object which can be a Menu, Report or Function.

The Collection definition can now use a new capability to loop one collection for each object of another collection. This functionality is introduced by enhancing the Collection attribute.

Collection Attribute Value- Keep Source: ().

The attribute Keep Source accepts various values which is used to specify, the In memory source retention of the collection. The various specifications like . , ..,Yes , No were used earlier for this. The source collection is retained along with the data object associated with the User Interface object in the current User Interface object hierarchy as per specification. The newly introduced specification "()." is used to keep the source collection with the parent UI object which is either Report or Function.

The dotted notation depends on the interface object hierarchy. If there are recursive explodes in a report then its difficult to use the dotted notation when the data is to be kept at report or function level. The new value Keep Source : (). is introduced to overcome this issue.

The **Keep Source : ().** signifies that keep the collection data available at primary level which can be Menu or Report or Function.

So now Keep Source attribute accepts following values:

- Keep Source : **NO** - The source collection data is not kept in memory
- Keep Source : **YES** - Keep the source collection data in the object associated with the current interface object
- Keep Source : **().-** Keep the source collection data in the data object associated with the primary owner interface object i.e. Menu or Function or Report
- Keep Source : **.... -** Each dot in the notation is used to traverse upward in the owner chain by number of dots specified, till a primary interface object is found.

In the scenarios where the data is to be kept at Primary interface object, the application developer can directly use Keep Source : (). without worrying about the interface object hierarchy.

Example:

```
         |

[Part: TB Report KeepSrc Part]

   Lines        : TB Report KeepSrc Title, TB Report KeepSrc Details

   Bottom Lines: TB Report KeepSrc Total

   Repeat       :TB Report KeepSrc Details:TB Report KeepSrc GroupsPri
```

The line repeats on collection 'TB Report KeepSrc GroupsPri' which displays all the groups belonging to Primary group. The line then explodes to display the subgroups.

```
[Line: TB Report KeepSrc Details]

   Explode : TB Report KeepSrc Group Explosion : $$KeyExplode
```

In the part *'TB Report KeepSrc Group Explosion'*, if the object is group then once again the line explodes to display the sub-groups or the ledgers belonging to the current sub group.

```
[Part: TB Report KeepSrc Group Explosion]

   Lines :TB Report KeepSrc Details Explosion
   Repeat:TB Report KeepSrc Details Explosion:TB Report KeepSrc SubGrp
   Scroll:Vertical

[Line: TB Report KeepSrc Details Explosion]
   Explode:TB Report KeepSrc Group Explosion :$$KeyExplode
   Explode:TB Report KeepSrc Ledger Explosion :$$KeyExplode
   Indent :If $$IsGroup Then 2*$$ExplodeLevel Else 3  * $$ExplodeLevel
   Local  :Field : Default : Delete: Border
```

The part '*TB Report KeepSrc Group Explosion'* is exploded recursively. So its useful to keep the data at the primary interface object level.

The collections *'TB Report KeepSrc GroupsPri'* and *'TB Report KeepSrc SubGrp'* both use the same source collection *'TB Report KeepSrcGroups'*. The collections are defined as follows:

```
[Collection: TB Report KeepSrcGroups]
   Type : Group
   Fetch: Name, Parent, Closing Balance

[Collection: TB Report KeepSrc GroupsPri]
   Source Collection : TB Report KeepSrc Groups
   Filter            : PrimaryGrp
   By                : Name: $Name
   Compute           : Parent: $Parent
   Keep Source       : ().

[Collection: TB Report KeepSrc SubGrp]
   Source Collection : TB Report KeepSrc Groups
   Filter            : SubGrp
   By                : Name: $Name
   Compute           : Parent: $Parent
   Keep Source       : ().

[Collection: TB Report KeepSrc Ledgers]
   Type          : Ledger
   Child Of      : #MyGroupName1
   Filter        : Zero Filter
   Fetch         : Name ,Parent, Closing Balance
```

```
[System: Formula]
   Zero Filter    : $ClosingBalance > 0 AND NOT $$IsLedgerProfit
   PrimaryGrp     : $$IsSysNameEqual:Primary:$Parent
   SubGrp         : $Parent = #MyGroupName1
```

Attribute Collection Change – Loop Collection

The data processing artifact of TDL i.e. "Collection", provides extensive capabilities to gather data from various data sources. The TDL application developers are aware that the source can be report, parent report, collection/s and external data sources like excel, XML etc.

It is possible to gather the data from multiple collections in one collection. Till this enhancement collection didn't directly support the capability to gather data dynamically from multiple collections based on the object context of another collection

The functionality was achieved using Filter and Child Of attributes of 'Collection' definition. Programming using these was tedious, time consuming and increased the code complexity as well. The new enhancement has simplified the TDL code development.

The 'Collection' attribute of collection definition is enhanced to repeat and combine the same collection based on the number and context of objects in another collection.

The present syntax of Collection attribute allows us to combine and collate the data from all the collections specified as comma separated list provided the no, order and data type of methods are same in each of the collection specified in the list.

The existing syntax of 'Collection' attribute is as below :

Syntax

```
      [Collection : <Collection Name>]
           Collection  : <List of  Data Collection>
```

Where,

<Collection Name> is the name of collection.

<List of Data Collection> is comma separated list of data collections.

Example:

```
[Collection : GroupLedColl]
   Collection   : TestGroups, TestLedgers
```

In above example the Collection *'GroupLedColl'* will contain the union of objects from both the collection *'TestGroups'* and *'TestLedgers'*.

The Collection attribute is enhanced to dynamically gather the data collections in context of each object of another collection. The Collection attribute now accepts two additional optional parameters.

The collection attribute is enhanced as given below

Syntax

```
      [Collection : <Collection Name>]
            Collection : <List of Data Collection> [ : <Loop Coll Name>[:+
                         <Condition for Loop Coll]]
```

Where,

<Collection Name> is the name of collection.

<List of Data Collection> is comma separated list of data collections.

<Loop Coll Name> is the name of loop collection and it is optional.

<Condition for Loop Coll> is optional. The condition is evaluated on each object of the Loop Collection.

The attribute Collection is now enhanced to repeat a collection over objects of another collection. Optionally a condition can be specified. This condition applies to the collection on which looping is desired. This collection on which the repetition is done is referred as Loop Collection. The collection names in the list of collection are referred as Data Collections. The data is populated in the resultant collection from the list of data collection.

Each data collection is gathered once for each object in the loop collection. If there are n objects in the loop collection, the data collections are gathered n-times in the context of each object in loop collection.

In the resultant collection objects are delivered as TDL Objects .This makes it mandatory to fetch the required methods in the Data Collection.

Example:

```
[Collection: VchOfMultipleLedgers]
   Collection: VchOfLedger: LedgerListColl:
                ($Parent starting with "Sundry")

[Collection: VchOfLedger]
   Type       : Vouchers : Ledger
   Child of   : $Name
   Fetch      : VoucherNumber, Date, Amount
```

The collection VchofLedger is the data collection. It is mandatory to fetch the required methods Voucher Number, Data and Amount in order to be available in the resultant collection.

You must have observed that the method $name of loop collection LedgerListColl is used in ChildOf attribute directly. This is because while the evaluation of ChildOf attribute the loop collection object context is available. If we are referring to the methods of loop collection directly in the attributes SOURCE VAR, COMPUTE VAR, COMPUTE, AGGR COMPUTE, FILTER and FILTER VAR we cannot do so. This is because while evaluating these attributes loop collection object context is not available. In order to make these methods available in the Data collection the following function is introduced.

New Function – $$LoopCollObj

The function $$LoopCollObj is introduced to refer any method of Loop Collection objects in the Data Collection. The Data Collection can use this function for the evaluation of expressions.

Syntax

```
$$LoopCollObj: <Method name from Loop Coll Obj>
```

Where,

<Method name from Loop Coll Obj> is the name of method from the object of loop collection.

Example:

A collection is created to gather all vouchers of all the loaded companies as follows:

```
[Collection: Vouchers of Multiple Companies]
   Collection    : VchCollection: Company Collection
   Sort          : Default      : $Date, $LedgerName
```

The objects in the collection 'Vouchers of Multiple Companies' is sorted based on the date and ledger name.

```
[Collection: VchCollection]
   Type       : Voucher
   Fetch      : Date, Vouchernumber, VoucherTypeName, Amount, MasterID,+
                LedgerName
   Compute    : Owner Company: $$LoopCollObj:$Name
```

Let us examine the Data Collection definition "VchCollection"

When the attribute Compute is evaluated the Loop collection object context is not available here. So to refer to the Company Name , the function $$LoopCollObj is mandatory.

```
[Collection: Company Collection]
   Type        : Company
   Fetch       : Name
```

Use Case

Scenario: A Stock Summary Report for Group Company.

The consolidated stock summary report of all the members of a group company.

The member companies of the group company can have different Unit of Measures and Currency.
Solution: The report displays the stock item name, unit of measurement and the combined closing balance of all the members of group company assuming the base currency is same.

At part level the line is repeated on the aggregate/summary collection *'GrpCmpSSRepeatColl'* as below

```
[Part : GrpCmpSSPart]
   Line   : GrpCmpSSLineTitle, GrpCmpSSLineDetails
   Repeat : GrpCmpSSLineDetails : GrpCmpSSRepeatColl
   Scroll : Vertical
   Common Border: Yes
```

The summary collection is defined as follows

```
[Collection: GrpCmpSSRepeatColl]
   Source Collection: GrpCmpSSLoopCollection
   By               : StkName: $Name
   By               : UOM: $BaseUnits
   Aggr Compute     : ClBal: SUM: $ClosingBalance
   Sort             : Default: $StkName, $UOM
```

Since the member companies may have different UOM , the grouping is done on the same. If the UOMs are same then the ClosingBalance is aggregated else the Items are displayed as separate line items with respective UOMs

We cannot perform aggregation directly on the resultant collection(which is created using data and loop collection).If required to do so, the same has to be used as a source collection for the aggregate/summary collection.

The source collection *'GrpCmpSSLoopCollection'* is defined as follows:

```
[Collection : GrpCmpSSLoopCollection]
   Collection: StkColl: GrpCmpColl
```

The data collection '*StkColl'* is gathered for each object of the loop collection *'GrpCmpColl'*. The collections are defined as below

```
;; Data Collection
[Collection : StkColl]
   Type : Stock Item
   Fetch: Name,BaseUnits,ClosingBalance

;; Loop Collection
[Collection: GrpCmpColl]
   Type     : Member List: Company
   Child Of : ##SVCurrentCompany
```

Assume that currently a group company 'Grp ABC' is loaded having three member companies A, B, C. The Stock Items details in each company are shown in following table:

Company Name	Stock Item	Unit of Measure	Closing Balance
Company A	Item 1	Nos	500
	Item 2	Kg	500
	Item 3	Nos	500
Company B	Item 1	Nos	400
	Item 3	Nos	800
Company C	Item 1	Nos	300
	Item 2	Nos	700
	Item 3	Nos	500

The following table demonstrates the objects in each collection:

Objects in collection GrpCmpColl	Objects in collection StkColl	Objects in collection GrpCmpSSLoopCollection	Objects in collection GrpCmpSSRepeatColl
3 - A, B, C	All stock items of First member company i.e. A	All stock items of all member companies	The sum of Closing balance is evaluated by grouping Stock Item name and Unit Of Measure
	Item 1 - 500 Item 2 - 500 Item 3 - 500	tem 1 - 500 Item 2 - 500 Item 3 - 500 Item 1 - 400 Item 3 - 800 Item 1 - 300 Item 2 - 700 Item 3 - 500	Item 1 - 1200 Nos Item 2 - 500 Kg Item 2 - 700 Nos Item 3 - 1800 Nos

Multi Column behavior with Multi-Company data

Various reports can be generated in Tally.ERP 9 relevant to the users business requirement. All the reports are generated in the context of SVCurrentCompany, SVFromDate and SVToDate

In the multi column report, the collection is gathered for each column of the report.
The code complexity is reduced with the introduction of Loop Collection in TDL language.

