



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.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form, by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Tally Solutions Pvt. Ltd.

Tally may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written licence agreement from Tally, the furnishing of this document does not give you any licence to these patents, trademarks, copyrights, or other intellectual property.

© 2009 Tally Solutions Pvt. Ltd. All rights reserved.

Tally, Tally 9, Tally9, Tally.ERP, Tally.ERP 9, Shoper, Shoper 9, Shoper POS, Shoper HO, Shoper 9 POS, Shoper 9 HO, TallyDeveloper, Tally Developer, Tally.Developer 9, Tally.NET, Tally Development Environment, Tally Extender, Tally Integrator, Tally Integrated Network, Tally Service Partner, TallyAcademy & Power of Simplicity are either registered trademarks or trademarks of Tally Solutions Pvt. Ltd. in India and/or other countries. All other trademarks are properties of their respective owners.

Version: TDL Reference Manual/1.0/August 2009

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 ie 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 ie., 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 Tally.ERP 9 Release 1.5.

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

1. Tally Definition Language – An Introduction	3
1.1 Tally Definition Language	4
1.1.1 Comparison with other Languages.....	4
1.2 The TDL Program - At a Glance	6
1.3 TDL Capabilities	7
1.4 TDL – Features	7
2. TDL Components	9
2.1 Writing a Basic TDL Program	9
2.1.1 Specification of TDL Files	9
2.2 TDL Interfaces	11
2.3 Hello TDL Program	11
2.3.1 Executing Multiple Files using Include Definition	13
2.4 TDL Components	14
2.4.1 Definitions	14
2.4.2 Attributes	17
2.4.3 Modifiers	22
2.4.4 Actions in TDL	26
2.4.5 Data Types	26
2.4.6 Operators in TDL	27
2.4.7 Special Symbols	29
2.4.8 Functions	29
3. Symbols and Prefixes	31
3.1 Access Specifiers/Symbol Prefixes	32
3.2 General Symbols	32
3.3 The Usage of @ and @@	32
3.3.1 Formula	32
3.4 The Usage of # and ##	34
3.4.1 Referencing a Field using #	34
3.4.2 Modifying existing Definitions using #	34
3.4.3 Accessing value from a Variable using ##	35
3.5 The Usage of \$ and \$\$	35
3.5.1 Accessing a Method using \$	35
3.5.2 Calling an Internal Function using \$\$	35
3.6 Commenting a Code using ;, ;; and /**/.....	36
3.7 Line Continuation Character (+)	36
3.8 Exposing Methods and Creating Procedures (.....)	37
3.9 Reinitialize Definitions (*)	37

3.10 Optional Definitions (!)	37
4. Dimensions and Formatting	41
4.1 Unit of Measurement	41
4.2 Dimensional Attributes	42
4.2.1 Sizing/Size Attributes	42
4.2.2 Spacing/Position Attributes	44
4.3 Alignment Attributes	45
4.3.1 Top Parts, Bottom Parts, Left Parts and Right Parts	45
4.3.2 Top Lines and Bottom Lines	47
4.3.3 Left Field and Right Field	47
4.3.4 Align	48
4.4 Some Specific Attributes	49
4.4.1 Inactive	49
4.4.2 Invisible	49
4.4.3 Widespaced	50
4.5 Definitions and Attributes for Formatting	50
4.5.1 Border	50
4.5.2 Style	52
4.5.3 Color	53
4.5.4 Background and Print BG Attribute	54
4.5.5 Format Attribute	55
5. Variables, Buttons and Keys	57
5.1 Attributes of a Variable	57
5.1.1 Type	57
5.1.2 Default	58
5.1.3 Persistent	58
5.1.4 Volatile	59
5.1.5 Repeat	59
5.2 The Scope of a Variable	60
5.2.1 Local	60
5.2.2 Global	60
5.2.3 Field Acting as a Variable	61
5.3 Modifying the Variable Value	62
5.4 Example - Variables	62
5.5 Buttons and Keys	63
5.5.1 Attributes of Buttons/ Keys	63
6. Objects and Collections	65
6.1 Objects	65
6.1.1 Tally Object Structure	66
6.1.2 Tally Objects Types	68
6.1.3 Object Context	71

6.2 Collections	72
6.2.1 Simple and Compound Collections	73
6.2.2 Sources of Collection	74
6.2.3 Creating a Collection	75
6.3 Object Association	77
6.3.1 Report Level Object association	78
6.3.2 Part Level Object Association	78
6.3.3 Line Level Object Association	80
6.3.4 Field Level Object Association	81
6.4 Methods	81
6.4.1 Internal Methods	82
6.4.2 User Defined/External Methods	82
6.4.3 Accessing Method	82
6.5 Collection Capabilities	85
6.5.1 Basic Capabilities	85
6.5.2 Advanced Capabilities	94
7. Actions in TDL	99
7.1 Categories of Action	99
7.2 Action Association	100
7.2.1 Action Association at Menu Definition	100
7.2.2 Action Association at Button/Key Definition	101
7.2.3 Action Association at Field Definition	102
7.3 Components of Actions	102
7.4 Global Actions	103
7.4.1 Action — Menu	103
7.4.2 Action — Create and Alter	105
7.4.3 Action — Modify Object	108
7.4.4 Action — Browse URL	110
7.5 Actions — Create Collection, Display Collection and Alter Collection	110
7.5.1 Action — Create Collection	110
7.5.2 Action — Display Collection	111
7.5.3 Action — Alter Collection	111
7.5.4 Collection Attributes	112
7.6 Object Specific Actions	113
7.6.1 Menu Actions — Menu Up, Menu Down, Menu Reject	113
7.6.2 Form Actions — Form Accept, Form Reject, Form End	114
7.6.3 Part Actions — Part Home, Part End, Part Pg Up	114
7.6.4 Line Actions — Explode, Display Object, Alter Object	115
7.6.5 Field Actions — Field Copy, Field Paste, Field Erase, Calculator	116
8. User Defined Fields	119
8.1 What is UDF?	119
8.1.1 Creating a UDF	119

8.1.2 To store the User Input in the UDF	120
8.1.3 To retrieve the value of UDF from an Object	120
8.2 Classification of UDF's	121
8.2.1 Simple UDF	121
8.2.2 Aggregate UDF	123
9. Reports, Printing and Validation Controls	127
9.1 Reports	127
9.1.1 Tabular Reports	127
9.1.2 Hierarchical Report (Drill down Report)	133
9.1.3 Column Based Reports	136
9.1.4 Auto-Column Reports	140
9.1.5 Automatic Auto-Column Reports	146
9.1.6 Columnar Report	148
9.2 Printing	148
9.2.1 Printing Techniques	149
9.2.2 Page Breaks	150
9.2.3 Frequently Used Attributes and Functions	153
9.2.4 Validation and Controls	155
10. Voucher and Invoice Customisation	161
10.1 Classification of Vouchers	161
10.1.1 Accounting Vouchers	161
10.1.2 Inventory Vouchers	162
10.1.3 Accounting-cum-Inventory Vouchers	162
10.2 The Structure of a Voucher Object	162
10.3 Customisation	164
10.3.1 Voucher Customisation	164
10.3.2 Invoice Customisation	173

Section II. TDL – Language Enhancements

1. General and Collection Enhancements	185
1.1 Attributes and Modifier Enhancements	185
1.1.1 New Attributes	185
1.1.2 Behavioral Changes of Attributes	188
1.1.3 The Attribute – Child Of to support Voucher Type	189
1.1.4 Attribute Modifiers	190
1.1.5 Behavioral Changes for Attribute Modifiers	191
1.1.6 Partial Attribute Support	193
1.2 Enhanced Special Symbols	193
1.2.1 Multi – line commenting in TDL source code using /* and */	193
1.2.2 Extension of modifying definitions using #	194

1.2.3 ‘*’ (Reinitialize) Definition modifier	194
1.3 Method Formula Syntax with Relative Object Specification	194
1.4 Enhancements – Object Association	196
1.4.1 Report Level Object Association	196
1.4.2 Part Level Object Association	197
1.4.3 Line Level Object Association	198
1.4.4 Field Level Object Association	199
1.5 Enhancements – Object Access via Interface Object	199
1.5.1 Identifying Part and Line Interface object with ‘Access Name’	199
1.5.2 Value Extraction	200
1.6 Bracket support in TDL	201
1.6.1 During the Function Call	202
1.6.2 In the language syntax for nesting formulas	203
1.7 Action Enhancements	203
1.7.1 Enhancements in Key Actions	204
1.7.2 New Actions	205
1.8 Events introduced	211
1.8.1 Event – On Form Accept	211
1.8.2 Event – On Focus	211
1.9 User Defined Function	212
1.10 New Functions	212
1.10.1 \$\$IsObjectBelongsTo.....	212
1.10.2 \$\$NumLinesInScope	213
1.11 Enhanced Collection Capabilities	213
1.11.1 Aggregation and Reporting	213
1.11.2 The Summary Collection is available through Tally ODBC Interface	222
1.11.3 HTTP XML Collection (GET and POST with and without Object Specification)	222
1.11.4 Usage As Tables	229
1.11.5 Dynamic Object support for HTTP–XML Information Interchange	233
1.11.6 Collection Capabilities for Remoting	234
2. Remote Compliant TDL Reports	235
2.1 Client/Server Architecture – An Overview	236
2.2 Tally Client/Server Architecture using Tally.NET	236
2.2.1 Tally.NET Server	237
2.2.2 Tally.ERP 9 Server	237
2.2.3 Tally.ERP 9 Client	238
2.3 Setting up Server Tally for Remote Access	238
2.4 Setting up the Client Tally	241
2.5 TDL – In a Client/Server Environment	243
2.6 TDL Enhancements for Remote	244
2.6.1 Collection Enhancements	244
2.6.2 Report Level Enhancements	247
2.6.3 Function on Request	250

2.6.4 Action Enhancements	252
2.7 Writing Remote Compliant TDL Reports	253
2.7.1 Fetching the single Object	253
2.7.2 Repeating Lines over a Collection	254
2.7.3 Using the same Collection in more than one Report	256
3. User Defined Functions	257
3.1 Functions – In General	257
3.2 Functions – In TDL	258
3.3 Function – Building Blocks	258
3.3.1 Definition Block	259
3.3.2 Procedural Block	261
3.4 Programming Constructs-In Function	262
3.4.1 Conditional Constructs	262
3.4.2 Looping Constructs	265
3.4.3 Control Constructs	268
3.5 Calling a Function	271
3.5.1 Using Action – CALL	271
3.5.2 Using – Symbol Prefix \$\$	271
3.6 Function Execution – Object Context	272
3.6.1 Target Object Context	272
3.6.2 Parameter Evaluation Context	272
3.6.3 Return Value Evaluation	273
3.7 Valid Statements inside a Function	273
3.7.1 Actions for Variable Manipulation	273
3.7.2 Action Enhancements and New Actions	275
3.7.3 Actions – For Object and Context Manipulation	279
4. What's new in Tally.ERP 9 Release 1.5	285
4.1 Collection Enhancements	285
4.1.1 Source Var	286
4.1.2 Compute Var	286
4.1.3 Filter Var	286
4.1.4 Sequence of Evaluation of Collection Attributes	287
4.1.5 Usage of the collection attributes Compute Var, Source Var, Filter Var	287
4.2 List Variables Introduced	289
4.2.1 List Variable	289
4.2.2 List Variable Manipulation	290
4.2.3 Functions Used with List Variables	294
4.2.4 Constructs introduced in functions for List Var	295
4.3 Dynamic Actions	296
4.4 New Functions	297
4.4.1 Function – \$\$TgtObject	297
4.4.2 Function – \$\$ContextKeyword	300

4.5 New Attribute – Trigger Ex	300
4.6 New Actions	302
4.6.1 Log Object	302
4.6.2 Log Target	302
4.7 Tally Command Line Parameters	303
4.8 Enhancements in Previous Release	306
4.8.1 Behavioral change in System Definitions	306
4.8.2 Action Browse URL is Enhanced	307
4.8.3 Collection Enhancements	307
4.8.4 New Function – \$\$DateRange	308
4.8.5 Action – SET VALUE	308
4.8.6 TDL Issues Resolved	308

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.

