



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.

© 2017 Tally Solutions Pvt. Ltd. All rights reserved.

Tally, Tally 9, Tally9, Tally.ERP, Tally.ERP 9, Tally.Server 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/April 2017

Preface

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

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

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

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

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

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

Happy programming folks!

Contents

Tally Definition Language – An Introduction

Tally Definition Language	4
<i>Comparison with other Languages</i>	4
The TDL Program - At a Glance	6
TDL Capabilities	7
TDL Features	7

TDL Components

Writing a Basic TDL Program	9
<i>Steps to create a TDL Program</i>	9
<i>Specification of TDL Files</i>	9
<i>Executing Multiple Files using Include Definition</i>	11
TDL Interfaces	11
'Hello TDL' Program	12
TDL Components	13
<i>Definitions</i>	13
<i>Attributes</i>	16
<i>Modifiers</i>	20
<i>Actions in TDL</i>	24
<i>Data Types</i>	25
<i>Operators in TDL</i>	26
<i>Special Symbols</i>	27
<i>Functions</i>	28

Symbols and Prefixes

Access Specifiers/Symbol Prefixes	29
General Symbols	30
The Usage of @ and @@	30
<i>Defining a Local Formula using @</i>	30
<i>Defining a Global Formula using @@</i>	31
The Usage of # and ##	31
<i>Referencing a Field using #</i>	32
<i>Modifying existing Definitions using #</i>	32
<i>Accessing value from a Variable using ##</i>	32
The Usage of \$ and \$\$	33
<i>Accessing a Method using \$</i>	33
<i>Calling an Internal Function using \$\$</i>	33
Commenting a Code using ;, ;; and /**/	34
Line Continuation Character (+)	34

Exposing Methods and Creating Procedures (_)	34
Reinitialize Definitions (*)	35
Optional Definitions (!)	35
Dimensions and Formatting	
Unit of Measurement	39
Dimensional Attributes	39
<i>Sizing/Size Attributes</i>	39
<i>Spacing/Position Attributes</i>	41
<i>Alignment Attributes</i>	42
Some Specific Attributes	46
<i>Inactive</i>	46
<i>Invisible</i>	46
Definitions and Attributes for Formatting	47
<i>Definition - Border</i>	47
<i>Definition - Style</i>	48
<i>Definition - Color</i>	50
<i>Attributes 'Background' and 'Print BG'</i>	50
<i>Attribute - Format</i>	51
Variables, Buttons and Keys	
Variable	55
<i>Attributes of a Variable</i>	55
<i>The Scope of a Variable</i>	57
<i>Modifying the Variable Value</i>	59
<i>Example - Variables</i>	60
Buttons and Keys	60
<i>Attributes of Buttons/Keys</i>	61
Objects and Collections	
Objects	63
<i>Tally Object Structure</i>	63
<i>Tally Objects Types</i>	65
<i>Object Context</i>	68
Collections	69
<i>Types of Collection</i>	70
<i>Sources of Collection</i>	71
<i>Creating a Collection</i>	71
Object Association	73
<i>Report Level Object association</i>	73
<i>Part Level Object Association</i>	74
<i>Line Level Object Association</i>	76
<i>Field Level Object Association</i>	77



Methods	77
<i>Types of Methods</i>	77
<i>Accessing Methods</i>	77
Collection Capabilities	80
<i>Basic Capabilities</i>	80
<i>Advanced Capabilities</i>	87
Actions in TDL	
Categories of Actions	93
<i>Action Association</i>	94
Components of Actions	96
Global Actions	97
<i>Action - Menu</i>	97
<i>Action - Modify Object</i>	99
<i>Action - Browse URL</i>	100
<i>Actions - Create and Alter</i>	101
<i>Actions - Create Collection, Display Collection and Alter Collection</i>	104
Object Specific Actions	106
<i>Menu Actions – Menu Up, Menu Down, Menu Reject</i>	106
<i>Form Actions - Form Accept, Form Reject, Form End</i>	106
<i>Part Actions – Part Home, Part End, Part Pg Up</i>	107
<i>Line Actions - Explode, Display Object, Alter Object</i>	108
<i>Field Actions - Field Copy, Field Paste, Field Erase, Calculator</i>	109
User Defined Fields	
What is UDF?	111
<i>Creating a UDF</i>	111
<i>Storing User Inputs in the UDF</i>	112
<i>Retrieving the value of UDF from an Object</i>	112
Classification of UDF's	112
<i>Simple UDF</i>	112
<i>Aggregate UDF</i>	114
Reports, Printing and Validation Controls	
Reports	119
<i>Tabular Reports</i>	119
<i>Hierarchical Report (Drill down Report)</i>	124
<i>Column Based Reports</i>	127
Printing	139
<i>Menu Action – Print/Print Collection</i>	140
<i>Button Action – Print Report</i>	140
<i>Page Breaks</i>	141
<i>Frequently Used Attributes and Functions</i>	144

Validation and Controls	146
<i>Field Level Attribute - Validate</i>	146
<i>Field Level Attribute — Unique</i>	147
<i>Field Level Attribute — Notify</i>	147
<i>Field Level Attribute - Control</i>	148
<i>Form Level Attribute - Control</i>	148
<i>Menu Level Attribute - Control</i>	149
<i>Report Level Attribute - Family</i>	149
Voucher and Invoice Customisation	151
Classification of Vouchers	151
<i>Accounting Vouchers</i>	151
<i>Inventory Vouchers</i>	152
<i>Accounting-cum-Inventory Vouchers</i>	152
The Structure of a Voucher Object	152
Customisation	154
<i>Voucher Customisation</i>	154
<i>Invoice Customisation</i>	162
Writing Remote Compliant TDL Reports	
Client/Server Architecture – An Overview	176
Tally Client/Server Architecture using Tally Software Services	176
<i>Tally.NET Server</i>	176
<i>Tally.ERP 9 Server</i>	177
<i>Tally.ERP 9 Client</i>	177
Setting up Tally.NET Server for Remote Access	178
Setting up the Client Tally	179
TDL – In a Client/Server Environment	180
TDL Enhancements for Remote	181
<i>Collection Enhancements</i>	181
<i>Report Level Enhancements</i>	184
<i>Function on Request</i>	188
<i>Action Enhancements</i>	189
Writing Remote Compliant TDL Reports	191
<i>Fetching the single Object</i>	191
<i>Repeating Lines over a Collection</i>	192
<i>Using the same Collection in more than one Report</i>	194
General and Collection Enhancements	
Definition, Attribute and Modifier Enhancements	195
<i>Attribute Enhancements</i>	195
<i>Modifier Enhancements</i>	200
<i>Behavioral change in System Definitions</i>	203

<i>Partial Attribute Support</i>	203
<i>Change in usage of 'BLANK' Keyword in Menu Items</i>	203
Enhanced Special Symbols	203
<i>Multi – line commenting in TDL source code using /* and */</i>	203
<i>Extension of modifying definitions using #</i>	204
<i>‘*’ (Reinitialize) Definition modifier</i>	204
Method Formula Syntax with Relative Object Specification	204
Enhancements - Object Association	206
<i>Report Level Object Association</i>	206
<i>Part Level Object Association</i>	207
<i>Line Level Object Association</i>	208
<i>Field Level Object Association</i>	209
Enhancements - Object Access via Interface Object	210
<i>Identifying Part and Line Interface object with ‘Access Name’</i>	210
<i>Value Extraction</i>	210
Bracket support in TDL	212
<i>During the Function Call</i>	212
<i>In the language syntax for nesting formulas</i>	213
<i>As a Mathematical Operator</i>	213
Action Enhancements	214
<i>Enhancements in Key Actions</i>	214
<i>New Actions</i>	215
Events introduced	222
<i>Event – On Form Accept</i>	222
<i>Event – On Focus</i>	222
User Defined Function	223
New Functions	223
<i>Function - \$\$IsObjectBelongsTo</i>	223
<i>Function - \$\$NumLinesInScope</i>	224
<i>Function - \$\$DateRange</i>	224
<i>Function - \$\$IsCollSrcObjChanged</i>	225
<i>Function - \$\$CollSrcObj</i>	225
Enhanced Collection Capabilities	226
<i>Aggregation and Reporting</i>	226
<i>The Summary Collection is available through Tally ODBC Interface</i>	238
<i>HTTP XML Collection (GET and POST with and without Object Specification)</i>	239
<i>Usage As Tables</i>	245
<i>Dynamic Object support for HTTP–XML Information Interchange</i>	249
Collection Capabilities for Remoting	251

User Defined Functions

Functions – In General	253
Functions – In TDL	254
Function – Building Blocks	254
<i>Definition Block</i>	255
<i>Procedural Block</i>	257
Valid Statements inside a Function	258
<i>Programming Constructs In Function</i>	258
<i>Actions used in a TDL Function</i>	267
Calling a Function	278
<i>Using the Action ‘CALL’</i>	278
<i>Using the Symbol Prefix ‘\$\$’</i>	278
Function Execution – Object Context	279
<i>Target Object Context</i>	279
<i>Parameter Evaluation Context</i>	279
<i>Return Value Evaluation</i>	279

What’s New in Release 5.5.2

Language Enhancements in Procedures (TDL)	283
<i>Action – Browse URL</i>	283
<i>Function – SysInfo</i>	283
<i>Attribute – Data Source</i>	283

What’s New in Release 5.4.9

Definition – Rule Set	287
<i>Attribute – Break On</i>	287
<i>Attribute – Walk On</i>	287
<i>Attribute – Rule</i>	288
<i>Attribute – Aggr Rule</i>	289
<i>Attribute – Rule Set</i>	289
<i>Attribute – Name Set</i>	290
<i>Attribute – Name Map</i>	290
<i>Function – EvaluateRuleSet</i>	290
Definition – Name Set	292
<i>Attribute – List Name/List</i>	293
<i>Function – NameGetValue</i>	293
Data Type – Flag Set	294
<i>Function – FlagGetValue</i>	294
<i>Function – FlagSetOR</i>	294
<i>Function – FlagSetAND</i>	295
<i>Function – FlagsIsAllTrue</i>	295
<i>Function – FlagsIsAllTrueFromLevel</i>	296

Function – <i>FlagsIsAnyTrue</i>	296
Function – <i>FlagsIsAnyTrueFromLevel</i>	297
Function – <i>FlagsCount</i>	298
Function – <i>FlagsCountFromLevel</i>	298
Function – <i>FlagGetDescription</i>	299
Function – <i>FlagsListDescription</i>	299
Function – <i>FlagsListDescriptionFromLevel</i>	300
Function – <i>AsFlagSet</i>	301
Data Type – Num Set	301
Function – <i>NumGetValue</i>	301
Function – <i>AsNumSet</i>	302
Other Enhancements	302
Attribute – <i>MAX</i>	302
What's New in Release 5.4.8	
Language Enhancements in Primitives (TDL)	303
Function - <i>IsEmpty</i>	303
Language Enhancements in Procedural (TDL)	303
Language Enhancements in Query (Collections)	303
Conditional <i>WalkEx</i>	303
Other Enhancements	304
Enhancements in Customisation using Productivity Suite	304
What's New in Release 5.3.8	
Action – Format Excel Sheet	305
What's New in Release 5.3	
Attribute – Confirm Text/Query Text	307
Action – Exec Excel Macro	307
What's New in Release 5.2	
Column-wise repeat of data over a collection	309
Function – <i>TplColumnObject</i>	309
What's New in Release 5.0	
Customisation using Productivity Suites	311
Other Language Enhancements	322
What's New in Release 4.8	
Data Importing Enhancements	329
Events Introduced	336
Action Enhancements	341
Function Enhancements	347
New Objects and Collection Attributes to support Banking	348
Miscellaneous Enhancements	350



What's New in Release 4.7

Developer Mode Enhancements 353
Event 'NatLangQuery' Introduced 357
ZIP - UNZIP 360
Columnar Capability in Edit Mode 366
New data types Introduced 367

What's New in Release 4.61

COM Data Types Support 389

What's New in Release 4.6

COM DLL Support in TDL 393
Developer Mode 400

What's New in Release 4.5

Platform Functions 417
Action Enhancements 422

What's New in Release 3.62

Multiple Orientation Support for Printing 425

What's New in Release 3.61

Action Enhancements 427
Function Enhancements 428

What's New in Release 3.6

Collection Enhancements 431
Action Enhancements 432
Platform Functions and Variables 433

What's New in Release 3.0

Collection Enhancements 439
Image Printing Capabilities 444
Enhanced Columnar Capability 448
Persisting Variables at System Scope in a User Specified File 458
New Events Introduced 462
Enhancement – Programmable Configuration 466
Optional Default TDL Loading 467
Refresh Issues in context of User Defined Function Evaluation 468

What's New in Release 2.0

TDL Procedural Enhancements 477
Variable Framework Enhancements 499
Event Framework Enhancements 513
Action Enhancements 515



TDL Enhancements for Remoting	517
Default TDL Changes	523
What's New in Release 1.8	
Invoking Actions on Event Occurrence - with System and Printing Events Introduced	533
Collection Enhancements	536
Evaluating expressions by Changing the Object Context with \$\$ReqOwner Introduced	557
Variable Framework with Compound Variables Introduced	573
Licensing Binding Mechanism	632
What's New in Release 1.61	
Narrowing Table Search	639
What's New in Release 1.6	
General Enhancements	641
Collection Enhancements	648
User Defined Functions Enhancements	657
New Functions	661
What's New in Release 1.52	
Collection Enhancements - Attribute 'Data Source' enhanced	665
Enhancements in User Defined Functions	666
New Functions	670
https URL support in Tally	671
What's New in Release 1.5	
Collection Enhancements	673
List Variables Introduced	676
Dynamic Actions	683
New Functions	684
New Attribute – Trigger Ex	687
New Actions	688
Tally Command Line Parameters	689
Appendix	

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 to 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 labelled 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, etc. 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 Users

Tally Software Services (TSS) is 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 client, addressing their Security and Privacy concerns.

1. Tally Definition Language

Tally Definition Language is the application development language of Tally. TDL has been developed to provide the user with the 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 re-usability. It comprises of Interface and Data objects. Interface Objects mainly determine the behaviour of the product in terms of user experience. Data objects are mainly used for data persistence in the Tally Database.

Any Tally.ERP 9 user can learn TDL and develop extensions for Tally. The entire source code is available as a 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 example, Assembly language or any native machine language.

Middle Level Languages

Middle Level Languages consist of syntax, rules and features just like the high level languages. However, they can also implement low level languages as part of the code. For example, 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 execution and the computer strictly follows it. The execution starts from the first line of the code to the last line, taking care of all the control statements and loops used in the program.

Fourth Generation Languages

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

E.g., Some 4GL are used to retrieve, store and modify data in the database using a single line instruction, whereas other 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 4G 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 analyse 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.

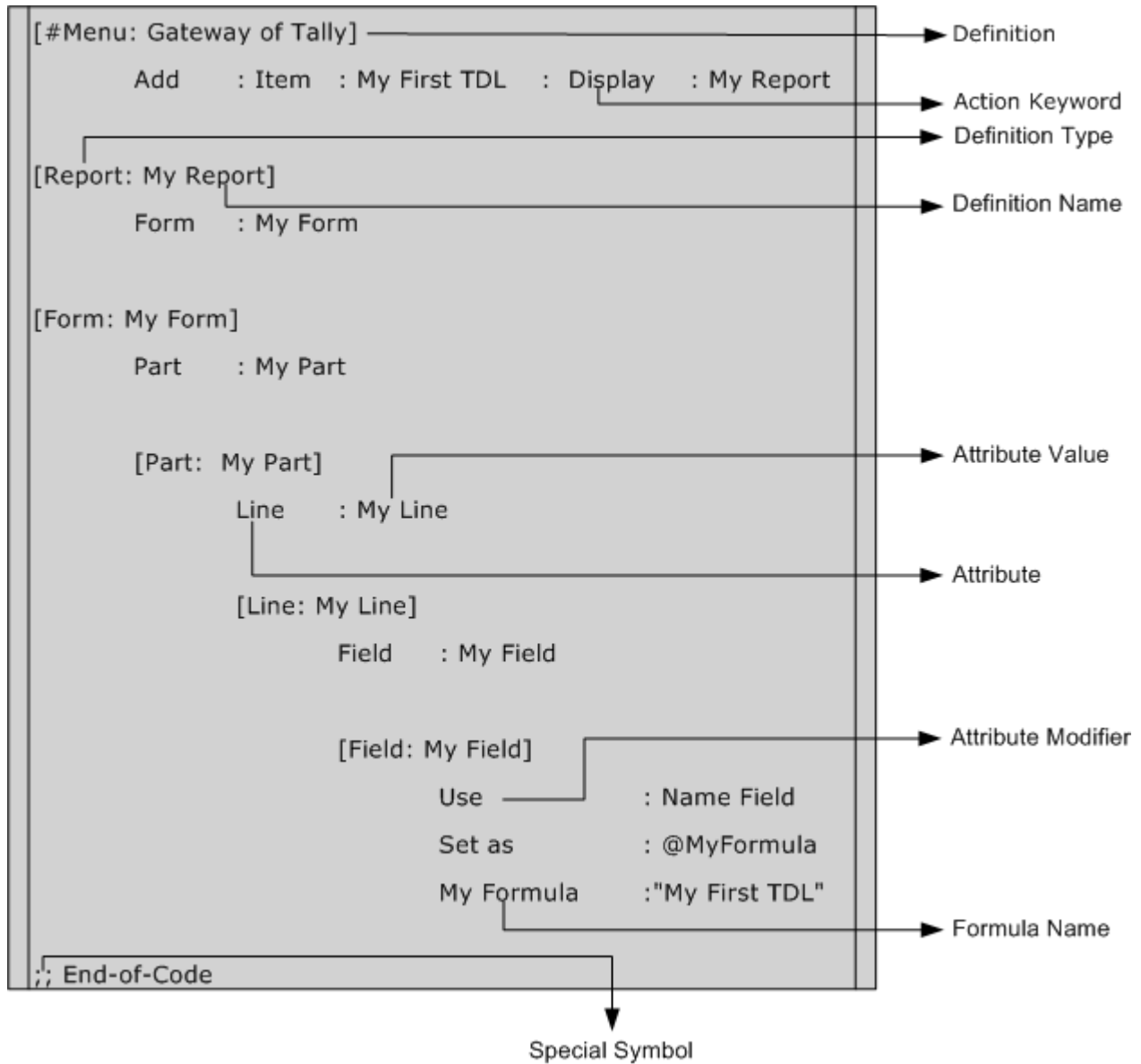


Figure 1.1 TDL Components

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 web-page 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 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. This capability is also provided in TDL, 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 name 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 that 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. During interaction, the user can select any sequence of actions. Based on his/her action, a particular segment of 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 times, the customer-specific requirements may seem like major 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 TSS.

- 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

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, an overview and basic functionality of each component involved in a TDL program will be provided.

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 a TDL file.

1.1 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, i.e.,.txt,.tdl and.tcp in Tally.ERP 9.

1.2 Specification of TDL Files

There are two ways of implementing the TDL code:

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

Specifying the 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 the 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 **TDL Configuration** screen, by going to **F12: Configuration-> Product & Features**, and clicking on **F4:Manage Local TDLs**. In this screen, set the value as YES for 'Load TDLs on Start up' and specify the Path\filename, with extension, in 'List of TDLs to preload on Tally Startup'. Following figure shows the TDL configuration screen:

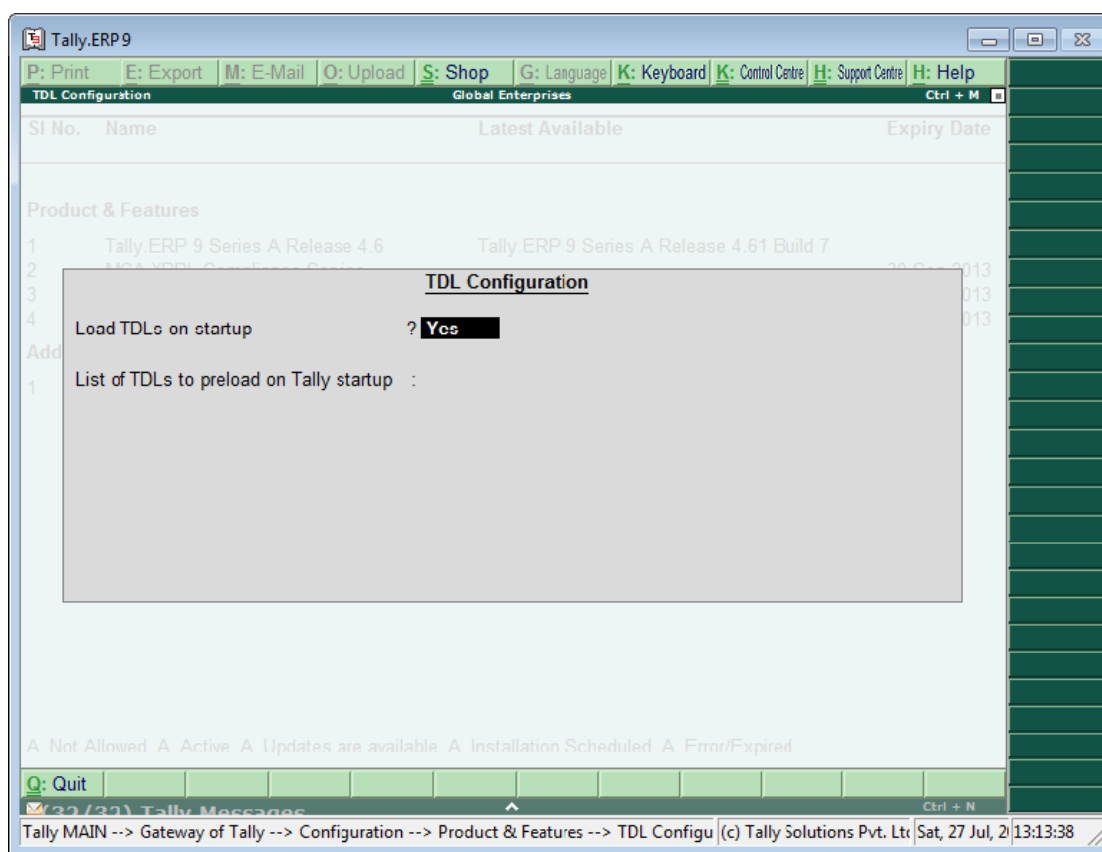


Figure 2.1 Specification of TDL files

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

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.

1.3 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, into the current file.

'Include', as the name suggests, gives us 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, either the file name or the complete path for the file has to be provided.

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]
```

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 definitions 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
- 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 TDL. The Report is executed from the existing Menu 'Gateway of Tally'.

Purpose: 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"
```

This code adds a new Menu Item 'First TDL' in the 'Gateway Of Tally' menu. When the Menu Item is selected the report, 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'. User inputs are not accepted in this report. The text 'Welcome to the world of TDL' is displayed in the Report, since it contains only one field.

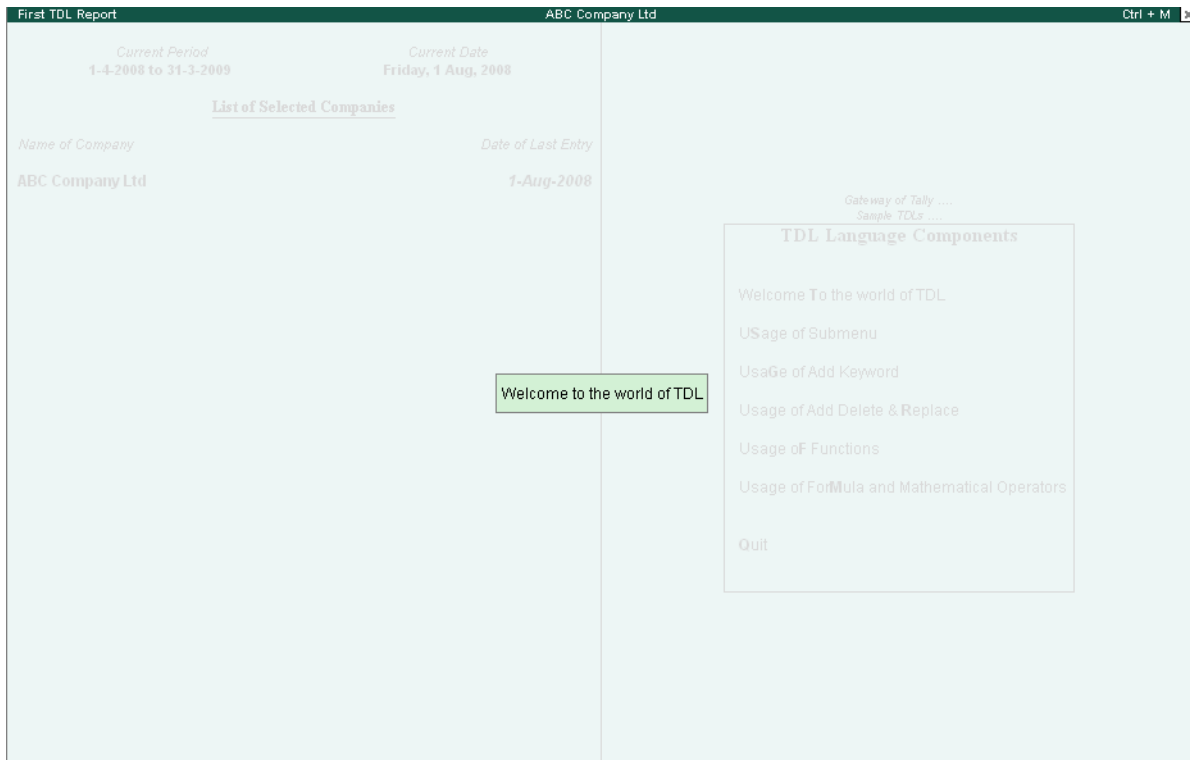


Figure 2.2 Output of Welcome to the world of TDL program

4. TDL Components

The TDL consists of Definitions, Attributes, Modifiers, Data Types, Operators, Symbols and Prefixes, and Functions. Let us now analyse 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 re-usability 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. All definitions start with an open square bracket and end with a closed bracket.

Syntax

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

Where,

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

<Definition Name> 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 this example, the type of definition is **Part** and the name of the definition is **PartOne**.

Types of Definitions

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 to create a user interface are referred to as Interface definitions. The definitions in this category are Menu, Report, Form, Part, Line, Field, 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., any 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: A Part consists of one or more Lines.

Line: A Line consists of one or more Fields.

Field: It is the place where data (constant or variable) is actually displayed/entered.

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

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

Data Definitions

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

Object: An object is the definition which consists of 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, though, possible to perform modifications on it. A Ledger/Group is an example of an internal object.

An object can further contain an object/objects.

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 behaviour 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 to as Formatting Definitions. The definitions in this category are Border, Style and Color.

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

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.

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.

Integration Definitions

Definitions which make 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' definition 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.

Action Definitions

The action definitions allow the user to define an action, to take place 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 a key combination. The action is performed when the associated key combination is pressed.

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.

Examples 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 behaviour 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 associated to a given attribute either directly, or through symbols and prefixes. Apart from a direct value association of the attribute, there are ways to associate alternate values dynamically, based on certain conditions prevailing at runtime.

Syntax

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

Where,

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

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

Example:

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

Classification of Attributes

The classification of attributes is done based on the number of values they accept, 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 'Single' type.

Example:

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

In this field, the string "TDL" is displayed, as 'Set As' is 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. This attribute 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 attribute 'Repeat' is an example of '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"
```

The values "Hello" and "TDL" are being assigned to the variables Var 1 and Var 2, respectively.

Triple and Triple List

Triple type attributes accept three values, and can't be specified multiple times.

Example:

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

Triple List type attributes accept three values, and can be specified multiple times.

Example:

```
Aggr Compute : TrPurcQty: Sum : $BilledQty
Aggr Compute: 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
```

Here, the options 'Sales Analysis' and 'Purchase Analysis' are added to the 'Gateway of Tally' Menu. For the option '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 there are more than one forms, then the first form is displayed by default. When one is 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 one chooses a Report that has no Forms defined, Tally.ERP 9 assumes that the Form Name is the same as the Report Name and looks for it. If it exists, Tally.ERP 9 displays it. Otherwise, Tally.ERP 9 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. It specifies the names 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 code segment 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 specifies the Lines contained in a Part.

Syntax

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

Example:

```
[Part : HW Part]

Line : HW Line1, HW Line2
```

Line Definition Attributes**Field/Fields**

The attributes '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 typically used 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, the attribute **Info** gets the precedence, and the value set within the attribute **Info** overrides the value set within the attribute **Set as**.

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.

It 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 a definition or on an attribute. They are classified as Definition Modifiers and Attribute Modifiers, based on whether a definition or attribute is being modified. Definition Modifiers are #, ! and *. Attribute Modifiers are Use, Add, Delete, Replace/Change, Option, Switch and Local.

The modifiers can also be classified into two types, based on the mode of evaluation:

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

Static/Load time Modifiers

These modifiers do not require any condition at the run time. The value is evaluated at 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>
```

Where,

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

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

<Attribute Value> 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' and '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>]

Where,

<Attribute Value> 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>

Where,

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

<Old Attribute Value > and <New Attribute Value> can be either 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 part 'Part2'. Now, only the properties of 'Part2' are applicable.

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
```

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 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>
```

Where,

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

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.

Example:

```
[Field : FldMain]
Option : FldFirst : cond1
Option : FldSecond : cond2
```

The field **FldFirst** is activated when cond1 is true. The field **FldSecond** is activated when 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 the 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, 'Switch' statements 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 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
```

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

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]
```

Where,

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

<Definition/Variable Name> is the name of definition/variable, on which the action is 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 a business language, supports business data types like amount, quantity, rate, etc., apart from the other basic types. The data types are classified as Simple Data Types and Compound Data Types.

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

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

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	
Rate	Price

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 four types of operators in TDL are as follows:

Arithmetic Operators

+	Addition
-	Subtraction
/	Division
*	Multiplication

Table 2.2 Arithmetic Operators

Logical Operators

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

OR	Returns TRUE if either of the expressions 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

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 <expression 1> is less than the value of <expression 2>.
>/Greater Than/More	Checks if the value of <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

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 behaviour, 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, etc.

Syntax

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

Example:

```
$$SysName : EndOfList
```

Here, the function \$\$SysName 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 & Prefixes, and Functions.

Symbols and Prefixes

Introduction

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

In TDL, there are a few symbols which are used for specific purposes. Some symbols are used as access specifiers, i.e., mainly used to access the value of a method, variable, field, formula, etc. Some are used for general purpose, such as modifiers.

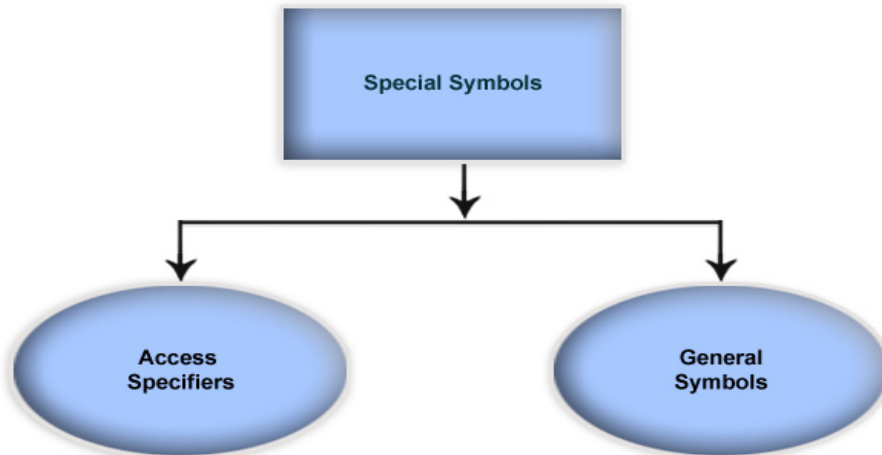


Figure 3.1 Symbol Categorization

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 a 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 SQL Procedure
*	Used to Reinitialize a Definition
!	Used to create an Optional Definition
#	Used as a definition modifier
Symbols	Usage

Table 3.2 General Symbols

3. The Usage of @ and @@

Formula

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

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

Classifications of formulae

- ❑ Local Formula
- ❑ Global Formula

3.1 Defining a Local Formula using @

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 by using the Symbol Prefix @.

Example:

```
[Field : CompanyNameandAddress]

    Set as : "Tally India Pvt Ltd, No 23 & 24, AMR Tech Park II, Hongasandra,+
            Bangalore"
```

This code can also 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
```

3.2 Defining a Global Formula using @@

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.

Example:

```
[System : Formula]  
    AmtWidth : 20  
  
[Field : RepTitleAmt]  
    Width    : @@AmtWidth  
  
[Field : RepDetailAmt]  
    Width    : @@AmtWidth  
  
[Field : RepTotalAmt]  
    Width    : @@AmtWidth
```

In this example, all the Fields assume the same width. If the width of the fields needs to be altered, a change is made only at the **[System: Formula]** Definition 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 this example, the value within the Field 'HW' is being set to the Field 'HW1'. In other words, 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 example, 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 this example, the existing Menu 'Gateway of Tally' (default Menu) has been altered to add the Item 'Hello World' and the Title of the Menu has been changed to 'Tally Gateway'. The existing Field 'LedParticulars' has also been altered to set its attribute 'Width' 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 the 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, the Global Variable value is assumed.

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 terms '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 the context to parent object and repeat the above steps

Example:

```
[Field : My Field]

    Set as : $Name
```

This 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 a 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 definition 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 they are 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 contrary, 'Option' executes all the Option statements matching the given conditions sequentially. Switch statements bearing different labels are similar to Option statements, as all 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 this code snippet, the condition specified in both the options will be checked, and the option satisfying the given condition will be executed. In this case, there is a possibility that more than one condition might be satisfied and get executed.

Example - Switch

```
[Line : MFTBDetails] Fields : MFTBName

    Right Fields   : MFTBDrAmt, MFTBCrAmt

    Switch : Case 1 : MFTBDtlsClsG1000 : $ClosingBalance > 1000

    Switch : Case 1 : 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 this 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 the label 'Case 1' will not be executed further, unlike 'Option'. The behaviour similar to 'Option' can be achieved by specifying different labels, if required.

Learning Outcome

- Access Specifiers and General symbols are the different special symbols used in TDL.
- The Access Specifiers @ and @@ are used for accessing the values 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 used for calling a function.

Dimensions and Formatting

Introduction

Dimensions are specifications. Dimensions in TDL are effective either in Display mode or in Print mode. Data in TDL does not have an absolute position of the dimensions specified, but relative. There are four definitions in TDL that attract dimensions. They are **FORM**, **PART**, **LINE** and **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)

2. Dimensional Attributes

Dimensional Attributes can be classified into two, i.e., Specific and General Attributes.

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

Table 4.1 Dimensional Attributes

The various dimensional attributes are as shown in the Table 4.1

2.1 Sizing/Size Attributes

Height and Width

The attribute 'Height' is used to specify the Height required for the Form, Part and Line Definitions, whereas the attribute 'Width' is used to specify the Width required for the Form, Part and Field Definitions. 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>
```

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
```

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
```

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.

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.

FullHeight and FullWidth

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

Syntax

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

Example:

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

FullHeight and FullWidth – Form Definition

The attribute 'FullHeight' decides whether to allow the form to consume the full 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 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.

FullHeight – Line Definition

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

FullWidth – Field Definition

The attribute 'FullWidth' decides whether the Field can consume the full available Width or not, depending on the logical value set. By default, the value set to this attribute is YES.

2.2 Spacing/Position Attributes

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 Form, Part and Line Definitions. Space Left and Space Right can be used in Form, Part and Field Definitions.

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
```

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.

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.

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.

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
```

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.

Indent – Field Definition

This attribute in the Field Definition is similar to the Space Left attribute, except that it 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 the expression itself. 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.

2.3 Alignment Attributes

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 in 'Form' as well as 'Part'

Definitions, whereas attributes 'Left Parts' and 'Right Parts' can be specified only in the '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

;; Attribute 'Left Parts' can be used only for 'Part' Definition

Right Parts : STKVCH Address

;; Attribute 'Right Parts' can be used only for 'Part' Definition

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 them from the bottom, in order to leave appropriate spaces. Similarly, Space Top impacts the Top Parts by moving them 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.

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 them from Right, in order to leave appropriate spaces. Similarly, Space Left impacts the Left Parts by moving them 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. In cases where the 'Vertical' Attribute is set to YES, all the parts within this part will be printed vertically. In such cases, the Left Parts will position at the Top of the Screen/Page and the Right Parts at the Bottom of the Screen/Page.

In cases 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 them from the bottom in order to leave appropriate space. Similarly, the attribute Space Top impacts the Top Parts by moving them from the Top in order to leave appropriate spaces. If the intent is to have multiple parts printed vertically, then the Part Attribute Vertical should be set to Yes. If the Vertical Attribute is set to No, then all the parts within this part will be printed horizontally. In such

circumstances, the Top Parts will be positioned at the Left of the Screen/Page while the Bottom Parts are positioned at the Right of the Screen/Page.



Both Parts and Lines are not allowed together within a Part. They are mutually exclusive entities. Either Parts or Lines can be specified at a time, within a Part.

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

Top Lines and Bottom Lines – Part Definition

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

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

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. 'Left Fields' starts printing from the Left to the Right of the Line, while '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' is not allowed, as by default, the 'Repeat' Attribute places the Field specified to the Right of the Screen/Page.

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.

Align

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

Syntax

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

Example:

```
Align : Right
```

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 in the Centre of the screen, and on the Left on printing. Depending on the width of the Form and page, the Form or Part will be displayed or 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 of the screen, and Top on print. Depending on the height of the form and page, the form will be displayed or printed, leaving equal amount of space on the top and bottom of the form.

Widespaced

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

Syntax

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

Example:

```
Widespaced : Yes
```

3. Some Specific Attributes

3.1 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 this 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.

3.2 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>
```

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 a certain condition.

Example:

```
[Field : Attr Invisible]

Set as      : "Invisible Attribute"

Invisible   : Yes
```

In this example, the Field attribute '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. Definitions and Attributes for Formatting

4.1 Definition - 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 predefined borders, which are part of default TDL, instead of user defined ones, 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>
```

Attributes - 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:** It forces double line to be printed. In its absence, single line is assumed.

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
```

Attribute - 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 value for a Black and White Monitor, and the second for 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
```

Attribute - PrintFG

This attribute 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
```

4.2 Definition - 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, i.e., 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>

Attribute - Font

It is a generic Font name supported by the Operating System. A Font is system dependent and we don't have any control over them. One can only select the required fonts from 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
```

Attribute - Height

This attribute should be specified without any measurement, since it is always measured in terms of Points. It can have value as a fraction, or as 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%
```

Attribute - Bold

The attribute 'Bold' can only take logical values/formula. In other words, it can take either a YES or NO. It signifies that the field using this Style should be printed in Bold.

Example:

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

Attribute - Italic

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

Example:

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

4.3 Definition - Color

The definition 'Color' is useful to determine the Foreground and Background Color for a Form, Part, Field or Border, 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>
```

Attribute - 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 the option to select from the wide range of 24 bit colors.

Example:

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

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

4.4 Attributes 'Background' and 'Print BG'

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
```

4.5 Attribute - Format

The attribute 'Format' is used in the Field definition. It 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 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, i.e., 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, in Display as well as Print Mode.
- The attribute 'Format' is used in the Field Definition. It determines the Format of the value being displayed/printed within the Field.

Variables, Buttons and Keys

Introduction

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

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

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

1. Variable

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

1.1 Attributes of a Variable

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

Type

This attribute determines the Type of the 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 behaviour of certain reports, based on its logical value, as configured by the user.

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 variable 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.

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.

Volatile

In cases where the 'Volatile' attribute in the Variable definition is set to YES, 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 the 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 the report R2, 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 **Group Name** Variable is set to YES, which means that the variable 'Group Name' can store multiple values, which have been received from multiple reports.

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 the columns need to be generated. Every time the Repeat is executed, the column for the 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 inputs from the user regarding the columns required. Variable **SVFromDate** gets repeated over the subsequent period in the Collection each time the column repeats.

1.2 The Scope of a Variable

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

- Local
- Global
- Field acting as a variable

Local

A Variable is termed as 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 the 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.

Global

A variable is termed as Global variable when it is defined under System Variable section. It means that the scope of the variable covers all 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 Local Variables above.

Syntax

```
[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.

Field Acting as a Variable

The **Variable** attribute in a 'Field' Definition is used to make the Field behave as a Variable. when value is entered/alterd 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.

1.3 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

1.4 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
```

;; Here, the variable value is modified with the Field contents specified in 'Set As'

In this 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.

2. Buttons and Keys

The actions in TDL can be delivered in three ways - 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 the key combination associated in the Key must be pressed to activate a key; whereas to activate a button, either it can be clicked or the key combination assigned for the button can be pressed.

2.1 Attributes of Buttons/Keys

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, 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 : NonColumnnar]
  Title : "No Columns"
```

Key/Keys

This 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 : NonColumnnar]
  Key : Alt + F5
```

Action

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

Syntax

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

Example:

```
[Button : NonColumnnar]
  Action : Set : ColumnnarDayBook : NOT ##ColumnnarDayBook
```


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 '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 'Objects' and 'Collections' 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 behaviour/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 collections. 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 collections. The structure is as shown in Figure 6.1.

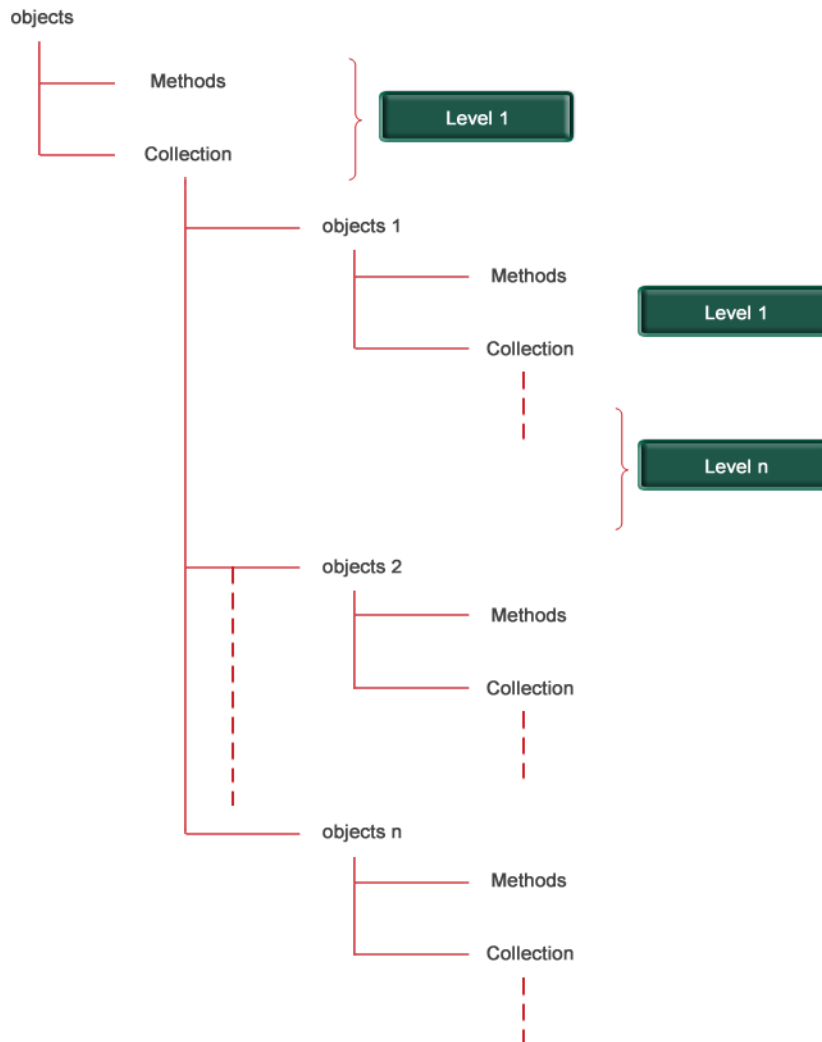


Figure 6.1 Tally Object Structure

Everything in TDL is an Object. As mentioned in the earlier chapters, Report, Menu, Company, Ledger, etc., 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 'BillwiseDetails'.

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 to 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

Objects in TDL are classified into 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.

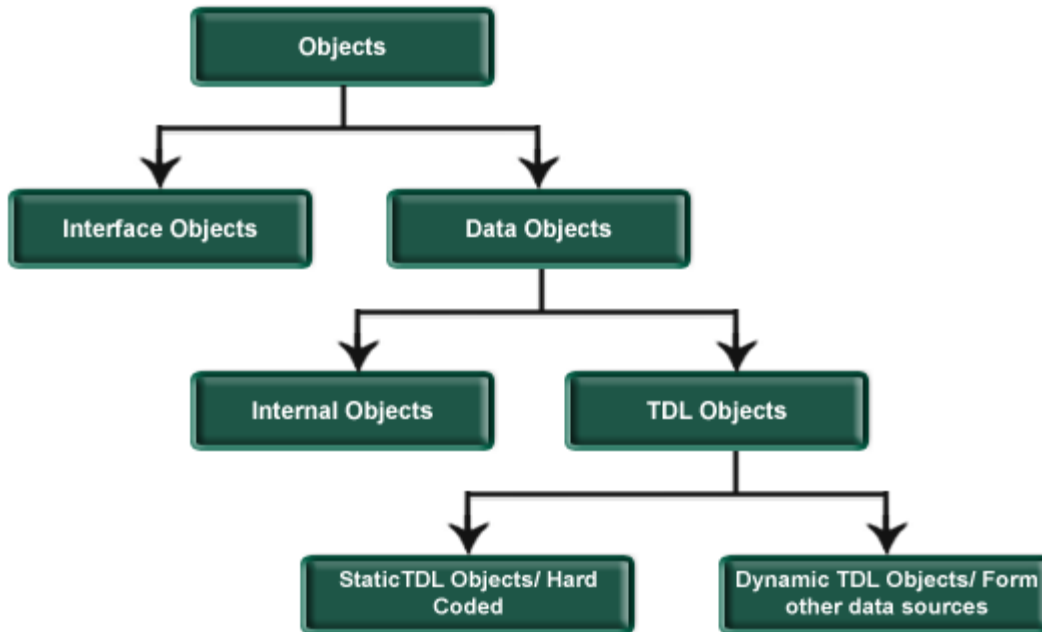


Figure 6.2 Classification of Objects

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 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.

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 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.

Example of Internal Objects and TDL Objects

Static TDL Objects/External Objects

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

Employee Details

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

In TDL, two objects have to be created such that EmpNo, Name, Date and Designation become the attributes 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
```

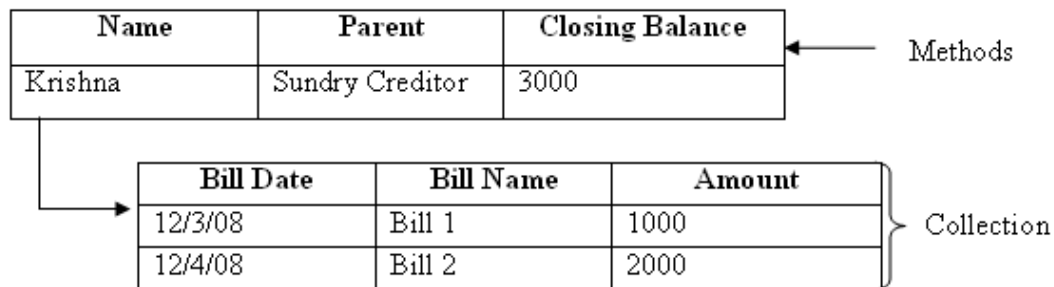
[Object : Emp 2]

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



This 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.

1.3 Object Context

Each Interface object exists in the context of a Data Object. An Interface object either retrieves information from the Data Object or stores information into 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 object context, consider the following 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, since there is no object in the context.

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 either 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 Internal Collections. The collections created by a user are called user defined collections. Object in a collection follow the Tally object structure, i.e., each object of the collection can contain Methods, Collections, and so on.

A collection can be a collection of objects or a collection of collections. Figure 6.3 shows the collection of objects.

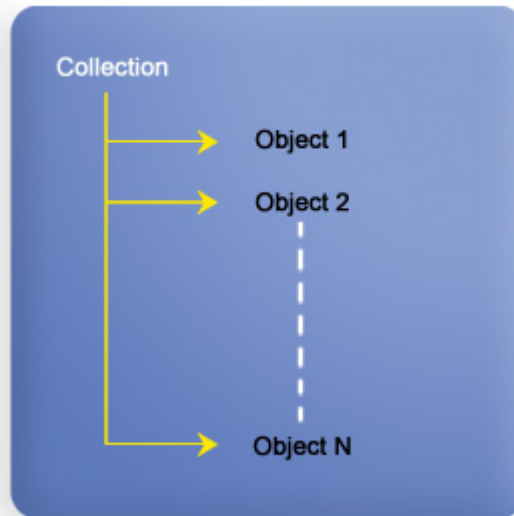


Figure 6.3 Collection of objects

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

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

2.1 Types of Collection

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

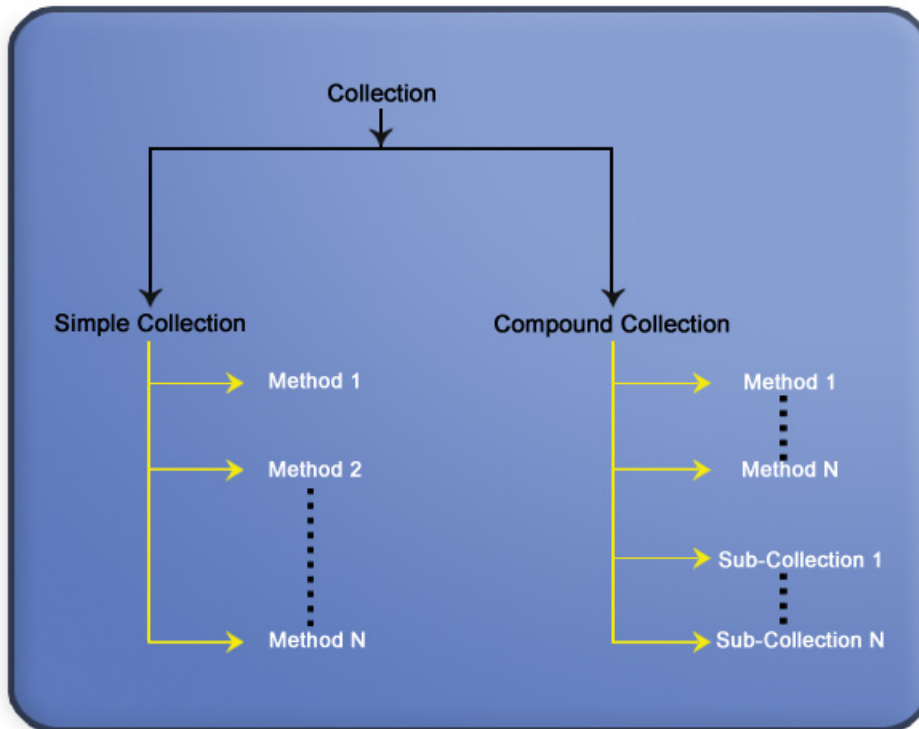


Figure 6.4 The classification of collection

Simple Collections

Simple collections contain only 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 artefact 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 Objects of this collection are 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.

Collection of Internal Objects

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

Syntax

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

Where,

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

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

Attribute – Type

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

Syntax

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

Where,

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

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

Example:

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

The collection **My Collection** consists of a collection of Ledgers which is an Internal object.

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 : <ObjectName>, <ObjectName>, ..... , <Object Name>
```

Where,

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

<Object Name> are the 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>
```

Where,

<List of Objects> 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 methods.

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 inherit 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.

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> is a Type of Primary Object.

<ObjectIdentifierFormula> 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.

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> is the name of the Collection of Secondary Objects.

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

<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 having 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.

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".

Attribute - Repeat

Syntax

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

Where,

<Coll Name> is the name of the Collection. If the Collection is present in 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 either be First or Last which denotes the position index.

<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.

Alternate 'Repeat' Syntax

Instead of specifying the '**<Coll Name>:[<Supplier Coll>:<SeekTypeKeyword>:<SeekCondition>]**', the new method formula syntax can be used as follows:

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 is 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 database, the storage name is prefixed with \$ symbol. TDL provides pre-defined methods and allows the user to create methods as well. Methods are classified as Internal or External.

4.1 Types of Methods**Internal Methods**

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

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/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.

4.2 Accessing Methods

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

Accessing data from the current Object

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

Syntax

```
$<MethodName>
```

Where,

<Method Name> 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>
```

Where,

<Method Name> is the name of the Method, which belongs to the Object.

<Object Name> is the name of the Object.

<Formula> 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, the following syntax may be used:

Syntax

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

Where,

<Method Name> is the name of the Method which belongs to the Collection.

<Collection Name> is the name of the Collection.

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

Example:

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

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> is given as the Collection Name [*<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 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].BillAllocations[1].Name
```

```
LedgerCondition : $LedgerName = @@InvPartyName
```

4. To get the Opening Balance 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 Collection as a Data Processing Artefact in TDL.

In previous sections, we saw that a Collection can contain objects from Tally Database or populate objects from external data sources as well. In further sections, we will discuss the capabilities of collection from the data processing perspective. Let's segregate the 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

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

Union

'Union' refers to creation of a Collection by combining multiple Collections. The total number of objects in the resultant Collection will be the sum of objects in all the Collections. The following figure shows a Collection of sub-collections, which can further be unions of collections, and so on.

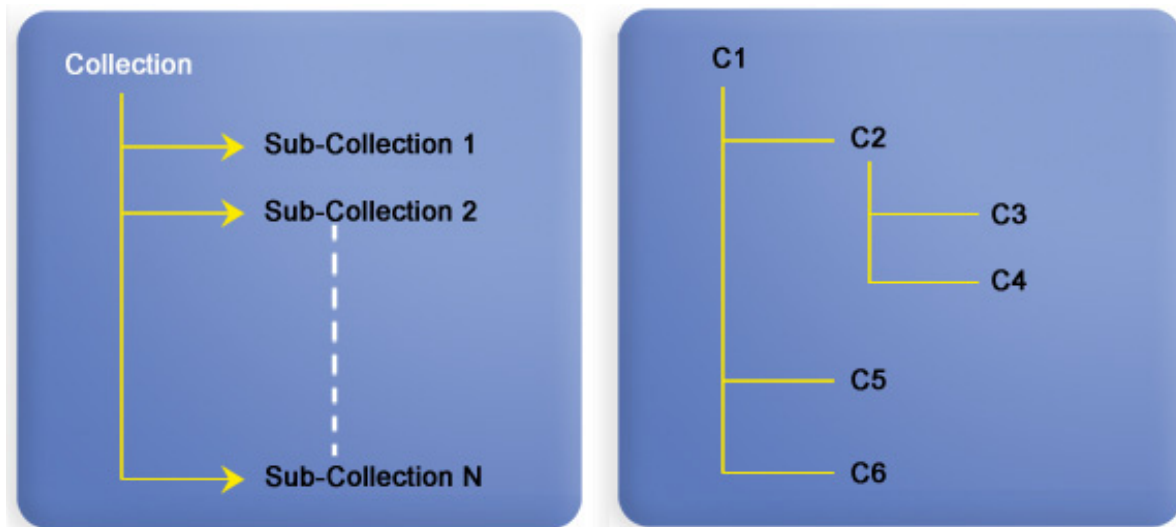


Figure 6.5 Collection of Sub-collection

This example shows that Collection C1 contains Collection C2 and Collection C3. Likewise, Collection C2 further contains Collection C4 and Collection C5. The attribute **Collection** is used to create a Union as follows:

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>
```

Where,

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

Example:

```
[Collection : GroupandLedger]
```

```
Collections : Group, Ledger
```

Here, the collections 'Group' and 'Ledger' are used, under the main collection **GroupandLedger**.

Filtering

If it 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**.

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 : ""
```

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. 'Belongs To' takes a logical value.

Syntax

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

Where,

<Logical Value> can be either YES or NO.

Example:

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

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, they can be separated by comma.

Syntax

```
Filter : <FilterName>
```

Where,

<Filter Name> 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 filter **Namefilter** is used to fetch only those objects, whose name contains the string “Ltd” or “Limited”.

Filtering Functions**Function - \$\$FilterAmtTotal**

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

Syntax

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

Where,

<CollectionName> is the name of a Collection,

<FilterExpression> is a System Formula.

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

<ValueExpression> is any valid expression, which is 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.

Function - \$\$FilterQtyTotal

It is similar to \$\$FilterAmtTotal, except that the Value Expression should evaluate to type Quantity.

Syntax

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

Where,

<CollectionName> is the name of Collection

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

Function - \$\$FilterCount

The function \$\$FilterCount is used to get the total number of Objects in a Collection, after the filters are applied.

Syntax

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

Where,

<CollectionName> is the name of a Collection.

<FilterExpression> 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)
```

It confirms whether the voucher has any Ledger under the Group **Bank** or **BankOD**.

\$\$IsLedOfGrp accepts two parameters and returns TRUE if parameter 1 is a ledger of a Group mentioned in parameter 2. GroupBankOD and GroupBank are functions which return the name of the reserved groups Bank OD and Bank.

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>
  
```

Where,

<CollectionName> is the name of the Collection.

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

<PositionSpecifier> denotes the position.

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

Example:

```

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

This 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

Function - \$\$GroupSundryDebtors

It returns the name of the group ‘Sundry Debtor’, even if the user renames it.

Syntax

```

    $$GroupSundryDebtors
  
```

Example:

```

[Collection : Sample Coll]

Type      : Ledgers

Child Of  : $$GroupSundryDebtors
  
```

This will populate the collection **Sample Coll** with all objects under the Group ‘Sundry Debtors’.

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, **\$\$GroupSundryDebtors** will populate the collection with all the objects that are under the Group ‘Sundry Debtor’ even if the user renames the group.

Function - \$\$GroupSundryCreditors

It returns the name of the group ‘Sundry Creditors’, even if the user renames it.

Syntax

```

    $$GroupSundryCreditors
  
```


Example:

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

This will populate the collection **Sample Coll** with all objects under the Group 'Sundry Creditors'.

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, \$\$GroupSundryCreditor will populate the collection with all the objects that are under the Group "Sundry Creditors", even if the user renames the group.

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.

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>
```

Where,

<List Of Methods> Is a comma separated list of methods. Sorting is done based on values of methods. The default sort order is ascending. Prefix Methods name with '-', for descending order.

Example:

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

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.

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 `$$CollectionFieldByKey`.

Function - `$$CollectionFieldByKey`

Syntax

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

Where,

<Method Name> is the name of the method.

<Key Formula> 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
```

Here, a search key is defined on \$name for collection **MyLedgers**. In the Field, value \$Closing Balance is retrieved based on ledger name. In this case, retrieval is faster than ordinary retrieval.

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

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`.

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>
```

Where,

<CollectionName> is the name of a collection.

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

Example:

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

It 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.”

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, and aggregation and grouping are Walk, By, Fetch, Compute, AggrCompute. A detailed discussion on enhancements for aggregation and Grouping using the new attributes will be covered in 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.

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>
```

Where,

<CollectionName> is the name of a Collection.

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

Example:

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

It 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 one is in the context of **Voucher** Object.

Function - \$\$CollQtyTotal

This function is used 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 Type **Quantity**.

Syntax

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

Where,

<CollectionName> is the name of a Collection

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

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>
```

Where,

<CollectionName> is the name of a Collection.

<ValueExpression> is 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. Here, **Height** is an external Method of Object **Inventory Entry** in a **Voucher**.

Usage as Tables

TDL allows to display the values obtained from the collection as a pop-up table. Earlier, the values of voucher and the ODBC data couldn'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.

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.

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 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”.

Collection Capabilities for Remoting

Enabling access to your organizational data ‘any-time, any-where’, and yet being truly usable, is what Tally.ERP 9 delivers. With remote access through Tally.NET Server, 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 database.
- 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 to as interface objects.

- Data is actually stored in the Data Objects. Data objects are classified into two types namely Internal objects and User defined/TDL Objects.
- A collection can be a collection of objects or a collection of collections.
- Collection, the data processing artefact of TDL provides extensive capabilities to gather data, not only from the Tally database, but also from external sources using ODBC, DLLs and HTTP.
- In TDL, Object association can be done at the following levels:
 - Report Level Association
 - Part Level Association
 - Line Level Association
 - Field Level Association
- Each piece of information stored in a 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 Object like Menu, Form, Line or Field.

1. Categories of Actions

Actions can be classified into two broad categories, viz

- Global Actions
- Object Specific Actions

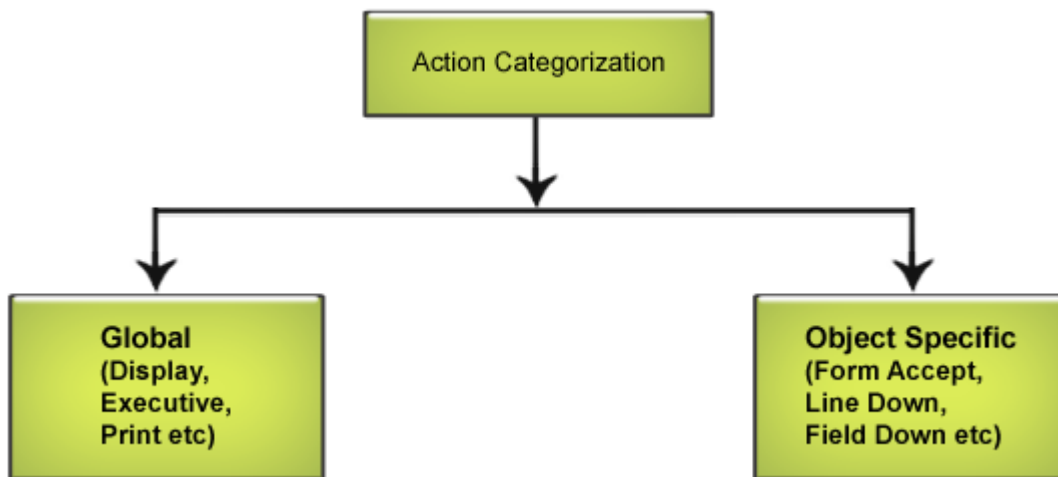


Figure 7.1 Action Categorization

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 the 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 a User Interface Object
Can be originated by a Menu, Button/Key or a Field	Can be originated by a Menu, Form, Line or a Field
Performed on a Report or a Menu	Performed on the relevant Interface Object
Example: Create, Display, Alter, Print, Print Report, Modify Object, Display Collection, etc.	Example: Line Up, Line Down, Explode ('Line' Object), Form Accept, Form Reject ('Form' Object), etc.

Table 7.1 Action Categorization

1.1 Action Association

Actions can be associated at various levels.

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,

<Action Keyword> can be any Global Action.

<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 this example, a Menu **Commonly Used Reports** is defined with 2 Items, viz.,

1. An Item Trial Balance is added displaying the 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 Menu, so the Action Value required is a Menu Name.

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,

<Action Keyword> can be any Global or Object Specific Action.

<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 Parameter and Actions **Create**, **Display**, **Alter**, etc., require a Report Name.

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** are not mandatory. If the Action Parameter is specified, then it prints the specified Report, else it prints the current Report.

Action Association at 'Field' Definition

Action Association at Field is done using Action Keyword with parameters and optional condition.

Syntax

```
[Field : <Field Name>]

    Action Keyword : <Action Parameters>[: Condition]
```

Where,

<Action Keyword> can be both Global or Object Specific Actions.

<**Action Parameters**> can be the Value on which these Actions could be performed.

<**Condition**> is optional. It restricts the action to be performed only if the condition returns TRUE.

Example:

```
[Field : My Trial Balance]

    Display : Group : $$IsGroup

    Display: Ledger : $$IsLedger
```

In this example, 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

2. Components of Actions

Any Action is always executed with respect to two contexts:

- Originator
- Executor

The **Originator** is one that originates the Action, viz., Menu, Form, Line or Field, e.g., a Down Arrow Key pressed. The event is passed from the current Report to the associated Form, Parts, Lines or Fields. Keys could be associated in Menu, Form, Line or Field. If the activated Key is found in Form, it searches further for Line Association, and then continues till Field. The Lowest Level Key Association gets the highest precedence. If the Key is associated at Form as well as Field, the Key Association at Field Level gets executed. In this case, the Field is the Originator.

The **Executor** 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 the executor 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, it will be executed by the Part to move the current focus to the subsequent line. Hence, Part is an executor of the Action 'Line Down'.

Originator	Executor
The Originator initiates the action by associating the Key or a Button	The Executor executes the action associated with the Key or Button, initiated by originator
Global Actions can be originated by Menu, Button/ Key or a Field, and Object Specific Actions by a Menu, Form, Line or a Field	Global Actions are executed by the originator object. However, Object Specific Actions can be executed by Objects other than originator

<p>The sequence followed to gather all Keys originating within a Report is Top to Bottom, i.e., from Report to Field definition. The lowest in the hierarchy gets highest precedence, e.g., if the same key is associated at both Form and Field definitions, the Key at Field Definition is considered for execution.</p>	<p>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 gets the highest preference, e.g., if the same key is relevant for both Part and Line definitions, the Key will be executed in context of the Line Definition.</p>
<p>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 executor here.</p>	<p>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 executor is the Part.</p>

Table 7.2 Components of Actions

3. Global Actions

As discussed, 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 fulfil the user requirements. Global Actions are mainly performed on three principal definition types, namely Report, Collection and Menu. Some frequently used Global Actions are discussed below:

3.1 Action - Menu

The Action 'Menu' acts only on the 'Menu' Definition, and vice versa. The value of the '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 the Action 'Menu', along with further Menu Definitions

```
[#Menu : Gateway of Tally]

    Add : Key Item : Sample Item : F : Menu : Sample Final Accounts
```

;; Menu Definition for the Menu to be displayed when the above Item is activated

```
[Menu : Sample Final Accounts]

    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 this example, the Default Menu Gateway of Tally is altered to add a new Item Sample Item, with the 'Menu' action displaying the Sample Final Accounts Sub Menu.

Sub Menu Sample Final Accounts will display all 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", the 'Display' Action takes the Report Name as its parameter, and is used to display the Reports, as is specified.

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 Definition to activate a Menu

```
[Button : Final Accounts]
```

```
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 Definition to Display a Report

```
[Button : Balance Sheet]
```

```
Key      : F6
```

```
Action : Display : Balance Sheet
```

/ Since the above Button activates a Report, it can be associated to both a Menu and 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 this example:

- 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 example, the Button **Final Accounts** is attached to Form 'Group Summary', which is incorrect since the Menu cannot be called from a Report.
- Another Button **Balance Sheet** is added to display Report **Balance Sheet** which is enabled in all Reports, using Form **Group Summary**, and also in the Menu **Display Menu**.
- The Button **Balance Sheet** can be attached to a Report as well as to a Menu, since the Report can be acted upon by a Report as well as a Menu.

3.2 Action - Modify Object

This action alters the methods of an Object at any level in Object Hierarchy. It supports modifying multiple values of an Object by specifying a comma-separated list of **Method: Value** pairs.

Syntax

```
Action : Modify Object : <PrimaryObjectSpec>.<SubObjectPathSpec>.
    MethodName : Value>[,Method Name : <Value> , ...]
    [,<SubObjectPathSpec>.MethodName :<Value>, ...]
```

The specifications given for <PrimaryObjectSpec>, <SubObjectPathSpec>, Method Name remain the same as described in the New Method syntax section in the topic 'Objects and Collections'.

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 second value onwards, Sub Object specification is optional. If Sub Object Specification is specified, the context is assumed to be the Primary Object specified for 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"
```

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 Menu cannot work on scopes like Selected, Unselected, etc., the scopes specified are ignored.
- ❑ Any formula specified in the value and evaluated, assumes Menu Object as 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.

3.3 Action - Browse URL

The Action 'Browse URL' is used to provide a link to any web browser, with a URL formula passed as a parameter.

Syntax

```
Action : Browse URL : <URL Formula>
```

Example: Field acting as a hyperlink

```
[Key : Execute Hyperlink]

Key      : Left Click

Action  : Browse URL : "www.tallysolutions.com"

[Field : Hyperlink Company]

Color   : Blue  Border : Thin Bottom
```

```
Key      : Execute Hyperlink
Set as   : "Tally Solutions Pvt. Ltd"
Local    : Key : Execute Hyperlink : Action : Browse URL: +
          http://www.tally.co.in
```

3.4 Actions - Create and Alter

'Create' and 'Alter' Actions act 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 depends on 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 value entered by user in Internal Method Name available within Object associated at Report*/*

```
[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 the user defined Groups. Field Attribute Table helps to restrict the user input to a predefined list*/.*

In this example:

- The Default Menu **Gateway of Tally** has 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'.
- Name and Group of the Ledger are stored in Internal Methods Name and Parent.

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"
```

;; Associating the Button to the Report

```
[Form : My Bank Vouchers]
```

```
Button : My Reco Button
```

In this example:

- Button **My Reco Button** is defined with 'Alter' action to alter the Report **Bank Recon** on pressing the Alt + F5 Key. It is used for entering dates in the Bank Reconciliation Report.
- The Button **My Reco Button** is associated to the Form **My Bank Voucher**

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 Objects

```
  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 a user defined Report defined while Ledger Creation

In the example mentioned above:

- ❑ Two default Keys are associated to a 'Line' Definition, that allows a selection of any of the lines, from the set of repeated lines.
- ❑ Action associated with these Keys is 'Alter Object', which means that on hitting the Key, the Object associated with the current Line must be altered.
- ❑ Mode: Display specified in the Keys signifies that 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.

3.5 Actions - Create Collection, Display Collection and Alter Collection

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 displayed 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.

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.

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 required to achieve these actions.

Collection Attributes - Trigger, Variable and Report

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
```

Attribute - 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>
```

<Report Name> is the Interface used to display the Object names in a Collection.

Attribute - 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>
```

where,

<Report Name> is the final report displayed, when an Object is selected from the Collection.

Attribute - 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>
```

where,

<Variable Name> is the variable storing the Object name for the subsequent Report 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
```

4. Object Specific Actions

Some of the Object Specific Actions are discussed in this section:

4.1 Menu Actions – Menu Up, Menu Down, Menu Reject

Actions 'Menu Up', 'Menu Down', 'Menu Reject', etc., act upon Menu. They are associated to all Menus (Default as well as User Defined TDL) 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
```

[System: Menu Keys] declares a list of Keys commonly required for a Menu. Since all 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, as they are inherited from above declaration.

4.2 Form Actions - Form Accept, Form Reject, Form End

Actions 'Form Accept', 'Form Reject', 'Form End', etc., act upon Form. They are associated to all Forms (Default as well as User Defined TDL) 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
```

[System: Form Keys] declares a list of Keys commonly required for a Report. Since all 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, as they are inherited from above declaration.

4.3 Part Actions – Part Home, Part End, Part Pg Up

The Actions 'Part Home', 'Part End', 'Part Pg Up', etc., act upon a 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 DisplayHome]
```

```
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
```

[System: Form Keys] declares a list of Keys commonly required for a Part. Since all 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.

4.4 Line Actions - Explode, Display Object, Alter Object

Line Actions - Explode, Display Object, Alter Object, etc., act upon a 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
```

[System: Form Keys] declares a list of Keys commonly required for a Line. Since all 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, as they are inherited from above declaration.

4.5 Field Actions - Field Copy, Field Paste, Field Erase, Calculator

The Actions 'Field Copy', 'Field Paste', 'Field Erase', 'Calculator', etc., act on Fields.

- Action **Field Copy** copies the current field (Field where the cursor is positioned) contents in the OS clipboard, which will be available later.
- Action **Field Paste** pastes the clipboard contents to the current Field.
- Action **Field Erase** is used to erase the contents of the current Field at a stretch, without hitting the Backspace or Delete Key.
- Action **Calculator** is used for Fields that require some computation, the result of which is to be returned to the Field. Fields taking Amounts / Numbers as 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

[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. It has been associated to the 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 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 required in order to store the data are all pre-defined by the platform. All the data is stored in the Tally database. By default, data is always stored in pre-defined storages only.

There may be instances when additional information needs to be stored in the existing objects. This need gave rise to concept of User Defined Fields (UDF). A UDF can be used to store additional information to the Tally database. In other words, UDFs store additional information into the existing objects.

1. What is UDF?

User Defined Fields have a storage component defined by the user. UDFs are stored in the current object context. They 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 the object, i.e., it is always associated to the object to which the report is associated, i.e., the object in context.

1.1 Creating a UDF

UDFs should be defined under the section **[System: UDF]**.

Syntax

```
[System : UDF]
    <Name of UDF> : <Data Type> : <Index Number>
```

Where,

<Name of UDF> Identifies the UDF. Ideally, it should describe the purpose for which it has been created.

<Data Type> is any of the Tally data types or 'Aggregate'.

<Index Number> can be any number between 1 and 65536.

Numbers falling between 1 to 9999 and 20001 to 65536 are opened for customisation, and those between 10000 to 20000 are allotted for Common development in TSPL. The user can create 65536 UDFs of each data type.



The index numbers 1 to 29 are already used for Default TDL.

Example:

```
[System : UDF]

MyUDF 1 : String : 20003

MyUDF 2 : Date: 20003
```

The advantage of UDF in Tally is that it automatically attaches with the current object. No specific declaration is required for object association, when the UDF is defined within system definition.

1.2 Storing User Inputs in the UDF

The attribute **Storage** is used to store the value entered in the field, in the current object context.

Syntax

```
Storage : <Default Storage/Name of UDF>
```

Where,

<Name of UDF> identifies the UDF. Ideally, it should describe the purpose for which it is created.

Example:

```
[Field : NewField]

Use          : NameField

Storage     : MyUDF
```

1.3 Retrieving the value of UDF from an Object

In the context of the current object, the value of a UDF can be accessed by prefixing \$ to the UDF name.

Syntax

```
$(Name of UDF)
```

Example:

```
[Field: NewField]

Use          : NameField

Set As      : $MyUDF
```

2. Classification of UDF's

The UDFs are classified into two types, which are as follows:

- Simple UDF
- Complex/Compound/Aggregate UDF

2.1 Simple UDF

It can store one or more values of a single data type. A UDF used for storage, stores the values in the context of the object associated at Line/Report level, by default. Only one value is stored in this case.

UDF to store a single value

The following example code snippet demonstrates how a UDF can be made use of 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 'Company' Object in this case. The UDF 'Vehicle' stores a single string value.

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.

Syntax

```
Repeat : <Line name > : < Name of UDF >
```

Where,

<Line Name > is the name of the line to be repeated.

<Name of UDF> identifies the name of the UDF to store multiple values

The example in the section "UDF to store single value" can be modified to store multiple values.

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**.

Creating collection of Values Stored in UDF

Multiple values stored in a UDF can be displayed as Table in a field. The Collection is defined as:

Syntax

```
[Collection : <Collection Name>]
    Type          : <UDF Name> : <Object Name>
    Format        : $<UDF Name>, 20
```

Example:

```
[Collection : CMP Vehicles]

    Type          : Vehicle : Company
    Childof       : ##SVCcurrentCompany
    Format        : $Vehicle, 20
    Title         : "Company Vehicles"
```

It creates a collection of values stored in the UDF of the current object 'Company'. This collection can be used in the 'Table' attribute of 'Field' definition. 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 be stored as one entity, 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 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.

Creating an Aggregate UDF

To create an Aggregate UDF, the data type '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 them as one entity, a UDF of type 'Aggregate' is defined, as shown in the example.

Using an Aggregate UDF

An Aggregate UDF defined does not associate each component field with it. The association will take place only when a Line is repeated over an Aggregate UDF and within that Line, there are fields which store the value into the component UDFs.

Syntax

Repeat : <Line name> : <Name of Aggregate UDF>

Where,

<Name of Aggregate UDF> is the name of the 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.

Using Aggregate UDF in a Sub-Form

'**Subform**' is an attribute that is used within 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> is the name of the Report to be displayed.

<Condition> 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

EPrint1  : String   : 1002

BAggre   : Aggregate : 1000
```



Currently, UDF values can be retained only if the sub-objects are stored as is. However, in sub-objects where the accepted expense ledger information is apportioned against inventory, they are not stored independently but are attributed to the item cost once the object is accepted. For example, in purchase or manufacturing journal vouchers, the additional cost table is not retained independently within the object and is apportioned against the item cost while accepting the voucher. Hence, the UDF cannot be added and retained in these sub-objects.

Learning Outcome

- ❑ UDFs are stored in the context of the current Object. They 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 the 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 previous lesson, the significance and usage of User Defined Fields was explained. The classification and creation of UDFs 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, financial statements are generated as Reports, based on the vouchers entered till date. Balance Sheet, Profit & Loss A/c, Trial Balance, etc., are some default Reports in Tally.

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.
- Column-Based Reports: A Report with multiple columns. Tally.ERP 9 caters to generating all these types of Reports.

1.1 Tabular Reports

They have the simplest format of all the Report formats. A typical Tabular report has the 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.

A typical Tabular Report has a fixed number of columns and is interactive, i.e., an end user can change its appearance. Day Book, Stock Summary, Trial Balance, Group Summary, etc., are some default Tabular Reports in Tally. The Tabular Report 'Stock Summary' is shown in Fig. 9.1

Designing a Tabular Report

A typical Tabular Report will have a Title Line, a 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 lesson 5: Variables, Buttons, Keys)
- Adding explosions to the lines

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		600.00	
Defective Items			14,000.00	
Dot Matrix Printers			17,400.00	
Laser Jet Printers				
Timber	200 MT	6,364.83	12,72,965.75	
Grand Total			13,23,331.73	

Figure 9.1 Stock Summary

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 called as ‘explosion’ in Tally. Line attributes **Explode** and **Indent**, and the function **\$\$KeyExplode**, are used for the same.

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>

Where,

<Part Name> is the name of the Part which displays additional information about the Line object.

<Condition> if True, will result in an explosion.

Function - \$\$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 Key combination.

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 9.2 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

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]
```

Reports, Printing and Validation

```

Use      : Name Field

Set as   : $Name

Variable : MyGroupName1
    
```

My TB Report		ABC Company Ltd		Ctrl + M	
Name	Parent	Debit	Credit		
Capital Account	Primary				55,00,000.00
Balsubramanian'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 9.3 Simple Interactive Tabular Report

The code for the exploded part is as follows:

```

[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
    
```

The Collection **My TB GroupLedgers** is a union of collections of Type 'Group' and 'Ledger', respectively.

[Collection : My TB GroupsLedgers]

Collection : My TB Groups, My TB Ledgers

The Variable **MygroupName1** is used in **Child Of** attribute, 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		
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 9.4 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.

Trial Balance		ABC Company Ltd		Ctrl + M
Name	Parent	Debit	Credit	
Capital Account	Primary		55,00,000.00	
Current Assets	Primary	3,16,10,171.92		
Bank Accounts	Current Assets	37,98,633.77		
Cash-in-hand	Current Assets	1,83,52,572.24		
Loans & Advances (Asset)	Current Assets	16,08,900.00		
Stock-in-hand	Current Assets	13,23,331.73		
Sundry Debtors	Current Assets	62,76,734.18		
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	
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	
Direct Expenses	Primary	33,240.00		
Fixed Assets	Primary	34,62,489.68		
Indirect Expenses	Primary	31,70,616.61		
Investments	Primary	5,00,000.00		
Loans (Liability)	Primary		27,27,116.03	
Purchase Accounts	Primary	2,34,71,437.50		
Sales Accounts	Primary		4,75,92,847.50	
TOTAL		6,22,47,955.71	6,09,31,620.43	

Figure 9.5 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.

Designing Hierarchical Reports

Hierarchical Reports 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 '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:

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 9.6 Trial Balance Report

When the key **Enter** is pressed by the user, the next screen displays the Sub Groups of the current Group as shown in Figure 9.7

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			

Figure 9.7 Trial Balance - Sub group

1.3 Column Based Reports

The reports in which the number of columns added or deleted as per the user inputs are referred to as column-based reports. There are four types of column-based reports in Tally, namely Multi-Column Reports, Auto-Column Reports, Automatic Auto-Column Reports and Columnar Reports. All these types are explained with examples in this section.

Multi-Column Reports

In Multi-Column reports, a column is repeated based on the criteria specified by user. Trial Balance, Balance Sheet, Stock Summary, etc., are some default Reports in Tally.ERP 9 having Multi column feature. Normally, this feature is used to compare values across different periods.

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 user inputs like Period, Company Name, Stock Valuation, etc., based on which columns are generated.

Following attributes are used at various components of a Report to incorporate the multi-column feature:

Attribute - Column Report

Attribute 'Column Report' of the 'Report' definition, facilitates the creation of multi-column reports.

Syntax

```
ColumnReport : <Report Name>
```

Where,

<Report Name> is the name of the report used to obtain user inputs from the options displayed.

Attribute - Repeat

Attribute 'Column Report' is associated with a variable, which in turn is specified in 'Repeat' attribute of 'Report' definition. Both attributes are specified in 'Report' definition to create a multi-column report.

Syntax

```
Repeat : Variable
```

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' attributes at the Report, "New Column", "Alter Column" and "Delete Column" buttons will be automatically added to 'MulCol TrialBalance' Report.

```
[Report : MulCol Trial Balance]
```

```
ColumnReport : MyMultiColumns
```

```
Repeat : SVCURRENTCOMPANY, SVFROMDATE, SVTODATE
```

Particulars		ABC Company Ltd 1-Apr-2008 to 1-Aug-2008 Closing Balance	
		Debit	Credit
Capital Account			55,00,000.00
Current Assets		3,16,47,171.92	
Current Liabilities			51,11,856.90
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

Product	Version	Edition	Configuration	Calculator
Tally POWER OF SIMPLICITY Tally.ERP 9	Release 3 (Beta) Build 96 (Release) as on 21-Jan-2009 Serial Number 782981874	Latest Users Account ID tally9@tallysolutions.com Tally.Net subscription valid till 13-Feb-2009	Gold Unlimited 1 TDLS Loaded Server Port 9000 Running as ODBC Server	

Figure 9.8 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			51,11,656.90	
Direct Expenses		33,240.00		
Fixed Assets		34,37,489.68		
Indirect Expenses		31,58,616.61		
Investments		5,00,000.00		
Loans (Liability)			27,27,116.03	
Purchase Accounts	2,34,71,437.50			
Sales Accounts			4,75,92,847.50	
GRAND TOTAL		6,22,47,955.71	6,09,31,620.43	

Column Details

From Date : 1-Apr-2007

To Date : 30-Apr-2007

Figure 9.9 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 the values of system variable which is captured in MyMultiColumns Report

```
[Report : MulCol Trial Balance]
Repeat      : SVCURRENTCOMPANY, SVFROMDATE, SVTODATE
```

2. Part Definition: Repeating Lines over objects of a Collection.

```
[Part : MulCol TB Details]
Lines      : MulCol TB Details
BottomLines : MulCol TB Total
Repeat     : MulCol TB Details : MulCol TB GroupLed
```

3. Line Definition:- Repeating Field

[Line : MulCol TB Details]

Fields : MulCol TB Name Field, MulCol TB Amount Field

Repeat : MulCol TB Amount Field

ABC Company Ltd				
Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		ABC Company Ltd 1-Apr-2007 to 30-Apr-2007	
	Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		55,00,000.00
Current Assets	3,16,47,171.92		1,55,52,006.13	
Current Liabilities		51,11,656.90		34,61,144.04
Direct Expenses	33,240.00		2,770.00	
Fixed Assets	34,37,489.68		19,05,731.88	
Indirect Expenses	31,58,616.61		1,23,345.91	
Investments	5,00,000.00			
Loans (Liability)		27,27,116.03		16,82,650.00
Purchase Accounts	2,34,71,437.50		26,41,980.00	
Sales Accounts		4,75,92,847.50		73,11,525.00
GRAND TOTAL	6,22,47,955.71	6,09,31,620.43	2,02,25,833.92	1,79,55,319.04

Figure 9.10 Multi Column Report

Auto-Column Report

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.