When the Data collections are gathered in the context of Company as Loop collection, in the resultant collection the object context is forcefully changed to current/owner/loaded company and the report is displayed.

Consider the following example to understand the Loop Collection behavior of multi column report for multiple companies .

Assume that there are three companies A, B and C. The company A has ledgers L1 and L2, B has ledgers L3 and L4, C has ledgers L5 and L6. The currently loaded company is A and the loop collection "My Company" has objects as A, B and C.

The collection is constructed as follows:

```
[Collection : LedCmpColl]
   Collection : MyLed : My Company

[Collection : My Led]
   Type   : Ledger
   Fetch : $Name, $ClosingBalance

[Collection : MyCompany]
   Type   : Company
```

When the multi column report is displayed for the first time all the ledgers are associated to current company A forcefully and their closing balance is displayed in the column as follows:

When an additional column is added for company B, the report is displayed as follows:

Ledger Name	A Closing Balance
L1	100
L2	200
L3	300
L4	400
L5	500
L6	600

When an additional column is added for company B, the report is displayed as follows:

Ledger Name	**A Closing Balance**	**B Closing Balance**
L1	100	
L2	200	
L3		300
L4		400
L5		
L6		

For this column,the collection is gathered with the current company B in context.

As a result the closing balances of ledgers belonging to company A and B are available and are displayed in their respective company columns. As the company C context is not available, the closing balances of ledgers L5 and L6 are not displayed at all.

When the column for company C is added the closing balances of ledgers L1, L2, L3, L4, L5 and L6 are displayed in the respective company column as follows:

Ledger Name	**A Closing Balance**	**B Closing Balance**	**C Closing Balance**
L1	100		
L2	200		
L3		300	
L4		400	
L5			500
L6			600

Points for consideration while usage

- The collection attribute Search Key can be specified only in the Summary Collection and not in the Data/Loop/Source Collection
- The Summary Collection using source collection created using loop collection concept can only be referred from elsewhere using the function $$CollectionFieldByKey. The other functions like $$CollectionField, $$CollAmtTotal are at present not supported.
- If the companies have different currencies and aggregation is done then the resultant values for the masters would not be displayed.
- In case stock items of the companies have different unit of measures and aggregation is done then the stock item name having different UOM would not be displayed at all in the list.

Changes pertaining to Parameter Collection

The internal collection "Parameter Collection" was used earlier in TDL at two places.

1. When objects in the specified scope needs to be referred from a report or Child Report
2. XML Data response after triggering the action HTTP Post, which is used either in Success/Error Report displayed for the user.

As we have enhanced the Data Source attribute of the Collection to populate it using the objects in specified scope from a current or Parent Report, the concept of Parameter Collection in this respect does not carry relevance any more. We need to use Data Source capability instead of Parameter Collection in codes to be written in future. For the existing TDL's which are already using Parameter Collection we have introduced a collection in Default TDL which uses Data Source with scope as selected. The TDLs which are using Parameter Collection for different scope need to make desired changes in the code.

In context of HTTP Post the usage remains as it is.

8.3 User Defined Function Enhancements

The user defined function can use a newly introduce looping construct which iterates for the given range of values and the action New Object is enhanced to accept a logical value.

LOOPING Construct – FOR RANGE

As explained earlier TDL allows different looping construct for varied usage. The existing loop constructs allows to loop on the objects in collection or on the tokenized string or condition based looping of a set of statement.

There are scenarios where the looping is to be performed for a range of values, for this a new loop FOR RANGE is introduced.

The newly introduced loop construct allows to loop on range of numbers or date. This loop can be used to repeat a loop for the given range of specified values. The range can either be incremental or decremental. The FOR RANGE loop can be used to get the Period Collection like functionality.

Syntax

```
      FOR RANGE: <Iterator Var> : <Data type> :  <StartRangeExpr> :  +
                 <EndRangeExpr> [:<Increment Expr>[:<DateRangeKeyword>]]
```

Where,

<Iterator Var Name> is the name of variable used for the iteration. This variable is created implicitly.

<Data Type> can be Number or Date only.

<StartRangeExpr> is an expression which evaluates to number or date values. It refers to the starting value of the range.

<EndRangeExpr> is an expression which evaluates to number or date values. It refers to the end value of the range.

<Increment Expr> is an expression which evaluates to number by which the <StartRangeExpr> value is incremented. Its optional and the default value is one.

<DateRangeKeyword> is optional and only applicable if the data type is Date. The values can be any one of 'Day', 'Week', 'Month' and 'Year'.

Example:

```
   |
01: FOR RANGE : IteratorVar : Number : 2 : 10 : 2
02: LIST ADD : EvenNo : ##IteratorVar
03: END FOR
   |
```

The values 2,4,6,8,10 are added in the List variable 'EvenNo' as the range of value ia 2 to 10 and the value is incremented by two with each iteration.

Example:

The following code snippet is used to calculate the number of weeks elapsed between System date and Current Date set in Tally

```
   |
09: FOR RANGE:IteratorVar: Date: ##SVCurrentDate:$$MachineDate:1: "Week"
10: LOG: ##IteratorVar
20: INCREMENT :Cnt
30: END FOR
50: LOG : "No of weeks Back Dated :"  +  $$String: ##Cnt
60: RETURN: ##Cnt
   |
```

Assume that the range for this is from 15 - Mar - 2009 to 30 - Mar - 2009. The values 15-Mar-2009, 22-Mar-2009, 29-Mar-2009 are logged as the increment is done by a 'Week' so there are three iterations of the loop. The number of weeks is logged using the counter.

Example:

```
   |
09: FOR RANGE: IteratorVar:Date: ##SVFromDate:##SVToDate: 1 : "Month"
10: LOG: $$MonthEnd:##IteratorVar
20: END FOR
   |
```

Assume that the range for this is from 1 - Jan - 2009 to 5 - Mar - 2009. The values 31 - Jan -2009, 28 - Feb - 2009 and 31 -Mar -2009 are logged.

New Function – $$LoopIndex

The TDL programming community is now aware about the enhancements that are introduced in TDL language and are efficiently implementing the same in the programs.

A vast area of possible extensions is unlocked by the User Defined Functions in TDL. User defined function gave the sequential control to TDL programmers. Many actions and looping constructs are introduced in User Defined Function. During the sequential execution the loops are used to iterate through a set of values. TDL allows nested loops as well.

There are scenarios where the loop counter is required for some evaluation. Presently a variable is defined and then at the end of loop its value is incremented. This variable can be used for some calculations if required. To avoid this inline declaration of variable which is solely used as a counter, a new function $$Loop Index is introduced.

The function $$LoopIndex gives the count of how many times the current loop is executed. In case of nested loops, the additional parameter <outer loop index number> can be used in the inner loop to get the current iteration count of the outer loop.

Syntax

```
$$LoopIndex [:<Outer Loop Index Expr>]
```

Where,

<Outer Loop Index Expr> can be an expression which evaluates to number. It is optional and the outer loop index number in the nested loop hierarchy from inner most loop to the outer most loop. For the current loop the value is zero by default, for the parent loop One and so on.

Consider following example :

```
[Function : LoopIndex Test]
   |
   |
05 :  WALK COLLECTION :  ………
   |
     WHILE : …….
        |
        |
     FOR : ……….
        SET : Var: $$LoopIndex
        LOG : ##Var
           |
     END FOR
     SET : Var1: $$LoopIndex:1
     |
   END WHILE
     |
     |
END WALK
```

The variable Var will hold the count of number of times the FOR loop is executed while the variable Var1 will have the count of WALK Collection loop execution.

Enhanced Action NEW OBJECT

The action New Object takes two parameters Object Type and Object Identifier.

The current syntax is:

Syntax

```
      NEW OBJECT : <Object Type> : [:<Object Identifier>]
```

Where,
<Object Type> is the type of the object to be created,
<Object Identifier> is the unique identifier of the object.

The Object Identifier is optional. In case this is not specified it creates a blank object of the specified type.

The actions Save Target/Alter Target/Create Target are used along with New Object for specific usage. There are three scenarios to consider for this

1. In case a Blank Object is created using New Object without specifying the Object Identifier, the action Save Target and Create Target will work and Alter Target would fail.
2. In case an existing object is brought into context by specifying Object Identifier with New Object, the action Save Target and Alter Target will work and Create Target would fail.

Save Target saves the current Object whether it is a blank new Object or an existing Object for Alteration.

3. When an Object Identifier is specified with New Object and the object does not exist in the database the Action Save Target fails as New Object does not create a blank object.

In order to overcome the scenario (3) the Action New Object has been enhanced to accept an additional parameter Force Create Flag along with Object Identifier. This forces the creation of an empty blank object in case the Object with Identifier does not exist in the database.

Syntax

```
      NEW OBJECT: <Object Type> : [:<Object Identifier>[:<Forced Create Flag>]]
```

Where,
<Object Type> is the type of the object to be created,
<Object Identifier> is the unique identifier of the object.
<Forced Create Flag> this flag is optional and it is required only if the <Object Identifier> is specified. If the Flag is set to TRUE then if the object identified by <Object Identifier> exists, the object is altered otherwise a new empty object of the specified type is created.

Example:

```
NEW OBJECT: Group   : ##EGroupName : TRUE
SET VALUE: Name   : ##NGroupName
```

```
    03 : SAVE TARGET
          |
```

If the ledger identified by *'##EGroupName'* exists in tally database then the objects is altered by action SAVE TARGET else a new object is created as the Forced flag is set to 'YES'.

8.4 New Functions

A new function SysInfo is introduced to get any system related information.

Function SysInfo

Platform has provided TDL Programmers with various functions that accept zero or more parameters, processes and returns the appropriate result. Apart from the Object/ Data Manipulation Functions, there are many system related information that are required to be retrieved.

Functions like MachineDate which returns System Date, MachineTime which returns System Time, etc. are supported by the platform today. Few more system related functions like Machine Name, Windows User Name, IP Address, etc. are required by TDL Programmers. Such system related information is bundled together into a single function SysInfo which is designed to accept different parameters based on the requirement and subsequently return the desired result.

Syntax

```
$$SysInfo:<Parameter>
```

Where,
<Parameter> can be any one of the ApplicationPath, CurrentPath, SystemDate, SytemTime, SystemTimeHMS, SystemName, IsWindows, WindowsVersion, WindowsUser, IPAddress, MACAddress

Example:

```
$$SysInfo:MachineName
```

The above will return the Machine Name in which current copy of Tally is running.