Eg: 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.

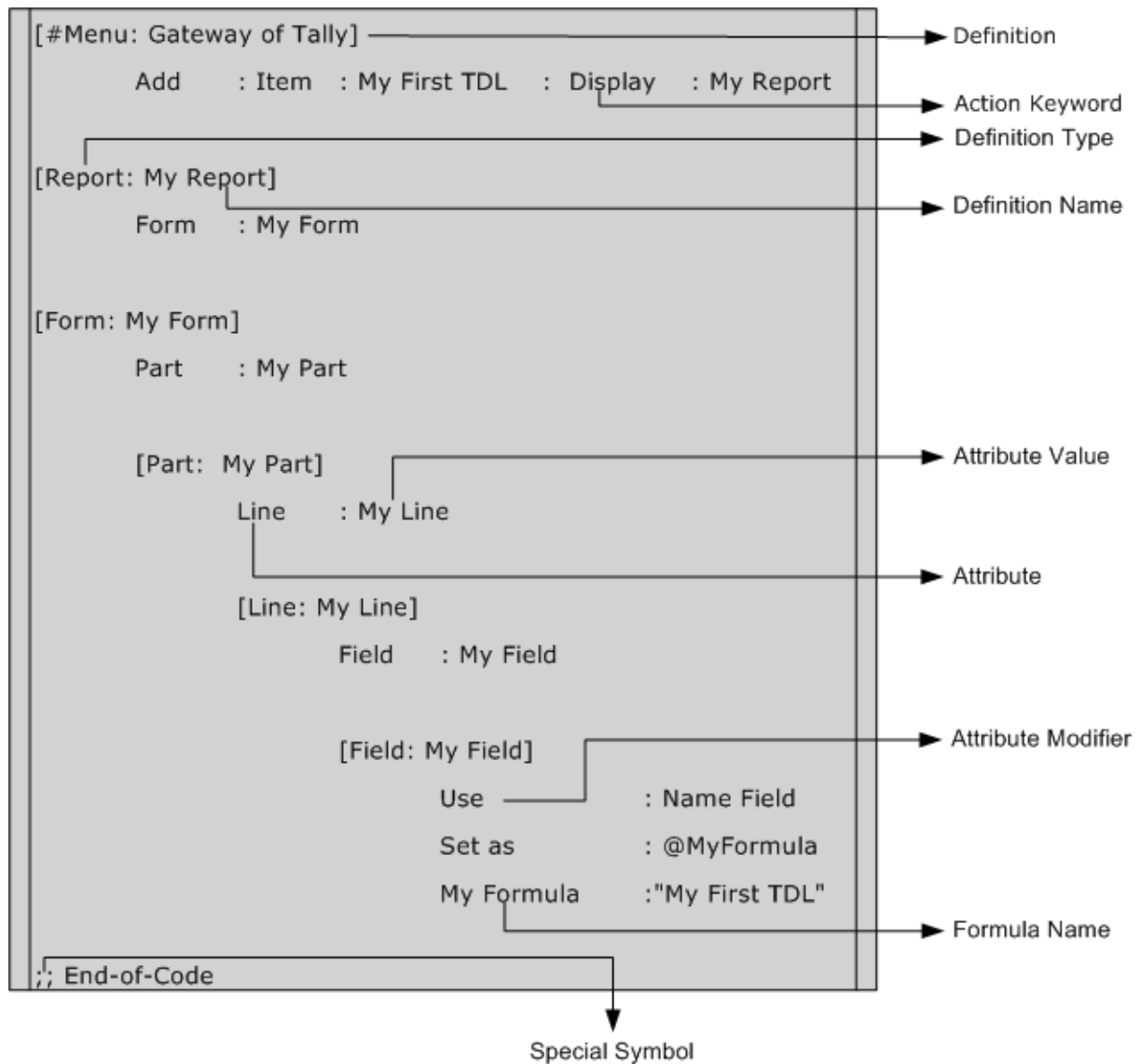


Figure 1.1 TDL Components

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 and 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 ie (.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

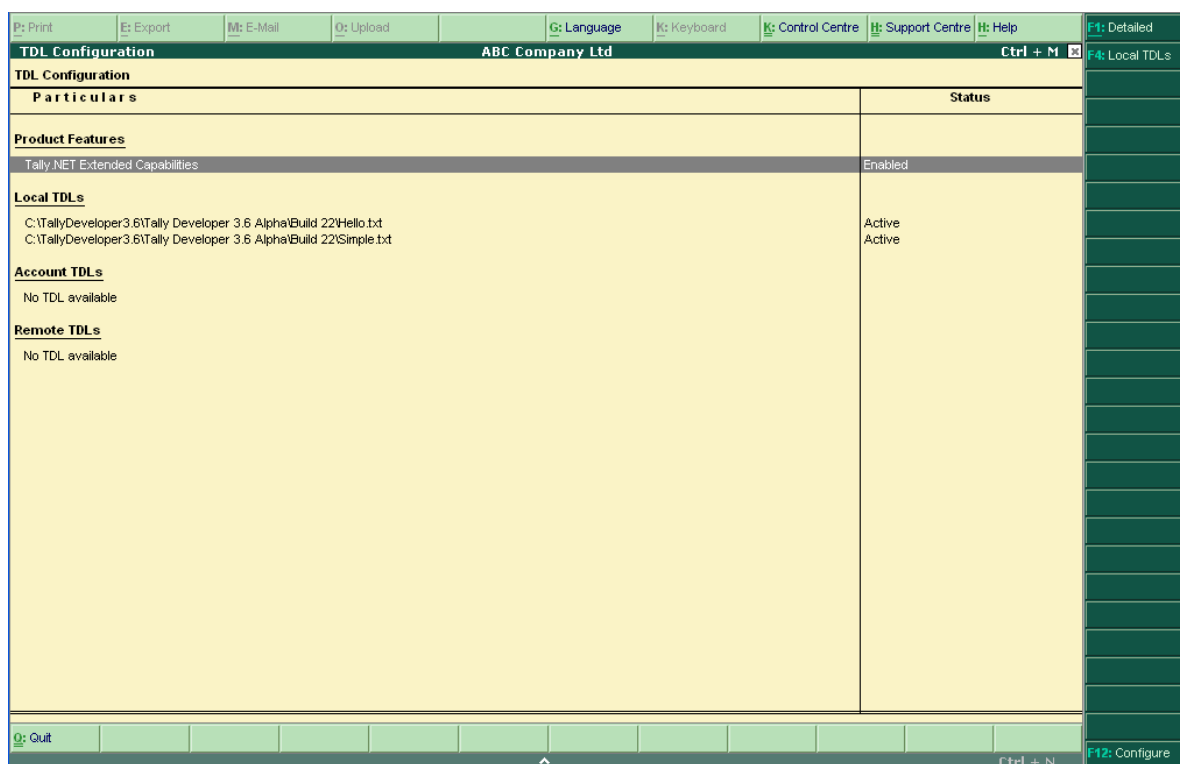
```
User TDL = Yes
TDL      = C:\Tally.ERP 9\MyReport.tcp
          or
TDL      = C:\Tally.ERP 9\MyReport.txt
```

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	
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

Figure 2.2 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 :

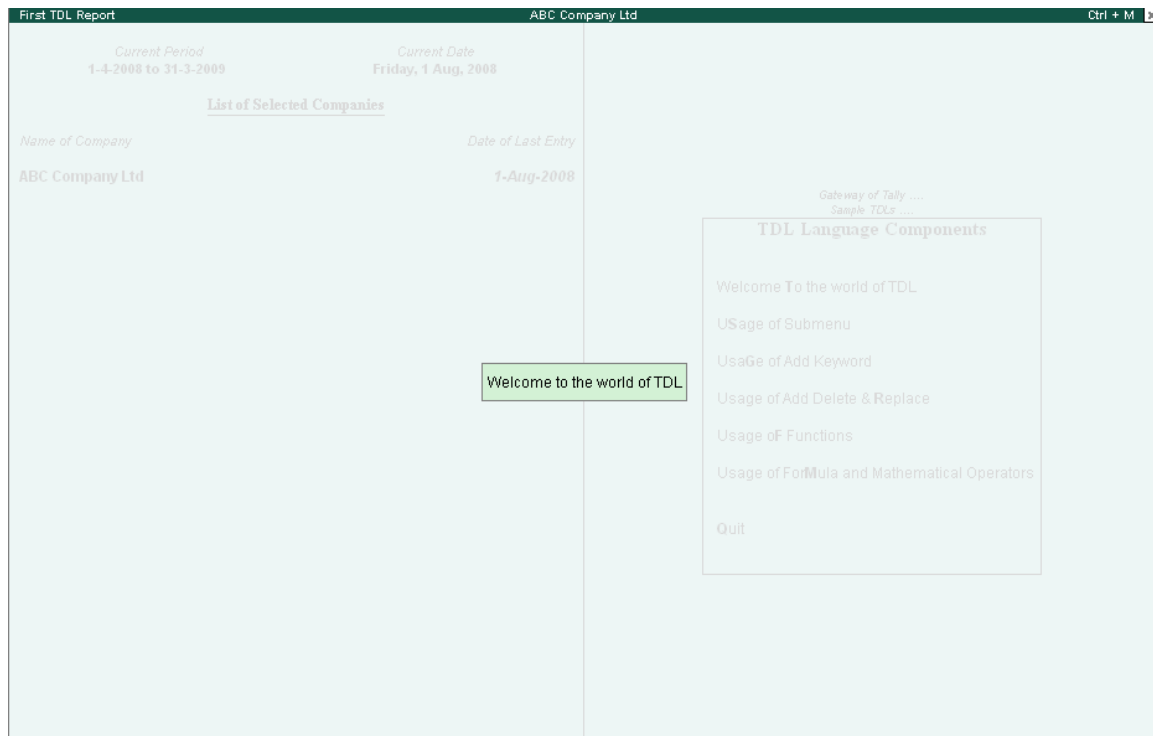


Figure 2.3 Output of Welocme to the world of TDL program

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 Compound 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



*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.