Designing an Auto-Column Report

Auto column Report is similar to a Multi column Report, except that here, a set of columns are repeated, instead of one. User input will decide the criteria on which the 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		ABC Company Ltd 1-Apr-2007 to 1-Aug-2008 Closing Balance	
		Debit	Credit
			55,00,000.00
Capital Account			
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 9.11 Auto Column Reports

Step 2: The Configuration Report 'MultiAutoColumns'

1. In configuration Report, the user will be given 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,
MyHalfYearly

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 object

```
[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 the object MyCurrentCompany, which

;; is used to control the display of the objects 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"
```


Reports, Printing and Validation

BelongsIf : "Yes"
 IsAgeWise : No
 Periodicity : "Month"

[Object : MyYearly]

Name : \$\$LocaleString:"Yearly"
 VarName : "SVFromDate, SVToDate"
 CollName : "Period Collection"
 BelongsIf : "Yes"
 IsAgeWise : No
 Periodicity : "Year"



Columns can be repeated over any collection. They are 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		51,11,656.80		
Direct Expenses		33,240.00		
Fixed Assets	34,37,489.68			
Indirect Expenses	31,58,618.61			
Investments	5,00,000.00			
Loans (Liability)		27,27,116.03		
Purchase Accounts	2,34,71,437.50			
Sales Accounts			4,75,92,947.50	
GRAND TOTAL		6,22,47,955.71	6,09,31,620.43	

Auto Repeat Columns		Column Details
Repeat Using	:	<input type="radio"/> Half-Yearly <input type="radio"/> Monthly <input checked="" type="radio"/> Quarterly <input type="radio"/> Yearly

Figure 9.12 Auto Repeat Columns

2. When the user selects any one of the options, 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 My SelectAuto

;; and displays the corresponding method value of VarName

```
[Field : My CollName]
```

```
Use          : Short Name Field
```

```
Invisible   : Yes
```

```
Set as      : $$Table : MySelectAuto : $CollName
```

```
Modifies    : DSPRepeatCollection
```

```
Set always  : Yes
```

```
Skip        : Yes
```

;; We are modifying the value of the default variable DSPRepeatCollection by the value of the Method CollName from the selected Object DSPRepeatCollection is repeated in the Default Variables SVCURRENTCOMPANY, SVFROMDATE and SVTODATE, which gets new values for each column

```
[Field : My StartPeriod]
```

```
Use          : Short Date Field
```

```
Invisible   : Yes
```

```
Set as      : if $$IsEmpty:$$Table:MySelectAuto:$Periodicity then+
              ##SVFromDate else if $$Table : MySelectAuto : +
              $Periodicity = "Day" then ##SVFromDate else +
              $$LowValue : SVFromDate
```

```
Set always      : Yes
```

```
Modifies       : SVFromDate Skip : Yes
```

;; Value of Variable SVFromDate is set here based on the Periodicity Method.

;; \$\$LowValue is a Function that returns the 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 the Variable SVToDate is set here based on the Periodicity Method.

;; MonthEnd is a Function that 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

Particulars	ABC Company Ltd 1-Apr-2007 to 30-Jun-2007		ABC Company Ltd 1-Jul-2007 to 30-Sep-2007		ABC Company Ltd 1-Oct-2007 to 31-Dec-2007	
	Closing Balance		Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		55,00,000.00		55,00,000.00
Current Assets	2,54,46,664.95		2,90,59,229.83		2,84,31,555.33	
Current Liabilities		55,13,819.50		74,10,593.12		47,51,275.11
Direct Expenses	8,310.00		8,310.00		8,310.00	
Fixed Assets	25,21,191.88		40,46,191.88		40,67,691.88	
Indirect Expenses	4,48,293.52		4,56,738.62		8,07,075.84	
Investments			5,00,000.00		5,00,000.00	
Loans (Liability)		23,17,277.46		22,27,143.65		26,18,979.84
Purchase Accounts	78,63,750.00		44,00,552.50		62,02,500.00	
Sales Accounts		1,89,11,775.00		94,51,825.00		84,38,121.00
GRAND TOTAL	3,62,88,210.35	3,22,42,871.96	3,84,71,022.83	2,45,89,561.77	4,00,17,133.05	2,13,08,375.95

Figure 9.13 Auto Column Report

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 autocolumn Report in Tally.ERP 9.

Designing an Automatic Auto Column Report

In order to design an Automatic Auto Column 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** has to be set to "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:

Particulars	ABC Company Ltd 1-Apr-2007 to 1-Aug-2008		Global Enterprises For 1-Apr-2007	
	Closing Balance		Closing Balance	
	Debit	Credit	Debit	Credit
Capital Account		55,00,000.00		
Current Assets	3,16,47,171.92		6,790.00	
Current Liabilities		51,11,856.90	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 9.14 Displaying Trial Balance for two different companies

Columnar Report

All the Voucher Reports which contain Accounting Information (Ledger and/or Group Info) available in Vouchers, 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

We have already understood the various types of reports and the techniques to generate them. An essential element of Reporting is printing. All the reports must be printable in one form or the other.

Printing Techniques: The techniques used for Printing are as follows:

2.1 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>:
      <ActionKeyword> : <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
```

Here, we are adding 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.

2.2 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 the list of employees is not required. A new report printed for various pay slips allotted to the selected employees is needed.

```
[Button : Print Selected Pay slips]

;; Associate this button to the current report displaying the 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 the inbuilt Collection 'Parameter
;; Collection' to print the user selection for the list of employees

  Action : Print Report : Multi Pay Slip Print
```

Scope : Selected Lines

[#Report : Multi Pay Slip Print]

Collection : Parameter Collection

Here, the Button 'Print Selected Pay slips' is defined with the 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 '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'.

2.3 Page Breaks

A Page Break is the point at which one page ends and another begins. Handling Page Breaks is important, as the current page should indicate continuation to the next page, while the next page must indicate that the current page is continued from the previous page. So, there must 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 are printed across multiple pages. Closing Page Break starts printing from the first page and prints on every page except the last page, e.g., **Continued...** to be printed at the bottom of each page. 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 is encountered first.

In TDL, Page Breaks can be handled **vertically** as well as **horizontally**.

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 the last page, the text Continued.... may be required at the end of each page, and the 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 this 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 the beginning of all the continuing pages. Since more than one part is used within Part definition, specify the alignment as '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. 1st Line for Closing Page Break and 2nd Line for 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"

```

Here, 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, the Line 'Op Page Break' is defined to use the above defined line 'Cl Page Break', which locally modifies only the field 'Particulars Fld'.

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 Line Attribute 'Page Break'. It is generally used to repeat a closing column at every closing page and opening column at every opening page. It takes **2 parameters**, viz. 1st Field for Closing Page Break & 2nd 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 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 this example, the Field **CI 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
It is a Vertical Page Break	It is a Vertical Page Break	It 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 breaks	Multiple Fields (Fields within Fields) can be printed at both closing and opening page breaks
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 page breaks

2.4 Frequently Used Attributes and Functions

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)
```

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
```

Functions**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 + ")"
```

Function - \$\$IsLastOfSet

It is used to check if the current Form is the last Form being printed. It doesn't require any parameter. It returns 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)
```

Function - \$\$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

It 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 '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)
```

Here, if the Balance number of lines is between 0-5, remaining lines will be printed on next page.

3. 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.

3.1 Field Level Attribute - Validate

This attribute checks if the given condition is satisfied. Unless the given condition for 'Validate' is satisfied, the user cannot move further, i.e., 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 example:

- ❑ The field CMP Name is a field in Default TDL which is used to create/ alter a Company.
- ❑ Attribute '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.

3.2 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.

3.3 Field Level Attribute — Notify

It is similar to attribute 'Validate'. The only difference is that it flashes a warning message and the cursor moves to the subsequent field. A system formula is added to display the warning message.

Syntax

Notify : <System Formula> : <Logical Condition>

Example:

```
[!Field : VCH Nrm1BilledQty]

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
```

Here, **VCH Nrm1BilledQty** is a default optional field in TDL used in a Voucher. 'Notify' pops up 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.

3.4 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 example, the field, **VCH Number** is a default field in TDL, used in a Voucher. The duplication of voucher numbers for a particular voucher type is prevented by using the attribute **Control**. The differences 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

3.5 Form Level Attribute - Control

This attribute achieves a higher level of control on the contents of a Form, compared to the 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 : ##SVCcurrentCompany

Control : DateBelowFromDate : $Date < $$SystemPeriodFrom

Control : DateBeyondToDate : $Date > $$SystemPeriodTo
```

In this 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.

3.6 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
```

Here, the menu item **Accounts Info** will be displayed only if the condition is satisfied. **\$\$Allow** checks if the current user has the rights to access the report displayed under the current Menu item. The value **AccountsMasters** has been derived from attribute 'Family' at 'Report' definition.

3.7 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"
```

Here, **Accounts Masters** will get added to the Security list. Without the user rights for 'Accounts Masters' in the Security controls, this report can neither be created, altered nor viewed.

Learning Outcome

- Tally.ERP 9 caters to 3 different types of Reports. These are:
- Tabular Reports: Reports with fixed number columns, which can be configured
- Hierarchical Reports: Reports in successive levels or layers
- Column based Reports: Reports with multiple columns
- 'Explode' is a Line attribute, 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.
- Multi column report is one where a column repeats, based on 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 2 levels, viz. Platform and TDL levels.

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 of 'Integrate Accounts and Inventory' is set to YES in F11:Accounting/Inventory Features.

1.3 Accounting-cum-Inventory Vouchers

Accounting-cum-Inventory Vouchers are transactions containing details pertaining to Accounts as well as Inventory. Purchase Order, Receipt Note, Rejection In, Debit Note, Purchase, Sales Order, Delivery Note, Rejection Out, Credit Note, Sales, etc. 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 stores 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:

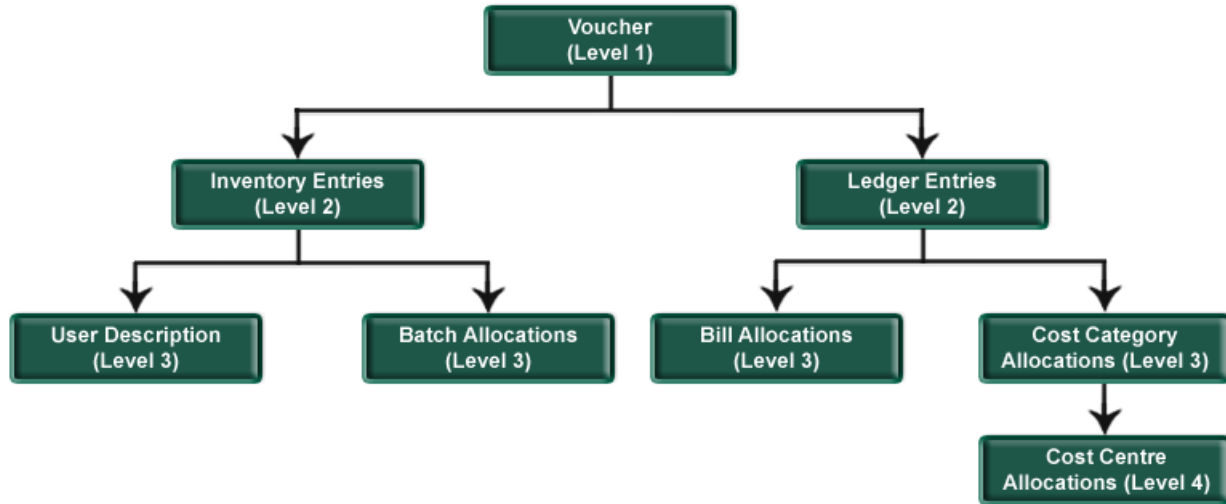


Figure 10.1 Hierarchy of Voucher objects

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.

In cases 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's consider the following examples 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 as follows:



Figure 10.2 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** are 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 follows:

ABC Company Ltd
5, 9th Cross
Margosa Road
Malleswaram

Receipt Voucher

No. : 211 Dated : 31-Mar-2008

Particulars	Amount
Account :	
Modern Advertisers Agst Ref 111 7,303.40 Cr Output ST - Advt. Services	7,303.40
 Cheque Number and Date : 11384 dt 31-Mar-2008	
Through : HDFC Bank	
On Account of : Full Amount is being received	
Amount (in words) : Rs. Seven Thousand Three Hundred Three and Forty paise Only	
	7,303.40

Authorised Signatory

Figure 10.3 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 10.4 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 listed below:

- Firstly, 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 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		
Party's A/c Name : Fuzitsy Systems		Cost Centre/Classes : Not Applicable		Price Level : Not Applicable		
Current Balance : 14,949.80 Dr		Sales Ledger : Sales - Exports		VAT/Tax Class: Exports		
Name of Item	Marks	No. of Packages	Quantity	Rate	per Disc. %	Amount
Wireless Keyboard	5463	18	5 Nos	\$ 75.00	Nos	\$ 375.00
Show Statutory Details ? No		Narration:		5463 18 5 Nos \$ 375.00 @ Rs. 43.28/\$ = Rs. 16,230.00		
Being Computer Accessories sold to Russia.						
Q: Quit	A: Accept	D: Delete	X: Cancel			

Figure 10.5 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 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.

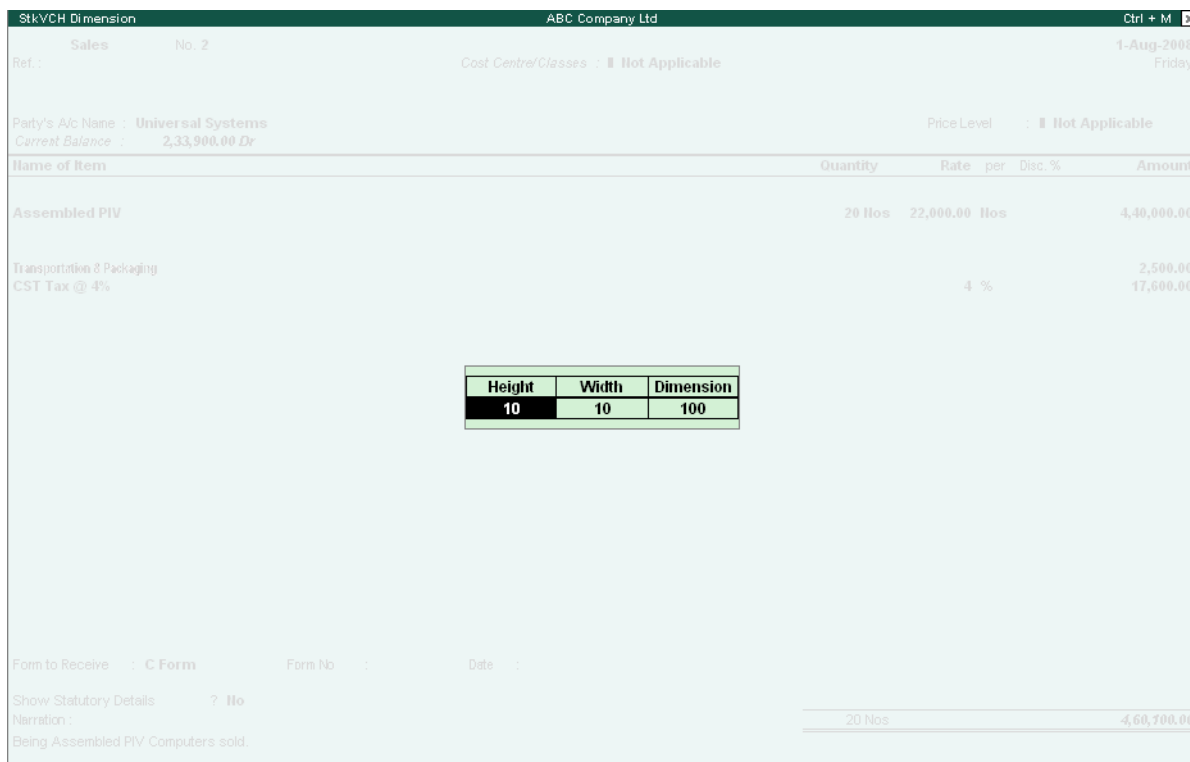
Care should be taken to consider all situations, while addressing similar requirements, such as with or without activation of Actual and Billed Quantity, with or without Batch wise screen, etc.

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
					4,60,700.00

Height	Width	Dimension
10	10	100

Figure 10.6 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]
```

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

Figure 10.7 Stock Item Allocation Screen

3.2 Invoice Customisation

Invoice Customisation can broadly be classified into two categories based on the requirement:

- ❑ Invoice Customisation – User defined format
- ❑ Invoice Customisation – Modifications to default format

Invoice Customisation – 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 the default format
- ❑ Replacing the existing format with a new one

Adding a new format along with the 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 in turn 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 :
SI. No.	Item Name	Quantity	Rate	Amount
1	Assembled PIV	20 Nos	220,000.00	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 10.8 Invoice Customisation - Comprehensive

Replacing the existing format with a 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 in turn requires the following format for both a Normal Invoice as well as a Simple Invoice.

ABC Company Ltd

Voucher Date : 1-3-2008 Voucher No : 2
 Party Name : Universal Systems

Sr No	Name	Billed Qty	Rate	Amount
1	Assembled PIV	20	22,000.00	440000.00
	Transportation & Packaging			2500.00
	CST Tax @ 4%		4%	17600.00
Totals		20		460100.00

Amount in words : Rs. Four Lakh Sixty Thousand One Hundred Only

For ABC Company Ltd

Authorised Signatory

Figure 10.9 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
```

Invoice Customisation – 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	

Sl No.	Marks & Nos./ Container No.	No. & Kind of Pkgs.	Description of Goods	Quantity	Rate	per	Dist. %	Amount
1			Assembled PIV	20 Nos	22,000.00	Nos		4,40,000.00
			<i>Transportation & Packaging</i>					2,500.00
			<i>CST Tax @ 4%</i>			4 %		17,600.00
Total				20 Nos				4,60,100.00

Amount Chargeable (in words) E. & O.E
Rs. Four Lakh Sixty Thousand One Hundred Only

Terms & Conditions :

- Training : 9 hours
- Tally Support : 3 months without any additional charges

Company's VAT TIN : **23424872389**
 Company's CST No. : **234234234**
 Company's Service Tax No. : **234234**
 Company's PAN : **EENMM16789**

Declaration
 We declare that this invoice shows the actual price of the goods described and that all particulars are true and correct.

for ABC Company Ltd
 Authorised Signatory

This is a Computer Generated Invoice

Figure 10.10 Invoice Customisation

Solution
Step 1

The default configuration Part **IPCFG Right** is altered to add the Line.

```
[#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 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’ is 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

Writing Remote Compliant TDL Reports

Introduction

Enabling access to your organisational data 'anytime, anywhere', and yet being truly usable, is what Tally.ERP 9 is capable of. With 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 this 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.

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 Levels for delivering this capability. The way TDL Reports have been changed in default TDL to optimise the performance and seamlessly work without clogging the network is the focus of this chapter. The idea is to reduce the server calls for accessing the data. The same concepts can be followed for making the customized Reports Remote Compliant.

Given below is the overall enabled environment using Tally.NET Server.

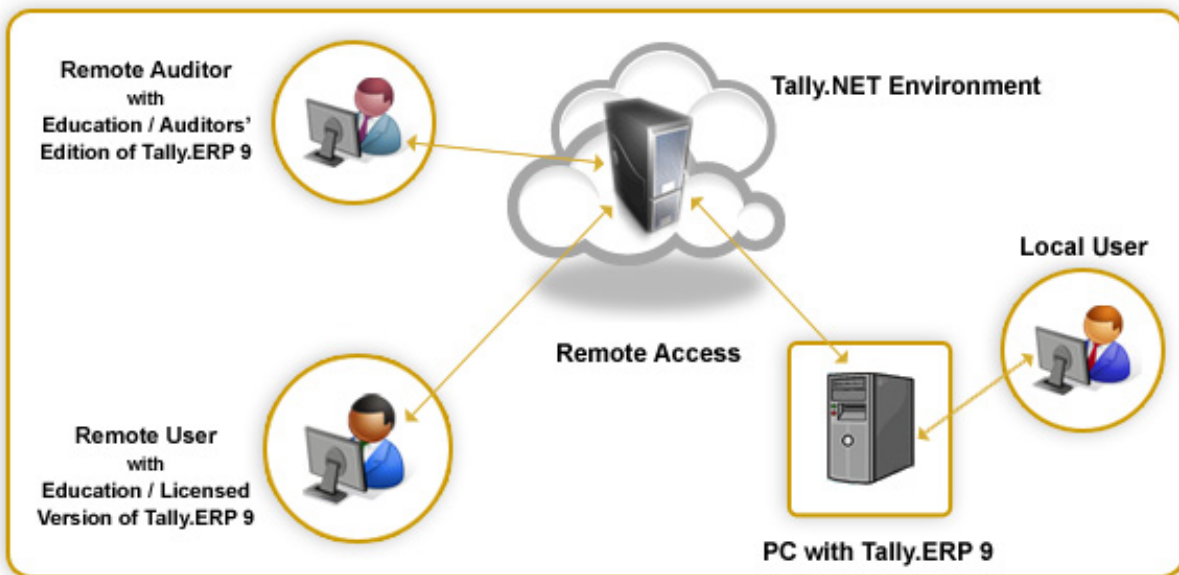


Figure 12.1 Overview of Tally.NET Server

We will begin our discussion with an overview of client/server environment in general, and then move on to Tally Client/Server, and the role of Tally.NET Server in such a scenario. The topics covered henceforth will focus on understanding the execution of TDL reports and optimising 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.

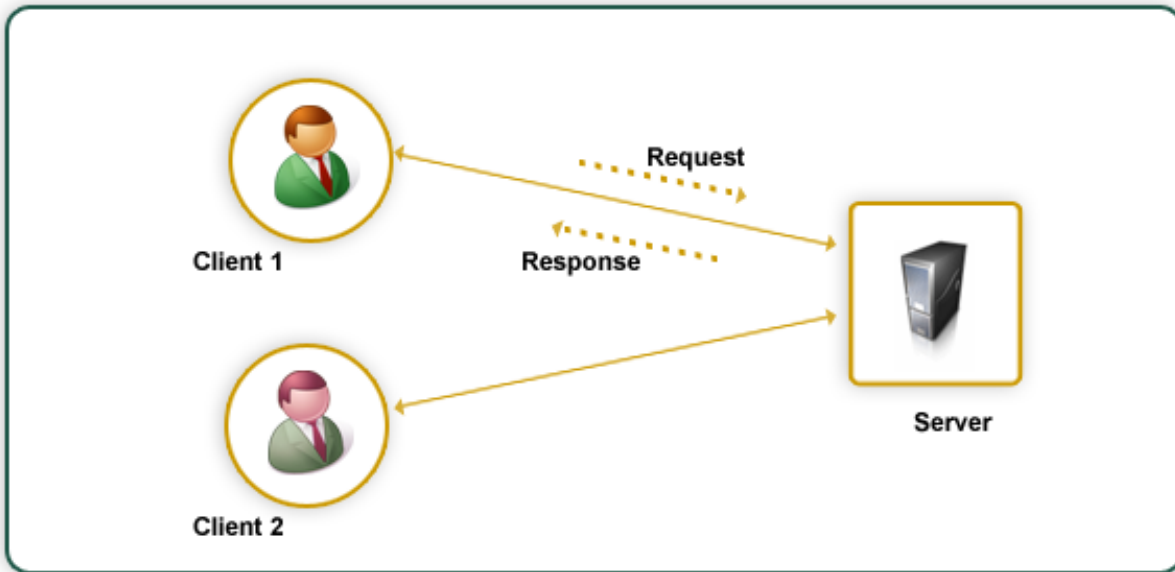


Figure 12.2 Block diagram of client/server architecture

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 Software Services

TSS (Tally Software Services) is a framework which provides a number of services to the Tally.ERP 9 users. The TSS architecture is derived from client/server architecture. In this architecture, Tally.ERP 9 Client is connected to Tally.ERP 9 server via middle-ware, i.e., Tally.NET Server. Following are the major components of the TSS architecture.

- Tally.NET Server
- Tally.ERP 9 Server
- Tally.ERP 9 Client

2.1 Tally.NET Server

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 TSS features. The user can utilize Tally.NET Server to Synchronize data, access online help and support and

manage licenses across locations, while 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 Server. Tally.NET Server takes care of user authentication when a remote user tries to connect to the Tally.ERP 9 Server.

TSS Features

- ❑ Register and Connect companies from Tally.ERP 9
- ❑ Create and maintain Remote Users
- ❑ Remote availability of Auditor's License
- ❑ Synchronize data
- ❑ Remote access of data by any user
- ❑ Use online help and support capability from within Tally or browser
- ❑ Application Management (across Multi-serial, Multi-Location, etc.) via Tally or browser

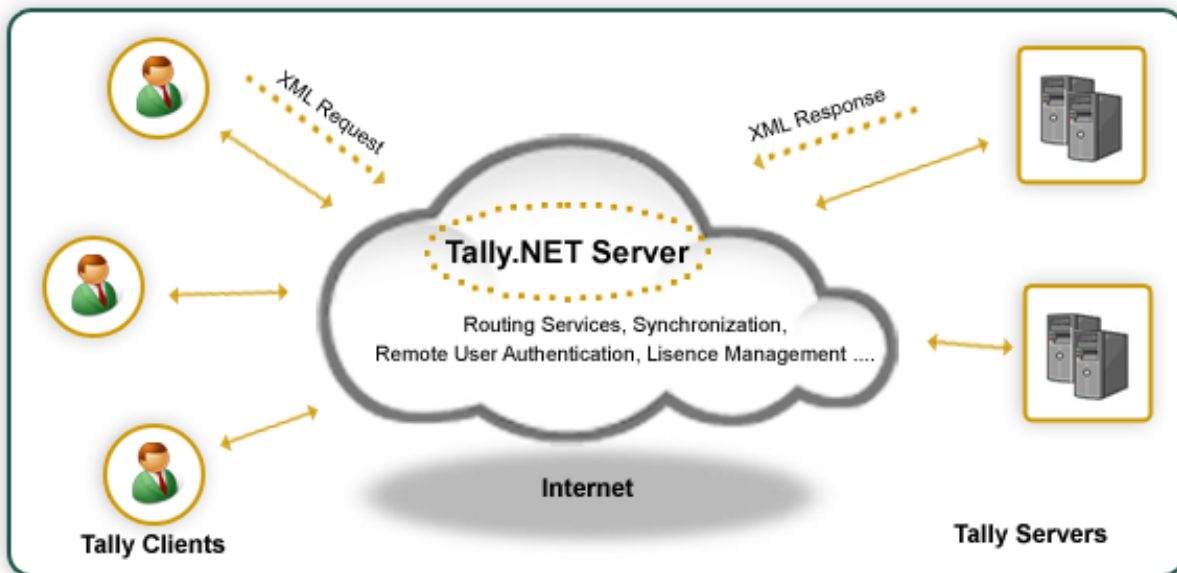


Figure 12.3 Tally.NET Server Architecture

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, etc., 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 Tally.NET Server for Remote Access

Following are the steps which need to be executed to setup the Tally.NET Server:

Step 1: Enable Security control to avail TSS features.

Go to **Gateway of Tally**. Click **F3 :Company Info > Alter**

Step 2: Configuring TSS features.

Go to **Gateway of Tally**. Click **F11:Features > TSS Features**

Step 3: Authorizing the Remote Users.

Go to **Gateway of Tally**. Click **F3 :Company Info > Security Control > Users & Passwords**



- *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 the client wants his Tally.NET Auditor/Tally.NET User to access data remotely.*
- *If 'Allow Local TDLs' is set to YES, then the client can load Local TDLs, in addition to remote TDLs. If it is set to NO, the client cannot load Local TDLs.*

Step 4: Connecting Companies to Tally.NET Server

Go to **Gateway of Tally**. Click **F4:Connect Company**

4. Setting up the Client Tally

The users classified as 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 Tally.NET Server

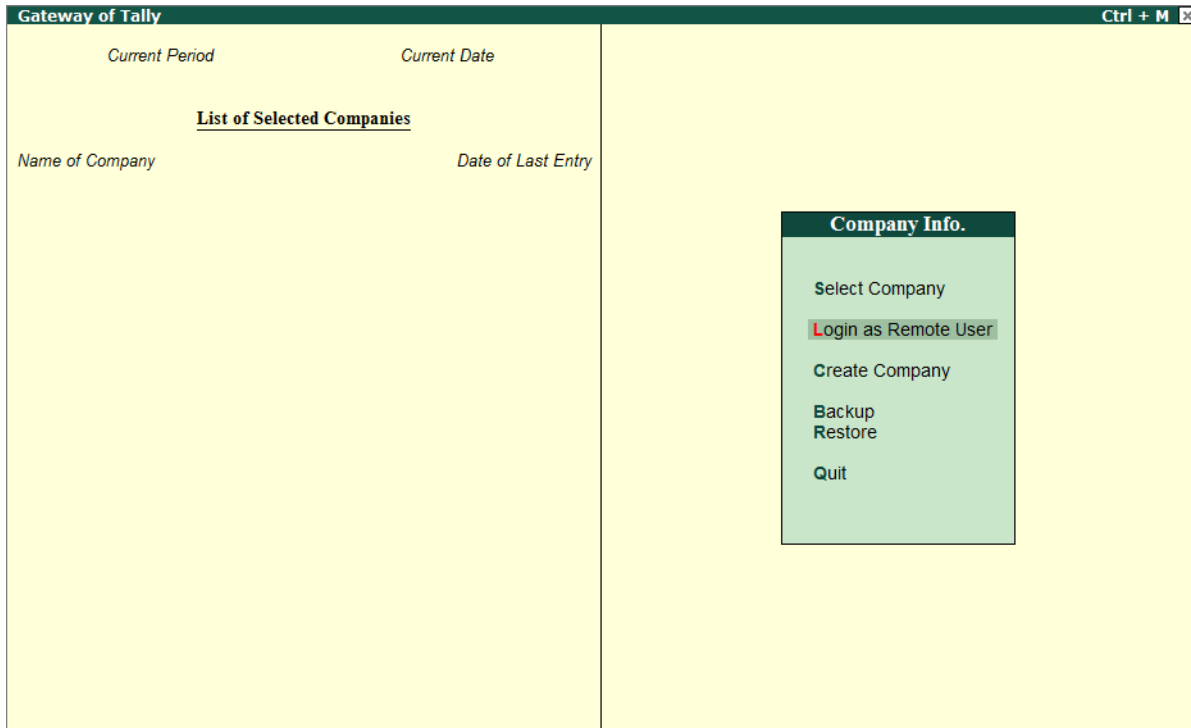


Figure 12.4 Connect to Tally.NET Server

Step 2:

Provide the User name and Password



Figure 12.5 Providing User Name and Password

After entering of valid user name & password, Tally displays 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
<u>Online Companies</u>				
ABC Company Ltd	tally1@tallysolutions.com	702088209	Rakesh	
ABC Exicse Company II	tally1@tallysolutions.com	702088218	Jai	
BOM	tally1@tallysolutions.com	702088218		
Haryana	tally1@tallysolutions.com	702088218		
Jonny Connect	tally1@tallysolutions.com	702088209	Arun	3318
National Traders	tally1@tallysolutions.com	702088218		
National Traders (TN Regular)Pay	tally1@tallysolutions.com	702088218		
Payroll	tally1@tallysolutions.com	702088218		
Profit and Loss - Kumaran	tally1@tallysolutions.com	702088209	Kumaran	
Service Tax - ERP	tally1@tallysolutions.com	702088218		
Tally Audit Demo and Co Krishna	tally1@tallysolutions.com	702088218	Mohan	
Tally ERP 9 Testing	tally1@tallysolutions.com	702088218	Tally World	
Tally Payroll 25th Feb	tally1@tallysolutions.com	702088218		
Test Remote	tally1@tallysolutions.com	702088218	TR	
Utkarsh Test Data	tally1@tallysolutions.com	702088218	Utkarsh	3275
<u>Offline Companies</u>				
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 12.6 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 a client/server environment, data resides in the server. A typical client will have only user interface. Whenever the client requires data, it has to send a request to the server with credentials, and the server will respond with the data.

In TSS environment, the server and the client exchange the request/response in encrypted XML format. When the client is Tally application, it will have only the user interface and needs to get data from the server on demand. A typical Tally application is developed using TDL. In TDL language, definitions are broadly classified as Data Objects and Interface Objects. Interface objects define the user interface and Data objects store the values in Tally primary or secondary database. Tally client will have only Interface Objects locally and the Data Objects need to be fetched from the server on request.

It is the TDL Programmer’s responsibility to fetch the required data from the Tally server to Tally Client.

6. TDL Enhancements for Remote

TDL language has been 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 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 a 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' attribute, methods doing aggregation or computation using 'Aggr Compute' & 'Compute' are also brought to the Tally Client.

Internally fetching a method will generate an XML fragment, which will be sent to the 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 the Client.

2. Compute

Syntax

`Compute : Method-Name : Method-Formula`

Where,

<Method-Formula> is any computational method, and

<Method-Name> denotes the name of the method.



Please refer 'TDL Enhancements for Tally.ERP 9.pdf' for further information on Collection attributes 'Aggr Compute', 'Compute' and 'Fetch'

Example: Fetching Name & Closing Balance of Ledger Object

Step 1:- Fetching **Name** and **Closing Balance** methods of 'Ledger' object

`[Collection : Ledgers]`

`Type : Ledger`

`Fetch : Name, Closing Balance, Parent`

`Compute : PClosingBalance : $ClosingBalance : Group : $Parent`

```
Format : $Name, 15
```

Step 2: Utilizing the fetched methods

a) As a Table

```
[Field : Sample Field]
```

```
Table      : Ledgers
```

```
Show Table : Always
```

b) In 'Repeat' at Part Level

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

```
Line      : Sample Line
```

```
Repeat   : Sample Line: Ledgers
```

```
[Line : Sample Line]
```

```
Fields  : Sample Fld1, Sample Fld2, Sample Fld3
```

```
[Field : Sample Fld1]
```

```
Use     : Name Field
```

```
Set as  : $Name
```

```
[Field : Sample Fld2]
```

```
Use     : Amount Field
```

```
Set as  : $ClosingBalance
```

```
[Field : Sample Fld3]
```

```
Use     : Amount Field
```

```
Set as  : $PClosingBalance
```

Sample Request Format XML file to fetch the internal methods and Compute method:

```
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <TALLYREQUEST>EXPORT</TALLYREQUEST>
    <TYPE>COLLECTION</TYPE>
    <ID>Ledger</ID>
  </HEADER>
```

```
<BODY>
  <DESC>
    <STATICVARIABLES>
      <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
      <SVCURRENTCOMPANYTYPE="String">DemoCompany</SVCURRENTCOMPANY>
      <SVCURRENTDATE TYPE="Date">20-Dec-2008</SVCURRENTDATE>
      <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>
      <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>
      <SVCURRENTKBLANGUAGEID TYPE="Number">1033
    </SVCURRENTKBLANGUAGEID>
  </STATICVARIABLES>
  <TDL>
    <TDLMESSAGE>
      <COLLECTION NAME="Ledger" ISMODIFY="No" ISFIXED="No"
        ISINITIALIZE="Yes" ISOPTION="No" ISINTERNAL="No">
        <TYPE>Ledger</TYPE>
        <METHOD>PClosingBalance:$ClosingBalance:Group:$Paren</METHOD>
        <NATIVEMETHOD>Name</NATIVEMETHOD>
        <NATIVEMETHOD>Parent</NATIVEMETHOD>
        <NATIVEMETHOD>ClosingBalance</NATIVEMETHOD>
      </COLLECTION>
    </TDLMESSAGE>
  </TDL>
</DESC>
</BODY>
<ENVELOPE>
```


6.2 Report Level Enhancements

Fetching the Object

When multiple methods of a single Object are required for a Report, then that Object can be fetched at Report level. For this purpose, new Report attribute '**Fetch Object**' has been introduced. Internally fetching an object will generate an XML fragment which will be sent to the 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.

<Method Name 1> denotes 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 this code snippet, **Ledger Name** is the variable which stores the name of the Ledger Object whose methods need to be fetched at the Report.

Sample Request Format XML file to fetch the object:

```
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <TALLYREQUEST>EXPORT</TALLYREQUEST>
    <TYPE>OBJECT</TYPE>
    <SUBTYPE>Ledger</SUBTYPE>
    <ID TYPE="Name">Cash</ID>
  </HEADER>
  <BODY>
    <DESC>
      <STATICVARIABLES>
        <SVCURRENTCOMPANY>Demo Company</SVCURRENTCOMPANY>
        <SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>
```

```

    <SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>

    <SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>

    <SVCURRENTDATE TYPE="Date">1-May-2008</SVCURRENTDATE>

    <SVVALUATIONMETHOD TYPE="String"></SVVALUATIONMETHOD>

    <SVBUDGET TYPE="String"> </SVBUDGET>

    <SVCURRENTKBLANGUAGEIDTYPE="Number">1033

    </SVCURRENTKBLANGUAGEID>

    <SVCURRENTUILANGUAGEIDTYPE="Number">1033

    </SVCURRENTUILANGUAGEID>

</STATICVARIABLES>

<FETCHLIST>

    <FETCH>Name</FETCH>

    <FETCH>Parent</FETCH>

    <FETCH>Closing Balance</FETCH>

</FETCHLIST>

</DESC>

</BODY>

</ENVELOPE>

```

Pre Fetching the Object

There are some scenarios in which it is required to set the values 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**' has been 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> denotes 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 this code snippet, variables are set once, and then the PreFetchObject is done, and once again the variables are set to make sure that the values of the variables which were dependent on the object, will set now.

Pre fetching the Collection

When the same collection is used in the Report either for repeating the line over its objects or multiple functions using the same, then a Collection of those objects can be pre fetched at the Report level. A new Report attribute '**Fetch Collection**' is introduced to pre fetch a Collection.

Syntax

```
Fetch Collection : <Collection 1> [,<Collection 2>..]
```

Where,

<Collection 1> is the collection whose objects need to be pre fetched at Report.

Example: Pre fetching Ledger collection

```
[Report : Sample Report]

Fetch Collection : Ledger

Local            : Collection : Fetch : Ledger
```

In this code snippet, Ledger Collection is pre-fetched.

Sample Request Format XML file to fetch the object:

```
<ENVELOPE>

  <HEADER>

    <VERSION>1</VERSION>

    <TALLYREQUEST>EXPORT</TALLYREQUEST>

    <TYPE>COLLECTION</TYPE>

    <ID>All Party</ID>

  </HEADER>

  <BODY>

    <DESC>

      <STATICVARIABLES>
```

```
<SVEXPORTFORMAT>BinaryXML</SVEXPORTFORMAT>

<SVUSEPARMLIST>No</SVUSEPARMLIST>

<SVFORTABLE>No</SVFORTABLE>

<SVCURRENTCOMPANY TYPE="String">Remote Vivek</SVCURRENTCOMPANY>

<SVCURRENTDATE TYPE="Date">2-May-2008</SVCURRENTDATE>

<SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>

<SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>

<SVVALUATIONMETHOD TYPE="String"></SVVALUATIONMETHOD>

<SVBUDGET TYPE="String"></SVBUDGET>

</STATICVARIABLES>

<TDL>

  <TDLMESSAGE>

    <COLLECTION NAME="All Party" ISMODIFY="No" ISFIXED="No"
      ISINITIALIZE="Yes" ISOPTION="No" ISINTERNAL="No">

      <TYPE>Ledger</TYPE>

      <BELONGSTO>Yes</BELONGSTO>

      <CHILDOF>$$GroupSundryDebtors</CHILDOF>

      <NATIVEMETHOD>OpeningBalance</NATIVEMETHOD>

      <NATIVEMETHOD>ClosingBalance</NATIVEMETHOD>

    </COLLECTION>

  </TDLMESSAGE>

</TDL>

</DESC>

</BODY>

</ENVELOPE>
```

6.3 Function on Request

Functions in TDL are defined and provided by the platform. A TDL programmer can only call a function. Now in client/server environment, functions can be evaluated by either the sever or the client or both the client and the server.

Based on this information, functions can be classified as follows:

1. Evaluated at client side
2. Evaluated at server side
3. Hybrid

Evaluated at client side

These are the functions which will be evaluated at the client side. For this, no server request is required from the client. If these functions require any parameter as data, then required data needs to be fetched from the server before the function is called.

Example:

\$\$KeyExplode, \$\$ExplodeLevel, \$\$Line, etc., are the functions which do not require any parameter from the Tally server and are executed at the Tally client.

Evaluated at server side

These are the functions which will be evaluated at the server side. For each call of a function, a request will be sent to the 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>
      </STATICVARIABLES>
    </DESC>
  </BODY>
</ENVELOPE>
```

```

<SVFROMDATE TYPE="Date">1-Apr-2008</SVFROMDATE>

<SVTODATE TYPE="Date">31-Mar-2009</SVTODATE>

<SVCURRENTDATE TYPE="Date">1-May-2008</SVCURRENTDATE>

<SVCURRENTKBLANGUAGEID TYPE="Number">1033</SVCURRENTKBLANGUAGEID>

<SVCURRENTUILANGUAGEID TYPE="Number">1033</SVCURRENTUILANGUAGEID>

</STATICVARIABLES>

</DESC>

</BODY>

</ENVELOPE>

```

Hybrid

These are the functions which will be executed on either the client or the 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 the sever, 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, the function will be executed on the client side, since it is assumed to be executed in the 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 the usage of this action, please refer to '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>
      </LEDGER NAME="Customer 1" RESERVEDNAME="">
    </TALLYMESSAGE>
  </BODY>
</ENVELOPE>

```

```
</MAILINGNAME.LIST>
<ALTEREDON>20090112</ALTEREDON>
<NAME TYPE="String">Customer 1</NAME>
<CURRENCYNAME>Rs.</CURRENCYNAME>
<PINCODE>560001</PINCODE>
<PARENT>Sundry Creditors</PARENT>
<ISDEEMEDPOSITIVE TYPE="Logical">Yes</ISDEEMEDPOSITIVE>
<SORTPOSITION> 1000</SORTPOSITION>
<OPENINGBALANCE>1.00</OPENINGBALANCE>
<LANGUAGENAME.LIST>
<NAME.LIST TYPE="String">
<NAME>Customer 1</NAME>
<NAME>Alias</NAME>
</NAME.LIST>
<LANGUAGEID> 1033</LANGUAGEID>
</LANGUAGENAME.LIST>
</LEDGER>
</TALLYMESSAGE>
</BODY>
</ENVELOPE>
```

7. Writing Remote Compliant TDL Reports

TDL programmer can optimize the performance of the Remote compliant TDL by minimizing the server request calls.

Mentioned below 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 the Object can be pre-fetched with the required methods. In this approach, only one server call is made to fetch 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

The following techniques are used to optimize the performance, when a line is repeated over a collection in a report to be displayed on the client.

Fetching the Methods

Whenever a collection is 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 the methods which are used in the fields. If 'Fetch' is not used, then the data will not be displayed in the field.

```
[Part : LedReport]

  Line   : LedReportDetails

  Repeat : LedReportDetails : Ledger

  Scroll : Vertical

[Line : LedReportDetails]

  Fields       : Led Name

  Right Field  : LedClosingBalance

  [Field : Led Name]

    Use       : Name Field

    Set as   : $Name

  [Field : LedClosingBalance]

    Use      : Amount Forex Field

    Set as  : $ClosingBalance

[#Collection : Ledger]

  Fetch : Name, Closing Balance
```

Function inside the 'Repeat'

When Lines are repeated over a Collection and a function is used at the field level, then each 'Repeat' will trigger an additional server request for function call. In this scenario, 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   : $Names

  [Field : LedClosingBalance]

    Use      : Amount Forex Field

    Set as   : $ClosingBalance

  [Field : LedSalesTotal]

    Use      : Amount Forex Field

    Set as   : $LedgerSalesTotal

[#Collection : Ledger]

  Fetch    : Name, Closing Balance

  Compute  : LedgerSalesTotal : +

            $$AsReqObj : $$FilterAmtTotal : LedVouchers : MyParty : $Amount
```

Repeating over Period Collection

In Reports where lines are repeated over Period Collection and values of each column is calculated over a period for the required object, e.g., 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 the Period Collection to fetch the values for each column.

```
[#Collection : Period Collection]

  Compute : TBalDebits : $TBalDebits : VoucherType : #VoucherTypeName

  Compute : TBalCredits : $TBalCredits : VoucherType : #VoucherTypeName

  Compute : TBalClosing : $TBalClosing : VoucherType : #VoucherTypeName
```

7.3 Using the same Collection in more than one Report

When more than one Report requires different methods of the Objects of the same Collection, then using the same collection with all the methods fetched in it reduces the performance. This can be improved in the following ways:

Fetching the required methods locally at Report

In the following code snippet, **Sample Report1** requires Opening Balance of a Ledger whereas **Sample Report2** requires Closing Balance. Instead of modifying the Collection to fetch both Opening Balance and Closing Balance, the same is localized in respective Reports.

```
[Report : Sample Report1]

    Local : Collection : Ledger : Fetch : Opening Balance

[Report : Sample Report2]

    Local : Collection : Ledger : Fetch : Closing Balance
```

Separate Collections for fetching different methods

In the following code snippet, two Collections are created for fetching the opening balance and the closing balance. Later, the first Collection can be utilized in **Sample Report1** and the second one in **Sample Report2**.

```
[Collection : Fetch Opening Balance]

    Type : Ledger

    Fetch : Opening Balance

[Collection : Fetch Closing Balance]

    Type : Ledger

    Fetch : Closing Balance
```

General and Collection Enhancements

Introduction

In Tally.ERP 9, major changes have been provided by the platform to enhance the TDL capabilities to help the programmer to develop and deploy solutions with ease. Major improvements have taken place in terms of language usage standardisation and performance improvements.

There have been breakthrough enhancements at Collection level to provide Remoting and Advanced Reporting capabilities. Collection is now a complete Data Processing Artefact in TDL.

This chapter provides in depth knowledge of the various enhancements in attributes, modifiers, method formula syntax and symbol prefixes. The foremost focus 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. Definition, Attribute and Modifier Enhancements

In Tally.ERP 9, new attributes and modifiers have been introduced to support the new capabilities. The behaviour of some of the existing attributes and modifiers has also changed.

1.1 Attribute Enhancements

New Attributes

New attributes that have been introduced are explained in this section.

Field Attribute – Set By Condition

The attribute **Set By Condition** is similar to a conditional ‘Set’ at Field level. If multiple ‘Set By Condition’ are 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 a formula.

Example:

```
[Field : Sample SetbyCondition]
```

```
Set as           : "Default Value"
```

```
SetbyCondition : ##Condition1 : "Set by Condition 1"
```

The Field **Sample SetbyCondition** will contain the value ‘Set by Condition1’ if the expression *Condition1* returns TRUE, else the Field will contain the value ‘Default Value’.

Field Attribute – Tool Tip

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 provide additional meaningful information using this attribute.

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. As against attributes 'Info' or 'Set As', this attribute value is independent of the Field Width.

Syntax

```
Tool Tip : <Value>
```

Where,

<Value> can be a String or a Formula.

Example:

```
[Field : Led Name]
```

```
Storage      : Name
```

```
Tool Tip     : "Please Enter the Name of the Ledger"
```

Report Attribute – Full Screen

It helps to control the display of command window/calculator pane. It is a logical type of attribute.

Syntax

```
Full Screen : Yes/No
```

If it is set to YES, the command window will be hidden, providing extra space when the report is displayed. The default value of this attribute is YES. In case of Sub-Report/AutoReport, if the value of this attribute is not specified, the default value is NO.

Example:

```
[Report : My Report]
```

```
Full Screen : Yes
```

Part Attribute – Retain Focus

It indicates that the part should retain information about the line which is currently in focus, even if the focus is moved to other part. This allows the part to make the same line as the current line when it gets back the focus.

Syntax

```
Retain Focus : Yes/No
```

Example:

```
[Part : LedPart]
```

```
Retain Focus : Yes
```

Part Attribute – Default Line

It 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 the collection of Legers, then the following code will highlight the line of Cash Ledger.

```
[Part : The Main Part]

    Default Line : $Name = "Cash"
```

Collection Attribute – Sub Title

Along with the Table title, sub titles for the columns can also be given. The attribute 'Sub Title' has been introduced in the 'Collection' definition for the same.

Syntax

```
Sub Title : <List of Comma Separated Strings>
```

Where,

<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"
```

It displays a table with two columns. Column titles are also displayed, using attribute Sub Title. Instead of using Sub Title attribute multiple times, a comma separated list can be given as follows:

```
Sub Title : $$LocaleString : "Name", $$LocaleString : "Op.Balance"
```

Behavioural Changes of Attributes

Enhancements have been done in the behaviour of the following attributes:

Attributes 'Set As' and 'Info'

As of Release 2.x, the attributes 'Set as' and 'Info' were treated as the same attribute with aliases. When 'Info' was 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 behaviour has been modified to treat both as individual attributes. When both these attributes are specified in a field, 'Info' takes precedence and 'Set as' is ignored.

Attribute 'Format'

When a collection is a union of collections, the 'Format' attribute in the collection behaves as a place holder for the columns. It is mandatory to specify 'Format' attribute in individual collections, when a collection is a union of collections.

Example:

```
[Collection : LedTable]

    Collection : DebtorsLedTable, CreditorsLedTable

    Format      : A, 20

    Format      : B, 25
```

Here, A and B act as dummy identifiers, and the second parameter is width. The collections **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
```

It displays a table of two columns. The width of first column is 20 and of second column is 25.

Attribute 'Sync'

The behaviour 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 value of ‘Sync’ attribute being set to NO, in the above code snippet, the Sync attribute finally has the value as YES.

Attribute ‘Child Of’ - to support Voucher Type

‘Child Of’ attribute is enhanced further to support Voucher Type. Now, a Collection of Vouchers of a particular Voucher Type can be constructed. Prior to this release, the same could be achieved by applying Filters to the Collection. But, the enhanced 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 the Collection.

<String Formula> can be a formula which results into the name of the Voucher Type

<Belongs To> is an optional attribute, which if used, takes **<Logical Value>**, i.e., YES or NO.

Example: 1

```
[Collection : Sales Vouchers]

    Type      : Voucher Type

    Child of  : $$VchTypeSales
```

‘Sales Vouchers’ is a collection of Vouchers, whose Voucher Type is the predefined voucher type ‘Sales’.

Example: 2

```
[Collection : Sales Vouchers]

Type      : Voucher Type

Child of  : $$VchTypeSales

Belongs To : Yes
```

'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'.

1.2 Modifier Enhancements

In TDL, attribute modifiers are classified as Static/Load time or Dynamic/Run-Time modifiers. Use, Add, Delete, Replace/Change are Static/Load Time modifiers. Option, Switch and Local are RunTime modifiers. The sequence of evaluation is generalized across 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

New Modifiers**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 modifier 'Option' compulsorily evaluates the conditions for all the options provided in the description code, and applies all the option statements which satisfy the evaluation conditions. This means that it is not 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 modifier '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, 'Switch' statements 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 group.

Syntax

```
Switch : <Label> : <Definition Name> : <Condition>
Switch : <Label> : <Definition Name> : <Condition>
```

If multiple 'Switch' statements are mentioned within a single definition, then the evaluation will be carried out up to the point where the 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
```

Here, out of multiple switch statements having same label, zero or one statement is 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
```

Here, multiple switch groups are created, and zero or one of the switch cases would be applied from each such group or label.

Behavioral Changes for Attribute Modifiers

The behaviour of the following attribute modifiers has been enhanced.

Changed precedence of Use

Behaviour of the attribute 'USE', which is used to inherit the properties from other definitions, has now changed. Irrespective of the order of specification of attributes within a definition, USE will be evaluated first. In other words, the order in which USE is specified is immaterial, as in any case, it will be evaluated first. If multiple USE attributes are specified in a single definition, they are evaluated in the order of their occurrence.

Example:

```
[Field : Attr Use1]
```

```
Set as : "This shows the changed behavior of 'Use' attribute"
```

```
Style : Large Bold
```

```
Use : Name Field
```

The Field **Attr Use1** uses existing Field **Name Field**. Since USE is having higher precedence over other attributes, Field **Attr Use1** will inherit all the attributes of **Name Field**. But, the style **Large Bold** at the Field **Attr Use1** will override the inherited Style within the Field **Name Field**.

Changed behaviour of Delayed Attribute Modifiers "Add/Delete/Replace"

Static/Load Time modifiers like Add, Delete and Replace can be called as Delayed Attribute modifiers, as they are having least precedence among Delayed Static/Load Time modifiers.

Now these modifiers are generalized across all definitions. Earlier for definitions *Report, Key, Color, Style, Border* and *Variable*, the delayed attributes were applied in their sequence of

appearance in the definition description. If more than one delayed attribute is used under any definition, then the attributes will be applied as they appear. This has been done to bring consistency across the definitions.

Example: 1

```
[Report : Test Report]

    Form      : Form1

    Delete    : Form

    Form      : Form2
```

The report **Test Report** won't have any Form, as the attribute 'Delete', which is evaluated last, deletes all the existing forms.

Example: 2

```
[Report : Test Report1]

    Form      : Form1

    Delete    : Form

    Add       : Form : Form2
```

As a result of this code snippet, the Report *Test Report1* will have one Form *Form2*, since, on deletion of all the Forms, Delayed attribute modifier *Add* is used to add a new Form *Form2*.

Enhanced Syntax of Delayed Attribute "Local"

Delayed attribute modifier 'Local', which is used to locally modify the attributes of any child definition, is now enhanced to accept nested Locals.

Syntax

```
Local : <DefinitionType1> : <DefinitionName1> [: <DefinitionType2> +
      : <Definition Name2> : ... ] : <Attribute> : <Value>
```

Where,

<Definition Type> can be a Form, a Part, a Line or a Field.

<Definition name> is the name of the definition type.

<Attribute> is the attribute of the Definition of which, the 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 : "SalesAmount"
```

The Field *Amt Field* is localized at the Report *Custom Report*, by using nested locals.

1.3 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 have now been 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 existing TDL Codes continue to work, without any changes required for the same. These warnings are thrown only by the compiler, during the compilation using TD9.

However, it is advisable to use # for existing System Formula alteration, and refrain from using # for System Menu Keys, Form Keys and UDF Definition, or using ! for any system descriptions.

1.4 Partial Attribute Support

Prior to Tally.ERP 9, all descriptions supported partial search on their attribute words. For example, '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 behaviour 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.

1.5 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 empty Menu Item.

2. Enhanced Special Symbols

In Tally.ERP 9, some new symbols have been introduced and the behaviour of the definition modifier '#' has been 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

Scope of modifying definitions using # is extended to System Formula definition, that is, to alter the value of the existing system formula. It helps to improve the performance with optimized formulae.

Example:

```
[#System : Formula]

    NameWidth      : 40

    MaxNameWidth   : 60
```

Here, the values specified to Formulae *NameWidth* and *MaxNameWidth* in DefTDL, are changed.

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 +
           & newlydefined"

    Lines : 1
```

3. Method Formula Syntax with Relative Object Specification

‘\$’ operator has been enhanced with new capabilities. It allows direct access to any object method, including its sub-collections to any level, with a dotted notation framework. Using this new capability, there is no need to repeat a line over a sub-collection to access it. 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 has been introduced to support access out of the scope of the Primary Object and to access Sub object at any level using (.) dotted notation with index and condition support.

The **new 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 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, the 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 keyword First or Last which is equivalent to specifying 1 or -1 respectively.

If both Index and Condition are 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 the '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, any Interface object exists in the 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 a data object. In Tally.ERP 9, Object association has 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 the Type of any Primary Object, and

<ObjectIdentifierFormula> is any formula which evaluates to the name of a Primary Object. It is optional.

Example: 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 the previous Report.

Example: In Tally.ERP 9

```
[Report : Sample Report]
```

```
    Object : Ledger : "Cash"
```

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

4.2 Part Level Object Association

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

Using 'Object' attribute specification in 'Part' definition.

Syntax: Prior to Tally.ERP 9

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

Where,

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

<SeekTypeKeyword> can be **First** or **Last**, which denotes the position index, and

<SeekCondition> is a filter condition to the supplier collection. It is optional.

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 the stock Item "Tally Developer" is associated with the Part 'Sample Part'.

Using 'Object Ex' attribute specification in 'Part' definition

From Tally.ERP 9 onwards, data object can be associated to Part by using the 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, 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 is used, the Object specification 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 the Part 'Sample Part'.

4.3 Line Level Object Association

An object can be associated to a Line by Part attribute "Repeat". Now, the Part attribute 'Repeat' has been enhanced to support the following:

- a. Extraction of collection from any Data object
- b. Extraction of collection from Interface Object associated Data object. This aspect will be elaborated in the section "Object Access via Interface Object".

Repeat Syntax: Prior to Tally.ERP 9

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

Where,

<Coll Name> is the name of the Collection. If the Collection is present one level down in the object hierarchy, then the 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 of **ledger entries** which satisfies the given condition.

In Tally.ERP 9 – Repeat Syntax

```
Repeat : LineName : 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, a field inherits the object 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 primary object, the 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 a unique access name. For this purpose, a new attribute 'Access Name' is introduced for 'Part' and 'Line' definitions.

Syntax

```
Access Name : Access Name Formula
```

Where,

<Access Name Formula> can be a formula which evaluates to a 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 a unique Access Name.

5.2 Value Extraction

Once Part and Line Interface objects are able to uniquely identify by 'Access Name', then the data object can be accessed by either the new function \$\$ObjectOf or 'New method formula syntax'.

Value Extraction by function \$\$ObjectOf

Methods of data object, which is associated to Interface Object, can be extracted by using the function \$\$ObjectOf.

Syntax

```
$$ObjectOf : <DefinitionType> : <AccessNameFormula> : <EvaluationFormula>
```

Where,

<DefinitionType> may be 'Part' or 'Line'

<AccessNameFormula> is a string through which a Part or Line can be uniquely identified, and

<EvaluationFormula> is a method that needs to be evaluated.

Example:

Line 'Sample Line' has Access Name as 'Sample Acc Name' and is in association with Ledger Object.

```
[Field : Sample Field]
```

```
Set as : $$Objectof : Line : "Sample Acc Name" : $Name
```

Field 'Sample Field' displays the name of the object Ledger which is associated with a Line whose access name is "Sample Line Acc Name".

Value Extraction by using new method formula

Methods of data object, which is associated to Interface Object, can also be extracted by using new method formula. With this approach, sub object's methods can be extracted.

Example:

Line 'Sample Line' has Access Name as 'Sample Acc Name' and in association with Ledger Object.

```
[Field : Sample Field]
```

```
Set as : $(Line, "MyLineAccessName").BillAllocations[1].OpeningBalance
```

Field 'Sample Field' displays the name opening balance of a ledger which is associated with a Line whose access name is "Sample Line Acc Name".

Repeat Syntax Using Access Name

Collection inside the data object, which is associated to the Interface Object, can be extracted by using the 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 : Sample Part]
```

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

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 the Part having the access name "MyPartAccessName"

6. Bracket support in TDL

Prior to Tally.ERP 9, usage of TDL language token bracket ('(' and ')') was restricted as mathematical operator only. From this release onwards, brackets can be used in 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 a function required expression and that expression contained any language token, then the TDL programmer was 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 places where the function parameter expects an identifier or a constant value.

Example: 1

Field 'Sample Fld' displays the first 5 characters of currently loaded Company's email address.

Prior to Tally.ERP 9

In this case, the First parameter to the function **\$\$StringPart** is an expression that contains the language token ':'. So, a formula needs to be created.

```
[Field : Sample Fld]
```

```
Set As : $$StringPart : @CmpEmailAddress : 0 : 5
```

```
CmpEmailAddress : $Email : Company : ##SVCcurrentCompany
```

In Tally.ERP 9

```
[Field : Sample Fld]
```

```
Set As : $$StringPart : ($Email:Company : ##SVCcurrentCompany) : 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, a formula 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 was a part of language syntax, language tokens were not permitted. This restriction led to the necessity of additional formulas, even when the formulas were 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 when 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

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

```
Line   : Sample Line
Repeat : (Sample Line) : My Collection
```

First parameter for 'Repeat' attribute is using bracket for identifier.

6.3 As a Mathematical Operator

In TDL, brackets are used as mathematical operator to set the precedence of evaluation.

Example:

```
[Field : Sample Fld]
```

```
Set As : 4*(5+6)
```

If parentheses are not used, then the Field 'Sample Fld' will display 26, otherwise 44.

7. Action Enhancements

Some of the existing actions have been enhanced to support the multi-line selection capabilities. Several new actions have also been 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** have been introduced. Scope attribute is introduced in **Key** definition and Selectable attribute is available at **Part** and **Line** definitions.

Attribute 'Scope' introduced in 'Key' definition

Through this attribute, scope for the Action(s) can be specified.

Syntax

`Scope : <Scope Keyword>`

Where,

<Scope Keyword> can have any of the following possible values: - *Current Line/Line, All Lines, Selected Lines, Unselected Lines and Report.*

Attribute 'Selectable' introduced in 'Part' and 'Line' definitions

Part Definition

At Part level, the attribute 'Selectable' indicates whether the lines owned by the particular 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 whether the line (or lines within the line) is selectable or not. The default value of attribute 'Selectable' for repeated lines is 'YES', and for non-repeated lines is 'NO'. The value is also inherited from Parent Part/Line, and the same can be overridden at Line level.

Syntax

`Selectable : <Logical Formula>`

where,

<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** have been enhanced to work with 'Report' scope.

7.2 New Actions

Following new actions have been introduced in the language:

Action - Modify Object

The action 'Modify Object' has been enhanced to alter a method of an object at any level. 'Modify Object' 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[,MethodName : <value> , ...]+
    [,<SubObjectPathSpec>.MethodName : <value>, ...]
```

The specifications given for <PrimaryObjectSpec>, <SubObjectPathSpec>, MethodName remain the 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, that is, 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 the **Name**, **Amount** and **Date** methods of the Sub object Bill Allocations in one line.

Action 'Modify Object' in a Menu Definition

In Menu definition, a button which has the action 'Modify Object', can be added.

Example:

```
[#Menu : Gateway of Tally]
```

```
Add : Button : Alter My Object
```

While associating a key with action 'Modify Object', the 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 assuming Menu Object as the requestor.
- ❑ Even Method values pertaining to Company Objects can be modified.
- ❑ A button can be added at the menu to specify the action Modify Object at the Menu level

Action - Set Object Values

This new action is similar to the action 'Modify Object'. The action 'Set Object Values' works 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 <Method Name>.

This action alters the current object in memory. When the Primary object is saved, the changes will be updated in Tally database.

Example:

```
[Key : My Key]
```

```
Action : Set Object Values : Opening Balance : ($$AsAmount : 10)
```

Action – Backup Company

The action ‘Backup Company’ 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. They can be repeated for multiple companies.

Example: Single Company

```
[Button : My Cmp Bk Button]
```

```
Title : BackUp Cmp
```

```
Action : BackUp Company: ", " : "C:\,C:\Tally.ERP 9\Data,Global +
      Enterprises,10037"
```

```
Key : Alt + G
```

Example: Multiple Company

```
[Button : My Cmp Bk Button]
```

```
Title : BackUp Cmp
```

```
Action : BackUp Company : ", " : "C:\,C:\Tally.ERP 9\Data,+
Global Enterprises,10037,C:\,C:\Tally.ERP 9\Data,+
TDL Demo,10027"
```

OR

```
[Button : My Cmp Bk Button]
```

```
Title : BackUp Cmp
```

```
Action : BackUp Company : ", " : @@MyCmpFor
```

```
[System : Formula]
```

```
MyCmpFor : "C:\,C:\Tally.ERP 9\Data,Global Enterprises,10037, +
C:\,C:\Tally.ERP 9\Data,TDL Demo,10027"
```

Action – Restore Company

This action allows to restore multiple companies in one go.

Syntax

```
Restore Company : <parameter sep char> : <String Formula>
```

Where,

<Parameter Sep Char> is a character used to separate parameter.

<String Formula> must evaluate to the value in the following order 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 the name of the Company.

<Company Number> is the number of the company.

These four values must be specified for each company. These can be repeated for multiple companies.

Example: Single Company

```
[Button : My Cmp Res Button]
```

```
Title : Restore Cmp
```

```
Action : Restore Company : ", " : "C:\Tally.ERP 9\Data,C:\,+
```

```
Global Enterprises,10037"
```

Example: Multiple Company

```
[Button : My Cmp Res Button]
```

```
Title : Restore Cmp
```

```
Action : Restore Company : ",", : "C:\Tally.ERP 9\Data,C:\,+  
Global Enterprises,10037,C:\Tally.ERP 9\Data,C:\,+  
TDL Demo,10027"
```

OR

```
[Button : My Cmp Res Button]
```

```
Title : Restore Cmp
```

```
Action : Restore Company : ",", : @@MyCmpFor
```

```
[System : Formula]
```

```
MyCmpFor : "C:\,C:\Tally.ERP 9\Data,Global Enterprises,10037, +  
C:\,C:\Tally.ERP 9\Data,TDL Demo,10027"
```

Action – ChangeCrypt Company

This action allows to change the TallyVault 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 **<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 the name of the Company.

<Company Number> is the number of the company.

These 5 values must be specified for each company, and can be repeated for multiple companies.

Example:

```
[Button : Chg Pwd]

  Title   : Change Pwd

  Key     : Alt + b

  Action  : ChangeCrypt Company : ", " : "C:\Tally.ERP 9\Data\10037, +
           NewPwd,OldPwd,Global Enterprises, 10037"
```

Action – Browse URL

It is used to open web browser with any URL formula passed as a parameter. A list of parameters separated by space can be specified, if the application accepts command line parameters. **Exec Command** is an alias for action Browse URL.

Syntax

```
Action : Browse URL : <URL Formula> [:<command line parms>]
```

Where,

<URL formula> is an expression which evaluates to any link to a web site.

<command line parms> is the List of command line parameters, separated by space. 'Browse URL' Key Action can be used to open web browser with any URL Formula.

Example:

```
[Button : Open Notepad]

  Title   : $$LocaleString:"Notepad"

  Key     : ALT + N

  Action  : Exec Command : Notepad : "Browse URL.Txt"

[Form : Hyperlink]

  Parts   : Hyperlink

  Button  : Open Notepad
```

Example: Field acting as a hyperlink

```
[Key : Execute Hyperlink]

  Key     : Left Click

  Action  : Browse URL : "www.tallysolutions.com"

[Field : Hyperlink Company]

  Color   : Blue

  Border  : Thin Bottom
```

```
Key      : Execute Hyperlink
Set as   : "Tally Solutions Pvt. Ltd"
Local   : Key : Execute Hyperlink : Action : Browse URL : +
          http://www.tally.co.in
```



Existing Action **Register Tally** has been removed for generalization; which is now replaced with Action **Browse URL**.

Action – HTTP Post

A new Key/Button Action **HTTP Post** has been introduced which will help in exchanging data with external applications using web services. In other words, 'HTTP Post' Action can be used to submit data to a server over HTTP and gather the response. This will enable a TDL Report to perform a HTTP Post to a remote location.

This Action will be discussed in detail under the topic **HTTP XML Collection**.

Action – Refresh TDL

A new Key/Button Action **Refresh TDL** has been introduced, which allows the TDL programmer to reload the active TDL Files, without having to restart Tally.

Syntax

```
Action : Refresh TDL
```

Example: Field acting as a hyperlink

```
[Key : Refresh TDLs]
```

;; Any Key can be assigned if Report already have F5 assigned

```
Key      : F5
```

```
Action : Refresh TDL
```

;; Refresh TDL will work from any Report

```
[#Form : Default]
```

```
Key : Refresh TDLs
```

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** has been introduced that can be specified within the **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> can be any one of the actions.

<Action Parameters> are the parameters of the action specified.

Example:

```
[Form : TestForm]
```

```
On : FormAccept:Yes:HttpPost : @@SCURL : ASCII : SCPostNewIssue : +
    SC NewIssueResp
```

8.2 Event – On Focus

A new Event **On: Focus** has been 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
```

Where,

<Condition> should return a logical value.

<Action> can be any action.

<Action Parameters> are parameters of the action specified.

Since On : Focus is a list type attribute, 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 result 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 have been introduced in the Language:

10.1 Function - `$$IsObjectBelongsTo`

The 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 an aggregation of objects. This function allows an easy link back into the native object type, and walking 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 the Name of the Object.

<BelongsToName> denotes the name of the object Type.

Example:

Whether Group *North Debtors* belongs to Group *Sundry Debtors* or not, can directly or indirectly be checked using the following statement:

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


10.2 Function - \$\$NumLinesInScope

Tally.ERP 9 onwards, various operations can be performed on multiple lines. To know how many lines were considered for any operation, the 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 the total number of selected lines in the Part to which it belongs.

10.3 Function - \$\$DateRange

A new Built-in function **\$\$DateRange** has been 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 Due Date.

<Due Date Expression> is a String Expression, and evaluates to Date.

<Flag> is a logical expression, and decides whether to include date given in second parameter.

Example:

```
SET VALUE: OrderDueDate : $$DateRange: "10 Days" : $Date : True
```

Method 'Order Due date' will have value as "10 Days" from \$Date, with \$Date also inclusive.

When you export a voucher to XML with due date, the tag **<orderduedate>** contains two attributes called **JD** and **P**. **JD** is the Julian Day of the given date. **P** is the due date mentioned in the field.

Julian Day (JD) value is calculated internally based on the calculation method of Tally.



All value extraction can now be achieved using GroupBy; hence \$\$Extract functions have been removed from Tally.ERP 9, e.g., \$\$ExtractGrpVal, \$\$ExtractLedVal, etc.

10.4 Function - \$\$IsCollSrcObjChanged

You can use the function `$$IsCollSrcObjChanged` to know when the source object is changed to the next source object, while the Walk or WalkEx attribute is being executed.

Syntax

```
$$IsCollSrcObjChanged
```

The return value of this function is Logical. It returns **True** for the source object and the first walked object. It will return **False** for the subsequent objects that are walked in the same source objects, till the source object is changed.

This function is especially useful when you want to aggregate the values for a source object while walking through multiple source object. This function will indicate that a source object is changed to the Aggregation computation expression.

Example:

```
[Collection: TN Summary Counts]
```

```
Source Collection      : My Vouchers
By                    : Summary      : "Summary"
Aggr Compute          : TotalIncluded : Sum : IF $$IsCollSrcObjChanged +
                        AND $IsVCHIncluded Then 1 Else 0
Aggr Compute          : TotalUncertain : Sum : IF $$IsCollSrcObjChanged +
                        AND $IsVchUncertain Then 1 Else 0
```

10.5 Function - \$\$CollSrcObj

You might need to evaluate some expression from the source collection context while walking in the sub-collection context to compute some value. For this purpose you can use the function `CollSrcObj` evaluate expression in the context of source object while walking the current object.

Syntax

```
$$CollSrcObj:<Expression>
```

Where,

Expression is any expression that needs to be evaluated in the context of the source object (methods, local formulae, and so on).

Example:

```
[Collection: TNAnnexureSummaryTemplate]
```

```
Source Collection      : My Vouchers
Walk                  : AllInventoryEntries
Compute               : TNeVATVchNo      : $$CollSrcObj:$VoucherNumber
```

```

Compute          : TNeVATVchDt          : $$CollSrcObj:$Date
By               : ...
AggrCompute     : ...

```

This function will work only when it is used in a collection context.

11. Enhanced Collection Capabilities

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

Attribute - Source Collection

In context of 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 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.

Attribute - Walk

Attribute **Walk** allows specifying further elements to walk on the source. 'Walk' is optional and if not specified, the methods pertaining to the 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-Object/ 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 the 'Inventory Entries' Object are available to the current collection.

In case objects pertaining to Batch Allocations are required, the Walk can be written as:

```
Walk : Inventory Entries, Batch Allocations
```

wherein, all the methods within Batch Allocations will be available to the current collection.

Attribute - 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.

Attribute - 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, or the Last value 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, Min, or Last*.

<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.

Attribute - 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 the attribute **Compute**.

Syntax

```
Compute : <Method-Name> : <Method-Formula>
```

Example:

```
Compute : Date : $Date
```

Method **Date** is being declared and made available for subsequent use.

Attribute - ReWalk and ReCompute

Many a time, there are requirements where objects being walked in a collection need to be walked once for certain computation followed by, ensuring that these computations are reflected back as methods in all the objects. The attributes **ReWalk** and **ReCompute** are introduced for this purpose. The objects are being walked in the first iteration with the values accumulated in some variable and if **ReWalk** is **Yes**, these objects are walked again so that the accumulated values are re-computed as methods along with objects

Syntax

```
ReWalk : <Logical Constant>
ReCompute : <MethodName> : <Expression>
```

Where,

<Logical Constant> is either Yes or No.

<MethodName> refers to the method where the result can be stored and referred to later.

Example:

```
/* Regular voucher collection for source */
[Collection: My Vouchers]
    Type          : Vouchers

/* Main Aggregate collection uses By with Walk and ReWalk for totling item amount
Query to execute:
Select $Item, $TotalAmount, $ItemTotalAmount from MainAggrCollection
*/
[Collection: Main Aggr Collection]
    Source Collection : My Vouchers
    Walk              : Inventory Entries

;; Compute var for keeping total from first walk (NOTE: Need to reset this for every voucher)
    Compute Var      : WalkTotal : Amount: $$NettAmount:##WalkTotal:$Amount

;;item name
    By                : Item : $StockItemName

;; itemwise total
    Aggr Compute     : TotalAmount : Sum: $Amount
```

```
; rewalk
```

```
Rewalk           : Yes
```

```
;; must be same as total amount (but reflects for all entries)
```

```
ReCompute       : ItemTotalAmount : ##WalkTotal
```

Attribute - 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 have 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 methods of collection Batch Allocation

```
Fetch : StockItemName, BilledQty, Amount, BatchAllocations.*
```

Attribute - Source Fetch

You can use the attribute **Source Fetch** to fetch methods from the source object context while in the current object context within Walk.

Syntax

```
Source Fetch: MethodSyntax, ...
```

Where,

MethodSyntax is the method or collections to be fetched from the source object context.

Example:

```
[Collection: TNAnnexureSummaryTemplate]
```

```
Source Collection    : My Vouchers
Walk                : AllInventoryEntries
Source Fetch       : Date
By                 : ...
AggrCompute        : ...
```

Attribute Source Fetch is not relevant in a Simple Collection or an Extract Collection without Walk and will be ignored.

If the source collection does not have Walk or WalkEx attributes, then Source Fetch will be ignored. Source Fetch will also be ignored if the source collection is of type native collection.

Attribute - Prefetch and Source Prefetch

When a collection is being gathered, Fetch is used to fetch the methods at source and also bring them to the target object. In TDL, we also define multiple external methods at object level which can also be fetched using the Fetch attribute. This ensures that the resultant method is created at source and is also pulled into the target object. Any further access to these external methods using \$ will provide the value directly, avoiding evaluation of method again.

However, there are scenarios, where Compute Var variables are used to evaluate expressions containing these external methods. Since Fetch is always evaluated after Compute Var, any access to external methods during Compute Var evaluation will end up evaluating again and again.

To solve this problem, an attribute **Prefetch** is introduced to declare the methods that are required to be fetched prior to Compute Var such that these Methods are available for usage within the current context thereby not requiring to re-evaluate the same expression each time they are used.

The sequence of methods declared in the Prefetch should be in the order of dependency, for instance, if Method 3 uses Method 2, and Method 2 uses Method 1, then, the declaration order in Prefetch should be Method 1, Method 2. A similar **Source Prefetch** is provided for the case of Source object context.

Syntax

```
PreFetch: <Method1,...>
Source Prefetch: <Method 1,...>
```

Where,

<Expression> is the value that needs to be evaluated.



- ❑ Source prefetch is applicable only when the walk is provided.
- ❑ Prefetch is evaluated before Compute Var but after Source Var.
- ❑ Source Prefetch is evaluated before Source Var

Example:

```
[Collection: LT_SummaryVchColl_With_PF]
```

```
Source Collection      : LT_SimpleVchColl
Walk                  : LedgerEntries
By                    : LedgerName          : $LedgerName
Aggr Compute          : Amount              : Sum              : $Amount
```

/ Two External Object Methods are pre-fetched so that these expressions get evaluated only once and used multiple times */*

```
Prefetch              : LedgerParent, IsLedDebtorCreditor
Compute Var           : vIsCreditAllowed : Logical : If $$IsEmpty:$LedCreditPer +
                        then No else Yes
Compute               : IsCreditAllowed  : ##vIsCreditAllowed
```

```
[#Object: Ledger Entry]
```

```
LedgerParent          : $Parent:Ledger:$LedgerName
IsLedDebtorCreditor  : $LedgerParent = $$GroupSundryDebtors OR +
                        $LedgerParent = $$GroupSundryCreditors
IsPartyLedBillWiseOn: If $IsLedDebtorCreditor then +
                        $IsBillWiseOn:Ledger:$LedgerName else No
LedCreditPer         : If $IsLedDebtorCreditor and
                        $IsPartyLedBillWiseOn then +
                        $BillCreditPeriod:Ledger:$LedgerName else ""
```

```
[Collection: LT_SimpleVchColl]
```

```
Type                  : Voucher
```

As we can observe, that the Method LedgerParent is being prefetched, as a result of which, while evaluating the subsequent Method IsLedDebtorCreditor, the value of LedgerParent will be available and is not required to be re-evaluated. Similarly, since IsLedDebtorCreditor is prefetched, the subsequent methods IsPartyLedBillwiseOn and LedCreditPer using this method are also using the prefetched value, without re-evaluating.

Attribute - 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 the parent one level up

- . - Single dot retains the data of the source collection in current object.
- .. - Double Dot retains the data of the Source Collection in current object's parent.
- ... - Triple Dot retains the data of Source Collection in 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 that 'Keep Source' is not applicable, as there is no actual source collection created.

Attribute - 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.

Attribute - 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**> is the data source file path.

<**Encoding**> is ASCII or UNICODE. It is Optional. The default value is UNICODE.

Example: 1

```
[Collection : My XML Coll]
DataSource : File Xml : "C:\MyFile.xml"
```

In this code snippet, the type of file is 'File XML' as the data source is XML file. The encoding is UNICODE by default, as it is not specified.

Example: 2

```
[Collection : My XML Coll]
DataSource : HTTP Xml : "http://localhost/MyFile.xml" : ASCII
```

In this code snippet, the type of file is 'HTTP XML', as the data source is obtained through HTTP. The encoding of the file 'MyFile.XML' is ASCII.



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

Data Roll up/summarization capability in TDL Collection

Data roll up/ summarization capability facilitates 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 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, resource requirement and enhanced performance by orders of magnitude especially concerning reports generation.

Example: 1

```
[Collection : My Source Collection]

    Type : Voucher

[Collection : My Summary Collection]

    Source Collection : My Source Collection

    Walk              : Ledger Entries

    By                : MyLedgeName : $LedgerName

    Aggr Compute      : My Total      : Sum : $Amount
```

In this 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 scenarios, current object itself is required as a Source and aggregation is performed on collection obtained from it or its sub-collections. In such circumstances, if we use Source Collection as Voucher, then the entire vouchers within the company will be scanned

unnecessarily to find the current one, which is a time consuming process. To avoid this, we can use *Source Collection: Default*, which will assume the current voucher as a Source.

```
[Collection : LedgerInAccAllocations]
```

```
Source Collection : Default
Walk             : InventoryEntries, AccountingAllocations
By              : LedgerName : $LedgerName
Compute         : RateOfVA : $RateOfVAT:TaxClassification:+
                  $TaxClassificationName
Aggr Compute    : Amount : Sum : $Amount
Filter          : IsVATLedgerinAcc
```

```
[System: Formula]
```

```
IsVATLedgerinAcc : $$IsSysNameEqual:VAT:$TaxType:Ledger:$LedgerName
```

While printing a voucher as an invoice, if an aggregation has to be done on its tax ledgers to show as summary within the invoice, this has to be collected from the accounting allocations of the same voucher.

Collection re-use, extraction and chaining support in TDL Collection

A collection can extract information from other collections, including its sub-objects with the choice of method(s), filter(s) and sort-order. Source Collection within a collection/collection(s) can be chained. In other words, Summary Collection can be used as Source Collection for some other Collections, 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 this code snippet, *My Parent Summary Collection* extracts a sub-set of information from a collection to an already summarized collection *My Summary Collection*.

Indexed or Searchable Collection on TDL defined keys

The capabilities discussed above extend the data gathering capabilities of TDL. However, business reporting in general and in Tally uses hierarchical presentation or columnar presentation, rather than simple table representation. This creates a unique and natural experience of working with the product and business data.

In case one simply repeats the summarized collection and gets the desired report, everything works fine with the existing capabilities. However, if the Report is having two or more dimensions like Ledger, 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 is 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 the 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 their 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

<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 the 'Summary Collection', 'LedgerName', 'Total' and 'Parent' are exposed to the 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.

HTTP – XML Collection

Consider the following XML data stored in file *TestXML.xml* 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 to, 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 the 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 parameters as follows:

```
<Node Name> : <Position>
```

<Path-to-Start-Node> is used to specify the path to reach the *<start node>* from the root node.

The path specification is:

```
<Root-node> : <Child Node> : <Start Pos> : <Child Node> : <Start Pos> ...
```

XMLObject attribute is used to specify the TDL Object specification. The following syntax is used for object specification:

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

/ The second Parameter in the Collection Type can be a Object type in case of a complex collection or a simple data type in case of simple collection */*

All these attributes cater to specific requirements based on the GET request or POST request, and whether the obtained data is stored in Tally.

Prerequisites for data transfer over HTTP

In order to retrieve the data available in *TestXML.xml* file from a remote server (Pre-defined IP Address), ensure that web service is running on the machine. Check for IIS Server Installation. The file *TestXML.xml* can be copied to the directory *C:\inetpub\wwwroot* to be accessible at the root and then the URL can be specified as follows: *http://localhost/TestXML.xml*.

If the XML request needs to be processed at the remote server by a file (.asp, .php, etc.), at least one web server (e.g., IIS, Apache etc) and PHP/ASP must be installed on the system.

Simple GET Request

If it is required to access the data (XML format) from remote server in a collection, it is sufficient to specify the URL of the server only. The attribute **RemoteURL** is used. The data thus obtained is available in the collection as objects and can be accessed as native methods.

The collection to populate XML Data available at the URL *http://Remoteserver/TestXML.xml* is created as follows:

Example:

```
[Collection : XML Get Collection]
```

```
Remote URL : "http://RemoteServer/TestXML.xml"
```

This collection can be used in a TDL Report to display the data retrieved. The method names will be same as the XML Tag names.

By default, all the data from XML file is made available in the collection. If only a specific data fragment is required, it can be obtained using the collection attribute *XML Object Path*.

Example:

From the XML file, if only the 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 the 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 the third customer, the collection is defined as follows:

```
[Collection : XML Get CollObjPath]
```

```
Remote URL      : "http://Remoteserver/TestXML.xml"
```

```
XML Object Path : PHONE : 2 : CUSTOMERS : CUSTOMER : 3 : ADDRESS : 1
```

Simple GET Request and mapping the response to TDL Object

The data available in XML format is at the URL “http://Remoteserver/TestXML.xml”. The data is required to be mapped as TDL Objects. The collection attribute *XML Object* is used to specify the object name to which the obtained data is mapped.

Example:

```
[Collection : XML Get Collection]
```

```
Remote URL : "http://Remoteserver/TestXML.xml"
```

```
XML Object : Customer Data
```

The Object specification for “Customer Data” is as follows:

```
[Object : Customer Data]
```

```
Storage : Name : String
```

```
Storage : EmpId : String
```

```
Collection : Phone      : XML Phone Coll      ;; Complex Collection
```

```
Collection : ADDRESS    : XML AddressColl     ;; Complex Collection
```

```
[Object : XML Phone Coll]

  Storage : OfficeNo : String

  Storage : HomeNo : String

  Storage : Mobile : String

[Object : XML AddressColl]

  Collection : AddrLine : String           ;; Simple collection
```

A Simple POST

If a TDL report is to be sent to the HTTP server as an XML request, and the XML response is to be obtained in the collection, then the collection attribute "Remote Request" is used. The attribute "Remote Request" takes a Report name as a parameter, which sends the request in XML format to the web page on the remote server. The response data received from the server is then available in the collection.

Example:

The Test.php page on the remote server accepts the data in the following XML format.

```
<ENVELOPE>

  <REQUEST>

    <NAME>Tally</NAME>

    <EMPID>00000</EMPID>

  </REQUEST>

</ENVELOPE>
```

The following collection sends the request in the above XML format with the help of a TDL report XMLPostReqRep. 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 *XML Tag* attribute is used at 'Part' and 'Field' Definitions.

```
[Part : XMLPostReqRep]

  XML Tag : "REQUEST"

  Scroll  : Vertical

[Field : XMLPostReqRepName]
```

```

XML Tag : "NAME"

Set As : " Tally "

[Field : XMLPostReqRepPwd]

XML Tag : " EMPID "

Set As : " 00000 "

```

The XML Tag *<Envelope>* is added by Tally while sending the XML request.

The response received from *http://Remoteserver/test.php* page is the same XML given previously. The data now available in the collection can be displayed in a report.

Post Request with Pre-request Report

A Pre-Request report is required when some inputs are to be accepted from the user, and the XML Request is to be generated out of those inputs. In that case, a TDL report is used which has to be accepted first. If the data captured through pre request report has to be sent to remote server for processing, then it has to be made available in the Request report. The input report name is specified as Pre-Request report.