Example of each parameter is explained considering the following system details:

Application Path is **C:\Tally.ERP9**
Data Path is **C:\Tally.ERP9\Data**
System Date is **27-Sep-2009**
System Time is **18:27**
System Time in Hours, Minutes and Seconds is **18:27:36**
System Name is **TallySystem1**
Operating System is **Windows 7 / Windows XP / Windows 2000 / Windows Vista**
Version of Windows is **5.1 (2600)**
User logged into Windows is **Tally.User1**
Network IP Address is **192.168.1.17**
Network Adapter's MAC Address is **0720fhac027a**

List of Parameters with corresponding Result

ApplicationPath – Returns the Folder path from where the current copy of Tally is executed.
Example:

```
$$SysInfo:ApplicationPath returns C:\Tally.ERP9
```

CurrentPath – Returns the Data path which is configured in Tally.INI residing within the application path.
Example:

```
$$SysInfo:CurrentPath returns C:\Tally.ERP9\Data
```

SystemDate – Returns the current System/ Machine Date.
Example:

```
$$SysInfo:SystemDate returns 27-Sep-2009
```

SystemTime – Returns the current System/ Machine Time.
Example:

```
$$SysInfo:SystemTime returns 18:27
```

SystemTimeHMS – Returns the current System/ Machine Time in Hour Minute Second Format.
Example:

```
$$SysInfo:SystemTimeHMS returns 18:27:36
```

SystemName – Returns the current System/ Machine Name.
Example:

```
$$SysInfo:SystemName returns TallySystem1
```

IsWindows – Returns Yes only if the current Operating System is Windows else returns No.
Example:

```
$$SysInfo:IsWindows returns Yes
```

WindowsVersion – Returns the current Windows Version with the Build Number.
Example:

```
$$SysInfo:WindowsVersion returns 5.1 (2600)
```

WindowsUser – Returns the Name of the User who has logged into the current Windows session.
Example:

```
$$SysInfo:WindowsUser returns Tally.User1
```

IPAddress – Returns the Network IP Address of the current system.
Example:

```
$$SysInfo:IPAddress returns 192.168.1.17
```

MACAddress – Returns the Network Adapter's Media Access Control Address of the current system.
Example:

```
$$SysInfo:MACAddress - 0720fhac027a
```

Corresponding Functions *ApplicationPath, CurrentPath, MachineDate, MachineTime, IsWindows, WindowsVersion, WindowsUser, IPAddress* and *MACAddress* are alternative functions available in default TDL. These functions are deprecated from platform.

9. Enhancements in Release 1.52

In this release the major enhancements have taken place at the collection level and in the User Defined Functions.

This document talks in depth on the usage of Data Source attribute in collection and the various looping constructs inside a function.

Few generic built -in functions, $$AccessObj, $$FirstObj and $$LastObj are introduced.

Https client capability is enhanced in Tally to exchange data with other application securely. Https sites can be used for ftp upload, post request and receiving the data in collection.

Bug fixes and the enhancements in the previous release can be referred from here as well.

9.1 Collection Enhancements

The TDL programmers are aware that data from various data sources can be gathered in a collection. Till the release 1.5 the data sources were Tally database, XML, HTTP, ODBC and DLL.
After the multi-line selection capability was introduced , a report or function can be launched from the current report based on the specified scope. The different scopes that can be specified are selected lines, current line, unselected lines etc.
Now the objects can also be gathered from the report or parent report using the Type for collection attribute Data Source. This new capability now allows the access of specific objects of a report from anywhere like functions, subsequent report or the current report itself.

The Data Source attribute of the collection is enhanced to support these two data sources in addition to the existing data sources. The collection can be created directly from the specified data source and can be displayed in a report.

Syntax

```
Data Source : <Type> : <Identity> [: <Encoding> ]
```

Where,
<Type> specifies the type of data source. File Xml, HTTP XML, Report, Parent Report.

<Identity> can be file path or the scope keywords. If the type is File XML or HTTP XML, <*identity*> is the data source file path. If the type is Report or Parent Report then the scope keywords '*Selected Lines*', '*UnSelected Lines*', '*Current Line*', '*All Lines*', '*Line*' and '*Sorting Methods*' are used as identity.

<Encoding> can be ASCII or UNICODE. This is Optional. The default value is UNICODE. If the data source type is Report or Parent report the encoding format is ignored.

Existing Data Source Types

Example 1: XML file as data source

```
[Collection : My XML Coll]
   Data Source : File XML : "C:\MyFile.xml"
```

In the above code snippet the type of file is *'File XML'* as the data source is XML file. The encoding is Unicode by default as it is not specified.

Example 2: HTTP as data source

```
[Collection : My XML Coll]
   Data Source : HTTP XML : "http:\\localhost\MyFile.xml": ASCII
```

In the above code snippet the type of file is '*HTTP XML*' as the data source is obtained through HTTP. The encoding of the file '*MyFile.XML*' is ASCII.

While specifying the URL, now the **https** site can be given in collection attributes Remote URL and Data Source.

Data Source Types Introduced

Example 1: Report as data source

The selected objects from the current report in which the collection is accessed is the data source for the collection '*MY Report Coll*'.

```
[Collection : My Report Coll]
   Data Source : Report : Selected Lines
```

Example 2: Parent Report as data source

```
[Collection : My Parent RepColl2]
   Data Source : Parent Report : UnSelected Lines
```

The objects associated with all the unselected lines from the parent report are gathered for the collection '*My Parent RepColl2*'.

The objects of the report with the given scope can be accessed from the report and functions which are called from the report.

9.2 Enhancements in User Defined Function

With every release, new functionalities and capabilities are added to user defined functions. In this release, dynamic action support is provided for simple actions inside a function and the looping construct Walk Collection is enhanced.

New looping constructs FOR COLLECTION and FOR TOKEN are introduced as well.
There are scenarios when the collection name is to be obtained from an expression while performing a Walk. Walk collection is enhanced to provide this functionality.

FOR COLLECTION loop introduced to walk the collection for a specific value. FOR TOKEN loop walks on the tokens within a string separated by a specified character.

Walk Collection

The Walk Collection is enhanced to accept Collection Name as an expression and an additional logical parameter. For example, now the collection name can be passed as parameter to the function while executing it.

Syntax

```
      Walk Collection : <Expression> [: <Rev Flag> ]
```

Where,
<Expression> can be any expression which evaluates to collection name.
<Rev Flag> can be an expression which evaluates to logical value. If it is True then the collection objects are traversed in reverse order. This parameter is optional. The Default value is False.

Example:

```
[Function : Test Function]
   Parameter : parmcoll
           |
           |
   05 : WALK COLLECTION : ##parmColl : Yes
```

The collection name is passed as a parameter to the function ‘*Test function*’ and it is walked in reverse order.

The code snippet to call the function ‘*Test function*’ from a key is as shown:

```
[Key: DC Call Function]
   Key : Enter
   Action : CALL : Test Function :##CollName
```

The collection name is passed through the variable ‘CollName’.

Dynamic Actions

Prior to Release 1.52, the dynamic action capability was available for global actions. It was possible to specify the Action Keyword and Action parameters as expressions. This allows the programmer to execute actions based on dynamic evaluation of parameters. The Action keyword can as well be evaluated dynamically.

The dynamic action capability is now introduced for simple actions inside a function. Expressions can be used to evaluate the name of the Action as well as Action parameters. The new action with Action keyword has been introduced to achieve this.

Syntax

```
      Action : <Expression> : <Action Parameters>
```

Where,

<Expression> can be any expression which evaluates to simple action name like LOG, DISPLAY etc.

<Action Parameters> are the parameters required by the specific action passed through an expression.

Example:

```
[Function : ObjFunc]
          |
          |
   02 : ACTION : LOG : "$" + ##pObjMethod
```

Inside a function, a global action can be called using dynamic action capability. In this case, the expression specified in the dynamic action is evaluated in the context of the function and then the global action is executed.

The context of function elements like Variables, Objects etc can be used while calling a global action dynamically. For example the variable name or methods of an object can be passed as an parameter while executing dynamic action.

Example:

```
[Function: Dynamic Action Within Function]
   Variable : DA Logical: Logical
          |
          |
   40: SET    : DA Logical : Yes
   50: ACTION : Display : if ##DALogical then "Trial Balance" else +
                          "Balance Sheet"
```

In function '*Dynamic Action Within Function*' first the expression is evaluated and then based on the value, the report '*Trial Balance*' is displayed.

Looping Constructs Introduced

Two new looping constructs, FOR COLLECTION and FOR TOKEN are introduced in user defined function .

For Collection

When a WALK COLLECTION is used inside a function, the object of collection is set as the current object in the context of iteration. i.e. the loop is executed for each object in the collection, making it as the current context.

The newly introduced FOR COLLECTION provides a context free walk as the current object is not set as current object context while looping. It loops on the collection for the specific value and returns the value in the iterator variable. The value of iterator variable can be referred by the actions inside the For Collection loop.

Syntax

```
      FOR COLLECTION:<IteratorVar>:<CollExprn>[:<Value Exprn: <Rev Flag> ]
```

Where,

<Iterator Var> is the name of variable user for the iteration. This variable is created implicitly.

<Coll Exprn> can be any expression which evaluates to collection name.

<Value Exprn> can be an expression and the value of this is returned in the iterator variable. If the value expression is not specified the name of the object is returned.

<Rev Flag> can be an expression which evaluates to logical value. If it is True then the collection objects are traversed in reverse order. This parameter is optional. The Default value is False.

Example:

```
[Function : Test Function]
           |
           |
   30: FOR COLLECTION: i : Group: $ClosingBalance > 1000
   40: LOG: ##i
   50: END FOR
```

The value Yes is logged in the file ‘TDLFunc.log’ if closing balance is greater than 1000 else No.

For Token

FOR TOKEN is used to walk a String expression separated by a delimiter character. It is used to loop on a String expression and returns one value at a time. The value is returned in the iterator variable.

Syntax

```
            FOR TOKEN:<IteratorVar>:<SrcStringExprn>[:<DelimiterChar>]
```

Where,

<Iterator Var Name> is the name of variable user for the iteration. This variable is created implicitly.

<Src String Exprn> can be any string expression separated by a *<delimiter Char>*.

<Delimiter Char> can be any expression which evaluates into a character used for separating string expression. It is optional. The default separator char is ':'.

Example:

```
[Function : Test Function]
          |
          |
   01: FOR TOKEN : TokenVar : "Tally:Shopper:Tally Developer" : ":"
   02: LOG : ##TokenVar
   03: END FOR
```

The above code snippet will give the output as shown below

```
Tally
Shopper
Tally Developer
```

For Each

The new alias is introduced for the FOR IN looping construct.

9.3 New Functions

New functions AccessObj, FirstObj and LastObj are introduced in this release.

Function $$AccessObj

The capability to access data objects associated with Interface objects was introduced in Tally.ERP9. The attribute 'Access Name' is used to specify name to Part or Line Definition. This name can be used to refer the Data Object associated with the Part or Line.

A new function $$AccessObj is introduced to evaluate the specified formula in the context of Interface object identified by the given definition type and access name.

Syntax

```
      $$AccessObj:<Definition Type>:<AccessNameFormula>:<Evaluation Formula>
```

Where,

<Definition Type> can be Part or Line.

<Access Name Formula> can be a formula which evaluates to string.

<Evaluation Formula> is a formula which is evaluated in the context of object identified by definition type and access name.

Example:

```
[Line : AccessObj]
   Fields: AccessObj
   AccessName: "AO1"
   On: Focus: Yes: CALL: AccessObj

   [Field : AccessObj]
     Set As : $Name

[Function: AccessObj]
   Variable: AccessObj: String
   00: SET : AccessObj : $$AccessObj:Line:"AO1":$Name
   10: LOG : ##AccessObj
```

The Line 'AccessObj' is identified by the access name 'AO1'. The access name is used while evaluating the value of $Name.

Function $$FirstObj and $$LastObj

The objects of the collection are available in the context of repeat line or while performing a walk inside a function. The functions $$FirstObj and $$LastObj can be used to find the first or last object of the collection respectively.