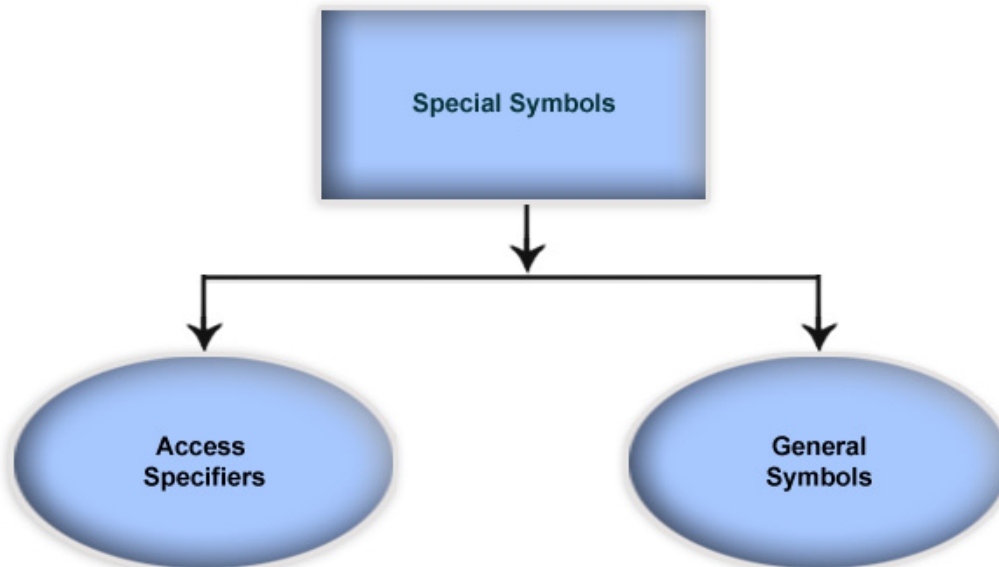


Figure 3.1 Symbol Categorization

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      : MFTBDtlsClsG1000 : $ClosingBalance > 1000
    Option      : MFTBDtlsClsL1000 : $ClosingBalance < 1000

[!Line: MFTBDtlsClsG1000]
    Local      : Field : MFTBDrAmt : Style : Normal Bold
    Local      : Field : MFTBCrAmt : Style : Normal Bold

[!Line: MFTBDtlsClsL1000]
    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: MFTBDtlsClsG1000 : $ClosingBalance > 1000
Switch      : Case 1: MFTBDtlsClsG11000 : $ClosingBalance < 1000

[!Line: MFTBDtlsClsG1000]
  Local : Field : MFTBDrAmt : Style : Normal Bold
  Local : Field : MFTBCrAmt : Style : Normal Bold

[!Line: MFTBDtlsClsG11000]
  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 circum-

stances, 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 SVDDate, 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 received 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 relationships 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 without 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.

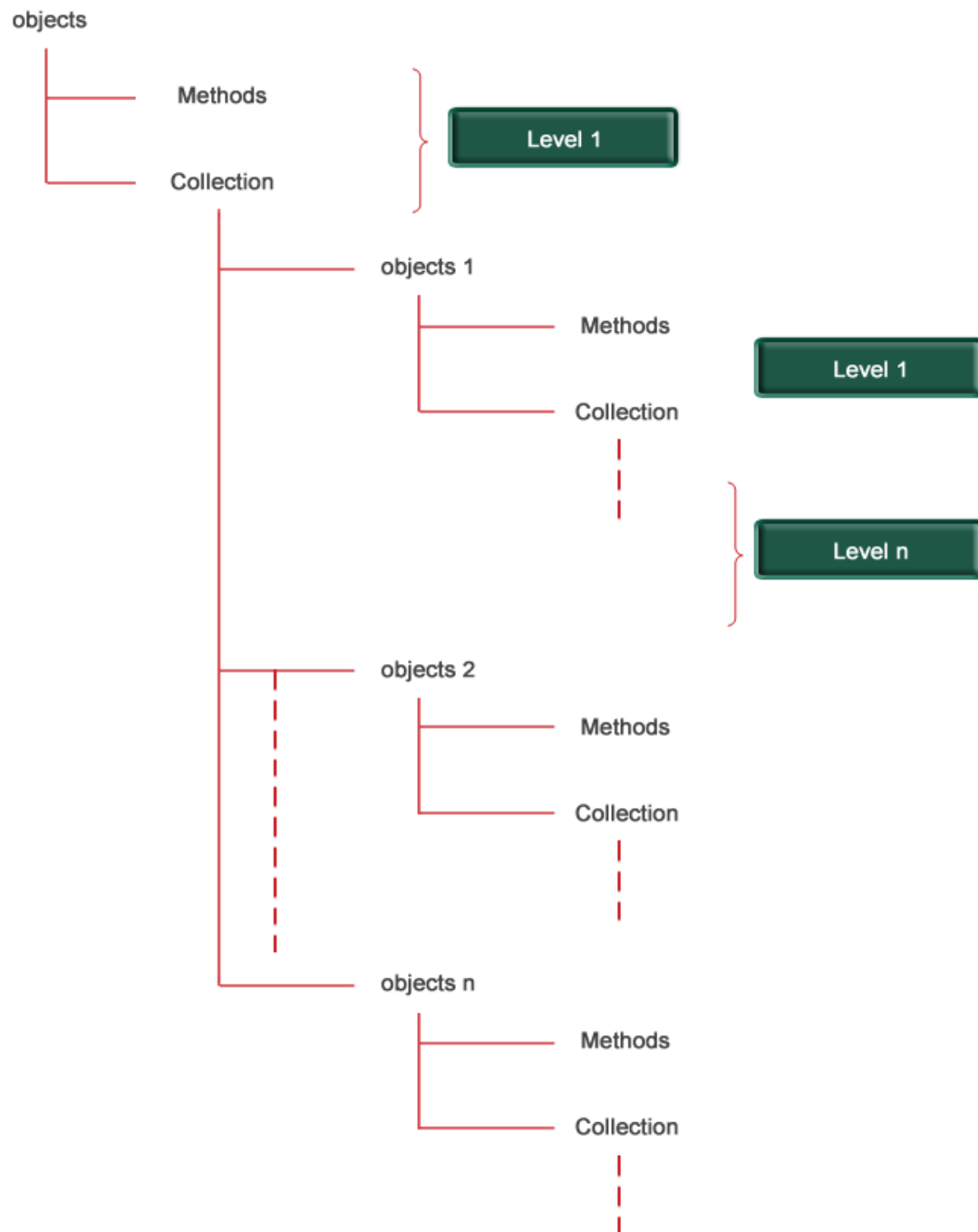


Figure 1.1 Tally Object Structure

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 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.

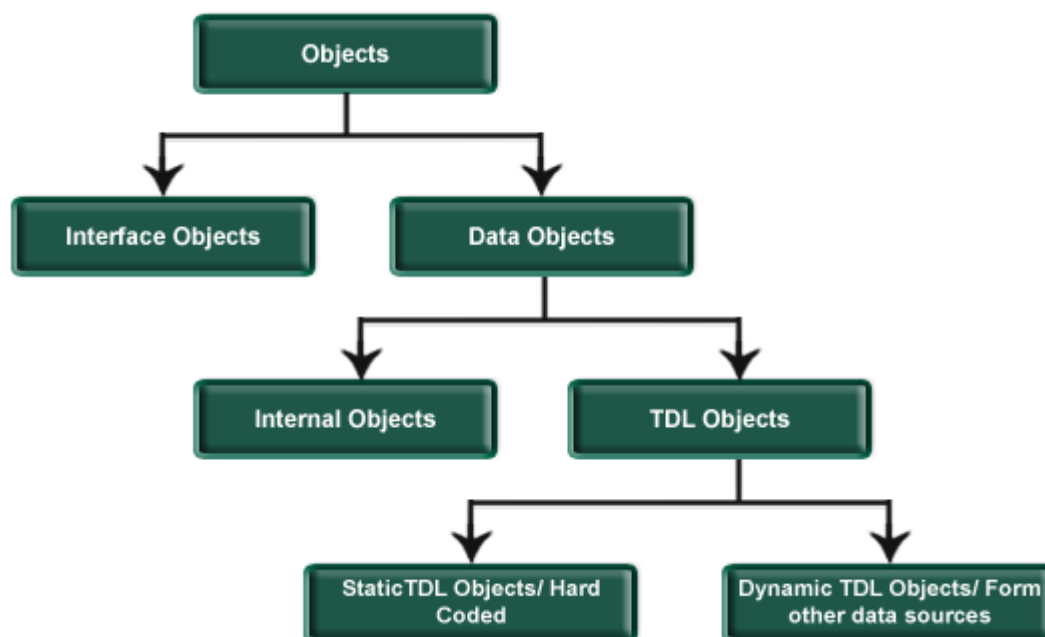


Figure 1.2 Classification of objects

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	Attributes
E001	Krishna	Aug 01	Manager	Objects
E002	Radha	Aug 01	Asst. Manager	

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.