```

[Collection : XML Post Collection]

Remote URL      : "http://localhost/test.php"

RemoteRequest  : XMLPostReqRep, XML PreReqRep : ASCII

XMLObjectPath  : CUSTOMER

```

Report *XMLPostReqRep* sends the XML request to the page *Test.php* in the format described earlier. Before sending the XML request to the page, data entered in the report *XML PreReqRep* must be accepted. The data entered in the Pre-Request report can also be sent to the remote server in the XML request. Both the reports are triggered when the collection is referred.

Action – HTTP POST

A new Key/Button Action **HTTP Post** has been introduced which will help in exchanging data with external applications using web services. In other words, 'HTTP Post' Action can be used to submit data to a server over HTTP and gather the response. This will enable a TDL Report to perform an 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, which can be ASCII or UNICODE.

<**Request Report**> is the name of TDL Report used for generating the XML Request to be sent.

<**Error Report**> is displayed in case of failure.

<**Success Report**> is displayed when the post is successful.

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 or failure is determined by <STATUS> tag in the standard message format. If it is 1, it is a success, otherwise it is a 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 Request as well as 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 code snippet 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 the 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 the platform, and stored in the 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 the table was not possible. Data from ODBC Collection was also not possible to be displayed.

From this Release onwards, all limitations pertaining to the usage of Collections as Tables have been completely eliminated. Any Collection which can be created in TDL can be displayed as a table now. Collection with Aggregation and XML Collections can also be used as Tables.

Prior to this release, the following types of Collections could not be used as Tables:

- ❑ Voucher Collection As Table
- ❑ Collections with Aggregation As Table
- ❑ Displaying information at lower levels in Object hierarchy in a Table
- ❑ Displaying aggregate methods in Table
- ❑ Displaying ODBC Collection As Table
- ❑ Displaying XML Collection As Table

Let us consider the following examples to understand the capability in a better way:

Voucher Collection As Table

Now, the Vouchers can be displayed as table in a field.

Example: Voucher Collection as Table

```
[Collection : Vch Collection]

Type      : Voucher

Filter    : PurcFilter

Format   : $VoucherNumber, 10

Format   : $VoucherTypeName, 25

Format   : $PartyLedgerName, 25

Format   : $Amount, 15

[System  : Formula]
```

```
PurcFilter : $$IsPurchase : $VoucherTypeName
```

```
;;Field displaying Table
```

```
[#Field : EI OrderRef]
```

```
Table      : Vch Collection
```

```
Show Table : Always
```

Collection with Aggregation As Table

Example: 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: 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
```


General and Collection Enhancements

```

Aggr Compute      : Amount : Sum : $Amount
Format           : $Name, 25
Format           : $BilledQty, 25
Format           : $Amount, 25

```

[System : Formula]

```
PurcFilter : $$IsPurchase : $VoucherTypeName
```

;;Field displaying table

[#Field : EI OrderRef]

```
Table      : Summ Collection
```

```
Show Table : Always
```

ODBC Collection As Table

Example: Data fetched from Excel file in Collection displayed as Table

In this example, the excel file “*Sample Data.xls*” containing data is present in the path “C:\Sample Data.xls”. If the complete path is not specified, it locates the Excel file in Tally application folder.

[Collection : ODBC Excel Collection]

```

ODBC      : "Driver= {Microsoft Excel Driver (*.xls)};DBQ= C: +
           \Sample Data.xls"

```

```
SQL      : "Select * From [Ledgers$]"
```

```
Format  : $_1, 25
```

```
Format  : $_2, 20
```

```
Format  : $_3, 15
```

```
Format  : $_4, 25
```

;; Field displaying table

[#Field : EI OrderRef]

```
Table      : ODBC Excel Collection
```

```
Show Table : Always
```

XML Collection as Table

This refers to XML Data fetched from Remote URL in Collection as Table. Following 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 this example, the complete URL of the file is *http://localhost/XMLData.xml*. Here, the file *XMLData.xml* is located in the local machine.

Instead of local host, the IP Address of the machine or 127.0.0.1 can be specified. The web service should be installed in the machine.

Example:

```

[Collection : XML Table]

RemoteURL      : "http://localhost/XMLData.xml"

XMLObjectPath  : CUSTOMER

Format         : $NAME, 25

```

;; Field displaying Table

```

[#Field : EI OrderRef]

Table         : XML Table

Show Table    : Always

```

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 the object added depends on the specification in the TYPE attribute. In case the TYPE attribute has not been 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 hold 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 **NEWOBJECT** 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 behaviour continues, since they are added by their parent objects and not by the Collection.

12. Collection Capabilities for Remoting

Enabling access to your organizational data 'any-time, any-where', and yet being truly usable, is what Tally.ERP 9 is capable of. With remote access through Tally.NET Server, 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 the attributes 'FetchObject' and 'FetchCollection' at the Report level, significantly help in the above functionality.

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

User Defined Functions

Introduction

TDL is a comprehensive 4G 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. Before the introduction of User- defined functions, TDL had very few aspects of 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 an Action list.

Now, with the introduction of “User Defined Functions”, a path-breaking development in the history of TDL, procedural programming aspects have been 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 Subroutines or Procedures. If it is required to execute a certain set of statements repeatedly to achieve a certain functionality, it is not a good programming practice to write the same set of statements in the program again and again. For example, n separate statements in a computer program require the sum of two numbers for some complex computation. Each statement will repeatedly compute a+b, with different sets 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 the 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. Performing some calculations
3. Executing a set of statements

2. Functions – In TDL

In TDL, prior to Tally.ERP 9, Functions were defined by the Platform and the TDL programmer could only call the function to achieve a certain functionality. From Tally.ERP 9 onwards, functions can be defined in the TDL layer. User Defined Function in TDL has been provided as a Definition which allows the user to specify a set of actions/statements to be executed, in the order as specified.

Traditionally, TDL was always a non-procedural, action-driven language. The sequence of execution was not in the hands of the programmer. But, with this development in a 'Function' Definition, Conditional evaluation of statements and looping has been made possible. User defined Functions basically can be used for performing complex calculations or executing a set of actions serially. Functions can accept parameter(s) as input, and return a 'Value' to the caller.

Functions give the following benefits to the TDL programmer:

- ❑ Conditional execution/evaluation of statements
- ❑ Execution of a set of statements repeatedly, generally referred to as loops
- ❑ Defining variables and storing values from intermediate calculation/process
- ❑ Accepting parameters from the calling segment of code
- ❑ Working on data elements like getting an object from the calling context, defining the function execution context, looping on the objects of a collection, etc.
- ❑ Returning a 'Value' to the caller of the function
- ❑ Performing 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 the following purposes:

Parameter specification

This implies specifying the list of parameters which are 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 sent by the caller of the Function.

<Data Type of the Variable> is the Data type of the Variable sent by the caller of the Function.

Example:

```

[Function : FactorialOf]

    Parameter : InputNumber : Number

```

The Function '\$\$FactorialOf' receives number as the parameter from the Caller.

Variable declaration

If a Function requires some Variable(s) for intermediate calculation, then those Variable(s) need to be defined. The scope of these Variable(s) will be within the Function, and the Variable(s) lose their value after exit from 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 in-line variables). If data type is not specified, the interpreter will look for a variable definition, with the name specified.

Example:

```
[Function : FactorialOf]

    Parameter : InputNumber : Number

    Variable : Counter      : Number

    Variable : Factorial    : Number
```

The Function '\$\$FactorialOf' requires intermediate Variables 'Counter' and 'Factorial' for calculation within the 'Function' Definition.

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 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 in-line variables). If data type is not specified, the interpreter will look for a variable definition with the name specified.

Example:

```
[Function : Sample Function]

    Static Variable : Sample Static Var : Number
```

The static variable 'Sample Static Var' retains the value between successive calls to the Function 'Sample Function'

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:

```
[Function : FactorialOf]
    Parameter : InputNumber : Number
    Returns   : Number
    Variable  : Factorial   : Number
```

The Function '\$\$FactorialOf' returns the value of type 'Number' to the caller of the Function

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:

```
[Function : Sample Function]
    Object : Ledger : "Party"
```

The Function 'Sample Function' will be in the context of the Ledger 'Party'

3.2 Procedural Block

This block contains a set of statements. These statements can either be a programming construct or 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:

```
[Function : DispStockSummary]
    01 : Display : Stock Summary
    02 : Display : Stock Category Summary
```

The Function 'DispStockSummary' is having two Actions with Label.

4. Valid Statements inside a Function

The statements used inside the procedural block of a function can be:

- A Programming Construct, as discussed in the previous sections
- 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.

4.1 Programming Constructs In Function

Conditional Constructs

IF-ENDIF

The 'IF-ENDIF' statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. Initially, the expression will be evaluated and based on the whether the expression is True or False, it transfers the execution flow to a particular statement.

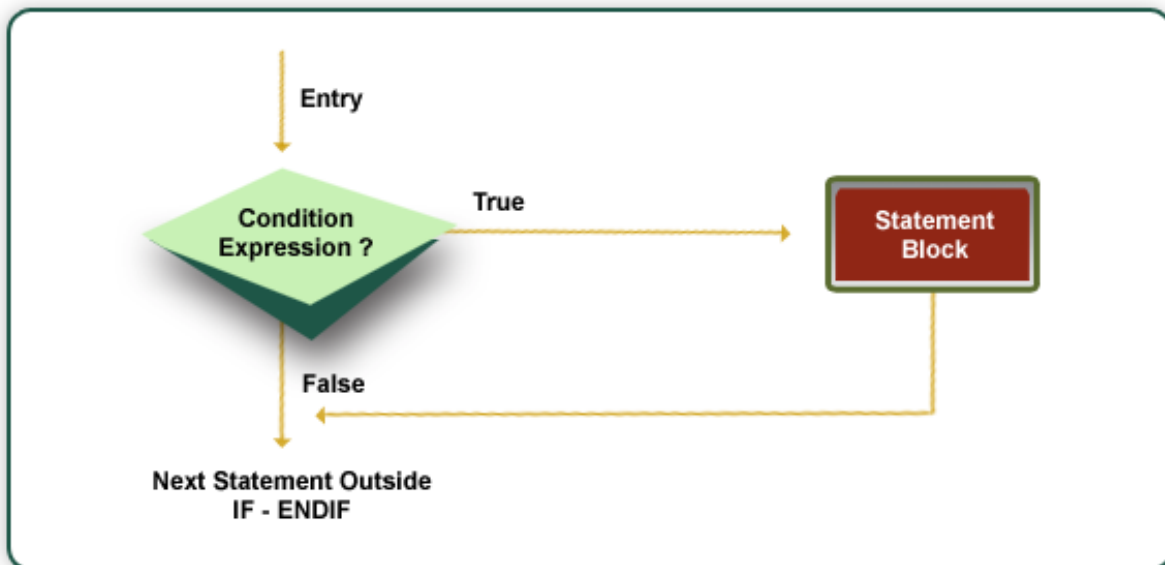


Figure 13.1 Flow Chart for IF – ENDIF

Syntax

```
IF : <Condition Expression>  
    STATEMENT 1  
    ...  
    STATEMENT N  
ENDIF
```

Example:

If the Function parameter sent to the Function 'FactorialOf' is less than zero, then it is multiplied by -1 to find the absolute value.

```
[Function : FactorialOf]

Parameter : InputNumber : Number

Returns   : Number

Variable  : Counter      : Number

Variable  : Factorial    : Number

1 : SET      : Counter      : 1

2 : SET      : Factorial    : 1

3 : IF ##InputNumber < 0

4 : SET      : InputNumber  : ##InputNumber * -1

5 : END IF

6 : WHILE : ##Counter <= ##InputNumber

7 : SET      : Factorial    : ##Factorial * ##Counter

8 : SET      : Counter      : ##Counter + 1

9 : END WHILE

10 : RETURN ##Factorial
```

DO-IF

When an IF-ENDIF statement block contains only one statement, then the same can be written in a single line by using DO-IF statement.

Syntax

```
DO IF : <Condition Expression> : STATEMENT
```

Example:

Here, IF - END IF statement is rewritten using the DO - IF statement.

```
[Function : FactorialOf]

Parameter : InputNumber : Number

Returns   : Number

Variable  : Counter: Number

Variable  : Factorial : Number

1 : SET      : Counter      : 1
```

```
2 : SET      : Factorial : 1
3 : DO IF : ##InputNumber < 0 : ##InputNumber * -1
4 : WHILE : ##Counter <= ##InputNumber
5 : SET      : Factorial : ##Factorial * ##Counter
6 : SET      : Counter   : ##Counter + 1
7 : END WHILE
8 : RETURN ##Factorial
```

IF-ELSE-ENDIF

The IF-ELSE-ENDIF statement is an extension of the simple IF-ENDIF statement. If the condition expression is True, then the 'True' block's statement(s) are executed; otherwise, the 'False' block's statement(s) are executed. In either case, either the True block or the False block will be executed, not both.

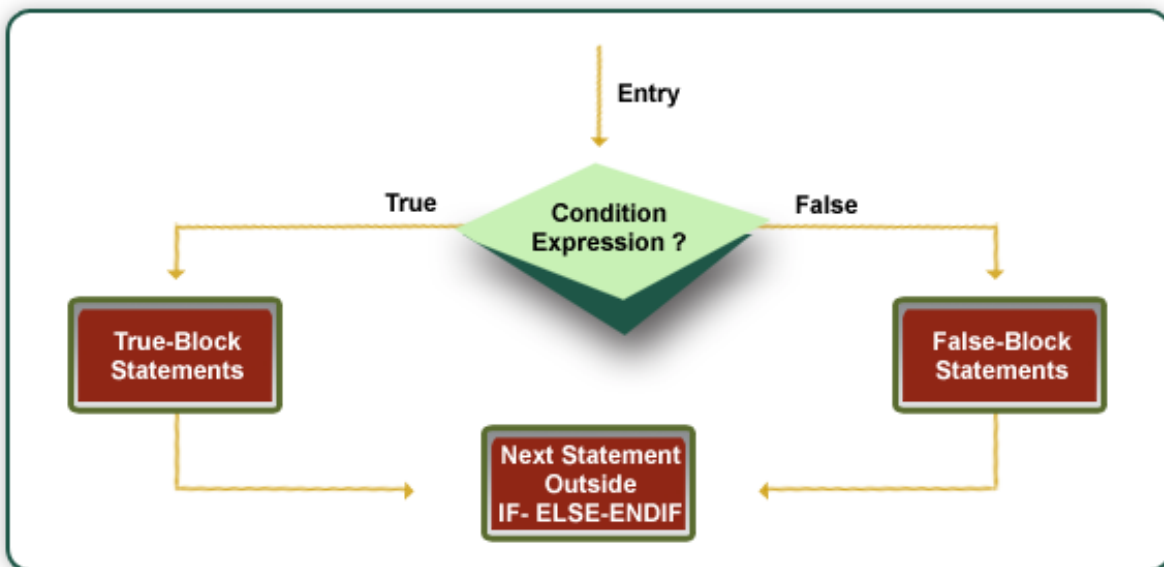


Figure 13.2 Flow Chart for IF – ELSE - ENDIF

Syntax

```
IF : <Condition Expression>
    STATEMENT 1
    ...
    STATEMENT N
ELSE
    STATEMENT 1
```

```
...  
STATEMENT N  
ENDIF
```

Example:

Finding the 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
```

Looping Constructs

WHILE – ENDWHILE

In looping, a sequence of statements is executed until some condition(s) for termination of the loop is 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 the 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.

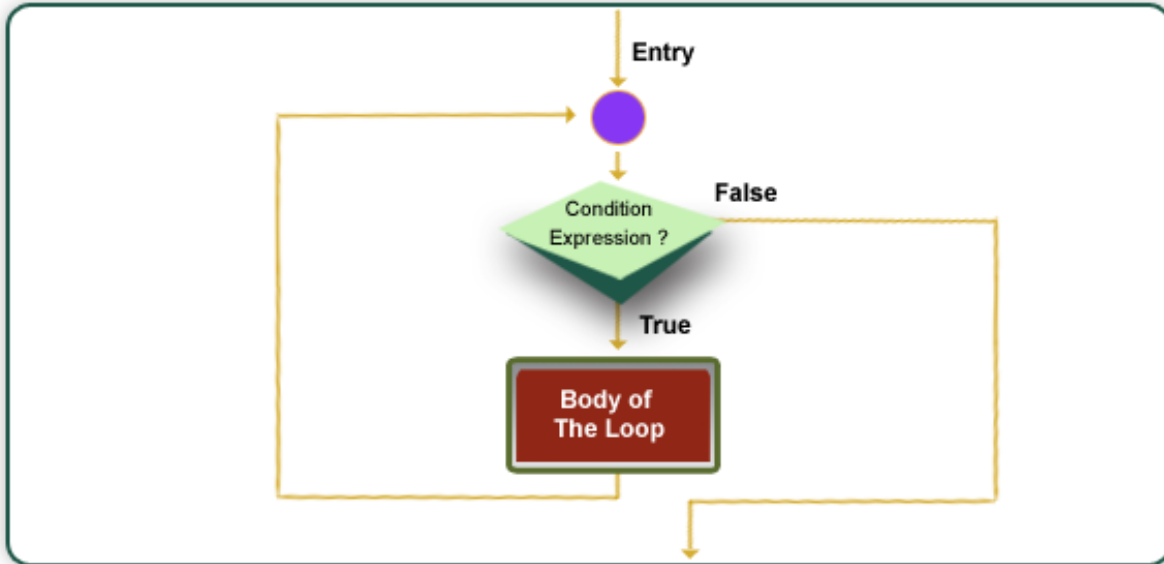


Figure 13.3 Flow Chart for WHILE – ENDWHILE

Syntax

```

WHILE : <Condition Expression>
        STATEMENT 1
        ...
        STATEMENT N
ENDWHILE
  
```

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 : END WHILE

7 : RETURN ##Factorial

The Function 'FactorialOf' repeats statements 4 and 5, till the given condition is satisfied.

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.

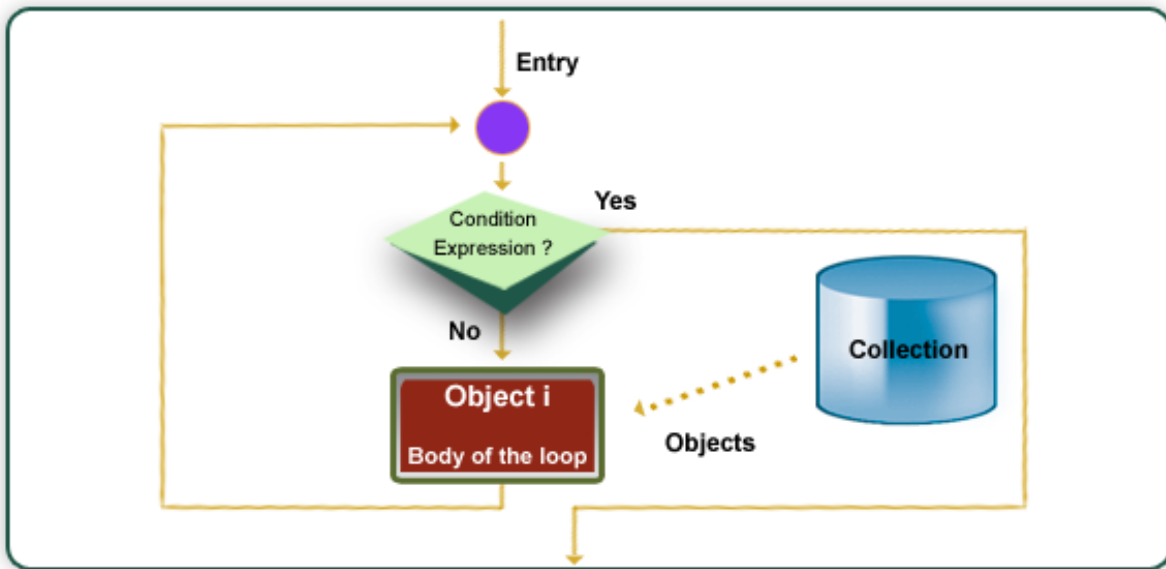


Figure 13.4 Flow Chart for WALK COLLECTION – ENDWALK

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
```

Control Constructs

Loops perform a set of operations repeatedly until the condition expression satisfies the given condition or the Collection is exhausted. Sometimes, when executing the loop, it becomes desirable to skip the part of the loop or to exit the loop as a certain condition occurs, or to save the current state and return back to the current state later.

BREAK

When a BREAK statement is encountered inside the loop, the loop is immediately exited and the control is transferred to the statement immediately following the loop. BREAK statement can be used inside WHILE – END WHILE and WALK COLLECION – END WALK. When loops are nested, the Break would only exit from the loop containing it.

Syntax

BREAK

Example:

```
[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
```

In the Function 'PrintNumbers', the loop is running from 1 to 10. But because of BREAK statement, the loop will be terminated as the counter reaches 6.

CONTINUE

In some scenarios, instead of terminating the loop, the 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 WHILE – END WHILE and WALK COLLECION – END WALK.

Syntax

CONTINUE

Example:

Function to Count the total number of Journal Vouchers

```
[Collection : Vouchers Coll]
    Type      : Voucher
[Function : CountJournal]
    Returns   : Number
    Variable  : Count : Number
    01 : SET : Count : 0
    02 : WALK COLLECTION : Vouchers Coll
    03 : IF : NOT $$IsJournal : $VoucherTypeName
    04 : CONTINUE
    05 : ENDIF
    06 : Count : ##Count + 1
    07 : END WALK
    08 : RETURN : ##Count
```

START BLOCK – END BLOCK

START BLOCK -- END BLOCK has been introduced to save the current state and execute some actions within the block and return back to the original state. This is handy in cases where the Object context needs to be temporarily switched for the purpose of executing some actions. Current source and target object contexts are saved and the further statements within the START BLOCK and END BLOCK section get executed. Once the END BLOCK is encountered, the Object context is restored back to the original state.

Syntax

START BLOCK

Block Statements

END BLOCK

Example:

```

10 : WALK COLLECTION : EDCollDetailsExtract
11 : INSERT COLLECTION OBJECT : InventoryEntries
12 : SET : QtyVar : $$String:$Qty + " Nos"
13 : SET : AmtVar : $$String : $Amt
14 : START BLOCK
15 :           SET OBJECT
16 :           SET VALUE : ActualQty : $$AsQty      : ##QtyVar
17 :           SET VALUE : BilledQty : $$AsQty      : ##QtyVar
18 :           SET VALUE : Amount      : $$AsAmount : ##AmtVar
18A : END BLOCK
19 : SET TARGET : ...
20 : SET VALUE : StockItemName : $Item
21 : END WALK

```

;; For Explanation on Object context, i.e., Source Object and Target Object,

;; Set Target, Set Object, etc., please refer to the Topic 'Function Execution - Object Context'

In this code snippet, the collection 'EDCollDetailsExtract' is being walked over and the values for the Objects within Voucher Entry are being set.

RETURN

This statement is used to return the flow of control to the calling program, with or without returning a value. When RETURN is used, the execution of the function is terminated and the calling program continues from where it had called the function.

Syntax

```
RETURN : <value expression>
```

Where,

<value expression> is optional, i.e., 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
```

4.2 Actions used in a TDL Function

Actions for Variable Manipulation

TDL provides various new actions that can be used inside User Defined Functions.

SET

This action is used to assign a value for a variable.

Syntax

```
SET : <VariableName> : <Value Expression>
```

Where,

<Variable Name> is the variable for which the value needs to be assigned.

<Value Expression> is the formula evaluating to a value.

Example:

```
[Function : FactorialOf]
```

```
Parameter : InputNumber : Number Returns : Number
Variable   : Counter     : Number
Variable   : Factorial   : Number
1 : SET      : Counter: 1
2 : SET      : Factorial: 1
3 : WHILE    : ##Counter <= ##InputNumber
4 :          SET : Factorial : ##Factorial * ##Counter
5 :          SET : Counter   : ##Counter + 1
6 : ENDWHILE
7 : RETURN   : ##Factorial
```

EXCHANGE

This action is used to exchange (swap) the values of two variables. But, values of only the 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 need to be swapped.

Example:

```
01 : EXCHANGE : Var1 : Var2
```

In this statement, both variables are of type 'Number' and their values are swapped.

INCREMENT

This action is used to increment the value of variables by any numerical value. INCREMENT is used inside the loop to increment value of control variables.

You can specify multiple variables by separating them with a comma.

Syntax

```
INCREMENT : <Variable Name> [,<Variable Name>, ...] [:<Number  
Expression>]
```

Where,

<Variable Name> is the name of the variable whose value needs to be incremented.

<Number Expression> is the numerical value by which the variable needs to be incremented. If no number is given, the variable will be incremented by 1.

Example:

```
[Function: VATVchrColWalk_Inv_GUJ_AnnxError_LoadVAR]
```

```
If : ##GUJeVATAnex201BError
```

```
Increment : vVATAnxTotalCount, vVATAnxErrorCount, vVATAnx201BTotalCount,  
vVATAnx201BErrorCount
```

```
Else
```

```
Increment : vVATAnxTotalCount, vVATAnx201BTotalCount
```

```
End If
```



The keyword **Last** will only work from release 5.4.8.

DECREMENT

This action is used to decrement the value of variables by any numerical value. DECREMENT is used inside the loop to decrement the value of the control variables.

Syntax

```
DECREMENT : <Variable Name> [,<Variable Name>, ...] [:<Number Expression>]
```

Where,

<Variable Name> is the name of the variable whose value needs to be decremented by the number specified.

<Number Expression> is the numerical value by which the variable needs to be decremented. If no number is given, the variable will be decremented by 1.

Example:

```
[Function : PrintNumbers]

Variable : Counter : Number

1 : SET      : Counter : 10

2 : WHILE   : ##Counter > 0

3 : LOG      : ##Counter

4 : DECREMENT : Counter

5 : ENDWHILE

6 : RETURN
```

In the function 'PrintNumbers', the loop is running from 10 to 1.

Action Enhancements and New Actions

The global actions have been enhanced, so that they can be called from the User Defined Functions. Some new actions have also been introduced.

Global Actions- Alter/Execute/Create

These are global actions meant for the purpose of opening a report in the mode specified. The return value of these actions is FALSE, if the user rejects the report, and TRUE, if the user accepts it.

Syntax

```
Display/Alter/Execute/Create : Report Name
```

Example:

```
[Function : CreateReport]

01 : Create : Ledger
```

The Function 'CreateReport' opens the Ledger Creation screen.

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:

```
[Function : DispPaySheet]

01 : Menu      : Statements of Payroll

02 : Display   : Pay Sheet
```

The Function 'DispPaySheet' opens the Pay Sheet and by pressing Escape, it will pop up the 'Statements of Payroll' Menu.

MSG BOX

This action is used to Display a message box to the user. It comes back to the original screen on the pressing 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 that 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 is to be greyed out during message display. It takes two values, i.e., YES/NO

Example:

```
01 : MSGBOX : "Return Value" : ##Factorial
```

QUERY BOX

This action is used to Display a confirmation box to the user and ask for a 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> indicates if the background window is to be greyed out during message display. It takes two values, i.e., YES/NO

<Escape as Yes> is a flag which indicates the response when the user presses the 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 pressing of an ESC key.

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> is the expression whose value needs to be passed to the calculator window.

Example:

While finding the factorial of a number, intermediated values are outputted to the 'Calculator' window using LOG action

```
[Function : FactorialOf]
```

```
Parameter : InputNumber : Number
```

```
Returns   : Number
```

```
Variable  : Counter : Number Variable : Factorial : Number
```

```
1 : SET : Counter : 1
```

```
2 : SET : Factorial : 1
```

```
3 : WHILE : ##Counter <= ##InputNumber
```

```
4 :     SET : Factorial : ##Factorial * ##Counter
```

```
5 :     SET : Counter   : ##Counter + 1
```

```
5a :     LOG : ##Factorial
```

```
6 : ENDWHILE
```

```
7 : RETURN : ##Factorial
```

SET LOG ON

While debugging a Function, sometimes it is required to conditionally Log the values of an expression. If logging is stopped, then logging can be re-started based on the condition Action SET LOG ON. This Action does not require any parameter.

Syntax

```
SET LOG ON
```


SET LOG OFF

This Action is used in conjunction with SET LOG ON. Log can be stopped by the Action SET LOG OFF. This Action does not require any parameter.

Syntax

```
SET LOG OFF
```

SET FILE LOG ON

This Action is similar to SET LOG ON. SET FILE LOG ON is used to conditionally Log the values of an expression to log file '**tdlfunc.log**'. This Action does not require any parameter.

Syntax

```
SET FILE LOG ON
```

SET FILE LOG OFF

This Action is used in conjunction with SET FILE LOG ON. Logging the file '**tdlfunc.log**' can be stopped by the Action SET LOG OFF. This Action does not require any parameter.

Syntax

```
SET FILE LOG OFF
```

Progress Bar Actions

Sometimes, a Function may take some time to complete the task. It is always better to indicate the user whether 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 the following Actions:

- START PROGRESS
- SHOW PROGRESS
- END PROGRESS

Action - START PROGRESS

This Action sets up the Progress Bar by mentioning the total number of steps involved in the task. In addition to this, the Title, SubTitle and Subject of the Progress Bar can also be given as parameters.

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 the Progress Bar, respectively.

Example:

```
START PROGRESS : ##TotalSteps : "TDS Migration":+
                @@CmpMailName : "MigrationVouchers.."
```

Action - 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 which denotes the amount of work completed.

Example:

```
SHOW PROGRESS : ##Counter
```

Action - END PROGRESS

When a task is completed, the Progress Bar can be stopped by using the Action END PROGRESS. This Action does not take any parameter.

Syntax

```
END PROGRESS
```

Actions – For Object and Context Manipulation

As already seen in previous sections, functions can operate on 3 object contexts, i.e., Requestor, Current Object and Target object context. When a function is invoked, the target object context will be the same as the current object context of the caller, i.e., the target object will be set to the current object.

Here, we will discuss the various actions for manipulation of Object and Context.

Action - NEW OBJECT

This action creates a new object from object specification and sets it as the target object. It takes only Primary Object as the 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, the target object can be set and saved to the Tally DB.

Action - INSERT COLLECTION OBJECT

This action inserts the new object of the type specified in the collection and makes it as the current target object. This object is inserted into the collection at the end. This Action takes only Secondary Collection as the parameter.

Syntax

```
INSERT COLLECTION OBJECT : <CollectionName>
```

Where,

<CollectionName> is the name of the Secondary Collection.

Example:

```
01 : INSERT COLLECTION OBJECT : Ledger Entries
```

This inserts a new object 'Ledger Entries' in memory under Voucher, and sets it as the target object. Later, by using other methods of this, the target object can be set and saved to Tally DB.

Action - SET VALUE

This action sets the 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. If the second parameter is not specified, it searches for the same method in the context object and the value is set based on it. If the source method name is same as in Target Object, then the Source Object method name is optional.

Example: 1

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

OR

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

These statements set the values of 'Ledger Entries' Object from the current Object context.

Example: 2

```
02 : WALK COLLECTION : Vouchers of My Objects
```

```
03 : NEW OBJECT : Voucher
```

;; Since the methods Date, VoucherTypeName are same in the source object and target object, they are not specified again as SET VALUE : DATE : \$Date.

```
04 : SET VALUE : Date
```

```
05 : SET VALUE : VoucherTypeName
```

Example: 3

```
[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
```

'Party 1' is a ledger under the Group 'North Debtors' and Party 2 is a Ledger under the Group 'South Debtors'. After the execution of this function, Party 2 will also come under the Group 'South Debtors'.

Action - RESET VALUE

This action 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 this code snippet, RESET VALUE resets the value of the method 'Ledger Name'

Action - CREATE TARGET/ACCEPT CREATE

It accepts the Target Object to the Company Data base, i.e., 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
```

Action - SAVE TARGET/ACCEPT

It accepts the Target Object to the Company Tally DB. If another object exists in the Tally DB with the same identifier, then the object is altered, else a new object is created.

Syntax

```
SAVE TARGET/ACCEPT
```

Action - ALTER TARGET/ACCEPT ALTER

It allows altering an existing object in the Tally DataBase. If the object does not exist, it results in an error.

Syntax

```
ALTER TARGET/ACCEPT ALTER
```

Action - SET OBJECT

It 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 gives the name of the Ledger. By Using 'SET OBJECT', the current Object is changed to first Bill allocation. So, the second \$Name is giving the Bill name. The Fourth line changes the current Object back to 'Ledger' using dotted notation.

Action - SET TARGET

This action sets the target object with the Object Specification. If no object specification is given, then 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 'Group' Object as the Target Object. Later, by using other methods of this, the target object can be set and saved to the Tally DB.

Usage of Object manipulation Actions:

Duplicating all payment Vouchers

```
[Function : DuplicatePaymentVouchers]
```

```
;;Process for each Payment Voucher
```

```
01 : WALK COLLECTION : My Vouchers
```

```
;; Create new Voucher Object as Target Object
```

```
02 : NEW OBJECT      : Voucher
```

```
;;For New Object, set methods from the First Object of the Walk Collection, i.e., from the 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 make it the 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 specification
```

```
10 : SET TARGET : ..
```

```
11 : END WALK
```

```
;;Save the Duplicated Voucher to the DB.
```

```
12 : CREATE TARGET
```

```
13 : ENDWALK
```

```
14 : RETURN
```

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 a return value is expected from the function execution. This value is used in the value expression of the calling program.

5.1 Using the Action ‘CALL’

Action ‘CALL’ can be used to call a function. It can be invoked from a Key, Menu Item or a Button.

Syntax

```
CALL: <Function Name> [: <Parameter List>]
```

Where,

<Function Name> is the name of a user defined function.

<Parameter List> is the list of parameters accepted by the function.

Example:

Calling the Function as a procedure with CALL action

```
[#Menu : Gateway of Tally]
```

```
Button : Call Function
```

```
[Button : Call Function]
```

```
Key    : Alt + F9
```

```
Title  : "Call Function"
```

```
Call   : DispStaturoryRpts
```

5.2 Using the Symbol Prefix ‘\$\$’

A Function can be called and executed by prefixing it with the symbol ‘\$\$’. This can be used inside a value expression or as a value for the ‘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 list of parameters accepted by the function.



- During Tally Startup, Tally executes a function with the name “TallyMAIN”
- Internal functions always override if both exist in the 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 the platform defined functions will be executed with the 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, i.e., 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 the caller object context and alter the values of the target object with those values.

The User has the 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.

\$\$TgtObject is used to evaluate the value of an expression in the context of the 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 the context of the caller. Parameter specification within the functions is just to determine the data type, order and number of parameters. These are basically the place-holders 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 a value. The data type of the 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 the previous section.

Section III
What's New in TDL

What's New in Release 5.5.2

1. Language Enhancements in Procedures (TDL)

1.1 Action – Browse URL

The action **Browse URL** is enhanced to accept a logical value as an optional fourth parameter to open the URL in **Admin** mode.

Syntax

```
Action: Browse URL: <URL Formula>: [<Command Line Parameters>: +  
                <Logical Expression1>: <Logical Expression2>]
```

Where,

<URL Formula> is an expression which evaluates to a link to a website or a folder path.

<Command Line Parameters> is the list of command line parameters separated by space. It is an optional parameter.

<Logical Expression1> can be any expression which evaluates to a logical value. When this parameter is set to yes, it executes the given action in hidden mode. It is an optional parameter.

<Logical Expression2> can be any expression which evaluates to a logical value. When this parameter is set to yes, it executes the action in admin mode. It is an optional parameter.

1.2 Function – SysInfo

The function **SysInfo** is enhanced to get the file name of the current application with the folder path.

Example

```
$$SysInfo: ModuleName
```

If the application path is **C:\Tally.ERP 9**, it returns **C:\Tally.ERP 9\tally.exe**.

1.3 Attribute – Data Source

The collection attribute **Data Source** is used to populate data to a collection dynamically from a variety of data sources. This attribute accepts the **Type** and **Identity** of the data source from where the data is to be retrieved. **Directory**, **File XML**, **Name Set**, and so on, are some of the data sources supported in TDL.

The collection attribute **Data Source** is enhanced to support **Rule Set**, **Num Set**, and **Flag Set** as new data source types. This attribute helps the developer to gather the data pertaining to **Rule Set**, **Num Set**, or **Flag Set** like other data source types. The information of **Num Set** and **Flag Set** give same results except for the value. The value of **Num Set** is a number and value of **Flag Set** is logical. These data sources can be used for generating reports using **Value** and **User Description** methods. Otherwise these can act as a debugging aid for the developer.

Syntax

```
[Collection: <Collection Name>]
  Data Source: <Type>: <Identity>
```

Where,

<Collection Name> is the name of the collection where the data is populated.

<Type> specifies the type of data source. This enhancement includes **Rule Set**, **Num Set** and **Flag Set** as data source types.

<Identity> is the name of the Rule Set/Num Set/Flag Set. It can be an expression which results to the name of **Num Set/Flag Set** as specified in the corresponding data type.

Example

```
[Collection: DataSourceFSRepo]
  Data Source: Flag Set: #FSDSVariable
```

The above code snippet helps to populate the collection **DataSourceFSRepo** with the data from the variable **FSDSVariable**, which returns a **Flag Set** name.

You can refer the below table to know more about this enhancement:

Data Source Type	Keyword	Parameters	Example
Rule Set	Rule Set	Name of the Rule Set definition	[Collection: DSRepo] Data Source: Rule Set: DSRuleSet
Flag Set	Flag Set	Expression evaluating to the value of Flag Set data type	[Collection: DSRepo] Data Source: Flag Set: \$FlagSetValue OR Data Source:Flag Set: ##FlagSetValue
Num Set	Num Set	Expression evaluating to the value of Num Set data type	[Collection: DSRepo] Data Source: Num Set: \$NumSetValue OR Data Source:Num Set:##NumSetValue

The list of methods used in **Rule Set/Flag Set/Num Set** as data source is listed below:

Method Name	Description	Method Value - Sample
Name	Rule Name which includes the complete dotted notation	IsOpBalGT1000.IsPANEmpty.IsContactEmpty
Additional Name/Formal Name	Leaf Rule Name	IsContactEmpty
Parent	Current Rule Set Name	Dimension2
Value	Value for individual flag or number i.e., Logical for Flag Set and Number for Num Set. It is not applicable for Rule Set.	Logical - Yes Number - 1
Sort Position	Index of the flag or number within the Flag Set and Num Set	5
Is Aggregate	IsAggregate flag for Rule Set, Num Set and Flag Set	No
Depth	Level or Depth of Rule Set, Num Set and Flag Set	3
User Description	Rule Set, Num Set and Flag Set description from corresponding Name Set	IsPartyContactEmpty

What's New in Release 5.4.9

In TDL, to make the programming simpler and to optimise the reporting and query area, we have introduced two definitions - **Rule Set** and **Name Set**, and two data types - **Flag Set** and **Num Set**.

1. Definition – Rule Set

Rule Set is a definition in TDL and can be used to define a set of rules. You can use this definition to evaluate the rules efficiently. Rule Set can be defined with single dimension or multiple dimensions along with flow control. The attribute **Rule Set** and related functions help to evaluate the defined Rule Sets. The result of a Rule Set is always an Object method of Flag Set data type. The functions related to Flag Set data type allow you to access the Flag Set. Storing the Flag Set result in data base is currently not supported. For example, during voucher acceptance, the Rule Set can be used to check the correctness of data.

The definition **Rule Set** makes TDL programming simpler because of its ability to allow a developer to scope only one validation at a time and not all of them together. It reduces the code size. It also provides the named identity to access flag set values. **Rule Set** increases the efficiency of the programme, which is measured on performance and the memory usage.

Syntax

```
[Rule Set: <Rule Set Name>]
```

Where,

<Rule Set Name> is the name of the Rule Set.

Various attributes and functions, which are related to Rule Set are explained below:

1.1 Attribute – Break On

Break On is a single type attribute with constant value. You can use the attribute **Break On** in Rule Set definition to determine the continuation of evaluation in the current rule set. It provides vertical flow control to evaluate the rules. Break on can also be mentioned with Rule attribute.

Syntax

```
Break On: <Logical Constant>
```

Where, <Logical Constant> is Yes/No.

While evaluating each rule,

- If the value of **Break On** matches with the result of the rule, then the evaluation flow will break and further rules are not evaluated in the current Rule Set. The remaining rule values are defaulted to **No**.
- If the attribute **Break On** is not specified, the evaluation flow continues, irrespective of the result of rule.

1.2 Attribute – Walk On

Walk On is a single type attribute with constant value. The attribute **Walk On** helps to determine whether the next dimension will be evaluated or not, for the current rule. It has diagonal flow control to evaluate the steps.

Syntax

Walk On: <Logical Constant>

Where, <Logical Constant> is Yes/No.

While evaluating each rule,

- If the value of **Walk On** value matches with the result of the rule, then the evaluation flow continues to evaluate the next dimension in the current rule context. If the next dimensions are not evaluated, the rule values are defaulted to **No**.
- If the attribute **Walk On** is not specified, the evaluation flow continues to evaluate the next dimension, irrespective of the result of rule.

1.3 Attribute – Rule

Rule is a Triple list attribute. You can use the attribute **Rule** to define a rule. It accepts three parameters.

Syntax

```
[Rule Set: <Rule Set Name>]
    Rule : <Rule Identifier>: <Logical Expression>[: <Keyword>]
```

Where,

<Rule Identifier> is the name of the rule to identify or access the rule later. "*" and "?" are not supported as rule names.

<Logical Expression> evaluates in the context of requester/source/target object being passed from evaluation context.

<Keyword> specifies the value of **Break On** sub-attribute for current rule. If no specification is provided, it inherits the **Break On** attribute value at current rule set level.

Possible options are

True/Yes: When the result of the rule is True/Yes, it stops the evaluation further in the current rule set. The next dimension gets evaluated for the current rule.

False/No: When the result of the rule is False/No, it stops evaluating further.

Never: It continues the evaluation, irrespective of rule result.

Example

```
[Rule Set: My Rule Set]
;; Yes/No or Not Specified
    Break On      : Yes

    Rule          : A1      : @@TNAnnexure1Included

    Rule          : A2      : @@TNAnnexure2Included: No
```

In the above example, **A1**, **A2** are rule identifiers. **@@TNAnnexure1Included** and **@@TNAnnexure2Included** are rule expressions.

1.4 Attribute – Aggr Rule

Aggr Rule is a Triple list type attribute and it accepts constant values as parameters. You can use **Aggr Rule** to specify various pre-defined aggregation for Rule Set.

Syntax

```
Aggr Rule: <Identifier>: <Type of aggregation>[: <Level Indicator>]
```

Where,

<Identifier> is to specify the identifier for the aggregate rule. It can be accessed from Flag Set like any other flags.

<Type of aggregation> is to specify the type of aggregation. (Any True / OR, All True / AND)

<Level Indicator> is to indicate the level for aggregation. It is always relative to current rule set. The default value of this parameter is One. This parameter to be specified by using a level number from current rule set to identify the level.

The aggregate flags are accessed using the same dotted notation like any other flags. Value of these rule flags are aggregation results, which are computed at the time of rule evaluation.

When multiple flag sets are aggregated, similar to **Aggr Compute: SUM** in collection, these aggregate flags are also further aggregated (counts) into Num Sets.

The first **Aggr Rule** result of the **root/top level Rule Set** is stored into the Zeroth flag. If no aggregate rule is specified in the hierarchy, the first rule result is stored at Zeroth flag.

Zeroth flag value is also referred as the **master aggregate rule value**.

1.5 Attribute – Rule Set

Rule Set is a Triple list attribute. You can use the attribute **Rule Set** to specify the rule set for next dimension of each rules in the current rule set. Rule Set attribute accepts three parameters.

Syntax

```
Rule Set: <Rule Set Name>: [<CurrentDefinitionRules> +  
: <NextDimensionRules>]
```

Where,

<Rule Set Name> is the name of the next dimension Rule Set. It is the Rule Set description reference to another Rule Set definition.

<CurrentDefinitionRules> are the rules in the current rule set and it has the given Rule Set as next dimension. If it is not specified, it assumes all rules in the current rule set have the given rule set as next dimension.

<NextDimensionRules> are the rules in next dimension Rule Set that have to be evaluated. If not specified, it assumes all rules in next dimension rule set are to be evaluated. The results are available as part of the Flag Set generated by Rule Set evaluation.

The attribute **Rule Set** accepts multiple values. Each rule can have a different Rule Set for its next dimension.

A Rule Set already present in the hierarchy at any level is not allowed to be specified as the next dimension again.

1.6 Attribute – Name Set

Name Set is a Single type attribute. You can use the attribute **Name Set** to specify the default name set definition for the current rule set. In general, Name Set contains rule descriptions.

Syntax

Name Set: <Name Set Definition Name>

Where,

<**Name Set Definition Name**> is the name of Name Set definition to map the current rule set. Name Set can be used in two ways:

1. To specify the default name of the name set definition name, when it is used with the attribute **Name Map**, without using the third parameter of Name Map.
2. Use the attribute Name Set without using Name Map – automatically it will get mapped based on the rules in the current rule set, if the identifiers in Name Set match the rule identifier.

1.7 Attribute – Name Map

Name Map is a triple list type attribute. You can use the attribute **Name Map** to specify explicit name mapping for Rule descriptions. It accepts three parameters.

Syntax

Name Map: <Rule Name>:<Name Identifier>[:<Name Set Definition Name>]

Where,

<**Rule Name**> is an identifier to Rule in current rule set. This name is defined in Rule attribute.

<**Name Identifier**> is specified in the Name Set definition.

<**Name Set Definition Name**> is an optional parameter and it is the name of Name Set definition. If this sub-attribute is specified, then the string value is accessed from Name Set which is mentioned here. If it is not mentioned it assumes the Name Set Definition Name mentioned in Name Set attribute in the current Rule Set.

Either the third parameter of Name Map or a Name Set attribute should be specified. Otherwise the specific name map will be ignored.

Name Map is useful to map the error strings inline with each Rule Set.

1.8 Function – EvaluateRuleSet

The function **EvaluateRuleSet** evaluates the Rule Set and returns a **Flag Set** data type. It returns a **Flag Set** type, which is constructed with flags and dimensions as per the **Rule Set** definition provided, and evaluated on the current contexts.

Syntax

`$$EvaluateRuleSet:<Rule Set Name>`

Where,

<Rule Set Name> is the name of the **Rule Set** definition. It returns **Flag Set** data type.

Static modifiers like Add, Change, and Delete are supported in rule set definition.

Local formula is supported with the definition rule set. The local formula can be accessed only from the Rule Set in which the local formula is defined.

Example – Rule Set

```
[Rule Set: My Rule Set]

;; Yes/No or Not Specified
Break On      : Yes

;;RuleName    ;; Rule Logical Expression ;; break on (override)
Rule          : A1          : @@TNAnnexure1Included
Rule          : A2          : @@TNAnnexure2Included      : No
Rule          : A3          : @@TNAnnexure3Included

...
;; Second Dimension RuleSet  ;; My Rules      ;; Dimension Rules (Filter)
RuleSet        : Anx_ErrorsA      : A1          : EA1, EA2
RuleSet        : Anx_ErrorsB      : A2, A3       : EB2, EB3

;; Aggregate Rules
Aggr Rule     : Is Included In Any Annexure: IsAnyTrue : 1
Aggr Rule     : Has Any Error : IsAnyTrue : 2

;; option1 Name Set mapping
[Rule Set: Anx_ErrorsA]

Break On      : Yes

;; ;;RuleName    ;; Rule Logical Expression ;;break/continue condition
Rule          : EA1          : @@TN_Error1
Rule          : EA2          : @@TN_Error2      : No
Rule          : EA3          : @@TN_Error3
```

```
[Rule Set: Anx_ErrorsB]

  Break On      : Yes

;; ;;RuleName ;; Rule Logical Expression ;; break/continue condition
  Rule          : EB1           : @@TN_Error4

  Rule          : EB2           : @@TN_Error5           : No

  Rule          : EB3           : @@TN_Error6
```

In the above example, **My Rule Set** is the name of the Rule Set. **A1**, **A2**, and **A3** are three different rules defined under **My Rule Set**. The rule **A2** has **Break On** override by setting its third parameter as **No**. In this case rule **A2** works based on the local value of **Break On**. **@@TNAnnexure1Included**, **@@TNAnnexure2Included**, **@@TNAnnexure3Included** are logical expressions and defined for **A1**, **A2**, **A3** respectively.

Anx_ErrorsA, **Anx_ErrorsB** are second dimension rule sets. Using these rule sets, it filters the first dimension rules based on the requirement. The rules **EA1** and **EA2** are second dimension rules and these rules are applicable only for the Rule **A1**. Similarly, **EB2** and **EB3** rules are applicable only for **A2** and **A3**.

Is Included In Any Annexure and **Has Any Error** are two aggregate rules defined to pre-compute the aggregation for the Rule Set **MyRuleSet** by identifying the level. The aggregate rule **Is Included In Any Annexure** evaluates in the first level whereas the aggregate rule **Has Any Error** evaluates in the second level.

2. Definition – Name Set

Name Set is a definition to ensure that strings are always segregated and separated from the source code expressions. It helps to define a set of Name-Value pairs. It is a convenient way to define static string data, and access the strings easily.

You can use Name Sets to define application string groups like – error strings for a given rule, static application data like list of states, list of countries, and so on. Name Set also supports being specified as the data source in a collection. It will result in a collection where each object represents a Name-Value pair in the Name Set.

You need to define all strings of the application together using Name Sets to make the corrections of strings easy. In other words, managing strings is made easier by using Name Set.

Syntax

```
[Name Set: <Name Set Name>]
```

Where,

<Name Set Name> is the name of the Name Set.

Various attributes and functions in Name Set are explained below:

2.1 Attribute – List Name/List

List Name is a Dual list type attribute with a constant name and value. You can use the attribute **List Name** to specify the identifier and the string. The identifier can be used to access the value stored in the list.

Syntax

```
List Name      : <string identifier 1> : <string 1>
List Name      : <String identifier 2> : <string 2>
```

Where,

<string identifier 1>, <string identifier 2> are user defined identifiers for the strings <string 1>, <string 2>.

2.2 Function – NameGetValue

Function **NameGetValue** returns a string in the Name Set based on the values passed. The function **NameGetValue** takes two parameters, **String identifier** and **Name Set Description Name**.

Syntax

```
$$NameGetValue:<String Identifier>:<Name Set Description Name>
```

Where,

<String Identifier> is the list name identifier.

<Name Set Description Name> is the name of the Name Set.

Example

```
[Nameset : AnnexureErrorStrings]

List Name      : E1      : "PAN Number Invalid"
List Name      : E2      : "Lorry Number Missing"
List Name      : E3      : "Commodity Code is Empty"

[Function NameSetTestFunction]

01 : LOG : $$NameGetValue:E2:AnnexureErrorStrings
;; This will print "Lorry Number Missing"
[Collection: My Collection]

Data Source : Name Set : AnnexureErrorStrings
```

The collection **My Collection** delivers 3 objects, as given below:

- \$Name: E1, \$Value = "PAN Number Invalid"
- \$Name: E2, \$Value = "Lorry Number Missing"
- \$Name: E3, \$Value = "Commodity Code is Empty"

3. Data Type – Flag Set

Flag Set data type stores a set of logical values/flags. It takes less memory to store flag set values compared to the memory takes to store individual logical values. It allows to perform operations on the stored values. This is a variable length data type and hence has no restriction on the number of flags. Currently, Flag Set is generated as a result of Rule Set evaluation. You can use **Flag Set** data type in Variables and Objects.

This data type is not directly input-able by the user and not directly displayable to the user.

The following **Flag Set** functions help you to perform the operations on **Flag Set** data type.

3.1 Function – FlagGetValue

You can use the function **FlagGetValue** to get the value of a flag inside a Flag Set. It returns a logical value.

Syntax

```
$$FlagGetValue:<Rule Identifier>:<Flag Set Expression>
```

Where,

<**Rule Identifier**> is a constant identifier or an expression that results in a rule identifier which is in dotted notation.

<**Flag Set Expression**> evaluates to a **Flag Set** value.

Example

```
$$FlagGetValue:A1.B2:$MyFlagSet
```

For flag names, a dotted notation must be used to denote fully qualified name of a flag. It shows the path of rule evaluation.

When "*" is passed as the Rule Identifier, it returns the master aggregate value (Zeroth flag value) of the Rule Set.

3.2 Function – FlagSetOR

The function **FlagSetOR** takes two flag sets as input and it performs the OR operation on them. It returns a **Flag Set** data type. Both flag sets must be obtained by evaluating the same rule set (potentially with different objects), otherwise this function fails.

Syntax

```
$$FlagSetOR:<Flag Set 1>:<Flag Set 2>
```

Where,

<Flag Set 1> is the first Flag Set expression.

<Flag Set 2> is the second Flag Set expression.

Example

```
$$FlagSetOR:$FlagSetA:$FlagSetB
```

3.3 Function – FlagSetAND

The function **FlagSetAND** takes two Flag Sets as input and it performs AND operation. It returns **Flag Set** data type. Both flag sets must be obtained by evaluating the same rule set (potentially with different objects), otherwise this function fails.

Syntax

```
$$FlagSetAND:<Flag Set 1>:<Flag Set 2>
```

Where,

<Flag Set 1> is the first Flag Set expression.

<Flag Set 2> is the second Flag Set expression.

Example

```
$$FlagSetAND:$FlagSetA:$FlagSetB
```

3.4 Function – FlagsIsAllTrue

The function **FlagsIsAllTrue** returns true, if all child rules under given rule are **True** (AND operator). It returns a logical type.

Syntax

```
$$FlagsIsAllTrue:<Parent Rule Identifier>:<Flag Set Expression> +  
[:<Belongs To>]
```

Where,

<Parent Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. This is used as parent flag.

<Flag Set Expression> is the expression which results in a Flag Set on which this operation is to be performed.

<Belongs To> is the logical value to decide whether only immediate children, or all children in the hierarchy of the given parent rule, to be considered. If this parameter is not specified, the default value is **No**.

Example

```
$$FlagsIsAllTrue:A1:$MyFlagSet:Yes
```


”*” can be passed as Rule Identifier to perform the operation from the root rule set level.

If the parent rule does not have any child rules, then the function returns **No**.

3.5 Function – FlagsIsAllTrueFromLevel

The function **FlagsIsAllTrueFromLevel** returns **True**, if all flags are True at a given level. It returns a logical value.

Syntax

```
$$FlagsIsAllTrueFromLevel:<Peer Rule Identifier>:<Flag Set Expression> +
    [:<Leaf Rule Filter>]
```

Where,

<Peer Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. It is used to specify the level and the starting point of the aggregation/operation/walk.

<Flag Set Expression> is the expression which results in a Flag Set on which this operation is to be performed.

<Leaf Rule Filter> is the Rule Name to filter the non-qualified leaf flag. If specified, only rules ending with this Rule Name will be considered for the aggregation. If not specified, all rules at the specified level will be considered.

Example

```
$$FlagIsAllTrueFromLevel:A1.B1:$MyFlagSet:B1
```

In the above example, the rule **B1** is the Leaf Rule Filter. **A1.B1** denotes the second level and the starting point for aggregation. Since **B1** is the Leaf Rule Filter, it considered **A1.B1** and other rules which ends with **B1**. The rules such as A1.B2 will not be considered.

“*” is not applicable for this function.

It walks all flags in the same level as that of the Rule Identifier specified, starting from the given rule identifier. To consider all rules at a certain level, specify the identifier of the first rule in that level.

3.6 Function – FlagsIsAnyTrue

The function **FlagsIsAnyTrue** returns **True**, if any child rule of a given flag is **True** (OR operator). It returns a logical value.

Syntax

```
$$FlagsIsAnyTrue:<Parent Rule Identifier>:<Flag Set Expression> +
    [:<Belongs To>]
```

Where,

<Parent Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. This is used as parent flag.

<**Flag Set Expression**> is the expression which results in a flag set on which this operation is to be performed.

<**Belongs To**> is the logical value to decide whether only immediate children, or all children in the hierarchy of the given parent rule, to be considered. If this parameter is not specified, the default value is **No**.

Example

```
$$FlagsIsAnyTrue:A1:$MyFlagSet:Yes
```

“*” is used to perform the operation from the root rule set level.

If the parent rule does not have any child rules, then the function returns **No**.

3.7 Function – FlagsIsAnyTrueFromLevel

The function **FlagsIsAnyTrueFromLevel** returns **True**, if any flags are **True** at a given level. It returns a logical value.

Syntax

```
$$FlagsIsAnyTrueFromLevel:<Peer Rule Identifier>:<Flag Set Expression>+  
:<Leaf Rule Filter>
```

Where,

<**Peer Rule Identifier**> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. It is used to specify the level and the starting point of the aggregation/operation/walk.

<**Flag Set Expression**> is the expression which results in a Flag Set on which this operation to be performed.

<**Leaf Rule Filter**> is the Rule Name to filter the non-qualified leaf flag. If specified, only rules ending with this Rule Name will be considered for the aggregation. If not specified, all rules at the specified level will be considered.

Example

```
$$FlagIsAnyTrueFromLevel:A1.B1:$MyFlagSet:B1
```

In the above example, the rule **B1** is the Leaf Rule Filter. **A1.B1** denotes the second level and the starting point for aggregation. Since **B1** is the Leaf Rule Filter, it considered **A1.B1** and other rules which ends with **B1**. The rules such as A1.B2 will not be considered.

“*” is not applicable for this function.

This function walks all flags in the same level as that of the Rule Identifier specified, starting from the given rule identifier. To consider all rules at a certain level, specify the identifier of the first rule in that level.

3.8 Function – FlagsCount

You can use the function **FlagsCount** to get the number of flags under a given parent rule matching the value passed. It returns a number type.

Syntax

```
$$FlagsCount:<Parent Rule Identifier>:<Flag Set Expression>: +
    [<Flag Value>:<Belongs To>]
```

<Parent Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. This is used as parent flag.

<Flag Set Expression> is the expression which results in a Flag Set on which the operation is to be performed.

<Flag Value> is a logical value. It can be either **True/Yes** or **False/No**. It is used to count flags that match this value. The default value is **True/Yes**.

<Belongs To> is the logical value to decide whether only immediate children, or all children in the hierarchy of the given parent rule, to be considered. If this parameter is not specified, the default value is **No**.

Example

```
$$FlagsCount:A1:$MyFlagSet:Yes:Yes
```

”*” is used to perform the operation from the root rule set level.

If the parent rule does not have any child rules, then the function returns **0**.

3.9 Function – FlagsCountFromLevel

You can use the function **FlagsCountFromLevel** to get the number of flags at the same level, across parents, matching the flag value passed. It returns a number type.

Syntax

```
$$FlagsCountFromLevel:<Peer Rule Identifier>:<Flag Set Expression>: +
    [<Flag Value>:<Leaf Rule Filter>]
```

Where,

<Peer Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. It is used to specify the level and the starting point of the aggregation/operation/walk.

<Flag Set Expression> is the expression which results in a flag set on which this operation is to be performed.

<Flag Value> is the logical value. It can be either **True/Yes** or **False/No**. It is used to count flags that match this value. The default value is **True/Yes**.

<Leaf Rule Filter> is the Rule Name to filter the non-qualified leaf flag. If specified, only rules ending with this Rule Name will be considered for the aggregation. If not specified, all rules at the specified level will be considered.

Example

```
$$FlagCountFromLevel:A1.B1:$MyFlagSet:Yes:B1
```

“*” is not applicable for this function.

If “?” is specified as the leaf rule filter, the leaf rules will be counted only once and duplicates will be ignored.

This function walks all the flags in the same level as that of the rule identifier specified, starting from the given rule identifier. To consider all rules at a certain level, specify the identifier of the first rule in that level.

3.10 Function – FlagGetDescription

You can use the function **FlagGetDescription** to get a **String** description for the given Rule Identifier. It returns a **string** type.

Syntax

```
$$FlagGetDescription:<Rule Identifier>:<Flag Set Expression>
```

<Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation.

<Flag Set Expression> is the expression which results in a flag set. It is used to identify the Rule Set from which the rule description is to be obtained.

Example

```
$$FlagGetDescription:A1:$MyFlagSet
```

“*” is not applicable for this function.

It returns a description string associated with the rule identifier.

3.11 Function – FlagsListDescription

You can use the function **FlagsListDescription** to get the list of string descriptions of all immediate child rules for the parent rule identifier passed, separated by the given character. It returns a **string** type.

Syntax

```
$$FlagsListDescription:<Parent Rule Identifier>:<Flag Set Expression>: +  
[<Flag Value>:<Separator Character>]
```

Where,

<Parent Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. It is the parent rule name.

<Flag Set Expression> is the expression which results in a Flag Set on which the operation is to be performed. It is also used to identify the Rule Set from which the rule descriptions are to be obtained.

<Flag Value> is the logical value. It can be either **True/Yes** or **False/No**. It is used to get descriptions of flags that match this value. The default value is **True/Yes**.

<Separator Character> is a string value to separate string descriptions. If not specified, the default separator is used.

Example

```
$$FlagsListDescription:A1:$MyFlagSet:Yes:", "
```

It considers all immediate child rules which match the flag value passed.

"*" can be passed as the parent rule identifier to get the descriptions from the rules in the first level.

3.12 Function – FlagsListDescriptionFromLevel

You can use the function **FlagsListDescriptionFromLevel** to get a list of unique string descriptions for the set of rules/flags from a given level, across parents that match the flag value passed. It returns a **string** type.

Syntax

```
$$FlagsListDescriptionFromLevel:<Peer Rule Identifier>:+
    <Flag Set Expression>:[<Flag Value>:<Separator Character>]
```

Where,

<Peer Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation. It is used to specify the level and the starting point of the aggregation/operation/walk.

<Flag Set Expression> is the expression which results in a Flag Set on which the operation is to be performed. It is also used to identify the Rule Set from which the rule descriptions are to be obtained.

<Flag Value> is the logical value. It can be either **True/Yes** or **False/No**. It is used to get descriptions of flags that match this value. The default value is **True/Yes**.

<Separator Character> is a string value to separate string descriptions. If not specified, the default separator is used.

Example

```
$$FlagsGetDescriptionListFromLevel:A1.B1:$MyFlagSet:Yes:", "
```

In the above example, all descriptions at the level of rule A1.B1 and starting from the rule A1.B1 will be considered. However, a description will be considered only once. Rule A1.B1 and A2.B1 will refer to the same description, (i.e., description of Rule B1, assuming in both cases B1 is referring to same rule) but the description will be taken only once.

3.13 Function – AsFlagSet

You can use the function **AsFlagSet** to convert a Num Set to a Flag Set which allows the use of all Flag Set functions. It creates an equivalent Flag Set considering all non-zero numbers as **True**, and zero numbers as **False**.

Syntax

```
$$AsFlagSet:<Num Set Expression>
```

Where,

<Num Set Expression> is the expression evaluates to a **Num Set value**.

Example

```
$$AsFlagSet:$MyNumSet
```

This conversion leads to a loss of count information in the resultant flag set. A **True** value in the flag set only represents a non-zero value in the original num set. However, the exact value cannot be obtained.

4. Data Type – Num Set

The data type **Num Set** stores and operates a set of numbers. This is a variable length data type and hence has no restriction on the number of values that it can hold.

This data type is not directly input-able by the user and not directly displayable to the user.

The following **Num Set** functions help you to perform the operations on **Num Set** data type.

4.1 Function – NumGetValue

Use the function **NumGetValue** to get a value inside a Num Set. It returns a number type.

Syntax

```
$$NumGetValue:<Rule Identifier>:<NumSet Expression>
```

Where,

<Rule Identifier> is a constant identifier or an expression that results in a rule identifier which is in dotted notation.

<Num Set Expression> is the expression that evaluates to a Num Set from which the number is to be fetched.

Example

```
$$NumGetValue:A1.B2:$MyNumSet
```

*** helps to extract the master aggregate value (Zeroth num value) of the rule set.

4.2 Function – AsNumSet

You can use the function **AsNumSet** to convert a Flag Set to a Num Set.

Syntax

```
$$AsNumSet:<Flag Set Expression>
```

Where,

<Flag Set Expression> is any expression of Flag Set data type.

Example

```
$$AsNumSet:$MyFlagSet
```

It creates an equivalent **Num Set** by considering all **False** values as **zero**, and **True** as **one**.

The **Num Set** generated by this function will contain only 0s and 1s.

5. Other Enhancements

5.1 Attribute – MAX

You can use the attribute **MAX** at field definition to specify the maximum number of characters that can be entered in the field. It also helps to control the field length with some specific number of characters based on a conditional expression. The expression is evaluated while opening the report.

Alias for this attribute is **Maximum**. This attribute accepts a number which is less than 972.

Syntax

```
MAX : <Numerical Expression/Constant>
```

Where,

<Numerical expression> evaluates to a number.

For example, in Tally.ERP 9, the field **PAN/National Identity Number** accepts only 10 characters, because India restricts the PAN number with 10 characters. However, by using this attribute enhancement, based on the country selected, we can control the length of the field.

Example

```
Max:If $$Number:$OpeningBalance:Ledger:"Cash"< 6 then 9 +
      else $OpeningBalance:Ledger:"Cash"
```

5.2 Function – ValidateTINMod97

You can use the function **ValidateTINMod97** to check if the TIN provided is valid or not. It takes an input string which is the TIN and returns a logical value. It returns **True**, if the input string is a valid TIN.

To check if the given string is a valid TIN, the function performs the following:

1. Extract the digits from the alphanumeric string and check if there are exactly 11 digits.
2. Rotate the number clockwise 4 times, perform MOD operation on the resulting number with 97, and check if the remainder is 1.

The TIN is not valid, if any of the above checks fail.

Syntax

```
$$ValidateTINMod97:<Alphanumeric Value>
```

Where,

<Alphanumeric Value> is the TIN having alphabets as prefix/suffix.

Example

```
[Field: Mod 97]
```

```
Use      : Name Field
```

```
Set as: 27240039198
```

```
Notify: IsValidTIN: Yes
```

```
[System: Formulae]
```

```
ValidateTin : $$ValidateTINMod97:$$Value
```

```
IsValidTIN  : If @@ValidateTIN then "Valid" Else "NOT Valid"
```


What's New in Release 5.4.8

1. Language Enhancements in Primitives (TDL)

1.1 Function - IsAnyEmpty

When you want to check if there is an expression that evaluates to empty among a set of expressions, you can use the function **IsAnyEmpty**. This function evaluates the expression parameters in the sequence specified in the code. It will return **True**, the moment an expression evaluates to empty and ignores the subsequent expressions.

Syntax

```
$$IsAnyEmpty:<Expression1>:<Expression2>:.....:<ExpressionN>
```

Where,

Expressions can be variables, formulae, functions, etc.

Example

```
[Collection: TNOldAnnexIASummaryWithoutError]
```

```
Compute Var : HasError : Logical :$$IsAnyEmpty:##svRefNo: +  
##svRefDate:##svLorryDate
```

Ensure that the sequence of parameters is such that higher the probability of an expression being Empty, earlier in the order they should be placed.

Previously, to evaluate more than one expression for empty, you had to use **\$\$IsEmpty** along with **OR** operator. Now, you can just use the new function to evaluate multiple expression in a single line.

2. Language Enhancements in Procedural (TDL)

The actions **Increment** and **Decrement** have been enhanced to accept multiple variable.

3. Language Enhancements in Query (Collections)

3.1 Conditional WalkEx

The condition parameter of the Collection attribute, **WalkEx** is now enhanced to check for every Source Object.

Syntax

```
Walk Ex: <Collection> [:Condition]
```

Condition is optional parameter. When provided, each source object will be evaluated for based on the condition to decide if Walk Ex (of specified collection) has to be executed.

3.2 Other Enhancements

Please click the function/attribute name to know more.

1. **\$\$IsCollSrcObjChanged**: Currently, in Extract Collections while walking the sub-objects, there is no direct way of identifying if the source object context has changed. For this purpose, a new function `IsCollSrcObjChanged` is provided to identify when a source object is changed during a Walk.
2. **\$\$CollSrcObj**: A new function `CollSrcObj` is provided to evaluate expression in the context of source object while walking the current object.
3. **Source Fetch**: A new attribute `Source Fetch` is provided to fetch methods from source object context while walking the current object.
4. **Aggr Compute**: Apart from the keywords `Sum`, `Min` and `Max`, the collection attribute `Aggr Compute` is enhanced to support the keyword **Last** to extract the method value from the last object.
5. **ReWalk and ReCompute**: The collection attributes `ReWalk` and `ReCompute` are provided for re-computation in collections.
6. **Prefetch and Source Prefetch**: New attributes `Prefetch` and `Source Prefetch` are provided to pre-fetch the object method and retain the values for the subsequent usage within the current context, without having to re-evaluate the expression (in the Object Method) each time it is used.

4. Enhancements in Customisation using Productivity Suite

Customisation using Productivity Suite is enhanced to take MS Excel as an input file and produce the specified Excel file as an output. The data is evaluated in the tokenized input file and then, only the values are written to the output file specified in the configuration screen.

If the output file is present, data will be updated to the appropriate cells of respective sheets provided in the input Excel file. If the output file is not present, then the same input Excel file will be created with the specified output file name.

To support this capability, **MS Excel** is introduced as a new **Resource Type** in Resource definition.

What's New in Release 5.3.8

As we are aware, File Input/Output capability is used to support read/write operations in an Excel or Text file.

Currently, if we write any value to an excel file using File I/O approach, the appropriate column has to be adjusted manually based on the cell width after completion of the task. Now, a new capability is introduced to update the Cell Properties, Width and Text Wrap.

These properties can be set in TDL using the action, Format Excel Sheet.

1. Action – Format Excel Sheet

Action **Format Excel Sheet** is used to set the cell properties of Excel sheet. It accepts two parameters.

Syntax:

```
Format Excel Sheet : <PropertyName> : <PropertyParms>
```

where,

<PropertyName> is a system keyword for an Excel cell property, viz., ColumnWidth or CellTextWrap.

<PropertyParms> is a list of required parameters for the specified PropertyName and the number of parameters vary based on the PropertyName.

Example:

```
[Function: FileIOExcel]
```

```
Variable : InputFile : String : "D:\SampExcel.xls"  
  
010 : Open File: ##InputFile : Excel : Write  
  
020 : Format Excel Sheet: CellTextWrap : 5 : 6 : Yes  
  
030 : Format Excel Sheet: ColumnWidth: 1 : 30  
  
040 : Close Target File
```

In the above example, **CellTextWrap** is the PropertyName and "5: 6: Yes" is the parameter required for the CellTextWrap property.

Supported Properties

ColumnWidth

Syntax:

```
ColumnWidth: <ColumnNumber> : <Width>
```

where,

<ColumnNumber> is used to specify the column number whose width is to be modified.

<Width> is the number used to set the new width of the column. Unit of measurement used for this parameter is Points.

CellTextWrap

Syntax:

```
CellTextWrap: <RowNumber> : <ColumnNumber> [ : <Enablewrapping> ]
```

where,

<RowNumber> is used to specify the row number of the cell.

<ColumnNumber> is used to specify the column number of the cell.

<Enablewrapping> is an optional parameter which specifies whether to wrap the text or not. The default value of this parameter is Yes.



An expression can also be specified in any of the parameters for the action.

What's New in Release 5.3

1. Attribute – Confirm Text/Query Text

The **Form** level attribute **Confirm Text/Query Text** is used to modify the text displayed in the confirmation window.

Syntax

```
Confirm Text : <String Expression>
```

or

```
Query Text : <String Expression>
```

Where,

<String Expression> is the text displayed in the confirmation window.

Example:

```
[Form: SmpConfirmText]
```

```
Part : SmpConfirmText
```

```
Confirm Text : "Save?"
```

In the above example, when the form **SmpConfirmText** is accepted, the query text **Save?** is displayed instead of the default text **Accept?**.

2. Action – Exec Excel Macro

The action **Exec Excel Macro** invokes the available macros defined in the Excel.

Syntax:

```
Exec Excel Macro : <Macro Name> [:<Parameter list>]
```

Where,

<Macro Name> is the name of the macro.

<Parameter list> can be n number of parameters that correspond to the parameters required by the Excel macro.

Example:

```
On : After Export : Yes : Exec Excel Macro : MacrotoComputeGraphs : PieChart
```

In the above example, when the macro **MacrotoComputeGraphs** is executed, it displays the values in the Excel as a pie chart.

What's New in Release 5.2

1. Column-wise repeat of data over a collection

The **Form** level attribute **Repeat** is used to repeat data of a collection column-wise in the reports created using productivity suites, using the function **TplColumnObject**.

Syntax

Repeat : <Token Name> : <Collection Name>

Where,

<Collection Name> is the name of the collection or a sub-collection.

<Token Name> is the name of the token specified in the document template for evaluating the value using the attribute **XML Map**.

1.1 Function – TplColumnObject

Function **TplColumnObject** evaluates the given parameter in the context of the column object. In the absence of this function, the expression is evaluated in the current context of the **Report**.

Syntax:

\$\$TPLColumnObject:<Expression>

Where,

<Expression> can be any expression which evaluates to any data type like, string, number, amount, and so on.

Example:

To print ledger names as columns:

- Design the document template as shown below:

\$LedName

Token is specified in one cell, based on the number of objects in the collection, the columns are added.

- Add the below code snippet in the required TDL.

```
[Form : Sample Report]
```

```
XML Map      : LedName      : @@TPLColObjName
```

```
Repeat       : LedName      : LedgerColl
```

```
[System : Formula]
```

```
TPL ColObjName : $$TPLColumnObject:$Name
```

```
[Collection : LedgerColl]
```

```
Type : Ledger
```

The output appears as shown below:

Ledger1	Ledger2	Ledger3
---------	---------	---------

In this example, the collection includes three ledgers. All the three ledgers are added as columns.

What's New in Release 5.0

1. Customisation using Productivity Suites

1.1 Introduction

Customisation using productivity suites is a facility to create reports in the required format with minimal time consumption and effort, using the applications available in productivity suites like Microsoft Office, Open Office, and so on. Using this facility, Tally.ERP 9 accepts a predefined document template designed using any of the productivity suites, and generates output in the desired format. For example, if the input is defined as a Word XML Document, the output is displayed in MS Word.

Based on the business requirements, there may be a need for change in the format of default documents like statutory forms provided in Tally.ERP 9. To reflect these changes in the default documents, users can now edit the document template using MS Word or MS Excel, and create the required layout.

The applications supported are:

- ❑ Microsoft Office (2003 and above): MS Word, MS Excel
- ❑ Open Office: Open Document Text (.odt), Open Document Spreadsheet (.ods)
- ❑ XML (Data Exchange)

1.2 Prerequisites for the User

The user is required to have any of the following productivity suites:

- ❑ MS Office 2003 or higher version
- ❑ Open Office
- ❑ Office Viewer



Tally.ERP 9 uses any compatible format based on the productivity suite available in the user system.

*To know more about designing or creating the document template, refer the section **Guidelines for Designing Document Template**.*

1.3 TDL Enhancements for the Capability

In Tally.ERP 9, all reports follow the design hierarchy of **Form→Part →Line→Field**. The same hierarchy is used for creating and printing the defined layout.

The user can design the required document template for print as a MS Word or MS Excel document, and save it in XML format. The designed document template contains the layout, and the corresponding data to be fetched. Once the document is designed, it is to be associated to the respective TDL Form definition. Once the document is associated to the Form definition,

Tally.ERP 9 accepts this document. In this case, the application does not proceed further into the hierarchy.

To support the capability Customisation using Productivity Suites, two Form level attributes and four platform functions are introduced. Also, the attribute value of Resource Type is extended to accept other file formats. The language enhancements are listed below:

- Form Attribute – Resource
- Form Attribute – XML Map
- TDL Function – \$\$TplLine
- TDL Function – \$\$WordInfo
- TDL Function – \$\$ExcelInfo
- TDL Function – \$\$IsFileTypeSupported
- Resource Types – WordXML, ExcelXML, XML, ODT, ODS

Form Attribute – Resource

The attribute **Resource** at Form definition is used to specify the name of the Resource definition, which provides the details of the document template. This document template is used for exporting/printing the Form.

Syntax:

```
[Form: <Form Name>]
  Resource : <Resource Name>
```

Where,

<Resource Name> is the name of the resource definition.

Form Attribute – XML Map

XML Map is a Form level list type attribute, used to provide correct value for the token(s) specified in the document template.

Token is an alpha-numeric value without any spaces, prefixed with \$. No special characters are allowed in the token name. It is an expression specified in the document template to map the values from Tally.ERP 9.

For example, \$Name, \$CmpName, \$FrDate, \$Val1

When the document template is used for exporting/printing:

- The defined tokens in the document template are substituted with the respective values by evaluating the expression in XML Map attribute, against the defined token.
- If the token is not defined in the document template, it evaluates as a method in current object context. A token refers to a method of current object context. In this case, defining XML Map is not necessary.
- XML Map also allows specification of repeatable data, when it is referenced in Tally.ERP 9. It creates required number of rows/entries for each object in the collection.

Syntax:

[Form: <Form Name>]

XML Map: <Map Name>: <Expression >[:<Collection Name>]

Where,

<Map Name> is the name of the token specified in the document template for evaluating the value.

<Expression> is a valid TDL expression such as method, variable, system formula.

<Collection Name> is specified where the token is defined for repeating the data. It is an optional parameter.

Example:

[Resource: InvoiceWord]

Source : "D:\Work\InvoiceWord.xml"

;;It is the word document saved as .xml(WORDXML)

Resource Type : Word XML

[Form: InvFrm]

Resource : InvoiceWord

XML Map : BasicSDN : @@DelNoteNo

XML Map : SVCompany : ##SVCURRENTCompany

XML Map : Rate : \$Rate : Inventory Entries

[System: Formula]

DelNoteNo : \$BasicShipDeliveryNote

In the example, **InvoiceWord** is the name of the resource. **BasicSDN**, **SVCompany** and **Rate** are the tokens which are defined in the document template, and values are taken from the respective second parameters, **@@DelNoteNo**, **##SVCURRENTCompany** and **\$Rate:Inventory Entries**.



For every token, Tally.ERP 9 looks for an XML Map definition

- If Tally.ERP 9 finds the specified token in the XML Map definition, it evaluates and replaces the specified expression.
- If Tally.ERP 9 does not find the specified token in the XML Map definition, it treats the token as a method in current object context, and evaluates the value.
- If the value corresponding to a token is not found in the XML Map or in any of the object storages/methods, the token is printed as it is in the document.

For example, \$DutyValue is the token, and there is neither XML Map with this name nor any storage/method in that object context, then system prints \$DutyValue as it is, indicating that the value corresponding to the token is not available.

- When collection name is specified, the specified paragraph or row in the document template is repeated for every collection object. When the objects are not found in the collection, paragraph or row is not displayed.

TDL Function – \$\$TplLine

Function \$\$TplLine provides line number or object index number in the current collection, where the paragraph or rows of the document template are repeated.

Syntax

`$$TplLine`

Example:

```
[Form: InvFrm]
```

```
Resource : InvoiceWord
```

```
XML Map : Line : $$TplLine
```

TDL Function – \$\$WordInfo

This function helps to find whether MS Word is installed in the user system or not. It accepts any one parameter, i.e., Version or IsDocxSupported.

- Version – to get the version number of MS Word.
- IsDocxSupported – to check whether the .docx format is supported or not.

Syntax:

`$$WordInfo : <Info Type>`

Where,

<Info Type> has two values, i.e., **Version** and **IsDocxSupported**

Example:

```
Set as : $$WordInfo : Version
Set as : $$WordInfo : IsDocxSupported
```

TDL Function – \$\$ExcelInfo

This function helps to find whether MS Excel is installed in the system or not. It accepts any one parameter, i.e., Version or IsXlsxSupported.

- Version – to get the version number of MS Excel.
- IsXlsxSupported – to check whether the .xlsx format is supported or not.

Syntax:

```
$$ExcelInfo : <Info Type>
```

Where,

<Info Type> has two values, i.e., **Version** and **IsXlsxSupported**

Example:

```
Set as : $$ExcelInfo : Version
Set as : $$ExcelInfo : IsXlsxSupported
```

The behavior of different versions of MS Office in the use of \$\$ExcelInfo and \$\$WordInfo are as shown below:

MS Office	Version	Docx Supported	Xlsx Supported
2003	11	No	No
2007			
2010	14	Yes	Yes
2013	15	Yes	Yes

TDL Function – \$\$IsFileTypeSupported

It returns the logical value, whether the provided format is supported in the system or not.

Syntax:

```
$$IsFileTypeSupported : <document extension>
```

Where,

<document extension> is the type of file extension.

Example:

```
$$IsFileTypeSupported:".odt"
$$IsFileTypeSupported:".ods"
```

The behavior of different versions of MS Office are as shown below:

MS Office	.odt Supported	.ods Supported
2003	No	No
2007	No	No
2010	No	No
2013	Yes	Yes

Resource Types – WordXML, ExcelXML, XML, ODT, ODS

In TDL, **Resource** definition is used to access and use the resources (images, icons, cursors, etc.) from a local disk, HTTP/FTP, or from a DLL/EXE. The formats supported are BMP/JPEG/ICON/CUR. These resources are allowed in **Part** definition, using the attribute **Image** for displaying, printing and exporting.

To support the facility Customisation using Productivity Suites, the capability of **Resource** definition is enhanced to use the resources like Word Xml, Excel Xml, Xml, odt, ods as resource types.

Syntax:

```
[Resource: <Resource Name>]
    Source           : <Path to File>
    Resource Type   : <Supported Resource>
```

Where,

<Path to File> is the path to the document or Image.

<Supported Resource> is the resource type which is supported. The supported resources are ODS/ODT/WORD XML/Excel XML/XML/Bitmap/Icon/Jpeg/Cursor.

Example:

```
[Resource: InvoiceWord]

    Source           : "D:\Work\InvoiceWord.xml" ;;this is the document saved as xml

    Resource Type   : Word XML
```

In the example, **InvoiceWord.xml** is the Word XML document template, and it prints in Word format from Tally.ERP 9.

1.4 Guidelines for Designing the Document Template

To design a document template consider the following

- Design the document template for a new format.
- Use the document template available in Word/ODT/Excel/ODS. These document templates may require modifications to get a proper output.

The file formats supported with this facility to print/design a document in printable format are:

Format	Format Type	Usage
Word Formats	Word 2003 XML Document (.xml)	For complex layouts, multiple tables, Forms, and so on.
	Open Document Text(.odt)	
Excel/Spreadsheet Formats	Excel Spreadsheet 2003 (.xml)	For listing summary, annexure, and so on.
	Open Document Source(.ods)	



- *Generate any suitable format which is suitable for customers/clients in Microsoft or OpenOffice.*
- *Designed document template is for generic purpose, it is recommended that generate the document templates in WordXML, & .ODT and ExcelXML & .ODS format.*

Creating Multiple Formats

To create document templates in multiple formats

1. Design the document template either in MS Office or Open Office.
2. Copy the document template, and **Save As** in the required format.

Example: 1

1. Design the document template in Word.
2. Save the document template as **Word 2003 XML document(.xml)**.
3. Open the designed document template.
4. Copy the content from the template.
5. Open an Open Office document, and paste the content.
6. Save as **.ODT**.
7. Open with any of the **Open Office** applications.

Example: 2

1. Design the document template in Word.
 2. Save the document template as **Word 2003 XML Document (.xml)**.
 3. Open the designed document template.
 4. Save as **.ODT** in Word.
 5. Open the document template using any of the Open Office applications.
 6. If the format is incorrect, correct the format, and save as **.ODT**.
3. Print the designed document template and check the font, alignment, width of values, data wrapping, image, value with page break, and so on, for any formatting errors.

Using Multilingual Capability in the Document Template

In Tally.ERP 9, to print/e-mail/upload the documents in different languages, the document templates are flexible and can be generated in the desired language. In multilingual document templates, the tokens are always in English.

Scenarios:

Scenario 1: Titles in English, and data in any other language

- a. Create the document template with all labels/titles in English.
- b. Token values are printed in the language used to enter data in Tally.ERP 9. The title appears in English, and data, based on language settings in Tally.ERP 9.

Scenario 2: Titles and data in a different language

- a. Create the document template with the labels/titles in any one language by setting the Windows keyboard language.
- b. The data is printed in the language used in Tally.ERP 9. The title appears in one language, while data will appear in the language set in Tally.ERP 9.



When you design document template using the multilingual capability use the font which supports Unicode. For example., Arial Unicode MS.