Function $$FirstObj

The function $$FirstObj returns the value of specified method for first object of the collection.

Syntax

```
      $$FirstObj:<MethodName>
```

Where,

<Method Name> is the name of a method of the current object in context.

Example:

```
   40 : LOG : "First Object: " + $$FirstObj:$Name
```

The function $$FirstObj logs name of the first object of the collection which is used in the Walk Collection.

Function $$LastObj

The function $$LastObj returns the value of specified method for last object of the collection.

Syntax

```
        $$LastObj:<MethodName>
```

Where,

<Method Name> is the name of a method of the current object in context.

Example:

```
      50 : LOG : "Last Object :" + $$LastObj:$Name
```

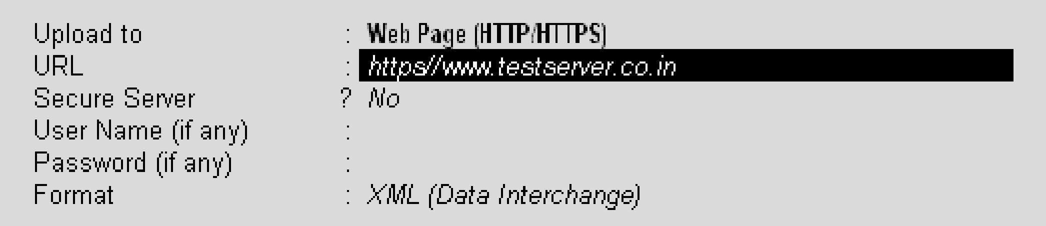
The function $$LastObj logs name of the last object of the collection which is used in the Walk Collection.

9.4 https URL support in Tally

Https client capability is enhanced in Tally to exchange data with other application securely. Https sites can be used for ftp upload, post request and receiving the data in collection. Now data can be uploaded to https site or request to the https site can be sent using action HTTP Post. URL for https site can be specified while gathering data in a collection.

Example 1: Upload

In the upload configuration screen the URL for https site can be given as shown:



```
Upload to              : Web Page (HTTP/HTTPS)
URL                    : https//www.testserver.co.in
Secure Server          ? No
User Name (if any)     :
Password (if any)      :
Format                 : XML (Data Interchange)
```

Figure 1.46 Upload Configuration Screen

Example 2: Action HTTP Post

```
[Button: PostButton]
   Key    : Ctrl+K
   Action : HTTP Post : @@MyURL: ASCII: HTTP Post ReqRep : +
            HTTP Post Response Report1  : HTTP Post Response Report

[System: Formula]
   MyURL  :  "https//www.testserver.co.in/CXMLResponse as per tally.php"
```

Example 3: In collection

```
[Collection : https Coll]
   Remote URL : "https//www.testserver.co.in/TestXML.xml"
```

10. Enhancements in Release 1.5

Following are the enhancements in the previous release:

10.1 Collection Enhancements

TDL supports the hierarchical data base structure. While designing any report the objects are first populated in collection before displaying them.

TDL also supports the concept of aggregate/summary collection for creating summarized reports. In the aggregate collection during the evaluation following three sets of objects are available:

- **Source Objects** : Objects of the collection specified in the Source Collection attribute.
- **Current Objects** : Objects of the collection till which the Walk is mentioned
- **Aggregate Objects:** Objects obtained after performing the grouping and aggregation

There are scenarios where some calculation is to be evaluated based on the source object or the current object value and the filtration is done based on the value evaluated with respect to final objects before populating the collection. In these cases to evaluate value based on the changing object context is tiresome and some times impossible as well.

The newly introduced concept of collection level variables provides Object-Context Free processing. The values of these inline variables are evaluated before populating the collection. The sequence of evaluation of the collection attributes is changed to support the attributes Compute Var, Source Var and Filter Var. The variables defined using attributes Source Var and Compute Var can be referred in the collection attributes By, Aggr Compute and Compute. The variable defined by Filter Var can be referred in the collection attribute Filter.

The value of these variables can be accessed from anywhere while evaluating the current collection objects.

Sometimes it is not possible to get a value of the object from the current object context, in such scenarios these variables are used.

Source Var

The attribute Source Var evaluates the value of based on the source object.

Syntax

```
Source Var : <Variable Name> : <Data Type> : <Formula>
```

Where,

<Variable Name> is name of variable.

<Data Type> is the data type of the variable.

<Formula> can be an expression formula which evaluates to value of the variable data type.

Example:

```
Source Var : Log Var: Logical  : No
```

The value of the LogVar variable is set to NO

Compute Var

The attribute Compute Var evaluates the value of based on the sub object of the source object.

Syntax

```
      Compute Var : <Variable Name> : <Data Type> : <Formula>
```

Where,
<Variable Name> is name of variable.
<Data Type> is the data type of the variable.
<Formula> can be an expression formula which evaluates to value of the variable data type.

Example:

```
   Compute Var:IName : String : if ##LogVar then $StockItemName else +
                                 ##LogVar
```

Filter Var

The attribute Filter Var evaluates the value of based on the objects available in collection after the evaluation of attributes Fetch and Compute.

Syntax

```
      Filter Var : MyFilVar : <Data Type> : <Formula>
```

Where,
<Variable Name> is name of variable.
<Data Type> is the data type of the variable.
<Formula> can be an expression formula which evaluates to value of the variable data type.

Example:

```
   Filter Var : Fin Obj Var : Logical : $$Number:$BilledQty > 100
```

Sequence of Evaluation of Collection Attributes

The collection attributes are evaluated as per the following sequence before populating the collection :

1. Source collection
2. Source Var
3. Walk
4. Compute Var
5. By
6. Aggr Compute
7. Compute
8. Filter Var
9. Filter

With the introduction of these attributes, the calls to function $$Owner, $$ReqObject, $$FilterValue, $$FilterAmtTotal etc can be reduced.

Usage of the collection attributes Compute Var, Source Var, Filter Var

In this section the use cases where the collection attributes can be used are explained.

Usage of Compute Var in Simple Collection

When Compute Var is used in a simple collection then before populating the objects in collection the Compute var is evaluated.

Consider the following Collection Definition :

```
[Collection : Test ComVar ]

   Type        : Group

   Compute Var : CmpVarColl : String : $Name

   Compute     : MyAmt : $$CollamtTotal: TestComVarSub:$OpeningBalance


[Collection : Test ComVar Sub]

   Type        : Ledger

   Child Of    : ##CmpVarColl

   Fetch       : Name, OpeningBalance
```

The sequence of evaluation is as follows :

1. The Type attribute is evaluated and the objects of specified type are identified.
2. Compute Var is evaluated and the name of first object i.e group name is set as a value of the variable CmpVarColl.
3. For this selected group, the method $MyAmt is evaluated. This gives the total amount of all the ledgers belonging to the group name in the variable CmpVarColl.
4. The steps 2, 3 are repeated for each group in the collection ‘Test Com Var’.
5. After computing the value method $MyAmt for each group the collection is populated with the objects.

The variable CmpVarColl can be referred also in the By, Aggr Compute and Filter attributes of collection.

Usage of Source Var, Compute Var, Filter Var in Aggregate collection

When these collection attributes are used along with other attributes the sequence of evaluation is as mentioned earlier. Let us try to understand this with the following collection definition:

```
[Collection : CFBK Voucher]

   Type              : Voucher

   Filter            : IsSalesVT

   Compute Var       : Src Var: Logical: $$IsSales:$VoucherTypeName


[Collection : Smp Stock Item]

   Source Collection : CFBK Voucher

   Source Var        : Str Var: String : $VoucherNumber +

                       "/" $VoucherTypeName
   Walk                  : Inventory Entries
   Compute Var           : IName : String : if ##StrVar CONTAINS "12" then +
                           $StockItemName else $StockItemName +
                           "-" + $$String:##StrVar
   By                    : IName: ##IName
   Aggr Compute          : BilledQty: SUM: $BilledQty
   Filter Var            : Fin Obj Var: Logical : $$Number:$BilledQty > 100
   Filter                : Final Filter

[System : Formula]
   IsSalesVT             : ##SrcVar
   Final Filter          : ##FinObjVar
```

The evaluation process is as follows:

1. The value of variable SrcVar is evaluated and referred in the Filter attribute of collection CFBK Voucher.
2. In the collection Smp Stock Item, the value of variable Str Var is evaluated on the first object of source collection CFBK Voucher.
3. Then Walk is performed and the Inventory entry objects are collected.
4. The value of variable IName is evaluated. If the Source Variable Src Var contains "12", then IName Variable stores only Stock Item Name method else Stock Item Name + value of the variable Str Var.
5. The grouping is done on the resultant value of IName variable.
6. The value of $BilledQty is computed.
7. The variable FinObjVar retains a logical value if method BilledQty is greater than 100
8. Based on the value of FinObjVar the filtering is done.
9. Finally the collection is populated with the filtered objects.

10.2 List Variables Introduced

The TDL programmer community is aware of the usage of variables and its usage as context free structures in TDL. Till this release two types of variables were supported "Simple and Repeat". The following scope can be defined for variables

- Report Level – commonly referred to as Local Variable
- System Level – commonly referred as Global variables
- Function Level – Local variables used inside user defined functions

The variable framework is enhanced to support a new type of a variable called List Variable which allows us to perform complex calculations on data available from multiple objects.

List Variable

List variable can store multiple values of single data type in the key: value format. Every single value in the List variable is uniquely identified by a 'key'. The 'Key' is of type String by default and is maintained internally.

List Var is alias of attribute List Variable.

Syntax

```
      List Variable : <Variable Name> [ : <Data Type>]
```

Where,

<Variable Name> is name of variable.

<Data Type> is the data type of the variable. It is Optional. If data type is specified a separate Variable definition is not required. Incase the data type is not specified a variable definition with the same name must be specified.

Example:

```
[Function : Test Function]
   ListVariable : List Var :  String
```

The variable *'List Var'* can hold multiple string values.

Example:

```
[Report : Test Report]
   ListVariable : List Var Rep :  String
```

The variable *'List Var Rep'* can hold multiple strings in the report scope.

The List variable provides a set of actions and internal function for the data manipulation which will be explained in the following section.

List Variable Manipulation

List variable support various data manipulation operations. Following operations can be performed on List variable:

1. Adding/Deleting Values
2. Populating List Var from a Collection – Action ListFill
3. Accessing Values using Function $$ListValue
4. Sorting the value in List Var

Adding/Deleting values in a List Variable

The actions used to add/delete values in the list variable are LIST ADD and LIST DELETE.

- ***Action LIST ADD***

The action LIST ADD is used to add the value in a List variable. The action LIST ADD adds single value at a time to the list variable identified by a key. If the value is added to the list with duplicate key, then the existing value is over written. LIST SET is alias for action LIST ADD.

Syntax

```
       LIST ADD : <List Var Name>  : <Key Formula> : <Value Formula>
```

Where,
<List Var Name> is the name of list variable.
<Key Formula> can be an expression formula which evaluates to unique string value.
<Value Formula> can be any expression formula which returns a value. The data type of the value must be same as that of List variable.

Example:

```
LIST ADD : TestFuncVar  : "Mobile"    : 9340193401
LIST ADD : TestFuncVar  : "Office"    : 08066282559
LIST ADD : TestFuncVar  : "Fax"       : 08041508775
LIST ADD : TestFuncVar  : "Residence" : 08026662666
```

The four values inserted in the list variable *'Test Func Var'* are identified by the key values *'Mobile'*, *'Office'*, *'Fax'* and *'Residence'* respectively.