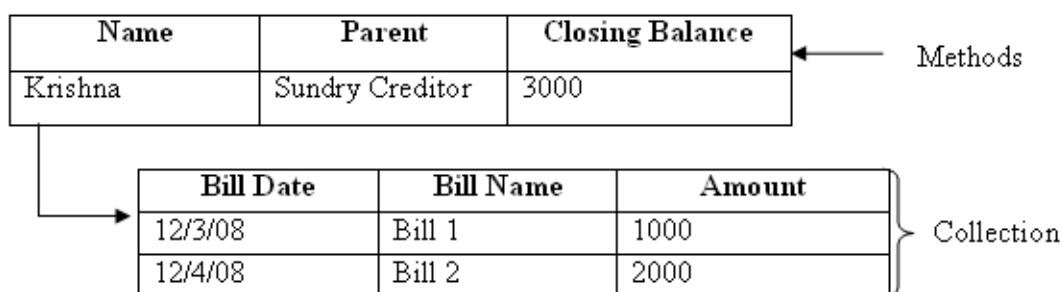
```
[Object : Emp 1]
EmpNo      : E001
Name       : "Krishna"
Date       : Aug 01
Designation : Manager
```

```
[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



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.

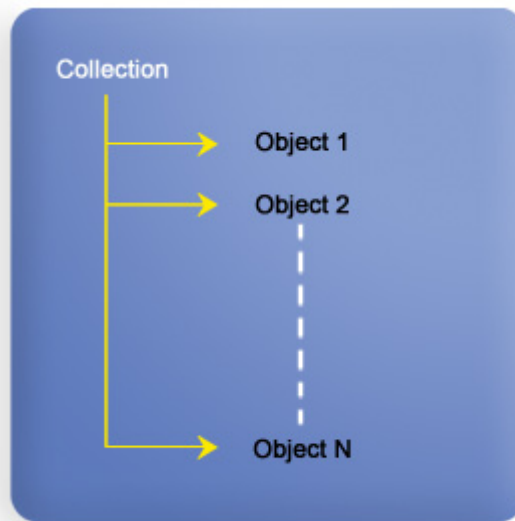


Figure 1.3 Collection of objects

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.

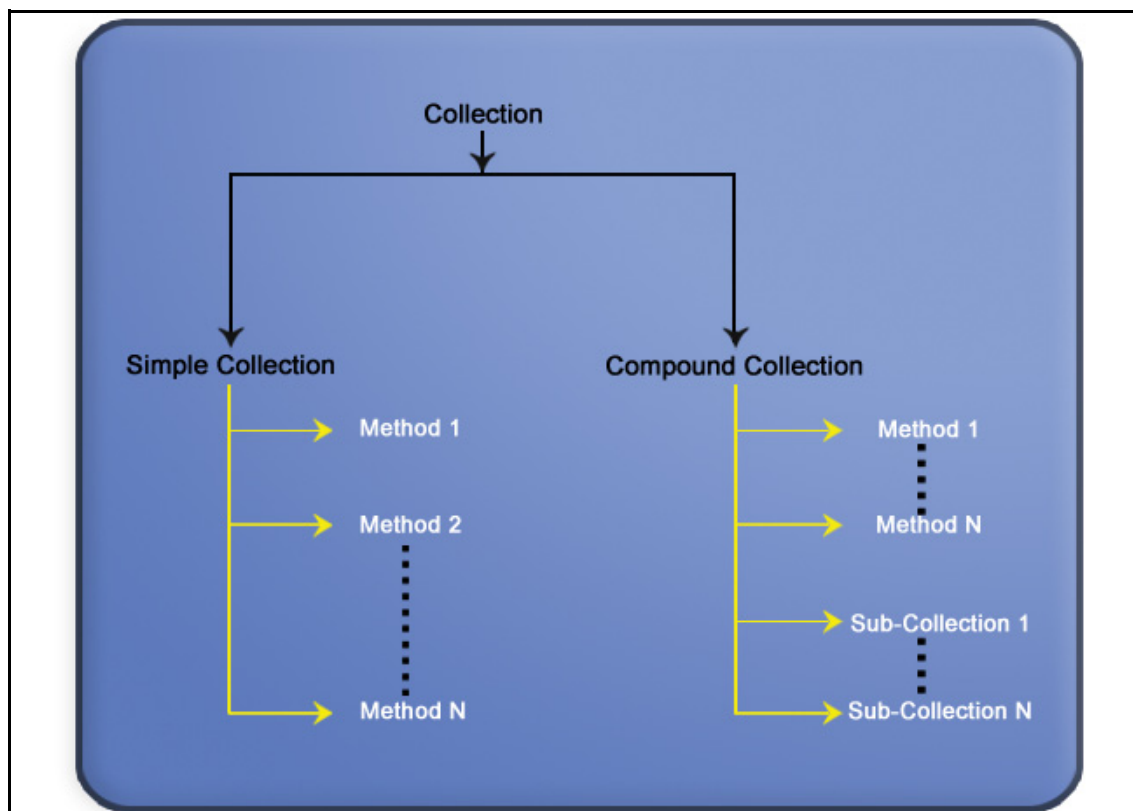


Figure 1.4 Classification of collections

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 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.

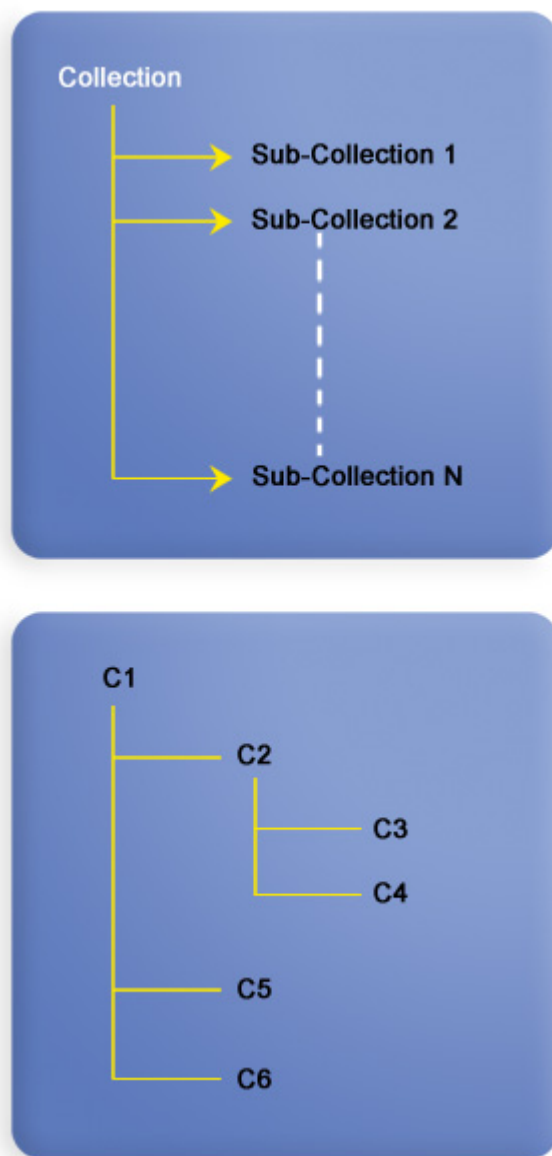


Figure 1.5 Collection of Sub-collection

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 summarize them in one scan. Overall, it reduces the TDL code complexity, resource requirement and increases performance drastically in case of reports generated using this new capability.

The attributes used for extraction, chaining, 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 Enhancements 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

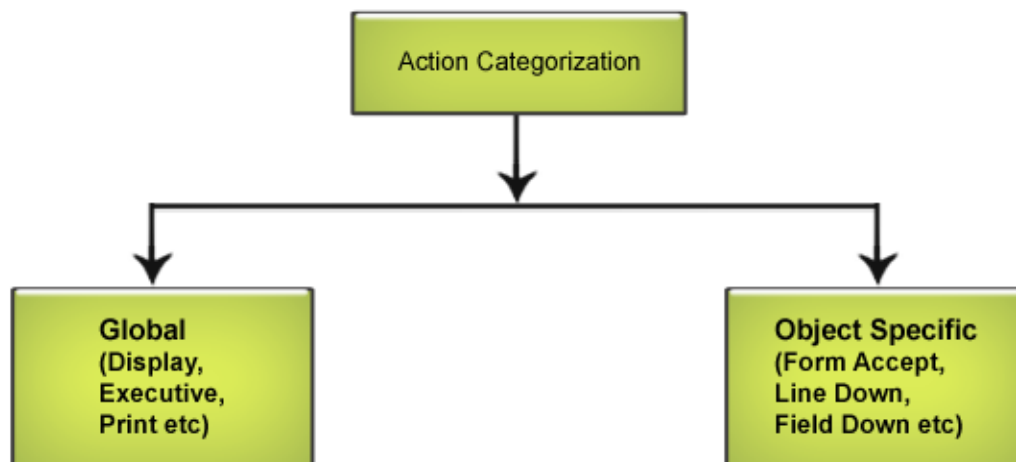


Figure 1.1 Action Categorization

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 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>.
Method Name : Value>[,Method Name: <Value> , ...]
[,<SubObjectPathSpec>.Method Name :<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:BrowseURL:+
              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 take 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 Attribute **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

Stock Summary		ABC Company Ltd		Ctrl + M
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
Q: Quit		R: Remove Line		
		U: Restore Line		
		U: Restore All		
		S: Select		

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.

My TB Report		ABC Company Ltd		Ctrl + M	
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:

Figure 1.6 Trial Balance Report

Trial Balance		ABC Company Ltd		Ctrl + M	
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			

135

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      : SVCCurrentCompany, SVFromDate, SVToDate
```

P: Print	E: Export	M: E-Mail	O: Upload	L: Language	K: Keyboard	C: Control Centre	S: Support Centre	H: Help	F2: Period	
MulCol Trial Balance ABC Company Ltd										
Particulars							ABC Company Ltd		Ctrl + M	F3: Company
							1-Apr-2008 to 1-Aug-2008		F5: Led-wise	
							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		

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

Column Details		ABC Company Ltd		Ctrl + M	
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				51,11,856.90	
Direct Expenses		33,240.00			
Fixed Assets		34,37,489.88			
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	
<div><div>Column Details</div><div>From Date : 1-Apr-2007</div><div>To Date : 30-Apr-2007</div></div>					
GRAND TOTAL		6,22,47,955.71		6,09,31,620.43	

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 : SVCCurrentCompany, 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

P: Print	E: Export	M: E-Mail	O: Upload	L: Language	K: Keyboard	C: Control Centre	S: Support Centre	H: Help	F2: Period	
MulCol Trial Balance				ABC Company Ltd				Ctrl + M		F3: Company
Particulars				ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		ABC Company Ltd 1-Apr-2007 to 30-Apr-2007		F5: Led-wise		
				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			

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"
```

Particulars		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"  
BelongsToIf : "Yes"  
IsAgeWise  : No  
Periodicity : "Month"
```

```
[Object: MyYearly]  
Name       : $$LocaleString:"Yearly"  
VarName    : "SVFromDate, SVToDate"  
CollName   : "Period Collection"  
BelongsToIf : "Yes"  
IsAgeWise  : No  
Periodicity : "Year"
```



Columns can be repeated over any collection. It is not restricted only to a Period.