General Guidelines

1. **Use platform defined resource types:** WordXML, ExcelXML, XML, .ODT and .ODS are considered as valid resource types for the capability Customisation using Productivity Suites. Any other resource types are treated as invalid, and does not print.
2. **Horizontally adjacent tables are not applicable for Excel/.ODS:** Two tables adjacent to each other horizontally (one table repeating on one set of data, and other table repeating another set of data) are not recognised as separate, as Excel considers the complete row as a single table.
3. **Design table within table for Word/.ODT:** To have multi-line address in a single cell, add a table inside that cell, with the required number of columns. When data is repeated for that table, the number of rows are added accordingly.

4. Print a single form in multiple pages:

- ❑ Check the headers for each page.
- ❑ Use the option **Fit to one page**, if required.
- ❑ Select the appropriate orientation, portrait or landscape.
- ❑ Adjust margins to fit more columns, and so on.

5. Horizontal repeat over a string to print each letter/digit in separate boxes is supported:

Horizontal repeat over a string is supported character-wise also. For example, if TIN number has 10 digits, and each digit is printed in separate boxes.

2	6	5	4	7	9	0	5	2	8
---	---	---	---	---	---	---	---	---	---

To achieve this, while designing the template in Word/ODT, create a table with number of cells equivalent to the number of characters in the data, and specify the token in the first cell, as shown below:

\$TIN																			
-------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

By adding the below code snippet in the TDL, the output appears as shown below:

[Form: sample]

XML Map : TIN : \$TinNumber ;;\$TinNumber = 4512468921

Repeat : TIN : String

4	5	1	2	4	6	8	9	2	1										
---	---	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--	--	--

If the document template does not support the character length of the data, data loss will occur.

\$TIN				
-------	--	--	--	--

In this case, add the code snippet in TDL shown below:

[Form: sample]

XML Map : TIN : \$TinNumber ;;\$TinNumber = 4512468771

Repeat : TIN : String

The output appears as shown below:

4	5	1	2	4
---	---	---	---	---

In this example, the document template contains only 5 cells, and the data has 10 characters. Hence, in the output, only 5 characters are displayed.

Print/Export/E-mail/Upload for Multiple Objects/Forms

When print/export/e-mail/upload is done for multiple objects, specified as part of **Collection** attribute at Report level, that many number of files are generated.

Consider the below points:

Print with preview:

- The number of files generated, opened and printed, equals the number of objects.
- The files are deleted once they are viewed and closed.
- The name of the file is randomly generated.
- The file extension is decided by the format selected in the configuration screen.

Print without preview:

- The number of files generated and printed equals the number of objects.
- The files are deleted once they are printed.
- The name of the file is randomly generated.
- The file extension is decided by the format selected in the configuration screen.

Export:

- The number of files generated equals the number of objects. Only the first file is opened.
- The name of file is user specified
- The extension is decided by the format selected in the configuration screen

Designing XML Formats

XML formats are used to interchange the data as per the personalized requirements. In Tally.ERP 9 these formats are used to upload files to government websites, for filing returns. These XML formats are either provided by the government.

Once the XML format is ready,

1. Define tokens in between the XML tags for values
 - Use the XML attribute `__tlyEmptyIf = "$<TokenName>"` to eliminate a particular tag from the XML.
2. Edit the XML using editors like Notepad, Notepad++, SlickEdit, and so on.
3. Save the edited document template as .xml file.

The XML tag can be repeated for a TDL collection of data. The repeated XML tags can further have tags which are repeated for other TDL collection of data.

Some Known Issues

- Merging rows are not supported in Excel.
- When exporting data, using XML(data interchange) Sysnames and data with special characters, for example, □Primary, are not considered.
- ExcelXML format does not consider images for print/export/email/upload.

Export

While exporting the documents specify the proper format in the export configuration screen, and proper file name extension to achieve the expected output. Provide the format and file extension based on requirements.

Export Format	Format	File Extension
WordXML	Word 2003 XML Document (.xml)	.xml
ODT	Open Document Text	.odt
ExcelXML	Excel	.xls
ODS	Open Document Spreadsheet	.ods

Protecting the Document Templates from Edit

Word XML:

- ❑ Open the document template in MS Word.
- ❑ Click **Restrict Editing** from **Review**.
- ❑ Select the option **Allow only this type of editing in the document**.
- ❑ Click **Yes, Start Enforcing Protection**, the screen **Start Enforcing Protection** appears.
- ❑ Enter the password details, if required.
- ❑ Click **OK** to accept the changes.
- ❑ Save the document template.

This will save the document as read only, and it can be edited only if users clear this option.

ODT:

- ❑ Open the document template in any Open Office application (Libre Writer).
- ❑ **Save As** the document template, and select the option **Save Password**.
- ❑ Enter the password details.
- ❑ Click **Cancel** and proceed.

It saves the document template as read only, and it can be edited only if users clear the option **Save Password**.

Excel XML:

- ❑ Open the document template in MS Excel.
- ❑ Click **Protect Sheet** button from **Review** menu.
- ❑ Select the options **Select locked cells, Select Unlocked Cells**, and click **OK**.
- ❑ Save the document template.

It saves the document template as read only, and it can be edited only if users clear the option.

ODS:

- ❑ Open the document template in any Open Office application (Libre Writer).
- ❑ Click **Save As** to save the document template, and select the option **Save Password**.
- ❑ Enter password details.
- ❑ Click **Cancel** and proceed.

It saves the document as read only, and it can be edited only if users clear the option **Save Password**.

2. Other Language Enhancements

2.1 Function – \$\$MakeMailName

The function **MakeMailName** is being used in TDL to construct the e-mailing details like server name, **To** e-mail addresses, **Cc** e-mail addresses, **Subject**, and so on. However, in TDL there is no capability to send a blind carbon copy (Bcc).

The function **MakeMailName** is enhanced to accept an optional parameter **Bcc** address. E-mail addresses specified here will not be displayed for recipients listed under **To** and **Cc**.

Syntax:

```

$$MakeMailName: <To Address>: <SMTP Sever Name>: <From Address>: +
                <CC Address>: <Subject>: <User Name>: <Password>: +
                <Use SSL Flag> [:<Use SSL on Standard SMTP Port> +
                [:<Bcc Address>]]
  
```

Where,

<To Address> is the e-mail address of the recipient.

<SMTP Server Name> is the name of the e-mail server from which the e-mail is sent.

<From Address> is the e-mail address of the sender. It must contain company name and e-mail address.

<Cc Address> is the e-mail addresses to whom the copy of this e-mail is sent.

<Subject> is the subject of the e-mail.

<User Name> is the authenticated user name on the secured server.

<Password> is the password for the user on the secured server.

<Use SSL Flag> can be **True** if the secured SMTP server is being used.

<Use SSL on Standard SMTP Port> can be **True** if the SSL is used on the default/standard SMTP Port.

<Bcc Address> is the e-mail address to whom the blind carbon copy of the e-mail is sent.

Example:

```

$$MakeMailName :mailserver.tallysolutions.com
                : "Tally" <"Frommail@tallysolutions.com">
                : "ToUser1@tallysolutions.com, ToUser2@tallysolutions.com,
                ToUser3@tallysolutions.com": "CcUser1@tallysolutions.com,
                CcUser2@tallysolutions.com": "Your Outstanding Payment": ""
                : "" : No : No : "BccUser1@tallysolutions.com,
  
```

BccUser2@tallysolutions.com"

2.2 Action – Delete Target

Action **Delete Target** is introduced to delete a primary object from the company.

Syntax:

Delete Target

Example:

```
[Function: Emp Led Deletion]
00 : Walk Collection : Empty Ledgers
10 :   Set Target
20 :   Delete Target
30 : End Walk
```

In the example, the action **Delete Target** is invoked. **Delete Target** then walks the collection **Empty Ledgers**, sets the current ledger object to target context, and deletes the ledger.



Masters can be deleted only when empty. For example, if there is a transaction under the master that is being deleted, then the master object cannot be deleted.

2.3 Data Type – Calendar

Calendar data types, viz., Date, Time, DateTime, Duration, and DueDate, were introduced in the previous releases to support various business requirements like, capturing date and time of entering a voucher, calculating weekly average log-in time of employees, and so on.

To extract the Hours, Minutes, Seconds, and Milliseconds from Time data type, a set of functions have been introduced.

Function – \$\$HourOfDay

The function **\$\$HourOfDay** is used to extract the hour from a value of the data type Time.

Syntax:

\$\$HourOfDay : <Time Expression>

Where,

<Time Expression> is a valid time value.

Example:

```
$$HourOfDay : ##UserInputTimeValue
```

Function \$\$MinuteOfDay

The function **\$\$MinuteOfDay** is used to extract the minutes from a value of data type Time.

Syntax:

```
$$MinuteOfDay : <Time Expression>
```

Example:

```
$$MinuteOfDay : ##UserInputTimeValue
```

Function \$\$SecondOfDay

The function **\$\$SecondOfDay** is used to extract the seconds from a value of data type Time.

Syntax:

```
$$SecondOfDay : <Time Expression>
```

Example:

```
$$SecondOfDay : ##UserInputTimeValue
```

Function \$\$MilliSecondOfDay

The function **\$\$MilliSecondOfDay** is used to extract the millisecond from a value of data type Time.

Syntax:

```
$$MilliSecondOfDay : <Time Expression>
```

Example:

```
$$MilliSecondOfDay : ##UserInputTimeValue
```

2.4 Definition – QueryBox

Query Box action is used to display the query box with two options. Now, a definition **QueryBox** is introduced to provide multiple options.

You can use the definition **QueryBox** to create a query box with specific queries in TDL. Using this definition multiple options can be provided, including a shortcut key to select each option, and text to explain the implication of each option in the query box. This definition has various attributes to control the query behavior, and an associated option for users to select.

Syntax

```
[QueryBox : Query Box Name]
    Title           : <Query box title>
    Horizontal Align : <Alignment of Query Box>
    Vertical Align   : <Alignment of Query Box>
    Query           : <Hot Key> : "<Query option string>" :
                    "<Additional description>"
    Query           : <Hot Key> : "<Query option string>" :
                    "<Additional description>"
                    :
                    :
```

Default : <Enter key option index> : <Escape key option index>

Where,

Title - This attribute displays the title of the query box.

Horizontal Align - This specifies the horizontal alignment of the query box on the screen. Values can be left, center or right aligned.

Vertical Align - Specifies the vertical alignment of the query box on the screen. Values can be top, center or bottom aligned.

Query - This attribute lists the query, key, and text to explain the implication of the option in the query box.

Default - This attribute sets the default value in number when the **Enter** or **Escape** key is pressed.

The first attribute corresponds to the **Enter** key, while the second attribute corresponds to the **Escape** key.

2.5 Action – Query Box Ex

The action **Query Box** is used to display the query box with dual options, and waits for the user to select an option. The action **Query Box Ex** is a procedural action which invokes a predefined query box. This action displays the query box as per the behavior specified in the **QueryBox** definition, and displays multiple options as specified in the definition. The selected option can be accessed by using the function **LastResult** after this action.

Syntax

Query Box Ex : <Query Box Name>

Where <Query Box Name> is the name of the **QueryBox** definition.

2.6 Attribute – Control Ex

A Form attribute **Control** controls the acceptance/rejection of a Form, based on the logical evaluation of the expression. Similarly, the Form attribute **Control Ex** can be used to invoke the defined **QueryBox**, and control the behavior of the Form. The selected response can be held in a variable.

Syntax

Control Ex : <Query Box Name> : <Logical Expression> [:<Variable Name>]

Where,

<Query Box Name> is the name of the **QueryBox** definition.

<Logical Expression> if specified as **Yes**, displays the query box.

<Variable Name> is a variable name which is updated with the selection index, post query box execution. It is an optional parameter.

If Variable Name is not specified, the selection index is not available for post query box execution, only **Control** behavior will work.



*The user response can be accessed using the variable value, which can be used in the event **On: Form Accept**, and the desired outcome can be achieved.*

2.7 Attribute – Unique

The Field attribute **Unique** is used to control the repeated Line with the Field having unique values. However, the uniqueness behavior ignores the noise characters, and considers the strings **tallyuser@tally.com** and **tally.user@tally.com** to be same. Hence, to check the uniqueness in terms of exact matching of strings, Field attribute **Unique** is enhanced to accept a second logical parameter (optional). The second parameter is valid only if the first parameter is **Yes**. If the second parameter is **Yes**, then the Field is checked for uniqueness of the strings.

Syntax

```
Unique : <Logical Expression> [: <Logical Expression>]
```

Where,

<Logical Expression> can be **Yes** to enable exact match for strings and vice versa.

By default, the second logical expression is **No**.

Example:

When both the logical expressions are specified as **Yes**, **tally.user@tally.com** and **tallyuser@tally.com** are considered two unique strings.

2.8 Attribute – WalkEx

Collection attribute **WalkEx** is used to walk the paths of the source collection, similar to attribute **Walk**. The advantage with **Walk** attribute is that more than one path can be traversed within a single pass, as against having two summary collections. It walks the same source for as much as sub-objects.

Attribute **Walk** walks over the sub-objects irrespective of whether it is required or not. To support conditional walk, attribute **WalkEx** has been enhanced to accept a logical expression as its second parameter. Based on the evaluation of the logical expression, the attribute **WalkEx** will or will not walk the paths specified in the collection list.

Syntax:

```
[Collection : MyCollection]
WalkEx : <Collection Name> [:<Logical Expression>]
```

Where,

<**Collection Name**> is the name of the collection name specifying walk, and aggregation/computation attributes.

<**Logical Expression**> is an optional logical parameter which determines whether to walk over the sub-objects or not.

What's New in Release 4.8

The Release 4.8 of Tally.ERP 9 comes with a number of TDL enhancements in Data Import, Events, Actions, Functions, etc., which have been discussed ahead in detail.

1. Data Importing Enhancements

Updation of data is vital for all kinds of reports generated in any business organisation. Hence, the designed system must be robust enough to accept the data with appropriate validation and controls. 'Garbage in, Garbage Out', as we have commonly heard, means that any invalid data entered into the system would result in an invalid output. Tally allows updation of data through various sources. They are:

- Manual Data Entry
- Data Updation through User Defined Functions, from any External Source
- Data Import from XML/SDF files, Synchronization and Third Party Application Request

Whenever data is received for updation from any source, validations like matching of Debit/Credit Totals, availability of dependent masters in the data while importing voucher, etc., are done prior to updating the database. These basic validations, without which Objects cannot exist in the database, are taken care of by the Tally platform, irrespective of the data source. However, there are certain business rules/controls like Entry outside Current Accounting Period not being accepted, Sales to Party exceeding Credit Limits not being allowed, etc., which also need to be applied. In the case of User Interface, these controls are provided within the default application, so that when the user makes entries, the data is accepted only after these controls are passed. While updating the Data through a User Defined Function, the programmer can apply the relevant checks prior to importing the Objects in the Tally Database.

Before this Release, during data import from XML/SDF files, Synchronization or from an external request, the data could not be validated for these business rules, and hence, the data import could not be stopped even if the Object was invalid. In Release 4.8, a few Enhancements have been made to give appropriate controls to the Programmer during the process of importing the data from any source. Thus, it empowers the TDL Programmer to take any desired action while Importing the Data. These enhancements are as listed below:

- Apart from the three existing Import File Events, namely 'START IMPORT', 'IMPORT OBJECT' and 'END IMPORT', a new Event '**After Import Object**' has been introduced to perform the desired action after importing every Object.
- Two Logical System Variables '**SVImportObject**' and '**SVContinuelmport**' have been introduced, which give control to the TDL Programmer to conditionally import the Objects, or to terminate the Import Process.
- Enhancements have also been introduced for sending a **customized response** to the requesting application, in case of SOAP Request.
- To complement all the above Enhancements, a set of new Functions like **\$\$ImportInfo**, **\$\$HTTPInfo**, **\$\$ImportType**, **\$\$LastImportError**, etc., have been introduced.

1.1 Import Events

The Import Events 'Start Import', 'Import Object' and 'End Import' are available in Tally.ERP 9 since Release 3.0. All these Import Events can be used within the Definition 'Import File'.

- The Event 'Start Import' is invoked only once, at the beginning of Data Import.
- The Event 'Import Object' is invoked for every Object, prior to importing the Object in current context.
- The Event 'End Import' is invoked only once, at the end of Data Import.

Import Event - After Import Object'

In Release 4.8, a new Event 'After Import Object' has been introduced to allow the TDL Programmer to take any desired action, every time an Object is imported. This Event will help the user to create appropriate logs after Importing of each Object, terminate the Import Process based on some condition, track/record changes that can be used for preparing the response (for SOAP requests), etc.

Syntax

```
[Import File : <Import File Definition Name>]
    On : After Import Object: <Logical Condition Expr> : <Action Keyword>
        + : <Action Parameters>
```

Where,

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the given Action.

<Action Keyword> is the Action to be taken once the System Event is triggered.

<Action Parameters> are the parameters required for the particular Action.

Example:

```
[Import File : All Masters]

    On : After Import Object : Yes : Call: Update Log

    On : End Import          : Yes : Call: Show Log

[Function : Update Log]

    00 : If : $$ImportAction = "Errors"

;; New Function introduced to check whether the updation of current object is successful or encountered with
an error

    10 : Log : $$LastImportError

;; New Function to return the Error encountered during last Import Object

    20 : End If

[Function : Update Log]

    10 : Exec Command : TDLFunc.Log
```

In this example, after import of each Object, the Event 'After Import Object' is triggered, which calls a Function 'Update Log' to update the logs with Error String, if any. Once the Import Ends, the Event 'End Import' is triggered, thereby calling the Function 'Show Log' to open the File *TDLFunc.Log* for browsing through the errors occurred, if any.



The currently imported object is available as the object in context.

1.2 System Variables

The Events 'Start Import', 'Import Object', 'After Import Object' and 'End Import' provide some controls to the TDL Programmer. In spite of having these controls, the Objects would get imported after performing the Actions corresponding to the Events, irrespective of the user choice. Hence, to give complete control to the user to take the suitable action, two new Logical System Variables '**SVImportObject**' and '**SVContinueImport**' have been introduced.

System Variable - SVImportObject

With the help of the System Variable 'SVImportObject', the TDL Programmer is empowered to decide whether to import the current object or not. It is a logical variable which communicates the same information to the system. If this Variable is set to FALSE or NO, the current Object will not be imported. The default value of this variable is TRUE or YES. Ideally, the value of this variable needs to be set/alterd only at the Event 'Import Object', which is triggered prior to importing the current Object.

Example:

```
[#Import File : All Masters]

    On : Import Object : $$IsStockItem:Call:DontoverrideExistingItem

[Function : DontoverrideExistingItem]

00 : IF : $$IsEmpty:$Name:StockItem:$Name

10 : Set : SVImportObject : True

15 : Msg Box : "Creation" : "Stock Item" + $Name + "being created"

20 : Else :

30 : Set: SVImportObject : False

35 : Msg Box : "Alteration" : "Stock Item" + $Name + "already +
                                exists and cannot be altered"

40 : End If
```

In this example, the Event 'Import Object' is triggered while importing the Masters and the Function 'DontoverrideExistingItem' is invoked only if the current Object is of Type 'Stock Item'.

This Function checks if the current Stock Item is available in the Tally Database. If the Stock Item is not found, then the Variable 'SVImportObject' is set to TRUE, which instructs the system to continue to import the current Stock Item Object. If the Stock Item is found in the current Company being Imported, the Variable 'SVImportObject' is set to FALSE, which prohibits the Importing of the current Stock Item Object.

System Variable - SVContinueImport

With the help of System Variable 'SVContinueImport', the TDL Programmer gets the control to decide whether to terminate the Import Process, or to continue further. It is a Logical Variable which communicates the same information to the System. If the Logical value of the Variable is set to FALSE or NO, the import process will be terminated. The default value is TRUE. Ideally, the value of this variable needs to be set/changed only at the Event 'Import Object' or 'After Import Object', since these events are triggered before and after the importing of every Object.

Example:

```
[#Import File : Vouchers]

    On : Start Import   : Yes   : Call : InitializeSystemVariable

    On : Import Object  : Yes   : Call : IncrementSystemVariable

    On : After Import Object : ##SystemVarforImport= 50 : Call +
                                   : TerminateImportProcess

[Function : InitializeSystemVariable]

    00 : Set           : SystemVarforImport : 1

[Function : IncrementSystemVariable]

    00 : Increment    : SystemVarforImport

[Function : TerminateImportProcess]

    00 : Set : SVContinueImport : False
```

In this example, it is intended to import only 50 Objects, beyond which, the Import process should be terminated. A variable 'SystemVarforImport' is initialized at the 'Start Import' Event. During the 'Import Object' Event, the value of this variable is Incremented for each Object. After importing 50 Objects, the Import Process is terminated by setting the Variable 'SVContinueImport' to FALSE.

1.3 Customized response with 'Response Report' attribute

When a SOAP Request is received to Import Data from an external application, Tally.ERP 9 imports the data and sends an appropriate default response to the Third Party Client. The default response sent to the requesting Clients after Importing the Data is as follows:

```
<RESPONSE>

    <CREATED>3</CREATED>

    <ALTERED>2</ALTERED>
```

```
<LASTVCHID>87</LASTVCHID>
<LASTMID>0</LASTMID>
<COMBINED>0</COMBINED>
<IGNORED>0</IGNORED>
<ERRORS>0</ERRORS>
<CANCELLED>0</CANCELLED>
</RESPONSE>
```

However, the various Clients sending requests have disparate Integration solutions implemented and hence, the expectation is that the response must be in the formats prescribed by their Integration Solutions. Thus, in order to provide flexibility to the TDL Programmer by supporting the required formats in disparate Integration scenarios, a new Import File Attribute '**Response Report**' has been introduced.

Attribute - Response Report

The attribute 'Response Report' has been introduced in the 'Report File' definition to allow the TDL Programmer to send the response back to the requesting clients. It accepts the Report Name as the parameter. This Report will be used to construct the XML Response, which will be sent back to the Clients.

Syntax

```
[Import File : <Import File Name>]
    Response Report : <Report Name>
```

Where,

<Import File Name> is any Import File Definition Name, for which the appropriate Response needs to be constructed.

<Report Name> is the Name of the Report which is used to construct the XML Response to be sent to the Client.

Example:

```
[Import File : Vouchers]
    Response Report : TallyCustomResponse
```

;; Report to be defined with appropriate XML Tags

In this example, the Response Report 'TallyCustomResponse' needs to be defined, with appropriate XML Tags. However, to update the values like the no. of vouchers created, altered, etc., some platform functions are required, which are explained in the next topic 'Functions Introduced'.

The Response report can also be used for recording additional information and using it at the time of preparation of response. When the Response report is specified, the system will only send the output as created in the Response Report, and hence, the TDL programmer has to make sure that the required output is generated from this Report. If the requestor wants to know what

happened to the individual objects sent, it is now possible to track the same using these Enhancements and send an appropriate response for each.

1.4 Functions Introduced

Function - \$\$HttpInfo

The Function \$\$HttpInfo is used for getting the details of the URL Host, Content-Length and Header information available during the receiving of the SOAP request. This Function accepts two parameters - 'InfoType' and 'Info Sub Type'. Info Type accepts the URL, the Content-Length and the Header. Info Sub Types can appear only for Info Type Header. When Info Type is Header, all the Header details like UNICODE, Connection, Content-Type, etc., can be extracted.

Syntax

```

$$HTTPInfo : <Info Type>
$$HTTPInfo : Header : <Info Sub Type>
    
```

Where,

<Info Type> can be URL, Content-Length or Header.

<Info Sub Type> is necessary only if the Info Type is a Header, and can consist of any Request Header information.

```

Request Header :-
POST / HTTP/1.0
Host: localhost:9000
Content-Type: text/xml; charset=Utf-16
UNICODE: YES
CONTENT-LENGTH: 12512
DISABLELOG: No

Request Data :-
<ENVELOPE>
  <HEADER>
    <TALLYREQUEST>Import Data</TALLYREQUEST>
  </HEADER>
    
```

Figure 1. SOAP Request

Let us see some examples with respect to the SOAP Request as seen in **Figure 1**.

Example:

```

$$HTTPInfo : ContentLength
    
```

From the SOAP request received, the function extracts the ContentLength and returns the value as **12512**.

```

$$HTTPInfo : Header : ContentType
    
```

From the SOAP request received, the function extracts and returns the value of the Header Content-Type, which is **text/xml; charset=Utf-16** in this case.

`$$HTTPInfo : Header : Unicode`

From the SOAP Request received, the function extracts the value of the Header UNICODE, which is **YES** in this case.

The Function `$$HTTPInfo` can be used to retrieve any value from the Header. For example, to retrieve the Mobile Number from an SMS received through 'NatLangQuery' Event, we can use the expression `$$HTTPInfo:Header:MobileNumber`

Function - `$$ImportType`

The Function `$$ImportType` is used to determine the type of Import, i.e., the source of data. The possible Import Types could be 'Sync', 'Migration', 'Remote', 'NatLang', 'SOAP' and 'Manual'.

Syntax

`$$ImportType`

Here, the Function returns the source of data, which could be any one of the above sources.

Function - `$$ImportAction`

This function is used to indicate the status of Import, i.e., whether the current Object was Created, Altered, etc. The possible results are 'Created', 'Altered', 'Ignored', 'Combined', and 'Error'.

Syntax

`$$ImportAction`

Here, the function returns the status of the Import of current object.

Function - `$$LastImportError`

The Function `$$LastImportError` can be used to extract the Import error description for the last object imported, which is helpful to retrieve after every import, and appropriate error logs can be maintained and displayed at the end of the Import Process. In case there is no Error while Importing the current Object, it would return the value as 'Empty'.

Syntax

`$$LastImportError`

Here, the function returns the Import error description for the last object imported.

Function - `$$ImportInfo`

The Function `$$ImportInfo` can be used to extract the details of the Imported Objects in terms of Number of Objects Created, Altered, Ignored, Combined, etc., and Errors encountered. This Function accepts a parameter InfoType.

Syntax

`$$ImportInfo : <Info Type>`

The permissible values for the parameter 'Info Type' are 'Created', 'Altered', 'Ignored', 'Combined' and 'Errors'. This Function returns the Number of Objects Imported as specified by the parameter Info Type. For instance, if the parameter 'Info Type' is specified as 'Created', the function returns the Number of Objects Created during the Import Process and if the Parameter contains Errors, it returns the Number of Errors encountered during the Import Process.

Example:

```
$$ImportInfo : Altered
```

This will return the total number of Objects altered during the Import Process.

2. Events Introduced

2.1 System Events - for Object Deletion and Cancellation

Event Framework has undergone significant changes during the recent past with the introduction of System Events 'System Start', 'System End', 'Load Company', 'Close Company' and 'Timer'.

In this Release, four new System Events viz., two Events for Object Deletion and two for Voucher Cancellation, have been introduced. With the introduction of these Events, whenever an Object is subject to Deletion or Cancellation, these events get triggered, which allows the TDL Programmer to take some appropriate action. Only on confirmation of Deletion or Cancellation, these events are triggered. In other words, only when the user confirms the deletion or cancellation of the object by responding with a YES, the relevant events get triggered.

Irrespective of the Source of Deletion or Cancellation, i.e., from an external XML Request, Tally User Interface, Remote Tally User Interface, etc., the appropriate events get triggered. Any Object deletion or cancellation event gets triggered only at the Server end. Let us understand these System Events in detail.

Delete Object Events – 'Before Delete Object' and 'After Delete Object'

Two new System Events **Before Delete Object** and **After Delete Object** have been introduced in this release. These events get triggered whenever any of the primary Objects defined in Tally Schema is deleted. For example, Object Company, Voucher, Ledger, etc.

As the names suggest, the Events **Before Delete Object** and **After Delete Object** are triggered before and after the deletion of the Object, respectively. The Current Object context would be available in both these Events. Triggering of the Event **After Delete Object** confirms the successful Deletion of the Object.

Syntax

```
[System : Events]
<Label> : Before Delete Object : <Logical Condition Expr> +
                                     : <Action Keyword> : <Action Parameters>
<Label> : After Delete Object  : <Logical Condition Expr> +
                                     : <Action Keyword> : <Action Parameters>
```

Where,

<Label> is a Unique and Meaningful Name assigned to the System Event handler.

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the given Action.

<Action Keyword> is the Action to be taken once the System Event is triggered.

<Action Parameters> are the parameters required for the particular Action.

Both these events are mutually exclusive. In other words, the System Event 'Before Delete Object' need not be necessarily triggered in order to trigger the System Event 'After Delete Object', and vice versa.

Example:

```
[System : Event]

BeforeStockItemDeletion : Before Delete Object : $$IsStockItem+
                        : Call : BeforeDeleteObjectFunc

AfterStockItemDeletion  : After Delete Object  : $$IsStockItem+
                        : Call : AfterDeleteObjectFunc

[Function : BeforeDeleteObjectFunc]

00 : Log : "Before Delete Object Event starts here"

10 : Log : $Name + "under the Stock Group" + $Parent + "is being +
        deleted by "+ $$CmpUserName

20 : Log : $MasterID

30 : Log : "Before Delete Object Event ends here"

[Function : AfterDeleteObjectFunc]

00 : Log: "Stock Item " + $Name + " Deleted"
```

In this example, the events are invoked only when the Stock Item Object is actually being deleted. Since the object context is available both before and after the object deletion, the object details such as Name, MasterID and Parent can be logged in either of the events. The Event 'After Delete Object' confirms the Object Deletion.

Cancel Object Events– Before Cancel Object and After Cancel Object

Similar to Delete Object Events, two new System Events **Before Cancel Object** and **After Cancel Object** have been introduced. Cancellation is applicable only to the Object 'Voucher'. As the names suggest, the Events **Before Cancel Object** and **After Cancel Object** are triggered before and after cancellation the of the 'Voucher' object, respectively. The Current Object context would be available in both these Events. Triggering of the Event **After Cancel Object** confirms the successful Cancellation of the 'Voucher' Object.

Syntax

```
[System : Events]

<Label> : Before Cancel Object : <Logical Condition Expr> +
        : <Action Keyword> : <Action Parameters>

<Label> : After Cancel Object  : <Logical Condition Expr> +
```

: <Action Keyword> : <Action Parameters>

Where,

<Label> is a Unique and Meaningful Name assigned to the System Event handler.

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the given Action.

<Action Keyword> is the Action to be taken once the System Event is triggered.

<Action Parameters> are the parameters required for the particular Action

Both these System Events are mutually exclusive, i.e., the Event 'Before Cancel Object' need not be necessarily triggered in order to trigger the Event 'After Cancel Object', and vice versa.

Example:

```
[System : Event]
```

```
BeforeVchCancellation : Before Cancel Object : Yes : Call +
                        : BeforeCancelObjectFunc

AfterVchCancellation  : After Cancel Object  : Yes : Call +
                        : AfterCancelObjectFunc
```

```
[Function : BeforeCancelObjectFunc]
```

```
Local Formula : StrMasterID: $$String : $MasterID

00 : Log : "Before Cancel Object Event " + @StrMasterID + "starts here"

10 : Log : $VoucherNumber

20 : Log : $VoucherTypeName

30 : Log : "Before Cancel Object Event " + @StrMasterID + "ends here"
```

```
[Function : AfterCancelObjectFunc]
```

```
Local Formula : StrMasterID : $$String : $MasterID

01 : Log : "Voucher with MasterID " + @StrMasterID + " cancelled"
```

In this example, the System Events **Before Cancel Object** and **After Cancel Object** are triggered the moment any voucher is cancelled. Since the object context is available both before and after the cancellation of the object, the details such as Voucher Number, Voucher Type Name, Master ID and Date of the cancelled voucher can be logged through both the user defined functions. The Event **After Cancel Object** confirms the Voucher Object Cancellation.

Points to remember

- The System Events for Object Deletion or Cancellation will be triggered when an Object gets deleted or cancelled from any Source, viz., from an External Third Party Request, from the Tally User Interface or from Remote Tally.
- In case multiple vouchers are selected, and subsequently cancelled or deleted:
 - The Event is triggered as many times as the number of vouchers selected. For instance, if five Vouchers are selected for Deletion in Daybook, the System Events for Deletion would be triggered five Times, once for each Voucher.
 - Only the methods fetched in the Collection used in the Report displaying the list of Vouchers would be available in the Deletion or Cancellation Context. For instance, if multiple Vouchers are selected in Daybook, only the methods fetched in the Collection used in Daybook would be available in the current context. However, the entire Voucher Object context (including all the methods) can be associated by using the Object Association Syntax within the User Defined Function, i.e., **Object: Voucher: "ID:" + (\$\$String:\$MasterID)**, and all the methods will be available in the context.
- In case of Remote Login, when the remote user deletes or cancels an object, these events are triggered at the server.

2.2 Introduction of new Object-Specific Events

We are aware of object-specific events like 'Focus', 'Form Accept', 'Before Print' and 'After Print', which were introduced in previous releases at various User Interface Objects.

In this Release, four new object-specific events viz., the Event **Load** at Report, the Event **Reject** at Form, the Event **Accept** at Field and the Event **After Import Object** at 'Import File' Definition, have been introduced to provide control in the hands of the TDL developer to perform any desired action, whenever these events are triggered.

Let us understand these Object-Specific Events in detail.

Event 'Load' - at Report Definition

The Event **Load** has been introduced at the 'Report' Definition. This event is triggered before the Report is displayed to the user. In other words, when this event is triggered, the Report is constructed and updated with variables, object context and the data. The Event 'Load' provides control to the TDL Layer, before Tally gets into wait loop, where the user will start operating on the Report. It may allow storing of the current state, etc.

Syntax

ON: Load : <Condition Expr> : <Action Keyword> : <Action Parameters>

Where,

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the specified Action.

<Action Keyword> is the Action to be executed, once the System Event is triggered.

<Action Parameters> are the parameters required for the given Action.

Example:

```
[#Report : ProfitandLoss]

    On : Load : @@IsLossIncurred : CALL : ShowMsgifLossEncountered
```

In this example, once the user loads or displays the report **ProfitandLoss**, the 'Load' event invokes the Function **ShowMsgifLossEncountered**, if the System Formula **IsLossIncurred** evaluates to TRUE.

Event 'Reject' - at Form Definition

The Event **Reject** has been introduced at the 'Form' Definition. This event gets executed in the Edit Mode, when the user quits the current Form without accepting or saving it. It allows the Programmer to perform the desired action on rejection of the Form. Once the Event 'Reject' is used at the Form Definition, the default Action 'Form Reject' is overridden, as a result of which, the action **Form Reject** has to be explicitly specified.

Syntax

```
ON : Reject : <Condition Expr> : <Action Keyword> : <Action Parameters>
ON : Reject : <Condition Expr> : Form Reject
```

Where,

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the given Action.

<Action Keyword> is the Action to be taken, once the System Event is triggered.

<Action Parameters> are the parameters required for the given Action.

Example:

```
[#Form : Accounting Voucher]

    On : Reject : Yes : Call : TDLRejectFunc

    On : Reject : Yes : Form Reject

[Function : TDLRejectFunc]

    00 : Msg Box : Status : "The form is rejected"
```

In this example, when the form is rejected, the Function **TDLRejectFunc** is invoked, which displays the message "The form is rejected". Once the execution of the function is over, the form gets rejected due to the explicit Action 'Form Reject'.

Event 'Accept' - at Field Definition

The Event 'Accept' has been introduced at the Field Definition. This event gets executed the moment an editable Field is accepted. Once the Event 'Accept' is used at the Field Definition, the default Action 'Field Accept' is overridden, as a result of which, the action 'Field Accept' to accept the Field contents, has to be explicitly specified.

When a Field is accepted, the event sequence runs in the following order:

1. The Field value is validated through the 'Validate' attribute
2. The 'Modifies' Variable is modified/updated.
3. The Event 'Accept' is executed
4. The Sub-Form is invoked



If the validation fails, then this event will not be executed, as it will result in an error to the user.

Syntax

```
ON : Accept : <Condition Expr> : <Action Keyword> : <Action Parameters>
ON : Accept : <Condition Expr> : Field Accept
```

Where,

<Logical Condition Expr> is a Logical Expression, which when evaluates to TRUE, executes the given Action.

<Action Keyword> is the Action to be taken, once the System Event is triggered.

<Action Parameters> are the parameters required for the given Action.

Example:

```
[#Field : Qty Primary Field]
```

```
On : Accept : @@IsNegativeClosQty : Call : MsgBoxforNegative
```

```
On : Accept : @@IsNegativeClosQty : Field Accept
```

Here, the event **Accept** is triggered on accepting the Field **Qty Primary Field**, which in turn, invokes the function **MsgBoxforNegative**. Once the function terminates, the field is accepted.

Event 'After Import Object' - at Import File Definition

A new Event 'After Import Object' has been introduced to empower the TDL programmers to perform some action every time an Object is imported. This Event helps to create appropriate logs after Importing each Object, or terminate the Import Process based on some condition.



For more details about the Event 'After Import Object', including the syntax and example, refer to the first topic 'Data Importing Enhancements'.

3. Action Enhancements

3.1 Function Actions

Batch Posting Actions - START BATCH POST and END BATCH POST

The introduction of User Defined Functions in TDL has empowered the TDL Programmers with one of the most important needs, i.e., Automation of Data Entry. Irrespective of the data source,

one can make use of User Defined Functions to update the database without much user intervention. However, in comparison to Import of Data through XML/SDF Files, updation of data through User Defined Functions takes a much longer time. A performance lag has been observed while updating data with User Defined Functions, as compared to updating data of a similar size by the Import method. In order to give a uniform user experience and better performance, the approach followed in data import has now been extended to data updation through User Defined Functions.

Importing of Data implicitly follows a Batch Posting approach, wherein the data to be incorporated in the database is updated in batches, thereby improving the performance. This approach has now been extended to data updation through User Defined Functions, with the introduction of two new Actions, namely **START BATCH POST** and **END BATCH POST**. Batch Posting Mode accumulates sets of Objects into batches, and pushes a whole batch of data into the database at a time, which optimizes the performance. Batch Posting Mode also requires the size of the Batch, which can be set as a parameter to the Action **START BATCH POST**.

Actions - **START BATCH POST** and **END BATCH POST**

The Actions **START BATCH POST** and **END BATCH POST** are procedural actions which can only be used within User Defined Functions. These Actions indicate that all the **Data Updation Actions** falling between the Actions **START BATCH POST** and **END BATCH POST** must be updated to the database in Batches. Instead of the data being directly updated to the database, the data is updated to a temporary file, and once the Batch Size limit is reached, the entire data is flushed from the temporary file into the database.

A Batch is a set of objects, wherein the batch size can be specified by the TDL Programmer as an optional parameter to the Action **START BATCH POST**. In the absence of this parameter, the default size of the batch is considered as 100 Objects. On encountering the batch size limit, the temporary file posts/flushes the data into the Tally database. On completion of the posting of the previous batch of data, the subsequent batch posting iteration takes place, and this cycle continues till the entire data is updated to the database. The bigger the batch size, the lesser the total number of batches, and the better the performance. The operation will be accomplished faster if the Batch Size specified is optimal. A very high Batch Size, beyond a particular point, may also deteriorate the performance. Hence, striking the right balance and specifying the optimal batch size is important to achieve the best performance.

Syntax

```
START BATCH POST [:<Batch Size>]
  NEW OBJECT: <Object Type>: <Object Name> : <Forced Update Flag>
    Action 1
    Action 2
    :
    :
    Action n
  SAVE TARGET
END BATCH POST
```

Where,

<Batch Size> is the size of each batch.



In the absence of Action END BATCH POST, the End of the User Defined Function is assumed as the end of Batch Posting. However, it is recommended to specify END BATCH POST, especially in presence of nested loops/ large volume of data.

Example:

```
[Function : Create Ledgers]
```

```
Parameter : pNumberofLedgers : Number : 1000

000      : START BATCH POST : 500

010      : For Range   : i : Number: 1: ##pNumberofLedgers

020      : New Object  : Ledger

030      : Set Value   : Name   : "Customer " + $$String:##i

040      : Set Value   : Parent : "Sundry Debtors"

050      : Create Target

060      : End For

070      : END BATCH POST
```

In this example, when the Function “Create Ledgers” is invoked, 1000 Ledgers, ranging from “Customer 1” to “Customer 1000”, are created under the group ‘Sundry Debtors’. If the Batch Posting feature was not used, the database files would get locked and released for updating every object, and this would continue till all the Objects were updated. Thus, 1000 cycles would take place, affecting the performance adversely. However, with Batch Posting approach, these Objects are updated in 2 batches of 500 Objects each, i.e., the database is locked and released only twice, thus improving the performance vastly. The impact of this enhancement can be observed during updation of data of large volume. For instance, the following table shows the statistics of time taken for creating 1000 ledgers using the given code, but with different Batch Size:

Particulars	Batch Size	Approx. time taken
Without Actions START BATCH POST and END BATCH POST	NA	3 Minutes 29 Seconds
With Actions START BATCH POST and END BATCH POST	100	4 Seconds
	500	2 Seconds
	1000	1 Second

Table 1. Batch Posting Statistics

As seen in the table, with the usage of the Actions START BATCH POST and END BATCH POST, the time taken reduces significantly. Also, it is noticed that with increase in the Batch Size, the performance keeps improving. The operation will be accomplished faster if the specified Batch Size is equal to the total number of Objects being updated, i.e., in this case, specifying the Batch Size as 1000 will give optimal performance as 1000 objects will be posted to the database at a time.

Limitation

- Greater the Batch Size, better the performance of data updation. However, the database files are locked from the beginning till the end of the write operation of each Batch. This results in unavailability of the Database for data updation from other sources executing parallelly, like data entry in a Multi-User Environment, Data Synchronization, etc. Thus, a higher value of Batch Size will improve the performance, but will slow down the simultaneous updation of Data. Hence, determining the Optimal Batch Size is necessary to strike the right balance for getting the best performance. A very high Batch Size, beyond a particular point, may deteriorate the performance.

Points to remember

- Actions START BATCH POST and END BATCH POST are not be used in a nested manner. If used so, only the first instance is considered, while the remaining ones are ignored.
- A Batch can only be ended or closed in a Function where it is initiated. If the user misses the closing part, the System will END the batch implicitly.
- One must **backup** the data prior to using the Batch Posting Feature, as incorrect usage of the same may lead to corruption/loss of data.
- For Import of Data also, we have a parameter similar to Batch Size, i.e., **Import Batch Size**, which can be used to improve the performance while Importing Data. This parameter can be specified in Tally.INI, in the absence of which, it is set to 100 by default. Setting this parameter to a higher value will decrease the time taken for Import, but will increase the wait time for other simultaneous data updation operations. Setting it to -1 will disable the Batch Posting feature for Import of Data, which means that the Data Import would consume a longer time, as every Object would need to be updated to the database individually.

Asynchronous Message Box Actions - START MSG BOX and END MSG BOX

These are asynchronous message boxes, i.e., the message box continues to appear till the action 'End Msg Box' is encountered or the function is terminated, whichever is earlier. Unlike action 'Msg Box', this action is executed asynchronously, i.e., it does not expect a key press from user. When executed, it displays the message box, and continues to execute the subsequent Actions.

Syntax

```

Start MSG BOX : <Title Expression> : <Message Expression>
               <Action 1>
               :
               <Action n>
End MSG BOX

```

Where,

<Title Expression> is the value that is displayed on the title bar of the message window.

<Message Expression> is the actual message displayed in the box. It can be an expression as well, i.e., the variable values can be concatenated and displayed in the display area of the box.

Example:

```
[Function : MsgBox Actions]
```

```
Variable : Counter      : Number
```

```
Variable : TotalCount  : Number : 100
```

```
Returns  : Number
```

```
Local Formula : StrTotalCount : ($$String : ##TotalCount)
```

```
Local Formula : StCounter      : ($$String : ##Counter)
```

```
00 : Start Msg Box: Status : "This Function creates" +
```

```
      + @StrTotalCount + "Ledgers"
```

```
10 : Start Progress : ##TotalCount : ##SVCCurrentCompany : +
```

```
      "Creating Ledgers" : "Please wait"
```

```
20 : While : ##Counter < ##TotalCount
```

```
30 :           New Object      : Ledger : "Ledger" + @StrCounter : Yes
```

```
40 :           Set Value : Name   : "Ledger" + @StrCounter
```

```
50 :           Set Value : Parent : "Sundry Debtors"
```

```
60 :           Save Target
```

```
70 :           Increment      : Counter
```

```
80 :           Show Progress : ##Counter
```

```
90 :           End While
```

```
100 :           End Progress
```

```
110 : End Msg Box
```

Here, the action 'Start Msg Box' invokes the message box and retains it till action 'End Msg Box' is encountered. Thus, the message box will continue to appear from label 00 to 110. In absence of 'End Msg Box', the msg box is automatically terminated when the Function **MsgBox Actions** ends.



*If nested **Start Msg Box** is executed, then the previous Message box is overwritten.
At any time, only one Message Box can be displayed.*

3.2 System Actions

Action - Load TDL

An Action 'Execute TDL' was introduced in Release 3.6 to load a TDL dynamically, execute some action, and then unload the TDL or keep it, depending on the Logical Value. With this Action, the TDL would get loaded. However, the execution of action was mandatory. In Release 4.8, an action 'Load TDL' has been introduced to load the TDL/TCP dynamically for only the current session of Tally. However, if the TDL File is already loaded due to being specified in Tally.INI, or through previous execution of the Action 'Load TDL'/'Execute TDL', it will not be loaded again. On closing the current session of Tally.ERP 9, the dynamically loaded file(s) will not be available for the subsequent Tally Session.

Syntax

```
LOAD TDL : <TDL/TCP File Path Expression>
```

Where,

<TDL/TCP File Path Expression> evaluates to the path of the TDL/TCP File to be loaded dynamically.

Example:

```
[Button : Load Dynamic TDL]
```

```
Key      : Alt + L
```

```
Action : Load TDL : @@TDLFilePath
```

```
[System : Formula]
```

```
TDLFilePath : "C:\Tally.ERP9\TDL\Samples.tcp"
```

In this example, on triggering the button 'Load Dynamic TDL', the action 'Load TDL' loads the TDL from "C:\Tally.ERP9\TDL\Samples.tcp". If this TDL is already loaded due to being specified in Tally.INI, or previous execution of the Action 'Load TDL', then the same will not be loaded again.



Local TDLs will be loaded at the Remote End if 'Allow Local TDLs' is enabled to the user logged in.

Action - Unload TDL

To unload the TDL dynamically from the current Tally Session, the Action 'Unload TDL' has been introduced. With this Action, the local TDL File(s), including the ones added through Tally.ini and those added dynamically using Actions 'Load TDL' or 'Execute TDL', can be unloaded. However, they would be unloaded only for the current Tally Session, and in the subsequent session, all the TDL/TCP files specified in Tally.INI will be loaded once again. Using this action, the Files can be unloaded by specifying either the TDL /TCP file name or the GUID of the TCP File.

Syntax

```
UNLOAD TDL : <TDL/TCP File Path Expression or GUID Expression>
```

Where,

<TDL/TCP File Path Expression or GUID Expression> evaluates to the path of the TDL/TCP File or GUID of the TCP File to be unloaded dynamically.

Example:

```
[Button : Unload Dynamic TDL]

    Key      : Alt + U

    Action   : Unload TDL : @@TCPFileGUID

[System : Formula]

    TCPFileGUID : "c2901088-349b-434b-946c-9ada601fd6b7"
```

In this example, on triggering the button 'Unload Dynamic TDL', the Action 'Unload TDL' unloads the Compiled TDL with the GUID "c2901088-349b-434b-946c-9ada601fd6b7". If the particular TDL is not found to be loaded, then the same is ignored. If the TCP File was dynamically loaded, then the same is removed from the List of TDL Files. However, if the TCP File was available in Tally.INI, then the same is removed temporarily and reloaded in the subsequent session of Tally.



- ❑ Account/Remote TDL file(s) cannot be unloaded using Action 'Unload TDL'.
- ❑ Once a TDL is unloaded explicitly, if one attempts to load such TDL file(s) by changing the TDL Configuration, the file(s) will not be loaded in that session.

4. Function Enhancements

4.1 Function - \$\$IsTDLLoaded

A new function 'IsTDLLoaded' has been introduced to check if a particular TDL is already loaded. This function returns TRUE if the particular TDL/TCP file is already loaded, and FALSE if it is not.

Based on the result of this function, further actions like Loading and Unloading of TDL, or executing a Report from the dynamically loaded TDL File, etc., can be performed.

Syntax

```
$$IsTDLLoaded : <TDL/TCP File Path Expression or GUID Expression>
```

Where,

<TDL/TCP File Path Expression or GUID Expression> evaluates to the path of the TDL/TCP File or GUID of the TCP File to be checked, whether it is loaded or not.

Example:

```
[Function : Display First TDL Report]

    00 : If          : $$IsTDLLoaded : @@TCPFileGUID

    10 : Display     : First TDL Report

    20 : Unload TDL : @@TCPFileGUID
```

```
30 : End If
```

```
[System : Formula]
```

```
TCPFileGUID : "c2901088-349b-434b-946c-9ada601fd6b7"
```

In this example, if the TDL with GUID "c2901088-349b-434b-946c-9ada601fd6b7" is loaded, then the Report 'First TDL Report' will get displayed. Subsequently, the TDL is unloaded.

4.2 Function - \$\$HttpInfo

It is used to get the details of URL Host, ContentLength and Header information available during the receiving of the SOAP request. It accepts two parameters - 'InfoType' and 'Info Sub Type'.

4.3 Function - \$\$ImportType

The Function \$\$ImportType is used to determine the type of Import, i.e., the source of data. The possible Import Types could be 'Sync', 'Migration', 'Remote', 'NatLang', 'SOAP' or 'Manual'.

4.4 Function - \$\$ImportAction

This function is used to indicate the status of Import, i.e., whether the current Object was Created, Altered, etc. The possible results are 'Created', 'Altered', 'Ignored', 'Combined', and 'Error'.

4.5 Function - \$\$LastImportError

The Function \$\$LastImportError can be used to extract the Import error description for the last object imported, which is helpful to retrieve after every import, and appropriate error logs can be maintained and displayed at the end of the Import Process. In case there is no Error while Importing the current Object, it would return the value as 'Empty'.

4.6 Function - \$\$ImportInfo

The Function \$\$ImportInfo is useful to extract the details of the Imported Objects in terms of Number of Objects Created, Altered, Ignored, Combined, etc., and Errors encountered. This Function accepts a parameter 'InfoType'.



For details of the functions \$\$HttpInfo, \$\$ImportType, \$\$ImportAction, \$\$LastImportError and \$\$ImportInfo, refer to the section 'Data Importing Enhancements'.

5. New Objects and Collection Attributes to support Banking

To support Banking, two new Primary Objects, i.e., objects of Type 'Pay Link' and 'Party Pay Link' have been introduced. These Objects are similar to Bills, i.e., with each transaction by individual payment modes like 'Cheque', 'Inter-Branch Transfers', etc., a 'Pay Link' object is created with respect to Bank and a 'Party Pay Link' object is created with respect to transaction to Party.

Along with the Object Types 'Pay Link' and 'Party Pay Link', three new attributes have been introduced for the 'Collection' definition. These attributes can be used for accessing the appropriate indexes within the collection to fetch the data. They are primarily used in the banking module.

5.1 Collection Attribute - Transaction Type

This attribute filters the transactions based on the Link master Transaction type. It accepts SysName as parameter to identify the type of banking transaction, e.g., 'Cheque', 'Credit Card', 'Inter-Bank Transfers', etc. It is applicable only for collections of Type 'Pay Link'/'Party Pay Link'.

Syntax

```
[Collection: <Collection Name>]
    Transaction Type: $$SysName:<Type Of Transaction>
```

Where,

<Type of transaction> is the link master transaction type used in banking. It can be Cash, Cheque, ATM, ECS, Cheque/DD, Interbank Transfer, Same Bank Transfer, Electronic Cheque and Electronic DD/PO.

5.2 Collection Attribute - Primary Status

It filters the transactions based on the (Primary) Status of Banking transactions stored in Link Master against each transaction. It accepts SysName as parameter, such as 'Returned', 'Cancelled', etc. It is applicable only for Collections of Type 'Pay Link' or 'Party Pay Link'.

Syntax

```
[Collection : <Collection Name>]
    Primary Status : $$SysName:<Status Parameter>
```

Where,

<Status Parameter> is the primary status of each transaction. It can be 'StatusTransacted', 'Unstable', 'Cancelled' and 'Returned'.

5.3 Collection Attribute - Secondary Status

This attribute filters the transactions based on the additional status of each transaction type. It accepts SysName as Parameter, such as 'Exported', 'Approved', 'Not Approved', etc. This is applicable only for Collections of Type 'Pay Link'.

Syntax

```
[Collection : <Collection Name>]
    Secondary Status : $$SysName : <Status Parameter>
```

Where,

<Status Parameter> is the secondary status of each transaction. It can be 'Exported', 'Approved', 'Not approved' or 'Primary'.

Example:

```
[Collection : Pay Link Coll]

Type           : Pay Link

Child of       : "ICICI"

Transaction Type : $$SysName:NEFT

Primary Status  : $$SysName:StatusTransacted
```

```
Secondary Status : $$SysName:NotApproved
```

```
Fetch           : *
```

Here, the Collection **Pay Link Coll** consists of Objects of Type **Pay Link**. The transaction type filters the collection of Type 'Pay Links' for transactions only of type NEFT, that are filtered with primary status as 'Status Transacted' and secondary status as 'Not Approved'.

6. Miscellaneous Enhancements

6.1 HTTP Log Changes

Hyper Text Transfer Protocol, commonly referred to as HTTP, is a communication protocol which is used to deliver data on the World Wide Web. In other words, HTTP provides a standardized way for computers to communicate with each other. HTTP specification includes information about how the client's request data will be constructed, how it will be sent to the server and how the server should respond to these requests.

Tally.ERP 9, being the complete business solution, also supports HTTP Protocol for exchanging messages between Tally and any Third Party Application. During interactions with the Third Party Applications, Tally.ERP 9 can act as a Client, as a Server or both.

As a Client, Tally.ERP 9 constructs appropriate requests in XML Format, sends the same to the specified URL Host, i.e., the Server, which could be any Third Party Application, including Tally. It then receives the response from the Server over HTTP. As a Server, Tally acts as per the incoming request from any Third Party Client, generates an XML Response and sends it to the Client. Usually, except the default Tally-to-Tally Data Synchronization, the Integration solutions are built by the TDL Programmer. While building such a solution, they require some debugging tools like logs to confirm if the right requests and responses are being communicated over HTTP.

Enable HTTP Logs in Developer Mode

A Configurable option is provided in Tally to log all the Information exchanged over HTTP, which if enabled, records the Request/Response communication between Tally and external applications. This helps the developers to debug and resolve the issues, if any, while setting up integrated solutions. HTTP Log information would be written to the file TallyHTTP.Log in the working Tally Application Directory. Previously, though this log was mainly used for development and debugging purposes, it was also available in the **Normal Mode** (Release Mode). Due to this, HTTP Requests/Responses would be unnecessarily logged for the users of Tally.ERP 9, who were usually not even consuming those logs. Also, it would needlessly consume some amount of time to update the HTTP Log on each occasion. Hence, from Release 4.8 onwards, 'Enable HTTP Log' Option has been removed from the Normal Mode of Tally.ERP 9, and will be available only in the **Developer Mode**. In other words, the Tally.ERP 9 users cannot see and enable HTTP Logs through the Advanced Configuration Screen (F12) in Normal Mode.

However, for development and debugging purposes, the same can be seen and enabled by the Developers/Integrators, while running Tally.ERP 9 in Developer Mode.



Tally Application can be executed in Developer Mode, using the parameter DevMode. For instance, if the Tally Application is installed in C:\Tally.ERP 9, then it can be executed in Developer Mode as C:\Tally.ERP9\Tally.Exe /DevMode.

Silent HTTP Exchange

At times, programmers would not want to log certain sensitive HTTP Information, inspite of HTTP Logs being enabled in Developer Mode. In order to explicitly control the Logs, the requestor can additionally send a Header DISABLELOG set to YES, which will disable the current log, irrespective of the Configuration or the Mode in which Tally is running. The default value is NO, i.e., in the absence of this Header, the HTTP Information will be logged to the file TallyHTTP.log, if Tally.ERP 9 is working in Developer Mode and HTTP Log is enabled in Configuration.

```
Request Header :-
POST / HTTP/1.0
Host: localhost:9000
Content-Type: text/xml; charset=utf-16
UNICODE: YES
CONTENT-LENGTH: 12512
DISABLELOG: No

Request Data :-
<ENVELOPE>
<HEADER>
<TALLYREQUEST>Import Data</TALLYREQUEST>
</HEADER>
```

Figure 2 Silent HTTP Exchange

In the above figure, we can see that the Header **DisableLog** is set to **NO**, due to which the information exchange between Tally and external application has been logged. If **DisableLog** would have been set to **YES**, then irrespective of Tally being run in Developer Mode and 'Enable HTTP Log' option being enabled, the HTTP information would not be logged to TallyHTTP.log.

6.2 Retrieving Original UDF Index No. of a Data within Associated Objects

A UDF can be used to store additional information into the Tally database. UDFs are stored in the current object context. Whenever a UDF is created and used in an already existing report, the data is stored in the context of the current object, i.e., it is always associated to the object to which the report is associated.

Previously, if the TDL or the TCP was lost or corrupted, then there was no way by which we could know the UDF details like the UDF Number, and hence, the retrieval of data related to the UDF was quite difficult. In the present Release, an XML attribute **Index**, within the UDF 'List Tag', has been introduced to help retrieve the original UDF number corresponding to the data available within the Objects associated with it. This UDF number will be available in the Index attribute in

the UDF List Tag, even when the TDL is not attached or is unavailable. The **Index** attribute will be available for Simple as well as Aggregate UDFs.

Example:

```
<UDF:TESTUDFNO.LIST DESC=" `Test UDF No' "ISLIST="YES" TYPE="String"
INDEX="1010">
```

```
<UDF:TESTUDFNO DESC=" `Test UDF No' ">Raam</UDF:TESTUDFNO></UDF:TESTUDFNO.LIST>
```

Here, the UDF number (1010) is displayed under the 'Index' Attribute in the UDF List Tag.

6.3 Date Limit Extended and Prefix Century Behaviour Introduced

- Till now, the Date data type used to support values from 1-1-1901 to 1-1-2099. From this release onwards, the support has been extended to the date 31-12-9999.
- Also, the following prefix century behaviour has been introduced (when a 2 digit year value is entered by the user) in the Date field. The century value prefixed by the system will depend upon (i) the 2-digit year value passed by the user (ii) the system year. Thus, the different cases that are possible are as follows.

	Case I: The year entered by the user is greater than or equal to 50	Case II: The year entered by the user is less than 50
When the current year as per the system date is greater than or equal to 50	In this case, the system will consider the year entered to be belonging to the same century as the system date. Example: If the System Date is 1-1- 2098, then if the user enters 86, it will be considered as 2086, i.e., of the same century as in system year.	The system will consider the year entered to be belonging to the next century compared to the system date. Example: If the System Date is 1-1- 2098, then if the user enters 24, it will be considered as 2124, i.e., of the next century compared to system yr.
When the current year as per the system date is lesser than 50	The system will consider the year entered to belong to the previous century compared to system date. Example: If the System Date is 1-1-2014, and the user enters the year as 86; the system will consider it as 1986, i.e., of the previous century.	The system will consider the year entered by the user to belong to the same century as the system date. Example: If the System Date is 1-1-2014, then if user enters 24; it will be considered as 2024 i.e. of the same century as system year.

What's New in Release 4.7

Following are the highlights of language enhancements in this Release, which have been discussed ahead in detail:

1. In Developer Mode, enhancements like creation of log files in **Excel** format, enriched Key Recording/Playback, and introduction of new Calculator Pane Commands, have been made.
2. Event **NatLangQuery** has been introduced to pass control to the TDL program, when a request is received by way of SMS or Calculator Pane commands.
3. **Zip/Unzip** Capability has been introduced, which allows compression/ decompression of files. Password protection is also supported.
4. Editing capability has been extended to Columnar Reports.
5. New data types **Date**, **DateTime** and **Duration** have been introduced, which along with the existing data types 'Date' and 'Due Date', are now collectively called as Calendar data types.

1. Developer Mode Enhancements

In market, there are customers using Tally.ERP 9 for their day-to-day operations and also developers or partners who build solutions for the customers. To empower the developers of Tally with various tools to build solutions efficiently, a new capability to operate Tally in the Developer Mode was introduced in Release 4.6. In other words, capability was introduced to operate Tally in two modes, viz., **Normal Mode**, to be used by the End Users and **Developer Mode**, to be used by the TDL Developers. Through the Developer mode, a host of Developer Tools have been offered for debugging the code, optimizing the performance of customized reports, recording the user operations and playing them back, etc.

To further enrich the Developer Mode experience, following enhancements have been incorporated in this release.

1.1 Output Profiler and Expression Diagnostics Information in Excel Format

Microsoft Excel is widely used in organizations for representing their Tabular data, as it comes with a galaxy of features for data representation. It is, therefore, ideal to output any tabular data in Microsoft Excel format, for easier and better data analysis.

From Release 4.7, the tabular Profiler or Debugger Expression data will dump the information in **Excel Format**, to enable the developer to analyze the data with ease. The behaviour will be:

- If the System has MS Office 2007 or below, then the output format will be '.xls'.
- If the System has MS Office 2010 or above, then the output format will be '.xlsx'.
- If the System does not have MS Office installed, then the behaviour will be the same as in Release 4.6, i.e., the format will be text files.

Thus, the developer can make use of Excel features like **sorting, filtering, graphical representation, etc.**, thereby attaining performance optimization quickly. For example, by sorting

the Profiled Information in descending order of Time or Count, the developer can quickly determine the artefact that has taken the longest time or the Collection that has been needlessly gathered multiple times, thus **optimizing the performance** quickly.

1.2 Key Recording and Playback Changes

With respect to Key Recording and Playback, following enhancements have been done:

Reading Capacity Increased

In **Release 4.6**, when a Macro gets recorded beyond 2000 characters, it is not possible to read back the characters beyond 2000 using the Function `$$FileRead`, and play them.

From **Release 4.7** onwards, the reading capacity of the Function `$$FileRead` has been increased to 4000 characters per line.

Splitting of Macros

Again, if the Macro gets recorded beyond 4000 characters, the Function `$$FileRead` cannot read further, which means that Keys beyond 4000 characters cannot be played back.

Hence, the Macro Recording capability has been enhanced in such a way that after recording 4000 characters, a new macro gets created automatically, and the subsequent keys are recorded in the new macro. The new macro will bear the same macro name, with a number concatenated to it. All the Macros created will be dumped in a single file, and replayed when required. Thus, **n** number of macros can be recorded in a file, read and played back.

For example, if the Name of the Macro is **Testing**, then after every 4000 characters, a new Macro is created with the Name **Testing-1**, **Testing-2**,...**Testing-n**. Thus, there is no limit to the number of characters which can be recorded, read and played back.

Action 'Dump Recording' with a File Name

In **Release 4.6**, using the Action **Dump Recording**, all the Keys, along with the Macro Name, were written to a file **Macros.Log**, by default.

From **Release 4.7** onwards, the Action 'Dump Recording' has been enhanced to write the macros to a File specified by the user. The Action 'Dump Recording' will accept 2 optional parameters, viz. **File Name** and **Separator**.

- If the **File Name** is left unspecified, then by default, the Action would dump the recording to the file **Macros.log**.
- If the **Separator** is left unspecified, then by default, the system would consider Tilde (~) as the default separator.

Syntax

```
Dump Recording [: <File Name> [: <Separator between keys>]]
```

Where,

<File Name> is the name of the file where macro and keys will be recorded.

<Separator between keys> is the separator used to differentiate the macro name from the keys.

Example:

```
[Button : Dump Recording]
    Title : "Dump"
```

Key : Alt + U

Action : Dump Recording : "BSView.txt" : "-"

On clicking the Button 'Dump Recording', a file **BSView.txt** is created in Tally Application Folder.



*The behaviour of Calculator Pane Command **Dump** is retained as in Release 4.6, i.e., it will create a file with the name **Macros.Log** in the Tally Application Folder.*

Action 'Trigger Key' Enhanced

When the macro keys are recorded using Key Recording Actions or when they are dumped into the Macros File from the Calculator Pane; in order to play them back, one needs to make use of the Action **Trigger Key**, which sends a list of keys in sequence to the system as if an operator is pressing those Keys.

Very often, some keys like **Enter, Up, Down, etc.**, are repeated in sequence more than once. For example, to scroll down to the 7th Voucher in Daybook, one needs to trigger the **Down** Key 6 times. Similarly, in an Invoice Entry, moving to the first Item field needs multiple hits of **Enter** key. Hence, the action 'Trigger Key' has now been enhanced to support the **<Key>:<Number>** combination in the syntax, which will trigger the particular **Key** for the specified **Number** of times.

Example: 1

Trigger Key : DD, Enter : 5, "Item 1", Enter

This is the same as:

Trigger Key : DD, Enter, Enter, Enter, Enter, Enter, "Item 1", Enter

Following happens when the above Action is invoked from Gateway of Tally:

- ❑ The First **D** navigates us to **Display**.
- ❑ The Subsequent **D** navigates us to **Daybook**.
- ❑ **Enter:5** triggers the **Enter** Key 5 times, i.e., Drills down into the current voucher, accepts 4 subsequent non-skipped fields, and moves to the 5th non-skipped Field in the Voucher.
- ❑ In the Fifth field, the text **Item 1** is entered.
- ❑ The Subsequent **Enter** then accepts the current field, and the focus is shifted to the next non-skipped field.

Example: 2

Trigger Key : DS, Enter:3

This action will take us through Display (**D**) -> Statement of Accounts (**S**) -> Outstandings (**1st Enter**) -> Receivables (**2nd Enter**); the **3rd Enter** selecting the first item in the list and displaying all the outstanding bills within it.

1.3 Calculator Pane Changes

For the convenience of the developer, certain calculator pane commands have been introduced in this Release.

□ **Command - Mode: ?**

Mode: ? will list all the modes available in the Calculator Panel.

```

Calculator
2> mode:?
3  Mode reset
4  Modes available
5  Profile ..... To enter the profiler commands
6  Debug ..... To enter the expression debugger commands
7  Recrd ..... To enter the recorder commands
    
```

Figure 1. Command **Mode: ?**

□ **Command - Help**

From any of the modes, **Help** command will list down all the supported commands for the particular mode, along with their purpose.

```

Calculator
29> Mode:Record
30  Mode set to Recording
31> help
32  start:MacroName..... starts recording of Macro
33  pause..... Pauses the macro recording
34  resume..... Resumes recording of Paused Macro
35  stop..... stops the macro recording
    
```

Figure 2. Command **Help**

□ **Command - Open**

From any of the modes, the **Open** command will open the most recently logged file. Consider the following examples:

- If the current mode is **Debug**, the file being opened will be either debug.xlsx, debug(1).xlsx, debug(2).xlsx, or debug(<n>).xlsx, whichever is the most recent in debugger log. (Depending on the available Excel Version, the file extensions will vary.)
- If the current mode is **Profile**, the file being opened will be either tdlprof.xlsx, tdlprof(1).xlsx, tdlprof(2).xlsx, or tdlprof(<n>).xlsx, whichever is the most recent in profiler log. (Depending on the available Excel Version, the file extensions will vary.)
- If the current mode is **Record**, the file being opened will be **macros.log**.

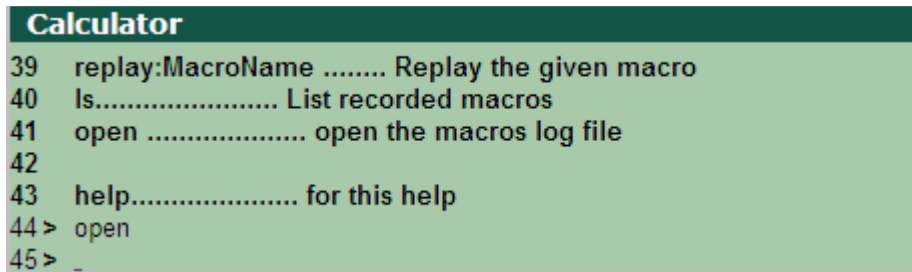


Figure 3. Command **Open**

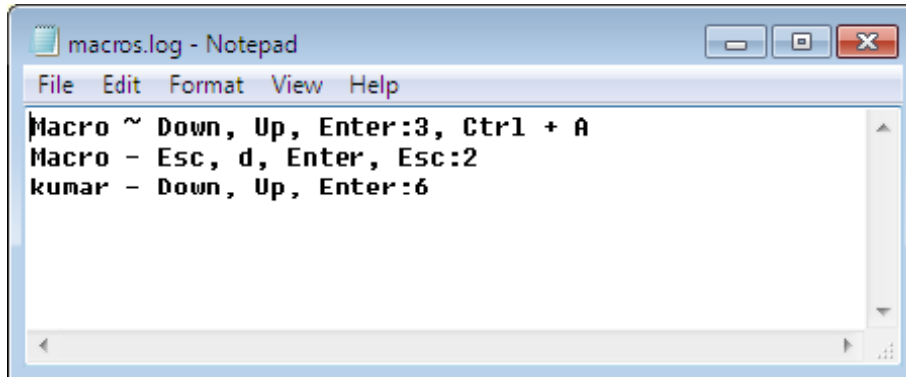


Figure 4. Macros.log File

2. Event ‘NatLangQuery’ Introduced

As we are already aware, Tally has a natural language processing capability which accepts queries either from the Calculator Pane or from SMS Request. Tally has the intelligence of parsing received/ given commands, in order to process the same. This parsed information is used by the system to process the query and deliver the result. However, in certain cases, queries received might not be understood by the system. There have also been requirements in the market to support data updation queries like Ledger, Voucher Creation, etc.

In order to cater to the above requirements, a new System Event **NatLangQuery** has been introduced. This event gives complete control in the hands of the TDL Developer, thereby enabling him to process the query received and do the needful. If the query is ignored by TDL, then the System continues to process it and provide the response as usual.

Syntax

```
[System : Event]
    <Event Name> : NatLangQuery : <Condition> : <Action> : <Action
        Parameters>
```

Where,

<Event Name> can be any unique name, indicating the purpose of the event.

<Condition> if evaluated to TRUE on receiving a query, an action will be executed.

<Action> is the Action Call which can be used for processing the Query received/given.

<Action Parameters> can be a Function Name and its required Parameters.

Example:

```
[System: Event]
```

```
Ledger Creation: NatLangQuery: @@IsLedgerinQuery: Call: Create Ledger
```

Whenever a Query is received, Tally checks the logical condition @@IsLedgerinQuery. If it evaluates to TRUE, then the function 'Create Ledger' is invoked.



*In the given syntax, <Action> can be any global Action like Display, Alter, Print, etc. However, **NatLangQuery** being a query from a remote location, it is not advisable to populate any report in the User Interface of Tally at the Server end.*

In order to support the event "NatLangQuery", the following System Variables, along with a Built-in TDL Function \$\$NatLangInfo, have been introduced.

2.1 System Variables Introduced**SVNatLangFullRequest**

This is a String Variable bearing the complete request / query string when a Query is received.

SVNatLangRequest

This is a String Variable bearing the part of the query that is not understood by the system.

SVNatLangResponse

This is a String Variable which carries the response back to the requestor. After processing the Query, this variable needs to be set with an appropriate response that is sent back to the requestor by Tally.

SVNatLangRequestProcessed

This is a Logical Variable which denotes the status of the Query Processed. After processing the Query, this variable needs to be set to YES to indicate if the Query was processed by TDL successfully.

- Update **YES** to indicate that the Query is processed by TDL and the response is prepared.
- Update **NO** to indicate that the Query is not processed / understood by TDL.

If updated as YES, then only the system will send the response specified in TDL. If updated as NO, the System processes the Query and sends the appropriate response.

2.2 Built-In TDL Function '\$\$NatLangInfo' Introduced

A Built-In TDL Function **\$\$NatLangInfo** has been introduced to provide certain information like Company Name, User Name, From Date, To Date, etc., from the Query received.

Syntax

```
$$NatLangInfo : <InfoType>
```

The valid values for <InfoType> are as follows:

- Company
- UserName

- FromDate
- ToDate
- ObjectName

Example:

\$\$NatLangInfo:Company returns the current Company Name.

\$\$NatLangInfo:UserName returns the Name of the User from whom the Query is received.

- If the request is from Calculator Pane, Tally responds with the current logged in user name.
- If the request is initiated through an SMS Query, Tally responds with the name of the Tally.NET User from whose device, the request is received.

\$\$NatLangInfo:FromDate returns the From Date of the period, based on the received query or from the recent context.

\$\$NatLangInfo:ToDate returns the To Date of the period, based on the received query or from the recent context.

\$\$NatLangInfo:ObjectName returns the Name of the Object, based on the received query or in the recent context. For example, if Query received is – “Sales for April 2013”, then NatLangInfo will return the following:

Object Name- Sales

From Date - 1-April-2013

To Date - 30-April-2013

2.3 Example to Demonstrate “NatLangQuery” event

Create Ledger RadheShyam SundryDebtors

Requirement

The requirement here is to create a Ledger with the name ‘RadheShyam’, under the Group ‘Sundry Debtors’.

Implementation

1. Firstly, we need to write a system event to trap the query received and perform the necessary action.

```
[System : Event]
```

```
    Create Ledger : NatLangQuery : @@IsCreateLedger : Call : Create Ledger
```

2. If the Request contains ‘Create Ledger’, then only the above action is to be performed, so the following system formula needs to be declared to check the value of the variable **SVNatLangFullRequest**.

```
[System : Formula]
```

```
    IsCreateLedger : ##SVNatLangFullRequest CONTAINS "Create Ledger"
```

3. When the action is being executed, firstly we need to tokenize the words from the string:

```
[Function : TokenizeQuery]
```

```
    000 : List Delete Ex : SMSStrings
```

```

010 : For Token      : TokenVar    : ##SVQueryRequest : " "
020 : List Add Ex    : SMSStrings  : ##TokenVar
030 : End For

[System : Variable]

List Var : SMSStrings : String

```

4. Finally, create the ledger with the Action 'New Object'

```

[Function : CreateLedger]

000 : Call: TokenizeQuery
010 : New Object : Ledger : ##SMSStrings[3] : Yes
020 : Set Value  : Name   : ##SMSStrings[3]
030 : Set Value  : Parent : ##SMSStrings[4]
040 : Save Target

```

5. Send the Response to the source of the query

```

050 : Set : SVNatLangResponse : "Ledger Created"
060 : Set : SVNatLangRequestProcessed : Yes

```

Similarly, the **NatLangQuery** Event can also be used to customize the interpretation of the queries being sent, and act accordingly.



- *We have used Space as a delimiter for Tokenizing Query Strings in the given example. One can specify any delimiter like Inverted quotes, comma, etc., to separate different strings.*
- *If customized with the help of Tokens, the Query Signature must be retained exactly in the same order and with the delimiters specified. In the given example, if the Query '**Create Ledger Keshav under SundryDebtors**' is specified, the code will fail, as the function expects the 4th Word to be a Group Name. Hence, the Programmers must communicate the Query Signature clearly with the end users.*

3. ZIP - UNZIP

ZIP is an archive file format that supports compression of data without any loss. A Zipped file may contain one or more files or folders in compressed form. The ZIP file format permits a number of compression algorithms. Originally created in 1989 by Phil Katz, the ZIP format is now supported by a number of software utilities.

The need for supporting this format in Tally.ERP 9 has been felt in various offline Integration projects. Data Exchange takes place between branches and their Head Offices, Distributors and the Principal Companies, etc., where the Head Offices/ Principal Companies having Tally or any

other ERP would require the data from Branches/ Distributors for performance visibility. Usually, Principal Companies require the Item-wise Sales Information of the distributors, which helps them in planning their Stocks.

For integration purpose, Head Offices/ Principal Companies generally get Tally.ERP 9 installed at Branches/ Distributors' locations. The day-to-day Transactions like Sales, Purchase Orders, etc., are then exported from Tally.ERP 9 and integrated by copying the appropriate XML files to FTP, which is consumed by the Head Offices/ Principal Companies.

At locations where the volume of transactions is large, the XML File becomes too bulky to upload to FTP, and subsequently, downloading from FTP takes a long time, thereby causing performance issues. Hence, zipping the file before uploading to FTP was necessary. This would save time both while uploading the File and while downloading it at the other end. Hence, the concept of Compression, i.e., Zip-Unzip has been introduced in Tally.ERP 9.

Along with the actions for Zip/Unzip capability, wildcards * and ? are also supported, as a part of folder/file specification. Asterisk(*) represents zero or more characters in a string of characters. For example, **t*.doc** considers all files starting with 't', bearing the extension .doc, e.g., Tally.doc, Tallyzip.doc, etc. Question Mark (?) represents any one character. For example, **TDLDebug?.*** considers all the files starting with 'TDLDebug', followed by any variable single character, and bearing any extension, e.g., TDLDebug1.xlsx, TDLDebug2.xlsx, TDLDebug1.log, etc.

3.1 ZIP Action

Zip action can be used to archive a set of folders/ files.

□ System Action - ZIP

System action **Zip** is useful when a single File or Folder Source needs to be zipped into a Target file.

Syntax

```
Zip : <Target File> : <Source Path> [:<Password> [:<Overwrite>
      [:<Include Sub-directory> [:<Show Progress Bar>]]]]
```

Where,

<Target File> is the name of the Zip File to be created, along with the Path.

<Source Path> is the path of the Folder/File(s), which is to be zipped. It can be a folder, or a file path (with or without wildcard characters).

<Password> is the password assigned to the Target zipped file.

<Overwrite> is the Logical Flag to specify the behaviour of the Action, if the Target File already exists. If **YES** is specified, the file will be overwritten. If **NO** is specified, the file will not be overwritten, and will remain as it is. The default value is **NO**.

<Include Sub-directory> is the Logical Flag to specify whether to include sub directories available in the specified source path or not. If the Source Path ends with a Folder, the entire Folder along with its Sub-Folders will be zipped, irrespective of this parameter. If the Source Path ends with a File Name Pattern, i.e., with wild cards, this parameter will be considered. If a **YES** is specified, all the files/sub-folders matching the wild card pattern will be included for Zipping. If **NO** is specified, they will not be included. The default value is **NO**.

<Show Progress bar> is the Logical Flag to specify if the Progress Bar needs to be shown during the Zipping Process. If 'YES' is specified, the Progress Bar will be shown, and if 'NO' is specified, it will not be shown. If no value is specified, the default value will be assumed as 'NO'.



*Wild Cards * and ? are supported only for the last information in the path. For example, C:\Wor?\Cust*.txt is invalid whereas C:\Work\Cust*.txt is valid.*

Example: 1

```
ZIP : ".\Target.zip" : "tally.ini": "Tally" : No
```

With the above Action, the following will be achieved:

1. The file **tally.ini** from the current Tally Application/Working Folder will be zipped to the File **Target.zip** in the Tally Application Folder itself.
2. The resultant Zip File will contain the password **Tally**.
3. If the file **Target.zip** exists in the current application folder, it will not be overridden.

Example: 2

To Zip all text files in work folder.

```
ZIP: "D:\Target.zip": "D:\Work\*.txt": "Tally": No: Yes: Yes
```

With the above Action, the following will be achieved:

1. All the text files from the folder **D:\Work** will be zipped to the File **Target.zip** in **D:**.
2. The resultant Zip File will contain the password **Tally**.
3. If the file **Target.zip** exists in **D:**, it will not be overridden as the 4th Parameter is set to **No**.
4. All the text files within the Sub-directories will be included under the Folder **D:\Work**, as the 5th Parameter is set to **Yes**.
5. The Progress Bar will be shown during Zipping of the Files, as the 6th Parameter is set to **Yes**.

Procedural Actions - Start Zip, Zip Add Path, Zip Exclude Path and End Zip

Procedural Actions **Start Zip**, **Zip Add Path**, **Zip Exclude Path** and **End Zip** are very useful in cases where Multiple Folders/Files need to be zipped into / excluded from a Target File.

Syntax

```
Start Zip : <Target File> [: <Overwrite>]
          Zip Add Path      : <Source Path> [: < Include sub-directory>]
          :
          :
          Zip Exclude Path : <Exclude Path>
          :
          :
          End Zip [: <Password> [: <Show Progress Bar>]]
```

Where,

<Target File> indicates the name of the resultant Zip File, and also includes the Folder Path.

<Overwrite> is the Logical Flag to specify the behaviour of the Action, if the Target File already exists. If **YES** is specified, the file will be overwritten. If 'NO' is specified, the file will not be overwritten, and will remain as it is. The default value is 'NO'.

<Source Path> is the path of the Files or Folders, which are to be zipped. It can be a folder or a file path (with or without wildcard characters).

<Include Sub-directory> is the Logical Flag to specify whether to include sub directories available in the specified source path or not. If the Source Path ends with a Folder, the entire Folder along with its Sub-Folders will be zipped, irrespective of this parameter. If the Source Path ends with a File Name Pattern, i.e., with wild cards, this parameter will be considered. If a 'YES' is specified, all the files/sub-folders matching the wild card pattern will be included for Zipping. If 'NO' is specified, they will not be included. The default value is 'NO'.

<Exclude Path> is the path of the Files/ Folders which need to be excluded from the Target Zip.

<Password> is the password assigned to the Target zipped file.

<Show Progress Bar> is the Logical Flag used to specify if the Progress Bar needs to be shown during the Zipping Process. If 'YES' is specified, then the Progress Bar will be shown, and if 'NO' is specified, it will not be shown. If no value is specified, the default value will be assumed as 'NO'.

Example: 3

```
Start ZIP : "Target.zip" : Yes
/* Overwrite the File Target.zip, if it exists */
  Zip Add Path : "tally.ini"
End Zip
```

The outcome of this example will be similar to the outcome of Example: 1. The only difference is that it is set to overwrite the target file **Target.Zip**, if it exists in the current application folder (as the **<Overwrite>** Parameter of the Action 'Start Zip' is set to **Yes**).

Example: 4

```
Start Zip : "Target.zip" : Yes
  Zip Add Path : ".\Tally.ini"
  Zip Add Path : "D:\Documents\*.doc" : Yes
/* Include Sub-Folders also */
  Zip Add Path : "C:\Work"
/* The Folder Work and the Files within this folder will be included in the Zip File */
End Zip
```

In this example, there are 3 Source Paths, which are required to be zipped:

- ❑ First Path indicates that the file **Tally.ini** from the current application folder is to be zipped.
- ❑ The Second Path indicates that the PDF files from within the **D:\Documents** Folder, including the Sub-Folders, need to be zipped.
- ❑ The Third Path indicates that the entire folder **C:\Work** needs to be zipped.

The above source files would be zipped to the target file **Target.zip**, which is specified in the Action **Start Zip**.

Example: 5

```
Start Zip : "Target.zip" : Yes
/* Overwrite the existing file in the target location */
Zip Add Path : ".\Tally.ini"
Zip Add Path : "D:\Documents\*.doc" : Yes
/* Include Sub-Folders also */
Zip Add Path : "C:\Work"
Zip Exclude Path : "*.txt"
End Zip
```

In this example, apart from using Action **Zip Add Path** to specify the first 3 source paths, the subsequent Action **Zip Exclude Path** is used to specify the exclusion of Folders or Files with the extension **.txt**. Thus, all the text files from the above specified source paths will be excluded.

3.2 UNZIP Action

The Unzip action can be used to extract the original files from the zipped files.

□ System Action - UNZIP

System Action **Unzip** is useful when all the folders/ files in the Source Zip File need to be completely unzipped, as they are. (This Action cannot be used in case of partial Unzip.)

Syntax

```
Unzip : <Target Folder>: <Source File> [: <Password> [: <Overwrite>
[:<Show Progress Bar>]]]
```

Where,

<Target Folder> is the path of the Target folder where the Unzipped Files need to reside.

<Source File> is the name of the Zip File to be unzipped.

<Password> is the Zip File Password. A Zip File bearing a Password cannot be extracted without the Password.

<Overwrite> is the Logical Flag to specify the behaviour if the Files being unzipped already exist in the Target Folder. The default value is 'NO'.

<Show Progress Bar> is the Logical Flag to specify if Progress Bar needs to be shown during the Extracting (Unzipping) Process. If YES is specified, the Progress Bar will be shown and if NO is specified, the Progress Bar will not be shown. The default value is 'NO'.

Example: 1

```
Unzip : "." : "D:\Target.zip" : "Tally"
```

In this example, the file **D:\Target.zip** will be unzipped entirely in the current Tally Application Folder. Since the Zip File bears a password **Tally**, same is being passed as the 3rd Parameter.

Example: 2

```
Unzip : "Documents" : "D:\Target.zip" : "Tally"
```

In this example, all the folders/ files within the Zip File **D:\Target.zip** will be extracted to the folder **Documents** in the current Tally Application folder.