Notes *The same List is considered in explaining the further examples.*

To add multiple values dynamically in the list variable, looping constructs WHILE, WALK COLLECTION etc can be used. LIST REMOVE is an alias for LIST DELETE.

Action LIST DELETE

The action LIST DELETE is used to delete values from the List variable. The action LIST DELETE allows to delete single value at a time or all the values in one go.

Syntax

```
LIST DELETE : <List Var Name> [ : <Key Formula>]
```

Where,
<List Var Name> is the name of list variable.
<Key Formula> can be an expression formula which evaluates to unique string value. In the absence of key formula, all the values in the list will be deleted. In other words, if key formula is omitted, it resets the list.

Example:

```
LIST DELETE : TestFuncVar  : "Office"
```

The value identified by key 'Office' is deleted from the list variable 'Test Func Var'.

```
LIST DELETE : TestFuncVar
```

All the values in the list variable 'Test func Var' are removed. The list variable is empty after the execution on the above action.

Populating List variable from a collection

Instead of using the looping constructs, multiple values from a collection can be added to the list variable in one statement. Action LIST FILL is used in this case.

Syntax

```
      LIST FILL : <List Var Name>  : <Collection Name : <Key Formula> +
                   : <Value Formula>
```

Where,

<List Var Name> is the name of list variable.

<Collection Name> is the name of collection from which the values are fetched to fill the list variable.

<Key Formula> can be an expression formula which evaluates to string value.

<Value Formula> can be any expression formula which returns a value. The data type of the value must be same as that of List variable.

The action LIST FILL returns the number of items added to the list variable.

Example:

```
    LIST FILL : TestFuncVar  : Group:$Name:$Name
```

Accessing List variable values

To access the value from a list variable a function is to be used. TDL provides the different functions to fetch the value from list variable identified by the given key.

- ***Function ListValue***

$$ListValue gives the value identified by the given key in the list variable.

Syntax

```
      $$ListValue:<List Var Name>:<Key Formula>
```

Where,

<List Var Name> is the name of list variable.

<Key Formula> can be an expression formula which evaluates to string value.

Example:

```
   $$ListValue:TestFuncVar:"Mobile"
```

The above function returns the values identified by the key 'Mobile' from the list variable 'Test func Var' when the function is executed.

Sorting value in a List variable

By default the values in the list variable sorted in the order of entry. TDL provides the facility to sort the values in the list variable either on key or value. The data type can be specified while sorting on key. Following action allows to change the sort order:

- List Key Sort
- List Value Sort
- List Reset Sort

These actions accept three parameters. First parameter is name of the List variable followed by the sorting flag and a key data type.

In the absence of <key data type> natural sorting method is used. In natural sorting method, the key data type is identified as one of the data types Date, Number and String.

Date data type accepts any valid date format. If it is not of data type and starts with a number or a decimal then it is assumed as Number. If it's neither Date nor Number then it's considered as String.

Different data types are compared in the following order as Number, Date and String.

Action LIST KEY SORT

This action allows sorting the list on key value. If the data type specified while sorting the list is different than the original, then this action will temporarily convert the original data type to the specified data type while comparing the elements for sorting the list and the list will be sorted based on the new data type specified. The original list and the key data type remains as it is on which a new sorting can be applied based on some other data type at any other point of time. LIST SORT is an alias of action LIST KEY SORT.

Syntax

```
LIST KEY SORT : <List Var Name>[: <Asc/Desc flag> : <Key Data Type>]
```

Where,

<List Var Name> is the name of list variable.

<Asc/Desc> can be YES/NO. YES is used sort the list in ascending order and NO for descending. If the flag is not specified then the default order is ascending.

<Key Data Type> can be String, Number etc. It's optional.

Example 1:

```
LIST KEY SORT : Test Func Var: YES : String
```

The values in the list variable are sorted in ascending order of the key.

Example 2:

In case a different data type is used for sorting then the key may become duplicate if the conversion fails as per the data type specified for sorting. If the key becomes duplicate then the insertion order of items in list variable is used for comparison.

```
LIST KEY SORT : Test Func Var: YES : Number
```

The action will convert the key to ZERO (0) for all the list items while comparing, as all the keys are of type Strings. In this case the insertion order will be considered for sorting. As a result the values in the list will be sorted in the following order: 9340193401, 08066282559, 08041508775, and 08026662666

In case the key contains numeric values like "11", "30", "35" and "20" which can be converted to number, then the list is sorted based on the key values else it converts them to ZERO and sorts the list as per order of insertion.

▫ *Action LIST VALUE SORT*

The action LIST VALUES SORT sorts the list items based on the value. As there can be duplicate values in the list the combination of key and value is considered as key for sorting duplicate values.

Syntax

```
LIST VALUE  SORT : <List Var Name>[: <Asc/Desc flag> : <Key Data Type>]
```

Where,
<List Var Name> is the name of list variable.
<Asc/Desc> can be YES/NO. YES is used sort the list in ascending order and NO for descending. If the flag is not specified then the default order is ascending.
<Key Data Type> can be String, Number etc. It's optional.

Example:

```
LIST VALUE SORT : Test Func Var: YES : String
```

The values in the list variable are sorted in ascending order of values.

▫ *Action LIST RESET SORT*

The action LIST RESET SORT retains the sorting back to the order of insertion.

Syntax

```
LIST RESET SORT: <List Var Name>
```

Where,
<List Var Name> is the name of list variable.

Example:

```
LIST RESET  SORT : Test Func Var
```

The action resets the sort order of the list variable 'Test func Var' to the order of insertion.

Functions Used with List Variables

TDL supports some function for the general operation like finding the total number of items in a list, checking whether the last action was successful etc.

Function ListValue

As explained earlier the function List Value to access the value from a list variable

Function ListCount

$$ListCount gives the total number of values available in the list variable.

Syntax

```
$$ListCount:<List Var Name>
```

Where,
<List Var Name> is the name of list variable.

Example:

```
$$ListCount:TestFuncVar
```

The above action returns the number of items in the list variable 'Test func Var' when the action is executed.

Function ListFind

The function ListFind is used to search if the value belonging to a specific key is available in the list variable. If the key is found $$ListFind returns TRUE otherwise it returns FALSE.

Syntax

```
$$ListFind:<List Var Name>:<Key Formula>
```

Where,
<List Var Name> is the name of list variable.
<Key Formula> can be an expression formula which evaluates to string value.

Example:

```
$$ListFind:TestFuncVar:"Mobile"
```

It returns either TRUE if the key **'Mobile'** is available in the list variable **'Test func Var'** or FALSE if the key is not available.

The function LastResult can be used to check whether the last excecuted action was successful.

- □ *If the last action that is executed is LIST ADD or LIST DELETE then the function returns TRUE if the action was successful and FALSE otherwise.*
- □ *If the last action that is executed is LIST FILL then the $$LastResult returns the number of items inserted in the list variable.*

Constructs introduced in functions for List Var

The FOR IN loop is supported to iterate the values in the list variable. The number of iteration depends on the number items in the list variable.

Syntax

```
FOR IN : < Iterator Var Name>: <List Var Name>
  .
  .
  .
END FOR
```

Where,
<Iterator Var Name> is the name of variable user for the iteration. This variable is created implicitly.
<List Var Name> is the name of list variable.

Example:

```
FOR IN : Cnt : Test Func Var
         LOG :  $$String:$$ListValue:TestFuncVar:##Cnt
END FOR
```

All the values of the list variable *'Test Func Var'* are logged in the file *'tdlfunc.txt'*.

10.3 Dynamic Actions

A new capability has been introduced with respect to Action framework where it is possible to specify the Action Keyword and Action parameters as expressions. This allows the programmer to execute actions based on dynamic evaluation of parameters. The Action keyword can as well be evaluated dynamically. Normally this would be useful for specifying condition based action specification in menu, key / button etc.In case of functions, as the function inherently supports condition based actions via IF ELSE etc, this would be useful when one required to write a generic function, which takes a parameter and later passes that to an action (as its parameter) which does not allow expressions and expects a constant.

This has been achieved with the introduction of a new keyword "Action" .The syntax for specifying the same is as given below.

Syntax

```
Action :<Action Keyword Expression>: <Action Parameter Expression>
```

Where,
<Action> is the keyword "Action" to be used for Dynamic Actions usage
<Action Keyword Expression> is an expression evaluating to an Action Keyword
<Action Parameter Expression> is an expression evaluating to Action Parameters

We can specify and initiate an Action from the following

- Menu Item
- Key Definition
- In a User Defined Function

At present the capability is valid for

- Global Actions like Display, Alter etc
- Global Actions inside User Defined Functions

Example:

1. **Dynamic Actions in Key/Button Definition**

```
[Button: Test Button]
   Key    : F6
   Action : Action : Display : @@MyFor

;; The Button Test Button initiates a dynamic Action which takes the parameter as a formula
[System : Formula]
   MyFor  : if ##SVCurrentCompany CONTAINS "ABC" Then "BalanceSheet" +
            else "TrialBalance"
```

Observe the usage of Action keyword twice in this. The first usage is the attribute "Action" for key definition. The second is the keyword "Action" introduced specifically for executing Dynamic Actions.

2. Dynamic Actions in User Defined Functions

```
[Button: Test Button]
   Key         : F6
   Action      :Call:TestFunc:"Balance Sheet"

[Function: Test Func]
   Parameter   : Test Func : String
;;The function Test Func executes a dynamic action which takes Action parameter as the parameter passed
to the function
   01 : Action : Display : ##TestFunc
```

10.4 New Functions

In this release two new functions are introduced - $$TgtObject ans $$ContextKeyword.

Function – $$TgtObject

In TDL normally all evaluation is done in the context of the Context object. With the introduction of aggregate collection and user defined function, apart from the requestor object and source object, now the target object context is also available.

The object which is being populated or altered is referred as the Target object. In simple collection, the source object and target object both are same. In case of the aggregate collection and user defined functions, the target object is different.

There are scenarios where the expression needs to be evaluated in the context of Target object, in such cases the $$TgtObject can be used.

A New Context Evaluation function $$TgtObject evaluates the expression in the context of the Target Object. Using the $$TgtObject values can be fetched from the target object without making the target object as the context object.

Syntax

```
$$TgtObject:<String Expression>
```

Where,
<String Expression> the expression and will be evaluated in the context of Target Object

Usage of $$TgtObject in User Defined Functions

In user defined function, while setting the methods values of target object, the expression needs to be evaluated in the context of target object itself. The $$TgtObject is used in this case.

Example:

The Ledgers 'Party 1' and 'Party 2' having some opening balance. The requirement is to add the opening balance of both the party's and set the resultant value as the opening balance of Party 2.

```
[Function: Sample Function]
   Object : Ledger : "Party 1"
   01     : NEW OBJECT : Ledger : "Party 2"
   02     : SET VALUE  : OpeningBalance : $OpeningBalance +
            $$TgtObject:$OpeningBalance
;; By prefixing $$TgtObject to opening balance the closing balance of Target Object i.e. Party 2 is retrieved.
   03     : ACCEPT ALTER
```

Here 'Party 1' is the source object and 'Party 2' is the target object. The opening balance of 'Party 2' is accessed using the $$TgtObject:$OpeningBalance.

$$TGTObject in Collection

In simple collection, the source object and target object both are same. In case of the aggregate collection and user defined functions, the target object is different.

The function $$TgtObject allows to access to the values from the target object itself while the collection is being populated. It is required in aggregate collection where the source object is not the same as target object.

The function $$TgtObject is useful when the values are to be populated in collection based on the values that are computed earlier.In aggregate collection the function $$TgtObject can used in the attributes Fetch, Compute and Aggr Compute of collection.