Auto Repeat Columns		ABC Company Ltd		Ctrl + M	
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 :

Column Details
Half-Yearly
Monthly
Quarterly
Yearly

Figure 1.12 Auto Repeat Columns

- 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 SVCCurrentCompany,
;; 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

P: Print	E: Export	M: E-Mail	O: Upload	L: Language	K: Keyboard	C: Control Centre	S: Support Centre	H: Help	F2: Period
AutoCol Trial Balance									
Particulars	ABC Company Ltd				ABC Company Ltd		ABC Company Ltd		Ctrl + M
	1-Apr-2007 to 30-Jun-2007				1-Jul-2007 to 30-Sep-2007		1-Oct-2007 to 31-Dec-2007		F3: Company
	Closing Balance				Closing Balance		Closing Balance		F5: Led-wise
	Debit		Credit		Debit		Credit		
Capital Account		55,00,000.00		55,00,000.00		55,00,000.00		5 more ... →	
Current Assets	2,54,46,664.95			2,90,59,229.83		2,84,31,555.33		55,00,000.00	
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		

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		ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		Global Enterprises For 1-Apr-2007	
		Closing Balance		Closing Balance	
Particulars		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.



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:

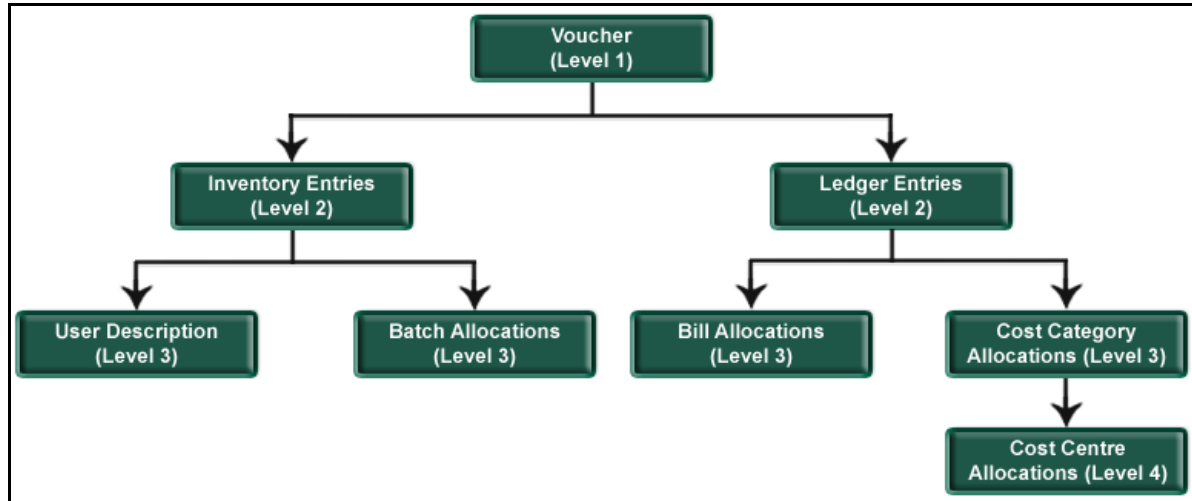


Figure 1.1 Hierarchy of Voucher objects

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 Collection 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		Cost Centre/Classes : Not Applicable		31-Mar-2008 Monday
Account : HDFC Bank Cur Bal : 1,39,229.82 Cr				
Particulars			Amount	
Modern Advertisers Cur Bal : 0.00 Cr Agst Ref : 111			7,303.40	
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 : 11384			7,303.40	

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																				
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 80%;">Particulars</th> <th style="width: 20%;">Amount</th> </tr> </thead> <tbody> <tr> <td colspan="2" style="padding: 5px;">Account :</td> </tr> <tr> <td style="padding: 5px;">Modern Advertisers</td> <td style="text-align: right; padding: 5px;">7,303.40</td> </tr> <tr> <td style="padding: 5px;">Agst Ref 111 7,303.40 Cr Output ST - Advt. Services</td> <td></td> </tr> <tr> <td colspan="2" style="height: 100px;"></td> </tr> <tr> <td colspan="2" style="padding: 5px;"> Cheque Number and Date : 11384 dt 31-Mar-2008 </td> </tr> <tr> <td colspan="2" style="padding: 5px;"> Through : HDFC Bank </td> </tr> <tr> <td colspan="2" style="padding: 5px;"> On Account of : Full Amount is being received </td> </tr> <tr> <td colspan="2" style="padding: 5px;"> Amount (in words) : Rs. Seven Thousand Three Hundred Three and Forty paise Only </td> </tr> <tr> <td></td> <td style="text-align: right; padding: 5px;"> <div style="border-top: 1px solid black; border-bottom: 3px double black; display: inline-block;"> 7,303.40 </div> </td> </tr> </tbody> </table>	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			<div style="border-top: 1px solid black; border-bottom: 3px double black; display: inline-block;"> 7,303.40 </div>	
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																					
	<div style="border-top: 1px solid black; border-bottom: 3px double black; display: inline-block;"> 7,303.40 </div>																				
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		Ref.:		5-Mar-2008 Wednesday		
		Cost Centre/Classes : J Not Applicable				
Party's A/c Name : Fuzitsy Systems				Price Level : J 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 define 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
```

Name of Item	Quantity	Rate	per	Disc. %	Amount
Assembled PIV	20 Nos	22,000.00	Nos		4,40,000.00
Transportation & Packaging					2,500.00
CST Tax @ 4%				4 %	17,600.00

Height	Width	Dimension
10	10	100

Summary	
20 Nos	4,60,700.00

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
<i>Tracking No. : 12142 Order No. : 5 Due on 12-Aug-2008</i>					
Assembly Floor	20 Nos	22,000.00	Nos		4,40,000.00
<i>Tracking No. : End of List Order No. :</i>					
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	22,000.00Nos	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 Comprehensive 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.

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				Invoice No. 2		Dated 1-Aug-2008		
				Delivery Note		Mode/Terms of Payment 90 Days		
				Supplier's Ref.		Other Reference(s)		
Buyer Universal Systems				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				
SI No.	Marks & Nos./ Container No.	No. & Kind of Pkgs.	Description of Goods	Quantity	Rate	per	Dist. %	Amount
1			Assembled PIV <i>Transportation & Packaging</i> <i>CST Tax @ 4%</i>	20 Nos	22,000.00	Nos		4,40,000.00
						4 %		2,500.00 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
Company's VAT TIN : 23424872389 Company's CST No. : 234234234 Company's Service Tax No. : 234234 Company's PAN : EENMM16789				<u>Terms & Conditions :</u> 1. Training : 9 hours 2. Tally Support : 3 months without any additional charges				
				Declaration We declare that this invoice shows the actual price of the goods described and that all particulars are true and correct.				
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

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 existing attributes and modifiers is also changed.

1.1 New Attributes

New attributes that are introduced are explained in this section.

1.1.1 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'.

1.1.2 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"
```

1.1.3 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
```

1.1.4 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
```

1.1.5 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"
```

1.1.6 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:

1.2.1 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.

1.2.2 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.

1.2.3 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

1.4.1 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.

1.5 Behavioral Changes for Attribute Modifiers

The behavior of following attributes is enhanced.

1.5.1 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*.

1.5.2 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*.

1.5.3 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.

1.6 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.

1.6.1 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.

4.2.1 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'.

4.2.2 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".

4.3.1 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

4.3.2 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
```

4.4 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.

5. 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

5.1 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.

5.2 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'.

5.2.1 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".

5.2.2 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".

5.2.3 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 Entries of the Voucher Object which is associated with Part having the access name "MyPartAccessName"

```
[Part: Sample Part]
```

```
Repeat : Sample Line:(Part, "MyPartAccessName").InventoryEntries
```

6. 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

6.1 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)
```

6.2 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
```

6.2.1 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)
```

7. Action Enhancements

Some of the existing actions are enhanced to support the multiline selection capabilities. Several new actions are also introduced in TDL.

7.1 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.

7.1.1 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*

7.1.2 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.

7.2 New Actions

Following new actions are introduced in the language:

7.2.1 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

7.2.2 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)
```

7.2.3 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 separated 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"
```

7.2.4 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 separated 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"
```

7.2.5 Action – ChangeCrypt Company

This action allows to change the TallyVault Password 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 separated 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"
```

7.2.6 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.

Syntax

Action: Browse URL: <URL>

Where,

<URL> is any link to a web site

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**.

7.2.7 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**.

7.2.8 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
```

8. 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

8.1 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:+
        SCNewIssueResp
```

8.2 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
```

9. 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. There 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”.

10. New Functions

Following functions are introduced in the Language:

10.1 \$\$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 be checked using the following statement.

```
$$IsObjectBelongsTo:Group:"North Debtors":$$GroupSundryDebtors
```

10.2 \$\$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.



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.

11. 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.

11.1 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.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.

11.1.5 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.

11.1.6 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.*`

11.1.7 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.

11.1.8 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.

11.1.9 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 XMI file. The encoding is Unicode by default as it is not specified.



Support for other data sources like ODBC, Dll will also be available in future releases.

11.1.10 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 : $LedgeName
    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.

11.1.11 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*.

11.1.12 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.

11.2 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.

11.3 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.

11.3.1 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.

11.3.2 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.

11.3.3 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
```

11.3.4 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
```

11.3.5 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.

11.3.6 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.

11.3.7 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 Success 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 its 1 it's 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.



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 response from the PHP application file "CXMLResponse as per Tally".

11.4 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

11.4.1 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
```

11.4.2 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
```

11.4.3 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
```

11.4.4 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 *XML-Data.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
```

11.5 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.

11.6 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.

The remoting capabilities are discussed in detail in the next section II **“Writing Remote Compliant TDL for Tally.ERP 9 ”**.

Remote Compliant TDL Reports

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

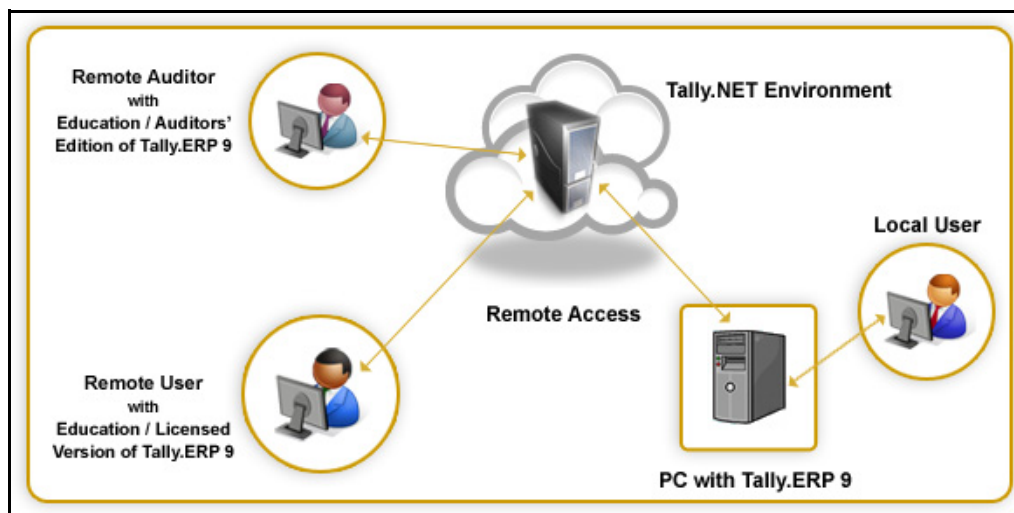


Figure 1.1 Overview of Tally.NET

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 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.