Procedural Actions - Start Unzip, Extract Path, Unzip Exclude Path and End Unzip

The Procedural Actions **Start Unzip**, **Extract Path**, **Unzip Exclude Path** and **End Unzip** are very useful in case of partial Unzip. Using the Action 'Extract Path', one can specify the Folder/ File Path to be marked for extracting. The action 'Unzip Exclude Path' can help to exclude the specified Folders/ Files from the Zip File and extract the rest. These actions can be used in both the cases, i.e., for partial unzip as well as for total unzip.

Syntax

```
Start Unzip : <Source File> [: <Password >]
            Extract Path      : <Folder/ File Path>
                        :
                        :
            Unzip Exclude Path : <Folder/ File Path>
                        :
                        :

End Unzip : <Target folder> [:< Overwrite> [:<Show Progress Bar>]]
```

Where,

<Source File> is the path of the Zip File which needs to be zipped.

<Password> is the password to access the Target zipped file.

<Extract Path> is the File/ Folder which need to be extracted from the Zip File.

<Unzip Exclude Path> is the path of Files/ Folders, which need to be excluded from the unzipping operation.

<Target Folder> is the path of the Target folder, where the Unzipped Files need to reside.

<Overwrite> is the Logical Flag to specify the behaviour if the Files being unzipped already exist in the Target Folder. The default value is 'NO'.

<Show Progress Bar> is the Logical Flag to specify if Progress Bar needs to be shown during the Extracting (Unzipping) Process. If YES is specified, the Progress Bar will be shown and if NO is specified, the Progress Bar will not be shown. The default value is 'NO'.

Example: 3

```
Start Unzip : "D:\Target.zip"
```

```
End Unzip   : "D:\Unzipped" : Yes
```

```
/* Overwrite the existing files, if any, in the Target Folder */
```

These actions will unzip all the folders/ files within **D:\Target.zip** to the folder **D:\Unzipped** and if any file already exists, the same will be overwritten (as the second parameter of the Action **End Unzip** is set to **Yes**.)

Example: 4

To extract only .txt and .doc files from the zip file.

```
Start Unzip : "D:\Target.zip"
```

```
    Extract Path : "*.txt"
```

```
    Extract Path : "*.doc"
```

```
End Unzip   : "."
```

In this example, only the *.txt and *.doc files from **D:\Target.zip** will be unzipped to the current Tally Application Folder.

Example: 5

```
Start Unzip : "D:\Target.zip"
```

```
    Extract Path      : "Samples\Supporting Files\"
```

```
    Unzip Exclude Path : "*.xls"
```

```
End Unzip   : "."
```

In this example, from **D:\Target.zip**, all the Files and SubFolders within the folder 'Supporting Files' under 'Samples' will be unzipped to the current Tally Application Folder, as the target folder is specified as a dot (.). Also, all the files with extension .xls will not be zipped.



A file which has been zipped from Tally.ERP 9 can be extracted by using any standard third party archiving tools like Winzip, Winrar, etc., and vice-versa.

Limitations of Zip/Unzip in Tally.ERP 9:

In the following cases, Zip/Unzip action will fail:

- If the number of files being Zipped is greater than 65535
- If the Size of the Zip File is greater than or equal to 4 GB
- If the Size of any File within the Zip File is greater than or equal to 4 GB

4. Columnar Capability in Edit Mode

Multiple-Column feature in Tally.ERP 9 has till date been used by the Tally users for various reporting needs like Comparative Analysis of Data across Multiple Periods (Months, Quarters, Years, etc.), Multiple Companies, Multiple Godowns, etc. It can also be used for comparative study of various parameters like Budget vs. Actual Performance, by getting the same displayed in the form of a column-based report.

It was felt highly desirable if, apart from these reporting functionalities, various data entry operations could also be performed in a Columnar or Tabular manner. This would not only facilitate data entry in a simple, user friendly manner, but also considerably reduce the time taken to enter the data. For instance, Attendance of Employees could be accepted against the Attendance Types or days, in a Tabular format. Similarly, other examples could be Employee-wise Pay Head-wise Salary Structure, Stock Item-wise Price Level-wise Price List, etc., where this capability could be used effectively. In order to make this possible in Tally, the Columnar capability was required to be extended to support Edit mode.

From Release 4.7 onwards, support for Multi Columns has been extended to Edit Mode too, to enable the developer to design and implement such functionalities in Tally, as per requirement.

To make this possible in TDL, the Horizontal Scrolling behaviour has been enhanced to work in **Edit Mode**, which will enable the developer to create a user friendly interface, thereby allowing the user to enter data conveniently in tabular format. To achieve the columnar scrolling behaviour; in TDL, fields within the required line can be repeated over either of the following:

- Collection of Objects, OR
- Sub-Collections under a Primary Object, provided the Sub-Collections contain Object(s)

In the absence of the above '**Repeat**' specification, the only criterion to repeat the Fields/ Columns is that the number of Fields to be repeated must be known to the Line.



*With respect to Object context, Horizontal Scroll will only happen if Objects/SubObjects, on which Columns are to be repeated, exist. In other words, in **Create Mode with Object Context**, Horizontal Scroll will not work as the number of fields to be repeated is not known.*

5. New data types Introduced

Introduction

A **Data Type** in a programming language is a classification, identifying one of the various types of data supported by that language, each having certain pre-defined characteristics. It determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. They are an integral part of every programming language and hence, almost all programming languages support a set of primitive data types.

TDL, being a domain-specific language, supports various domain-specific data types like Quantity, Rate, Amount, Rate of Exchange, etc., apart from the basic data types like String, Number, Date and Logical.

In **Release 4.7**, a few Date and Time related data types have been introduced to support various business requirements. All the data types pertaining to date and time are now collectively referred to as **Calendar data types**, which are as follows:

- DATE
- TIME
- DATETIME
- DURATION

□ DUE DATE

However, two data types, viz., **Date** and **Due Date** were already supported in TDL. The data types **Time**, **DateTime**, and **Duration** are the ones which have been introduced in **Release 4.7**.

With the introduction of these data types, now various business functionalities like capturing Date and Time of entering a voucher, calculating the weekly average clock-in time of employees, etc., will be possible in Tally.ERP9.

Calendar Data Types

As already discussed, Calendar Data Types comprise of the data types **Date**, **Time**, **DateTime**, **Duration** and **Due Date**, of which, 'Time', 'DateTime' and 'Duration' have been introduced in Release 4.7.

Apart from the new data types, a few supporting Functions, Formats, Input Keywords and Qualifiers have also been introduced for each. **Formats** can be specified to indicate the format in which the value has to be displayed. The concept of **Input Keywords** and **Qualifiers** has been introduced to assist the Tally user in data entry operations, as well as to minimize the effort of the programmer in setting the values of any particular calendar data type within a Field. For example, in a Field of data type 'Date', specifying the Input keyword '**Week**' as the value, will lead to setting of the current week's beginning date as the value. Also, Qualifiers like '**This**', '**Next**', etc., if specified along with the Input Keywords, will lead to storing and displaying of values determined by the 'Input Keyword-Qualifier' combination. For example, specifying '**Next Week**' will return the beginning date of the following week as the value. All of these have been discussed in detail in the following sections.

5.1 DATE

Specifying the data type as **Date** indicates that the data container can only hold Date values. The data container can be a **UDF**, a **Variable** or a **Field**. The date values can range from January 1, 1901 to December 31, 2098. The default separator character within a Date is 'Hyphen', e.g., 22-12-2011.

Example:

```
[System : UDF]

    DateOfPurchase : Date : 1107

/* A new System UDF DateofPurchase of Type Date is declared. */
[Field : Date of Purchase]

    Type      : Date

    Storage   : DateOfPurchase

/* A Field DateofPurchase of Type Date is defined for updating values in the UDF DateOfPurchase */
```

Format Keywords

Apart from specifying the Type as **Date**, one of the various available **Formats** can also be specified, using the attribute '**Format**'.

The various Formatting Keywords for the data type 'Date' are listed in the following table. If no Format is specified, the default format 'Universal Date' is assumed.

Format	Field Value Specified	Value Returned
Short Date	22-Dec-2011	22-12-2011
Long Date	22-Dec-2011	Thursday, 22 Dec, 2011
Universal Date (default format)	22-Dec-2011	22-Dec-2011
Month Beginning	1-Dec-2011	Dec-2011
	22-Dec-2011	22-Dec-2011
Month Ending	31-Dec-2011	Dec-2011
	22-Dec-2011	22-Dec-2011

Table 1. Date Formats for Data Type **Date**

Example:

```
[Field : Date of Purchase]
```

```
Type      : Date
```

```
Storage   : DateOfPurchase
```

```
Format    : "Short Date"
```

/ The Format specified within the Field **DateofPurchase** is **Short Date**, and hence, the date is returned in **dd-mm-yyyy** format. */*

Input Keywords

There are certain Input keywords, which can be specified instead of the date itself. This increases the ease of data entry. The Input Keywords available for the data type **Date** are as follows:

Input Keyword	Alias	Value Returned
Today	Today, Day, Now	Current system date
Tomorrow	Tomorrow, Tommorrow	Next day's date
Yesterday	YDay	Previous day's date
Week		Current week's beginning date
Month	Mth	Current month's beginning date
Year	Yr	First date, i.e., 1-Jan, of the current year

Table 2. Input Keywords for Data Type **Date**

Example:

[Field : Date of Purchase]

Type : Date

Storage : DateOfPurchase

Format : "Short Date"

Set As : Today

/ The input keyword **Today** sets today's date as the value within the Field **Date of Purchase**. */*

Date Qualifiers

There are also some date qualifiers, which can be used in combination with input keywords, to return the required date value. Qualifiers cannot be used independently, but they can be provided as additional specification for the input keyword. The various date qualifiers are as follows:

Date Qualifier	Usage (along with Input Keywords)
Financial	Financial Year, This Financial Year, Prev Financial Year, Next Financial Year, Last Financial Year
This	This Week, This Month, This Year, This Financial Year
Prev/Last	Prev Week, Prev Month, Prev Year, Prev Financial Year, Last Week, Last Month, Last Year, Last Financial Year
Next	Next Week, Next Month, Next Year, Next Financial Year

Figure 3. Date Qualifiers

[Field : Date of Purchase]

Type : Date

Storage : DateOfPurchase

Format : "Short Date"

Set As : Last Financial Year

/ In **Last Financial Year**, '**Last**' and '**Financial**' are qualifiers, while '**Year**' is an Input Keyword. It sets the field value as the 1st day of the last/previous financial year. */*



*Week is assumed to begin on **Sunday**. For example, on 20th August, 2013, the value of the Field set as 'Prev Week' is returned as 11th August, 2013, which is the Sunday of the week prior to the current week.*

Functions

Various Type casting and Manipulation Functions have also been provided, corresponding to the Data Type **Date**. They are as follows:

□ Function - \$\$Date

The function \$\$Date is used to convert a valid date value from any other data type to the data type Date. This is mainly required when manipulations on the date need to be performed.

Syntax

```
$$Date : <Expression>
```

Where,

<Expression> can be any expression, which evaluates to a valid date value.

□ Function - \$\$MonthOfDate

The function \$\$MonthOfDate returns the month corresponding to the date specified as the parameter.

Syntax

```
$$MonthOfDate : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

The result returned will be of Type 'Number'. For example, if the specified date is **05-04-2013**, then the function returns the value **4**, as the date belongs to the 4th month of the year.

□ Function - \$\$DayOfWeek

The function \$\$DayOfWeek returns the weekday corresponding to the date specified as the parameter.

Syntax

```
$$DayOfWeek : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value. For example, if the specified date is **05-04-2013**, then the result would be **Friday**.

□ Function - \$\$DayOfDate

This function returns a number to represent the day, corresponding to the date specified as the parameter. For example, Sunday will be represented by the number 1, Monday by 2, and so on.

Syntax

```
$$DayOfDate : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **5**, as the corresponding day is **Friday**, which is the 5th day of the week.

□ Function - \$\$YearOfDate

The function \$\$YearOfDate returns a number representing the year of the date specified as the parameter.

Syntax

```
$$YearOfDate : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

The result returned by the function will be of Type 'Number'. For example, if the specified date is **05-04-2013**, then the result would be **2013**.

□ Function - \$\$ShortMonthName

The function \$\$ShortMonthName returns the short form of the name of the month, corresponding to the date specified as the parameter, e.g., Jan, Feb, etc.

Syntax

```
$$ShortMonthName : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the date value is **05-04-2013**, then the value returned by the function will be **Apr**.

□ Function - \$\$FullMonthName

The function \$\$FullMonthName returns the full name of the month, corresponding to the date specified as the parameter, e.g., January, February, etc.

Syntax

```
$$FullMonthName : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value. For example, if the specified date is **05-04-2013**, then the result would be **April**.

□ Function - \$\$WeekEnd

This function returns the date of the first Sunday following the date specified as the parameter. However, if the argument date, i.e., the date specified, itself corresponds to the day Sunday, then the date corresponding to the following Sunday will be returned.

Syntax

```
$$WeekEnd : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **07-Apr-2013**, being the following Sunday. If the date specified is itself **07-04-2013**, then the date returned will be **14-Apr-2013**, i.e., the date of the following Sunday.

□ Function - \$\$MonthEnd

It returns the date of the last day of the month, corresponding to the date specified as parameter.

Syntax

`$$MonthEnd : <Date Expression>`

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **30-Apr-2013**.

□ Function - \$\$YearEnd

This function returns the last date of the year, considered to be starting from the date specified as the parameter.

Syntax

`$$YearEnd : <Date Expression>`

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **4-Apr-2014**, which is one year from the date specified, i.e., the last day of the year, if the specified date is considered as the current date.

□ Function - \$\$MonthStart

The function \$\$MonthStart returns the date of the first day of the month, corresponding to the date specified as the parameter.

Syntax

`$$MonthStart : <Date Expression>`

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **1-Apr-2013**.

□ Function - \$\$FinYearBeg

This function is used to fetch the starting date of the company's financial year, corresponding to a particular specified date. Two parameters are required for this function. The 1st parameter is the Date value for which the corresponding Financial Year's Beginning date is to be identified. The 2nd parameter is the "Financial Year From" date of the company, i.e., the time from which the financial year of the company normally starts. Specification of this parameter is essential as the Financial year is determined by the law of the country, and can be Apr-Mar, Jan-Dec, etc.

Syntax

`$$FinYearBeg : <First Date Expression> : <Second Date Expression>`

Where,

<First Date Expression> evaluates to the Date, for which the corresponding Financial Year's Beginning date is to be identified.

<Second Date Expression> is used to specify the normal financial year beginning date (irrespective of the year).

For example, if the first parameter is **05-04-2013** and the second parameter date is **1-4-2010**, then the result would be **1-Apr-2013**.

□ **Function - \$\$FinYearEnd**

This function is used to fetch the end/last date of the company's financial year, corresponding to a particular specified date. Two parameters are required for this function. The 1st parameter is the Date value for which the corresponding Financial Year's Ending date is to be identified. The 2nd parameter is the "Financial Year From" date of the company, i.e., the time from which the financial year of the company normally starts. Specification of this parameter is essential as the Financial year is determined by the law of the country, and can be Apr-Mar, Jan-Dec, etc.

Syntax

\$\$FinYearEnd : <First Date Expression> : <Second Date Expression>

Where,

<First Date Expression> evaluates to the Date, for which the corresponding Financial Year's Ending date is to be identified.

<Second Date Expression> is used to specify the normal financial year beginning date (irrespective of the year).

For example, if the first parameter is **05-04-2013** and the second parameter date is **1-4-2010**, then the result would be **31-Mar-2014**.

□ **Function - \$\$PrevYear**

The function \$\$PrevYear returns the previous year's date, respective to the date specified as the parameter, i.e., the date exactly a year ago.

Syntax

\$\$PrevYear : <Date Expression>

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **5-Apr-2012**.

□ **Function - \$\$NextYear**

The function \$\$NextYear returns the next year's date, corresponding to the date specified as the parameter, i.e., the date exactly a year from now.

Syntax

\$\$NextYear : <Date Expression>

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **5-Apr-2014**.

□ Function - \$\$PrevMonth

The function \$\$PrevMonth returns the previous month's date, corresponding to the date specified as the parameter, i.e., the date exactly a month ago.

Syntax

```
$$PrevMonth : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **5-Mar-2013**.

□ Function - \$\$NextMonth

The function \$\$NextMonth returns the next month's date, corresponding to the date specified as the parameter, i.e., the date exactly a month from now.

Syntax

```
$$NextMonth : <Date Expression>
```

Where,

<Date Expression> can be any expression, which evaluates to a valid date value.

For example, if the specified date is **05-04-2013**, then the result would be **5-May-2013**.

5.2 TIME

A new Data Type **TIME** has been introduced, which represents an absolute time of the day. Specifying the Data Type as 'Time' indicates that the data container can hold only the Time values. The data container can be a UDF, a Variable or a Field. A value of this data type describes the time, with milliseconds precision, i.e., using the sub-parts HOURS, MINUTES, SECONDS, and MILLI- SECONDS. Thus, the format is **hh:mm:ss:MMM**

By default, Colon is the separator between the sub-parts of the time value, which can also be altered by the user. For example, **16:35:12:348**

Example:

```
[System : UDF]
```

```
TimeOfPurchase : Time : 1108
```

```
/* A new System UDF TimeofPurchase of Type Time is declared. */
```

```
[Field : Time of Purchase]
```

```
Type      : Time
```

```
Storage   : TimeOfPurchase
```

```
/* A Field Time of Purchase of Type Time is defined for updating values in the UDF TimeOfPurchase */
```

Format Keywords

Just as in the case of 'Date' data type, various Formatting Keywords have also been introduced for the 'Time' data type, to render the **Time** in various formats. The formats are listed in the following table.

Format	Alias	Value of Field	Value Returned
12 hour (Default Format)	12 hr	16:35:12:348	4:35 PM
24 hour	24 hr	16:35:12:348	16:35
WithSeconds	With Sec, With Secs	16:35:12:348	4:35:12 PM (12 hr format) 16:35:12 (24 hr format)
With MilliSeconds	With MilliSecs, With MilliSec	16:35:12:348	4:35:12:348 PM (12 hr format) 16:35:12:348 (24 hr format)
Prefix AMPM	AMPM Prefix	16:35:12:348	PM 4:35:12:348 (12 hr format, with millisec) By default, the AM/PM designator is suffixed to the time value. Specifying the format 'Prefix AMPM' makes the AM/PM designator get appended as prefix, instead of suffix.
No Zero		0:00	If this format is specified, the time won't be displayed, if the time value in the field is 0:00
Separator: '<Symbol>'		16:35:12:348	This format specifier is used to change the time separator. For example, Format : "Separator: '/'" will return the value as 4/35 PM, instead of 4:35 PM

Figure 4. Format Keywords for Data Type **Time**

Input Keywords

The input Keywords available for the data type **Time** are as shown in the following table:

Input Keyword	Value Returned
Now	Current system time value
Midnight	Midnight time value, i.e., 0:00
Noon	Noon time value, i.e., 12:00

Figure 5. Input Keywords for Data Type **Time**

Functions

□ Function - \$\$Time

The Function \$\$Time is used to convert a valid time value from any other data type to the Data Type Time. It is mainly required when manipulations on the time need to be performed.

Syntax

```
$$Time:<Expression>
```

Where,

<Expression> can be any expression, which evaluates to a valid Time value.

5.3 DATETIME

A new Data Type **DateTime** has also been introduced. Specifying the Data Type as **DateTime** indicates that the data container can hold only the values of type 'DateTime'. The data container can be a UDF, a Variable or a Field. A value of data type 'DateTime' represents a date, along with the absolute time of the day. The date and the time are described using the sub-parts DAY, MONTH, YEAR, HOUR, MINUTE, SECOND, and MILLISECONDS. Thus, the format is **dd-mm-yy hh:mm:ss:MMM**

By default, the separator for the 'Date' part is Hyphen (-) and for the 'Time' part is Colon (:). However, the same can be altered by the user. The date-time combination values can range from January 1, 1901 00:00:00:000 to December 31, 2098 23:59:59:999

Example:

```
[System : UDF]
```

```
DateTimeOfPurchase : DateTime : 1109
```

/ A new System UDF DateTimeOfPurchase of Type DateTime is declared. */*

```
[Field : Date and Time of Purchase]
```

```
Type      : DateTime
```

```
Storage : DateTimeOfPurchase
```

/ A Field DateandTimeofPurchase of Type DateTime is defined for updating values in the UDF DateTimeOfPurchase */*

Input Formats

Date Only

This Format is specified to accept both Date and Time values in the Field, but display only Date. In other words, if this Format is specified within a field of type DateTime, then the field can accept both Date and Time values, but will display only Date.

For example, consider a Field of Type DateTime, where the Format is specified as 'Date Only'. If the Input keyword 'Today' is entered in this Field, then instead of displaying the current date and time, the field will display only the current date.

Example:

```
[Field : Date of Purchase]
```

```
Type      : DateTime
Storage   : DateTimeOfPurchase
Format    : "Long Date, Date Only"
```

/ Both Date and Time values will be accepted as input in the Field **DateofPurchase**, but only the **Date** value will be displayed. */*

Time Only

This Format is specified to accept both Date and Time values in the Field, but display only Time. In other words, if this Format is specified within a field of type DateTime, then the field can accept both Date and Time values, but will display only Time.

For example, consider a Field of type DateTime, where the Format is specified as 'Time Only'. If the Input keyword 'Now' is entered in this Field, then instead of displaying the current date and time, the field will display only the current time.

Example:

```
[Field : Time of Purchase]
Type      : DateTime
Storage   : DateTimeOfPurchase
Format    : "12 hour, Time Only"
```

/ Both Date and Time values will be accepted as input in the Field **TimeofPurchase**, but only the **Time** value will be displayed. */*

Input Keywords

Input Keywords available for the date type **DateTime** are as shown in the following table:

Input Keyword	Alias	Value Returned
Today	Day, Now	Current system date & time.
Tomorrow	Tomorrow, Tomorrow	Next day's date & time, i.e., the date and time exactly a day later.
Yesterday	YDay	Previous day's date & time, i.e., the date & time exactly a day before.
Week		The 'Date' part is set as the Current week's beginning date.

Month	Mth	The 'Date' part is set as the Current month's beginning date.
Year	Yr	The 'Date' part is set as the Current year's beginning date.

Figure 6. Input Keywords for Data Type **DateTime**



For the keywords 'Tomorrow', 'Yesterday', 'Month', 'Week' and 'Year', the time already available in the Field, i.e., the 'Time' Part of the DateTime value, is set as the time. If the 'Time' value is not available, then it will be set to 0:00.

Date Qualifiers

All the qualifiers for the data type **Date** are applicable for the data type **DateTime** as well.

Functions

□ Function - \$\$DateTime

The Function \$\$DateTime is used to convert a valid Date and/or Time (DateTime) value from any other data type to the data type 'DateTime'. This is mainly required when manipulations on DateTime values need to be performed.

Syntax

`$$DateTime : <DateTime Expression>`

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

□ Function - \$\$AddHours

The function \$\$AddHours is used to add a certain specified number of hours to a value of data type DateTime.

Syntax

`$$AddHours : <DateTime Expression> : <Value>`

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of hours to be added to the DateTime value.

For example, if the specified DateTime is **05-04-2013 5:20 PM** and the <Value> to be added is **4**, then the value returned by the function would be **5-Apr-2013 9:20 PM**.

□ Function - \$\$SubHours

It is used to subtract a certain specified number of hours from a value of data type DateTime.

Syntax

`$$SubHours : <DateTime Expression> : <Value>`

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of hours to be subtracted from the DateTime value.

For example, if the specified DateTime is **05-04-2013 5:20 PM** and the **<Value>** to be subtracted is **4**, then the result would be **5-Apr-2013 1:20 PM**.

□ **Function - \$\$AddMinutes**

The function **\$\$AddMinutes** is used to add a certain specified number of minutes to a value of data type DateTime.

Syntax

\$\$AddMinutes : **<DateTime Expression>** : **<Value>**

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of minutes to be added to the DateTime value.

For example, if the specified DateTime is **05-04-2013 5:20 PM** and the value to be added is **20**, then the result would be **5-Apr-2013 5:40 PM**.

□ **Function - \$\$SubMinutes**

The function **\$\$SubMinutes** is used to subtract a certain specified number of minutes from a value of data type DateTime.

Syntax

\$\$SubMinutes : **<DateTime Expression>** : **<Value>**

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of minutes to be subtracted from the DateTime value.

For example, if the specified DateTime is **05-04-2013 5:20 PM** and the **<Value>** to be subtracted is **20**, then the result would be **5-Apr-2013 5:00 PM**.

□ **Function - \$\$AddSeconds**

The function **\$\$AddSeconds** is used to add a certain specified number of seconds to a value of data type DateTime.

Syntax

\$\$AddSeconds : **<DateTime Expression>** : **<Value>**

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of seconds to be added.

For example, if the specified DateTime value is **05-04-2013 5:20:15 PM** and the <Value> to be added is **15**, then the result would be **5-Apr-2013 5:20:30 PM**.

□ Function - \$\$SubSeconds

The function \$\$SubSeconds is used to subtract a certain specified number of seconds from a value of data type DateTime.

Syntax

```
$$SubSeconds : <DateTime Expression> : <Value>
```

Where,

<DateTime Expression> can be any expression, which evaluates to a valid Date and/or Time (DateTime) value.

<Value> is the number of seconds to be subtracted from the DateTime value.

For example, if the specified DateTime value is **05-04-2013 5:20:15 PM** and the <Value> to be subtracted is **15**, then the result would be **5-Apr-2013 5:20:00 PM**.

Sub Types introduced for the Data Type 'DateTime'

Two subtypes have also been introduced for the data type 'DateTime':

Date

This SubType is specified to accept both Date and Time values in the Field, but display only Date. In other words, if this SubType is specified within a field of type DateTime, then the field can accept both Date and Time values, but will display only Date.

For example, consider a Field of type DateTime, where the sub-type is specified as 'Date'. If the Input keyword 'Today' is entered in this Field, then instead of displaying the current date and time, the field will display only the current date.

Example:

```
[Field : Date and Time of Purchase Date]
```

```
Type      : DateTime : Date
```

```
Storage : DateTimeOfPurchase
```

Time

This SubType is specified to accept both Date and Time values in the Field, but display only Time. In other words, if this SubType is specified within a field of type DateTime, then the field can accept both Date and Time values, but will display only Time.

For example, consider a Field of type DateTime, where the sub-type is specified as 'Time'. If the Input keyword 'Now' is entered in this Field, then instead of displaying the current date and time, the field will display only the current time.

Example:

```
[Field : Date and Time of Purchase Time]
```

```
Type      : DateTime : Time
```

Storage : DateTimeOfPurchase

Here, both Date and Time will be accepted but only the 'Time' part of the DateTime value is displayed in the field. Manipulation can be done on date as well as time.



Two fields with subtypes as Date and Time can have the same storage and the changes from one field will be reflected in the other field.

5.4 DURATION

Values of this data type will represent the interval between two Date and/or Time values (two DateTime values), measured in years, months, weeks, days, hours, minutes, and seconds.

Example:

[System : UDF]

TenureOfService : Duration : 1110

[Field : Tenure Of Service]

Type : Duration

Storage : TenureOfService

/ A Field **TenureOfService** of Type **Duration** is defined for updating values in the UDF **TenureOfService** */*

Formats

The format options available for the **Duration** data type are:

Format	Alias	Value of Field	Value Returned
Days (Default)	Day, Dys	25	25 Days
Years	Year, Yr, Yrs	25	25 Years
Months	Month	25	25 Months
Weeks	Week, Weak, Wks	25	25 Weeks
Hours	Hour, Hr, Hrs	25	25 Hours
Minutes	Mins, Minute, Min	25	25 Minutes
Seconds	Secs, Second, Sec	25	25 Seconds
YMD (Years, months and days)		416	1 Year, 1 Month and 20 Days
HMS (Hours, minutes and seconds)		5000	1 Hour, 23 Minutes and 20 Seconds

Figure 7. Format Options for Data Type **Duration**

Example:

```
[Field : Tenure Of Service]
```

```
Type      : Duration
```

```
Storage   : TenureOfService
```

```
Format    : "Months, Days"
```

Points to Remember:

- Any combination of the formats can be provided. However, **Week** is an independent format and cannot be clubbed with any other format. Let's see the following illustrations:
 - In the above example, if the value entered by the user in the Field is **100**, then the lowest (in terms of duration) of the formats specified will be considered as the input format, i.e., **100** will be considered as **100 days** and then, the same will be displayed as output in terms of months and days, i.e., **3 months and 10 days**.
 - If the Format is specified as '**Weeks, Days**', then the value returned will be in **Days**, as 'Weeks' is an independent entity and cannot be clubbed with any other format.
- If the value entered is less than the lowest value (in terms of duration) among the formats specified, then the value returned will be zero. For example,
 - If **6 days** is entered and the format is weeks, then the value returned will be **0 weeks**.
 - If **8 days** is entered and the format is weeks, then **1 week** will be displayed.

Functions

- Function - **\$\$Duration**

The Function **\$\$Duration** is used to convert a valid duration value from any other data type to the data type **Duration**. This is mainly required when manipulations on values of type **Date**, **Time** or **DateTime** need to be performed.

Syntax

```
$$Duration : <Expression>
```

Where,

<Expression> can be any expression evaluating to a valid duration value like 10 days, 5 hrs, etc.



*A value of Type 'Date', when subtracted from another value of Type 'Date', will result in a value of Type 'Number', and **not** 'Duration'. If the resultant value is needed to be of Type 'Duration', the function **\$\$Duration** has to be used.*

- Function - **\$\$GetEndDateTime**

If the initial/starting **DateTime** value and the **Duration** value are provided, then the Function **\$\$GetEndDateTime** will return the final/ending **DateTime** value.

Syntax

```
$$GetEndDateTime : <DateTime Expression> : <Value>
```

Where,

<DateTime Expression> can be any expression, evaluating to a valid Date/DateTime value.

<Value> is used to specify the Duration value.

For example, if the specified Datetime value is **05-04-2013 12:12 AM** and the Duration value is set as **10 days**, then the result would be **15-Apr-2013 0:12 AM** (12:12 AM same as 0:12 AM)

□ **Function - \$\$GetStartDateTime**

If the final/ending DateTime value and the Duration value are provided, the Function \$\$GetStartDateTime will return the initial/starting DateTime value.

Syntax

```
$$GetStartDateTime : <DateTime Expression> : <Value>
```

Where,

<DateTime Expression> can be any expression, evaluating to a valid Date/DateTime value.

<Value> is used to specify the Duration value.

For example, if the specified DateTime value is **05-04-2013 12:12 AM** and the Duration value is **11 hours 30 minutes**, then the result would be **4-Apr-2013 12:42 PM**.

□ **Function - \$\$GetDuration**

If two DateTime values, representing the Initial/Starting and Final/Ending DateTime values, are provided, then the function \$\$GetDuration will return the duration between them.

Syntax

```
$$GetDuration : <First DateTime Expression> : <Second DateTime Expression>
```

Where,

<First DateTime Expression> can be any expression, which evaluates to a valid Date/DateTime value.

<Second DateTime Expression> can be any expression, which evaluates to a valid Date/DateTime value.

For example, if the first parameter is **05-04-2013 12:12 AM** and the second parameter is **15-Apr-2013 0:12 AM**, then the value returned would be **10 Days**. (12:12 AM is the same as 0:12 AM).

□ **Function - \$\$AddDuration**

The function \$\$AddDuration adds a duration value to another duration value. Two duration values are provided as parameters. If a particular parameter contains only a number and no unit is specified, it is assumed to be that many number of days.

Syntax

```
$$AddDuration : <First Duration Expression> : <Second Duration Expression>
```

Where,

<First Duration Expression> can be any expression, which evaluates to a valid Duration value or a Number.

<**Second Duration Expression**> can be any expression, which evaluates to a valid Duration value or a Number.

If the first Duration parameter is set to **10 Days** and the second Duration parameter is set to **4 Months**, then the result would be **130 Days**.

□ **Function - \$\$SubDuration**

This function subtracts the specified duration or from another duration value. Two duration values are provided as parameters. If a particular parameter contains only a number and no unit is specified, it is assumed to be that many number of days.

Syntax

\$\$SubDuration : <First Duration Expression> : <Second Duration Expression>

Where,

<**First Duration Expression**> can be any expression, which evaluates to a valid Duration value or a Number. It is the value to be subtracted, and should be less than or equal to the other value.

<**Second Duration Expression**> can be any expression, which evaluates to a valid Duration value or a Number.

If the first parameter Duration is set to **10 Days** and the second Parameter Duration is set to **4 Months**, then the result would be **110 Days**.

5.5 DUE DATE

The values of Data Type **Due Date** are used to represent the Due Date in cases like Purchase Order, Bill Credit Period, etc., in Tally.ERP 9. These values actually comprise of two date values, i.e., the 'From' Date and the last date by which honouring of the commitment is due (i.e., the 'due date'). A flexible range of values can be specified. The value can be a Date (the 'due date') or the Duration, in terms of Days, Weeks, Months and Years, from the starting date ('From' date). The acceptable duration values used to specify the due date range from 0 to 89 years.

Input Formats

Four input formats can be specified for values of data type **Due Date**. If no input format is specified, then the default format **Days** is considered.

Format	Value Returned
Days (Default)	Due Date in Days
Weeks	Due Date in Weeks
Months	Due Date in Months
Years	Due Date in Years

Figure 8. Input Formats for data type Due Date



- *As the default format is 'Days', on entering only the number 30, 30 days will be considered.*
- *The value is displayed in the same format in which it has been entered.*

Functions

□ Function - \$\$DateRange

This function is used to convert a valid Due Date value from any other data type to the data type **Due Date**. It is used to set the values to a method of Type 'Due Date' programmatically. For example, while creating a Purchase Order Voucher programmatically using User Defined Functions, to set the value to method OrderDueDate (of type Due Date), we need to use this Function.

Syntax

```
$$DateRange : <Value> : <Date Expression> : <Logical Expression>
```

Where,

<Value> must evaluate to a Date OR the duration in terms of Days, Weeks, Months and Years.

<DateExpression> must evaluate to a Date, which is considered as the 'From' Date.

<Logical Expression> must evaluate to a logical value. This logical value denotes whether the specified 'From' date is also to be included, while calculating the due date.

For example, if the first parameter is set to **2 M**, the second Parameter to **05-04-2013**, and the third Parameter to **YES**, then the method, say 'OrderDueDate', will be set as **04-06-2013**, i.e., 2 Months from the specified 'From' date, including the 'From' date. However, if the logical expression is **NO**, then the value stored in the method would be **05-06-2013**.

□ Function - \$\$DueDateByDate

This function returns the due date value in 'Date' format. The parameter passed to this function is a value of type Due Date.

Syntax

```
$$DueDateByDate : <Due Date Expression>
```

Where,

<Due Date Method> can be any expression, which evaluates to a value of Type Due Date.

In continuation to the previous example, if the method 'OrderDuedate', evaluated with the Logical value set as **YES**, is provided as the parameter, the result would be **04-06-2013**, i.e., the Due Date in 'Date' format.

□ Function - \$\$DueDateInDays

This function provides the due date value in 'number of days' format. The parameter passed to this function is a value of type Due Date. The function returns the due date in terms of the total number of days, considered from the 'From' date.

Syntax

```
$$DueDateInDays : <Due Date Expression>
```

Where,

<Due Date Method> can be any expression, which evaluates to a value of Type Due Date.

□ Function - `$$IsSetByDate`

This function checks and indicates if the due date is mentioned in terms of 'Date', instead of 'Number of Days'. It returns a logical value, i.e., 'Yes' OR 'No'. If the due date's format is 'Date', then the function returns 'Yes', else it returns 'No'.

Syntax

```
$$IsSetByDate : <Date Expression>
```

Where,

<Date Expression> can be any value of Type Due Date.

5.6 Compatibility of Data Type 'DateTime' with Data Types 'Date' and 'Time'

1. Setting a data container (Field, UDF or Method) of type **Date** OR **Time** with a value of data type **DateTime** is possible. For example, if the value of a field of type Date or Time is set by referring to another field of type DateTime, the particular date value OR time value is taken, respectively, and the same is displayed in the Field.

Example:

```
[Field : Date of Purchase]

Type      : Date

Storage   : DateOfPurchase

Format    : "Long Date"

Set as    : #DateandTimeOfPurchase
```

Date and TimeOfPurchase is a Field containing a value of type **DateTime**. When this field is being referred to in the Field **Date of Purchase** of type **Date**, the 'Date' part of the value is extracted and set as the value of the field.

2. **Date** and **Time** functions can be used on a value of type **DateTime**. The vice versa is also true, i.e., **DateTime** functions can also be used on a value of type **Date** or **Time**.

Example:

```
$$DayOfWeek : $DateTimeOfPurchase

/* Date function is used on method value of type DateTime */
$$DayOfDate : #DateandTimeOfPurchase

/* Date function is used on field value of type DateTime */
$$Time      : #DateandTimeOfPurchase

/* Time function is used on field value of type DateTime */
$$DateTime  : #DateOfPurchase

/* DateTime function is used on field value of type Date */
```

5.7 Constraints and Assumptions for Calendar Data Types

TIME

1. Data Type **Time** currently does not support time zones.
2. Data Type **Date** is independent of the Data Type **Time**. This means that if the time is changed from PM to AM, or vice versa, the Date does not change.
3. Time is **cyclic** in nature, i.e., if 12 hours are subtracted from 1 am, it results in 1 pm.

DATETIME

In a field of type **DateTime**, for entering the time value, date must be entered first.

When a value of type **Date** is converted to type **DateTime**, then default value of time is taken as 0:00.

When a value of type **Time** is converted to type **DateTime**, then the default value of Date is assumed as the current date.

DURATION

In **Duration** data type, if no unit is specified, then it is by default taken as days. For example, in a Field of type 'Duration', if the value entered is **10**, then it is considered as **10 days**.

Negative Duration is not supported.

5.8 COM Support to Calendar Data Types

We are already aware that the COM DLL Support feature had been introduced for TDL in Release 4.6, to support COM DLL Processing. But, the support was restricted to only a few data types. In the Release 4.61, the support had been extended to many more data types. (For Further details on COM Capability, refer to 'What's New in Release 4.6' and 'What's New in Release 4.61' in this document.)

From Release 4.7 onwards, the COM support has been extended for two other data types, viz., Time and DateTime.

What's New in Release 4.61

1. COM Data Types Support

In Release 4.6, a new Definition Type **COM Interface**, a new Action **Exec COM Interface**, and new Functions **\$\$COMExecute** and **\$\$IsCOMInterfaceInvokable** had been introduced to extend Support for COM Servers. However, this support was limited to only a few COM Data Types.

From Release 4.61 onwards, the following COM Data Types will also be supported:

COM Data Type	TDL Data Type	Other Permissible TDL Data Types	Range of Values/Other Details
String	String	Number, Date, Logical, Amount	
Float	Number	String, Amount	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$
Double/Number	Number	String, Amount	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$
Currency/Amount	Amount	String, Number	This would have a precision of 4 decimal places, rather than 5, as in Tally.ERP 9. If Tally sends the number in 5 decimal places, DLL will round it off to 4 decimal places.
Char	String(1st character of String is used)	Number, Date, Logical, Amount (Only the First letter) For e.g., if the number is 987, then the result char would be 9 and similarly, for the other data types. For Date, it depends on the Date Format.	Single Character
Byte/Unsigned char	Number	String, Amount	0 to 255
Short/Wchar	Number	String, Amount	-32,768 to 32,767
Unsigned short	Number	String, Amount	0 to 65,535

Long	Number	String, Amount	-2,147,483,648 to 2,147,483,647
Long Long	Number	String, Amount	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Unsigned Long	Number	String, Amount	0 to 4,294,967,295
Unsigned Long Long	Number	String, Amount	0 to 18,446,744,073,709,551,615
Integer	Number	String, Amount	-2,147,483,648 to 2,147,483,647
Unsigned Integer	Number	String, Amount	0 to 4,294,967,295
Bool/Boolean/Logical	Logical	String (Yes/No, 0/1, True/False)	True/False
Date	Date	String	
Variant	String, Date, Amount Number, Logical,		This can be an Out or InOut parameter. The value for the data type can be any one of the following data types, viz String, Amount, Number, Date OR Logical.
Scode	Number	String	0 to 4,294,967,295 This is a kind of an error code data type, used by Windows API.

Figure 1. COM Data Types

The parameters can also be of the Other Permissible TDL data types, as mentioned in the table, in place of the data types mentioned in the column titled 'TDL Data Type'. Irrespective of whether the parameter is an In, Out OR InOut parameter, Tally implicitly converts these data types to the respective COM Data Types.

In **Release 4.6**, while declaring the Parameters for COM Interface, only a limited number of data types could be accepted as data type for the parameter.

Let's understand this with the help of the following example:

DLL Code	TDL Code - COM Interface Definition
<pre>using System; using System.Collections.Generic; using System.Linq; using System.Text; namespace MathLib { public class MathClass { public double Add(double pDouble) { return pDouble + 9; } } }</pre>	<pre>[COM Interface : TSPL Smp Add] Project : MathLib Class : MathClass Interface : Add Parameters : p1 : Number : In Returns : Number</pre>

In the codes shown in the table, the Interface **Add** of Class **MathClass** had the Parameter Data type as **Double** in DLL, while the same had to be mapped to **Number** as the Parameter Data type in TDL.

From **Release 4.61** onwards, the COM Data Types listed in 'Other Permissible TDL Data Types' column in the table are also supported. Thus, in this particular example, the data type of the Parameter can also be specified as **Double** in place of **Number**, and hence, the same can be rewritten as:

```
Parameters : p1 : Double : In
```

However, while invoking the COM Server, the data type must be a TDL Data Type, viz. Number, String, Amount, Date OR Logical.

Example:

```
[Function : TSPL Smp Addition]
```

```
Parameter : InputNo : Number
```

```
00          : Exec COM Interface : TSPL Smp Add : ##InputNo
```

```
10          : Log: $$LastResult
```



*It is not necessary to have the above TDL Data Type as **Number**. It could also be a **String** or an **Amount**. However, the value within the String should be a **Number**.*

What's New in Release 4.6

1. COM DLL Support in TDL

A **dynamic link library (DLL)** is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.

DLL support has been provided in TDL for quite a while. The focus was primarily on extending Tally for bringing in capabilities which could not be achieved within Tally. '**CallDLLFunction**' allowed calling native/unmanaged DLLs which were written and compiled in C++.

In further releases, this moved a step further where the support was extended to use **Plug-In** and **Activex Plug-In**, where the XML output from the DLL could be used as a data source in the collection artefact, and thereby, consumed in TDL. This paved the way for new possibilities on extending Tally as per customer needs, which required interactions with external hardware, etc. However, this required changes in the DLL as per the processing capabilities of the collection. The XML output from the collection had to be necessarily from a function within DLL which had to be named mandatorily as **TDLCollection** within the DLLClass.

The **Component Object Model (COM)** is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher level software services. These DLLs provide the standard benefits of shared libraries. Component Object Model (COM) was introduced by Microsoft to enable inter process communication and dynamic object creation across a varied range of programming languages. In other words, objects can be implemented in environments seamlessly different from the one in which they are created. This technology specifies manipulation of data associated with objects through an Interface. A **COM interface** refers to a predefined group of related functions that a COM class implements. The object implements the interface by using the code that implements each method of the interface, and provides COM binary-compliant pointers to those functions, to the COM library. COM then makes those functions available to the requesting clients.

COM DLL Support will pave the way for providing features like 'Tally For Blind' using the generic speech API provided by Microsoft. This Release comes with enhanced capabilities in the language to support COM DLL processing. A new definition type called **COM Interface** has been introduced for the same.

1.1 COM Servers and COM Clients

A **COM Server** is any class/method that provides services to clients. These services are in the form of interface implementations which can be called by any client that is able to get a pointer to one of the interfaces on the server object. A **COM Client** makes use of a COM Server, and uses its services by calling the methods of its interfaces.

COM Server DLLs are DLLs which have objects exposed to COM and can be used by COM Clients. Let's take the following DLL Code as an example:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace MathLib
{
    public class MathClass
    {
        public double Add(double pDouble)
        {
            return pDouble + 9;
        }

        public int Divide(int Dividend, int Divisor, out int Remainder)
        {
            int Quotient;

            Quotient = Dividend/Divisor;

            Remainder = Dividend - (Divisor * Quotient);

            return Quotient;
        }

        public void datefunc(ref DateTime date)
        {
            date = date.AddDays(40);
        }
    }
}
```

In C# DLL, three functions are called. They are:

- **Add** - it takes an input parameter, and returns it, after adding 9 to it.

- **Divide** - it accepts a Dividend and a Divisor as inputs, and returns the Quotient, along with the Remainder.
- **datefunc** - it accepts a parameter 'date' as input, and updates it by adding 40 days to it.

1.2 Registering the DLL

For .NET COM Servers, we need to register with a parameter/**CodeBase**.

Example:

```
regasm/CodeBase <DLL Name with absolute path>
```

For further details on steps to register DLLs, please refer to the section **How to Register DLLs**.



If the DLL is written using VB.NET or C#.NET; before building the DLL, please ensure that the COM Visible Property is set.

After registering the DLL, ensure that the registry entries are available as under:

- **ProgID**: The value for which should be **<ProjectName>.<ClassName>**
- **Inprocserver32 OR InprocHandler32 OR LocalService** – The default value for these keys in registry should be the path to the DLL/EXE which we have registered.

*To find the Registry Entry, one can open **regedit**, and locate the project name of the DLL that has been registered and ensure the above.*

1.3 Implementation in Tally.ERP 9 using TDL

A new definition 'COM Interface' has been introduced to call a function available in external DLL/ EXE (COM Server). This will help the TDL Developer to use external libraries and devices. With this enhancement, Tally.ERP 9 can now act as a COM Client.

Definition - COM Interface

The definition 'COM Interface' has been introduced to accept the external DLL/EXE details like Project name, Class name, Function name and other required parameters.

Syntax

```
[COM Interface : <COM Interface name>]
```

Where,

<COM Interface name> is the name of the COM Interface definition.

Attributes supported by definition 'COM Interface':

The following attributes are supported in this definition:

- **Attribute - PROJECT**

This attribute is used to specify the name of the project.

Syntax

```
Project : <Project Name>
```

Where,

<Project Name> is the name of the Project/Namespace of the COM Server.

□ Attribute - CLASS

This attribute is used to specify the Class name under the project specified.

Syntax

```
Class : <Class name>
```

Where,

<Class Name> is the name of the Class of the COM server to be used.

□ Attribute - INTERFACE

This attribute is used to specify the interface name under the class that needs to be executed. It corresponds to the name of the function within the DLL Class to be executed.

Syntax

```
Interface : <Interface name>
```

Where,

<Interface Name> is the name of the actual Interface name of the DLL which is to be called.

□ Attribute - PARAMETER

This attribute denotes the list of parameters along with their data types required by the COM Interface. The TDL Data types supported are:

- Number
- Long
- String
- Logical
- Date

Syntax

```
Parameter : <Parameter Name> : <Data type> [: <Parameter Type>]
```

Where,

<Parameter Name> is the name of the Parameter.

<Data Type> is the data type of the parameter being passed.

<Parameter Type> is the nature of the parameter, viz. In (Input), Out (Output) or InOut (Both Input and Output)

For instance:

Parameter of type String for accepting output will be written as:

```
Parameter : Parm1 : String : Out
```

In the absence of the specification of the nature of Parameter, the parameter is defaulted as in parameter



All the parameters must sequentially correspond to the ones accepted by the COM Interface.

□ **Attribute - Returns**

This attribute denotes the return data type of the COM Interface.

Syntax

Returns : **<Return Data Type>**

Where,

<Return Data Type> is the data type of the value returned by the COM Interface.

Let us define the COM Interfaces required for the previous DLL code:

```
[COM Interface : TSPL Smp Addition]
```

```
Project      : MathLib  
Class       : MathClass  
Interface   : Add  
Parameters  : p1: Number : In  
Returns     : Number
```

```
[COM Interface : TSPL Smp Division]
```

```
Project      : MathLib  
Class       : MathClass  
Interface   : Divide  
Parameters  : p1 : Long : In  
Parameters  : p2 : Long : In  
Parameters  : p3 : Long : Out  
Returns     : Long
```

```
[COM Interface : TSPL Smp AddDate]
```

```
Project      : MathLib  
Class       : MathClass  
Interface   : DateFunc
```

```
Parameters : p1 : Date : InOut
```

In this code, 3 COM Interfaces are defined, each for executing 3 different functions inside DLL:

- ❑ **MathLib** is the DLL Project Name
- ❑ **MathClass** is the DLL Class under the Project **MathLib**
- ❑ **Add**, **Divide** and **DateFunc** are the Functions under the Class **MathClass**, which are specified in the Attribute **Interface**.

Action – Exec COM Interface

A new action **Exec COM Interface** has been introduced to invoke the defined COM Interface.

Syntax

```
Exec COM Interface: <COM Interface name> [: <Parameter 1> [: <Parameter 2>..... [: <Parameter N>]...]]
```

Where,

<COM Interface Name> is the name of the COM Interface Definition.

[: <Parameter 1> [: <Parameter 2>..... [: <Parameter N>]...]] are the subsequent parameters, which are passed considering the following aspects:

- ❑ If the parameter corresponds to IN parameter, it can take any expression or constant.
- ❑ If the parameter corresponds to an OUT or an InOut Parameter, then only the variable name must be specified, without prefixing a **##**. In other words, expressions are not supported. The variable, in case of:
 - a) **InOut** Parameter, will send the variable value to the Interface as input, and in return, will bring back the value altered by the Interface.
 - b) **Out** Parameter, will only bring back the updated value from the DLL.

In the previous TDL Code, three COM Interfaces are defined. A function is then called, inside which the action **Exec COM Interface** is used to invoke the COM interface definitions as follows:

```
[Function : TSPL Smp Execute COM Interfaces]

Variable : p1 : Number : 90

Variable : p2 : Number : 102

Variable : p3 : Number : 5

Variable : p4 : Number

Variable : pDate : Date

00 : Exec COM Interface : TSPL Smp Addition : ##p1

10 : Log : $$LastResult

20 : Exec COM Interface : TSPL Smp Division : ##p2 : ##p3 : p4

25 : Log : $$LastResult
```



```

30 : Log : ##p4
40 : Set : pDate : $$Date : "20-04-2013"
50 : Exec COM Interface : TSPL Smp AddDate :pDate
60 : Log : ##pDate

```



If this Action is invoked from within a TDL function, then the Objects created are retained until all the Actions within the function are executed. This behaviour was designed so that multiple functions in a COM Class can be used on the same COM Object (state of COM Object is maintained). For instance, there are some functions of a COM server which depend on each other, and are required to be called in sequence, then the same object needs to be present and functions should be called on that same object to give correctness.

As a Global action, this would create a COM Object once per COM Interface execution. In other words, if there are two functions of a COM Server and they depend on each other, then this action would work only if used within a procedural Code.

Function - \$\$COMExecute

This function is similar to the Action **Exec COM Interface**, except that Interfaces with only In Parameters can be executed with the Function **\$\$COMExecute**. In other words, this Function can only execute a 'COM Interface', if it has no **Out Parameters**.

Syntax

```

$$COMExecute : <COM Interface name>: [<Parameter 1>[:<Parameter 2> [...
                                     [: <Parameter n>]....]]]

```

Where,

<COM Interface Name> is the name of the COM Interface definition.

[<Parameter 1>[:<Parameter 2>[...[:<Parameter n>]....]]] are the IN parameters, which correspond to the parameters specified in the COM Interface definition.

As mentioned earlier, the first function of DLL only takes In Parameters. Hence, the function COMExecute can be used only for the first COM Interface definition in the example shown above.

Example:

```

$$COMExecute : TSPLSmpAddition : ##p1

```

Function - \$\$IsCOMInterfaceInvokable

This function just checks if the 'COM Interface' description which was defined, could be used or not. If the COM class of the interface is not available in the COM server, it would return FALSE; while if the class and the function invoked in the COM class are present, then the interface is invokable, and hence TRUE would be returned.

Syntax

```

$$IsCOMInterfaceInvokable : <COM Interface name>

```

Where,

<COM Interface name> is the name of the COM Interface Definition.

Scope and Limitations:

- Only a COM Server which implements its classes using IDispatch interface (Automation interface of COM), can be used with this.
- For other Native DLLs (DLLs that contain raw processor directly-executable code, e.g., Win32 DLL) or ones which do not comply with the above, another wrapper DLL can be made which makes use of it and exposes the functionality to TDL.
- The data types supported are Long, Number, String, Logical and Date, which are mapped to the following data types in IDL:

IDL Type	Parameter data type in TDL
Signed Int	Long
Double	Number
BSTR	String
Boolean	Logical
Date	Date

- Out parameters are supported in this capability. But, functions which take other data types than specified in the table are currently not supported in TDL, and hence, cannot be used.

2. Developer Mode

In market, there are customers using Tally.ERP 9 for their day-to-day operations. There are also developers/ partners who build solutions within the product to suit the customer requirements. For the entire spectrum of persons using Tally.ERP 9, the only mode of execution available upto now was the default mode. In order to empower the developers of Tally with various tools which will help them to build solutions optimally and efficiently, the very new **Developer Mode** has been introduced.

From Release 4.6 onwards, Tally can operate in 2 modes, viz. **Normal Mode** which will be used by the end users, and the **Developer Mode**, which will be used by the developers. The Developer mode will offer various tools for debugging the code and optimizing the performance of reports.

The various Tools introduced for TDL Developers in Developer Mode are:

1. Profiler

Profiler is a very useful tool in the hands of the TDL Programmer. Usually, any requirement can be achieved in multiple ways using TDL. But, it is highly difficult for the programmers to choose the optimal way. Using Profiler, programmers can check the time taken to execute each TDL artefact, along with the count of how many times they have been executed. This ensures redundancy check, and the code can be optimized for the best performance and user experience.

2. Expression Diagnostics

This is a very handy tool for the TDL Programmer. At times, while writing complex expressions for huge projects, it becomes difficult to identify the expression that has failed. Usually, for debugging such code, TDL Programmers had to resort to invoking User Defined Functions and logging the

values to trace the point of failure. With Expression Diagnostics, now the system automatically dumps every expression along with their resultant value in the log. For expressions which have failed to execute, the resultant value would be set to FAILED. With this, the programmer can easily reach the exact expression that has failed and correct the same without much delay. This would save a lot of programmer's time, which can be used for other projects.

3. Key Recording, Playback and Triggering the Keys

This feature will be useful in instances where developers require to execute certain keystrokes repetitively to test/ retest the code output and confirm if the same is in line with the customer requirement. It will help in doing automated QA and ensuring utmost quality for the customer. For instance, if a data entry screen has been customised by incorporating additional fields, sub-forms, etc., then in order to validate if the data entry performance is affected, one can record the keystrokes for saving a voucher, and replay them as many times as required.

4. Onscreen Tooltip

While developing extensions on Tally, developers usually navigate through default TDL to locate the appropriate definition name and alter their attributes within their code. At times, it becomes very difficult for the developers to identify the right field names, since there are several options at various stages and finding the right ones requires a lot of effort. Hence, to make developer's life easier, a very critical tool, i.e., the Onscreen Tooltip, has been introduced. When the mouse pointer is placed on a Field, the definition name of the Field is displayed. If the pointer points at a place where no field exists, the definition name of the Report is displayed.

All these Tools/Enhancements will be available only if the developer executes Tally in Developer Mode, using the Command Line Parameter **DevMode**.

□ Command Line Parameter - DevMode

Command Line Parameter **DevMode** has been introduced to execute Tally in Developer mode.

Syntax

```
<Tally Application Path>\Tally.exe /DevMode
```

Example:

```
C:\Tally.ERP9\Tally.exe /DevMode
```

On invoking Tally in Developer Mode, the Tally Screen is as shown below:



Figure 1. Tally.ERP 9 in Developer Mode



Developers must not execute the above at the Client end unless required, because if the client continues to work in this mode, it might affect the performance adversely.

□ **Function - \$\$InDeveloperMode**

This function is used to check if Tally is currently working in Developer mode. It returns TRUE if the Tally application is in Developer Mode, else returns FALSE.

This tool can be used when the developer needs to write some additional code for testing purpose, so that the testing code does not appear to the end user, thereby executing only if the product is running in Developer Mode.

Syntax

`$$InDeveloperMode`

Example:

```
Option : DevModeDefaultMenu : $$InDeveloperMode
```



Extensive use of this function in the TDL code may lead to performance issues even in Normal mode.

Let us discuss the tools/capabilities and their usage, in detail:

2.1 Profiler

As briefed before, Profiler is a useful tool which helps the developers to check the performance of the TDL Program, thus optimizing the code. It returns the execution time and count of the various TDL artefacts. It gathers information of Collections, User Defined Functions and Expressions.

The **steps** to get the profiler information are:

- Start the Profiler
- Execute the desired Report
- Dump the Profiler and/or Stop the Profiler, with the file name

The dumped profiler information when opened in Textpad, is as shown below:

```

**** TDL Functions Profiler ****
-----
**** TDL Expressions Profiler ****
-----
Count      .      Time taken(ns)      .      Req Type      .      Requester
-----
1           .      1                   .      Report        .      CFBK Rep
1           .      1                   .      Key           .      DSP AutoColumns
210        .      1                   .      Collection    .      DSP AutoColumns
91         .      1                   .      Collection    .      CFBK Summ Voucher
91         .      1                   .      Collection    .      CFBK Summ Voucher
2         .      1                   .      (null)        .      (null)
7         .      1                   .      (null)        .      (null)
54         .      1                   .      (null)        .      (null)
54         .      1                   .      (null)        .      (null)
44         .      1                   .      (null)        .      (null)
54         .      1                   .      (null)        .      (null)
630        .      2                   .      Field         .      CFBK Rep Party
2715      .      3                   .      Collection    .      CFBK Voucher
630        .      3                   .      Field         .      CFBK Rep Party
2715      .      11                  .      Collection    .      CFBK Voucher
1         .      146                 .      Form          .      CFBK Rep
630        .      146                 .      Field         .      CFBK Rep Party
630        .      146                 .      Field         .      CFBK Rep Party
-----
**** TDL Expressions Profiler ****
-----
**** TDL Collection Profiler ****
-----
Count      .      Time taken(ns)      .      Req Type      .      Requester
-----
1           .      1                   .      Key           .      DSP AutoColumns
1           .      139                 .      Field         .      CFBK Rep Party
1           .      141                 .      Field         .      CFBK Rep Party
1           .      146                 .      Variable      .      FName
1           .      146                 .      Variable      .      FName
1           .      147                 .      Part          .      CFBK Rep
1           .      151                 .      Part          .      CFBK Rep
-----

```

Figure 2. Dumped Profiler Information

As seen in the figure, the profile information shows the time taken for evaluating every expression as well as how many times (Count) the same expression or collection was evaluated/ gathered.

Developers are already aware that apart from performing various arithmetic operations, Calculator Panel can also be used to issue select Queries like Select \$Name, \$Parent from Ledger, Select * from Company, etc. Now, the **Calculator Pane** can also be used by developers for Profiling, Debugging, Key Recording and Playing back by setting various modes.

Commands used for Profiling

The following Calculator Pane Commands are supported for the profiling information:

Profiler Mode

It sets the mode to Profiler, which means that Profiling Commands will be accepted. It provides certain calculator pane commands to the developer in order to check the performance of code.

Syntax

MODE: Profile



```
Calculator Ctrl + N X
1> Mode:Profile
2 Mode set to Profile
3> Start
4> Stop
5> Dump
6>
```

Figure 3. Issuing commands in the Calculator Pane in Profiler Mode

Once mode is set to Profile, commands issued in the Calculator Pane work in Profiling Context.

Start

This command starts gathering the count and time taken for evaluating a TDL Artefact in memory.

Syntax

START

Stop

This command is used to end the profiling.

Syntax

STOP

Dump

This command is used to dump the collected profile data to the file **tdlprof.log**. It also clears the memory once the data is dumped.

Syntax

DUMP

Dumpz

This command is used to dump the collected profile data, including artefacts which have consumed negligible time, i.e., zero processing time, into the file **tdlprof.log**. It also clears the memory after updation to the file.

Syntax

DUMPZ

Status

It checks the status of the profiler, and returns the statement **Profiler is ON** or **Profiler is OFF**.

Syntax

STATUS

Reset

This command is used to clear the existing profile data from the Memory.

Syntax

RESET

Help

This command gives the list of Profiler commands, with description of their purpose.

Syntax

`HELP`

Actions used for Profiling

Apart from Calculator Pane Commands, there are several other TDL Actions provided, to programmatically execute the profiling operations. They are:

□ Action - Start Profile

This action is used to start the profiling. The Count, Time and other usage information of every function, collection, etc. gathered along with expressions within the report, are profiled in memory.

Syntax

`Start Profile`

Example:

```
[Button : Start Profiling]
```

```
Key      : Alt + S
```

```
Action : Start Profile
```

□ Action - Dump Profile

This action is used to dump all the profiled information to the file. It also clears the memory after dumping the information.

Syntax

`Dump Profile [: <File Name>[: <Logical Value>]]`

Where,

<File Name> is the name of the file to which the information has to be written. In the absence of the 'File Name' Parameter, the default file updated will be `tdlprof.log`

<Logical Value> if set to YES, the 'Dump Profile' action also includes zero time-taking artefacts. If it is enabled, the action is similar to calculator pane command **DumpZ**, else it is similar to **Dump**.

Example:

```
[Button : Dump Profiling]
```

```
Key      : Alt + R
```

```
Action : Dump Profile : "Profiled @ " + @@SystemCurrentTime
```

□ Action - Stop Profile

This action is used to stop the profiling. If the Optional parameter 'File Name' is passed, then information is also dumped into the file, without requiring the action Dump Profile.

Syntax

`Stop Profile [: <File Name>]`

Where,

<File Name> is the name of the file to which the information has to be written.

Example:

```
[Button : Stop Profiling]
```

```
Key      : Alt + T
```

```
Action  : Stop Profile
```

Functions used for Profiling

The following function has been introduced to support profiling:

□ **Function - \$\$IsProfilerOn**

This function is used to check the current status of the TDL Profiler. It returns logical value TRUE, if the status of the profiler is ON.

Syntax

```
$$IsProfilerOn
```

Example:

```
[Function : Switch Profiler OnOff]
```

```
10 : If : $$IsProfilerOn
```

```
20 : Stop Profile
```

```
30 : Else
```

```
40 : Start Profile
```

```
50 : End If
```

2.2 Expression Diagnostics

This will help the developers to debug the TDL Program much faster by evaluating the complex expressions and logging the values evaluated at every stage. In other words, this feature would provide the breakup of the expression, the result of each sub-expression, as well as the expression which has failed to evaluate.

The **steps** to get the Expression Diagnostics information are:

- Start the Expression Diagnostics
- Execute the desired Report
- Dump the information and/or Stop the Expression Diagnostics with the file name.

The dumped debugger information, when opened in Textpad, is as shown below:


```

=====
**** Debug Log ****
=====
Expression      Result      Obj Type      Obj Name
-----
$$SetAutoColumns:FNams      Yes
##DSPRepeatCollection      CFBK Party
$$IsSales:$VoucherTypeNone      No      Voucher      ID:4668
$VoucherTypeNone      Payment      Voucher      ID:4668
$$IsSales:$VoucherTypeNone      No      Voucher      ID:4718
$VoucherTypeNone      Purchase      Voucher      ID:4718
$$IsSales:$VoucherTypeNone      Yes      Voucher      ID:4721
$VoucherTypeNone      Sales      Voucher      ID:4721
$PartyLedgerNone      Raj & Co      Inventory Entry      Item 1
$BilledQty      10 Nos      Inventory Entry      Item 1
$$IsSales:$VoucherTypeNone      Yes      Voucher      ID:8
$VoucherTypeNone      Sales      Voucher      ID:8
=====

```

Figure 4. Dumped Debugger Information

As seen in the figure, each expression in the developer's code, as well as the default codes are evaluated and the result values are shown for the purpose of debugging. If any expression evaluation fails, the Result would display as **Failed**.

Commands used for Expression Diagnostics

The Calculator pane commands used for Expression Diagnostics are as follows:

Debugger Mode

This sets the mode to **Debug**, which means that any Debugging Commands will be accepted.

It provides certain calculator pane commands in the hands of developer in order to diagnose the errors in the code, as well as to evaluate and/or set the values to the environment variable.

Syntax

MODE: Debug



Figure 5. Issuing Debugging commands in the Calculator Pane

Once mode is set to Debug, commands issued in the Calculator Pane will work in Debug Context.

Start

Start command is used to start diagnosing the data. Subsequent to issuing this command, when any report is viewed, the data will start gathering every expression along with their values, and will be updated in the log file later.

Syntax

START

Dump

This command dumps the collected Expressions data to the file **tdldebug.log**. It also clears the memory.

Syntax`DUMP`**Status**

This command is used to check the status of the debugger and returns the statement **Expression Debugger is ON** or **Expression Debugger is OFF**.

Syntax`STATUS`**Eval**

This calculator pane command is used for evaluating an expression.

Example:

```
EVAL : ##SVFromDate
```

In this example, the value of **SVFromDate** will be returned in the Calculator Pane.

Set

A variable value can also be set from within a Calculator Pane. This feature helps the developer to set the variable value in the Calculator Pane itself and check the behavioral change. This will speed up the testing process of the Developer by not requiring him to write the code, create an interface element to alter the variable values, and then check the same.

Example:

```
SET : DSPShowOpening : Yes
```

This will set the value of the variable **DSPShowOpening**. For instance, prior to viewing Trial Balance, one can set the value of this variable in the Calculator Pane and the Trial Balance will be displayed with the Opening Column.

Print

Print displays the value of a system variable.

Example:

```
Print : SVFromDate
```

This will return the **SVFromDate** value.

Reset

This command is used to clear the existing diagnosis data from Memory.

Example:

```
RESET
```

Stop

This command will stop further diagnosis of data.

Example:

```
STOP
```

Help

This gives the list of Debug commands, with description of their purpose.

Example:

```
HELP
```

Actions used for Expression Diagnostics

Apart from Calculator Pane Commands, there are several TDL Actions provided to programmatically execute the debugging operations. They are:

□ Action - Start Debug

This action is used to start debugging. All expressions evaluated would be debugged. The information collected can be dumped to a file anytime.

Syntax

```
Start Debug
```

Example:

```
[Button : Start Debugging]
```

```
Key      : Alt + D
```

```
Action : Start Debug : "Debugged @ " + @@SystemCurrentTime
```

□ Action - Dump Debug

This action is used to dump all debugging information to the file. It also clears the diagnosis data from the memory, once dumped.

Syntax

```
Dump Debug:<File Name>
```

Where,

<File Name> is the name of the file to which the information has to be written.

Example:

```
[Button : Dump Debugging]
```

```
Key      : Alt + E
```

```
Action : Dump Debug : "Debugged @ " + @@SystemCurrentTime
```

□ Action - Stop Debug

This action is used to stop debugging. If the file name is passed, then the information is also dumped into the specified file.

Syntax

```
Stop Debug [: <File Name>]
```

Where,

<File Name> is the name of the file to which the information has to be written.

Example:

```
[Button : Stop Debugging]

Key      : Alt + O

Action  : Stop Debug : "Debugged @ " + @@SystemCurrentTime
```

Functions used for Expression Diagnostics

The following function has been introduced to support Debugging operations:

□ **Function - \$\$IsDebuggerOn**

This function is used to check the current status of the Expression Debugger. It returns logical value TRUE if the status of the Debugger is ON.

Syntax

```
$$IsDebuggerOn
```

Example:

```
Inactive : NOT $$IsDebuggerOn
```

2.3 Key Recording and Playback

The recording and playback features will allow the developers to record all the keyboard keys and later play/replay them, as and when required.

This will help the developers to automate and replay certain keystrokes repeatedly during testing of the code for various performance testing needs, as well as while debugging a project, and automate multiple steps required to do the needful.

The **steps** to get the Key Recording done are:

- Start the Recording
- Navigate through the sequence of key strokes
- Dump the Recorded information
- Stop the Recording

The dumped recorded information when opened in Textpad, is as shown below:

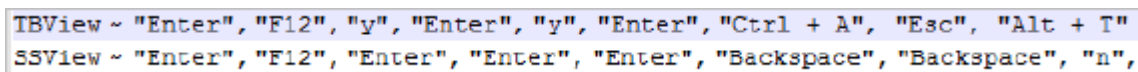


Figure 6. Dumped Recorded Information

As seen in figure, the macros are appended to the file **macros.log** within the application path. Comma-separated Keys are written in the file against each macro name separated with the separator symbol specified while dumping the macro from the memory to the file.

Commands used for Key Recording and Playback

The following Calculator Pane Commands are supported for Key Recording and Playback:

Record Mode

Record Mode gives various commands to help the developer in recording keys. The mode should be set by specifying the mode as Record.

Syntax

```
MODE : Record
```

The following commands can be specified in the Calculator Pane once the Mode is set to Record:

Start

This command is used to start recording a macro.

Syntax

```
Start[: <Macro name>]
```

Where,

<Macro name> is the name assigned to a macro. In the absence of a Macro Name, it is defaulted to the name **Macro**.

Example:

```
START : BSVIEW
```

Replay

It is used to replay the recently recorded macro from the memory, but before the same is dumped into a file. If the macro is already dumped into a file, the same is cleared from the memory and will not be available for Replay. The macro from the file can then be replayed using File I/O Approach and Action 'Trigger Key' (explained later), but cannot be replayed through Calculator Pane.

Syntax

```
REPLAY
```

Pause

This command is used to pause the recent recording.

Syntax

```
PAUSE
```

Resume

This command is used to resume the recently paused recording.

Syntax

```
RESUME
```

Dump

This command is used to dump the recording to the file **macros.log**. The macro name will be separated from the keys with the 'separator character' specified as the parameter. It also clears the keys in the memory.

Syntax

```
DUMP[: ~]
```

Stop

This command is used to stop recording the recent macro.

Syntax

STOP

LS

This command is used to list the macros which are recorded and available in the memory. The dumped macros will not be listed as they are cleared from the memory as soon as they are dumped in the file.

Syntax

LS

Help

This command provides a list of all the recording commands, with description of their purpose.

Syntax

HELP



- *To reset command prompt to regular behaviour, you need to mention **MODE**.*
- *To perform all the above actions or Calculator Pane Commands, Tally must be executed in **DevMode**.*

Actions used for Key Recording and Playback

Apart from Calculator Pane Commands, there are several TDL Actions provided to programmatically execute the Key Recording operations. They are:

□ Action - Start Recording

This action is used to start recording every key entered in the memory, with the specified macro name. In the absence of optional parameter Macro Name, default name assumed will be 'Macro'.

Syntax

Start Recording[: <Macro Name>]

Where,

<Macro Name> is the name of the macro to be recorded in memory.

Example:

```
[Button: Start Recording]
```

```
Title: "Start"
```

```
Key: Alt + C
```

```
Action: Start Recording: "BS View"
```

□ Action - Pause Recording

This action pauses the recording, which can be resumed further. For instance, while recording multiple Vouchers, we might have to run through the Report to check the Number of vouchers, Amount, etc., and then resume recording the Vouchers.

Syntax

Pause Recording

Example:

```
[Button : Pause Recording]

Title   : "Pause"

Key     : Alt + U

Action  : Pause Recording
```

□ Action - Resume Recording

This action resumes the paused recording.

Syntax

Resume Recording

Example:

```
[Button : Resume Recording]

Title   : "Resume"

Key     : Alt + M

Action  : Resume Recording
```

□ Action - Dump Recording

This action dumps all the recordings to a file with the given separator. Each recording is dumped with its name and keys. This also clears the keys in the memory.

Syntax

Dump Recording : <Macro Name> : <Separator between keys>

Where,

<Macro Name> is the name of the macro recorded in memory.

<Separator between keys> is the separator to be used between the recording name and keys.

Example:

```
[Button : Dump Recording]

Title   : "Dump"

Key     : Alt + G

Action  : Dump Recording : "BSView" : ","
```

□ Action - Stop Recording

This action stops the recording.

Syntax

Stop Recording

Example:

```
[Button : Stop Recording]
```

```
Title : "Stop"
```

```
Key : Alt + N
```

```
Action : Stop Recording
```



The recordings once dumped in a file against a name using the above actions, can be replayed by reading the file using File I/O approach and Triggering the keys in a loop using the Action 'Trigger Key', which will be covered ahead.

□ Action - Trigger Key

When the macro keys are recorded using Key Recording Actions or when they are dumped into the Macros File from the Calculator Pane Command; in order to play them back, one needs to make use of the Action **Trigger Key**, which sends the list of keys in sequence to the system as if an operator is pressing those Keys. The Keystrokes of a required macro can be copied from Macro Log file and pasted against the 'Trigger Key' Action, which triggers all those Keys in sequence as required.

Also, Trigger Key accepts a value with Inverted Quotes, which means - trigger this as a value in the current field. For example, V, "Cash", etc. If the triggered keys enclosed within quotes (" ") are executed from a menu, they will be considered as menu keys. For example, "DAS" from Gateway of Tally menu will take the user through Display -> Account Books -> Sales Register.

Syntax

Trigger Key : <Comma Separated Keys/Values>

Example:

```
Trigger Key : V, F5, Enter, "Cash", Enter, "Conveyance", Enter, "50", Ctrl+A
```

Functions used for Key Recording

The following function has been introduced for key recording:

□ Function - \$\$Recorder Status

This function is used to check the status of the recorder. It returns a String value to indicate whether it is in Started, Stopped or Paused mode.

Syntax

`$$RecorderStatus`

2.4 Onscreen Tooltip

At times, it becomes difficult for the developer to navigate through the default TDL and identify the fields that are used in the target report. Hence, the **Onscreen Tooltip** feature has been provided for the developers, who can identify the Field/ Report Names by placing the mouse pointer wherever required.

This behavior will work only in **Developer Mode** which means that the Tally Application must have started with a Command Line Parameter **/DevMode**.

As seen in the following screen capture, the **Field Name** is displayed as a **Tooltip**, when the cursor is pointed on the desired field within the Report.

2-4-2010	NextGen Systems	Purchase	4
3-4-2010	Computer Junction	Receipt	
3-4-2010		Purchase	5
3-4-2010	HP India Ltd.,	Purchase	6

Figure 7. Field Name displayed as Tooltip when cursor points to a Field

In the next screen capture, it can be seen that the **Report Name** is displayed as a **Tooltip**, when the cursor is not pointed on any of the fields within the Report.

Day Book (In Developer Mode)		ABC Company Ltd
Day Book		
Date	Particulars	

Figure 8. Report Name displayed when the cursor does not point on a Field

What's New in Release 4.5

Introduction

The TDL language is enriched with new capabilities, based on emerging needs, from time to time. With the introduction of the new product Tally.Server 9, in this release, a few generic actions and functions have been introduced.

1. Platform Functions

1.1 Function - `$$IsAccountingVch`

This function checks if the specified voucher type is Accounting Voucher or not. It returns a logical value.

Syntax

```
$$IsAccountingVch : <VoucherTypeName>
```

Where,

<VoucherTypeName> is the name of the voucher type.

Example:

```
Set As : $$IsAccountingVch : ##TSPLSmp_Information
```

1.2 Function - `$$IsInvVch`

This function checks if the specified voucher type is Inventory voucher or not (excluding Order vouchers). It returns a logical value.

Syntax

```
$$IsInvVch : <VoucherTypeName>
```

Where,

<VoucherTypeName> is the name of the voucher type.

Example:

```
Set As : $$IsInvVch : ##TSPLSmp_Information
```

1.3 Function - `$$IsPayrollVch`

This function is used to check if the specified voucher type is Payroll Voucher or not. It returns a logical value.

Syntax

```
$$IsPayrollVch : <VoucherTypeName>
```

Where,

<VoucherTypeName> is the name of the voucher type.

Example:

```
Set As : $$IsPayrollVch : ##TSPLSmp_Information
```

1.4 Function - \$\$IsOrderVch

This function is used to check if the specified voucher type is Order Voucher or not. It returns a logical value.

Syntax

```
$$IsOrderVch : <VoucherTypeName>
```

Where,

<VoucherTypeName> is the name of the voucher type.

Example:

```
Set As : $$IsOrderVch : ##TSPLSmp_Information
```

1.5 Function - \$\$IsProdTallyServer

This function is used to check whether the product is Tally.Server 9 or not. It returns TRUE if the product is Tally.Server 9.

Syntax

```
$$IsProdTallyServer
```

Example:

```
[Function : TSPL Smp IsProdTallyServer]
```

```
00 : If      : $$IsProdTallyServer
```

```
10 : Msg Box : "Server Check" : "The Current Product\n is Tally Server"
```

```
20 : ELSE    :
```

```
30 : MSGBOX  : "Server Check" : "The Current Product\n is not Tally Server"
```

```
40 : ENDIF
```

1.6 Function - \$\$ExcelInfo

This function is used to get the Excel 'version' and to check whether 'XLSX' format is supported.

Syntax

```
$$ExcelInfo : <Keyword>
```

Where,

<Keyword> can be **IsXLSXSupported** or **Version**. The keyword **IsXLSXSupported** returns TRUE, if the format "xlsx" is supported, while **Version** returns the Excel version number.

Example:

```
Set As : $$ExcelInfo : IsXLSXSupported
```

1.7 Function - \$\$IsServiceRunning

This function is used to check if the specified service is running or not. It returns TRUE if the service is running.

Syntax

```
$$IsServiceRunning : <Service Name>
```

Where,

<Service Name> can be any expression which evaluates to the name of the service.

Example:

```
[Function : TSPL Smp IsServiceRunning]
```

```
00 : If : $$IsServiceRunning : "Tally.Server 9"  
10 : MSG Box : "Service Check" : "The Current Service \n is Running"  
20 : ELSE:  
30 : MSGBOX : "Service Check " : "The Current Service \n is not Running"  
40 : ENDIF
```

1.8 Function - \$\$IsServiceInstalled

This function is used to check if the specified service is installed on the system or not. It returns TRUE if the service is installed.

Syntax

```
$$IsServiceInstalled : <Service Name>
```

Where,

<Service Name> can be any expression, which evaluates to the name of the service.

Example:

```
[Function : TSPL Smp IsServiceRunning]
```

```
00 : If : $$IsServiceInstalled : "AppMgmt"  
10 : Msg Box : "Service Check" : "The Current Service \n is installed"  
20 : ELSE :  
30 : MSGBOX : "Service Check" : "The Current Service \n is not Installed"  
40 : ENDIF
```

1.9 Function - \$\$ReadINI

The function is used to read the INI file, and get the value of any parameter in the INI.

Syntax

```
$$ReadINI : <Path\File Name> : <Section Name> : <Parameter> [: <Index>]
```

Where,

<Path\File Name> is the filename of the INI file, along with the path.

<Section Name> is the section name in the INI file.

<Parameter> is the Parameter whose value is to be fetched from the INI.

<Index> is an optional parameter. It can be used when multiple values for the same parameter are accepted.

Example:

```
Set As : $$ReadINI : "C:\Tally.ERP9\tally.ini": "TALLY": "User TDL"
```

1.10 Function - \$\$IsUserAllowed

This function verifies and returns TRUE if user is allowed to perform the specified operation on current Tally.Server9.

Syntax

```
$$IsUserAllowed : <Username> : <Operation> : <Tally Server Name>
```

Where,

<User Name> is the name of the user.

<Operation> is the operation that the user wants to perform. This can be any one of BackUp, Restore, Rewrite, Create Company, Split Company and Monitor Tool.

<Tally Server Name> is the name of the Tally.Server 9

Example:

```
Set as : $$IsUserAllowed : $User_name1 : "Restore" : ##SvTallyServer
```

1.11 Function - \$\$IsTSAuthorised

This function checks if security is enabled on the specified Tally.Server 9. It returns TRUE if security is enabled, else it returns FALSE.

Syntax

```
$$IsTSAuthorised : <TallyServer Name>
```

Where,

<Tally Server Name> is the name of the Tally.Server 9.

Example:

```
[Function : TSPL Smp IsTSAuthorised]
```

```
00 : If : $$IsTSAuthorised : ##SvTallyServer
10 : MSG Box : "Security Check" : "Security Control is Enabled"
20 : ELSE :
30 : MSGBOX : "Secuirity Check " : "Security Control \n is not Enabled"
40 : ENDIF
```

1.12 Function - \$\$TSPingInfo

This function is used to retrieve the information related to Tally.Server 9 like License mode, Number of license subscription days left, etc.

Syntax

```
$$TSPingInfo : <TallyServer> : <Keyword>
```

Where,

<Tally Server Name> is the name of the Tally Server.

<Keyword> can be any one of the keywords- Iseducational, LicenseExpiryDaysLeft and HasINFO.

- **HasINFO** returns TRUE or FALSE depending on whether the client is able to get information for this Tally.Server 9 (otherwise Iseducational and LicenseExpiryDaysLeft fail in a TDL expression).
- **Iseducational** returns TRUE if the Tally.Server 9 is running in Educational mode.
- **LicenseExpiryDaysLeft** returns the number of subscription days remaining.

Example:

```
Set As : $$TSPingInfo : ##SvTallyServer : ##Ping_Operation
```

1.13 Function - \$\$IsTSCompany

This function checks whether the current company is opened through Tally.Server 9 or not. It returns a logical value.

Syntax

```
$$IsTSCompany : <Company Name>
```

Where,

<Company Name> is the name of the company.

Example:

```
Set As : $$IsTSCompany : $CompanyStorage
```

1.14 Function - \$\$SelectedNonTSCmps

This function returns the total number of companies which are not loaded via Tally server, i.e., companies from local or shared data folders.

Syntax

```
$$SelectedNonTsCmps
```

Example:

```
[Function : TSPL Smp IsProdTallyServer]
```

```
00 : If : $$SelectedNonTsCmps
```

```
10 : Msg Box : "Company Check" : $$SelectedNonTSCmps;+
```

```
"Total Companies not loaded via Tally server is \n"
```

```
20 : ELSE:
30 : MSGBOX : " Company Check " : "All Companies Have been +
      Opened Via Tally Server "
40 : ENDIF
```

1.15 Function - \$\$IsTSPath

This function checks whether the given path is Tally.Server path or not.

Syntax

```
$$IsTSPath : <Path>
```

Where,

<Path> is any expression, which evaluates to a Tally Server Name or a Data Location Name.

Example:

```
Set As : $$IsTSPath : "Data1:"
```

2. Action Enhancements

2.1 Action - DisconnectUser

This action is used to disconnect the users from companies that are accessed from Tally.Server9. It will display a warning message to the clients to close the company within 2 minutes, before forcing a close.

Syntax

```
DisconnectUser : <Tally Server Name> : <Company Name> : <User Name>
```

Where,

<Tally Server Name> is the name of the Tally.Server9, from which the companies are being accessed.

<Company Name> is the name of the company to be disconnected. '*' can be used to specify all companies.

<User Name> is the name of the user to be disconnected from the company specified. '*' can be used to specify all the users across companies.

Example:

```
Action : DisconnectUser : "HoServer" : * : "abc@abc.com"
```

2.2 Action - ForceDisconnectUser

It is used to forcefully disconnect users from companies that are accessed from Tally.Server 9.

Syntax

```
ForceDisconnectUser : <TallyServer Name> : <Company Name> : <User Name>
```


Where,

<Tally Server Name> is the name of the Tally.Server from which the companies are being accessed.

<Company Name> is the name of the company to be disconnected. '*' can be used to indicate 'all companies'.

<User Name> is the name of the user to be disconnected from the company specified. '*' can be used to indicate 'all users across companies'.

Example:

```
Action : ForceDisconnectUser : "HoServer" : * : "abc@abc.com"
```

2.3 Action - StartService

This action is used to start the specified service.

Syntax

```
StartService : <Service Name>
```

Where,

<Service Name> can be any expression which evaluates to the name of the service.

Example:

```
Action : Start Service : "Tally.Server 9 - 1"
```

2.4 Action - StopService

This action is used to stop the specified service.

Syntax

```
StopService : <Service Name>
```

Where,

<Service Name> can be any expression which evaluates to the name of the service.

Example:

```
Action : Stop Service : "Tally.Server 9 - 1"
```

2.5 Action - WriteINI

This action is used to Add/Alter the value of any parameter in the specified INI file.

Syntax

```
WriteINI : <Path\FileName> : <SectionName> : <Parameter> :  
[<Value> : [:<Index>]]
```

Where,

<PathFile Name> is the path and filename of the INI file

<Section Name> is the section name in the INI file

<Parameter> is the name of the parameter whose value is to be set in the INI.

<Value> is the value to be set for the given parameter. In the absence of any value, the existing value will be removed.

<Index> is an optional parameter. It can be used when multiple values are accepted for the same parameter. In the absence of the index parameter, the value of the last index will be updated.

Example:

```
Action : WriteINI : "C:\Tally.ERP9\tally.ini" : "TALLY" : "UserTDL" : "Yes"
```

What's New in Release 3.62

1. Multiple Orientation Support for Printing

Currently, Tally supports printing of multiple forms within a single report in single orientation only, i.e., all the forms within a report can be printed either in Portrait or in Landscape format only, as specified in the report.

In order to support printing of multiple forms within a single report in different orientations, the behaviour of Variable **SVPrintOrientation** has been enhanced.

To achieve this behaviour, the local declaration of the Variable 'SVPrintOrientation' at 'Report' level is mandatory, and its value must be set using the form level attribute 'Set Always' in individual forms.

This will be very useful in scenarios where multiple forms are being printed from a single report. For example, from a Payment Voucher, one needs to print both Voucher and Payment Advice. However, the Payment Voucher needs to be printed in 'Portrait' form, and the Payment Advice in 'Landscape'.

Example:

```
[Report : TSPL Smp VarSVPrintOrientation Extended]
    Form : TSPLSmpVarSVPrintOrientationFrm + TSPLSmpVarSVPrintOrientationFrm2
;; Mandatory Local declaration of Variable SV Print Orientation
    Variable : SV Print Orientation : String
[Form : TSPL Smp VarSVPrintOrientation Frm1]
;; Orientation of this Form is set to 'Portrait'
    Set Always : SV Print Orientation : "Portrait"
[Form : TSPL Smp VarSVPrintOrientation Frm2]
    Use : TSPL Smp VarSVPrintOrientation Frm1
;; Orientation of this Form is set to 'Landscape'
    Set Always : SV Print Orientation : "Landscape"
```

In this example, the report '**TSPLSmpVarSVPrintOrientation Extended**' has two forms, viz. **TSPLSmpVarSVPrintOrientationFrm1** and **TSPLSmpVarSVPrintOrientationFrm2**.

Based on the value set to the variable 'SV Print Orientation', both the forms are being printed in the respective orientation.

What's New in Release 3.61

1. Action Enhancements

1.1 Action - Browse URL Ex

As we are aware, there is an action **Browse URL/Execute Command** used to open a web page or invoke an external application. There may be cases where subsequent actions are dependent on the completion of the previous action, i.e., the closure of external application. For example, the current action is used to execute an external file, which unzips/ extracts various other files. The subsequent actions use these extracted files to process further. In such cases, **Browse URL**, when used, will trigger the requested application and continue executing the subsequent actions.

Hence, in order to bring sync between the calling and the called application, a new action **Browse URL Ex/Execute Command Ex** has been introduced. The Action **Browse URL Ex**, when triggered, waits till the external application is closed, and then allows the application to resume with the subsequent action. Similar to 'Browse URL', the Action 'Browse URL Ex' can be used to:

- ❑ Open a web Page in the browser
- ❑ Open an external file with its associated application
- ❑ Run an executable application
- ❑ Open a folder in explorer



BrowseURLEx is useful for URL, folder and executable without extension (e.g., tally instead of tally.exe), and it has similar behaviour as Browse URL.

Syntax

BrowseURL Ex : <URL/File path/executable file path/folder path> +
[:<Command line Parameters>]

Where,

<URL/File path/executable file path> can be:

- ❑ A URL, which can be opened in a browser
- ❑ A file, which is to be opened with its associated application
- ❑ An executable file, which is to be run/executed
- ❑ A folder, which is to be opened in the explorer

<Command line Parameters> is an optional parameter. For example, Zip application may need parameter **d** to decompress, **c** to compress, etc.

Example:1

To open a URL in browser:

```
Action : Browse URL Ex : "http://google.com"
```

Example:2

To open a pdf file in pdf reader:

```
Action : BrowseURL Ex : "C:report.pdf\"
```

In this example, report.pdf will be opened in default PDF reader of the system. This can be useful while reading a report, which is exported in PDF format.

Example:3

To open an executable file and wait for it to complete:

```
Action : BrowseURL Ex: "C:\7zip.exe": "d software.7z"
```

In this example, 7zip is opened and the application waits until it finishes, i.e., the running application first waits for 7zip.exe to finish decompressing of software.7z, and then it proceeds further.

Example:4

To open a folder:

```
Action : BrowseURL Ex : "C:\abc"
```

2. Function Enhancements

2.1 Function - \$\$FileReadRaw

We have a function **\$\$FileRead** to read the contents from a text file, which was designed to ignore quotes, comments, spaces, etc., while reading the entire line. Now, a new function **\$\$FileReadRaw** has been introduced, similar to the Function **\$\$FileRead**, except that the Function **\$\$FileReadRaw** can read lines with:

- Quotes
- Comment characters (; /* */)
- Spaces & Tabs

Syntax

```
$$FileReadRaw[:<Number>]
```

Where,

<Number> denotes the number of characters to be read.

Example:

```
[Function : Test FileReadRaw]
```

```
Variable : GetPath : String
```

```
Variable : Get_Download_FileLine : String
```

```
000 : Set          : GetPath    : "C:\TextSource.txt"
010 : Open File   : ##GetPath : Text : Read
020 : While       : (TRUE)
030 : Set          : Get_Download_FileLine : $$FileReadRaw
```

Using this function, we can read lines with quotes, comment characters, spaces, tabs, etc. If **\$\$FileReadRaw** is specified with parameter, the behaviour is same as that of Function **\$\$FileRead**. If specified without parameters, the entire line is read without ignoring quotes, spaces, etc.

What's New in Release 3.6

1. Collection Enhancements

1.1 New methods LastModifiedDate and LastModifiedTime for File Properties

In 'Collection' definition, Directory as a Data Source was supported in Release 3.0, where the properties of the file, i.e., Name, FileSize, IsDirectory, IsReadOnly and IsHidden, were supported as the Methods. In **Release 3.6**, two new methods called **LastModifiedDate** and **LastModifiedTime** have been introduced, to extract additional file properties in Tally.

This can be very useful in Integration scenarios with Tally using external Files. The last imported date and time can be validated against the Last Modified Date and Time of the File prior to importing from the file.

Method - \$LastModifiedDate

\$LastModifiedDate – It returns the date on which the file was last altered. The format supported is *dd-mm-yyyy*.

Syntax

`$LastModifiedDate`

Example:

```
[Collection : ListofFiles]

    Data Source : Directory : "C:\"

    Format      : $Name, 25

    Format      : $FileSize, 15

    Format      : $IsReadOnly, 15

    Format      : $LastModifiedDate, 15
```

Here, \$LastModifiedDate will return the date on which the file was last modified, e.g., **27-May-2012**

Method - \$LastModifiedTime

\$LastModifiedTime – It returns the time at which the file was last altered. The format supported is *hh:mm:ss (24 hours)*

Syntax

`$LastModifiedTime`

Example:

```
[Collection : ListofFiles]

Data Source : Directory : "C:\\"

Format      : $Name, 25

Format      : $FileSize, 15

Format      : $IsReadOnly, 15

Format      : $LastModifiedDate, 15

Format      : $LastModifiedTime, 15
```

Here, \$LastModifiedTime will return the time at which the file was last modified, e.g., **16:28:18**

Following screen capture displays the last modified date and the last modified time in the table:

File Listing				
File Name	File Size	Read Only	Last Modified Date	Last Modified Time
ReleaseNotes.txt	1061	No	20-Dec-2007	19:36:12
Temp634304420579825782.xml	2012	No	13-Jan-2011	17:07:00
Test1.txt	3615	No	17-May-2012	17:56:21
Test.txt	171	No	28-Dec-2011	16:55:35
Test.xml	3611	No	17-May-2012	17:55:44

Figure 1. File properties containing Last Modified Date and Last Modified Time

2. Action Enhancements

2.1 Action - Execute TDL

TDL action **Execute TDL** has been introduced to programmatically load TDL, perform some actions and subsequently, unload the TDL or keep the TDL loaded for the current Tally session.

This can prove to be very useful in cases where the user needs to programmatically associate a TDL on the fly, for performing some operations.

Syntax

```
EXECUTE TDL : <TDL/TCP File Path>[: <Keep TDL Loaded Flag> : +
<Action>: <Action Parameters>]
```

Where,

<TDL/TCP File Path> specifies the path of the TDL/TCP File to be loaded dynamically.

<Keep TDL Loaded Flag> is the Logical Flag to determine if the TDL should be kept loaded after the action is executed. If the value is set to YES, then the TDL will continue to be loaded, else the Executed TDL will be unloaded after the execution of the specified action.

<Action> specifies the global Action to be performed, i.e., Display, Alter, Call, etc.

<Action Parameters> are the parameters on which the Action is performed. For example:

- If the action is **Call**, the Action Parameters contain the Function name along with the Function Parameters, if any.
- If the Action is **Display, Alter**, etc., then the Parameter should be the Report Name.

Example: 1

```
[Function : TDL Execution with Keep TDL Loaded enabled]

00 : Execute TDL : "C:\Tally.ERP9\BSTitleChange.TDL" :+
                                     Yes : Display : Balance Sheet

10 : Display : Balance Sheet
```

The file **BSTitleChange.TDL** contains the following lines of code:

```
[#Report : Balance Sheet]

Title : "TDL Executed Programmatically"
```

In this example, the TDL **BSTitleChange.TDL**, which is used to change the Title of the report Balance Sheet, is loaded dynamically by executing the action 'Execute TDL'. The **Keep TDL Loaded Flag** is set to **YES** in the above code snippet. Based on the subsequent action **Display: Balance Sheet**, the Balance Sheet report will be shown with a new Title. The next statement also displays the Balance sheet. The title for this will again be the same, i.e., the changed title, as the dynamic TDL is still loaded, even after the action 'Execute TDL' has completed its execution.

Example: 2

```
[Function : TDL Execution with Keep TDL Loaded enabled]

00 : Execute TDL : "C:\Tally.ERP9\BSTitleChange.TDL" : No +
                                     : Display : Balance Sheet

10 : Display : Balance Sheet
```

Here, the report **Balance Sheet** would be displayed twice and the title of the first report is "TDL Executed Programmatically", i.e., the changed title as per **BSTitleChange.TDL**; whereas, in the second report, the title is Balance sheet, since the attribute **Keep TDL Loaded Flag** is set to **NO**.

3. Platform Functions and Variables

3.1 Function - \$\$PrinterInfo

Function **\$\$PrinterInfo** has been introduced to extract the settings information for any installed printer. This function is very useful to get the information of the printer, based on which, we can determine the dimensions for pre-printed invoice, etc.

Syntax

```
$$PrinterInfo : <Printer Name> : <Information Type>
```

Where,

<Printer Name> refers to the name of the printer for which the information is required.

<**Information Types**> are permissible information types like `PrintSizeInInches`, `LeftMarginInMMs`, etc.

Example:

```
$$PrinterInfo : HPLaserJet4250PCL6 : PrintSizeInInches
```

The list of permissible Information Types are:

- **LeftMarginInMMs** returns the Number which denotes the space to be left on the Left side of the page in Millimeters.
- **TopMarginInMMs** returns the Number which denotes the space to be left on the Top of the page in Millimeters.
- **RightMarginInMMs** returns the Number which denotes the space to be left on the Right side of the page in Millimeters.
- **PrinterExists** returns Logical value (YES/NO), indicating if the Printer Exists or not.
- **PrintSizeInInches** returns the dimensions which denotes the Print Area in Inches, i.e., excluding the Margins.
- **PrintSizeInMMs** returns the dimensions which denotes the Print Area in Millimeters, i.e., excluding the Margins.
- **PrintSizeInLines** returns the dimensions which denotes the Print Area in Lines, i.e., excluding the Margins.
- **PaperSizeInInches** returns the dimensions which denotes the Paper Size in Inches, which includes the Margins.
- **PaperSizeInMMs** returns the dimensions which denotes the Paper Size in Millimeters, which includes the Margins.
- **PaperSizeInLines** returns the dimensions which denotes the Paper Size in Lines, which includes the Margins.
- **PaperType** returns the selected Type of the Paper, e.g., A4, A5 Small, etc.
- **PortName** returns the Port Name configured for the Printer.
- **Orientation** returns the Orientation Type of Paper, i.e., Landscape or Portrait.

The following screen capture displays the selected Printer details for all the information types:

<u>Function PrinterInfo</u>	
Printer	: HP LaserJet 4250 PCL6
Left Margin (MMs)	: 4
Top Margin (MMs)	: 4
Right Margin (MMs)	: 0
Printer Exists	: Yes
Print Size (Inches)	: (8.11" x 7.95")
Print Size (MMs)	: (206 mm x 202 mm)
Print Size (Lines)	: (81 cols x 47 lines)
Paper Size (Inches)	: (8.27" x 11.69")
Paper Size (MMs)	: (210 mm x 297 mm)
Paper Size (Lines)	: (83 cols x 70 lines)
Paper Type	: A4 Small
Port Name	: 90
Orientation	: Landscape

Figure 2. Printer Details

3.2 Function - \$\$IsInternetActive

\$\$IsInternetActive is a function which helps to determine if the Internet is currently active. It returns TRUE if the Internet is accessible, else returns FALSE. It can be used to perform conditional operations, i.e., based on the Internet Connectivity, certain actions can be triggered.

This function checks if the internet is active, such that the operations pertaining to connecting to web pages, emailing, uploading files to FTP, etc., can be performed.

Syntax

\$\$IsInternetActive

Example:

```
[Function : EmailIfConnected]

00 : IF      : $$IsInternetActive

;; Function called to Email O/s Stmts

10 : Call : Email Outstanding Statements

20 : ENDIF
```

In this example, the Outstanding Statements are E-Mailed, if Internet connection is present.

3.3 Function - \$\$CaseConvert

Prior to this release, the function **\$\$Upper** has been used to convert the string expressions to upper case, but there were no functions available for other conversions like Lower case, Title Case, etc. To overcome the difficulty of converting the string to Lower case, Title case, etc., a new function **\$\$CaseConvert** has been introduced, to convert the case of the given expression to the specified case format. This function will return a string expression in the converted format.

This function is very useful when one needs to follow the case rules to display the Name of the company, Name of the bank, etc.

Syntax

```
$$CaseConvert : <CaseKeyword> : <Expression>
```

Where,

<CaseKeyword> can be All Capital, Upper Case, All Lower, Lower Case, Small Case, First Upper Case, Title Case, TitleCaseExact, Normal, Proper Case, etc.

- ❑ **All Capital/UpperCase** converts the input expression to upper case.
- ❑ **All Lower/LowerCase/SmallCase** converts the input expression to lower case.
- ❑ **First Upper Case** converts the first letter of the first word in a sentence to upper case. Other characters will remain as they are.
- ❑ **TitleCase** converts the input expression to Title case, i.e., the principal words should start with capital letters.
 - It will not convert the prepositions, articles or conjunctions, unless one is the first word.
 - It will ignore a subset of words from capitalization like the, an, and, at, by, to, in, of, for, on, cm, cms, mm, inch, inches, ft, x, dt, eis, dss, with, etc. For this subset of words, the original strings' cases will be preserved.
- ❑ **TitleCaseExact** converts the input expression to Title case, i.e., the principal words will start with capital letters.
 - It will not convert the prepositions, articles or conjunctions, unless one is the first word.
 - It will ignore a subset of words from capitalization like the, an, and, at, by, to, in, of, for, on, cm, cms, mm, inch, inches, ft, x, dt, eis, dss, with, etc. This subset of words will be converted to small case.
- ❑ **Proper Case** converts the input expression to Title case, i.e., all the words in a sentence should start with capital letters.
- ❑ **Normal** preserves the input expression as it is.

<Expression> is any expression of type 'String'.

Example: 1

To convert the expression to Upper case:

```
[Field : String Convert]
```

```
Set as : $$CaseConvert:UpperCase : "Tally solutions Pvt. Ltd."
```

In this example, the function returns "TALLY SOLUTIONS PVT. LTD." in the field 'String Convert'.

Example: 2

To convert the expression to Lower case:

```
[Field : String Convert]
```

```
Set as : $$CaseConvert : LowerCase : "Tally Solutions Pvt. Ltd."
```

Here, the function returns, "tally solutions pvt. ltd." in the field 'String Convert'.

Example: 3

To convert the expression to Title Case:

```
[Field: String Convert]
```

```
Set as : $$CaseConvert : TitleCase : +  
        "To convert the strINg to Title case"
```

Here, the function returns "To Convert the StrINg to Title Case" in the field 'String Convert'.

Example: 4

To convert the expression to Title Case Exact:

```
[Field : String Convert]
```

```
Set as : $$CaseConvert : TitleCaseExact : +  
        "To convert the string to Title case"
```

Here, the function returns "To Convert the String to Title Case" in the field 'String Convert'.

Example: 5

To convert the expression to First upper case:

```
[Field : String Convert]
```

```
Set as : $$CaseConvert:FirstUpperCase : "Tally solutions pvt. ltd."
```

Here, the function returns "Tally solutions pvt. ltd." in the field 'String Convert'.

3.4 Function - \$\$RandomNumber

A random number is a number generated by a process whose outcome is unpredictable, and which cannot be subsequently reliably reproduced. In other words, Random numbers are numbers that occur in a sequence such that, the values are uniformly distributed over a defined interval and it is impossible to predict future values based on past or present ones.

In this release, a new TDL function **\$\$RandomNumber** has been introduced to generate Random Numbers. In case of auditing, this can be useful for auditors who would like to pick up some vouchers randomly for authentication.

Syntax

```
$$RandomNumber [ :<MinRange> [ :<MaxRange> ] ]
```

Where,

<Min Range> and **<Max Range>** are optional. In the absence of Max Range, Long Max is considered, i.e., $(2^{31}) - 1 = 2147483647$. In the absence of Min Range, ZERO(0) is considered.

We can generate random numbers in different ways:

No Parameters: Don't pass any parameters, i.e., just invoke **\$\$RandomNumber**. Default values are assumed.

Only with the MinRange Parameter: Here, there is no need of passing Max range. In this scenario, Random number is generated from the given Min Range.

Both MaxRange and MinRange as Parameter: In this scenario, random numbers are generated for given range.

Example: 1

With no Parameters

```
Set As : $$RandomNumber
```

This code will return a Random Number between 0 and 2147483647.

Example: 2

With MinRange Parameter only

```
Set As : $$RandomNumber:9999
```

This code returns Random Numbers between 9999 and 2147483647, the random number being greater than or equal to 9999. Here, value of MinRange is 9999 and MaxRange is 2147483647.

Example: 3

With both Parameters (MinRange and MaxRange)

```
Set As : $$RandomNumber : 9 : 9999
```

This code returns Random Numbers between 9 and 9999.

3.5 Variable - SVPrintOrientation

Variable **SVPrintOrientation** has been introduced to set the required Printer Orientation, that is, Portrait or Landscape, within a Report. It is recommended to declare a local variable within the function or report and set the variable value, to avoid the system Printer Configuration changes to be effected globally.

This is useful where a Report needs to be printed in a different orientation, e.g., Landscape. For e.g., if one needs to print the cheques in 'Landscape' mode and other reports in 'Portrait' mode, then there is no need to keep switching the printer settings from Portrait to Landscape, and vice versa, based on the report getting printed. For Cheque Printing Report, one can default Landscape Orientation.

Example:

```
[#Report : Balance Sheet]
;; Local Variable Declaration
Variable : SVPrintOrientation : String
Set      : SVPrintOrientation : "Landscape"
```

Since the variable is locally declared and updated within the Report **Balance Sheet**, the same will not affect the global printer settings.

What's New in Release 3.0

In this release, the programmable configuration support has been extended to the actions 'Print Report', 'Export Report', 'Upload Report' and 'Email Report'.

A new attribute **Plain XML** has been introduced at Report level to export the report in plain XML format. New functions **\$\$StrByCharCode** and **\$\$InPreviewMode** have been introduced, whereas the functions **\$\$Inwords** and **\$\$ContextKeywords** have been enhanced. New events 'Start Import', 'Import Object' and 'End Import' have been introduced for 'Import File' definition.

1. Collection Enhancements

1.1 Collection Attribute 'WalkEx' Introduced

With every release, performance improvements are being brought, especially with respect to the data gathering and processing artefact 'Collection', used to gather and deliver data to presentation layers. Performance is enhanced drastically if collection is gathered judiciously as per usage.

Sometime back, the Collection attribute 'Keep Source' had been introduced for performance enhancement. This was used to retain the source collection gathered once with the specified Interface Object, i.e., either with the current Object or its parents/owners. It drastically improved the performance in scenarios where the same source collection was referred multiple times within the same Object hierarchy chain.

Similarly, there are scenarios where there is Union of multiple collections using the same source collection, and each collection walks over its sub objects across different paths, and computes/aggregates the values from sub level. In such cases, significant CPU cycles will be utilized in gathering and walking over the same Source Object along different paths more than once.

A new attribute WalkEx has been introduced, which when specified in the resultant collection, allows us to specify a collection list. The source collection specification is available in the resultant collection. The collections referred in WalkEx do not have any source collection specified and contain attributes only to walk the source collection and aggregate over Sub Objects of an already gathered collection. The advantage of using WalkEx is that all walk paths specified in the collection list are traversed in a single pass for each object in the source collection. This results in improvements in performance drastically.

□ Collection Attribute - WalkEx

Syntax

```
[Collection : <Collection Name>]
      WalkEx : <Collection Name1>, <Collection Name2>,...
```

Where,

<Collection Name1>, **<Collection Name2>**, and so on, are the collection names specifying Walk and aggregation/computation attributes.

Example:

The requirement here is to generate a Report displaying Item Wise and Ledger Wise amount totals for all Vouchers.

Using the Union Approach

;;The source collection "Voucher Source" is a Collection of Vouchers

```
[Collection : VoucherSource]
```

```
    Type : Voucher
```

;;The collection using "Voucher Source" as a source collection, and walking over Ledger Entries Sub-Object, aggregating Amount by grouping over Ledger Name

```
[Collection : Ledger Details]
```

```
    Source Collection : VoucherSource
```

```
    Walk              : AllLedgerEntries
```

```
    By: Particulars   : $LedgerName
```

```
    Aggr Compute      : Tot Amount : Sum: $Amount
```

```
    Keep Source       : ().
```

;;The collection using Voucher Source as a source collection, and walking over Inventory Entries Sub-Object, aggregating Amount by grouping over Stock Item Name

```
[Collection : StockItem Details]
```

```
    Source Collection : VoucherSource
```

```
    Walk              : AllInventoryEntries
```

```
    By                : Particulars: $StockItemName
```

```
    Aggr Compute      : Tot Amount : Sum: $Amount
```

```
    Keep Source       : ().
```

;;The Resultant Collection, which is a union of objects from the above two collections "Ledger Details" and "StockItem Details"

```
[Collection : Union LedStk Vouchers]
```

```
    Collection : Ledger Details, StockItem Details
```

In this example, both the collections 'Ledger Details' and 'StockItem Details' are using the same Collection 'Voucher Source'. We can observe that while gathering and summarizing values from the source collection, each object of the collection 'Voucher Source' is traversed twice for aggregating objects over two different paths, i.e., once for 'Ledger Entries' and then for 'Inventory Entries'. The report finally uses the Union collection 'Union LedStk Vouchers' to render the same.

Let us now move on to the new approach using “WalkEx” to achieve the same

Using the WalkEx Approach

;;The source collection “Voucher Source” is a Collection of Vouchers

```
[Collection : VoucherSource]
```

```
    Type : Voucher
```

/ The collection “UnionLedStkVouchers” is the resultant collection which will contain all the Objects obtained out of walks and multiple walks over the same Source Collection. The Report finally uses this Collection. The attribute WalkEx is specified which has values as collection names “Ledger Details” and “StockItem Details”*/*

```
[Collection : Union LedStk Vouchers]
```

```
    Source Collection : VoucherSource
```

```
    WalkEx           : Ledger Details, StockItem Details
```

```
    Keep Source     : () .
```

*/*The Collection “Ledger Details” walks over “AllLedgerEntries” Sub-Objects and aggregates the amount by grouping over Ledger Name. Note the absence of source collection. */*

```
[Collection : Ledger Details]
```

```
    Walk           : AllLedgerEntries
```

```
    By             : VchStockItem : $LedgerName
```

```
    Aggr Compute  : VchLedAmount : Sum: $Amount
```

*/*The Collection “StockItem Details” walks over “AllInventoryEntries” Sub-Objects and aggregates the amount by grouping over Stock Item Name. Note the absence of Source Collection in this. */*

```
[Collection : StockItem Details]
```

```
    Walk          : AllInventoryEntries
```

```
    By            : VchStockItem : $StockItemName
```

```
    Aggr Compute  : VchLedAmount : Sum : $Amount
```

The Collections used within ‘WalkEx’ use the same Source Collection. Each Object of “Voucher Source” is walked across “Ledger Entries” and “Inventory Entries” in a single pass. Thus, there is an exponential improvement in performance as it traverses each object only once, to gather the values for the resultant collection. In the case of Union Collection, for every Collection using different walk path, the Source Collection Object is being traversed again and again.

‘Walk Ex’ Attribute - Usage Implications

Let us consider the following code design, to understand the implication on various other collection attributes, in cases where ‘Walk Ex’ is used.

```
[Collection : Src Coll] ;; Source Collection
```

...

;;Resultant Collection "Res Coll" is using the Source Collection "Src Coll", and Walk Ex Collections "Walk Ex Coll 1" and "Walk Ex Coll 2" are specified

```
[Collection : Res Coll]
```

```
Source Collection : Src Coll
```

```
WalkEx           : Walk Ex Coll 1, Walk Ex Coll 2
```

```
[Collection : Walk Ex Coll 1] ;;Walk Ex Coll 1
```

```
Walk : Path1, SubPath1, SubSubPath1
```

```
By   : GroupName1 : $Method1
```

```
[Collection : Walk Ex Coll 2] ;;Walk Ex Coll 2
```

```
Walk : Path2, SubPath2, SubSubPath2
```

```
By   : GroupName2 : $Method2
```

The following table shows the attributes of 'Collection' definition and their applicability in the Resultant collection as well as WalkEx collections.

Attributes	Resultant Collection	Walk Ex-Collections
Source Collection	Specified and applicable	Ignored
Keep Source	Specified and applicable	No significance
Is Client Only	Specified and Considered	Ignored
Sorting	Specified and applicable	Ignored
Filtering	Specified and applicable	Ignored
Max	Specified and applicable	No significance
Parm Var	Specified and Considered	Ignored but to be inherited from the resultant collection
Source Var	Specified and Considered	Specified and applicable
Compute\Filter Var	Specified and applicable	Specified and applicable
Fetch	Specified and applicable	Specified and applicable
Compute	Specified and applicable	Specified and applicable
Aggr Compute	No significance	Specified and applicable
Walk\By	If Specified these two attributes WalkEx will be ignored	Specified and applicable

1.2 Directory as a Data Source

As we are already aware, a collection can be populated dynamically using the data available from a variety of external data sources. A common attribute 'Data Source' is used to specify the Type and identity of the source from where the data is to be retrieved. Thereafter, the data is available as objects and the associated information can be extracted from them using the corresponding methods. For example, if the data is populated from an XML file, the tag names are referred to as the method names. In case the data is populated from a compound variable, the corresponding member variable names are referred to as method names.

Prior to this release, the Data Sources supported were:

- ❑ XML available over HTTP/HTTPS using Post/Get methods
- ❑ XML File available within the local disk or over a network
- ❑ Output XML from an External DLL
- ❑ Specific Objects from Current/Parent Report
- ❑ Variable

Syntax

Data Source : <Type> : <Identity> [:<Encoding>]

Where,

<Type> specifies the type of data source: File Xml/HTTP Xml/Report/Parent Report/DLL/Variable

<Identity> can be file path, scope, etc., depending on the type specification.

<Encoding> can be ASCII or UNICODE. This is Optional. The default value is UNICODE.

From release 3.0 onwards, the collection attribute **Data Source** has been enhanced to support "**Directory**" as data source type. This will enable to gather all information pertaining to the contents of the disk directory/folder. Each folder constituent, i.e., either File or Directory, along with its corresponding details, are available as an object in the collection.

Let us consider the directory/folder "ABC", as shown in the following figure:

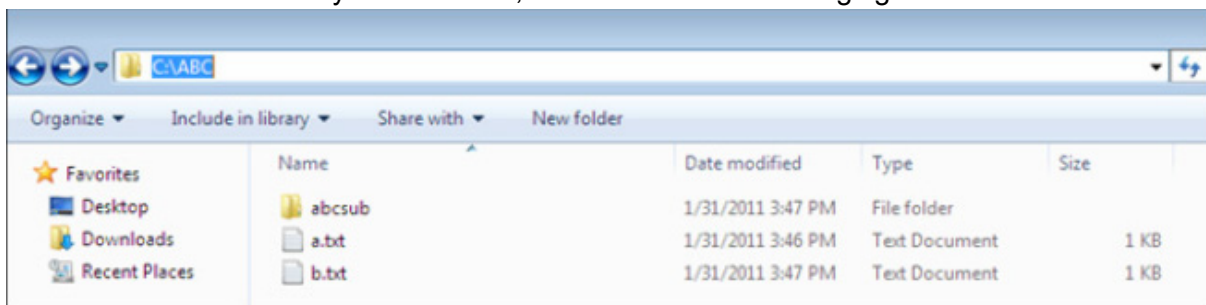


Figure 1. Folder path

The folder contains two files "a.txt" and "b.txt" and the folder "abcsub". In order to retrieve the item details along with the corresponding information like type, size and date modified within a collection, the attribute 'Data Source' can be specified with the **new enhanced syntax** as:

Syntax

[Collection : <Collection Name>]

Data Source : <Type> : <Identity> [:<Encoding>]

Where,

<Collection Name> is the name of the collection where the data is populated as objects.

<Type> specifies the type of data source. As per the new enhancement, it is "Directory".

<Identity> is the Directory/folder Path, when the type specified is 'Directory'

<Encoding> can be ignored for Type "Directory".

The following code will populate the collection "ABC Contents" with the folder contents from the path "C:\ABC". In this case, each of the items, i.e., a.txt, b.txt and "abcsub", will be available as separate objects of the collection. The related information pertaining to each object will be available as methods \$name, \$filesize, \$IsDirectory, \$IsReadOnly, \$IsHidden, etc.

```
[Collection : ABC Contents]
```

```
  Data Source : Directory : "C:\ABC"
```



\$filesize is applicable only if the object is of type FILE.

2. Image Printing Capabilities

Over the years, there has been a major requirement from the user community to enable Image Printing in Tally. Earlier, we used to achieve this capability by creating a new Font Type by associating an Image with a particular character and further using the 'Style' for the field. This had a few limitations in terms of image size, resolution and color. Also, it was not a clean way of incorporating the feature.

From this release onwards, Image Printing is being supported with the help of the following latest enhancements:

- 'Graph Type' attribute of Part allowing the specification of BMP, enabled for 'Print' Mode
- A new Definition Type '**Resource**' introduced in TDL

The configuration settings allow the user to specify the location of the Image file and the same is printed as a logo on the top left of the following default Reports.

- Sales Invoice - both Normal and Simple formats
- Delivery Note/Challan
- Debit Note
- Credit Note
- Outstanding Receivables
- Remainder Letters
- Pay Slips
- Purchase Order
- Receipt voucher

- Confirmation of Accounts

2.1 Part Attribute – Graph Type

Prior to Tally.ERP 9 release 3.0, specification of Image (as BMP only) within a part was supported. The attribute **Graph Type** of the part is used for the same.

Syntax

```
[Part : PartName]
```

...

```
Graph Type : Yes/Bitmap Image Path
```

Where,

<Graph Type> accepts 'Yes', or a path of the bmp file. If the value of <Graph Type> is "Yes", then it will be treated as a graph. If the value is not "Yes", then system will look for a Bitmap file with the given expression. However, the bitmap image was only supported in the 'Display' Mode.

From this Release onwards, this capability has been extended to 'Print' mode also. A few points are to be considered:

- The attribute 'Graph Type' supports Bitmap and JPG/JPEG. If the image type is specified as a JPEG/JPG it will consume the same. If it is of any other type, this will be considered as BMP and the same will be located from the path specified. If the file is unavailable or is not a valid image file, then the area allocated for image will be blank.
- The Part containing 'Graph Type' cannot display or print any contents and any contents specified within the fields will be ignored. That's why it requires the specification of dummy lines and fields within the part.
- Part Height and Width must be apportioned appropriately as per the Image size to be printed [If the height and/or width is not given, then the system will take the actual image size and use the same for display]. If the user has specified Height and/or width which is different from actual image size, then the system will do proportionate resizing of the image to fit into the given area [For example, let's say that area allocated for the image is 300 X 300 and actual image size is 150 X 75. Then, the system will display image in 300 X 150].

2.2 New Definition Type – Resource

A new definition type "Resource" has been introduced in TDL. This will allow accessing and using the resources (images, icons, cursors, etc.) from a local disk, HTTP/FTP or from a DLL/EXE. The image formats supported at present are BMP/JPEG/ICON/CUR. The resource thus created can be used in a 'Part' definition using the attribute "Image". This is applicable both in 'Print' and 'Display' mode.



However, when the same report is exported in PDF, only BMP and JPEG are supported. Other file formats will be ignored.

Syntax

```
[Resource : Name]
    Source      : <Path to Image file>
    Resource    : <NameOfResource> [:<DLL/EXE Name with path>]
    Resource Type : Bitmap/Icon/Jpeg/Cursor
```

Attributes - SOURCE, RESOURCE and RESOURCE TYPE

□ Attribute – SOURCE

This is a 'Single' type attribute, and hence accepts only one parameter. It allows us to specify the image file path. This file can be a local disk file, or a file available over an HTTP/FTP path.

Syntax

```
Source : <Path to Image file>
```

Where,

<Path to Image file> can be any string expression which evaluates to the file path, along with Filename and extension.

Example:1

```
[Resource : CmpImage]
    Source : "C:\Tally.bmp" ;; where the image "Tally.bmp" is available in local disk
```

Example:2

```
[Resource : CmpImage]
    Source : "Http://www.tallysolutions.com/images/tallyHTTP.JPGEG"
;; where the image tallyHTTP.JPGEG is available over an HTTP Path
```

□ Attribute – RESOURCE

This is a 'Dual' type attribute and accepts two parameters. The first parameter refers to the resource name present in an Exe/DLL. The second parameter is used to specify the path and the name of Exe/DLL. However, this is optional. In case not specified, the system will look for the resource within tally.exe itself.

Syntax

```
Resource : <NameOfResource> [:<DLL/EXE Name with path>]
```

Where,

<Name of Resource> is a string expression which evaluates to the name of the resource present in the specified DLL/EXE (When resources are added to DLL/EXE, the user can give a separate name for the resource).

<DLL/EXE Name with path> can be a string which evaluates to the complete DLL/EXE path.

Example:1

```
[Resource : CmpImage]
    Resource : "TITLEICON"
```


;; This uses the resource "TITLEICON" present in Tally exe, as we have not specified the EXE path.

Example:2

```
[Resource : CmpImage]
```

```
Resource : "60040" : C:\ProgramFiles\WindowsNT + \Accessories\wordpad.exe
```

```
Resource Type : BMP
```

;; This uses the resource "60040" present in the wordpad.exe, as we have specified the EXE path as the second parameter.



The attribute "Source" and "Resource" are mutually exclusive, i.e., either of them can be used. We cannot use both together. If both are specified in TDL, then the system will use SOURCE and ignore the RESOURCE attribute.

□ Attribute – RESOURCE TYPE

This is a 'Single' type attribute, and hence accepts only one value as a parameter. It allows the specification of Type of the resource. 'Type' can be any one of the standard windows image resources like Bitmap, Icon, Cursor or JPEG. The type specified in 'Resource Type' will be used for loading the image appropriately.

'Resource Type' is a mandatory attribute and must be specified for all sources. If not specified, the type would be taken as 'Bitmap' by default.



For the Icon resources - the nearest sized Icon will be taken. For example, if we have two Icons 16X16 and 32X32 and the part size is 20X20, then the 16X16 icon will be used for displaying.

Syntax

```
Resource Type: BMP/Icon/Jpeg/Cursor
```

Example:

```
[Resource : CmpImage]
```

```
Source : "C:\Tally.bmp" Resource Type : BMP
```

Part Attribute - Image

□ Part Attribute – IMAGE

With the introduction of the new attribute "Image", it is possible for the resource created by using the Definition "Resource" to be used in the Part.

Syntax

```
[Part : PartName]
```

```
...
```

Image : <Resource Name>

Where,

<Resource Name> is the name of the Resource definition.

Example:

[Part : Part ABC]

Image : CmpImage

3. Enhanced Columnar Capability

3.1 Columnar Reports in General

A matrix report looks like a grid. It contains a row of labels, a column of labels, and information in grid format that is related to both the row and column labels. In Tally, two dimensional matrix reports can be designed using the Auto column report approach (using Repeat Variables). Traditionally, these types of Reports are referred to as columnar Reports. In particular, Matrix report is a variant of automatic auto column reports, where the columns are repeated over a variable associated at the Report. The collection repeated with this variable is used to populate the repeated values into the variable. The method value in the detail line is extracted from a different collection, based on the corresponding row and column indexes.

Following is a typical two-dimensional matrix report showing the total number of stock items sold for each party.

	Party 1	Party 2	Party 3	Party 4	...
Stock Item 1	100 Nos	160 Nos	220 Nos	180 Nos	...
Stock Item 2	120 Box	210 Box	250 Box	120 Box	...
Stock Item 3	250 Kg	170 Kg	180 Kg	240 Kg	...
:	:	:	:	:	:

Figure 2. Two-dimensional matrix report

3.2 Enhanced Capabilities for Columnar Reporting

The latest enhancements in the area of Columnar Reporting enables us to design the reports using a new approach altogether. A field within a line can display method values from multiple objects of the collection. Context Free repeat within the part and line enable repetition on simple as well as list variable values. These features give a better control in the hands of the programmer

These features give a better control in the hands of the programmer in designing such reports.

If we consider the above report layout, the labels in columns can now repeat over a collection of Parties. The data in the cells can be populated based on the combination of row and column label values across the dimensions. In the previous table, for example, the highlighted cell contains the value of total sales quantity corresponding to the party “Party 2” for the Stock Item “Stock Item 2”.

The following enhancements have been enabled to achieve this functionality:

- ‘Repeat’ Attribute for Part and Line over a Collection

- Context Free Repeat for Part and Line, together with SET/Break On
- Usage of the function \$\$LineObject

Attribute 'Repeat' Enhancements – Part and Line

The 'Repeat' Attribute has been enhanced consistently across 'Part' and 'Line' Definitions to support "Context Based" as well as "Context Free" Repeat.

□ Attribute REPEAT – Part Definition

The common syntax allows the repetition of a contained line, with or without a collection.

Syntax

```
[Part : <Part Name>]
  Repeat : <Line Name> [: <Collection>]
  Set    : <Count>
```

Where,

<Part Name> is the name of the part.

<Line Name> is the name of the line to be repeated.

<Collection> is the name of collection on which the line is repeated. It is an optional parameter.

<Count> denotes the number of times the line is to be repeated, if Collection Name is not specified.

Context based Repeat – The 'Repeat' attribute of the Part can repeat the contained line over a collection. Each line in this case is associated with each object of the collection. This was the earlier capability even before Tally.ERP 9

Context Free Repeat – From Release 1.8, the Collection parameter in the above syntax has been made optional. This allows the repetition of a contained line without a collection. Since the no. of times the line has to be repeated is not known, the usage of the attribute SET to specify the count becomes mandatory. In case of 'Edit' mode, the attribute 'Break On' can be used to specify the terminating condition for repetition.

□ Attribute REPEAT – Line Definition

So far, the 'Repeat' attribute at 'Line' definition has been accepting only a field name which internally uses the repeat behaviour of the Report and Variable for determining the no. of times it can be repeated. This attribute has now been enhanced to support the consistent syntax to enable "Context Based" and "Context Free" repetition of the same field horizontally.

Syntax

```
[Line : <Line Name>]
  Repeat : <Field Name> [: <Collection Name>]
  Set    : <Count>
```

Where,

<Line Name> is the name of the Line.

<Field Name> is the name of the Field to be repeated.

<Collection Name> is the name of the collection on which the Field is repeated. It is optional.

<Count> denotes the no. of times the Field has to be repeated, if collection name is not specified.

Context based Repeat – The 'Repeat' attribute of the line can repeat the contained field over a collection. Each field in this case is associated with each object of the collection.

Context Free Repeat – The collection parameter in the above syntax is optional. This allows the repetition of a contained field without a collection. Since the no. of times the field is to be repeated is not known, the usage of the attribute SET to specify the count becomes mandatory. In case SET is not specified, the Field will be repeated as per the existing Columnar behaviour.

Example 1: Item-Wise Party-Wise sales quantity report using Context-Based Repeat of Field

Following screen shows the Item-wise-Party-wise Report using enhanced columnar capability:

Particulars	Global Traders	Amar Computer Peripherals	Silverplus Computers	Supreme Computers Peripherals	Horizon Systems	Office Automation Systems	Computer World	Nirmaan Timbers	Worldwide Computers	AVT Computers
IBM PIV	--- ... 6 more				1 Nos				17 more ... ---	
Wireless Keyboard					40 Nos					
CDROM Disks 100s	35 Box									
CDROM Disks 10s	100 Nos									
Dust Covers										
Timber								285 MT		
HCL PIV										
Assembled PIV			125 Nos				75 Nos		105 Nos	
Keyboard										
USB Pen Drives 64 MB	113 Nos				5 Nos					
HP Laserjet 1010 Series		10 Nos								
TVS MSP 245 132 Col Printer				15 Nos						14 Nos
Wireless Mouse					52 Nos					
Mouse Pad	170 Nos				200 Nos					
Mother Board										
TVS MSP 245 80 Col Printer						115 Nos				

Figure 3. Item-wise-Party-wise Report

Following is the code snippet to design the above report using enhanced columnar capability:

1. Collection definitions for Stock Item, for Party, for getting the values, etc., are as follows:

```
[Collection : Smp CFBK Voucher]
```

```
Type : Voucher
```

```
Filter : Smp IsSalesVT
```

```
[Collection : Smp Stock Item]
```

```
Source Collection : Smp CFBK Voucher
```

```
Walk : Inventory Entries
```

```
By : IName : $StockItemName
```

```
Aggr Compute : BilledQty : SUM : $BilledQty
```

```

Keep Source      : ( ) .
Filter           : SmpNonEmptyQty
[Collection      : Smp CFBK Party]
Source Collection : Smp CFBK Voucher
Walk             : Inventory Entries
By              : PName : $PartyLedgerName
Aggr Compute    : BilledQty : SUM : $BilledQty
Keep Source      : ( ) .
Filter           : Smp NonEmptyQty
[Collection n    : Smp CFBK Summ Voucher]
Source Collection : Smp CFBK Voucher
Walk             : Inventory Entries
By              : PName      : $PartyLedgerName
By              : IName      : $StockItemName
Aggr Compute    : BilledQty : SUM : $BilledQty
Keep Source      : ( ) .
Search Key      : $PName + $IName

```

;; System Formula

```

[System : Formula]
Smp IsSalesVT    : $$IsSales      : $VoucherTypeName
Smp NonEmptyQty : NOT $$IsEmpty : $BilledQty

```

From these Collections, following can be observed:

- The Rows, i.e., Stock Items, are repeated over the Collection 'Smp Stock Item'.
- The Columns, i.e., Party Names, are repeated over the Collection 'Smp CFBK Party'.
- The Intersection values between these Rows and Columns, i.e., Item wise Party wise Sales Quantity, are set using the Collection 'Smp CFBK Summ Voucher'. This Collection is indexed on Methods \$PName + \$IName using the Collection Attribute 'Search Key'. Thus, the Collection is indexed on Party Name and Stock Item Name, which makes it unique across all the Objects within the Collection 'Smp CFBK Summ Voucher'.
-
-

2. The Lines for Title and Details are repeated for the Party Names as shown below:

[Line : Smp CFBK Rep Title]

```
Use      : Smp CFBK Rep Details
Local   : Field : Default : Type      : String
Local   : Field : Default : Align   : Center
Local   : Field : Smp CFBK Rep Name   : Set as      : "Particulars"
Local   : Field : Smp CFBK Rep Name   : Widespaced : Yes
Local   : Field : Smp CFBK Rep Party  : Set as      : $PName
Local   : Field : Smp CFBK Rep Party  : Lines      : 0
Local   : Field : Smp CFBK Rep ColTotal : Set as      : "Total"
```

[Line : Smp CFBK Rep Details]

```
Fields  : Smp CFBK Rep Name, Smp CFBK Rep Party, Smp CFBK Rep Col Total
Repeat  : Smp CFBK Rep Party : Smp CFBK Party
```

Title Line uses the detail line where the Field "Smp CFBK Rep Party" is repeated over the Collection "Smp CFBK Party". In the Title Line, the Field "Smp CFBK Rep Party" is set with the value "\$PName", which sets the Party Names from the Collection "Smp CFBK Party".

3. Retrieving the values in cells based on Party name available from context and stock item name available in the field as shown below:

[Field : Smp CFBK Rep Name]

```
Use      : Name Field
Set as   : $IName
Display  : Stock Vouchers
Variable : Stock Item Name
```

[Field : Smp CFBK Rep Party]

```
Use      : Qty Primary Field
Set as   : $$ReportObject:$$CollectionFieldByKey:$BilledQty:+
          @SKFormula:SmpCFBKSummVoucher
SKFormula : $PName + #SmpCFBKRepName
Format   : "NoZero"
Border   : Thin Left
```

In this code snippet, we can observe that the Field “Smp CFBK Rep Party” is the intersection between rows & columns. The value is gathered from the Collection “Smp CFBK Summ Voucher” using the function **\$\$CollectionFieldByKey**, where the Index Key in the current context is passed as a parameter. “\$PName” in the current object context returns the Party Name. Similarly, the Field Value “#SmpCFBKRepName” in the current context returns the Stock Item Name. Hence, the Search Key Index “Party Name + Stock Item Name” for every Intersection point is passed to this function, which extracts and returns the corresponding Quantity from the Collection.

4. Calculating Field Level Totals, i.e., Stock Item Totals, across all Parties is done using the Line Attribute ‘Total’ and the Function **\$\$Total**, as shown below:

```
[Line : Smp CFBK Rep Details]

    Total   : Smp CFBK Rep Party

[Field : Smp CFBK Rep Col Total]

    Use     : Qty Primary Field

    Set as  : $$Total:SmpCFBKRepParty
```

Line “Smp CFBK Rep Details” contains an Attribute ‘Total’, which accepts Field Names as its value. In other words, we declare at the Line that the Fields are to be summed for later use. This sum gets accumulated and rendered in the Field “Smp CFBK Rep Col Total”, where the function **\$\$Total** returns the accumulated Total for the given Field Name as the Parameter to this Function.

New Built-in Function **\$\$LineObject**

Since the Line Attribute ‘Field’ can now be repeated over a Collection, wherein the Object context inherited from the Line is overridden in the Field; to switch back to the parent context, i.e., Line’s object context and extract the required method value, a New Function **\$\$LineObject** has been introduced.

□ Function - **\$\$LineObject**

Syntax

```
$$LineObject : <String Formula>
```

Where,

<String Formula> can be any expression that gets evaluated in the Object context associated at the current field’s parent ‘Line’ in the Interface Object hierarchy.

Interactive Reporting capabilities using Aggregated or External objects

The Actions “Remove Line”, “Show Last Removed Line” and “Show Removed Lines” work on the concept of Object Identifier. Whenever the collection of internal objects is rendered as a report, the default buttons “Remove Line” and “Restore Line” using the above actions work on them as they are uniquely identifiable.

In cases where the Collection used contains aggregated Objects, or objects from an external data source like XML, etc., the objects available do not contain a unique identifier. When such collections are rendered, the Actions mentioned above do not work.

In order to overcome the problem, the behaviour of the attribute 'Search Key' has been enhanced to assign a unique key for such Object Types. It takes a single method or a combination of methods, which will serve as a unique identifier to each object of the aggregated or external collection. It has to be ensured that each object in the collection must contain unique values for the method which is assigned as the key.

Attribute 'Search Key' enhanced

Syntax

```
[Collection : <Collection Name>] Search Key : <Expression>
```

Where,

<Expression> evaluates to a unique identifier for each object of the collection. It is usually a combination of method names separated by '+', which must make a unique combination for each object of the Collection.

Example:1

Please observe the previous sample Item-Wise Party-Wise report, wherein **Alt + R** Key combination does not work for Removal of Line, as there is no unique identifier for the Line Object. Each line in the example is repeating over the objects of the collection "Smp Stock Item". To specify the unique Object identifier, this Collection is altered by specifying the 'Search Key' attribute, with a unique combination of Methods as value. In this case, it is the method name \$Iname, i.e., the Stock Item Name, based on which the objects are grouped.

```
[#Collection : Smp Stock Item]
    Search Key : $IName
```

Example: 2

Following is another example using external data objects as available in the following XML file, containing the data for Students and corresponding marks in various subjects.

```
<StudData>
    <Student>
        <Name>Rakesh</Name>
        <Subject>
            <Name>History</Name>
            <Mark>90</Mark>
        </Subject>
        <Subject>
            <Name>Civics</Name>
            <Mark>90</Mark>
        </Subject>
    </Student>
</StudData>
```



```
</Subject>
<Subject>
  <Name>Kannada</Name>
  <Mark>90</Mark>
</Subject>
</Student>
<Student>
  <Name>Uma</Name>
  <Subject>
    <Name>History</Name>
    <Mark>80</Mark>
  </Subject>
  <Subject>
    <Name>Civics</Name>
    <Mark>50</Mark>
  </Subject>
  <Subject>
    <Name>Kannada</Name>
    <Mark>65</Mark>
  </Subject>
</Student>
<Student>
  <Name>Prashanth</Name>
  <Subject>
    <Name>History</Name>
    <Mark>50</Mark>
  </Subject>
  <Subject>
```

```

        <Name>Civics</Name>

        <Mark>90</Mark>

    </Subject>

    <Subject>

        <Name>Kannada</Name>

        <Mark>90</Mark>

    </Subject>

</Student>

</StudData>

```

The data populated from the above XML is displayed as a columnar report as follows:

Student Name	History	Civics	Kannada
Rakesh	90	90	90
Uma	80	50	65
Prashanth	50	90	90

Figure 4. Student-wise-Subject- wise Marks Report

Student-wise Subject-wise Marks information is listed in tabular form, as shown in the figure. Now, on removing the selected line(s), the required lines must be removed. Since this report is constructed out of an external source, i.e., XML Data, the same requires a unique identifier for each object in the repeated line. In this case, it is the Student Name; hence, the Search Key should contain this as an identifier.

Following is the sample code required to display the above report in a columnar fashion, with the Remove/Restore Line behaviour incorporated:

```

[Report : Ext XML Data Stud]

    Form : Ext XML Data Stud

[Form : Ext XML Data Stud]

    Parts : Ext XML Data Stud

    Bottom ToolBar Buttons : BottomToolBarBtn8, BottomToolBarBtn9,+

                            BottomToolBarBtn10

[Part : Ext XML Data Stud]

    Lines      : Ext XML Data Stud Heading, Ext XML Data Stud Info

    Repeat     : Ext XML Data Stud Info : Ext XML Data Students

```

Scroll : Vertical

CommonBorder : Yes

[Line : Ext XML Data Stud Heading]

Fields : Ext XML Data Stud Name, Ext XML Data Stud Mark

Repeat : Ext XML Data Stud Mark : Ext XML Data Stud Subj Summary

Local : Field : Default : Type : String

Local : Field : Default : Style : Normal Bold

Local : Field : Default : Align : Centre

Local : Field : Ext XML Data Stud Name : Set As : "Student Name"

Local : Field: Ext XML Data Stud Mark: Set As: \$SubjectName

Local : Collection: Ext XML Data Stud SubjSummary: Delete: Filter

Local : Collection: Ext XML Data Stud SubjSummary: +

Delete : By: StudentName

Border : Thin Top Bottom

[Line: Ext XML Data Stud Info]

Fields : Ext XML Data Stud Name, Ext XML Data Stud Mark

Repeat : Ext XML Data Stud Mark: Ext XML Data Stud Subj Summary

[Field : Ext XML Data Stud Name]

Use : Name Field

Set As : \$Name

[Field : Ext XML Data Stud Mark]

Use : Number Field

Set As : \$\$Number:\$SubjectTotal

Align : Right

Border : Thin Left

[Collection : Ext XML Data Students]

Data Source : File XML : "D:\StudData.xml" : Unicode

XML Object Path : Student : 1 : StudData

```

Search Key          : $Name

[Collection : Ext XML Data Stud Subj Summary]

Source Collection   : Ext XML Data Students

Walk               : Subject

By                 : StudentName : $..Name

By                 : SubjectName : $Name

Aggr Compute       : SubjectTotal : SUM : ($$Number:$Mark)

Keep Source        : ().

Filter             : ForThisStudent

[System : Formula]

ForThisStudent     : $StudentName = $$ReqObject:$Name

```

In this code, Line 'Ext XML Data Stud Info' is repeated over the Collection 'Ext XML Data Students', where Search Key is specified to be \$Name. Hence, the Remove/Restore Line behaviour will work.

4. Persisting Variables at System Scope in a User Specified File

As announced in Release 2.0, we are aware that the variables at the Report scope can be persisted in a user specified file using the action SAVE VARIABLE. This can be re-loaded as required using the action LOAD VARIABLE.

The latest enhancements in variable persistence allow the user to persist and re-load the variables at System Scope (in a User Specified File) as well.

4.1 Action – SAVE VARIABLE

The action SAVE VARIABLE, which is used to persist the Report Scope Variables in a user specified file, now allows us to persist the System Scope Variables also. Syntax of this action remains the same. The desired behaviour is achieved with changes in variable list specification.

Syntax

```
SAVE VARIABLE : <FileName> [:<Variable List>]
```

Where,

<File Name> is the name of the file in which the report scope/ system scope variables are persisted. The extension .PVF will be taken by default, if the file extension is not specified.

<Variable List> is the comma-separated list of variables that need to be saved in the file.

Variable List specification changes

- Now '*' can also be used to specify the variable list, which means all at 'current scope'.
 - The current scope can either be 'System' or 'Report'.

- Specifying '*' will ignore the 'Persist' flag and save all the variables in the scope, irrespective of "Persist: Yes" at the 'Variable' definition level.
2. If Variable list is not provided, it will persist all the variables which are set as "Persist: Yes" at the Variable definition level.
 3. Dotted notation syntax is also supported in the variable list specification for scope specification. However, this cannot be used for SUB levels. It can be used only for accessing parent scope variables.
 - Single Dot "." refers to current scope, Double Dot ".." to parent scope, Triple Dot "... " to grandparent scope, and so on.
 - "(" refers to the System Scope.

4.2 Action – LOAD VARIABLE

The action LOAD VARIABLE, which is used to load the Report Scope Variables in a user specified file, now allows us to load the System Scope Variables also. Syntax of this action remains the same. The desired behaviour is achieved with changes in variable list specification.

Syntax

```
LOAD VARIABLE : <FileName> [:<Variable List>]
```

Where,

<File Name> is the name of the file in which the report scope/ system scope variables are persisted. Specifying file extension is mandatory while loading variable values.

<Variable List> is the comma-separated list of variables that need to be loaded from the file.

Variable List specification changes:

1. While loading, '*' is not relevant and will be ignored.
2. While loading, 'Persist' flag of the variable is ignored. It is assumed that the variable must have a persist flag OR it is saved forcefully and hence to be loaded.

Example:1

There is a requirement to persist values of all system scope variables in a user specified file and load the values from the file whenever required. Refer to the following code snippet:

```
[#Menu : Gateway of Tally]
```

```
    Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

*;;Buttons **SLSystemScopeSave & SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions **SAVE VARIABLE & LOAD VARIABLE**.*

```
[Button : SLSystemScopeSave]
```

```
    Key      : Alt+F
```

```
    Action   : SAVE VARIABLE : SLSystemScope.pvf : *
```

```
    Title    : "Save Sys Var"
```

*Values of all system scope variables will be persisted in the file **SLSystemScope.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button : SLSystemScopeLoad]
```

```
Key      : Alt + L  
Action   : LOAD VARIABLE : SLSystemScope.pvf  
Title    : "Load Sys Var"
```

*Values of all system scope variables will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.*

Example:2

There is a requirement to persist values of all system scope variables which are set as “**Persist : Yes**” at variable definition level in a user specified file, and load the values from the file whenever required. Refer to the following code snippet:

```
[#Menu : Gateway of Tally]  
  
Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

Buttons **SLSystemScopeSave** & **SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions **SAVE VARIABLE** & **LOAD VARIABLE**.

```
[Button : SLSystemScopeSave]  
  
Key      : Alt+F  
Action   : SAVE VARIABLE : SLSystemScope.pvf  
Title    : "Save Sys Var"
```

*Values of all variables at system scope which are set “**Persist : Yes**” at variable definition level, will be persisted in the file **SLSystem- Scope.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button : SLSystemScopeLoad]  
  
Key      : Alt + L  
Action   : LOAD VARIABLE : SLSystemScope.pvf  
Title    : "Load Sys Var"
```

*Values of all variables will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.*

Example: 3

There is a requirement to persist the system scope variables **SVSymbolInSign** & **SVInMillions** in a user specified file, and load the values of these variables from the file whenever required. Refer to the following code snippet:

```
[#Menu : Gateway of Tally]  
  
Add : Button : SLSystemScopeSave, SLSystemScopeLoad
```

*Buttons **SLSystemScopeSave** & **SLSystemScopeLoad** are added at the Gateway of Tally Menu to execute the actions **SAVE VARIABLE** & **LOAD VARIABLE**.*

```
[Button : SLSystemScopeSave]
```

```
Key      : Alt+F

Action   : SAVE VARIABLE : SLSystemScope.pvf : + SVSymbolInSign,SVInMillions

Title    : "Save Sys Var"
```

*Values of the system scope variables **SVSymbolInSign** & **SVInMillions** will be persisted in the file **SLSystemScope.pvf** on execution of the action **SAVE VARIABLE**.*

```
[Button : SLSystemScopeLoad]
```

```
Key      : Alt + L

Action   : LOAD VARIABLE : SLSystemScope.pvf : + SVSymbolInSign,SVInMillions

Title    : "Load Sys Var"
```

Values of the system scope variables **SVSymbolInSign** & **SVInMillions** will be loaded from the file **SLSystemScope.pvf** on execution of the action **LOAD VARIABLE**.

Example: 4

The following report is displayed in 'Create' mode from a menu item.

```
[Report : Smp SLReport]

Form      : Smp SLForm

Variable  : SaveLoadVar1, SaveLoadVar2
```

The variables **SaveLoadVar1** & **SaveLoadVar2** are declared at Report Scope.

```
[Form : Smp SLForm]

Parts    : Form SubTitle, Smp SL Part

Button   : Smp SaveVar, Smp LoadVar
```

Buttons **SmpSaveVar** & **SmpLoadVar** are added at 'Form' Level to execute the actions **SAVE VARIABLE** & **LOAD VARIABLE**.

Let us look into the following scenarios to persist and load System Scope as well as Report Scope Variable values:

1. Persist & Load all Report Scope Variables & a specific System Scope Variable

```
[Button : Smp SaveVar]

Key      : Alt + S

Action   : SAVE VARIABLE : SLReportCfg.pvf: *, ().SVInMillions

Title    : "Save Variable"
```

*Values of all variables declared at report scope and the value of system scope variable **SVInMillions** will be persisted in the file **SLReportCfg.pvf** on execution of the action **SAVE VARIABLE**. (The variable **SVInMillions** is prefixed with (). to denote the same as System Scope Variable).*

```
[Button : Smp LoadVar]
```

Key : Alt + L

Action : LOAD VARIABLE : SLReportCfg.pvf : *, ().SVInMillions

Title : "Load Variable"

Variable list specification* will be ignored. Values of all report scope variables and the value of system scope variable **SVInMillions** will be loaded from the file **SLReportCfg.pvf** on execution of the action **LOAD VARIABLE**.

2. Persist and Load a specific Report Scope variable & a specific System Scope variable

[Button : Smp SaveVar]

Key : Alt + S

Action : SAVE VARIABLE: SLReportCfg.pvf : SaveLoadVar1,+ ().SVInMillions

Title : "Save Variable"

*Value of Report scope variable **SaveLoadVar1** and value of system scope variable **SVInMillions** will be persisted in the file **SLRe- portCfg.pvf** on execution of the action **SAVE VARIABLE**.*

[Button : Smp LoadVar]

Key : Alt + L

Action : LOAD VARIABLE : SLReportCfg.pvf : SaveLoadVar1,+ ().SVInMillions

Title : "Load Variable"

*Value of Report scope variable **SaveLoadVar1** and value of system scope variable **SVInMillions** will be loaded from the file **SLRe- portCfg.pvf** on execution of the action **LOAD VARIABLE**.*

5. New Events Introduced

As a part of the Language enhancements, in recent past, there have been significant enhancements as a part of the **Event Framework**. Before this release, events introduced were mostly related to handling the application start up and close, and company loading and unloading. The Object specific events were mainly focused around trapping events while rendering the data on the screen, and printing.

In this Release, events have been introduced to handle user specific requirements on data manipulation during Export and Import of data. With the introduction of the Events **Start Import**, **Import Object** and **End Import**, the programmers have got complete control to manipulate the data prior to importing the same into the company. This can be useful in scenarios like Inter-Branch data transfers; where Delivery Note in a branch gets transformed into Receipt Note in the other, Sales transaction in a Branch gets transformed into Purchase transaction in the other, and so on. Also, an action **Import Object** has been introduced to begin the Import process.

While exporting Full objects to XML and SDF formats; with the introduction of Export Events **Before Export**, **Export Object** and **After Export**, the user will be able to trap these events and get an access to the object being exported, which can be altered as required before export. This

can be useful in scenarios like changing required information during export, not displaying price/ amount of the stock item while synchronizing Delivery Note to the branch offices, creating a consolidated sales entry from all the sales transactions of the day, etc.

5.1 Import Events

The following events can be used within **Import File** Definition:

Event - Start Import

Syntax

On : Start Import : <Logical Condition> : <Action> : <Action Parameters>

If the logical condition specified returns TRUE, the Event **Start Import** executes the actions before beginning the import process. At this stage, the data objects will not be available, since it is prior to gathering the data from the file. This event can be used to communicate any messages to the user like starting the import process, etc.

Event - Import Object

Syntax

On : Import Object : <Logical Condition> : <Action> : <Action Parameters>

If the logical condition returns TRUE, event **Import Object** executes the actions after gathering the Objects from File, before importing the same in the current company. At this stage, data objects are available, since it is post gathering the data from file. This event is useful to manipulate & transform data from one form to another, i.e., from Receipt Note to Delivery Note, etc.

Syntax

On: Import Object: <Logical Condition>: Import Object

If the Event 'On Import Object' is used, it overrides the default Import Object behaviour, as a result of which, we need to explicitly specify to begin importing the objects. After performing the necessary actions prior to importing the objects, the Action Import Object must be specified to instruct the system to continue the import process.

Event - End Import

Syntax

On : End Import : <Logical Condition> : <Action> : <Action Parameters>

If the logical condition specified returns TRUE, the Event **End Import** executes the actions after importing the objects. At this stage, the data objects will not be available, since it is post importing the objects within the current company. This event can be used to communicate any messages to the user like 'ending the import process', 'Import Successful', etc.

Example:

```
[#Import File : Vouchers]
```

```
On : Start Import : Yes : Call : Start Import
```

```
On : Import Object : Yes : Call : Change Values
```

```
On : Import Object : Yes : Import Object
```

```
On : End Import : Yes : Call : End Import
```

```
[Function : Start Import]

    00 : MSGBOX : "Status" : "Starting Import Process"

[Function : Change Values]

    00 : SET VALUE : Narration : $Narration + " - Updated by +
        Import Object Event"

    10 : SET TARGET : LedgerEntries[1]

    20 : SET VALUE : LedgerName : "Branch Ledger"

[Function : End Import]

    00 : MSGBOX : "Status" : "Imported data successfully"
```

In this example, before importing the data, **Narration** Method is being altered and the first **Ledger Name** is being altered to **Branch Ledger**. Before starting and after ending the import process, appropriate messages are being displayed to the user.

5.2 Export Events

The following events can be used within **Form** Definition:

Event - Before Import

Syntax

```
On : Before Export : <Logical Condition> : <Action> : <Action Parameters>
```

If the logical condition specified returns TRUE, the Event **Before Export** executes the action before beginning the export. This event can be used to communicate any message to the user.

Event - Export Object

Syntax

```
On : Export Object : <Logical Condition> : <Action> : <Action Parameters>
```

If the logical condition specified returns TRUE, the Event **Export Object** executes the action before the object is exported. The user will get the object being exported, which can be altered as required before export. The form level Export Object is used to get an access to the object associated at the Report Level and manipulate the same before exporting.

Syntax

```
On : After Export : <Logical Condition> : <Action> : <Action Parameters>
```

If the logical condition specified returns TRUE, the Event **After Export** executes the action at the end of Form Export. This event can be used to communicate any message to the user.

The following events can be used within **Line** Definition:

Event - Export Object in Line Definition

Syntax

```
On : Export Object : <Logical Condition> : <Action> : <Action Parameters>
```

If the logical condition returns TRUE, the Event **Export Object** executes the action before every object is exported. The line level 'Export Object' is used to get an access to each object associated at the line level and manipulate the same before exporting.

Example:

```
[Form : ExpEvtForm]

    On          : Before Export : Yes : Call : Export Start

    On          : After Export  : Yes : Call : Export End

    Part       : ExpEvtPart

    Button     : Export Button

    Full Object : Yes

[Part : ExpEvtPart]

    Line      : ExpEvtLine

    Repeat    : ExpEvtLine : ExpLedger

    Scroll    : Vertical

[Line : ExpEvtLine]

    On          : Export Object: Yes: Call: ExportObject:$$Line

    Fields     : ExpEvtFld1, ExpEvtFld2

    Full Object : Yes

[Collection : ExpLedger]

    Type      : Ledger

    Fetch     : Name, Parent

[Function : ExportStart]

    00 : MSGBOX : "Status" : "Starting Export"

[Function : ExportObject]

    Parameter : LineNo : Number

    01 : INSERT COLLECTION OBJECT : Name

    02 : SET VALUE : Name : "Led" + "-" + $name+"-" + "00" + $$String : ##LineNo

[Function : ExportEnd]

    00 : MSGBOX : "Status" : "Ending Export"
```

In this example, the line is repeated over the Collection **ExpLedger** which is of type **Ledger**. The event **Export Object** at the line level will be triggered before exporting of every ledger object. The function "Export Object", which is called on occurrence of the event, inserts a new object for the collection "Name" and the method **Name** (alias name) will be set with the new value by concatenating the strings "Led", Name of the ledger and the line no. prefixed with "00".

Before starting and after ending the export, appropriate messages are being displayed to the user through the events 'Before Export' and 'After Export' at 'Form' Level. The exported fragments of XML and SDF outputs can be seen in the following figures, in which we can observe that an alias name is created with the value as set inside the function:

```

- <NAME.LIST TYPE='String">
  <NAME>Customer1</NAME>
  <NAME>Led-Customer1-002</NAME>
</NAME.LIST>

- <NAME.LIST TYPE="String">
  <NAME>Customer2</NAME>
  <NAME>Led-Customer2-003</NAME>
</NAME.LIST>
    
```

Figure 5. XML Format

ULE0000022Name	Customer1
ULE0000022Name	Led-Customer1-002
ULE0000032Name	Customer2
ULE0000032Name	Led-Customer2-003

Figure 6. SDF Format

6. Enhancement – Programmable Configuration

Prior to Tally.ERP 9 release 1.52, when multiple reports were printed or mass mailing was being done in a sequence; before each **Action**, a configuration report was displayed for user input. This would interrupt the flow, thereby requiring a dedicated person to monitor the process, which is time consuming. This had been addressed in Tally.ERP 9 release 1.6, by providing an optional logical parameter to suppress the repeated display for the configuration screen, before the invocation of global actions 'Print', 'Export', 'Upload' and 'Email'.

6.1 Actions enabled for Programmable Configurations

In order to print, export, upload and email the current report in context, the actions 'Print Report', 'Export Report', 'Upload Report' and 'Email Report' are used. Prior to this release, the programmable configuration was not supported for these actions. With the latest enhancement, the display of configuration screen can be suppressed for these actions also. The syntax of these actions supporting programmable configurations is:

Syntax

```
<Action Name> [ :<Report Name> [:<Logical Value>]]
```

Where,

<Action Name> can be any of 'Print', 'Export', 'Mail' and 'Upload'. The actions 'Print Report', 'Export Report', 'Upload Report' and 'Email Report' have also been enabled in the latest release.

<Report Name> is name of the report or a dot (.). Since 'Print Report', 'Export Report', 'Upload Report' and 'Email Report' take the current report in context, and the subsequent parameter is the logical parameter for suppressing the configuration, dot (.) signifies the specification of the current Report Name. This is an optional parameter. However, it is mandatory in case suppress configuration is to be enabled.

<Logical Value> can be TRUE/FALSE or YES/NO. It is an optional parameter. By default, the value is NO. If set to YES, the configuration screen would not be displayed.



The variables to be set as per the requirement of each Action is done in the same way as discussed in prior releases. Refer to the topic "Programmable Configuration for Actions" in Release 1.6 document for more details.

Example:

To export current report without displaying the configuration screen:

```
40: EXPORT REPORT: . : TRUE
```

7. Optional Default TDL Loading

Many Third Party Applications use Tally's rapid application development environment to render various complex reports using Tally Definition Language (TDL). Tally.ERP 9 acts as a front end application for various external databases to retrieve and manipulate information, as and when required. Tally, being a comprehensive business application, loads all the TDLs required as per the functional aspects of the Application. In cases where the third party applications require using Tally purely as a development platform, loading of complete application TDLs may prove to be expensive in terms of startup time.

This release onwards, the application TDLs are segregated as:

- **Base TDL Files** – This contains the commonly required templates like styles, variables, buttons, etc., which can be used by any report which is rendered.
- **Default TDL Files** – This contains the TDLs which are specifically meant for functional requirements of the Tally.ERP 9 application.

This has enabled us to launch Tally using the minimal Base TDL files avoiding the overhead of loading the Default TDL files. This can be achieved by using the command line parameter /NODEF.

7.1 Command Line Parameter - NODEF

Syntax

```
<Tally Application> /NODEF
```

Example:

```
D:\Tally.ERP9\Tally.Exe/NODEF/NOINITDL/TDL:"D:
\Party\CustomReports.TDL"
```

In this case, the Tally.ERP 9 application would start only with Base TDLs, without loading the default TDL Files, which means that Tally Application would start rapidly. None of the INI TDLs will be loaded due to the parameter **/NOINITDL**. Only the TDL file passed with the parameter **/TDL**, D:\Party\CustomReports.TDL will be loaded.



*In line with the above enhancement, the product Tally.Developer 9 Release 3.0 will also support the command line parameter **/NODEF**. In case the application needs to be started with only the Base TDLs, the option **/NODEF** will be used.*

8. Refresh Issues in context of User Defined Function Evaluation

As we are already aware, the TDL Procedural artefact “Function” is used in two scenarios:

1. **Evaluation** – where the function is expected to perform some computation and return the result to the expression within which it is called. The usage is similar to a Predefined function. In evaluation mode, the function is called using a “\$\$”.

Example:

```
[Field : My Field]
Set as : $$MyUserFunction : Parameter1 : Parameter2
```

2. **Execution** – where the function is expected to perform certain set of tasks, which changes the state of the application or the data. The usage is similar to a Predefined Action. In execution mode, the function is called using the keyword “Call” and can be invoked from a Key/Button, a Menu item, an Event, or from within another function.

Example:

```
[Key : My Key]
Action : CALL : MyUserActionFunction : Parameter1 : Parameter2
```

In case of predefined Functions, whenever a function accesses and manipulates certain UI elements like variable, field, etc., or data elements like method values of objects, a link is established between the element and the calling UI. Each time these get manipulated, the function gets re-evaluated, new values are calculated and the corresponding UI is refreshed with new values.

Let us look at the following example to articulate this better:

```
[Variable : My Variable]

    Type    : String

[Field : My Field]

    Type    : String

    Set as  : ##MyVariable
```

When the report is started, the 'Set as' attribute of the field 'My Field' is evaluated. During this evaluation, a link between the Field 'My Field' and the Variable 'My Variable' (which is accessed for its value) is established.

Consider a scenario where, say, a F12:configuration of the report changes the 'MyVariable' value. Now, the system would automatically determine that the Field 'My Field' was depending on the value of the variable, which has changed now; and hence, RE-EVALUATE the field's 'Set As' attribute to get its new value.

In case of a TDL procedural "Function", we faced certain issues, where the fields calling the function for some evaluation were not refreshed with new values when the accessed elements got modified elsewhere, and the function did not get re-evaluated. To articulate this better, let's extend the previous example by using a user defined function.

```
[Variable : My Variable]

    Type : String

[Field : My Field]

    Type    : String

    Set as  : $$MyUserFunction

[Function : My User Function]

    Returns : String

    01      : RETURN : ##MyVariable
```

In this case, the system would not establish any relation between the Field and the variable, as it is processed via a function; and hence, when the Value of the variable is changed elsewhere, the Field's 'Set As' will not get re-evaluated automatically to get its new value.

This issue has been resolved in this Release. The related refresh problems which might have been have faced by the users in context of using "Function" in the evaluation scenario, have been resolved. However, in some negligible cases, we may hit with performance issues due to repeated refresh. This mainly happens when the modification of values of UI / data elements like objects, variables, etc., causes the regeneration of linked UI elements. To overcome the same, certain rules have been established and implemented at the platform level itself. In very few cases where one may require a slight change in design of the function, using the new actions and functions may be useful.

8.1 Function - \$\$ExclEvaluate

This function, when prefixed to an expression, helps in evaluating it without establishing the link with the UI elements. There may be a few cases where the programmer would not want the system to establish relationship between the caller and the object being accessed, to refresh the value in subsequent modification. In such cases, prefixing \$\$ExclEvaluate would indicate the same to the system.

Example:

```
[Variable : My Variable]
```

```
  Type : String
```

```
  [Field : My Field]
```

```
    Type    : String
```

```
    Set as : $$ExclEvaluate:##MyVariable
```

OR

```
    Set as : $$ExclEvaluate:$MyUserFunction
```

```
[Function : My User Function]
```

```
  Returns : String
```

```
  01      : LOG      : ##MyVariable
```

```
  02      : RETURN  : "Constant String"
```

8.2 Action START SUB ... END SUB

In evaluation mode, the dependent regenerations of UI elements are deferred till the function exit. In cases where it is desired to trigger regenerations based on the set of statements as and when they occur, one can enclose the statements within the START SUB - END SUB action block.

To articulate this better, let's take the previous example, where the Variable is being accessed by a field. The following function, on a button press, changes the value of the Variable two times.

```
[Function : My User function]
```

```
  01 : SET : My Variable : "First Value"
```

```
  02 : SET : My Variable : ##MyVariable + ", Second Value"
```

In normal scenario, as both SET actions are modifying the value of the variable, the field (dependent on this variable) would get re-validated twice. However, the platform has the ability to do it only once during the end of the function by default, when the function is called in EVALUATION mode.

To change this behaviour to refresh the field twice, these two SET actions can be covered inside START SUB - END SUB as follows:

```
[Function : My User function]
```



```
01 : START SUB
02 : SET : My Variable : "First Value"
03 : SET : My Variable : ##MyVariable + ", Second Value"
04 : END SUB
```

8.3 Action - SUB ACTION

The purpose of this action is the same as START SUB - END SUB. The only difference is that this action takes an Action to be executed as parameter. The former one encloses a set of Actions inside the block.

Following is the alternative of the previous code by using SUB ACTION, rather than using the START SUB - END SUB action block.

```
[Function : My User function]
01 : SUB ACTION : SET : My Variable : "First Value"
02 : SUB ACTION : SET : My Variable : ##MyVariable + ", Second Value"
```

8.4 Action START XSUB ... END XSUB

In execution mode, the dependent regenerations are handled as and when they occur. In cases where we would like to defer regenerations based on the set of statements, we can enclose the statements within the START XSUB ... END XSUB block.

Let's take the following example to demonstrate this:

```
[Field : My Field]
Set as : $Value1 + $Value2
;; field value depends on the Value1 and Value2 of the current object
[Function: ModifyCurrentObj]
01 : SET VALUE : Value1 : "Something else"
02 : SET VALUE : Value2 : "Another value"
```

This code would normally cause the field to be re-evaluated twice during the function execution. However, enclosing it in an XSUB block would convert it into a single re-evaluation as below:

```
[Function : ModifyCurrentObj]
01 : START XSUB
02 : SET VALUE : Value1 : "Something else"
03 : SET VALUE : Value2 : "Another value"
04 : END XSUB
```

8.5 Action – XSUB ACTION

The purpose of this action is the same as START XSUB and END XSUB. The only difference is that this action takes an Action to be executed as parameter. The former one encloses a set of Actions inside the block.

Following is the alternative of the previous code by using XSUB ACTION, rather than using the START XSUB ... END XSUB block.

```
[Function : ModifyCurrentObj]

    02 : XSUB ACTION : SET VALUE : Value1 : "Something else"

    03 : XSUB ACTION : SET VALUE : Value2 : "Another value"
```

8.6 Report Attribute - 'Plain XML' Introduced

Tally provides the capability to export any report in XML format. The XML generated is in standard format for better readability, i.e., line ending characters after each closing tag, indentation for each sub tag, etc. Most of the applications can directly consume the data available in standard format. However, there are some legacy and non-standard applications which require an XML without formatting and applied styles. They consume the entire unformatted XML available as a single string, without even a new line character.

A new attribute 'Plain XML' has been introduced in 'Report' definition. This attribute generates the XML without applying any formats and styles.

Syntax

```
Plain XML : <Logical Expression>
```

Where,

<Logical Expression> can be any expression which evaluates to logical value YES/NO.

Example:

```
[Report      : Simple Trial balance]

Form        : Simple Trial balance

Title       : "Trial Balance"

Plain XML   : YES
```

8.7 Attribute – Format for Quantity Datatype

In Tally quantity of a Stock item can be expressed using a Simple or Compound Unit of Measure.

Simple Unit – Unit of measure used to express the quantity of an Item. E.g., kgs, nos, pcs, etc.

Compound Unit – The unit of measure which is a combination of Simple units related to each other by a conversion factor, is termed as a Compound Unit. Examples of compound units are kg of 1000 gms, dozen of 12 nos, etc. In case of compound unit, the highest unit is referred to as the Base/Primary unit and the sub units thereafter, are referred to as the Tail units. The quantity is always expressed in terms of the Primary unit. A compound unit can be nested further to contain another compound unit as a Tail unit, up to any no. of levels. E.g., Bag of 10kgs of 1000 gms.

Example, if the unit of measure used for a Stock item “Grains” is Bag of 10kgs of 1000 gms and the closing balance is 12-5-250 bags, it means that the quantity of items is 12 bags 5 kgs 250 gms. Whenever the tail unit quantity crosses the conversion factor, it adds up to the bigger unit. If the gms part exceeds 1000 in this example and the value is 12-5-1250 bags, then it will be converted to 12-6-250 bags.

In TDL, the data type to support representation & storage of data of the above type is Quantity. It comprises of subtypes Number, Base/Primary units, Alternate/Secondary units and unit symbol.

We know that when a method of type ‘Quantity’ is retrieved in a report, it is always expressed in terms of primary units. In case the Unit of Measure is a nested compound unit, the user may require the quantity in terms of any of the units in the entire Compound unit chain. The ‘Format’ attribute of Field has been enhanced to specify the Tail unit, in which the quantity value needs to be extracted.

Syntax

```
[Field : <Field Name>]
    Type      : Quantity
    Set As    : $<Method Name>
    Format    : "Tail Units:" + <String Expression>
```

Where,

<String Expression> must evaluate to any Tail Unit Name used in the Item.

Example:

As per the previous example, the unit of measure used for the Stock item “Grains” is Bag of 10kgs of 1000 gms and the closing balance is 12-5-250 bags. In a field, we may require to retrieve the value in kgs or gms instead on bags. For this, the following specification can be used:

```
[Field : Qty Format Enhancement]
    Use      : Qty Primary Field
    Set As   : $ClosingBalance
    Format   : "Tail Units:" + "kgs"
```

- If Format is “Tail Units:kgs”, value returned is 125kgs250 gms=12X10 kgs+5kgs & 250 gms
- If the Format is “Tail Units:gms”, the value returned would be 125250 gms = 12X10X1000 gms+ 5X1000 gms +250 gms.



Appropriate conversions take place as per the conversion factors set in the nested Compound unit chain.

8.8 Field Attribute - ‘Cell Write’ Introduced

When the data is exported from an external application to Excel Format, especially in the following scenarios, Excel faces refresh issues. Here, we are considering the scenarios when Tally exports the data into Excel.

1. When a cell in an Excel Template is having a formula which depends on multiple cells which are being written from Tally. If one out of these cells is having a drop-down list, then the excel formula is not refreshed after the Export.
2. If the design of Excel template is depending on one of the Excel cells, and this cell is written by Excel Export from Tally, then the template using the contents of this cell will not take these changes into effect.

This problem can be addressed at the TDL level by writing those data corresponding to cells prior to on the one on which the rest of the cells containing the formula/template are dependent. The rest of the data can be written as a chunk only.

For this purpose, a new attribute '**Cell Write**' has been introduced at **Field**. This attribute enables writing of the specific field value in the Excel file, before the entire information gets written.



This attribute has to be used judiciously and strictly as per the above scenarios, since this will increase the export time multifold.

Syntax

Cell Write : <Logical Value>

Where,

<Logical Value> can be YES or NO.

Example:

```
[Field : VAT acc Rate Fld]
```

```
Cell Write : YES
```

8.9 Function - '\$\$StrByCharCode' Introduced

Everyone is aware that the Indian government has recently launched a symbol to represent the Indian currency. To display the same in Tally.ERP 9, a function \$\$StrByCharCode has been introduced in TDL. The function \$\$StrByCharCode accepts the 'ASCII' code or 'Unicode', and displays the corresponding special symbol. This function can be used in scenarios where the special symbols are to be displayed in Tally.ERP 9, e.g., foreign currency symbol.

Syntax

\$\$StrByCharCode : <ASCII code/Unicode>

Where,

<ASCII code/Unicode> can be any expression which returns a valid ASCII or Unicode number. (This number must be in decimal system).

For example the ASCII code for the new rupee symbol is 8377, for Carriage Return is 13, etc.

Example:

```
[Field : StrByCharCode Report]
```

```
Set AS : $$StrByCharCode:@@CodeChar
```

```
[System : Formula]
```

```
Code Char : 8377
```

The new Rupee symbol is displayed in the field 'StrByCharCode Report'.

8.10 Function - '\$\$InPreviewMode' Introduced

In scenarios where the printing events 'Before Print' and 'After Print' were used to trigger an Action, the action used to get called even if the report was in 'Preview' mode. To overcome this problem, the function \$\$InPreviewMode has been introduced, using which, the events can be triggered conditionally as required.

The function \$\$InPreviewMode checks if the report is in 'Preview' mode or not. It is useful in scenarios where some specific controls are to be applied, related to actual Printing. For example, a document can be printed only once, the voucher cannot be altered or deleted after printing an invoice, etc.

Syntax

```
$$InPreviewMode
```

Example:

```
[#Report : Printed Invoice]
```

```
On : Print : NOT $$InPreviewMode : CALL : UpdateDocSetPrintedFlag
```

In this case, the Action created using function '\$\$UpdateDocSetPrintedFlag' is triggered only in 'Print' mode and not in 'Preview' mode.

8.11 Function - '\$\$RemoteUserId' Introduced

In a remote environment, multiple users connect to the same company and access the data therein. All the TDLs available at the server are enabled for the Remote user. There may be scenarios where some restrictions need to be applied to the data access based on the user identity. This can be achieved at the TDL level by using a new function \$\$RemoteUserId, which will return the user name of the remote user accessing the TDL.

Syntax

```
$$RemoteUserId
```

This function, when called in TDL, will return the user name of the remote user at the Server end.

8.12 Function - '\$\$InWords' Enhanced

Till now, the function \$\$InWords accepted only 'Amount' data type and displayed the amount in words. Now, it has been extended to accept 'Number' data type as well, and display it in words.

Syntax

```
$$InWords : <Expression> : <Format String>
```

Where,

<Expression> can be any expression which evaluates to an Amount or a Number.

<Format String> is any string expression used to specify the format, e.g., Forex, No Symbol, etc.

Example:

```
[Field : InWords]
```

```
Set as : $$InWords:100000
```

The function displays "ONE LAKH" in the field 'InWords'.

8.13 Function - '\$\$ContextKeyword' Enhanced

Till now, the function \$\$ContextKeyword has been used to return the Title of the Report or Menu. For scenarios like adding a report to the list of favourites, where the Definition name of the current report is required instead of the report Title, the function \$\$ContextKeyword has been enhanced to return the Report name or Definition name.

Now, the function \$\$ContextKeyword accepts two logical parameters as follows:

```
$$ContextKeyword : [:<1st Logical Expression>] +
                  [:<2nd Logical Expression>]
```

Where,

<1st Logical Expression> can be any expression which evaluates to YES/NO. The default value of this parameter is NO and it returns the Title of the current report. If the value is specified as YES, then the title of the parent report is returned. If no report is active, then the parameter is ignored. If the attribute 'Title' is not specified in 'Report' definition, then by default, the name of the Report definition is returned.

<2nd Logical Expression> can be any expression which evaluates to YES/NO. It specifies that the name of the Report definition should be returned, instead of the Title of the Report.

Example:

```
[Field : Context Keyword Rep]
```

```
Set As : $$ContextKeyword : No : Yes
```

Here, the function \$\$ContextKeyword returns the name of the current report definition.

Example:

```
[Field : Context Keyword Parent]
```

```
Set As : $$ContextKeyword : Yes : Yes
```

Here, the function \$\$ContextKeyword returns the name of the parent Interface definition, i.e., either a Menu Definition Name or the Parent Report Definition Name.

What's New in Release 2.0

1. TDL Procedural Enhancements

With every Release, the TDL Procedural Capabilities are getting strengthened at a commendable pace. The latest along this path is the File Input/Output Capability.

1.1 TDL Procedural File Input/Output Capabilities

As we are aware, any High level programming language will support Reading and Writing From/To multiple hardware devices. It will have predefined constructs in the form of functions to Read from and Write to a File as well. This file can reside on the hard disk or on a network, which can be accessed via various protocols HTTP or FTP.

This capability introduced in TDL will now pave the way for supporting import/export operations from the Tally DataBase in the most amazing way. It will now be possible to export almost every piece of information in any Data Format which we can think of. The Text and Excel Formats are supported, which allow data storage in SDF-Fixed Width, CSV-comma separated, etc., sufficing the generic data analysis requirements of any business.

The TDL artefacts used for supporting various Read/Write operations are Actions and Functions. These are made to work only inside the TDL Procedural Block. 'Write' operations are mostly handled using Actions, and all the file Access and Read operations are performed using Functions. These give tremendous control in the hands of the programmer for performing the data manipulations To/From the file. And that too, on a file present on a network accessible using the protocols FTP and HTTP. Since these artefacts operate on characters and not bytes, the file encoding ASCII/UNICODE does not have any effect on the operations.

File Contexts

The entire procedural Read/Write artefacts basically operate on two file contexts:

- **Source file Context**

When a file is opened for Reading purpose, the context available for all the 'read' operations is the Source File Context. All the subsequent 'read' operations are performed on the Source File Context.

- **Target file Context**

When a file is opened for Writing purpose, the context available for all the 'write' operations is the Target File Context. All the subsequent Write operations are performed on the Target File Context.

It is important to understand that these contexts are available only inside the procedural block (User Defined Function), where the files are opened for use. The file context concept is different from the concept of Object Context where the Object context is carried over to all its child Objects. File Contexts are only available to the functions, and the subsequent functions are called from within the parent Function. The called function can override the inherited context by opening a new file within its block.

The file context created by opening a file is available only till the execution of the last statement. Once the control is out of the function, the file is automatically closed. However, it is highly recommended as a good programming practice to close a file on exit.

Both the file contexts, i.e., Source and Target file contexts, are available simultaneously. This makes it possible to read from one file and write to another file simultaneously.

File Operations

A programming language supporting File Read/Write typically supports the following operations:

- Open- This operation identifies the file which needs to be opened for Read/Write purpose.
- Close- This operation closes the opened file after Read/Write.
- Read- This is an operation to read the data from an opened File.
- Write- This is an operation to write the data to an opened File.
- Seek- This is an operation to access the character position in an already opened file.
- Truncate- This is an operation which will delete a particular number of characters/entire contents of the file.

General File Operations

As discussed, the entire procedural Read/Write concepts basically operate on two file contexts, i.e., a source file context and a target file context. Source context is used to read the contents from a file which is opened for reading purpose, whereas the target context is used to write the data to a file which is opened for writing purpose. Since both these file contexts are available simultaneously, it is possible to read from one file and write to another file.

□ Action – OPEN FILE

It is used to open a text/excel file for read/write operations. The file can reside in Hard Disk, in the main memory or on FTP/HTTP site. Also, it is used to open a file for read/write operations.

If no parameters are used, then a memory buffer will be created, which will act as a file. This file will be in both read / write mode and will act as both the source and the target context. Based on the mode specified (read/write), the file automatically becomes the source file or the target file, respectively.

Syntax

```
OPEN FILE [:<File Name> [:<File Format> [:<Open Mode> [:<Encoding> ]]]]
```

Where,

<File Name> can be any expression which evaluates to a regular disk file name like C:\Output.txt, or to a HTTP/FTP site like "ftp://ftp.tallysolutions.com/Output.txt"

<File Format> can be Text or Excel. By default, 'Text' mode will be considered, if not specified. Also, during opening of an existing file, if the mode does not match, the Action will fail.

<Open Mode> can be Read / Write. By default, it is Read. If a file is opened for 'Read' purpose, then it must exist. If 'Write' mode is specified, and the file exists, it will be opened for updating. If the file does not exist, a new file is created. If the file is opened in 'Write' mode, it is possible to read from the file as well.

<Encoding> can be ASCII or Unicode. If not specified, 'Unicode' is considered as the default value for 'Write' Mode. In 'Read' mode, this parameter will be ignored and considered, based on the encoding of the file being read.

Example: 1

```
10 : OPEN FILE : "Output.txt" : Text : Write : ASCII
```

A Text File 'Output.txt' is opened in 'Write' mode under the Tally application Folder, and if it already exists, the same will be opened for appending purpose.

Example: 2

```
10 : OPEN FILE : "http://www.tallysolutions.com/Output.txt" : Text
```

A Text File 'Output.txt' is opened in 'Write' mode at the HTTP URL specified. If the file already exists, the same will be opened for appending purpose.

Example: 3

```
10 : OPEN FILE : "C:\Output.xls" : Excel : Read
```

An Excel File 'Output.xls' is opened in 'Read' mode under C drive, and if the file does not exist, the Action will fail.



Please refer to the functions like `$$MakeFTPName` and `$$MakeHTTPName` for constructing the FTP and HTTP URLs using the various parameters like server name, username, password, etc. Refer to Tally.Developer 9 Function Browser for help on usage.

□ Actions – CLOSE FILE and CLOSE TARGET FILE

A file which is opened for Read/Write operation needs to be closed, once all the read/write operations are completed. However, if the files are not closed explicitly by the programmer, these are closed by default when the function is returned. But, it is always recommended to close the file after the operations are completed.

Files which are opened in the current function can only be closed by it. If a function inherits the file contexts from the caller function, then it cannot close these files; however, it can open its own instances of the files. In such cases, the caller context files will not be accessible.

□ Action – CLOSE FILE

This action is used to close an opened source file.

Syntax

```
CLOSE FILE
```

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
.
.
30: CLOSE FILE
```

In this example, the Excel file 'Output.xls', which is opened for reading purpose, is closed.