Example:

A report is to be designed for displaying the stock item, the date on which the maximum quantity of an item is sold and the maximum amount is received.

The collection is defined as follows:

```
[Collection: Src Voucher]
   Type              : Vouchers: VoucherType
   ChildOf           : $$VchTypeSales

[Collection: Summ Voucher]
   Source Collection: Src Voucher
   Walk              : Inventory Entries
   By                : ItemName: $StockItemName
   Aggr Compute    : MaxDate : SUM :IF $$IsEmpty:$$TgtObject:$ItemDet+
                     OR $$TgtObject:$ItemDet < $Amount THEN $Date ELSE +
                       $$TgtObject:$MaxDate
   Aggr Compute      : ItemDet: MAX: $Amount
```

While creating a collection ‘Summ Voucher”, $$TgtObject is used to get the date on which the maximum sales amount is received for each stock item. $ItemDet gives the maximum amount received for individual item. The conditions checks if the evaluated $ItemDet is empty for the stockitem or it is less than the current amount of the stock item of the source object then the current date is selected otherwise the value of $MaxDate is retained.

Following Table shows the evaluation of values with respect to target object:

Source Object	**Current Objects**	**Target Objects**
3 Sales Voucher	8 Inventory Entries	3
Sales Voucher -1 Dated - 7/7/09	Item 1 - Rs.500 Item 2 - Rs.500 Item 3 - Rs.500	Item 1 - 7/7/09 - Rs 500 Item 2 - 9/7/09 - Rs 700 Item 3 - 8/7/09 - Rs 800
Sales Voucher -2 Dated - 8/7/09	Item 1 - Rs.400 Item 3 - Rs.800	
Sales Voucher -3 Dated - 9/7/09	Item 1 - Rs.300 Item 2 - Rs.700 Item 3 - Rs.500	

Function – $$ContextKeyword

A New function $$ContextKeyword is used to get the title of the current Report or Menu. It is used to search the context sensitive /online help based on the report or Menu title.

Syntax

```
      $$ContextKeyword [:Yes/No]
```

The default value is No. If the value is specified as YES, then the title of the parent report is returned. If no report is active then the parameter is ignored.

If the attribute Title is not specified in report definiton, then by default it returns the name of report definition.

Example:

```
[Report : Context Keyword Function]
   Form    : Context Keyword Function
   Title   : "New Function Context Keyword"
             |
             |
[Field : Context Keyword Function]
   Use     : Name Field
   Set As  : $$ContextKeyword
```

The functions returns the Title of the current report i.e "New Function Context Keyword".

If the parameter value yes is specified then the title of the report from where the report "Context Keyword Function is called.

10.5 New Attribute – Trigger Ex

When a table is displayed from a field and a new value is to be added to the same table, the attribute **Trigger** is used. It invokes a report. For example, adding new number in fields using dynamic tables such as Tracking number, order No etc.

Syntax

```
      Trigger : <Report Name> : <Trigger Condition>
```

Where,
<Report Name> name of report which is invoked if the **<Trigger Condition>** is true.The value entered in the Ouput field of the **<Report Name>** is added to the table in the field.

Example:

```
[Field: FieldTrigger]
   Use         : Name Field
   Table       : New Number, Not Applicable
   Show Table  : Always
   Trigger     : New Number: $$IsSysNameEqual:NewNumber:$$EditData
   CommonTable : No
   Dynamic     : " "
```

In the field *"Field Trigger"*, a report *"New Report"* is called when the option New Number is selected from the pop up table.

When the value has to be obtained from the complicated flow, a report name does not suffice. To support this functionality a new attribute Trigger Ex is introduced.
The attribute **Trigger Ex** allows to add values to the dynamic table through an expession or user defined functions.

Syntax

```
TriggerEx : <Value-expression> : <Trigger Condition>
```

Where,
<Value Expression> is an expression/function which evaluates to a String if the **<Trigger Condition>** is true. The string value thus obtained is added to the dynamic table.

Example:

```
[Field: FieldTriggerEx]
   Use          : Name Field
   Table        : Ledger, New Number, Not Applicable
   Show Table   : Always
   TriggerEx    : $$FieldTriggerEx: $$IsSysNameEqual:+
                  NotApplicable:$$EditData
   CommonTable : No
   Dynamic      : ""
```

n the field if the user selects any ledger from the table, the function **$$FieldTriggerEx** returns the parent i.e Group name of the ledger selected and adds to the table “Ledger”.

```
[Function: FieldTriggerEx]
   01: RETURN: $Parent:Ledger:$$EditData
```

*Press **Backspace** in the report to view the additions to the table Ledger.*

10.6 New Actions

Two new actions LogObject and LogTarget are introduced to log the object, its method and collection contents.

Log Object

The action Log Object is introduced as global action. It accepts filename as a parameter. In this file the context object, its method and collection are logged.

Syntax

```
      Log Object[:<path\filename>[:<Overwrite Flag>]]
```

Where,

<path/filename> is optional. It accepts the name of file along with the path in which the log is created. If no file name is specified the contents of object are logged in "TDLfunc.log" when logging is disabled otherwise it logs in to the **Calculator** pane.

<Overwrite Flag> is used to specify whether the contents should be appended or overwritten.

The default is **No**, which appends the content in the file. If **YES**, then the file is overwritten.

Example:

```
 [Function: FuncLedExp]

          |

   Object          : Ledger

          |

   10: Log Object : LedgerObj.txt
```

Log Target

The action Log Target is function specific action. It accepts filename as a parameter. In this file the log of object, its method and collection is created for the target object.

Syntax

```
      Log Target[:<path\filename>[:<Overwrite Flag>]]
```

Where,

<path/filename> is optional. It accepts the name of file along with the path in which the log is created. If no file name is specified the contents of object are logged in "**TDLfunc.log**" when logging is disabled otherwise it logs in to the **Calculator** pane.

<Overwrite Flag> is used to specify whether the contents should be appended or overwritten.

The default is **No**, which appends the content in the file. If **YES**, then the file is overwritten.

Example:

```
 [Function: FuncLedExp]

       |

  05: Set Target

       |

  10: Log Target : LedgerObj.txt
```

10.7 Tally Command Line Parameters

While executing tally, now command line parameters can also be given. Tally now accepts command line parameters as explained in the next section.

- ### */NOINITDL*

This parameter will start Tally.ERP without loading any **TDL** specified in the Tally.ini file.

Syntax

```
/NOINITDL
```

- ### */TDL*

This parameters will start Tally.ERP and the specified **TDL** file loaded and can be specified multiple times. The path can be optional, if the TDL file is in the Tally folder.

Syntax

```
/TDL:<path\filename>
```

Where,
<path/filename> is the name of TDL file along with the path.

- ### */NOINILOAD*

This parameter will start Tally.ERP without loading any **Company** specified in the Tally.ini file.

Syntax

```
/NOINILOAD
```

- ### */LOAD*

This parameter starts Tally.ERP and the specified company is loaded and can be specified multiple times.

Syntax

```
/LOAD:<Company Number>
```

- ### */VARIABLE*

This parameter allows to specify inline system variables of specified data type and can be specified multiple times.

Syntax

```
/VARIABLE:<Variable Name>:<Data Type>
```

Where,
<Variable Name> is the name of inline variable. It must be unique.
<Data Type> is any of the primary data type.

- ### */SETVAR*

This parameter allows to specify the value of system variable or inline variable.

Syntax

```
/SETVAR:<Variable Name>:<Value>
```

Where,
<Variable Name> is the name of system variable or inline variable.
<Value> has to be a is any of the primary data type.

- ### */NOGUI*

This parameter hides the GUI(Graphical User Interface) of Tally. It performs the specified ACTION without showing the tally interface based on a non-GUI or GUI action. It starts tally without showing the tally window, performs the action and exits tally for non GUI actions like executing a batch of job. If the action is a GUI action which invokes a report, menu or a message box then the Tally window will be shown until the user quits.

- ### */ACTION*

This parameter starts Tally application with the specified action and it quits Tally application when the user exits.

Syntax

```
/ACTION:<Action Name>[:<Action Parameter>]
```

Where,
<Action Name> is the name of any of the Global actions.
<Action Parameter> is optional.It has to be specified based on the action.

- ### */PREACTION*

This parameter starts Tally, loads the company and executes the specified action before displaying the Main Menu of Tally.

Syntax

```
/PREACTION:<Action Name>[:<Action Parameter>]
```

Where,
<Action Name> is the name of any of the Global actions.
<Action Parameter> is optional. It has to be specified based on the action.

- ### */POSTACTION*

This parameter starts Tally, loads the company and executes the specified action when the user quits Tally.

Syntax

```
/POSTACTION:<Action Name>[:<Action Parameter>]
```

Where,
<Action Name> is the name of any of the Global actions.
<Action Parameter> is optional. It has to be specified based on the action.

- *Only one of the action parameters can be specified at a time.*
- *The actions specified with /**PREACTION** and /**POSTACTION** are not executed for each time the Tally application is restarted due to the change in configuration settings. The action specified with /**PREACTION** is executed when Tally starts for the **First** time. The action specified with /**POSTACTION** is executed during the Last exit from Tally application..*

Example:

Considering that *"C:\Tally.ERP 9"* is the Folder where the Tally.exe is available. The corresponding TDL file "BackUP.txt" for functions is available in the sample folder.

- ***/NOINITDL & /TDL***

```
"C:\Tally.ERP 9\Tally.exe" /NOINITDL /LOAD:00009 "/TDL:C: \Tally.ERP 9
\TDL \SecurityTDL.txt" /TDL:MasterTDL.txt
```

The above command ignores all the TDLs specified in Tally.ini file while loading Tally. It starts Tally application and loads the TDLs - '*SecurityTDL.txt'* and '*MasterTDL.txt*'.

- ***/NOINILOAD with /LOAD***

```
"C:\Tally.ERP 9 \Tally.exe" /NOINILOAD /LOAD:00009
```

The above command ignores all the companies specified in Tally.ini file while loading Tally. It starts Tally application and loads the company identified by **00009**.

- ***/VARIABLE***

```
"C:\Tally.ERP 9 \Tally.exe" /LOAD:00009 /VARIABLE:MyLogicalVar:Logical
```

The above command starts Tally application and with a logical variable **MyLogicalVar**.

- ***/SETVAR and /ACTION***

```
"C:\Tally.ERP9 \Tally.exe" /SETVAR:ExplodeFlag:Yes /LOAD:00009
   /ACTION:DISPLAY:TrialBalance
```

The above command set the value of variable **ExplodeFlag** to **YES** and directly displays **Trial Balance** report.

- ***/PREACTION***

```
"C:\Tally.ERP 9 \Tally.exe"/LOAD:00009 /PREACTION:CALL:BackupBeforeEn-
try
```

The above command starts Tally application, loads the company identified by **00009** and calls the function " **BackUpBeforeEntry** before displaying the main menu.

- ***/POSTACTION***

```
"C:\Tally.ERP 9 \Tally.exe" /LOAD:00009 /POSTACTION:CALL:BackupOnExit
```

The above command starts Tally application and loads the company **00009** and calls the function **BackupOnExit** when the user quits Tally.

- ***/NOGUI***

```
"C:\Tally.ERP 9 \Tally.exe" /NOGUI /LOAD:00009 /ACTION:CALL:BackupSched-
ule
```

The above command starts Tally application, executes the function **BackupSchedule** without displaying the tally window.