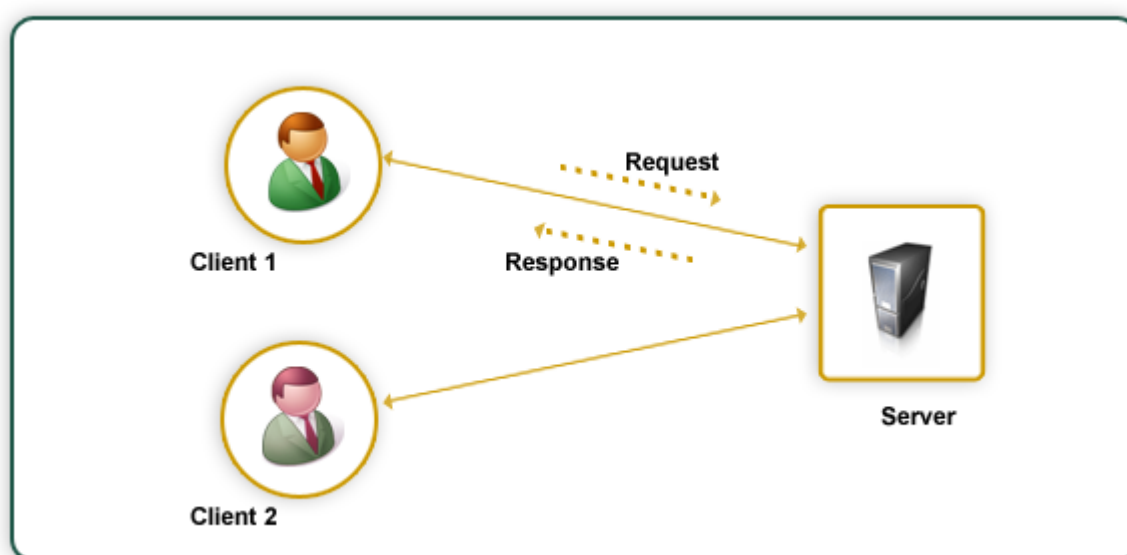


Figure 1.2 Block diagram of client/server architecture

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.

2.1.1 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

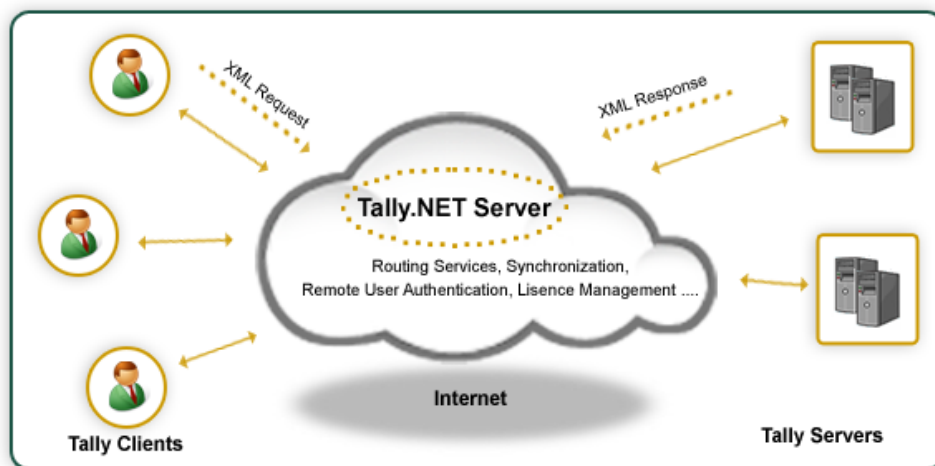


Figure 1.3 Tally.NET Architecture

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**

Company Alteration		Ctrl + M	
Name : ABC Company Ltd			
Mailing & Contact Details			
Mailing Name	: ABC Company Ltd	Currency Symbol	: Rs.
Address	: 5, 9th Cross Margosa Road Malleswaram	Maintain	: Accounts with Inventory
		Financial Year from	: 1-4-2007
		Books beginning from	: 1-4-2007
Company Details			
Security Control			
Statutory compliance for	: India	Disallow opening in Educational mode	? No
State	: Karnataka	Use Security Control	? Yes
PIN Code	: 560022	(Enable Security to avail Tally.NET Features)	
Telephone No.	: 098234723	Name of Administrator	: A
E-Mail	: contact@abc.com	Password	: *
		Repeat Password	: *
		Use Tally Audit Features	? Yes
Base Currency Information			
Base Currency Symbol	: Rs.	Show Amounts in Millions	? No
Formal Name	: Indian Rupees	Put a SPACE between Amount and Symbol	
Number of Decimal Places	: 2	Decimal Places for Printing Amounts in Words	
Is Symbol SUFFIXED to Amounts	? No		
Symbol for Decimal Portion	: Paise		
			Accept ?
			Yes or No

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 Operations Alteration Ctrl + M

Current Period: _____ Current Date: _____

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

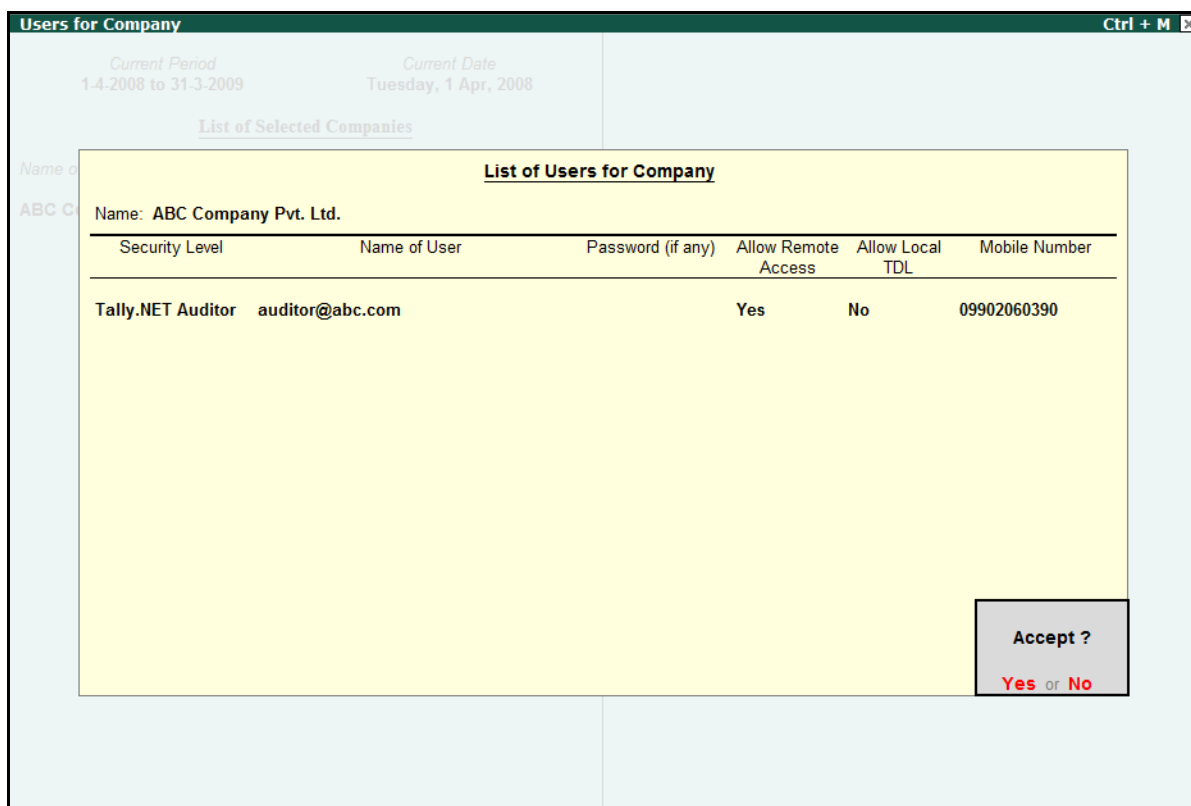
Accept ?
Yes or No

F1: Accounts F2: Inventory F3: Statutory F4: Tally.NET F5: Audit

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**



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

Figure 1.6 Authorizing the Remote Users

TABLE 1.