▫ **Action – CLOSE TARGET FILE**

This action is used to close an opened target file.

Syntax

CLOSE TARGET FILE

Example:

```
10 : OPEN FILE : "Output.txt" : Text : Write
.
.
.
30 : CLOSE TARGET FILE
```

In this example, the text file 'Output.txt', which is opened for writing purpose, is closed.

General Functions - \$\$TgtFile, \$FileSize

▫ **Function - \$\$TgtFile**

All the file accessing functions, for both text and excel files, operate on the source file context. The function **\$\$TgtFile** can be used to switch to the target file context temporarily. It evaluates the expression passed, in the context of the target file.

Syntax

\$\$TgtFile : Expression

Example:

In this example, the objective is to Read the contents of a cell in Sheet 1 and copy it to a cell in the Sheet 2 of the same file. The function opens the File "ABC.xls" in 'Write' mode.

[Function : Sample Func]

```
Variable : Temp : String
10 : SET      : Temp : ""
20 : OPEN FILE : "Output.xls" : Excel : Write
30 : ADD SHEET : "Sheet 1"
40 : WRITE CELL : 1 : 1 : "Item A"
50 : SET: Temp : $$TgtFile : $$FileReadCell:1:1
60 : ADD SHEET : "Sheet 2"
70 : WRITE CELL : 1 : 1 : ##Temp
80 : CLOSE TARGET FILE
```

In this example, there is no file present in the source file context, as the file is opened in the 'Write' mode. In such case, for reading a value from Sheet 1, the expression **\$\$FileReadCell:1:1** will return no value. Prefixing the expression with **\$\$Tgtfile** will temporarily change the context to Target File for the evaluation of the expression, and will fetch the value from the cell 1 of Sheet 1, existing in the Target File context.

□ **Function - \$\$FileSize**

This function will return the size of the file, specified in bytes. It takes an optional parameter. If the parameter is not given, then it works on the current context file and returns the size.

Syntax

```
$$FileSize [:<FileName>]
```

Where,

<FileName> is an expression, which evaluates to the file name, along with the path.

Example:

```
10 : Log : $$FileSize : "Output.xls"
```

It gives the size of the Excel file 'output.xls', in terms of bytes.

Read/Write Operation on Text Files

Writing to a File

Various Actions have been introduced in order to write to a text file. These Actions operate on the Target File context. The scope of these Actions is within the TDL procedural block (User Defined Functions), where the file is opened and the context is available.

□ **Action – WRITE FILE**

This Action is used to append a file with the text specified. The 'write' operation always starts from the end of the file. This Action always works on the target file context.

Syntax

```
WRITE FILE : <TextToWrite>
```

Where,

<TextToWrite> can be any expression evaluating to the data that needs to be written to the file.

Example:

```
10 : OPEN FILE : "Output.txt" : Text : Write
```

```
20 : WRITE FILE : "Krishna"
```

```
30 : CLOSE TARGET FILE
```

Here, a txt file 'Output.txt' is opened in 'write' mode and the content 'Krishna' is appended at the end of the file.

□ **Action – WRITE FILE LINE**

This Action is similar to WRITE FILE, but it also places a new line character (New Line/Carriage Return) after the text. All the subsequent 'writes' begin from the next line. This Action always works on the target context.

Syntax

```
WRITE FILE LINE : <TextToWrite>
```

Where,

<TextToWrite> can be any expression evaluating to the data that needs to be written to the file.

Example:

```
10 : OPEN FILE : "Output.txt" : Text : Write
20 : WRITE FILE LINE : "Line 1"
30 : WRITE FILE LINE : "Line 2"
40 : CLOSE TARGET FILE
```

Here, a txt file 'Output.txt' is opened in 'Write' mode, and two more lines are appended at the end of the file.

- **Action – TRUNCATE FILE**

This action removes the contents of the file by setting the file pointer to the beginning of the file and inserting an 'end of file' marker. It can be used to erase all characters from an existing file, before writing any new content to it.

Syntax

```
TRUNCATE FILE
```

Example:

```
10 : OPEN FILE : "Output.txt" : Text : Write
20 : TRUNCATE FILE
30 : WRITE FILE : "New Value"
40 : CLOSE TARGET FILE
```

In this example, the entire contents of the existing txt file 'Output.txt' are removed and 'New Value' is inserted subsequently.

- **Action – SEEK FILE**

This Action operates on the Target File Context. It is used to move the file pointer to a location as specified by the number of characters. As we already know that it is possible to Read and Write from the Target File context, all the subsequent Reads and Writes will be starting from this position. By Default, if the file position is not specified, the 'Read' pointer will be always be from the beginning of file and the 'Write' pointer will be from the end of the file.



It has already been covered how to Read from the Target File Context by using the function \$\$TgtFile.

Syntax

```
SEEK FILE : <File Position>
```

Where,

<File Position> can be any expression, which evaluates to a number which is considered as the number of characters.

Reading a File

Some functions and Actions have been introduced, which can operate on the Source File context to read from the file or access some information from them. The scope of these functions is within the TDL procedural block (User Defined Functions) where the file is opened, and the context is available. It is also possible to read from the Target File Context by using the function \$\$TgtFile.

□ Function - \$\$FileRead

This function is used to read data from a text file. It takes an optional parameter. If the parameter is not specified or has value as 0, it reads one line and ignores the 'end of line' character. However, file pointer is positioned after the 'end of line' character, so that the next read operation starts on the next line. If the no. of characters is mentioned, it reads that many no. of characters.

Syntax

```
$$FileRead [:<CharsToRead>]
```

Where,

<CharsToRead> can be any expression which evaluates to the number of characters to be read.

Example:

```
10 : OPEN FILE : "Output.txt" : Text : Read
20 : LOG : $$FileRead
30 : CLOSEFILE
```

In this example, the first line of the text file 'Output.txt' is being read.

□ Function - \$\$FileIsEOF

This function is used to check if the current character being read is the End of file character.

Syntax

```
$$FileIsEOF
```

□ Action – SEEK SOURCE FILE

This Action works on a source file context. It sets the current file pointer to the position specified. Zero indicates the beginning of the file and -1 indicates the end of the file. The file position is determined in terms of the characters. All the subsequent reads begin from this position onwards.

Syntax

```
SEEK SOURCE FILE : <File Position>
```

Where,

<File Position> can be any expression evaluating to a number, which is the no. of characters.

Example:

```

10 : OPEN FILE : "Output.txt" : Text : Read
20 : SEEK SOURCE FILE : 2
30 : LOG : $$FileRead
40 : CLOSE FILE

```

In this example, the first line of the file 'Output.txt' is read, starting from the 3rd character.

Read/Write Operation on Excel Files**Setting the Active Sheet**

For an Excel file, all the Read and Write operations will be performed on the Active Sheet.

- **Action – SET ACTIVE SHEET**

This Action is used to change the Active Sheet during Read and Write operations.

Syntax

```
SET ACTIVE SHEET : <Sheet Name>
```

Where,

<Sheet Name> is an expression evaluating to a string, which is considered as name of the sheet.

Example:

```

10 : OPEN FILE : "Output.xls" : Excel : Read
20 : SET ACTIVE SHEET : "Sheet 2"
30 : Log : $$FileReadCell : 1 : 1
40 : CLOSE FILE

```

In this example, an Excel sheet Output.xls is opened in Read mode, 'Sheet 2' is made active and the content is read from the first cell.

Writing to a File

Various Actions have been introduced in order to write to an excel file. These Actions operate on the Target File context. The scope of these Actions is within the TDL procedural block (User Defined Functions), where the file is opened and the context is available.

- **Action – ADD SHEET**

This Action adds a sheet in the current workbook opened for writing. The sheet will always be inserted at the end. If a sheet with the same name already exists, it will be made as active.

Syntax

```
ADD SHEET : <Sheet Name>
```

Where,

<Sheet Name> is an expression evaluating to a string, which is considered as name of the sheet.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Write
20 : ADD SHEET : "New Sheet"
```

Here, an existing Excel sheet 'Output.xls' is opened in 'Write' mode and the new sheet 'New Sheet' is inserted at the end of the workbook.

□ Action – REMOVE SHEET

This Action removes the specified sheet from current workbook. The entire contents of the sheet will be removed. This Action will fail if the workbook has only one sheet, or if the specified sheet name does not exist in the workbook.

Syntax

```
REMOVE SHEET : <Sheet Name>
```

Where,

<Sheet Name> is an expression evaluating to a string, which is considered as name of the sheet.

Example:

```
10 : OPEN FILE      : "Output.xls" : Excel : Write
20 : ADD SHEET      : "New Sheet"
30 : REMOVE SHEET  : "Sheet1"
```

In this example, a workbook is created with a sheet named 'New Sheet'.

□ Action – RENAME SHEET

This action renames a work sheet.

Syntax

```
RENAME SHEET : <Old Sheet Name> : <New Sheet Name>
```

Where,

<Old Sheet Name> and **<New Sheet Name>** can be any expression, evaluating to a string, which will be considered as the name of the sheet.

Example:

```
01 : OPEN FILE      : "Output.xls" : Excel : Write
02 : RENAME SHEET  : @@OldSheetName : @@NewSheetName
04 : CLOSE TARGET FILE
```

In this example, the existing sheet is renamed with a new sheet name.

□ Action – WRITE CELL

This Action Writes the specified content at the cell address specified by the row and column number of the currently active sheet.

Syntax

```
WRITE CELL : <Row No> : <Column No> : <Content To be Written>
```

Where,

<Row No> and **<Column No>** can be any expression which evaluates to a number, which can be used to identify the cell

<Content To be Written> can be any expression which evaluates to data, which needs to be filled in the identified cell.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Write : ASCII
15 : ADD SHEET : "New Sheet"
20 : WRITE CELL : 1 : 1 : "Krishana"
30 : CLOSE TARGET FILE
```

It opens an Excel File 'Output.xls', adds a new sheet, and in that sheet, the first cell will have the content as 'Krishna'.

□ **Action – WRITE ROW**

This Action writes multiple cell values at a specified row in the Active sheet. The no. of values separated by commas are written, starting from initial column no. specified, for the row specified.

Syntax

```
WRITE ROW : <Row No> : <Initial Column No> : <Comma Separated Values>
```

Where,

<Row No> and **<Initial Column No>** can be any expression which evaluates to a number, which can be used to identify the cell.

'Comma Separated Values' can be expressions separated with comma, which evaluate to data that needs to be filled, starting from the cell mentioned by 'Row Number' and 'Initial Column Number'.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Write
20 : ADD SHEET : "New Sheet"
30 : WRITE ROW : 1 : 1 : @@Val1, @@Val2
40 : CLOSE TARGET FILE
```

Here, cells (1,A) and (1,B) are filled with the values from expressions 'Val1' and 'Val2'.

□ **Action – WRITE COLUMN**

This Action writes multiple cell values at a specified column in the Active sheet. The no. of values separated by commas are written starting from the initial row no., specified for the column.

Syntax

```
WRITE COLUMN : <Initial Row No> : <Column No> : <Comma Separated Values>
```

Where,

<Initial Row No> and **<Column No>** can be any expression, evaluating to a number, which can be used to identify the cell.

'Comma Separated Values' can be expressions separated with comma, which evaluate to data that needs to be filled, starting from the cell mentioned by 'Initial Row Number' and 'Column Number'.

Example:

```
10 : OPEN FILE      : "Output.xls" : Excel : Write
20 : ADD SHEET      : "New Sheet"
30 : WRITE Column   : 5 : 5 : @@Val3, @@Val4
40 : CLOSE TARGET FILE
```

In this example, cells (5, E) and (6,E) are filled with the values from expressions 'Val3' and 'Val4'.

Reading a File

Some functions and Actions have been introduced which can operate on the Source File context to read from the file or access some information from them. The scope of these functions is within the TDL procedural block (User Defined Functions), where the file is opened and the context is available. It is also possible to read from the Target File Context by using the function \$\$TgtFile.

- **Function - \$\$FileReadCell**

This function returns the content of the cell identified by the mentioned row number and column number of the active sheet.

Syntax

```
$$FileReadCell : <Row No> : <Column No>
```

Where,

<Row No> and **<Column No>** can be any expression which evaluates to a number used to identify the row number and the column number.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : SET ACTIVE SHEET      : "Sheet 1"
20 : Log : $$FileReadCell : 1 : 1
```

The Function \$\$FileReadCell Logs the contents of the first cell of the excel sheet 'Sheet 1'.

- **Function - \$\$FileGetSheetCount**

This function returns the number of sheets in the current workbook.

Syntax

```
$$FileGetSheetCount
```

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       : $$FileGetSheetCount
```

The Function `$$FileGetSheetCount` returns the total number of sheets in the Excel sheet 'Output.xls'.

- **Function - \$\$FileGetActiveSheetName**

This function returns the name of the active sheet.

Syntax

```
$$FileGetActiveSheetName
```

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       : $$FileGetActiveSheetName
```

The name of the Active sheet is returned after opening the Excel file 'Output.xls'.

- **Function - \$\$FileGetSheetName**

This Function returns the name of the sheet at a specified index.

Syntax

```
$$FileGetSheetName : <Sheet Index>
```

Where,

<Sheet Index> can be any expression which evaluates to a number as the Sheet Index.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
Log : $$FileGetSheetName:1
```

The Function `$$FileGetSheetName` gives the name of the sheet at a particular index

- **Function - \$\$FileGetSheetIdx**

This Function Returns the Index of the sheet for a specified sheet name.

Syntax

```
$$FileGetSheetIdx : <Sheet Name>
```

Where,

<Sheet Name> can be any expression which evaluates to the name of the Excel Sheet.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       : $$FileGetSheetIdx : "Ledgers"
```

The Function `$$FileGetSheetIdx` gives the index number of the sheet name.

□ Function - `$$FileGetColumnName`

This Function gives the column name in terms of alphabets for the given index.

Syntax

```
$$FileGetColumnName:Index
```

Where,

<Index> can be any expression which evaluates to the Index number.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       : $$FileGetColumnName : 10
```

The Function `$$FileGetColumnName` returns the value **J**.

□ Function - `$$FileGetColumnIdx`

This function returns the index of the column for a given alphabetical name.

Syntax

```
$$FileGetColumnIdx : <Name>
```

Where,

<Name> can be any expression which evaluates to the name of the column in alphabets.

Example:

```
10 : OPEN FILE : "Output.xls" : Excel : Read
20 : Log       : $$FileGetColumnIdx:AA
```

The Function `$$FileGetColumnIdx` returns the value as **27**.

Use Case – Import from Excel**Scenario**

ABC Company Limited, which is into trading business, is using Tally.ERP 9. It deals with purchase and sale of computers, printers, etc. The company management wants to import the stock items from the Excel sheet, or a text file into Tally.ERP 9.

Functional Demo

A configuration report is added in Tally.ERP 9 to accept the file location, work sheet name, column details, etc. An option to display the error report can also be specified.

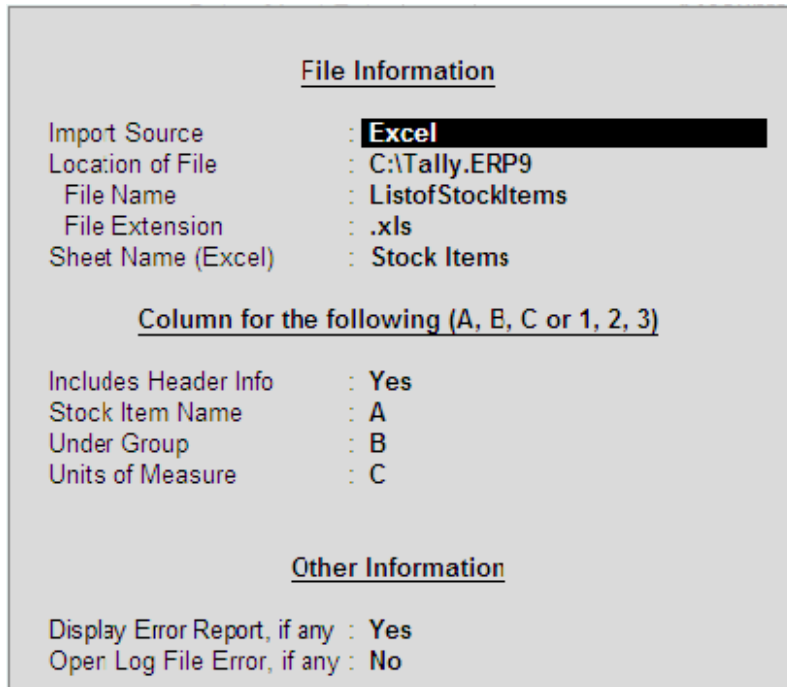


Figure 1. The Configuration Report

By default, 'Excel' format is selected. But, the user can also select the Import source format as 'Text' and specify the file details. The text separator character should be specified as well, in addition to the column details.

File Information

Import Source : **Text**

Location of File : C:\Tally.ERP9

File Name : ListofStockItems

File Extension : txt

Column for the following (A, B, C or 1, 2, 3)

Includes Header Info : **Yes**

Stock Item Name : **1**

Under Group : **2**

Units of Measure : **3**

Text Separator Character : ,

Other Information

Display Error Report, if any : **Yes**

Open Log File Error, if any : **No**

Source

Excel

Text

Figure 2. The Configuration Report

Once the details are entered, a confirmation message is displayed to start the import process.

If the user has selected the option to display the error report after successful import, the report is shown with imported stock items and status as “Imported successfully”, as seen in the figure:

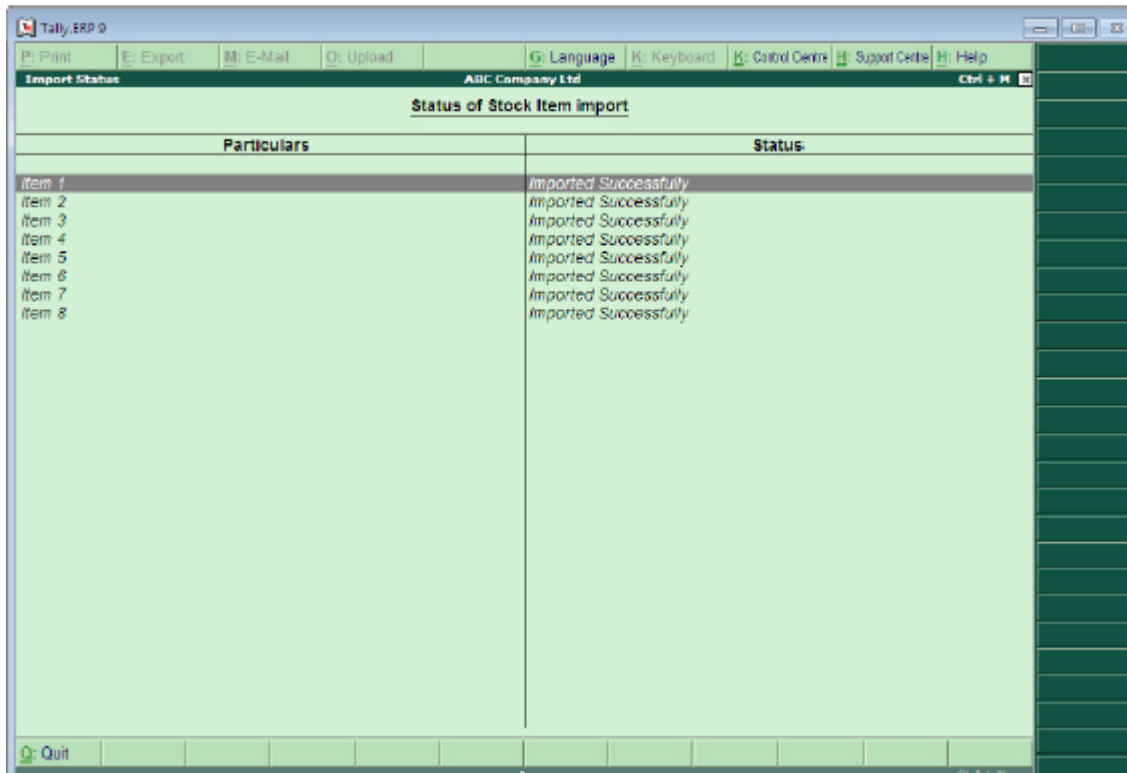


Figure 3. Success Report

If the user has selected the option to display the Log file, then after the import, the log file is displayed as follows:

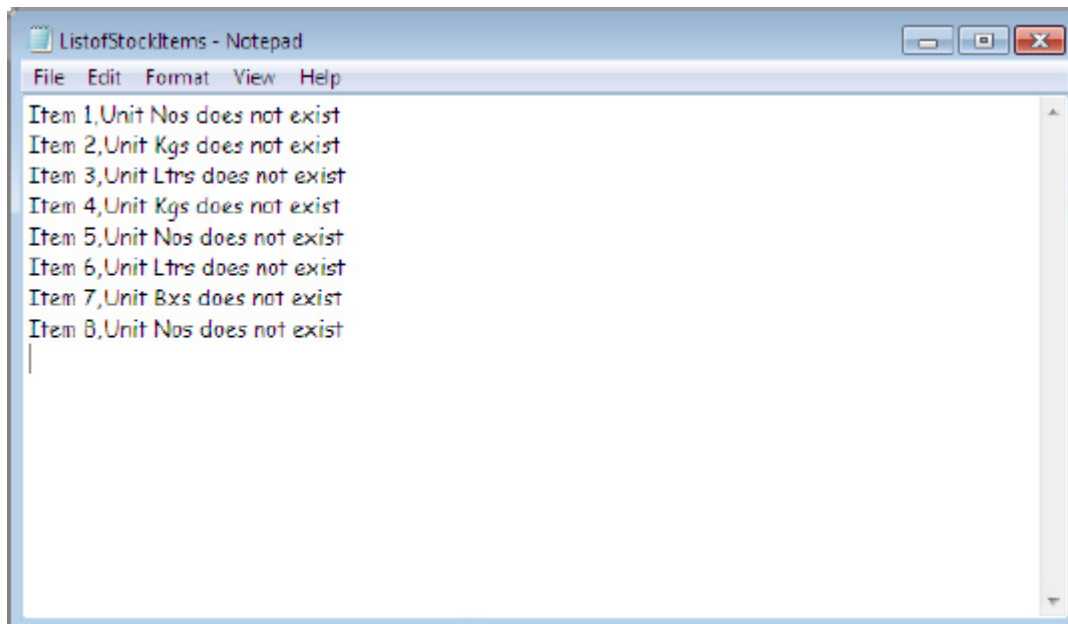


Figure 4. Log File

The imported items are now available in the Stock Item list as follows:

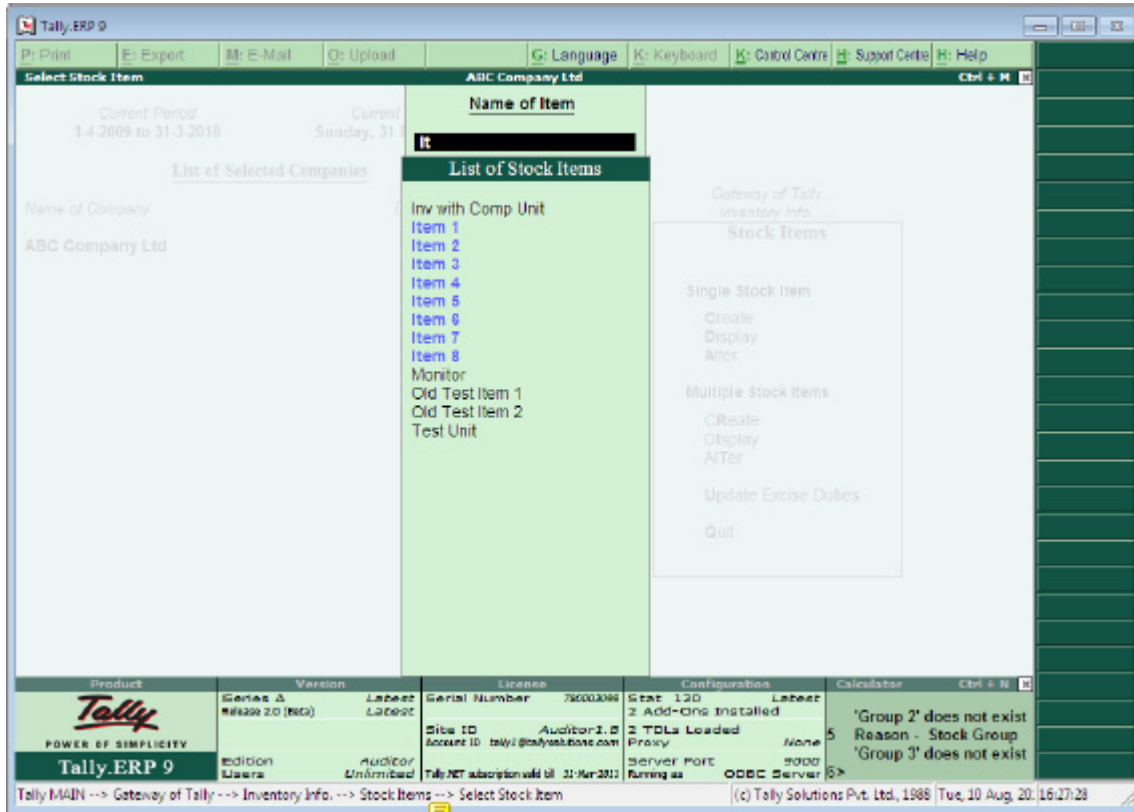


Figure 5. List of Stock Items

In case the import is unsuccessful, the error report, with the reason of failure, is displayed as follows:

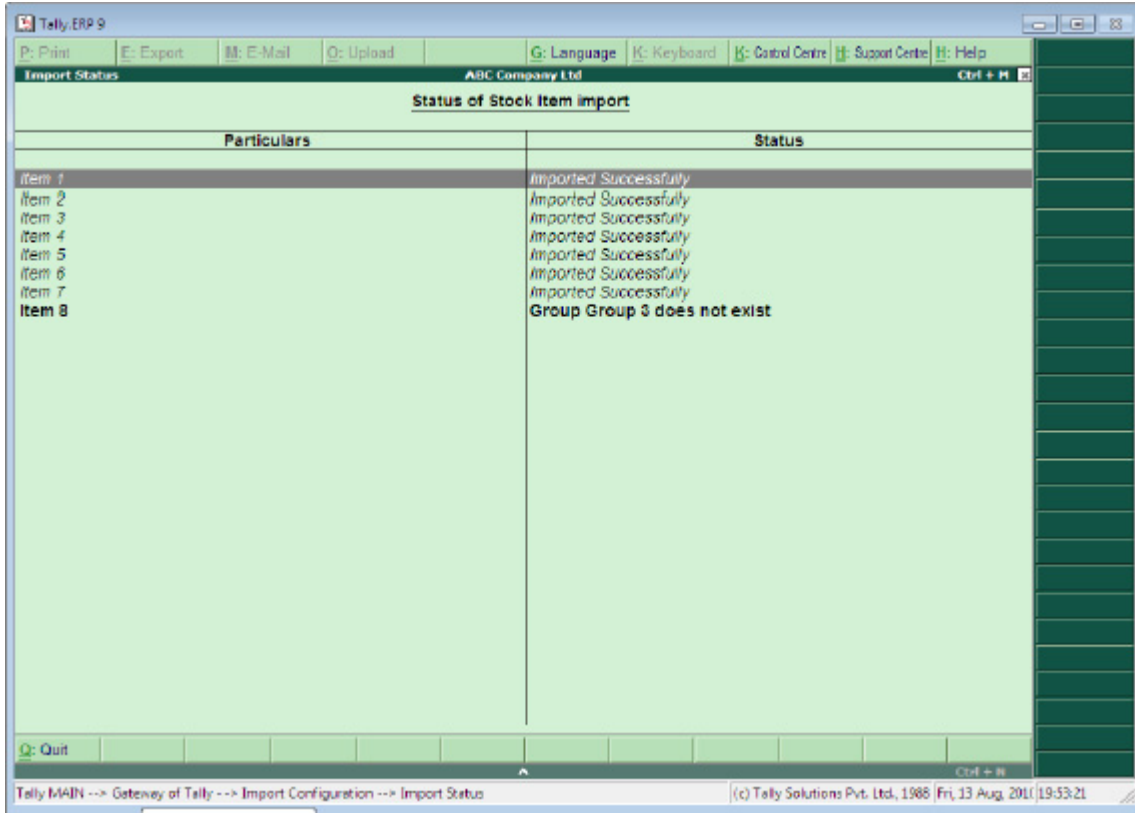


Figure 6. Error Report

Solution Development

The import from the excel file is simplified, as the user can specify the import details. The file I/O capabilities are used to develop the solution. The steps followed to achieve the requirement are:

1. A report is designed to display the configuration report. The value entered by the user is stored in a system variable.

```
Local : Field : Name Field : Modifies : SIC Source : Yes
```

```
Local : Field : Name Field : Variable : SIC Source
```

```
|
|
```

```
Local : Field : Name Field : Modifies : SIC DirPath : Yes
```

```
Local : Field : Name Field : Variable : SIC DirPath
```

```
[System : Variable]
```

```
SIC Source : "Excel"
```



```
SIC DirPath : "C:\Tally.ERP9"
```

2. On form accept, the function to import the stock item is called.

```
On : Form Accept : Yes : Form Accept
```

```
On : Form Accept : Yes : Call : Smp Import Stock Items
```

3. A function "Smp Import Stock Items" is defined.

a. In this function, first of all, the format of the source file is checked, and then, the action 'Open File' is used to open the file in 'Read' mode accordingly.

```
20 : IF : ##SICSource = "Excel"
```

```
30 : OPEN FILE : @@TSPLSMPTotFilePath : Excel : READ
```

```
40 : ELSE :
```

```
50 : OPEN FILE : @@TSPLSMPTotFilePath : Text : READ
```

```
60 : ENDIF
```

b. The data from the Excel cells are read and added as items in the list variable.

```
120 : WHILE : NOT $$IsEmpty:($$FileReadCell : ##Row : +
```

```
##ItemColumns.ItemName)
```

```
130 : LIST ADD EX : Item Details
```

```
140 : SET : ItemDetails[$$LoopIndex].ItemName : $$FileReadCell : +
```

```
##Row : ##ItemColumns.ItemName
```

```
150 : SET : ItemDetails[$$LoopIndex].ItemGrp : $$FileReadCell : +
```

```
##Row : ##ItemColumns.ItemGrp
```

```
160 : SET : ItemDetails[$$LoopIndex].ItemUOM : $$FileReadCell : +
```

```
##Row : ##ItemColumns.ItemUOM
```

```
170 : INCREMENT: Row
```

```
180 : END WHILE
```

c. If source format is 'Text', the text file is read line by line and added as items to the list variable.

```
210 : WHILE : NOT $$FileIsEOF
```

```
220 : SET : Temp Var : $$FileRead
```

```
230 : IF : NOT $$IsEmpty:##TempVar AND (NOT ##SICIncHeader OR +
```

```
(##SICIncHeader AND $$LoopIndex > 1))
```

```
240 : LIST ADD EX : Item Details
```

```
250 : SET : ItemDetails[##Counter].ItemName : $$SICExtractDet: +
```

```

                ##TempVar : ##ItemColumns.ItemName

260 : SET : ItemDetails[##Counter].ItemGrp : $$SICExtractDet : +
                ##Temp Var : ##ItemColumns.ItemGrp

270 : SET : ItemDetails[##Counter].ItemUOM : $$SICExtractDet: +
                ##TempVar : ##ItemColumns.ItemUOM

280 : INCREMENT : Counter

290 : ENDIF

300 : END WHILE

```

d. A collection is populated using the List variable as data source.

```

[Collection : TSPL SMP Imp StockItem]

    Data Source : Variable : Item Details

[Collection : TSPL SMP Imp StockItem Summ]

    Source Collection : TSPL SMP Imp StockItem

    By : SICStockItem : $ItemName

    By : SICStockGroup : $ItemGrp

    By : SICStockUOM : $ItemUOM

    Filter : TSPL SMP NonEmpty Item

```

e. Now, the Stock Item objects are created. If the item can't be imported, then the item details are written in the error file or compound variable, based on the format selected for displaying, i.e., 'Report' or 'Log'.

```

380 : WALK COLLECTION : TSPL SMP Imp StockItem Summ

390 :     SET : Last Status : ""

400 : IF : $$IsEmpty : $Name : StockItem : $SICStockItem

410 : NEW OBJECT: Stock Item

420 :     SET VALUE : Name : $SICStockItem

430 : IF : NOT $$IsEmpty : $Name : StockGroup : $SICStockGroup

440 : SET VALUE : Parent : $SICStockGroup

450 : ELSE :

460 : SET : LastStatus : "Group" + $SICStockGroup + "does not exist"

```

```
470 : ENDIF
480 : IF : NOT $$IsEmpty:$Symbol:Unit:$SICStockUOM
490 : SET VALUE : Base Units : $SICStockUOM
500 : ELSE :
510 : SET : LastStatus : "Unit" + $SICStockUOM + "does not exist"
520 : ENDIF
530 : IF : $$IsEmpty : ##LastStatus
540 : SAVE TARGET
550 : SET : Last Status : "Imported Successfully"
560 : ENDIF
570 : ENDIF
```

;; Writing Import Status to the LOG File, if LOG File is to be displayed at the end

```
580 : IF : ##SICOpenLogFile
590 : WRITE FILE LINE : $SICStockItem + ##SICTextSep + ##LastStatus
600 : ENDIF
```

;; Updating List of Compound Variables if the Status is to be displayed in a Report

```
610 : IF : ##SICDisplayReport
620 : LIST ADD EX : ItemImportStatus
630 : SET : ItemImportStatus[##Counter].ItemName : $SICStockItem
640 : SET : ItemImportStatus[##Counter].Status : ##LastStatus
650 : INCREMENT : Counter
660 : ENDIF
670 : END WALK
```

- f. If the format selected is 'Report', then the stock item name and the status is updated in a compound variable; whereas, if the format selected is Log file, then the action 'Write File' is used to write in the file.

```
WRITE FILE LINE : $SICStockItem + ##SICTextSep + ##LastStatus
```

- g. After import, if the user wants to display error report, a function is called to display the same.

```
690 : IF : ##SICDisplayReport
700 : DISPLAY : TSPL Smp SIC Error Report
```

```
710 : ENDF
```

h. After the import, if the user has selected to display the log file, then the log file is displayed.

```
720 : IF : ##SICOpenLogFile
```

```
730 : EXEC COMMAND : @@TSPLSmpErrorFilePath
```

```
740 : ENDF
```

4. The Error Report displays the reason of failure, if the Stock Item cannot be imported. In error report, the line is repeated over the collection populated, using list variable as the data source.

1.2 Function Parameter Changes – Optional Parameters

Prior to this Release, while invoking a user defined function, it was mandatory to pass values to all the parameters declared within the function. Now, the capability has been introduced to have optional parameters. The function will execute smoothly even if the caller of the function does not pass the values to these optional parameters. However, the caller of the function must pass all the mandatory parameters. Only the rightmost parameters can be optional, i.e., any parameter from the left or middle cannot be optional.

If the Parameter value is supplied by the calling Function, then the same is used, else the default Parameter value provided within the 'Parameter' Attribute is used as the Parameter value. For this enhancement, the Function attribute '**Parameter**' has been modified to accept parameter value.

Syntax

```
[Function : <Function Name>]
  Parameter : <Parameter Name1> : <Data Type>
  Parameter : <Parameter Name2> : <Data Type>
  Parameter : <Parameter Name3> : <Data Type> [: Parameter Value]
  Parameter : <Parameter Name4> : <Data Type> [: Parameter Value]
```

Where,

<Parameter Name1> and **<Parameter Name2>** are mandatory parameters for which, values must be passed while calling the function.

<Parameter Name3> and **<Parameter Name4>** are optional parameters for which, values may or may not be passed while calling the function. If values for these parameters are passed, the parameter value specified within the 'Parameter' Attribute is ignored. In absence of these values, the specified parameter value is taken into consideration.



Parameter Value indicates **Optional Parameters**, and all the **Optional Parameters** should be the rightmost elements of the function.

Example:

```
[Function : Split VchNo]

;; this Function returns the number part of voucher number from a string
;; For e.g., Voucher Number is Pymt/110/2010-11. This Function will return only '110'.
Parameter : pVchNo : String

Parameter : pSplitChar : String : "/"

;; usual separator

00 : FOR TOKEN : TokenVar : ##pVchNo : ##pSplitChar

10 : IF      : $$LoopIndex = 2

20 : RETURN : ##TokenVar

30 : ENDIF

40 : END FOR
```

While invoking the function **\$\$SplitVchNo**, only the Voucher No is passed. The 2nd Parameter is optional and the default value is "/". It is passed only if the separator character is other than "/".

Optional Parameters can be very useful where the Parameter values remain constant in most of the cases; and rarely require some change.



A small change has been done in the way function parameters are tokenized. The last parameter passed to the function is not broken up into sub-parts now. This is particularly useful in cases where we require the result of one function to be treated as a parameter to another function. In other words, if a function requires 4 parameters, it tokenizes only till 3 parameters and all the subsequent values are considered as the 4th parameter (last parameter).

2. Variable Framework Enhancements

In the prior releases, we have experienced major changes to the Variable Framework in the form of introduction of Compound Variables and List Variables. Continuous enhancements and changes are being made to ensure consistency and uniformity across the TDL framework. The following enhancements have taken place in variable framework recently.

2.1 Variable Persistence at 'Report' Scope

Variables at 'report' scope can now be persisted into a user specified file. This is stored in a standard variable format and also allows reloading the report scope variables from the specified file. The Actions SAVE VARIABLE and LOAD VARIABLE have been introduced for this purpose.

□ Action - SAVE VARIABLE

The action SAVE VARIABLE is used to persist the Report Scope Variables in a user specified file.

Syntax

```
SAVE VARIABLE : <FileName> [:<Variable List>]
```

Where,

<FileName> is the name of the file in which the report scope variables are persisted. The extension .PVF will be taken by default, if the file extension is not specified.

<Variable List> is the list of comma-separated variables that need to be persisted in the file. Specifying the variable list is optional.



If the Variable List is not specified, all the variables at the 'Report' scope, which have 'Persist' attribute set to YES, will be persisted in the specified file.

We need not declare the variable at System level unless it is required to persist the same in the default configuration file tallycfg.tsf.

Example:

Let us assume that the variables **EmpNameVar** and **EmpIDVar** are declared at the Report Scope, and the same need to be persisted in a user specified file. We can achieve this using the newly introduced actions **SAVE VARIABLE** and **LOAD VARIABLE**. The buttons SAVEVAR and LOADVAR are added at the Form Level for the same.

```
[Button : SaveVar]
```

```
Key      : Alt + S
```

```
Action : Save Variable : SmpVar.pvf : EmpNameVar, EmpIDVar
```

The action SAVE VARIABLE will persist the values of the variables EmpNameVar and EmpIDVar in the file **SmpVar.pvf**

□ Action - LOAD VARIABLE

The action LOAD VARIABLE is used to reload the report scope variables from the specified file.

Syntax

```
LOAD VARIABLE : <FileName> [:<Variable List>]
```

Where,

<FileName> is the name of file in which the 'report' scope variables are persisted. The extension .PVF will be taken by default, if the file extension is not specified.

<Variable List> is the comma-separated list of variables that need to be loaded from the file. It is optional. In case it is not specified, all the variables saved in the file will be loaded.

Example:

In the previous example, we have persisted values of the Report Scope Variables **EmpNameVar** and **EmpIDVar** in the file **SmpVar.pvf**. Now, let us see how to re-load these 'report' scope variables from the file.

```
[Button : LoadVar]
```

```
Key      : Alt + L
```

```
Action : LOAD VARIABLE : SmpVar.pvf : EmpNameVar, EmpIDVar
```

The action **LOAD VARIABLE** will load the report scope variables **EmpNameVar** and **EmpIDVar** from the file **SmpVar.pvf**.



Member Variable Specification or Dotted Notation Specification is not allowed for specifying Variable list for both the actions SAVE VARIABLE and LOAD VARIABLE. It has to be a variable name identifier at the current report scope.

2.2 Variable Copy

The contents of a variable can now be entirely copied from one instance to another instance.

▣ Action - COPY VARIABLE

The action COPY VARIABLE is used to copy the content from one variable (Source) to another variable (Destination). This action is supported for all types of variables (Simple/Compound/List Variables).

Syntax

```
COPY VARIABLE : <Destination Variable> : <Source Variable>
```

Where,

<Destination Variable> is the name of the Simple/Compound/List Variable.

<Source Variable> is the name of the Simple/Compound/List Variable, from which the content has to be copied.

Example: Copying from Simple Variable to Simple Variable

```
[Function : SimpleVar Copy Function]
```

```
VARIABLE : SimpleVar1 : String : "Employee1"

VARIABLE : SimpleVar2 : String

10 : COPY VARIABLE : SimpleVar2 : SimpleVar1

20 : LOG : "Source" + ##SimpleVar1

30 : LOG : "Destination" + ##SimpleVar2
```

In this example, the variables **SimpleVar1** and **SimpleVar2** are declared at the Function level. After execution of the action **COPY VARIABLE**, the content of the variable is copied from **SimpleVar1** to **SimpleVar2**.

Example: Copying from Compound Variable to Compound Variable

Let us suppose that the following compound variables are defined:

```
[Variable : Employee1]
```

```
Variable : EmpName      : String : "Praveen"

Variable : Designation  : String : "Manager"
```

```
[Variable : Employee2]

Variable : EmpName      : String

Variable : Designation : String
```

In the function below, contents are copied from Compound Variable Employee1 to Employee2:

```
[Function : Compound Var Copy Function]

VARIABLE : Employee1

VARIABLE : Employee2

10 : COPY VARIABLE : Employee2 : Employee1

20 : LOG : "Source" + ## Employee1.EmpName

30 : LOG : "Source" + ## Employee1.Designation

40 : LOG : "Destination" + ## Employee2.EmpName

50 : LOG : "Destination" + ## Employee2.Designation
```



The content will be copied from a member variable of a Compound Variable (Source) to another member variable of a compound variable (Destination), based on the member variable names, since more than one member variable may have the same data type.

Example: Copying from List Variable to List Variable

Let us suppose that the following compound variables are defined:-

```
[Variable : Employee1]

Variable : EmpName      : String

Variable : Designation : String

[Variable : Employee2]

Variable : EmpName      : String

Variable : Designation : String
```

In the following function, the compound variables **Employee1** and **Employee2** are declared as **List Variable**. We are copying all the elements from the compound list variable Employee1 to the compound list variable **Employee2**.

```
[Function : ListVar Copy Function]

LIST VARIABLE : Employee1, Employee2

10 : LIST FILL : Employee1 : Employees : $Name : $Name
```



```

20 : LIST FILL : Employee1 : Employees : $Name : $Designation + :
    Designation
30 : COPY VARIABLE : Employee2 : Employee1
40 : LOG      : "Source Variable - Employee"
50 : FOR IN : KEY VAR : Employee1
60 : LOG      : $$LISTVALUE:Employee1 : ##KEYVAR : EmpName
70 : LOG      : $$LISTVALUE:Employee1 : ##KEYVAR : Designation
80 : END FOR
90 : LOG : "Destination Variable - Employee"
100 : FOR IN : KEY VAR : Employee2
110 : LOG      : $$LISTVALUE:Employee2 : ##KEYVAR:EmpName
120 : LOG      : $$LISTVALUE:Employee2 : ##KEYVAR:Designation
130 : END FOR

```

2.3 Scope Specification in Variable Dotted Syntax

The Dotted Notation Syntax for Variables (##) has now been enhanced to allow specification of scope / relative scope, etc.

Syntax

```

..      (DOUBLE DOT) denotes owner scope
...     (TRIPPLE DOT) denotes owner's owner scope and so on
().    denotes a system scope

```

Where,

<Definition Type> is the name of the definition such as Report, Function, etc., in the current execution chain.

<Definition Name Expression> can be any expression which evaluates to a Definition Name. The Definition Name Expression is optional.

(**<Definition Type>**, **<Definition Name Expression>**) can be used for absolute scope specification. The element (**<Definition Type>**, **<Definition Name Expression>**) has to be in the current execution chain, else one will not be able to refer to the same.

Example:

Let us suppose that the Variable **TSPLSMPScopeVar** is declared at System Scope.

```
[Variable : TSPLSMPScopeVar]
```

```
    Type : String
```

```
[System : Variable]
```

```
TSPLSMPScopeVar : "System Scope"
```

The function **TSPLSMP ScopeSpec** is called from a Menu. We have declared the variable **TSPLSMPScopeVar** in the 'Function' scope also.

```
[Function : TSPLSMP ScopeSpec]
VARIABLE      : TSPLSMPScopeVar
01 : SET      : TSPLSMPScopeVar : "Function Level"
02 : Display  : TSPLSMP ScopeSpec
```

The following report is displayed from the function **TSPLSMP ScopeSpec**. We have declared the variable **TSPLSMPScopeVar** in the 'Report' Level also.

```
[Report : TSPLSMP ScopeSpec]
Form        : TSPLSMP ScopeSpec
Variable    : TSPLSMPScopeVar
Set         : TSPLSMPScopeVar : "Report Level"
```

Following are the field definitions of the report **TSPLSMP ScopeSpec**. Let us see the variable values at the field level by specifying the scope in Variable Dotted Syntax.

```
[Field : TSPLSMP ScopeSpecCurrent]
Use        : Name Field
Set As    : ##TSPLSMPScopeVar
;Variable value in this field will be "Report Level" (Current Scope)
```

```
[Field : TSPLSMP ScopeSpecOwner]
Use        : Name Field
Set As    : ##..TSPLSMPScopeVar
Border    : Thin Left Right
```

;;Variable value in this field will be "Function Level" (Owner's Scope)

```
[Field : TSPLSMP ScopeSpecSystem]
Use        : Name Field
Set As    : ##().TSPLSMPScopeVar
Border    : Thin Left
```

;;Variable value in this field will be "System Level" (System Scope)

```
[Field : TSPLSMP ScopeSpecAbsolute]
```

Use : Name Field

Set as : ##(Function, "TSPLSMP ScopeSpec").TSPLSMPScopeVar

Border : Thin Left

;;Variable Value in this field will be "Function Level" (Absolute Specification)

2.4 Definition Name and Instance Name of Variable can be different now

A variable can be declared in a scope in two ways, i.e., either by specifying the name of the variable (in this case, a separate variable definition is required) or by specifying the name of the variable and a data type (in this case, a separate variable definition is not required; and hence, is called as inline declaration).



In this chapter, we will go through the 'Report' Scope variable declaration, syntax and examples. It is applicable for other scopes also.

Let us look into the variable declaration syntax of Report Scope.

Syntax

`[Report : <Report Name>]`

;;This syntax expects a separate variable definition in the same name

`Variable : <Variable Names>`

OR

;;Inline declaration

`Variable : <Variable Names> [:<Data Type>[:<Value>]]`

OR

`List Variable : <Variable Names> [:<Data Type>[:<Value>]]`

Example:

`[Report : SMP Report]`

`Variable : Emp Name`

`Variable : Emp Relation : String`

`List Variable : Employee1`

`List Variable : Employee2 : String : "Prem"`

`[Variable : Emp Name]`

`Type : String`

`[Variable : Employee1]`

`Variable : EmpName : String`

```
Variable : EmpID : String
```

Now, the 'Data Type' parameter can be pointing to a variable definition; in which case, it will allow one to have a variable which has the instance name and definition name different. This allows flexibility to create two instances of a compound structure in the same scope, with different instance names, without requiring to duplicate the definition. This capability is available at all the scopes where variable declaration is allowed.

Existing Syntax

```
[Report : <Report Name>]
  Variable : <Variable Names>
    OR
  Variable : <Variable Names> [:<Data Type>[:<Value>]]
    OR
  List Variable : <Variable Names> [:<Data Type>[:<Value>]]
```

New Enhanced Syntax

```
[Report : <Report Name>]
  Variable : <Variable Names>
    OR
  Variable : <Variable Names> [:<Data Type>[:<Value>]]
    OR
  List Variable : <Variable Names> [:<Data Type>[:<Value>]]
    OR
  Variable : <Instance Names> : [<Variable Name>]
    OR
  List Variable : <Instance Names> : [<Variable Name>]
```

Where,

<Instance Names> is the list of Simple/Compound/List Variables separated by comma (instance variables).

<Variable Name> is the Simple or Compound variable name. A separate variable definition is required. It should not be an inline variable.

Example: 1

Given here is the definition of a Compound Variable "Employee".

```
[Variable : Employee]
  Variable : EmpName      : String
  Variable : Designation : String
```

Now, we can create a variable instance using the definition of another variable. Let us understand this with the help of the following 'Report' definition:

```
[Report      : Employee Report]
```

;;An instance is declared with the name as 'Prem' and definition name as 'Employee'. The variable instance 'Prem' will inherit the entire structure of the variable definition 'Employee'.

```
Variable : Prem : Employee
```

;;An instance is declared with the name as 'Ramesh' and definition name as 'Employee'.

```
Variable : Ramesh : Employee
```

;;Locally, the instance "Ramesh" is modified to add a member variable.

```
Local      : Variable : Ramesh : Add : Variable : EmpID : String
```

;; Two instances are declared with the names "Kamal" and "Vimal", and the definition name as "Employee"

```
Variable : Kamal, Vimal: Employee
```

;; A List Variable instance is declared with the name "EmployeeList" and the definition name as "Employee"

```
List Variable : EmployeeList : Employee
```

Example: 2

```
[Report : TSPL SMP Variable Instance]
```

```
Variable : Employee      : String : "Suresh"
```

```
Variable : New Employee : Employee
```

In this example, we are trying to declare a variable instance 'New Employee', which is of the type of another variable 'Employee'. This will NOT work because the variable 'Employee' is declared as inline and an explicit Definition does not exist for the same.

Hence, inline variables cannot be used to declare another variable instance.

2.5 Use Case – Multiple Email Configurations

Scenario

ABC Company Ltd., a manufacturing company, is having the Head Office in Bangalore and branch offices in Delhi, Mumbai, Kolkata and Chennai. The company uses Tally.ERP 9 at all the locations.

The Head Office and Branch Offices are using the e-mail capability of Tally extensively to send remainder letters, outstanding statements, etc., to the customers.

The System Administrator at the Head office will be facilitating the Branch office staff for email configurations in Tally. The company is using its own mail server and also another mail server "SIFY". If there is a change in the mail server, the system admin needs to communicate the information to branch staff, and they will be updating the email configurations in Tally.ERP 9.

Now, the company wants to set the email configurations centrally for all the branches so that the branch staff need not struggle for email configurations, particularly when there is a change in the mail server. This solution provides the facility of saving multiple configurations in multiple file names, and later loading them from the file, based on user selection.

Requirement Statement

Presently in Tally.ERP 9, users need to set email configurations locally & update required details.

Now, the configurations can be created centrally and shared among the locations. Thus, the user need not set email configuration every time. They have to simply load the configuration from the file. This can be achieved using newly introduced actions SAVE VARIABLE & LOAD VARIABLE.

Functional Demo

Before looking into the design logic, we will have a functional demo.

Let us suppose that ABC Company Ltd. is using its own mail server and another mail server Sify in its Head Office and its branch offices.

Saving Email Configurations

Let us suppose that the System Administrator in Head Office wants to save the required email configurations in Tally.ERP 9 for HO and Branches

Gateway of Tally → F12 (Configure) → E-Mailing. The email configuration screen will appear as follows:

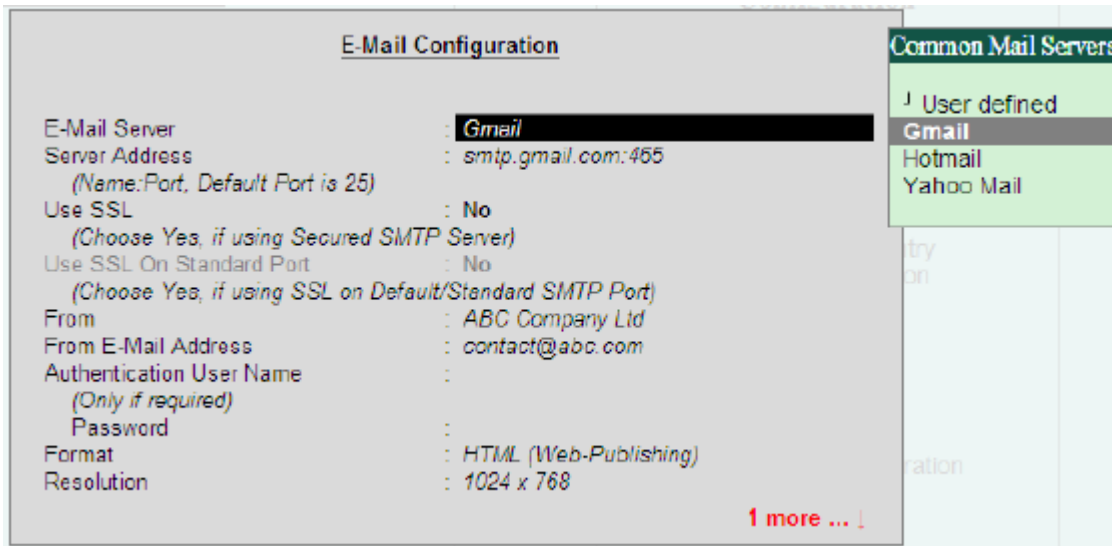


Figure 7. Email Configuration Screen

The System Admin needs to save the configurations for mail servers abc and Sify. Hence, he has to specify Email server as "User Defined" and enter the required configuration settings as follows:

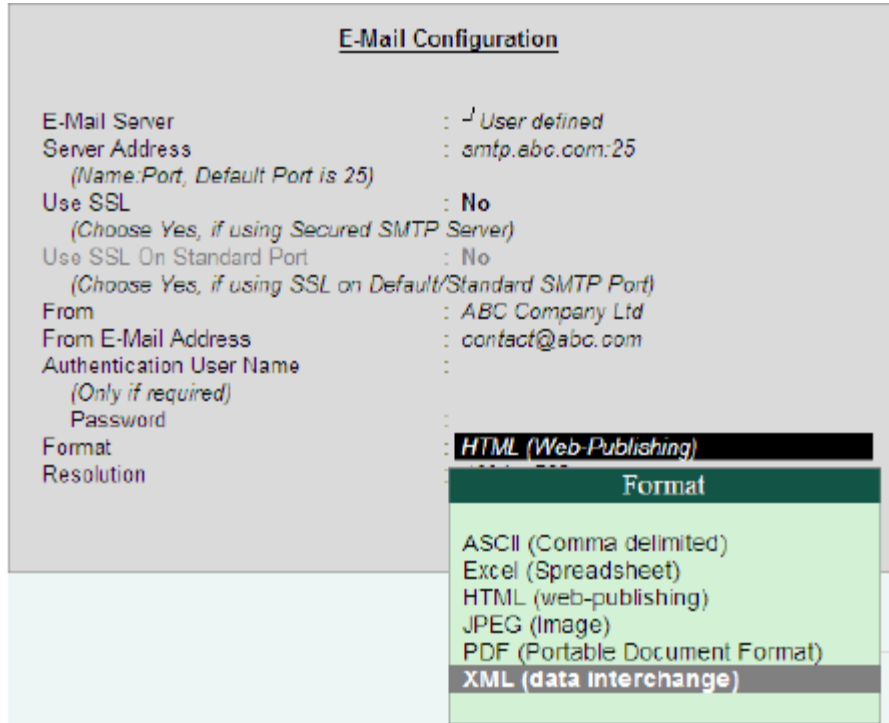


Figure 8. User Defined Configuration

Now, the System Admin has to press **Alt+S**, or click on the Button **Save Config**. The following screen will appear, where he has to enter the configuration file name:

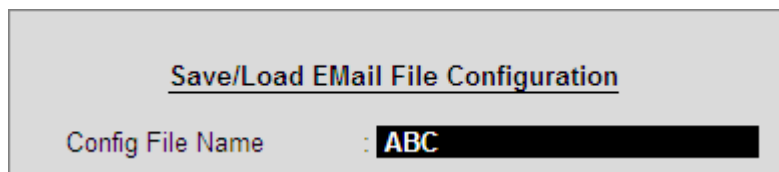


Figure 9. Save Configuration Screen

Once the System Admin accepts this screen, the configuration details will be saved in the file "abc.pvf". Similarly, he has to create the Configuration for the mail server "Sify". The files will be created in Tally.ERP 9 application folder, as shown in the following screenshot:

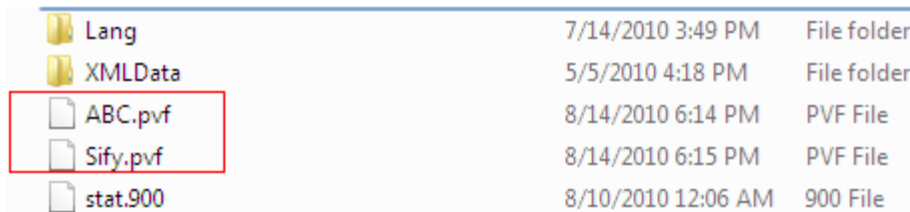


Figure 10. Files in Application folder

The admin can share these two files to the staff in HO and Branch Offices, and they should place the file in the respective Tally.ERP 9 Application folders.

Loading Configurations

Gateway of Tally → F12(Configure) → E-Mailing. The Email configuration screen will be displayed with the previously set configurations.

Now, the user at HO/Branch wants to load the configurations for the email server “abc”. He has to press **Alt+L** or click on the Button “Load Config”, and enter the file name, as shown in the figure:

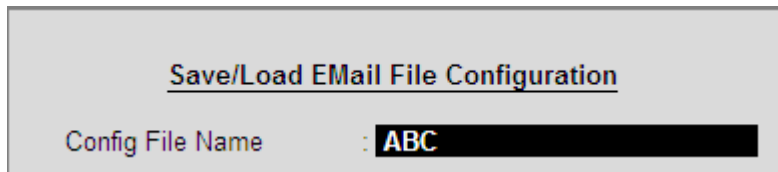


Figure 11. Load Configuration Screen

Accept the screen. The Email Configuration Report will display the configuration details loaded from the file “abc”. Accept the configuration screen, and the settings will be applicable to all reports. Suppose the User now wants to mail the report Balance Sheet. He has to select ‘Balance sheet’ and press **Alt + M**. The following configuration report will appear:

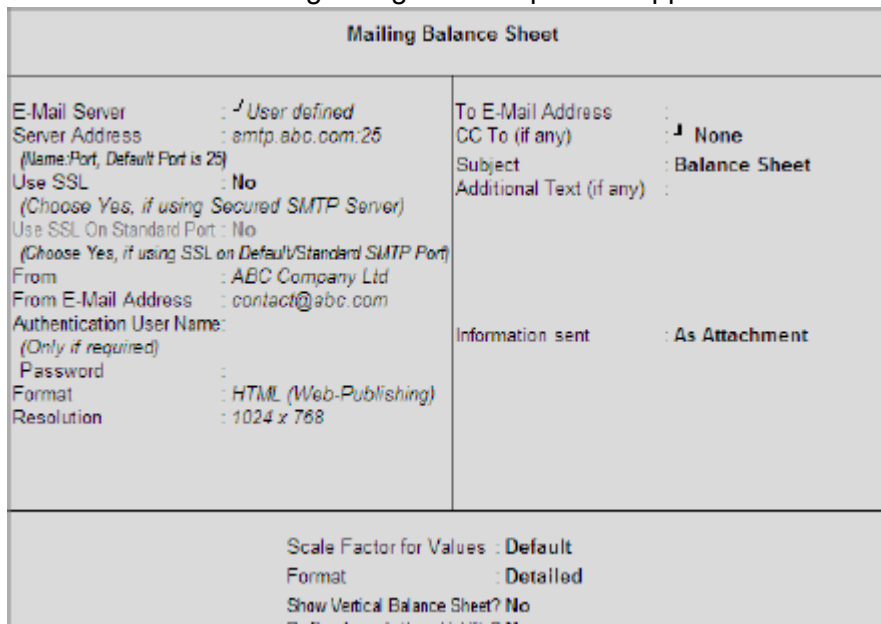


Figure 12. Email Configuration Screen

Note that the configuration details are changed as per the selected configuration.

Now, the user wants to change the email server as “Sify”. Go to **Gateway of Tally → F12(Configure) → E-Mailing**. Press **Alt + L**. Enter the file name as “Sify” and press **Enter**. The email configuration screen will have new configurations loaded from the file “Sify”.

Similarly, we can save/load multiple configurations.

Solution Development

The steps followed to achieve Saving of Multiple Email Configurations are:

1. Declaring variables at 'Report' Level

Variables SVMailServerName, SVMailServer, SVMailFormat, SVMailUseSsl, etc., are declared at 'Report' Level. All these variables have the attribute Persistent set as YES at the Definition level.

```
[#Report : EMail Configuration]

    Variable : SVMailServerName, SVMailServer, SVMailFormat, SVMailUseSsl
    Variable : SVMailUseSSLOnStdPort, SVMailAuthUserName, SVExportFormat
```

2. Saving Configuration

A Button is added to the Form, and the action will call a User Defined Function. In a User Defined Function, we are executing a report to accept a File Name from the user. We are persisting all the report scope variables in the specified file through the Action SAVE VARIABLE.

3. Loading Configurations

A Button is added to the Form and on clicking it, the action will call a User Defined Function. In the User Defined Function, we are executing a report to accept the File name from the user. We are reloading the report scope variables from the file through the Action LOAD VARIABLE. Please refer to the following code snippet for Save and Load configurations.

```
[Function : TSPL Smp SaveLoadVar]

    Parameter : IsSaveVar : Logical : Yes

    Variable : ConfigNameWithExt : String : Yes

    00 : EXECUTE : TSPL Smp SaveLoadConfig

;; Correcting the file name entered with or without extension by the user
    06 : IF: ##SaveLoadConfigName CONTAINS ".Pvf"

    10 :     SET : ConfigNameWithExt : ##SaveLoadConfigName

    20 : ELSE :

    30 :     SET : ConfigNameWithExt : ##SaveLoadConfigName + ".pvf"

    40 : ENDIF

;; Saving or Loading the variables based on parameter value
    50 : IF : NOT $$IsEmpty : ##SaveLoadConfigName

    60 :     IF : ##IsSaveVar

    70 :         SAVE VARIABLE : ##ConfigNameWithExt

    80 :     ELSE :

    90 :         LOAD VARIABLE : ##ConfigNameWithExt
```

```
100 : ENDIF
```

```
110 : ENDIF
```

The corresponding field values need to reflect the values of the variables loaded from the file. This is handled by using the following code:

```
Local : Field : DSPMailServer : Set as : +
    If #DSPMailServerName Contains $$SysName:UserDefined +
    Then ##SVMailServer Else + If #DSPMailServerName
    NOT Contains $$SysName:UserDefined + Then
    $$GetMailServerAddr : #DSPMailServerName +
    Else ##SVMailServer
Local : Field : DSPMailServerName      : Set As : ##SVMailServerName
Local : Field : DSPMailFormat          : Set As : ##SVMailFormat
Local : Field : DSPMailUseSsl          : Set As : ##SVMailUseSsl
Local : Field : DSPMailUseSSLOnStdPort : Set As : ##SVMailUseSSLOnStdPort
Local : Field : DSPMailAuthUserName    : Set As : ##SVMailAuthUserName
Local : Field : DSPFinalExportFormat   : Set As : ##SVExportFormat
```

Also, if the field values are changed, the Report level variables need to be modified with those values. This is handled using the following code:

```
Local : Field : DSP MailServerName : Modifies : SVMailServerName: Yes Local:
Field : DSP MailServer : Modifies : SVMailServer : Yes
Local : Field : DSP MailFormat : Modifies : SVMailFormat : Yes
Local : Field : DSP MailUseSsl : Modifies : SVMailUseSsl : Yes
Local : Field : DSP MailUseSSLOnStdPort : Modifies : +
    SVMailUseSSLOnStdPort : Yes
Local : Field : DSP MailAuthUserName : Modifies : + SVMailAuthUserName : Yes
Local : Field : DSP FinalExportFormat : Modifies : + SVExportFormat : Yes
```

On Accepting of the Form EMail Configuration, we are calling a User Defined Function to set the System Variable values. Thus, the changed configuration details will be available for all the reports. Please refer to the following Code Snippet:

```
[Function : TSPL Smp Update System Variables]
```

```
10 : SET : ().SVMailServerName      : ##SVMailServerName
20 : SET : ().SVMailServer          : ##SVMailServer
30 : SET : ().SVMailFormat          : ##SVMailFormat
40 : SET : ().SVMailUseSsl          : ##SVMailUseSsl
50 : SET : ().SVMailUseSSLOnStdPort : ##SVMailUseSSLOnStdPort
60 : SET : ().SVMailAuthUserName    : ##SVMailAuthUserName
70 : SET : ().SVExportFormat        : ##SVExportFormat
```

3. Event Framework Enhancements

This is a path-breaking enhancement in Tally which will enable scheduled execution of any Action. This has been supported with the introduction of a System Event called Timer. We can have a set of timer events of specified durations and trigger an Action on the same. For example, if we require Synchronization to be triggered at every one hour, we can define a Timer event which triggers the action 'Sync'. Actions for Starting and Stopping the timer have been provided.

3.1 Timer Event

As we are already aware, Events like **System Start**, **System End**, **Load Company**, **Close Company**, **On Form Accept**, etc., introduced earlier as a part of the Event Framework, require user intervention. Automated events which can be used to take timely backups, display automated messages, etc., were not possible earlier. With the breakthrough introduction of **Timer Event**, performing Timer based automated events is possible now. System Event **Timer** has been introduced to perform the required set of operations automatically at periodic intervals.

Syntax

```
[System : Event]
    <Timer Name> : TIMER : <Condition> : <Action> : <Action Parameters>
```

Where,

<Timer Name> is the user defined name for the timer event.

<TIMER> keyword indicates that it is a Time-based event.

<ConditionExpr> should return a logical value.

<ActionKeyword> is any one of the actions.

<Action Parameters> is nothing but the parameters of the action specified.

We can have multiple Event Handlers with unique names which can invoke specific Actions at different intervals. In order to specify the interval for the various Timers and to begin and end the Timers, the associated Actions introduced are 'Start Timer' and 'Stop Timer'.

Actions - Start Timer and Stop Timer

□ Action - START TIMER

It starts the specified timer and accepts the Timer Name and Duration in seconds as the action parameters.

Syntax

```
START TIMER : <Timer Name> : Duration in seconds
```

Where,

<Timer Name> is the user defined name for the timer event.

□ Action - STOP TIMER

This Action stops the specified timer, and it accepts the Timer Name as its parameter.

Syntax

```
STOP TIMER : <Timer Name>
```

Where,

<Timer Name> is the user defined name for the timer event.

Following is an example of scheduling automatic backups every hour:

Example:

```
[System : Event]
```

```
;; Setting up timer event to call a function
```

```
Auto Backup : TIMER : TRUE : CALL : Take Backup Function
```

```
;; Starting the Timer when Tally Application Starts
```

```
Schedule Backup : System Start : TRUE : START TIMER : Auto Backup : 3600
```

```
;; Adding Keys to 'Company Info' Menu
```

```
[#Menu : Company Info.]
```

```
Add : Keys : Stop Backup Timer
```

```
;; Declaring a Key to Stop the Timer
```

```
[Key : Stop Backup Timer]
```

```
Key : Alt + S
```

```
Action : Stop Timer : Auto Backup
```

```
Title : "Stop Backup"
```

In this example, following is done:

- **Auto Backup**, a **Timer Event** is declared under **System Event** to invoke the Function
- **Take Backup Function** at periodic intervals, as specified within the Action **Start Timer**.

- **Schedule Backup**, a **System Start** event is declared under **System Event** to **Start** the above **Timer 'Auto Backup'** and execute the specified action every 3600 Seconds, i.e., every hour.
- A corresponding Key to **Stop** the **Timer** is associated to Menu **Company Info**, which is defined to **Stop** the **Timer**. User can stop the timer if he chooses not to continue taking automatic backups any further.

Timer Events can be very useful in many cases like displaying Exception Reports, Negative Balances intimation, Inventory Status below Minimum or Reorder Level, Outstanding Reminders, Auto Sync at regular intervals, and many more.

4. Action Enhancements

New actions have been introduced in this release, viz. Refresh Data, Copy File and Sleep.

4.1 Action - Refresh Data

In Tally, whenever any report is being viewed, it contains the most recent updates till the last entry. If any report is left open and subsequently viewed later, possibly few more entries would have gone in the system entered by various other users on the Network. Hence, the report which is currently being viewed is older. To view the updated report, the user has to exit the report and once again, enter the Report. To solve this problem, a new action 'Refresh Data' has been introduced, which refreshes the data in memory automatically, as and when required.

Syntax

REFRESH DATA

Refresh Data can be used along with **Timer Event** and every few seconds, the Report can be refreshed automatically to display the updated information.

Example:

```
[System : Event]

  Refresh Timer : TIMER : TRUE : Refresh Data

[#Form : Balance Sheet]

  Add : Keys : Enable Refresh

[Key : Enable Refresh]

  Key : Alt + R

  Action : Start Timer : Refresh Timer : 300
```

In this example, **Refresh Timer**, a **Timer Event** is declared under **System Event** to invoke the Action **Refresh Data** at periodic intervals. A key **Enable Refresh** is added in the Balance Sheet Report, which will be used to Start the Timer **Refresh Timer** every 5 minutes.



The Action 'Refresh Data' is a Company Report - Specific Action. It will always require a Report in memory to Refresh the Data.

4.2 Action - SLEEP

Action SLEEP has been introduced to specify time delays during the execution of the code. For few seconds, the system will be dormant or in suspended mode.

Syntax

SLEEP : <Duration in Seconds>

<Sleep> is the action which halts the functioning of the Application for a few seconds as specified in **<Duration>**.

Example:

```
[#Menu : Gateway of Tally]

  Add : Item : Trial Balance after 10 secs : CALL : TBAfterSleep

[Function : TBAfterSleep]

  00 : SLEEP      : 10

  10 : DISPLAY : Trial Balance
```

In this example, the system will halt for 10 seconds and display the Trial Balance subsequently.

4.3 Action - Copy File

A new Action 'Copy File' has been introduced, which allows:

- Copying from one location to another within the same System
- Uploading of Files from a given Path to a FTP Site
- Downloading of File from FTP Site to the specified location/folder

Syntax

Copy File : <Destination File Path> : <Source File Path>

Where,

<Destination File Path> can be any expression evaluating to a valid local/FTP path.

<Source File Path> can be any expression evaluating to a valid local/FTP path.

Example:

```
CopyFile : ##MyDstFilePath : ##MySourceFilePath
```

If any of the File path is an FTP path, the same can be constructed using functions like `$$MakeFTPName`. It accepts various parameters like servername, username, password, etc. The following code snippet sets the value of the variable **MyDstFilePath** using the function

\$\$MakeFTPName.

```
SET : MyDstFilePath : $$MakeFTPName : ##SVFTPServer : ##SVFTPUser : +  
    ##SVFTPPassword : @SCFilePathName
```

The function \$\$MakeFTPName uses the various parameters which are System Variables captured from the default configuration reports.

5. TDL Enhancements for Remoting

There have been various enhancements at the TDL level to enable Remote Edit Capability in the product. The enhancements are as follows:

□ 'Fetch Object' Attribute Changes

The attribute 'Fetch Object' has been supported at Report, Form, Field and the Function level as well. The Object Name specification in the syntax allows expressions now. It is also possible to specify multiple Object Names separated by the Fetch Separator Character. A new function \$\$FetchSeparator has been introduced to return this character.

□ 'Fetch Values' Attribute Introduced

The evaluation of External Methods of an Object requires Object Context to be available at the Client End. A new Attribute 'Fetch Values' has been provided at the 'Report' level to specify the list of External Methods.

□ 'Multi Objects' Attribute Introduced

Whenever multiple Objects of the same collection are getting modified at the Client's End, a new attribute called 'MultiObjects' is introduced at the Report Level to enable the same.

□ 'Modifies' Attribute Changes

The 'Modifies' attribute of the field has been changed to accept a third parameter (optional) which is an expression. This allows the variable to be modified with the value of the expression rather than the field value.

□ Collection Attribute 'Parm Var' Introduced

As we already know, the 'Collection' Artefact evaluates the various attributes either during initialization or at the time of gathering the collection. It may require various inputs from the Requestor context for the same.

The direct reference of values/expressions from the report elements and objects in the collection at various points creates various issues like code complexity, performance lapses and non availability of these values on Server in Remote Environment.

In order to overcome these, a new Collection attribute 'Parm Var' has been introduced. 'Parm Var' in collection is a context-free structure available within the collection. The requestors Object context is available for the evaluation of its value. This is evaluated only once in the context of the caller/requestor. This happens at collection initialization, and the expression provided to it is evaluated and stored as a variable, which can be referred within any of the attributes of the collection anytime, and is made available at the Server end as well.

Lets understand each of these in detail.

5.1 'Fetch Object' Attribute Changes

When multiple methods of a Single Object/Multiple Objects of the same type are required, then that Object can be fetched at Report, Form, Field and Function levels.

Report Level

'Fetch Object' attribute has been enhanced at 'Report' level to take an expression instead of a variable name that evaluates to the name of an object.

The existing syntax of the 'Fetch Object' attribute at report level is as follows:

Syntax: Prior to 2.0

```
Fetch Object : <Object Type> : <Variable Identifier > : <List of methods>
```

Example:

```
Fetch Object: Ledger: LedgerName: Name, Parent, ClosingBalance
```

Syntax: 2.0 Onwards

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
Fetch Object: Ledger: ##LedgerName: Name, Parent, ClosingBalance
```

Here, since the Object name is an expression, we need to prefix the variable name with ##.

Form Level

Attribute 'Fetch Object' has been introduced at 'Form' Level. In scenarios where multiple forms are available at a report; for each form, we require to fetch methods pertaining to different objects.

Syntax

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[!Form : AccoutingViewVoucher]
```

```
Switch : AccVoucherView : NormalAccoutingViewVoucher : +
```

```
    NOT$$IsAttendance : ##SVVoucherType
```

```
Switch : AccVoucherView : AttdAccoutingViewVoucher : +
```

```
    $$IsAttendance : ##SVVoucherType
```

```
[!Form : AttdAccoutingViewVoucher]
```

```
Fetch Object : AttendanceType : @@AttdEntryList : +
```

```
    AttendanceProductionType, AttendancePeriod, BaseUnits
```

```
[!Form : NormalAccoutingViewVoucher]
```

```
FetchObject : Ledger : @@AllLedEntryList : Name, Parent, ReserveName
```


Field Level

There may be scenarios where we may need to Fetch Object values dynamically based on the current field values. For example, the field may be associated with a Table of ledgers. Based on the ledger selected, the corresponding methods of the Object require to be fetched. In such cases, this attribute will be useful.

Syntax

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[Field : LED VAT Class]
```

```
Fetch Object : TaxClassification : $$Value : FirstAlias, RateofVAT, + TaxType
```

Function Level

There may be scenarios where the method values need to be fetched based on the Object name passed as a parameter to the function. In such cases, 'Fetch Object' at the function level is required.

If we have already fetched the object methods at the 'Report' or 'Field' level, the same will be propagated to the called function. In case it is not fetched earlier, the same can be fetched at the 'Function' level as well. This enables dynamic fetching of Objects.

Syntax

```
Fetch Object : <Object Type> : <Expression> : <List of methods>
```

Example:

```
[Function : FillUsingTrackingObj]
```

```
Parameter      : pTrackKey : String
```

```
Fetch Object : Tracking Number : ##pTrackKey : *.*
```

In case the same set of methods for multiple objects needs to be fetched, the multiple Object Names need to be specified in the syntax of 'Fetch Object', separated by the Fetch separator character.

▣ Function – \$\$FetchSeparator

This function returns C_FETCH_SEPARATOR character that is used for separating multiple object names in FETCH OBJECT attribute. There may be scenarios where the same set of methods needs to be fetched from multiple objects. In that case, it is possible to specify multiple object names in the 'Fetch Object' syntax, separated by the character which is returned from the function **\$\$FetchSeparator**.

Example:

```
Fetch Object : Ledger : "Debtor North" + $$FetchSeparator + "Debtor South" :  
Name, Parent, ClosingBalance
```

5.2 'Fetch Values' Attribute Introduced

This is a report level attribute which allows computation of values for user defined (external) methods, based on the current Object context available.

Syntax

```
Fetch Values : <List of methods>
```

Example:

```
[Report : VAT Classification]

Object : Tax Classification

Fetch Values : MasterID, CanDelete
```

5.3 'Multi Objects' Attribute Introduced

This is a 'Report' level attribute which is required to be specified, in case Multiple Objects of the same collection are being added/modified in a Report. It is required specifically in case of multi master creation or alteration.

Syntax

```
MultiObjects : <Edit Collection>
```

Where,

<Edit Collection> is the name of the Collection for which modifications are to be done.

Example:

```
[Report : Multi Ledger]

Multi Objects : Ledger Under MGroup
```

5.4 'Modifies' Attribute Changes

This is a field level attribute enhanced further to take a third optional parameter. Prior to Tally.ERP 9 Release 2.0, if a field had 'Modifies' parameter, the field value would be set to the variable. And based on this variable value, some calculations or concatenation was required to be performed. An invisible field was required for the same. With the enhancement in this Release, one can modify the variable value at the same field itself using an expression, i.e., the field, and the variable may have different values.