Appendix

Objects can be composed of methods and collection. The collection can be made up of objects which is again a combination of methods and collection and so on. This chain can go up to any number of levels. The following diagram represents the structure of an object in general.

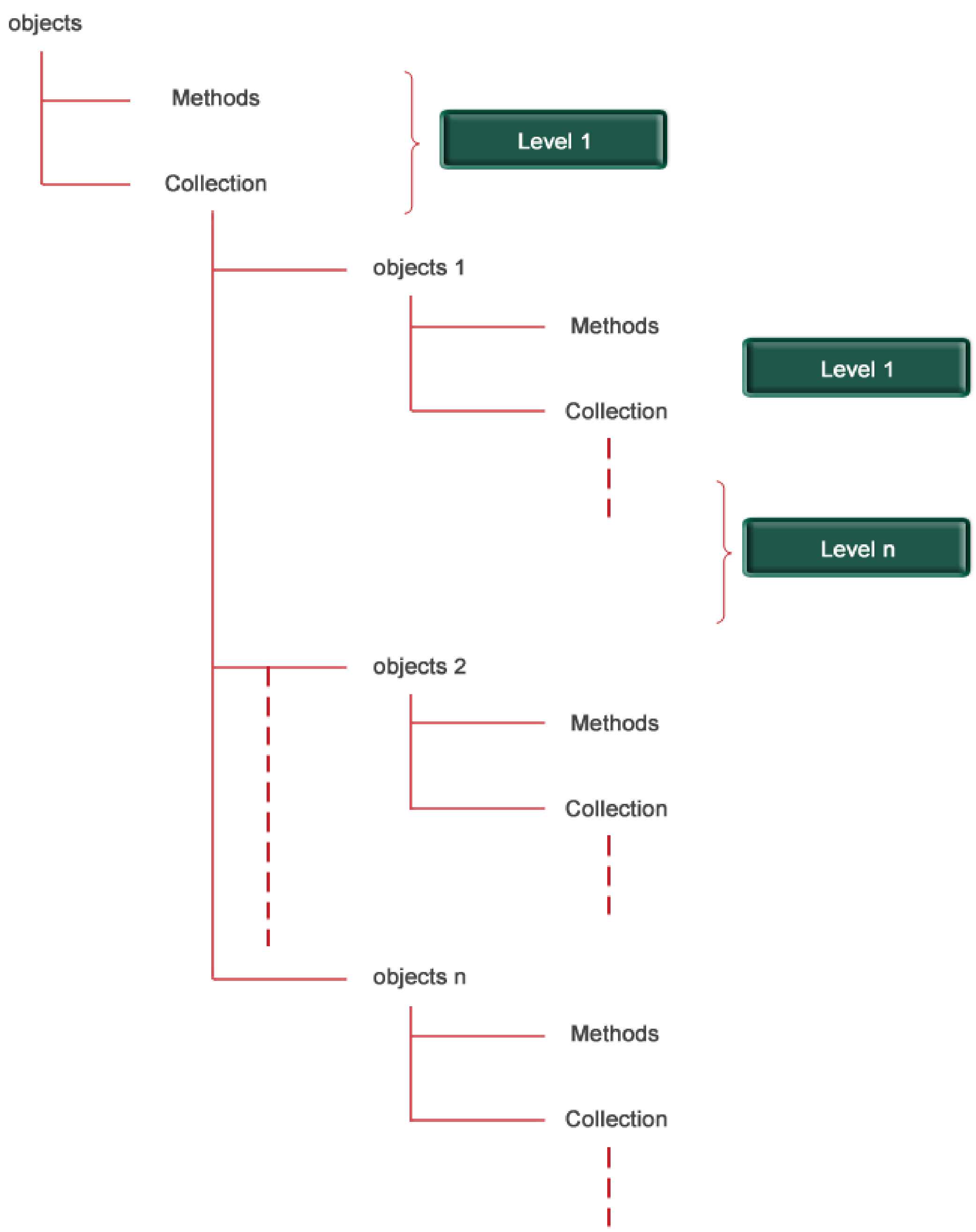


Figure 1 Object Structure

The detailed structure of the masters Company, Ledger, Group, Stock Item and transaction object Voucher is described in this section.

Company

Company objects contains various methods and collections. Some of the collections further contains sub-collections. Figure 2 shows the complete structure of Company objects. The availability of methods and collection depends on the features that are activated while creating the company or through F11 Features and F12 configuration settings.

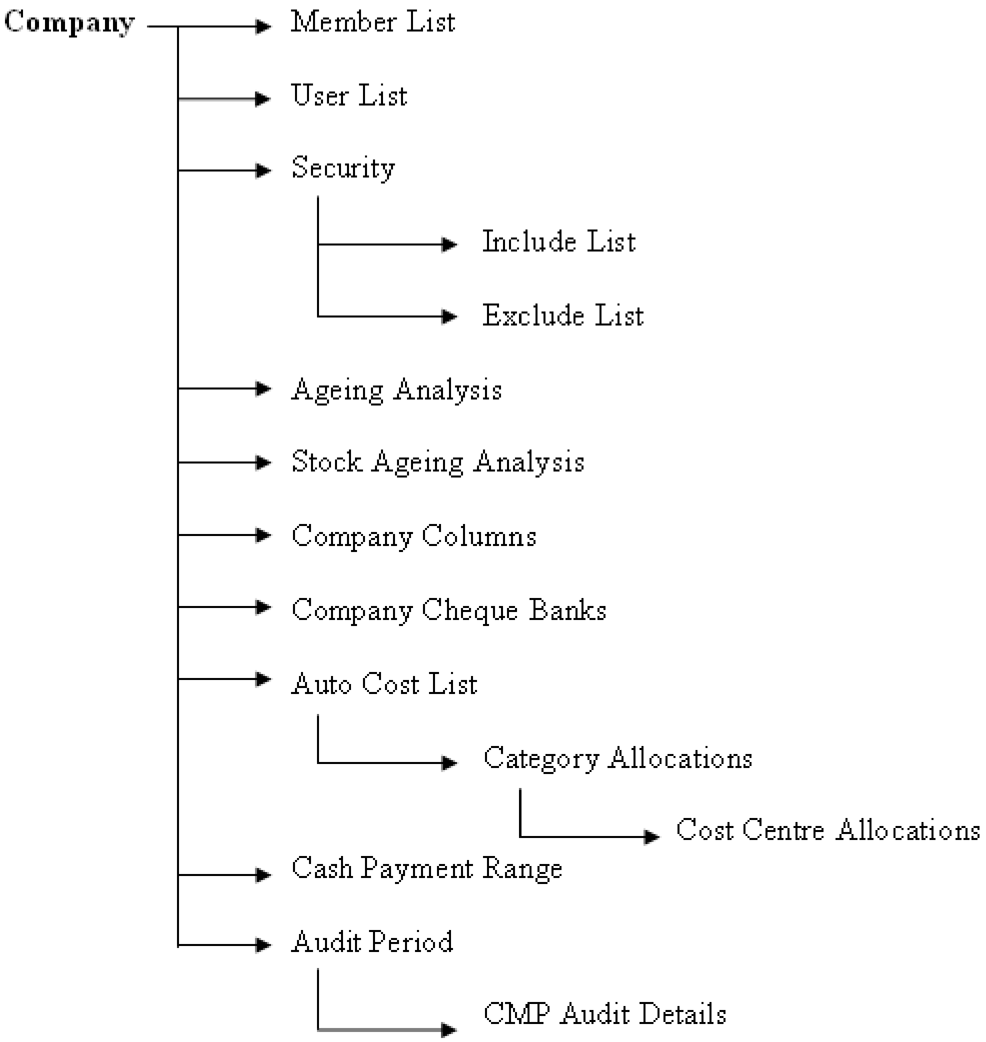


Figure 2 Structure of Company Object

The company object contains methods and collections at first level. Methods $Name, $BooksFrom, $ExciseRange etc are available at Level 1. Some of the collections further contain sub-collection which inturn contains a sub collection. For Example, the collection AutoCostList contains a sub - collection Category Allocations at Level 2. Category allocations again contain sub-collection Cost Centre allocations at Level 3.

Some methods and collections of Company Object:

Name	Type	Description
Name	Method	To fetch the name of Company
Address	Collection	Address of the Company
State Name	Method	To fetch the state name
Pincode	Method	To fetch the Pincode
Email	Method	To fetch the Email id
VATTINNumber	Method	To fetch the VAT No details

Address Collection

Name	Type	Description
Address	Method	To fetch the address of the Company

Group

Group object contain methods $Name, $Parent, $IsBillWiseOn, $IsDeemedPositive, $Overdue-Bills etc. and one sub-collection Language Name.

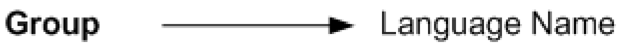


Figure 3 Group Object

Some methods and collections of Group Object:

Name	Type	Description
Language Name	Collection	Group Name in various languages
Parent	Method	Parent of current group name
OpeningBalance	Method	Opening Balance
ClosingBalance	Method	Closing Balance
DebitTotals	Method	DebitTotals
CreditTotals	Method	CreditTotals

Language Name Collection

Name	Type	Description
Name	Method	Ledger Name in selected language
Language	Method	Language Name
LanguageId	Method	Language ID

Ledger

Ledger objects contains methods $Name, $Parent, $LedgerPhone etc. and collections Address, Bill allocations etc at Level 1. The features activated through F11 features and F12 configuration settings effectively decides the availability of methods and collection for Ledger Object.

The complete hierarchy of Ledger object is as shown in the following figure.

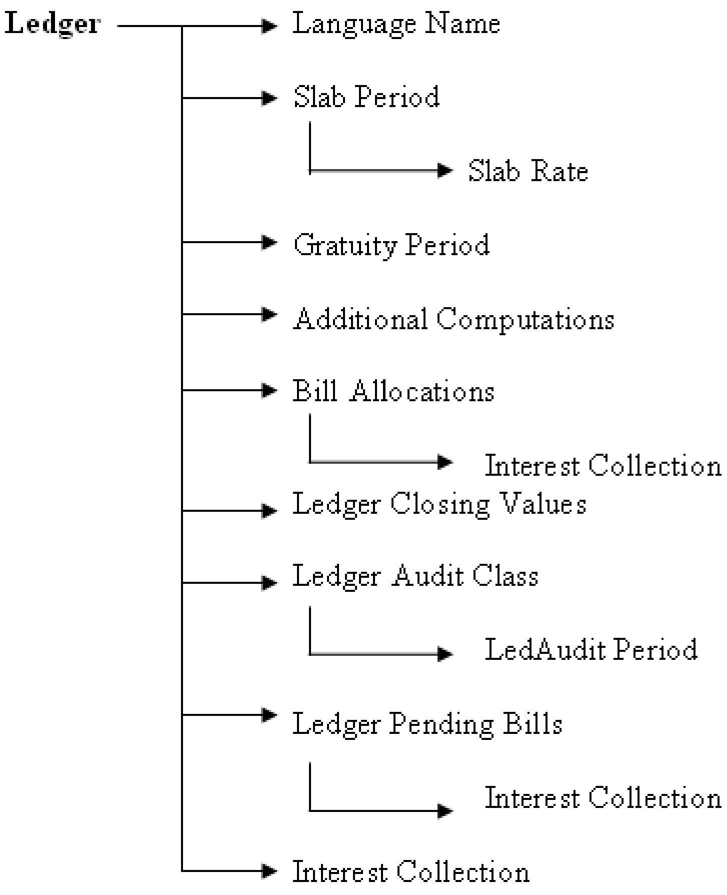


Figure 4 Ledger Object

The collection Bill Allocations won't be available if the option "Maintain Bill-Wise Details" is set to NO in F11 Accounting features.