- ❑ 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**

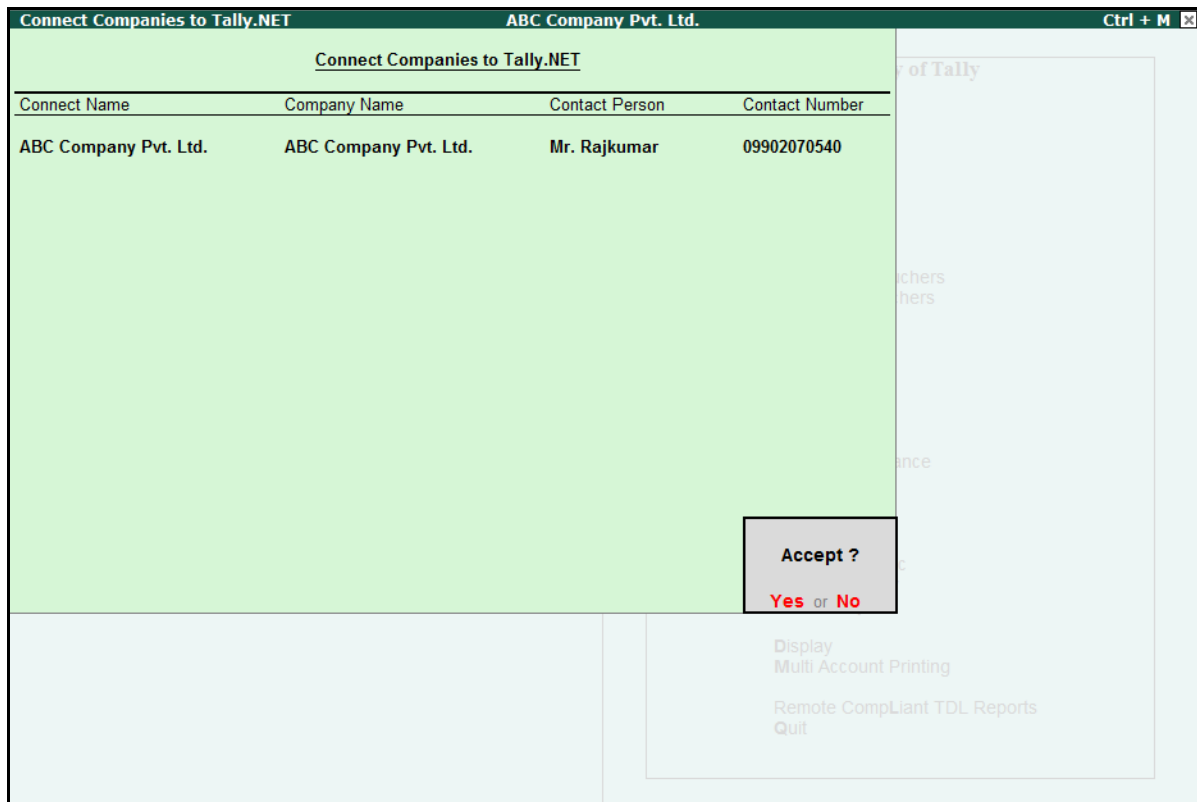


Figure 1.7 Connecting companies to Tally.NET

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

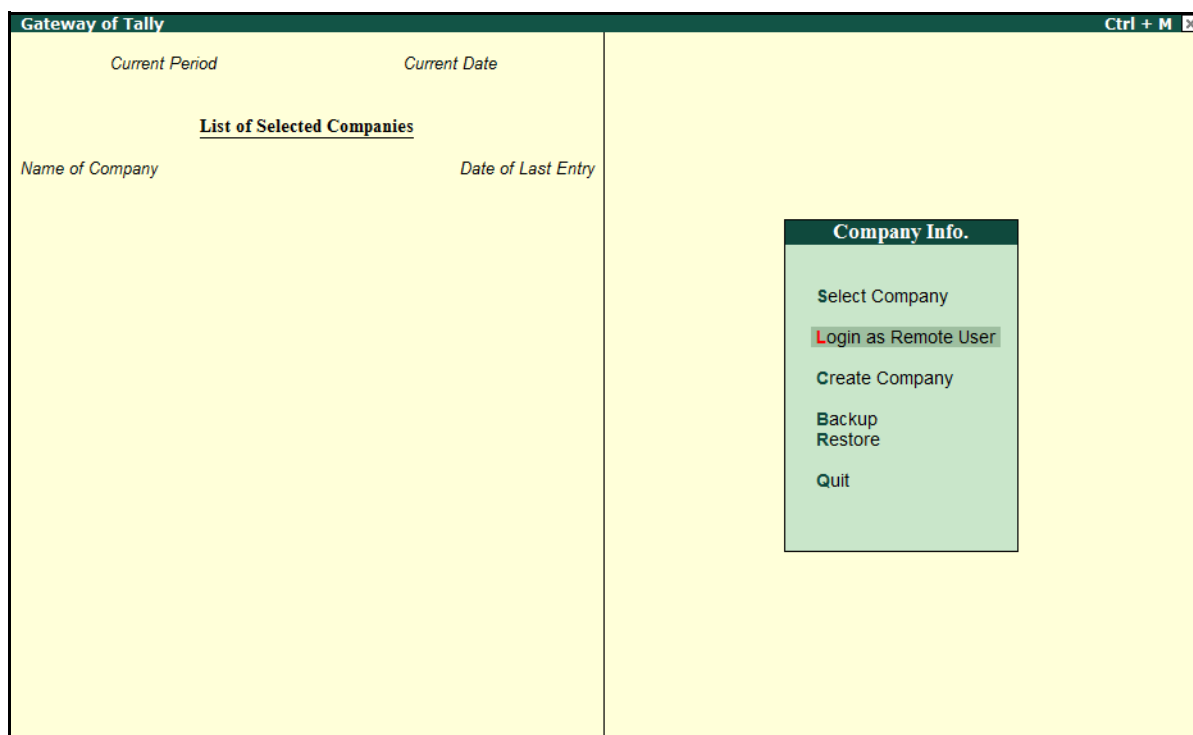


Figure 1.8 Connecting Tally.NET

Step 2:- Provide the User name and password



The screenshot shows a dialog box titled 'Login As Remote Tally.NET User'. It contains the following text and input fields:

Login As Remote Tally.NET User

Your E-Mail ID : auditor@abc.com

Your Tally.NET Password : [REDACTED]

*(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 Excise 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.

TABLE 2.



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

6.2.1 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>

```

6.2.2 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: $LastVoucherDate: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

6.2.3 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>
    </DESC>
  </BODY>
</ENVELOPE>
```

```
<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

6.3.1 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.

6.3.2 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>
```

6.3.3 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>
<LANGUAGE.LIST>
  <NAME.LIST TYPE="String">
    <NAME>Customer 1</NAME>
    <NAME>Alias</NAME>
  </NAME.LIST>
  <LANGUAGEID> 1033</LANGUAGEID>
</LANGUAGE.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.

7.2.1 Fetching the Methods

Whenever a collection is referred to in a Report, the required methods need 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
```

7.2.2 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, the 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
```

7.2.3 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

7.3.1 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
```

7.3.2 Separate 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

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:

A 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 i.e. 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

3.1.1 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

```

3.1.2 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 loses 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
```

3.1.3 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
```


3.1.4 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
```

3.1.5 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"
```

3.2 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

4.1.1 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 conjunction 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.

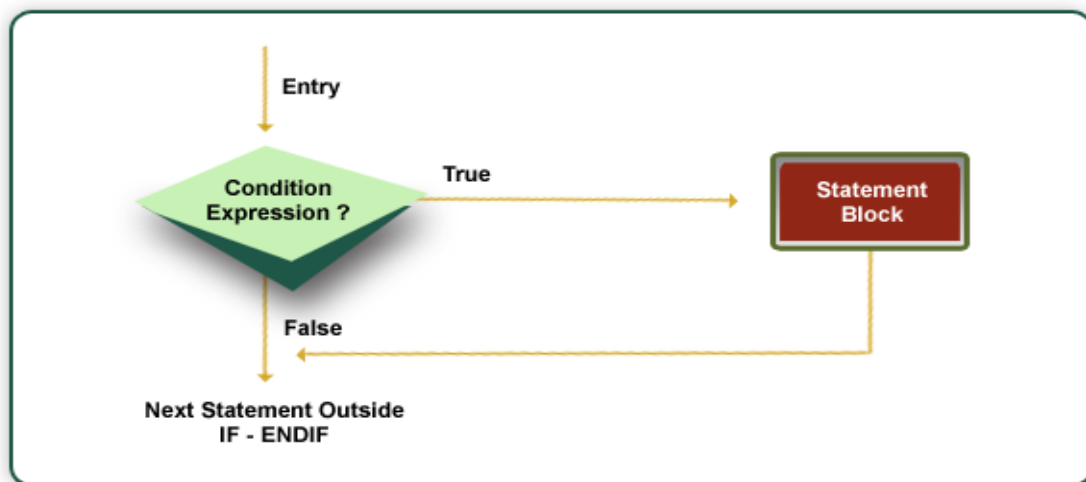


Figure 1.1 Flow Chart for IF – ENDIF

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
```

4.1.2 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

```

4.1.3 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.

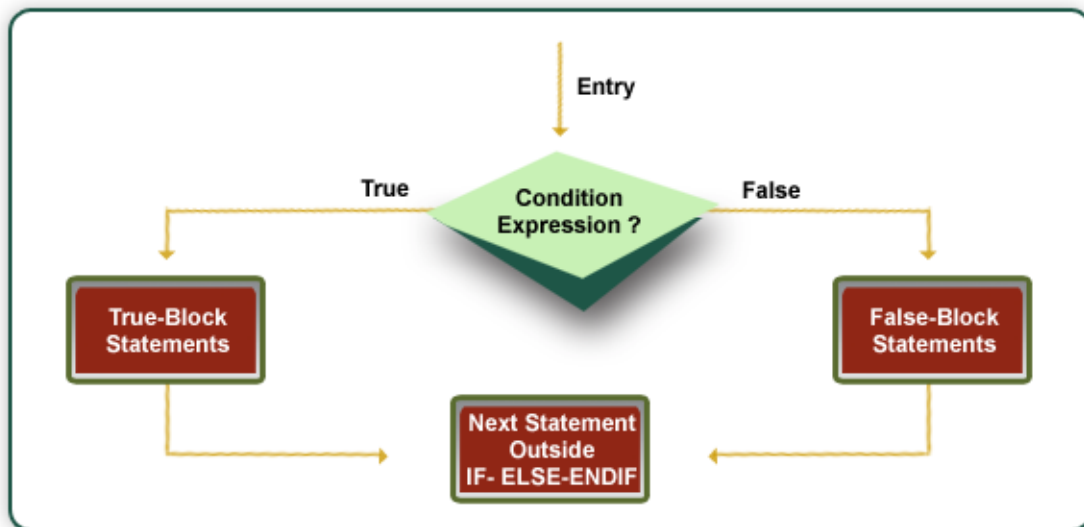


Figure 1.2 Flow Chart for IF – ELSE - ENDIF

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
05 : RETURN  : ##C
06 : END IF
07 : ELSE
08 : IF      : ##B >  ##C
09 : RETURN  : ##B
10 : ELSE
11 : RETURN  : ##C
12 : END IF
13 : END IF
```

4.2 Looping Constructs

4.2.1 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.

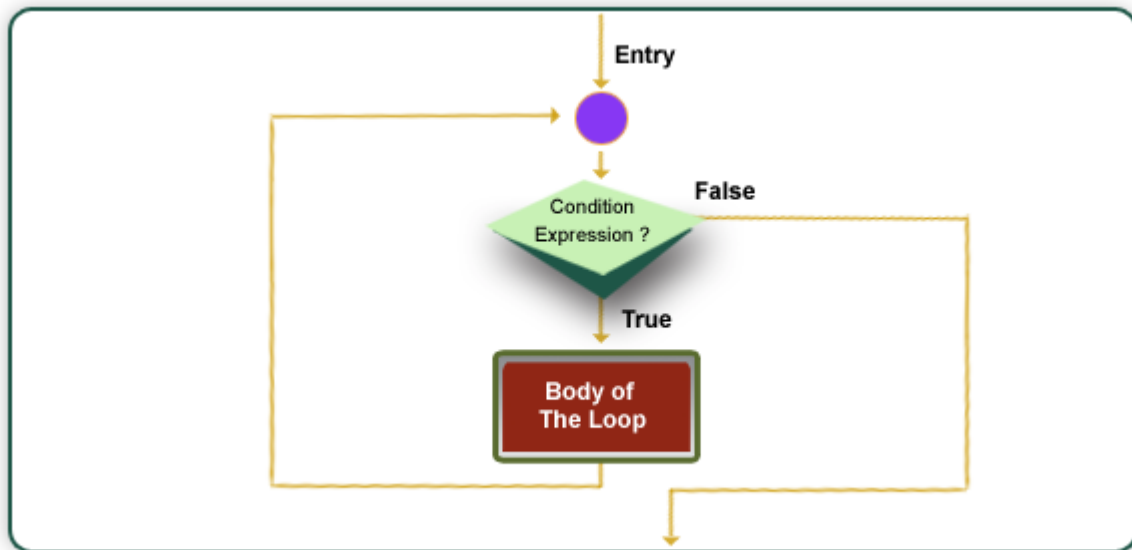


Figure 1.3 Flow Chart for WHILE – ENDWHILE

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
  
```

4.2.2 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.

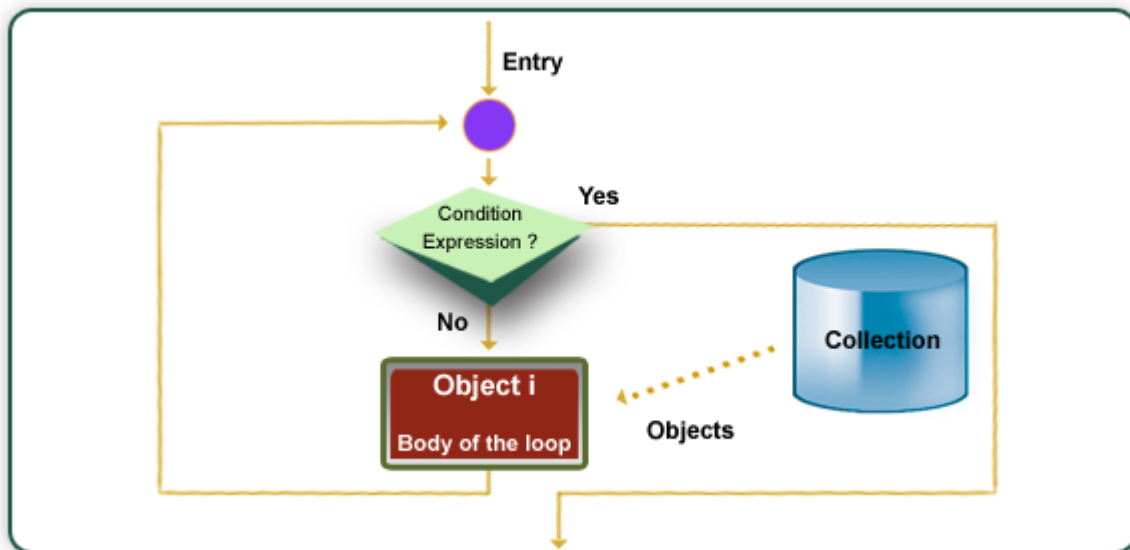


Figure 1.4 Flow Chart for WALK COLLECTION – ENDWALK

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.

4.3.1 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 COLECCION – 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
```

4.3.2 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 COLECCION – 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
```

4.3.3 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 : ET 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.

4.3.4 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     : DispStatutoryRpts
```

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 "Tally-MAIN"
- 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.

7.1.1 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

```

7.1.2 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.

7.1.3 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
```

7.1.4 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.

7.2.1 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 Ledger Creation screen

```
[Function : CreateReport]
```

```
01 : Create : Ledger
```

7.2.2 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
```

7.2.3 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 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
```


7.2.4 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.

7.2.5 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

7.2.6 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:"MigrationVouchers.."
```

7.2.7 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
```

7.2.8 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
```

7.2.9 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
```

7.2.10 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
```

7.2.11 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
```

7.2.12 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
```

7.2.13 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
```

7.3 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.

7.3.1 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.

7.3.2 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
```

Inserts 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.

7.3.3 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.

Example:

```
01: SET VALUE : Ledger Name : $LedgerName
02: SET VALUE : IsDeemedPositive : $IsDeemedPositive
03: SET VALUE : Amount : $Amount
```

The above statements setting the values of Ledger Entries Object from the current Object context.

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
```

7.3.4 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'

7.3.5 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
```

7.3.6 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

7.3.7 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

7.3.8 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 dotted notation.

7.3.9 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.

7.3.10 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 : SET 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 Target Object
06 : INSERT COLLECTION OBJECT : Ledger Entries
;;Set the Values of the Target 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 Tally.ERP 9 Release 1.5

In the release the major enhancements have taken place at the collection level and variable framework as a whole, along with introduction to a new variable type "List Variable "

This documents talks in depth on the usage of context free constructs Source Var, Compute Var and Filter Var in collection. The usage of \$\$TgtObject is now extended to work with collections as well.

Also covered are a few general enhancements, function \$\$ContextKeyword and attribute Trigger Ex at the field level which allows addition of values dynamically to table using TDL functions and expressions.

Bug fixes and the enhancements in the previous release be referred from here as well

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.

1.1 Source Var

The attribute Source Var evaluates the value of based on the source object.

Syntax

```
Source Var : <Variable Name> : <Data Type> : <Formula>
```

<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

1.2 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>
```

<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
```

1.3 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>
```

<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
```

1.4 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, \$\$Filter-Value, \$\$FilterAmt can be reduced.

1.5 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.

1.5.1 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.

1.5.2 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.

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.

2.1 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>]

<Function Name> is name of the function.

<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.

2.2 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

2.2.1 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.

Syntax

```
LIST ADD : <List Var Name> : <Key Formula> : <Value Formula>
```

LIST SET is alias for action LIST ADD.

<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.



The same List is considered in explaining in further examples.

:

To add multiple values dynamically in the list variable, looping constructs WHILE, WALK COLLECTION etc can be used.

▣ **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>]
```

LIST REMOVE is an alias for LIST DELETE.

<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.

2.2.2 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>
```

<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
```

2.2.3 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>
```

<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.

2.2.4 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:

1. List Key Sort
2. List Value Sort
3. 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 its 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.

Syntax

LIST KEY SORT : <List Var Name>[: <Asc/Desc flag> : <Key Data Type>]

LIST SORT is an alias of action LIST KEY SORT.

<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>]
```

<List Var Name> is the name of list variable.

<Asc/Desc> 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.

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>
```

<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.

2.3 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.

2.3.1 Function ListValue

As explained earlier the function List Value to access the value from a list variable

2.3.2 Function ListCount

\$\$ListCount gives the total number of values available in the list variable.

Syntax

```
$$ListCount:<List Var Name>
```

<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.

2.3.3 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>
```

<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 executed 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.*

2.4 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
```

<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*'.

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 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
  01 : Action : Display : ##TestFunc
```

;;The function Test Func executes a dynamic action which takes Action parameter as the parameter passed to the function

4. New Functions

In this release two new functions are introduced - \$\$TgtObject and \$\$ContextKeyword.

4.1 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

`$$TargetObject:<String Expression>`

Where,

<String Expression> the expression and will be evaluated in the context of Target Object

4.1.1 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`.

4.1.2 `$$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 be 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. \$ItemMaxAmt gives the maximum amount received for individual item. The conditions checks if the evaluated \$ItemMaxAmt 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	

4.2 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 definition, 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.

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 expression 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      : ""
```

In 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.

6. New Actions

Two new actions LogObject and LogTarget are introduced to log the object, its method and collection contents.

6.1 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
```

6.2 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 Targett[:<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
```

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.

7.1.1 /NOINITDL

This parameter will start Tally.ERP without loading any **TDL** specified in the Tally.ini file.

Syntax

/NOINITDL

7.1.2 /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.

7.1.3 /NOINILOAD

This parameter will start Tally.ERP without loading any **Company** specified in the Tally.ini file.

Syntax

/NOINILOAD

7.1.4 /LOAD

This parameter starts Tally.ERP and the specified company is loaded and can be specified multiple times.

Syntax

```
/LOAD:<Company Number>
```

7.1.5 /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.

7.1.6 /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.

7.1.7 /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.

7.1.8 /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.

7.1.9 /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.

7.1.10 /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.

7.1.11 /NOINITDL & /TDL

`"C:\Tally.ERP 9\Tally.exe" /NOINITDL "/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.

7.1.12 /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**.

7.1.13 /VARIABLE

`"C:\Tally.ERP 9 \Tally.exe" /VARIABLE:MyLogicalVar:Logical`

The above command starts Tally application and with a logical variable **MyLogicalVar**.

7.1.14 /SETVAR and /ACTION

```
"C:\Tally.ERP9 \Tally.exe" /SETVAR:ExplodeFlag:Yes /LOAD:00009/ACTION:DIS-  
PLAY:TrialBalance
```

The above command set the value of variable **ExplodeFlag** to **YES** and directly displays **Trial Balance** report.

7.1.15 /PREACTION

```
"C:\Tally.ERP 9 \Tally.exe" /LOAD:00009 /PREACTION:CALL:BackupBeforeEntry
```

The above command starts Tally application, loads the company **00009** and calls the function The above command starts Tally application and loads the Entry before displaying the main menu.

7.1.16 /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.

7.1.17 /NOGUI

```
"C:\Tally.ERP 9 \Tally.exe" /NOGUI /ACTION:CALL:BackupSchedule
```

The above command starts Tally application, executes the function **BackupSchedule** without displaying the tally window.

8. Enhancements in Previous Release

8.1 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.

8.2 Action Browse URL is Enhanced

The action **Browse URL** now accepts parameters. 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" "Test.txt"
```

```
[Form : Hyperlink]
```

```
Parts      : Hyperlink
```

```
Button     : Open Notepad
```

8.3 Collection Enhancements

8.3.1 The attribute Data Source now supports "HTTP XML"

While creating a HTTP Collection, the encoding format can be specified in the Data Source attribute.

Data Source attribute now supports "HTTP XML" as file type in addition to "File XML".

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 : HTTP Xml : "http:\\localhost\\MyFile.xml": ASCII
```

8.4 New Function – \$\$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
```

8.5 Action – SET VALUE

The action SET VALUE behaviour is enhanced wherein the value expression is now Optional.

Prior to Release 1.3, action SET VALUE required two parameters viz. the method name of Target object and Source Object. Now if the source method name is same as in Target Object, then the Source Object method name is optional.

Example:

Prior to Tally.ERP9 Release 1.3

```
01: SET VALUE : Ledger Name : $LedgerName
```

Tally.ERP9 Release 1.3 Onwards,

```
01: SET VALUE : Ledger Name
```

8.6 TDL Issues Resolved

This issue has been resolved.

- User defined methods in TDL can get cached and cause refresh issue during editing of objects.

This issue has been resolved.

- **SVUserName** variable was losing its value after company Login.

This issue has been resolved.

- ▣ **Modify Object** failing to access variables from function when function is called from On: Form Accept event.

This issue has been resolved.