Some methods and collections of Ledger object:

Name	Type	Description
Name	Method	Ledger name
Parent	Method	Parent group of ledger
Address	Collection	Address of the party
Mailing Name	Method	Ledger Mailing Name
Ledger Phone	Method	Phone number
Ledger Contact	Method	Contact person name
IsBillwiseOn	Method	Checks whether Billwise Details are required for the specified Ledger.
Bill Allocations	Collection	Opening Bill Details

Bill Allocations Collection

Name	Type	Description
BillDate	Method	Bill date
Name	Method	Bill name
OpeningBalance	Method	Opening balance of the bill

Stock Group

The Group Object contains many methods namely $Parent, $BaseUnits etc. and one sub - collection Language Name.

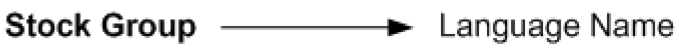


Figure 5 Stock Group Object

Some methods and collections of Stock Group Object :

Name	Type	Description
Name	Method	Name of Stock Group
Parent	Method	Name of Parent
Opening Balance	Method	Opening balance
Closing Balance	Method	Closing balance

Stock Item

The methods $Name, $BaseUnits, $Description etc. and collections Language Name, Batch Allocations and Component List etc belongs to the object stock Item. The features activated through F11 features and F12 configuration settings effectively decides the availability of methods and collection for Stock Item Object.

The complete hierarchy of Stock Item object is as shown in the following figure 6:

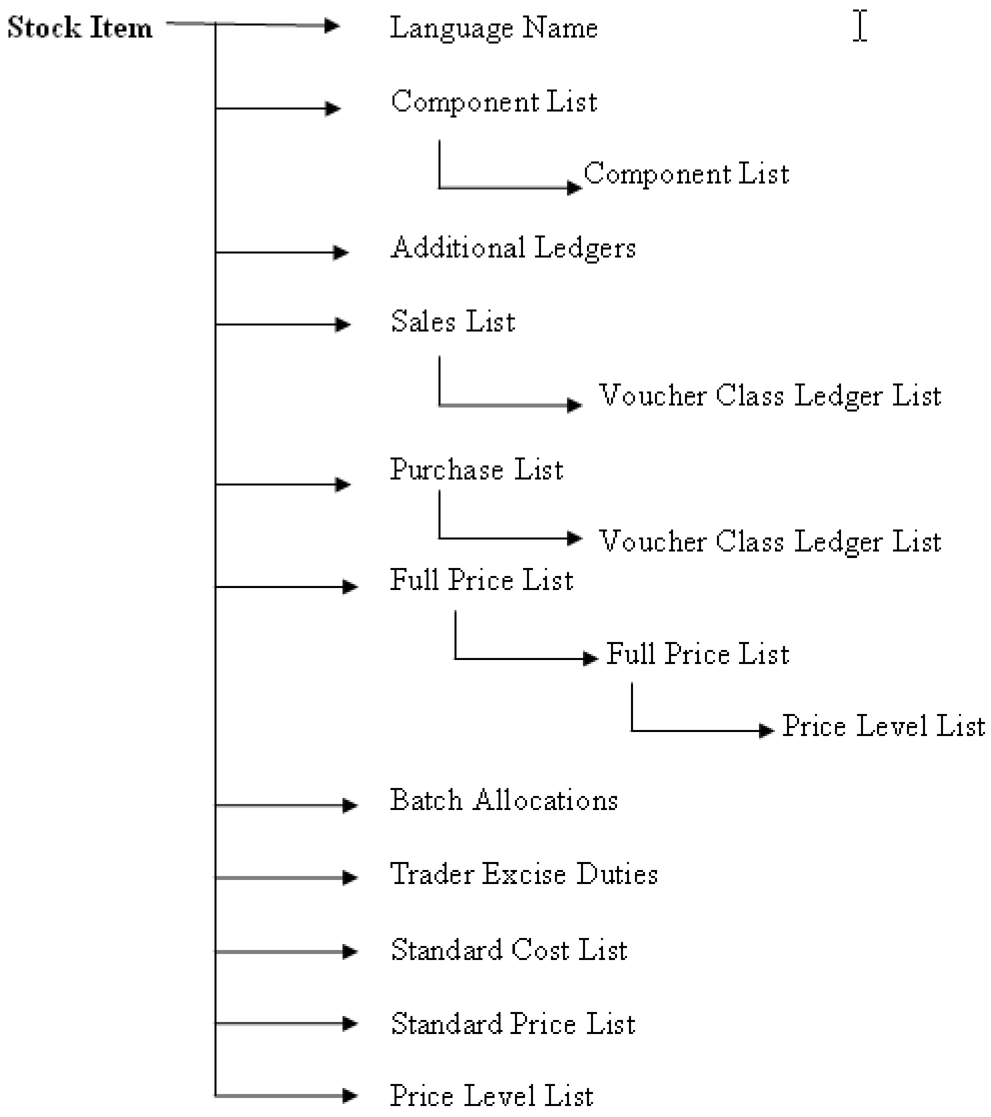


Figure 6 Stock Item Object

The collections Component List, Sales List, Purchase List and Full Price List further contain a sub-collection.

Some methods and collections of Stock Item object:

Name	Type	Description
Name	Method	Name of the Stock Item
Parent	Method	Parent name of the Stock Item
Category Allocations	Collection	Stock Item Category name
BaseUnits	Method	Stock Item Primary units
Description	Method	Description of Stock item
OpeningBalance	Method	Opening Balance in Quantity
ClosingBalance	Method	Closing Balance in Quantity
BatchAllocations	Collection	Opening Batch Details

BatchAllocations Collection

Name	Type	Description
BatchName	Method	To fetch the name of batch
GodownName	Method	Godown name
OpeningBalance	Method	Opening balance
ExpiryPeriod	Method	Expiry period

***Note**: For the details of Category Allocations collection please refer Voucher object*

Voucher

Voucher object is the most complex object in TDL. There are so many methods and collections at Level 1 and most of the collections further have methods and sub-collection. The availability of methods and collection is based on the features activated through F11 features and F12 configuration settings.

The following figure 7 shows the complete hierarchy of the Voucher object:

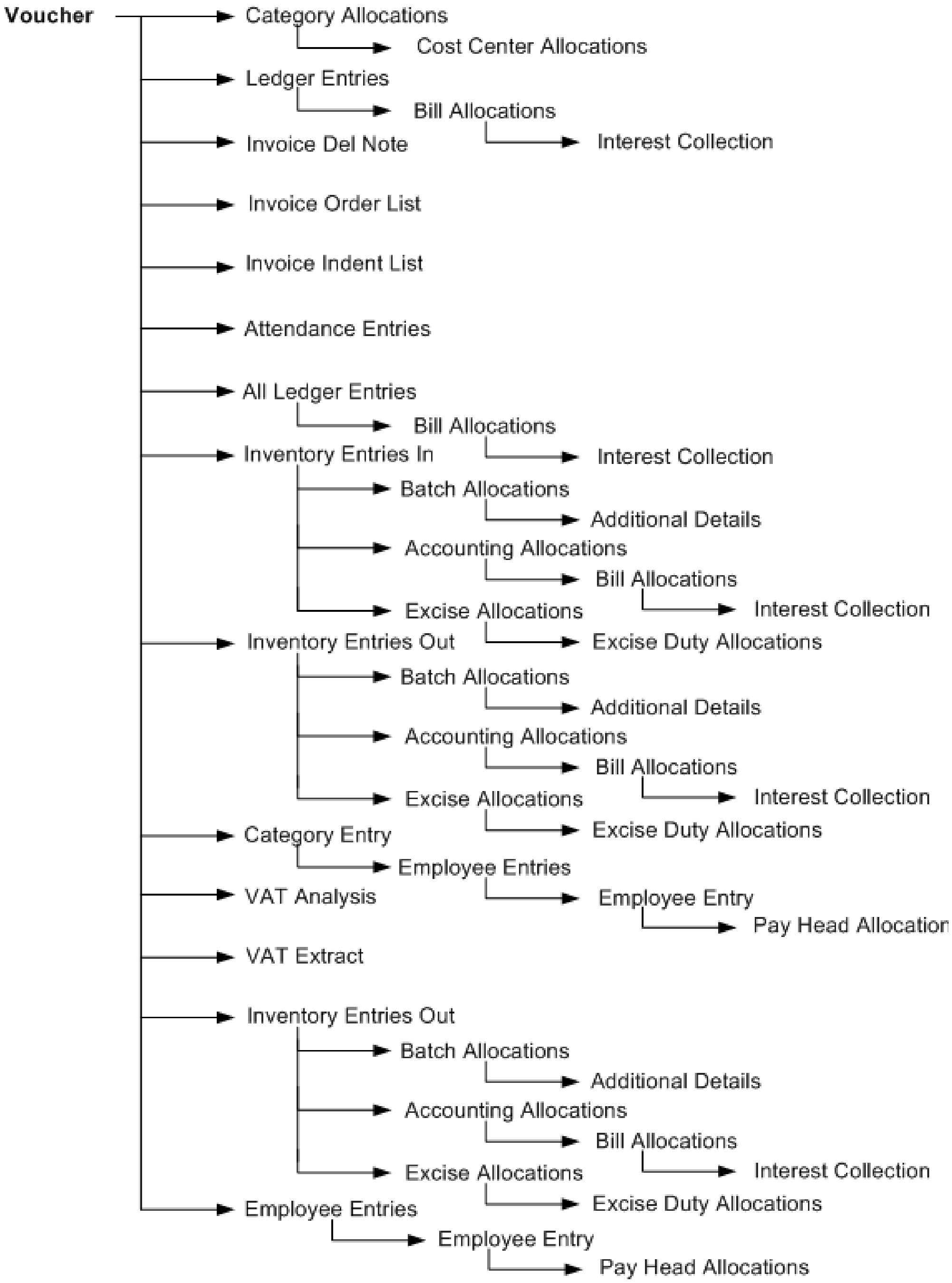


Figure 7 Voucher Object

The collection Ledger Entries, Inventory Entries, All Ledger Entries and All Inventory Entries Collections are widely used in the reports and for invoice customization.

Some methods and collections of Voucher object:

Name	Type	Description
Date	Method	Voucher Date
VoucherNumber	Method	Voucher number
VoucherTypeName	Method	Name of the Voucher Type
PartyLedgerName	Method	Party Name in voucher
Narration	Method	Narration of the voucher
LedgerEntries	Collection	Ledgers involved in the transaction
InventoryEntries	Collection	Inventory details

LedgerEntries Collection

Name	Type	Description
LedgerName	Method	Ledger
Amount	Method	Amount
BillAllocations	Collection	Bill Details
CategoryAllocations	Collection	Category Details

***Note**: For the details of Bill Allocations details please refer Ledger object*

InventoryEntries Collection

Name	Type	Description
StockItemName	Method	Name of the Stock Item sold to the party
BilledQty	Method	Quantity of the item sold to the party
Rate	Method	Rate of the Stock Item
Amount	Method	Amount
Batch Allocations	Collection	Batch details
UserDescription	Method	Description entered

***Note**: For the details of Batch Allocations collection please refer Stock Item object*

CategoryAllocations Collection

Name	Type	Description
Category	Method	Category Name
CostCentreAllocations	Collection	Cost Centre Details

CostCentreAllocations Collection

Name	Type	Description
Name	Method	Name of the Cost centre
Amount	Method	Amount