Syntax: Prior 2.0

```
Modifies : <Variable Name> : <Logical Value>
```

Where,

<Variable Name> is the name of the variable.

<Logical Value> is any expression which evaluates to a logical value.

Example:

```
[Field : BatchesInGodown]

Modifies : DSPGodownName : Yes
```

Syntax: 2.0 onwards

Modifies : <Variable Name> : <Logical Value> : <expression>

Where,

<Variable Name> is the name of the variable.

<Logical Value> is any expression which evaluates to a logical value.

<Expression> can be used to modify the variable value within the field.

Example:

```
[Field : BatchesInGodown]
```

```
Modifies : DSPGodownName : Yes : ##DSPGodownName + " - Godown"
```

Here, considering that the field value is 'Main location', output would be **Main location - Godown**.

5.5 Collection Attribute 'Parm Var' Introduced

As we already know, the 'Collection' Artefact evaluates the various attributes either during initialization or at the time of gathering the collection. It may require various inputs from the Requestor context for the same. For example, the evaluation of 'Child of' and 'Filter' attributes happens at the time of gathering the collection. It requires certain values from Requestors context like \$name. In 'Filter' attribute, if \$name of each object is to be compared with \$name of the Requestors context, then we have to refer it as \$ReqObject:\$name. The direct reference of values/expressions from the report elements and objects in the collection at various points, creates a few issues as follows:

- ❑ Code complexity is increased, as observed in the Filter example above.
- ❑ The performance is impacted, as there is are repeated references in case of Filters.
- ❑ In a Remote Environment; where the Requestor Context is not available within the collection at the Server side

In order to overcome the above, a new Collection attribute 'Parm Var' has been introduced.

We already have the capability of declaring inline variables at collection level using the Attributes Source Var, Compute Var and Filter Var. These are context free structures available within the collection for various evaluations. For storing values in these, the various object contexts available are Source Objects, Target Objects, etc. One more attribute called 'Parm Var' has been introduced in collection, which is a context-free structure available within the collection. The request-ors Object context is available for evaluation of its value. This is evaluated only once in the context of caller/requestor. This happens at collection initialization and the expression provided to it is evaluated and stored as a variable which can be referred within any of the attributes of the collection anytime, and is made available at the Server end by passing it with the XML Request.

- ❑ **Collection Attribute - Parm Var**

The attribute ParmVar evaluates the value of the variable based on the requestor object's context.

Syntax

Parm Var : <Variable Name> : <Data Type> : <Formula>

Where,

<Variable Name> is the name of variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression which evaluates to the value of 'Variable' data type.

Example:

```
[Part : Groups and Ledgers]

    Lines   : Groups and Ledgers

    Repeat  : Groups and Ledgers : List of Groups

    Scroll  : Vertical

[Line : Groups and Ledgers]

    Fields      : GAL Particulars

    Right Fields : GAL ClosBal

    Explode     : List of Ledgers : ##ExplodeFlag

[Part : List of Ledgers]

    Lines   : List of Ledgers

    Repeat  : List of Ledgers : Smp List of Ledgers

[Collection : Smp List of Ledgers]

    Type     : Ledger

    Child Of : $Name
```

In the collection **Smp List of Ledgers**, the **Child of** attribute is evaluated based on the method **\$Name** which is available from the Group Object in context. The line **Groups and Ledgers (Requestor Object)** is associated with a Group Object.

In a Remote environment, when such a Report is being displayed at the Clients end, the request for the collection gathering goes to the Server End. At the server end, the Requestor Context is not available. So, the evaluation of \$Name will fail. To overcome such a situation, a new attribute called "Parm Var" has been introduced, which is a context-free structure available within the collection. It evaluates the expression based on the Requestors Context, thereby available at the Server Side also.

The Collection is Redefined as follows using the attribute **ParmVar**

```
[Collection : Smp List of Ledgers]

    Type : Ledger

    Child Of : ##ParmLedName
```

```
Parm Var : ParmLedName : String : $Name
```

The value of variable “**ParmLedName**” is evaluated at the Client side based on method \$name available from Group Object Context, and sent to the Server. While gathering the objects at the server side, the attribute ‘ChildOf’ is evaluated, which uses the variable **ParmLedName** instead of \$Name, available at the Server.

6. Default TDL Changes

In the release 2.0, many new features like Remote Edit, SMS support, etc., have been introduced. The TDL language is also enriched with new capabilities to support these features. Using the new language capabilities, the source code of Tally.ERP 9 Release 2.0 has also been enhanced. The changes have been made in many definitions. For example, the values of some of the attributes have been changed, new attributes have been added and formulas have been rewritten.

Although, it has been tried to ensure maximum backward compatibility, there may be cases where the application developer may require to validate/rewrite the existing TDL codes to make them compatible with Tally.ERP 9 Release 2.0. In this section, the changes have been summarised in terms of listing the definitions. Although sincere efforts have been made in the direction of providing a comprehensive listing of definitions, one may come across a few cases where changes have been made. If any of these definitions are being used in customisations, one must refer to the source code changes available with the latest Release of TDE.

6.1 Mandatory ‘Fetch’ at the Collection Level

This release onwards, **Fetch** is mandatory in every collection. All the methods which are required to be used in a Report are to be fetched at the Collection level.

6.2 Voucher Creation

Whenever a new Voucher is being created, it is important to take care of the following:

- The variable name “SVViewName” has to be set to System Names
 - AcctgVchView – For all Accounting vouchers
 - InvVchView – For all Inventory vouchers, except Stock Journal voucher
 - PaySlipVchView – For Payroll vouchers
 - ConsVchView – For Stock Journal voucher
- The method ‘PersistedView’ has to be set to the value of the variable ‘SVViewName’.

Example:

```
ds : Set : SVViewName : $$SysName : AcctgVchView

10 : NEW OBJECT : Voucher

    |

    |

Aa : Set Value : PersistedView : ##SVViewName

30 : CREATE TARGET
```

6.3 Extract Collections List and Usage as Tables

Many existing 'collection' definitions have been converted as Extract Collections. So, if these collections are used in any of the user TDLs, the code needs to be rewritten for Tally.ERP 9 Release 2.0. Many fields which were using the old collections as Tables have been modified to use the Extract Collections now. The 'Table' Attribute has been changed for those fields.

The following Table shows the fields in which extract collections are used in the 'Table' attribute:

Field Name	Table Name OLD	Extract Collection/Table Name
EI AccAllocName	Inv SalesLedgersAlloc Inv Purch Ledgers Inv Sales Income Ledgers Inv Purch Expense Ledgers NonInv Purch Support Ledgers NonInv Sales Support Ledgers	Inv SalesLedgersAlloc Extract Inv Purch Ledgers VchExtract Inv Sales Income Ledgers Extract Inv Purch Expense Ledgers Extract NonInv Purch Support Ledgers - VchExtract NonInv Sales Support Ledgers - VchExtract
EI Consignee	Party Ledgers, Cash Ledgers Invoice Ledgers	Party Cash Ledgers Extract Invoice Ledgers Extract
EICommonLED	Inv Sales Ledgers Inv Purch Ledgers Inv Sales Income Ledgers Inv Purch Expense Ledgers	Inv Sales Ledgers Extract Inv Purch Ledgers Extract Inv Sales Income Ledgers Extract Inv Purch Expense Ledgers Extract
VCH VATClass	VCH VAT Sales ClassificationVCH VCH VAT Purc ClassificationVCH	VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract
VCH AccAllocVAT Class	VCH VAT Sales ClassificationVCH VCH VAT Purc ClassificationVCH	VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract
VCH POS PartyContact	Party Ledgers	Party Ledgers Extract
VCHACC StockItem	Vch Stock Item	Vch Stock Item Extract
VCHJRNLSockItem	Vch Stock Item	Vch Stock Item Extract
ACGLLed	GainLoss Ledgers	GainLoss Ledgers Extract
ACLSFixedLed	Cash Class Ledgers	Cash Class Ledgers Extract

ACLSLed	Cash Ledgers Normal Ledgers Normal Ledgers, Cash Ledgers Non CENVAT Ledgers Non CENVAT Ledgers, Cash Ledgers,	Cash Ledgers VchExtract Normal Ledgers Extract Normal Cash Ledgers Extract Non CENVAT Ledgers Extract Non CENVAT Cash Ledgers Extract
EI AccDesc	Sales Support Ledgers Purchase Support Ledgers	Sales Support Ledgers VchExtract Purchase Support Ledgers VchExtract
VCH AccVATClass	SD Sales Classification Etc...	SD Sales Classification Extract Etc ...
VCHIndentStockItem	Vch Stock Item	Vch Stock Item Extract
VCHBATCH Godown	Stockable Godown JOB Stockable Godown	Stockable Godown VchExtract JOB Stockable Godown VchExtract
VCHBATCH OrdName	Active Batches	Active Batches VchExtract
VCHBATCH NrmlName	Active Batches	Active Batches VchExtract
VCHBATCH JrnlName	Active Batches	Active Batches VchExtract
EI VATClass	SD Sales Classification VCH VAT Sales Classifica- tionVCH VCH VAT Purc ClassificationVCH VAT Sales With Rate ClassificationVCH VAT Purc With Rate ClassificationVCH Addl VAT Sales With Rate ClassificationVCH Addl VAT Purc With Rate ClassificationVCH CessOn VAT Sales With Rate ClassificationVCH CessOn VAT Purc With Rate ClassificationVCH CST Sales With Rate Classification CST Purc With Rate Classification	SD Sales Classification Extract VCH VAT Sales ClassificationVCH Extract VCH VAT Purc ClassificationVCH Extract VAT Sales With Rate ClassificationVCH Extract VAT Purc With Rate ClassificationVCH Extract Addl VAT Sales With Rate ClassificationVCH Extract Addl VAT Purc With Rate ClassificationVCH Extract CessOn VAT Sales With Rate ClassificationVCH Extract CessOn VAT Purc With Rate ClassificationVCH Extract CST Sales With Rate Classification Extract CST Purc With Rate Classification Extract
VCHBATCH GRNName	Active Batches	Active Batches VchExtract
POS BatchName	Active Batches	Active Batches VchExtract
VCHBATCH DealerGodown	Stockable DealerGodown	Stockable DealerGodown VchExtract
VCHBATCH ExciseMfgrGodown	Stockable ExciseMfgrGodown	Stockable ExciseMfgrGodown VchExtract

VCHBILL TDSLedger	TDS Ledger Table	TDS Ledger Table VchExtract
VCHBILL STaxLedger	Service Tax Ledger Table	Service Tax Ledger Table VchExtract
VCHCSTCAT Name	Voucher Cost Category	Voucher Cost Category VchExtract
VCHCST Name	Cost Centre All Cost Centre	Cost Centre VchExtract All Cost Centre VchExtract
STKVCH Ledger	Party Ledgers, Cash Ledgers	Party Cash Ledgers Extract
PF CashBank Ledger	Cash Ledgers	Cash Ledgers VchExtract
VCH AttdEmpName	Payroll DeactivationEmployees	Payroll DeactivationEmployeesExtract
VCH AttdType	List of Attendance Types	List of AttdTypesExtract
VCH AutoAttdType	List of Attendance Types	List of AttdTypesExtract
VCH EmpName	Payroll Cost Centres Manual Vch Employees Under Category	Payroll CostCentres AsVCHExtract Manual AsVchEmployees Under CategoryExtract
PAYROLLFixedLed	Payroll Liab Ledgers	Payroll Liability LedgersExtract
Payroll VCH Emp- Cat-Particulars	List of CostCategories	List of CostCategories Extract
Payroll VCH Emp- Particulars	Payroll Cost Centres Manual Vch Employees Under Category	Payroll CostCentres VCHExtract Manual Vch Employees Under CategoryExtract
PayrollVCHPayhead Name	Vch Pay Heads	Vch Pay Heads Extract
Payroll FunctionAuto-CategoryName	Payroll Vch Categories	Payroll Vch Categories Extract
Payroll FunctionAuto-toCostTables	Payroll Cost Centres AutoVch Employees Under Category	AutoVch PyrlCostCentres VCHExtract AutoVch Employees Under CategoryExtract
Payroll FunctionAuto-PayheadName	Payroll Ledgers	AutoVch PayrollLedgersExtract
TDSAUTOLedger	Normal Ledgers	Normal Ledgers Extract
TDSFilter Bank	Cash Class Ledgers	Cash Class Ledgers Extract
EI TrackOrder	InvSalesOrders InvPurcOrders	InvSalesOrders, Not Applicable, EndOfList, NewNumber InvPurcOrders, Not Applicable, EndOfList, NewNumber

EI SalesOrder	InvSalesOrders InvPurcOrders	InvSalesOrders, Not Applicable, EndOfList, NewNumber InvPurcOrders, Not Applicable, End- OfList, NewNumber
SRVTPartyName	Service Party Ledgers	Service Party Ledgers Extract
SRVTPartyBillName	Pending Party Bills	Pending Party Bills Extract

6.4 Modified Definition List and corresponding Changes

Changes in 'Set As'

Definition Type	Definition Name
Part	VCH Excise SubCat Tax Rate
	Trader PurcTypeofDuty
	VCH Excise SubCat Tax Rate
Field	VCH Excise SubCat Tax Rate
	Trader PurcTypeofDuty
	TDS TaxPartyLedger
	VCH TaxPymtDetails
	Trader PLARG23SINo
	Trader SupplierRG23No
	Trader OriginalRefNo
	Trader MfgrImprName
	Trader DN SupplierInvNo
	Trader DN SupplierInvDate
	Trader DN NatureofPurc
	Trader DN QtyPurchased
	Trader DN QtyReturn
	Trader DN AssessableValue
	Trader CN SupplierInvNo
	Trader CN SalesInvDate
	Trader CN SalesInvNo
	Trader CN QtySold
	Trader CN QtyReturn
	Trader CN SplAEDOfCVDNotPassOn
	DealerInv AmtofDuty
	DealerInv DutyPerUnit

Options Added–In Alter Mode

In the definition **[Form: Voucher]**, the option for ‘Alter’ mode is added and it is used to list all the fetches.

Definition Type	Definition Name
Form	Voucher

‘Fetch Object’ Added:

Definition Type	Definition Name
Field	VCH StockItem

‘Compute Var’ and ‘Fetch’ Attributes Added

Definition Type	Definition Name
Collection	Vouchers of FBT Category Calc
	Memo Vouchers of FBT Category Calc
	FBTCategoryCalc
	Vouchers of Regular FBT Category Calc
	Vouchers of Recovered FBT Category Calc
	VCHInTNo
	VCHInTNoG
	VCHInTNoB
	VCHInTNoBG
	VCHOutTNo
	VCH OutTNoG
	VCH OutTNoB
	VCH OutTNoBG
	TaxBill Details
	PayFunctionCaterotyCollection
	PayFunctionEmployeeCollection
	AllStatLedgersSlabSummary
	AllPFStatLedgers
	Admin AutoFil JrnlEmployees
	AutoFil PF Ledgers
	Admin AutoFil Employees
	AdminAutoPayableColl
	AdminAutoPayableColl PayrollSrc

	AdminAutoPayableCollJrnl
	AdminAutoPayableColl JrnlSrc
	PFESI EmployeeFilter Summary
	PFESI EmployeeFilter Vouchers
	Excise RG23DNoColl
	Trader ListOfPurcCleared
	Trader ListOfPurcNonCleared
	Trader ListOfMultiPurcCleared
	Trader ListOfMultiPurcNonCleared
	ExciseDealer Inventory Entries
	ExciseDealerInvoice InventoryEntries
	TDS DeductSameVoucher
	TDS TaxObjPartyBills
	TDS ITIgnoreNOP
	TDSDuty LedTable
	TaxObj AgstTableDebitNote
	TaxObj AgstTable
	SRCTaxObj AgstTable
	TDS CommonPartyLedger
	TaxObj AdvAgstTable
	TaxObj DedTable
	Pending TCS Bills
	PndgTaxBillsTillCurVchDate
	TaxBillColl
	TCS Vouchers of Party
	Pending Tax Bills
	BankColl
	InvSalesOrders
	ExciseInvSalesOrders
	InvPurcOrders
	InvOutTrackNumbers
	InvInTrackNumbers
	Pending Sales Orders
	VCHSo
	Pending Purc Orders
	VCHPo

	VCH OutTNo Src
	InPending Tracking Numbers
	OutPending Tracking Numbers
	Pending Bills
	STX SalePending TaxObj
	Pending STaxParty Bills
	STX CrDrNotePending TaxObj
	STX CategorywisePending TaxObj
	STX RcptPending TaxObj
	STXSource
	STX SalePending TaxObj
	STX JV SalePending TaxObj
	STXTaxObjOutput GAR7PymtAlloc
	STXTaxObjInput GAR7PymtAlloc

Few Attributes added and 'DebugExec' Action Used

Definition Type	Definition Name
Function	ESIDeductionPayFunction
	ESIEligibilityPayFunction
	ESIContributionPayFunction
	PFNormalPayFunction
	PTNormalPayFunction
	PTMonthlyPayFunction
	FirstEPF12PayHeadAbsVal Function
	FirstEPF833PayHeadAbsVal Function
	PFNormalVchPayFunction
	PTMonthlyVCHPayFunction
	PTNormalVchPayFunction
	ESIDeductionVchPayFunction
	ESIEligibilityVCHPayFunction
	ESIEligibilityOnCurrentEarnVch
	ESIEligibilityOnSpecifiedFrmlVch
	ESIContributionVchPayFunction
	FirstEPF12PayHeadAbsVchVal Function
	FirstEPF833PayHeadAbsVchVal Function

	IsExciseRG23DNoExistsFunc
	IsSpecialAEDOfCVDEExistsInStkItem
	SetTDSPymtDetails
	VoucherFill
	OrderObjExists
	TrackingObjExists
	FillUsingVoucher
	CopyBatchAllocationsValues
	STCatCheck
	STCatRate
	STCatCessRate
	STCatSecondaryCessRate
	STCatAbatementNo
	STCatAbatPer
	STCatCheck

What's New in Release 1.8

1. Invoking Actions on Event Occurrence - with System and Printing Events Introduced

In any language, event handling is one of the powerful features, as it allows the developer to perform some operation based on some implicit action. In order to detect the events and to perform some action based on the event, a proper Event Framework is required.

Prior to this release, the Events **Form Accept** and **Focus** had been introduced. In this release, there has been a major enhancement in Event Framework as a whole. We will see in detail the events supported in TDL. Let's start with an overview of **Event Framework** and types of events.

1.1 Event Framework Overview

When the user does something, an event takes place. Events are actions which are detected by a program and can change the state of system or execution flow. Events can occur based on user actions, or can be system-generated. In TDL, the Key Framework is mainly used to handle user actions like keyboard and mouse events. This can be considered as a part of Event Framework.

We know that TDL is a definition language which does not have any explicit control on the flow of execution. The programmer has no control over what will happen when a particular event occurs. There are certain attributes like SET/PRINTSET, used to initiate some action on occurrence of event/change of state (like report construction, etc.). In this scenario, there is a need of a generic Event Framework, which allows the programmer to trap the events and initiate an action/set of actions in the state when the event has occurred.

The event framework allows the specification of an Event Handler, where it is possible to specify an **Event Keyword**, a **Condition** to control event handling and the **Action** to be performed. The process of detecting an event and executing the specified action is called as **Event handling**.

1.2 Types of Events

When the user operates the application, different types of events are generated. The events are classified as **System Events** or **Object-Specific Events**, based on their origin.

System events are for which no object context is available, when they occur.

Example:

Tally application launch.

Object Specific events are performed only if the specific object context is available.

Example:

Form Accept is a Form-specific event.

System Events

In TDL, a new type '**Events**' has been introduced in the **System** definition. All the system events are defined under this definition. As of now, TDL event framework supports the following four system events, viz. System Start, System End, Load Company and Close Company.

Syntax

```
[System : Events]
    Label : <EventKeyword> : <ConditionExpr> : <ActionKeyword> : +
        <Action Parameters>
```

Where,

<Label> is a name assigned to the event handler. It has to be unique for each event handler.

<EventKeyword> can be one of System Start, System End, Load Company or Close Company.

<ConditionExpr> should return a logical value.

<ActionKeyword> can be any one of the actions.

<Action Parameters> are parameters of the action specified.

The events **System Start** and **System End** are executed when the user launches or quits Tally application, respectively. The events **Load Company** and **Close Company** are executed when the user loads or closes a company, respectively.

Example:

```
[System : Events]
    AppStart1 : System Start : TRUE : CALL : MyAppStart
```

The function *MyAppstart* is called as soon as the Tally application is launched.

Object Specific Events

Objects specific events can be specified for the associated object only.

Example: Before Print event is specific to 'Report' object.

The attribute **ON** is used to specify the **object specific events** as follows:

Syntax

```
ON : EventKeyword : <ConditionExpr> : <ActionKeyword> :
    <Action Parameters>
```

Where,

<EventKeyword> can be any one of 'Focus', 'Form Accept', 'Before Print' and 'After Print'.

<ConditionExpr> should return a logical value.

<ActionKeyword> can be any one of the actions.

<Action Parameters> are parameters of the action specified.

ON is a list type attribute, so **a list of actions** can be executed when the specific event occurs.

□ Event – FORM ACCEPT

The event **Form Accept** is specific to 'Form' object; hence, can be specified only within Form definition. A list of actions can be executed when the form is accepted, which can also be based on some condition. After executing the action Form Accept, the current object context is retained. So all the actions that are executed further, will have the same object context.

The event 'Form Accept', when specified by the user, overrides the default action Form Accept. So, when 'Form Accept' event is triggered, the Form will not be accepted until the user explicitly calls the action 'Form Accept'.

Example:

```
[Form : TestForm]
```

```
On : FormAccept : Yes : HttpPost : @@SCURL : ASCII : SCPostNewIssue : +
    SC NewIssueResp
```

Action **Http Post** is executed when the event 'Form Accept' is encountered. But, the form will not be accepted until the user explicitly calls the action Form Accept on event Form Accept as follows:

```
On : FormAccept : Yes : Form Accept
```

Now, after executing the action Http Post, Tally will execute the action Form Accept as well.

□ Event – FOCUS

The event **Focus** can be specified within the definitions Part, Line and Field. When Part, Line or Field receives focus, a list of actions get executed, which can also be conditionally controlled.

Example:

```
[Part : TestPart2]
```

```
On : FOCUS : Yes : CALL: SCSetVariables : $$Line
```

□ Event – BEFORE PRINT

The event **Before Print** is specific to 'Report' object; so it can be specified only within 'Report' definition. The event 'Before Print' is triggered when the user executes the 'Print' action. The action associated with the event is executed first, and then the report is printed.

A list of actions can be executed before printing the report, based on some condition.

Example:

```
[Report : Test Report]
```

```
On : BEFORE PRINT : Yes : CALL : BeforeRepPrint
```

The function *BeforeRepPrint* is executed first and then the report *Test Report* is printed.

□ Event – AFTER PRINT

The event **After Print** can be specified for Report, Form, Part and Line definitions. It first prints the current interface object and then executes the specified actions for this event. A list of actions can be executed after printing the report based on some condition. **Print** is an alias for After Print.

Example:

```
[Line : LV AccTitle]
```

```
On : After Print : Yes : CALL : SetIndexLV : #LedgerName
```

The function *SetIndexLV* is called after printing the line *LV AccTitle*. So, if there are 10 lines to be printed, the function will be called ten times.

2. Collection Enhancements

2.1 Using External Plug-Ins as a Data Source for Collections

Introduction

A Dynamic Link Library takes the idea of an ordinary library one step further. The idea with a static library is for a set of functions to be collected together, so that a number of different programs could use them. This means that the programmers only have to write code to do a particular task once, and then, they can use the same function in lots of other programs that do similar things.

A Dynamic Link Library is similar to a program, but instead of being run by the user to do one thing, it has a lot of functions “exported”, so that other programs can call them. There are several advantages of this. First, since there is only one copy of the DLL on any computer used by all the applications that need the library code, each application can be smaller, thus saving disk space. Also, if there is a bug in the DLL, a new DLL can be created and the bug will be fixed in all the programs that use the DLL just by replacing the old DLL file. DLLs can also be loaded dynamically by the program itself, allowing the program to install extra functions without being recompiled.

What is DLL?

A **Dynamic Link Library** (DLL) is a library that can be called from any other executable code, i.e., either from an application or from another DLL. It can be shared by several applications running under Windows. A DLL can contain any number of routines and variables.

Dynamic Link Library has the following advantages:

- **Saves memory and reduces swapping:** Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in memory. In contrast, Windows must load a copy of the library code into memory for each application that is built with a static link library.
- **Saves disk space:** Many applications can share a single copy of the DLL on disk. In contrast, each application built with a static link library has the library code linked into its executable image as a separate copy.
- **Upgrades to the DLL are easier:** When the functions in a DLL change, the applications that use them do not need to be recompiled or re-linked, as long as the function arguments and return values do not change. In contrast, statically linked object code requires that the application be re-linked when the functions change.

A potential disadvantage of using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module.

Differences between Applications and DLLs

Even though DLLs and applications are both executable program modules, they differ in several ways. To the end user, the most obvious difference is that DLLs are not programs that can be directly executed. From the system's point of view, there are two fundamental differences between applications and DLLs:

- An application can have multiple instances of itself running in the system simultaneously, whereas a DLL can have only one instance.
- An application can own things such as a Stack, Global memory, File handles, and a message queue, but a DLL cannot.

Types of DLL

When a DLL is loaded in an application, there are two methods of linking, i.e., Load-time Dynamic Linking and Run-time Dynamic Linking. Static Linking happens during program development time, whereas dynamic linking happens at run time.

□ Load time Dynamic Linking /Static Linking

In Load-time Dynamic Linking, an application makes explicit calls to exported DLL functions like local functions. To use load-time dynamic linking, a header(.h) file and an import library (.lib) file are provided, while compiling and linking the application. Thus, the linker will provide the system with information required to load the DLL and resolve the exported DLL function locations at load time.

□ Run-time Dynamic Linking /Dynamic Linking

Dynamic linking implies the process that Windows uses to link a function call of one module to the actual function in DLL. In Run-time Dynamic Linking, an application calls either the function **LoadLibrary** or the function **LoadLibraryEx** to load the DLL at run time. After the DLL is successfully loaded, the function **GetProcAddress** can be used to obtain the address of the exported DLL function that has to be called. In case of run-time dynamic linking, an import library file is not required.

Please note that **Tally** does not support Static Linking; only Dynamic Linking is possible.

DLL Approach in Tally

As discussed before, **Dynamic Link Library (DLL)** is a file that can contain many functions. We can compare it with the library functions provided with many programming languages like C, C++. In Tally, there is provision to access the external functions by uploading the DLLs. In general, the DLLs can be generated using VC++, VB, .Net framework, etc., and can be invoked from TDL. Hence, using TDL, the functions of DLL can be invoked to perform the necessary operations.

Why it is required in Tally?

In Tally, all functions are not required for all the customers. Only generalized features are included to keep the functionality of Tally simple. But, for some customers, basic Tally may not cater the need. For that, we may need to extend the functionality of Tally by writing programs in TDL. TDL is designed to handle the functions in-built in Tally. For the functions that are not available in Tally, we use DLL, wherein we can include many functions and use it in Tally by calling those functions.

How to use DLL in Tally?

Loading the DLL's

1. Copy the DLL file to Tally folder, say C:\Tally.ERP9.

DLL points to the external functions that are to be loaded during startup of Tally application. Tally loads DLLs from the source to the memory, and DLL functions are available with Tally for usage.

OR

2. Register the DLL file using setup program or Command prompt.

In TDL, DLL can be invoked by using **CallDLLFunction** and **DLL Collection**.



*The **CallDLLFunction** is a platform function which was already available earlier, and is NOT a part of Collection Enhancements. It has been discussed here just as an additional information. **DLL Collection** is an enhancement in this Release, which has been emphasized in the subsequent sections.*

Function - \$\$CallDLLFunction

The internal Function **\$\$CallDLLFunction** can be used to call an external DLL containing multiple Functions

Example:

If a DLL "TestDll" contains two functions **FuncA** and **FuncB**, where

- FuncA takes one parameter of 'String' Data Type and returns a String
- FuncB takes a parameter of 'String' Data Type and Executes the Function. It only returns the status of the function execution (boolean value)

The syntax of invoking the DLL from TDL will be as follows:

Syntax

```
[Field : <Field Name>]
    Set As : $$CallDllFunction : <DllName>:<FunctionName> : +
        <Param 1> : <Param 2> ....
```

Where,

<DLLName> is the name of the DLL.

<Function Name> is the name of the Function.

<Param1> and **<Param2>**.... are the Parameters for the function.

The value returned from the function will be available in the field.

To call FuncA

```
[Field : Field2]
```

```
    Use : NameField
```

```
;; Assuming Field1 is of Type 'String'
```

```
Set As : $$CallDllFunction : TestDll : FuncA : #Field1
```

To call FuncB

```
[Key : Key1]
```

```
Key : Ctrl+A
```

```
Action : Set : VarStatus : $$CallDllFunction : TestDll : FuncB : #Field1
```

This key can be associated to a Form or a Menu. The function **FuncB** in TestDll can be used to return the status of the execution, i.e., Success/Failure (1/0). This value can be obtained in a variable in TDL and used to display the appropriate message to the user.

\$\$CallDllFunction can be used to call any function which returns single values. If the function returns an array of values, then it is advisable to use **\$\$DLLCollection**.

Let us have an overview of the usage of DLL Collection.

DLL Collection, its Attributes and Usage

Tally now provides a TDL interface to obtain data sets in Collection from external Plug-Ins. These Plug-Ins are written as DLL's which can be used to fetch external data (i.e., from Internet, external Database, etc.). These DLL's should return a valid XML which can be easily mapped into TDL Collection. In other words, TDL developer can provide a simple string value and/or XML to the DLL function. The DLL gives XML data as an output. Collection takes this data and converts it into objects and object methods, which can be accessed in TDL like other objects.

DLL collection will be very useful in the following scenarios:

1. To display stock quotes from the internet
2. To get data from different formats like CSV, HTML
3. External device interfaces
4. RFID Barcode scanner
5. Petrol Pump device interface
6. Foot fall count
7. External application interfaces
8. GAS distributor application
9. To get attendance details in Pay Roll through swipe

In DLL collection, support is being provided for Plug-Ins and ActiveX-Plug-Ins.

- **Plug-Ins:** DLLs created using C++ or VC++. These DLLs need not be registered separately.
- **ActiveX Plug-Ins:** DLLs created by using VB6, VB.Net, C#.Net, etc. These DLLs have to be registered. Registration process has been explained in detail later.

At present, the 'Collection' definition allows working with a C++ DLL, VB DLL, .Net DLL, etc., which has a function defined by the name **TDLCollection** (The function has to be created by this name ONLY). This function delivers an XML which is available as objects in the Collection.

Attributes of DLL Collection

The attributes of DLL Collection can be categorized as follows:

- For specifying the source

- Data Source
- For sending inputs to DLL
 - Input Parameter
 - Input XML
- For validating/formatting the data received from DLL
 - Break On
 - XSLT
- For selective conversion of XML
 - XML Object
 - XML Object Path
- **Attribute - Data Source**

The attribute **Data Source** is used to set the source of the collection. By using this attribute, the actual DLL is invoked to TDL for further process.

Syntax

```
[Collection : <Collection Name>]
  Data Source : <Type> : <Identity> [:< Encoding>]
```

Where,

<Type> specifies the type of data source, i.e., File XML, HTTP XML, Report, Parent Report, Variable, PlugIn XML, AxPlugIn XML.

<Identity> can be file path / source of DLL.

<Encoding> can be ASCII or UNICODE. It's applicable only for File XML and HTTP XML.

a. For Plug-in DLL

Syntax

```
Data Source : PlugIn XML : <Path to dll>
```

Where,

<Type> is 'PlugIn XML'.

<Identity> identifies the source of DLL, i.e., the path of DLL.

Example:

```
Data Source : PlugIn XML : mydll.dll
```

b. For ActiveX DLL

Syntax

```
Data Source : AxPlugin XML : < Project Name>.<Class Name>
```

Where,

<Type> is "AxPlugin XML".

<Identity> identifies the source of DLL, i.e., < Project Name>.<Class Name>

Example:

```
Data Source : AxPlugin XML : DLLEg1.MyClass
```

For C#.Net, which has concept of namespaces, the source identifier is “Namespace.ClassName”

Syntax

```
Datasource : AxPlugin XML : <namespace>.<classname>
```

Example:

```
Datasource : AxPlugin XML : testcsharpdll.Class1
```

□ Attribute - Input Parameter

The attribute **Input Parameter** is used to pass a single string value to the DLL function.

Syntax

```
Input Parameter : <Expression>
```

Where,

<Expression> returns a string value, which is used to pass to the DLL function.

Example:

```
Input Parameter : Test string
```

In this example, 'Test String' is the string value, which is used to pass to the specified DLL.

□ Attribute - Input XML

The attribute **Input XML** is used to pass the XML format data to the DLL function.

Syntax

```
Input XML : <Post Request Report Name>, <Pre-Request Report Name>
```

Where,

<Post Request Report Name> is the name of the TDL report. It is responsible for generating XML data, which is passed to the DLL function as input.

<Pre-Request Report Name> is optional. It is used to get any input from the end user.

Example:

```
Input XML : DLLRequestReport
```

□ Attribute - Break On

The attribute **Break On** is used to validate the XML data received from the DLL function. If the XML data contains the string specified in this attribute which is referred as error string, then the validation fails and the collection will not be constructed.

Syntax

```
Break On : <String Expression1>, <String Expression 2> ....
```

Where,

<String Expression 1>, **<String Expression 2>**... gives the string values which act as error string to validate the XML data.

Example:

```
Break On : My Error String
```

If XML data received from DLL function contains “My Error String”, then the collection will not be constructed, just as in XML collection.

□ Attribute - XSLT

The attribute **XSLT** is used to transform the XML document received from DLL function to another XML document. It will be applied before constructing the collection. This attribute is same like in XML collection.

Syntax

```
XSLT : <XSLT File name>
```

Where,

<XSLT File name> is the name of the XSLT file name.

Example:

```
XSLT : "C:\Myfile.xslt"
```

□ Attribute - XML Object

The attribute **XML Object** is used to represent the structure of DLL collection object to which the obtained data is mapped. It is an optional attribute, and is same like in XML collection.

Syntax

```
XML Object : <Object Name>
```

□ Attribute - XML Object Path

The attribute **XML Object Path** is used to set the starting XML node from where the object construction starts. If only a specific data fragment is required, it can be obtained using the collection attribute ‘XML Object Path’. This attribute is same like in XML collection.

Syntax

```
XML Object Path : <StartNode> : <StartNodePosition> : <Path to start node>
```

Where,

<StartNode> gives the name of the starting XML Node.

<StartNodePosition> gives the position of the starting XML Node.

<Path to Start Node> gives the path of the starting XML Node.

<Path to start node> can be extended as follows:

```
<root node> : <child node> : <start position> : <child node> : <start position>:....
```

Example:

```
XML Object Path : MyNode : 1 : Root
```




- i. In **DLL collection**, all the attributes except **Datasource** are optional.*
- ii. All error messages related to DLL collection are stored in **dllcollection.log** file.*

Usage of DLL Collection attributes

The following examples demonstrate the usage of DLL collection attributes:

Example: Data Source - AxPlugin Xml

XML data received from the ActiveXDLL "testdll.class1" is to be displayed in a Report. For this, a DLL XML collection is constructed and only a fragment of XML data is to be populated in the collection. Consider the following input XML fragment:

```
<EmpCollection>
  <Emp>
    <Name>Emp1</Name>
    <EmpId>101</EmpId>
    <Designation>Manager </Designation >
  </Emp>
  <Emp>
    <Name>Emp2</Name>
    <EmpId>102</EmpId>
    <Designation >Senior Manager </Designation>
  </Emp>
</EmpCollection>
```



The same XML has been used to explain all further examples.

The TDL code snippet for generating the report is as follows:

```
[Part : DLL Coll Part]

Lines : DLL Coll Line1, DLL Coll Line2

Repeat : DLL Coll Line2 : My DLL Collection
```

```
Scroll : Vertical

[Line : DLL Coll Line1]

Fields : DLL Coll Field1

    [Field      : DLL Coll Field1]

        Set As : "Retrive fragment EMP List from XML data"

[Line : DLL Coll Line2]

Fields : SL No, Emp Name, Emp ID, Emp Desig

    [Field : SL No]

        Use      : Name Field

        Set As : $$Line

    [Field : Emp Name]

        Use      : Name Field

        Set As : $Name

    [Field : Emp ID]

        Use      : Name Field

        Set As : $EmpId

    [Field : Emp Desig]

        Use      : Name Field

        Set As : $Designation

[Collection : My DLL Collection]

    Datasource : AxPlugin XML : testdll.class1

    XML Object Path : Emp : 1 : EmpCollection
```

In this example, the attribute **Datasource** is used to set the source of DLL, i.e., the class name from the DLL "**testdll.class1**". The attribute **XMLObjectPath** retrieves the XML fragment starting from the first **<EMP>** tag under the XML tag **<EmpCollection>** from the specified DLL. The XML data thus fetched from the DLL is then displayed in a Report. Here, **Emp** is the name of the starting XML Node, **1** is the position of the starting XML Node and **EmpCollection** is the path of the starting XML node



*In this case, DLL has to be registered. The registration process is explained in detail in the section “**Implementation and Deployment of DLL**”.*

Example: Data Source - Plugin XML

In the previous example, **ActiveX Plugin** DLL was used. Now, instead of ‘ActiveX Plugin’ DLL, the data source is simple **Plugin** DLL.

The source keyword **PluginXML** is used in the attribute Data source. In this case, only the DLL name must be specified.

The collection definition is as follows:

```
[Collection : My DLL Collection]

Datasource      : Plugin XML : testdll.dll

XML Object Path : Emp : 1 : EmpCollection
```



The only difference from the previous example is that, here the DLL registration is not required. What's just required is to Copy the DLL to Tally.ERP 9 folder and execute the program.

Example: Attribute - Input XML

There are scenarios where the DLL expects some input as in XML fragment before sending the required XML output. The DLL XML collection attribute **InputXML** allows sending the Input XML to the source DLL in XML format. As explained earlier, the attribute InputXML accepts two parameters, i.e., **PostReqReport** and **PreReqReport**.

The collection is defined as follows:

```
[Collection : InputXMLCollection]

Data Source      : AxPlugin XML : TestDLL.Class1

XML Object Path : Emp : 1 : EmpCollection

Input XML       : PostReqRep, PreReqRep
```

In this example, the report ‘PreReqRep’ accepts the user input and the report ‘PostReqRep’ generates the input XML, which is sent to the DLL. The response received from the DLL is populated in the collection **InputXMLCollection**.

The reports ‘PostReqRep’ and ‘PreReqRep’ are defined as follows:

```
[Report : PostReqRep]

Form      : PostReqReport
```

```

Export : Yes

[Form : PostReqReport]

.

.

[Line : PostReqReport]

Fields : Short Name Field, PostReqReportName, Name Field, +
        PostReqRepID, Simple Field, PostReqRepDesig

Local  : Field : Short Name Field : Set As : "Name:"

Local  : Field : Name Field      : Set As : "Emp ID:"

Local  : Field : Simple Field    : Set As : "Designation:"

[Field : PostReqReportName]

Set As : ##PreReqNameVar

XMLTag : "Name"

.

.
    
```

;; Pre Request Report accepting User Inputs

```

[Report : PreReqRep]

Form : PreReqReport

.

.

[Part : PreReqReport]

Lines : PreReqReport Name, PreReqReportID, PreReqReportDesig

[Line : PreReqReport Name]

Fields : Short Name Field, PreReqReport Name

Local  : Field : Short Name Field : Info : "Enter Employee Name:"

[Field : PreReqReport Name]

Use      : Name Field

Set As   : "Enter your Name"
    
```

```

Width      : 50

Modifies   : DLLPreReqNameVar

.

.

```

```
[System : Variable]
```

```
DLLPreReqNameVar : ""
```

Example: Attribute - InputParameter

In scenarios where only one value is to be sent as an input to the source DLL, the attribute **Input Parameter** can be used as follows:

```
[Collection : InpParameterColl]
```

```
Data Source : AxPlugin XML : TestDLL.Class1 XML Object Path : result
```

```
Input Parameter : ##InputParameterVar
```

The value of the variable **InputParameterVar** is sent as an input to the DLL "**TestDLL.Class1**". The response received is available in the collection **InpParameterColl**.

Example: Attribute - BreakOn

The following code snippet validates the XML received from the DLL "**tesdll.class1**".

```
[Collection : DLL XML Get CollObjPath]
```

```
Datasource      : AxPlugin XML : testdll.class1
```

```
XML Object Path : Emp : 1 : EmpCollection
```

```
Breakon         : Manager
```

Break On attribute is used to check whether the error string "**Manager**" exists in the output xml. If the error string exists, the XML is considered as an invalid XML and an empty collection is created. Otherwise, the XML is considered as valid and the collection is populated from the received XML fragment.

Signature of function 'TDL Collection' in the DLL

The DLL created using any programming language, when called from Tally, must contain a main function named as TDL Collection. The signature of this function is specific to each programming language.

The detailed signature of the function **TDL Collection** in different languages is as follows:

For VC++ DLL

Consider the following example for VC++ DLL to generate an XML fragment for Employee details. This DLL accepts the input from the TDL, and returns an XML file as output from DLL. Using this XML fragment, it constructs a collection.

What's New in Release 1.8

```
extern "C" HRESULT    declspec(dllexport)
TDLCollection (const wchar_t * pInputParam,
               const wchar_t * pInputXML,
               wchar_t ** pXMLCollection,
               long * pCollectionSize)
{
    *pCollectionSize = 1024;
    if ((*pXMLCollection = (wchar_t *)
        (CoTaskMemAlloc (*pCollectionSize * sizeof(wchar_t))))== NULL)
    {
        return -1;
    }
    wcscpy (*pXMLCollection,L"<EmpCollection>\
        <Emp>\
            <Name>Emp1</Name>\
            <EmpId>101</EmpId>\
            <Designation>Manager</Designation>\
        </Emp>\
        <Emp>\
            <Name>Emp2</Name>\
            <EmpId>102</EmpId>\
        < Designation >Senior Manager</Designation>\
        </Emp>\
        </EmpCollection>"\
    );
};
```

In this example, there are different inputs given as parameters to the function **TDLCollection**.

- **pInputParam**: It is an input value to the DLL and is a string value of collection attribute “**Input Parameter**”. The TDL passes an input parameter to the DLL.

- **pInputXML**: It is an input to DLL and XML data constructed using collection attribute "**Input Xml**"

Output values from **TDLCollection** function:

pXMLCollection: O/P buffer containing resultant data, based on which, collection is constructed.

pCollectionSize: Number of wide characters, including the terminating NULL character.

For VB 6 DLL

Consider the following example for displaying the values in XML format using VB6. Here also, two parameters are being passed to the TDL Collection.

```
Public Function TDLCollection(pInputParam As String,
                             pInputXML As String) As String
    TDLCollection = "<Root>
                    <Name>Amazing</Name>
                    <Name>Brilliant</Name>
                    </Root>"
```

End Function

In this example, two attributes are being given as parameters to the function **TDLCollection**.

- **pInputParam**: Simple string value to the function, as specified in 'Collection' definition, using the attribute "**Input Parameter**".
- **pInputXM**: Input value in Xml format, as specified in the 'Collection' definition, using the attribute "**Input XML**".

The function must return an output String value in XML format.

For C#.Net DLL

Consider the following example for .Net DLL to convert the input string's case to *upper case*. Here, **TDLCollection** is passed two parameters.

```
public string TDLCollection (string pInputParam, string pInputXml)
{
    string resultxml;// to contain xml to be sent back to Tally
    if (!String.IsNullOrEmpty(pInputXml))
    {
        resultxml = pInputXml.ToUpper();
    }
    else
    {
```

```
        resultxml = null;
    }
    return resultxml;
}
```

In this example, XML data is being passed to the function **TDLCollection**. All the data present in various tags are converted to upper case.

The Input XML will be as follows:

```
<Root>
  <Name>
    <fname>fname 1</fname>
    <lname>lname 1</lname>
  </Name>
  <Name>
    <fname>fname 2</fname>
    <lname>lname 2</lname>
  </Name>
</Root>
```

The output XML will be as follows:

```
<Root>
  <Name>
    <fname>FNAME 1</fname>
    <lname>LNAME 1</lname>
  </Name>
  <Name>
    <fname>FNAME 2</fname>
    <lname>LNAME 2</lname>
  </Name>
</Root>
```

Inputs to the **TDLCollection** function:

- **pInputParam**: It is an input value to the DLL and is a string value of collection attribute “**Input Parameter**”. The TDL passes an input parameter to the DLL.
- **pInputXML**: It is an input value to the DLL and the XML data constructed using collection attribute “**Input Xml**”.

For VB.Net DLL

In VB.Net, the signature for the function **TDL Collection** is as follows:

```
Public Function TDLCollection (ByVal pInputParam As String,  
                               ByVal pInputXML As String) As String
```

Inputs to the **TDLCollection** function:

- **pInputParam**: It is an input value to the DLL and is a string value of collection attribute “**Input Parameter**”. The TDL passes an input parameter to the DLL.
- **pInputXM**: This is an input value to the DLL and the XML data constructed using collection attribute “**Input Xml**”.

Implementation and Deployment of DLL

Once the DLL is ready for deployment, the following should be ensured for implementation of the same:

1. The dependency for the particular DLL needs to be checked, based on the environment in which it is developed. The necessary environment needs to be installed for the same.
2. The DLL needs to be registered in the system where it is to be deployed. This can be done in two ways:
 - Registering the DLL manually.
 - Running the setup program which is created for deployment.

Dependencies with respect to DLLs created using various Environments

- **Created using .NET framework**: For DLLs created using VB .NET, C# .NET, etc., we require Microsoft .Net Framework. For example, if the DLL is created using Visual Studio 2005, then Microsoft .Net Framework 2.0 or above should be installed on the system.
- **Created using Visual Basic 6.0**: For DLLs created using VB 6, we require service pack 6 to be installed on the system.

References

.Net Framework can be downloaded and installed from the following link: [http://
download.microsoft.com/download/6/0/f/60fc5854-3cb8-4892-b6db-bd4f42510f28/dotnet-
fx35.exe](http://download.microsoft.com/download/6/0/f/60fc5854-3cb8-4892-b6db-bd4f42510f28/dotnet-
fx35.exe)

Service Pack 6 for Visual Basic 6.0 can be downloaded from the following links: [http://
www.microsoft.com/downloads/details.aspx?FamilyId=9EF9BF70-DFE1-42A1-A4C8-
39718C7E381D&displaylang=en](http://www.microsoft.com/downloads/details.aspx?FamilyId=9EF9BF70-DFE1-42A1-A4C8-39718C7E381D&displaylang=en)

Multi part - [http://www.microsoft.com/downloads/details.aspx?familyid=83BF08E6-012D-4DB2-
8109-20C8D7D5C1FC&displaylang=en](http://www.microsoft.com/downloads/details.aspx?familyid=83BF08E6-012D-4DB2-8109-20C8D7D5C1FC&displaylang=en)

How to register DLLs?

After downloading the necessary environment, the DLL needs to be registered before it is used and called by the Tally program. As already discussed, there are two ways in which a DLL can be registered.

Let's discuss the two ways of Registering a DLL (**Manual & Set Up**) one by one:

Manual Registration

□ For VB6 DLLs

1. Copy the DLL file to the specific folder say C:\Tally.ERP9
2. Open Command Prompt and change the current directory to the folder where DLL is copied, i.e., :Tally.ERP9
3. Type the command `RegSvr32 <DLL Name>`
4. After the command is entered in command prompt, a message box is displayed as shown:

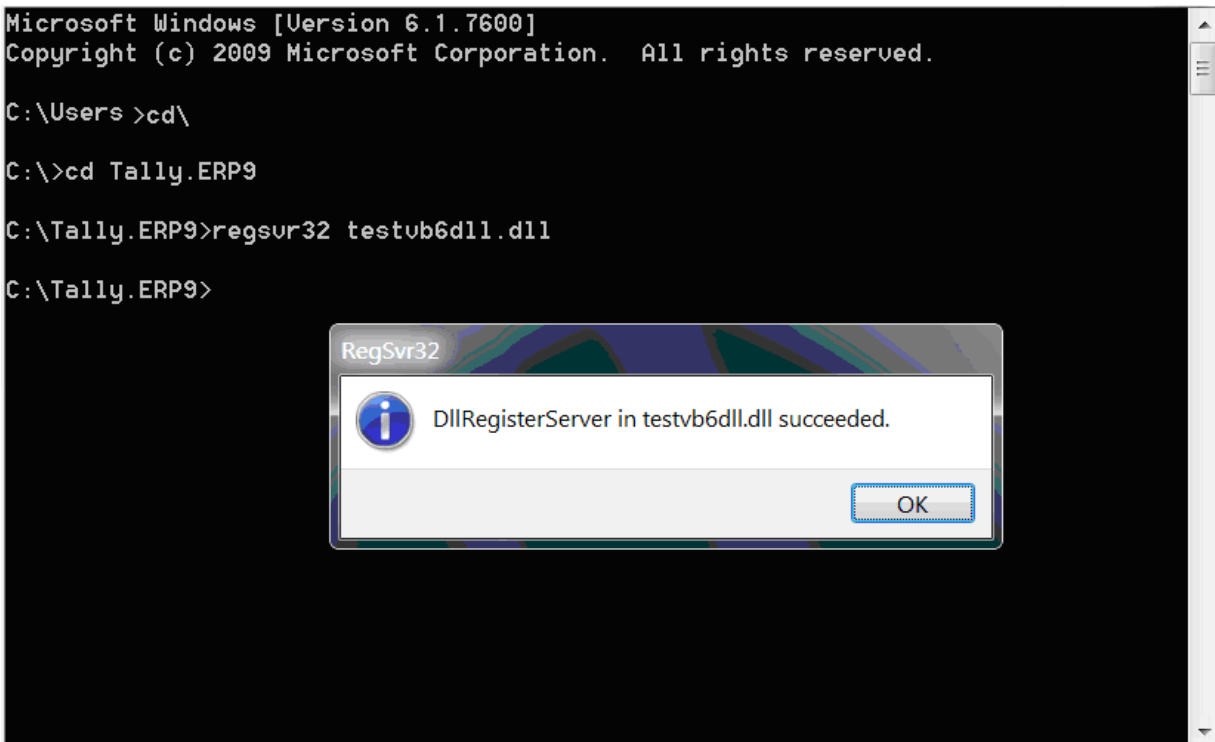


Figure 1. Registering DLL using command prompt

Now you can use the DLL for calling from Tally.

□ For .NET DLLs

To register .Net DLL, the **RegAsm** command needs to be used, instead of **RegSvr32**.



Manual registration does not automatically take care of missing dependencies. So, it is always advisable to use Set Up Programs for Registration.

Registering DLLs by using Setup Program

To register using this method, double click on the setup program and proceed with the installation. This automatically registers the required DLL into the selected folder.

Creating a Set Up Program

The creation of Set Up program varies from one language to another. Please refer to any learning material for Set Up creation specific to your Development Environment. As an example and common usage, we will just discuss creating Set Up using VB 6. The steps for deployment of **VB6** DLLs using Package and Deployment Wizard are as follows:

1. Open the VB project where you want to create a setup program.
2. Select Package and Deployment wizard from Add-In menu.
3. If the option isn't there, choose Add-In Manager and double click Package and Deployment wizard.
4. Proceed with the Wizard options.

For more details, please refer to the following links:

<http://www.configure-all.com/deployment.php>

<http://www.developerfusion.com/article/27/package-deployment-wizard/2/>

2.2 Dynamic Table Support - using 'Unique' Attribute

The **Unique** attribute of 'Collection' definition is used to control the display of unique values in the table for a specified method, based on values selected from the table previously in a field. The display of values is changed dynamically based on the field value.

Existing syntax

The **existing syntax** of the attribute 'Unique' is:

Syntax

```
Unique : <Table Object Method> [,<Field Object Method>]
```

Where,

<Table Object Method> is a method whose value is uniquely displayed in the table.

<Field Object Method> is the storage/method, which is associated with the field which is used to control the display of Table values dynamically. If a particular table object method value from the Table is selected in the field, then that value is removed from the table based on the value of <Field Object Method>. This parameter is optional.

Example:

```
[Part : StkBat]
```

```
Repeat : GrpLedLn : StkItemColl
```

```

[Line : GrpLedNm]
    Field : StkIt, StkBatNm
    [Field : StkIt]
        Use      : Name Field
        Storage  : ItemName
    [Field : StkBatNm]
        Use      : Name Field
        Table    : BatList
        Storage  : BtName
        Show Table : Always
        Dynamic  : Yes
[Collection : BatList]
    Title      : "List of Batches"
    Type       : Batch
    Format     : $BatchName,20
    Child of  : #StkIt
    Unique    : $BatchName, $BtName
[Collection : StkItemColl]
    Type      : StockItem
    Fetch     : Name
[System : UDF]
    BtName    : String : 2010
    ItemName  : String : 2010

```

The table "Bat List" is used to display batch names in a Table attached to the field "StkBatNm". The storage associated with the field is "BtName". Once the Batch name is selected in the field "StkBatNm", in the next line, the table will be populated with batches which are not selected previously in the field.

Even if some stock items belong to more than one batch, the table won't display the common batches, since it may have been already selected in the field for a different stock item. To provide this flexibility for controlling the uniqueness of data, the attribute 'Unique' has been enhanced.

New Enhanced Syntax

The new enhanced syntax is:

Syntax

```
Unique : <Table Object Method> [,<Field Object Method> +
        [,<Extended method>]]
```

Where,

<Table Object Method> is a method whose value is uniquely displayed in the table.

<Field Object Method> is the storage/method, which is associated with the field which is used to control the display of Table values dynamically. If a particular table object method value from the Table is selected in the field, then that value is removed from the table based on the value of <Field Object Method>. This parameter is mandatory if <Extended method> is specified, else it is optional.

<Extended Method> is a storage/method, whose value specifies whether the previous value of the field object method should be used to control unique values display in the table. If the current value of the value of <Extended Method> is same as that of previous values, then <Field Object Method> value is considered while populating unique values in the table. Otherwise, the <Field Object Method> value is ignored to set the unique values in the table. This parameter is optional.

The collection and definition is modified as follows, so that while populating unique values of Batch names in the table, StockItem name is also considered apart from the value of the field storage/method "BtName", i.e., if the same stock item is selected in the field which has been selected previously, then the field storage/method value "BtName" is considered for controlling display of Batches, else it is ignored.

Example:

```
[Collection : BatList]

Title      : "List of Batches"

Type       : Batch

Format     : $BatchName,20

Child of   : #StkIt

Unique     : $BatchName, $BtName, $ItemName
```

Here, the method **\$Itemname** used in the 'Unique' attribute is the storage defined in the field '**StkIt**'.

Use Case:

Consider the following Scenario:

Stock Item	Batch Name
Item 1	Batch A
	Batch B

	Batch C
Item 2	Batch A
	Batch C
Item 3	Batch A
	Batch B
	Batch C

There are two fields in the line, one of which displays stock item name and the other displays batches. The selected batch is stored in a UDF, say **BtName**.

Following table displays the values in each field and unique values in tables based on selection:

Line No	Value in Field 1	Values in Table	Selected Value in Field 2
1	Item 1	Batch A Batch B Batch C Primary Batch	Batch A
2	Item 2	Batch A Batch C Primary Batch	Batch C
3	Item 3	Batch B Batch C Primary Batch	Batch B

2.3 Using Variable as a Data Source for Collections

Collection attribute **Data Source** has been enhanced to support 'Variable' as a data source. Now, variable element(s) can be gathered as objects in the collection and the respective simple member variables are available as methods. Member list variables will be treated as sub-collections.

Syntax

Data Source : <Type> : <Identity> [:<Encoding>]

Where,

<Type> is the type of data source, i.e., File XML, HTTP XML, Report, Parent Report, Variable.

<Identity> can be file path/scope key words/variable specification, based on type of data source.

<Encoding> can be ASCII or UNICODE. It's applicable for the types File XML & HTTP XML.



Please refer to the topic "Using Variable as a Data Source for Collections" under Variable Framework for more clarity, with examples.

3. Evaluating expressions by Changing the Object Context with \$\$ReqOwner Introduced

In a programming language, an expression is a combination of values, variables, operators and functions that are evaluated according to the rules of their precedence and association. Similarly, expressions in TDL can be a combination of Method/Variable/Field/Constant Values, and Function/Formula evaluation results.

Example: For TDL Expression

```
$Name + ##VarTest + $$MachineDate + @@FormABC + 90 + #FieldXYZ
```

Where,

Name is a Method, **VarTest** is a Variable, **MachineDate** is a Function, **FormABC** is a System Formula, **90** is a constant value, and **FieldXYZ** is a Field.

Methods, Variables and Fields are **Leaf components** in an expression as other components like Formulae or Functions finally evaluate into either one of these, or result into constants.

A TDL Expression always gets evaluated in the context of an Interface (Requestor) and Data Object. Whenever a report is constructed, an Interface object hierarchy is created, i.e., Report contains a Form, Form contains a Part, and so on. Every Interface Object is associated with a Data Object. In the absence of explicit data object association, an implicit anonymous object gets associated. A method value is evaluated in context of the data object, while Variable and Field values are evaluated in context of Interface object. There may be cases where we would require evaluating these in a context different from the implicit context (Interface and Data). TDL provides many functions which provide the facility to change the Data or Interface Object Context. A change in Data Object context does not change the current Interface (Requestor) Object context and a change in Interface Object Context does not change the current Data Object Context.

We can categorize these functions into mainly two categories:

- Data Object Context Switching Functions
- Interface Object Context Switching Functions

3.1 Data Object Context Switching Functions

The Tally database is hierarchical in nature, in which the objects are stored in a tree-like structure. Each node in the tree can be a tree in itself. An object in Tally is composed of methods and collections. 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 collections. The Internal Object hierarchy is predefined in Tally and cannot be altered. These can only be persisted in Tally Database.

Every Interface Object exists in the context of a Data Object which is associated to it. As discussed above, an expression (specifically method value) gets evaluated in context of the Data

Object associated with the Requestor (Interface Object). By using the functions as given ahead, we can change the Data Object Context for expression evaluation.



Switching the Data Object Context does not imply a change in the current Requestor (Interface Object)

In all the subsequent examples, we will be using the 'Voucher' Data Object hierarchy to demonstrate the various scenarios for Context Change. Hierarchy of the 'Voucher' Object is as shown in the following figure.

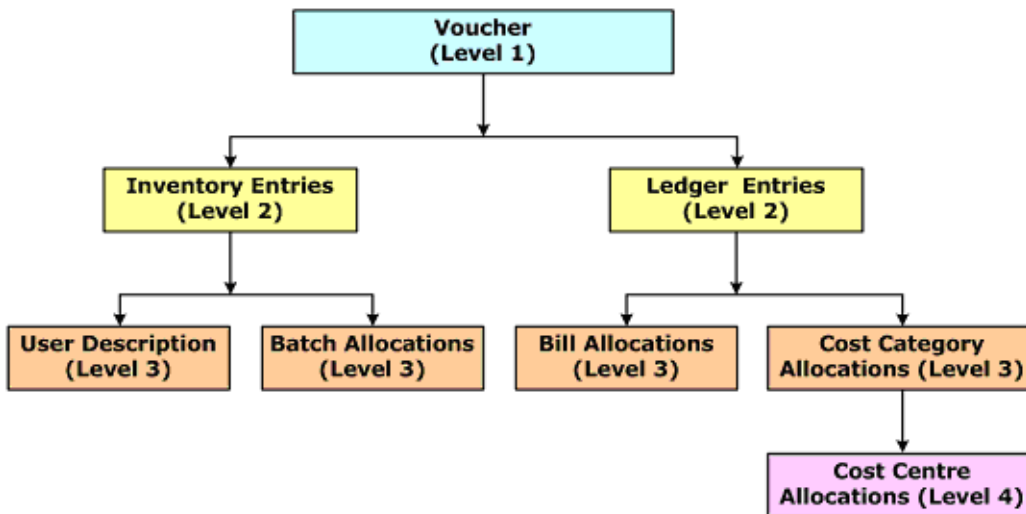


Figure 2. Data Object Hierarchy of Voucher

□ **Function - \$\$Owner**

The function **\$\$Owner** evaluates the given expression in the context of the parent data object in the Data Object hierarchy chain, i.e., it will change the Data Object context to the parent of the current Data Object in context. For example, if the current object context is **Batch Allocations**, then to access the value from **Inventory Entries**, which is its parent data object, the **\$\$Owner** function can be used.

Syntax

`$$Owner : <Expression>`

Example: 1

In this example, let us assume that the "Bill Allocations Amount" field (Requestor) exists in context of **Bill Allocations** Data Object. In order to evaluate the method "**Amount**" from **Ledger Entries** Object Context, we need to use the function **\$\$Owner**.

`[Field : Bill Allocations Amount]`


```
Set As : $$Owner : $Amount
```

In this field, method **Amount** from parent object **LedgerEntries** is set by using **\$\$Owner** function.

Example: 2

Similarly, let's assume that the current data object context for the field "**Bill Allocations Remarks**" is **Bill Allocations**, and we need to evaluate the method **Narration** from **Voucher** Object.

```
[Field : Bill Allocations Remarks]
```

```
Set As : $$Owner : $$Owner : $Narration
```

In this field, **Narration** from object **Voucher**, which is 2 levels above in hierarchy, is set using **\$\$Owner** twice. In other words, we are requesting for method **Narration** from the owner of owner. Alternatively, in these examples, we can use the dotted method formula syntax.

```
[Field : Bill Allocations Amount]
```

```
Set As : $..Amount
```

```
[Field : Bill Allocations Remarks]
```

```
Set As : $...Narration
```

In these examples, .. denotes the parent and ... denotes the Grand Parent.

□ Function - \$\$BaseOwner

Function **\$\$BaseOwner** evaluates the given expression in the context of the base/primary data object in the Data Object hierarchy chain available with the 'Report' Object (in memory).



Since the entire Data Object hierarchy is cached with the Object associated at the Report, the function \$\$BaseOwner changes the Data Object context to the Object associated at the Report.

For example, if the current data object context is **Batch Allocations**, then to access the method value from **Voucher**, **\$\$BaseOwner** function can be used.

Syntax

```
$$BaseOwner : <Expression>
```

Example:

As per the Voucher hierarchy, let's assume that our current data object context for the field "**Bill Allocations Remarks**" is **Bill Allocations**. In order to access the value of Method **Narration** from Voucher Object, which is the base/primary object in the object hierarchy chain, we can use the function **\$\$BaseOwner**.

```
[Field : Bill Allocations Remarks]
```

```
Set As : $$BaseOwner : $Narration
```

In this field, the Method **Narration** from the base Object **Voucher** is set, using **\$\$BaseOwner**.

Alternatively, in the above example, we can use the dotted method syntax.

```
[Field : Bill Allocations Remarks]

Set As : $.Narration
```

In this example, **()** navigates to the Primary/Base Data Object

□ Function - \$\$PrevObj

Function **\$\$PrevObj** evaluates the given expression in the context of the previous data object of the collection, which contains the current data object in context.

Syntax

```
$$PrevObj : <Expression>
```

Example:

Assume that a line is being repeated over a collection of Outstanding Bills, which is sorted based on **PartyName**. After every Party Info, a Total Line is needed to print subtotal for current Party.

```
[Line : Outstanding Bills]

Option : Partywise Totals : $$PrevObj : $PartyName! = $PartyName

[!Line : Partywise Totals]

Add : Lines : At Beginning : Party SubTotal Line
```

In this example, an optional line will be included only if the method **PartyName** of the previous object is not equal to that of the current object.

□ Function - \$\$NextObj

Function **\$\$NextObj** evaluates the given expression in the context of the next data object of the collection, which contains the current data object in context.

Syntax

```
$$NextObj : <Expression>
```

Example:

Assume that a line is being repeated over a collection of Outstanding Bills, which is sorted based on **PartyName**. After every party Info, a Total Line is needed to print the subtotal for current Party.

```
[Line : Outstanding Bills]

Explode : Partywise Totals : $$NextObj : $PartyName! = $PartyName
```

In this example, a part is exploded, provided the method **PartyName** of the next object is different from that of the current object. This will enable explosion for each party only once, and thereby, we can easily achieve the subtotal line as desired.

□ Function - \$\$FirstObj

Function **\$\$FirstObj** evaluates the given expression in the context of the first data object of the collection, which contains the current data object in context.

Syntax

```
$$FirstObj : <Expression>
```

Example:

Assume that a line is being repeated over the ledger collection, where in a field, we require the first object's name to be set.

```
[Field : First Name]
```

```
Set As : $$FirstObj : $Name
```

In this example, a Field **First Name** is set as the **Name** Method of the first object in the Collection.

□ Function - \$\$LastObj

Function **\$\$LastObj** evaluates the given expression in the context of the last data object of the collection, which contains the current data object in context.

Syntax

```
$$LastObj : <Expression>
```

Example:

Assume that a line is being repeated over the ledger collection, where in a field, we require the last object's name to be set.

```
[Field : Last Name]
```

```
Set As : $$LastObj : $Name
```

In this example, a Field **Last Name** is set as **Name** Method of the last object in the Collection.

□ Function - \$\$TgtObject

As we already know, apart from Interface (Requestor) and current Data Object Context, there is one more context available with reference to User Defined Functions and Aggregate Collections, i.e, the Target Object Context. In case of functions, the object being manipulated is the Target Object. In case of aggregate Collection, the object being populated in the resultant collection is the Target Object.

There are scenarios where the expression needs to be evaluated in the context of the Target object. In such cases, the function **\$\$TgtObject** can be used. Using **\$\$TgtObject**, values can be fetched from the target object without setting the target object as the current context object.

Syntax

```
$$TGTObject : <Expression>
```

Example: 1

Consider writing a Function to import a Voucher from Excel, wherein the Source Object is the Collection gathered out of Objects in an Excel Worksheet, the Target Object being the Voucher and its sub objects. While setting value to Inventory Entries sub-object, the Target Object is

changed to **Inventory Entries** and the Source Object continues to be **Excel Objects**. In order to set values to the methods **Quantity** and **Rate**, Stock Item context is required since Unit Information is available for Item. Hence, **\$\$TgtObject** Function is prefixed to the Expression **@BillQty** and **@BillRate**, in order to evaluate these Methods in the context of the Target Object, which is the 'Inventory Entries' Object.

```
[Function : Import Voucher]
```

```
Local Formula : BillQty : $$AsQty : $ExcelBilledQty

Local Formula : BillRate : $$AsRate : $ExcelItemRate

90 : INSERT COLLECTION OBJECT : Inventory Entries

100 : SET VALUE : BilledQty : $$TgtObject : @BillQty

110 : SET VALUE : Rate : $$TgtObject : @BillRate

120 : SET TARGET ..

130 : SAVE TARGET
```

Example:2

Consider another example where, while populating a summary collection of Sales Vouchers, we need to track the maximum sales amount for each Item with the date on which the maximum sales was triggered.

```
[Collection : Src Voucher]
```

```
Type : Vouchers : VoucherType

ChildOf : $$VchTypeSales
```

```
[Collection : Summ Voucher]
```

```
Source Collection : Src Voucher

Walk : Inventory Entries

By : ItemName : $StockItemName
```

;; The following returns the Date and Amount for an Item, on which Maximum sales has happened

```
Aggr Compute : MaxDate : SUM : IF$$IsEmpty : $$TgtObject : $MaxItemAmt +

OR $$TgtObject : $MaxItemAmt <$Amount THEN $Date ELSE +

$$TgtObject : $MaxDate
```

;; MaxItemAmt is a method in the Target Object. Hence, the function \$\$TgtObject is used to evaluate the Method

;; MaxItemAmt in Target Object Context

```
Aggr Compute : MaxItemAmt : MAX : $Amount
```

In this example, while populating the Summary Collection, Method **MaxItemAmt** is being computed for Maximum Amount. Subsequently, Date is also computed by validating if the current object's Amount is greater than the previous computed Amount. Since Maximum Amount so far is computed and accumulated in the Target Object being populated, we need to access it using the function **\$\$TGTOBJEct**. Hence, **\$\$TgtObject:\$MaxItemAmt** evaluates the Method **MaxItemAmt** in the context of the computed Target Object **MaxItemAmt**.

□ Function - **\$\$LoopCollObj**

As we are aware, it is now possible to gather Data Collection in context of each object of another collection, which is referred to as a Loop Collection. To access the methods of **Loop Collection** Objects from within the Data Collection, **\$\$LoopCollObj** is used, with which the expression is evaluated in the context of the Loop Collection Objects.

Syntax

```
$$LoopCollObj : <Expression>
```

Example:

To see a consolidated list of vouchers across all the loaded companies.

```
[Collection : Company Collection]
```

```
Type : Company
```

```
Fetch : Name
```

```
[Collection : Vouchers of Multiple Companies]
```

```
Collection : MultiCmpDB VchCollection : Company Collection
```

```
Sort : Default : $Date, $LedgerName
```

```
[Collection : MultiCmpDB VchCollection]
```

```
Type : Voucher
```

```
Fetch : Date, Vouchernumber, VoucherTypeName, Amount, MasterID, +
```

```
LedgerName
```

```
Compute : Owner Company : $$LoopCollObj : $Name
```

In this example, the function **\$\$LoopCollObj** changes the context to Loop Collection Objects, which is the Company Collection and hence, returns the company name.

□ Function - **\$\$ReportObject**

The function **\$\$ReportObject** evaluates the given expression in the context of the Data Object associated with the Report Interface Object.

One of the important Use Cases of **\$\$ReportObject** is its usage in purview of in-memory Collection gathering. Whenever a collection is gathered, it is retained in memory with the Data Object of the current Interface (Requestor) Object. If the same collection is being used in expressions again and again, then it is beneficial from the performance point of view to attach it to

the 'Report' Object and evaluate it in the context of 'Report' Object n number of times. This eliminates the need to re-gather the collection every time in context of other Data Objects.

Syntax

```
$$ReportObject : <Expression>
```

Example: 1

From **Bill Allocations** Data Object context, Voucher No. of Report Object **Voucher** is required.

```
[Field : Bill No]
```

```
Set As : $$ReportObject : $VoucherNumber
```

Example: 2

In a Report, Sales of each Item against the corresponding Parties is required.

```
[Collection : CFBK Voucher]
```

```
Type : Voucher
```

```
Filter : IsSalesVT
```

```
[Collection : CFBK Summ Voucher]
```

```
Source Collection : CFBK Voucher
```

```
Walk : Inventory Entries
```

```
By : PName : $PartyLedgerName
```

```
By : IName : $StockItemName
```

```
Aggr Compute : BilledQty : SUM : $BilledQty
```

```
Search Key : $PName + $IName
```

```
[Field : CFBK Rep Party]
```

```
Use : Qty Primary Field
```

```
Set as : $$ReportObject : $$CollectionFieldByKey : $BilledQty : +
```

```
@MyFormula : CFBKSummVoucher
```

```
MyFormula : ##PName + #CFBKRepName
```

Here, the function **\$\$ReportObject**, during its first execution, retains the collection within the Voucher Object (which is the Data Object associated with the 'Report' Object). During subsequent calls, method values are fetched from the Objects available in the 'Report' Data Object, instead the entire Collection being regathered again. This helps in improving the performance drastically.

▣ Function - \$\$ReqObject

This function evaluates the given expression in context of the Data Object associated with the Interface (Requestor) Object. There may be scenarios where during expression evaluation, Data

Object context changes automatically and all the methods referred to are evaluated in context of the changed Data Object. The Data Object associated with the Interface (Requestor) Object is lost. Specifically in those cases, where we need to evaluate methods in context of the data object associated with the Interface (Requestor) Object, we will use the function **\$\$ReqObject**.

Syntax

```
$$ReqObject : <Expression>
```

Example:

A Report is required to display Ledgerwise Sales Totals

```
[Field : Fld LedgerSalesTotal]

    Set As : $LedgerSalesTotal

[#Collection : Ledger]

    Compute : LedgerSalesTotal : $$FilterAmtTotal : LedVouchrs : +
            MyParty : $Amount

[Collection : Led Vouchers]

    Type    : Voucher

    Filter  : OnlySales

[System : Formula]

    My Party    : $PartyLedgerName = $$ReqObject : $Name

    Only Sales : $$IsSales : $VoucherTypeName
```

In this example, a new method **LedgerSalesTotal** is added in the Ledger Object to compute the Sales Total from all the Vouchers filtered for the current Party Ledger Object. The Interface Object (Requestor) for this method is the field "**FldLedSalesTotal**". In the Formula **My Party**, current Ledger Name must be checked with the Party Ledger Name of the Voucher Object, which is the current Data Object context. The Data Object associated with the Requestor is Ledger Object. So, in order to evaluate the method **\$name** from the Interface (Requestor) Object's Data Object context, the function **\$\$Reqobject** must be used.

▣ Function - \$\$ObjectOf

As we are already aware, there is the capability to identify a Part and Line Interface Object using a unique Access Name. A Form/Report can be identified from any level using the Definition Type. The function **\$\$ObjectOf** is used to evaluate the expression in context of the Data Object associated with the Interface Object identified by the Access Name.

The Interface Object being referred to, should be assigned a unique AccessName via **Access Name** attribute.

Syntax

```
$$ObjectOf : <DefinitionType> : <AccessNameFormula> : <Evaluation Formula>
```

Example:

```
[Part : Cust Object Association]
```

```
    Lines      : Cust Object Association
```

```
;; Object associated at Part
```

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

```
;; Access Name specified so that this part can be accessible
```

```
    Access Name : "CustLedger"
```

```
;; In some other fields across parts, we can access the methods of the Ledger Object associated with the part "CustObjectAssocia- tion", using the function $$ObjectOf
```

```
[Field : Ledger Parent]
```

```
    Set as : $$ObjectOf : Part : "CustLedger" : $Parent
```

The Part "**Cust Object Association**" is associated with the Ledger Object "**Customer**". It is identified by the Access Name "**CustLedger**". The Field **Ledger Parent** from a different Part accesses the method **\$Parent** from the Ledger object 'Customer', as it is the Object associated with the part **Cust Object Association**, identified by Access Name "**CustLedger**".

▣ **Function - \$\$Table**

It evaluates the **expression** in the context of 'Table' object, which is selected in the given **Field**.

Syntax

```
$$Table : <Field Name> : <expression>
```

Example:

```
[Field : Vehicle Number]
```

```
    Table      : List of Vehicles
```

```
    Show Table : Always
```

```
[Field : Vehicle Type]
```

```
    Set as : $$Table : VehicleNumber : $VehType
```

```
[Field : Vehicle YOP]
```

```
    Set as : $$Table : VehicleNumber : $VehYOP
```

```
[Collection : List of Vehicles]
```

```
    Type      : Veh AggUDF : Company
```

```
    ChildOf   : ##SVCcurrentCompany
```

```
    Format    : $VehNo, 20
```

```
    Format    : $VehType, 40
```



```
Format : $VehYOP, 4  
Fetch  : VehNo, VehType, VehYOP
```

;; For Remote Client End

In this code, the table is displayed in the field "Vehicle Number". In the other fields 'Vehicle Type' and 'Vehicle YOP', `$$Table` is used to evaluate the methods `$VehType` and `$VehYOP` in context of the Data Object selected in the field "Vehicle Number".

□ Function - `$$TableObj`

`$$TableObj` is similar to the function `$$Table`. The expression is evaluated in context of the Data Object selected in the Table in the field specified. The difference of this function from the function `$$Table` is that, in case no object is selected in the Table, or expression evaluation fails, `$$Table` returns a blank string. In such a case, `$$TableObj` returns a logical value (FALSE) as the result.

Syntax

```
$$TableObj : <Field Name> : <expression>
```

Example:

A Field needs to be skipped based on the selection of the table in a field.

```
[!Field : VBOrdDueDRNote]  
  
Skip : $$TableObj : VCHBATCHOrder : $$IsOrder
```

In this example, if the Object selected in the Field **VchBatchOrder** is an Object Order, then the current field needs to be skipped.

3.2 Interface Object Context switching functions

Objects used for designing the User Interface are referred to as Interface objects. Interface objects like Report and Menu are independent items and can exist on their own. The objects Form, Part, Line and Field can't exist independently. A Report can have more than one Form, Part, Line and Field definitions but at least one has to be there. The hierarchy 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

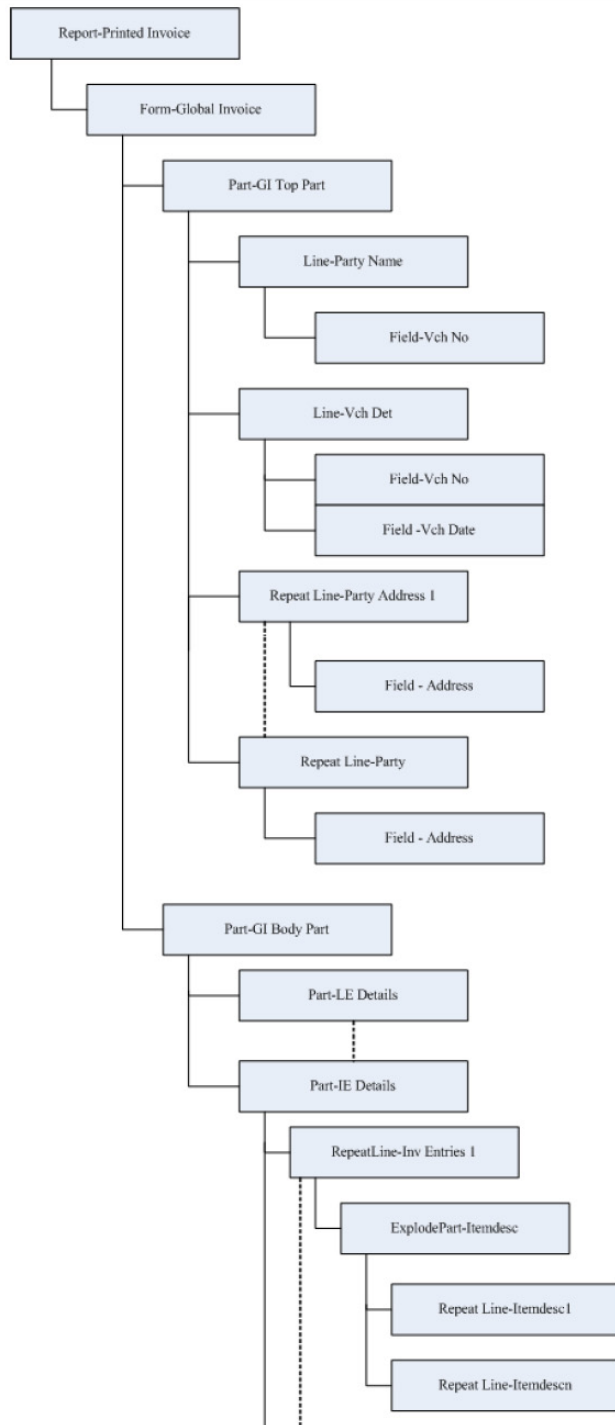


Figure 3. Interface Object Hierarchy

We can take an example of the Simple Customized Invoice Report (as given in the diagram) in order to understand the containment hierarchy of Interface Objects.

A set of available attributes of interface objects have been predefined by the platform. A new attribute cannot be created for an interface object. Interface objects are always associated with a Data Object and essentially add, retrieve or manipulate the information in Data Objects.

At run time, when a report is constructed after the evaluation of all of the above, a complete hierarchy of Interface Objects is created. As we have already discussed, an expression is evaluated in the context of the current Interface Object, which is referred to as the Requestor, and the Data Object associated to it. We will now discuss the switching functions which will change the Interface Object Context for expression evaluation.



Switching the Interface (Requestor) Object Context does not imply a change in the current Data Object context.

□ Function - **\$\$AsReqObj**

The function **\$\$AsReqObj** is used to save the Interface (Requestor) context to the current object, for the evaluation of the expression. All the future references done using **\$\$ReqObject** will point to the saved Interface Object context. The actual requestor is overridden using the function **\$\$AsReqObject**.

Syntax

\$\$AsReqObj : <Expression>

Example:

Here, a Table of Company Vehicles is displayed in a Field “**Select Vehicle**”, which exists in the context of the **Voucher** Object. The table is filtered on the basis of Unused Vehicles.

```
[Field : Select Vehicle]
```

```
;; In Voucher Entry
```

```
Table : CMP Vehicles
```

```
Storage : VCHVehicle
```

```
[Collection : CMP Vehicles]
```

```
Type : Company Vehicles : Company
```

```
Childof : ##SVCURRENTCOMPANY
```

```
Format : $VehicleNumber, 20
```

```
Format : $VBrand, 10
```

```
Title : "Company Vehicles"
```

```
Filter : Unused Veh
```

```
[System : Formula]
```

```
Unused Veh : $$AsReqObj : $$FilterCount : PrevSalesVchs : + UsedVehicle = 0
```

```

Used Vehicle : $$ReqObject : $VehicleNumber = $VCHVehicle

Only Sales   : $$IsSales : $VoucherTypeName

[Collection : PrevSalesVchs]

Type       : Voucher

Filter     : Only Sales
    
```

In this example:

- Field **Select Vehicle** is the Interface (requestor) object, associated with the data object
- Voucher.
- **Table/Collection** of Company Vehicles is displayed in the Field.
- Table is filtered for **Unused vehicles**.
- This collection contains the list of Vehicle Numbers which need to be compared with the ones used in the previous sales vouchers. Since Requestor is the Field with the data object 'Voucher', Function **\$\$ReqObject** will get evaluated in the context of 'Voucher' Object, which is not expected. Hence to make the current collection, i.e., **CMP Vehicles**, as **requestor object** for future reference, Function **\$\$AsReqObj** is used.
- In the Function **\$\$FilterCount**, when the object context changes to the list of sales vouchers, the Function **\$\$ReqObject** evaluates the parameter **\$VehicleNumber** in the context of the requestor Collection **CMP Vehicles** set earlier using **\$\$AsReqObj**, and compares the same with the Voucher UDF **VchVehicle** stored in the respective vouchers.
- **Function - \$\$ReqOwner**

The Function **\$\$ReqOwner** evaluates the given expression in the context of the Interface (Requestor) object's Owner in the current Interface object hierarchy. For instance, Report is an owner requestor for Form, Form is an owner requestor for Part, and so on. From the Line, when the function **\$\$ReqOwner** is used, the expression gets evaluated in the context of the Part containing the current line.

Syntax

```

$$ReqOwner : <Expression>
    
```

Example:

```

[#Menu : Gateway of Tally]

Add : Key Item : ReqOwner Sample : W : Alter : ICCF ReqOwner

[Report : ICCF ReqOwner]

Form       : ICCF ReqOwner

Variable   : VarReqOwner : String : "Keshava"

[Form : ICCF ReqOwner]

Parts     : ICCF ReqOwner
    
```

```

[Part : ICCF ReqOwner]

  Lines : ICCF ReqOwner

  [Line : ICCF ReqOwner]

    Fields : ICCF ReqOwner

    [Field : ICCF ReqOwner]

      Set As      : $$FunctoreturnReqOwner

      Set Always : Yes

[Function : FunctoreturnReqOwner]

  Variable : VarReqOwner   : String       : "Madhava"

  Variable : Temp          : String       : $$ReqOwner : ##VarReqOwner

  01 : MSGBOX : ##VarReqOwner : ##Temp

  10 : RETURN : $$ReqOwner   : ##VarReqOwner

```

In this example, the Variable **VarReqOwner** is declared & initialized in a Report as well as in a function. From the Field, the function **\$\$ReqOwnerFunc** is referred to perform some computation and return the result. Since, **\$\$ReqOwner** is used in the Function and Field is the Requestor Owner for Function, the Field walks back the Interface (Requestor) Object hierarchy to fetch the Variable value. Hence, the Variable value **Keshava** of the nearest Interface Object, i.e., of the Report is returned.

▣ **Function - \$\$AccessObj**

As we are already aware, there is the capability to identify a Part and Line Interface Object using a unique Access Name. The function **\$\$AccessObj** changes the Interface Object context to the one identified by the Access name to evaluate the expression

The Interface Object being referred to should be assigned a unique Access Name via **Access Name** attribute.

Syntax

```

$$AccessObj : <DefinitionType> : <AccessNameFormula> : <Evaluation Formula>

```

Example:

```

[Line : ABC]

  Access Name : "AccABC"

[Field : XYZ]

  Set As : $$AccessObj : Line : "AccABC" : ##VarABC

```

In this example, the function **\$\$AccessObj** changes the Interface Object context from the field **"XYZ"** to the line **"ABC"**, which is identified by the Access Name **"AccABC"**. The variable value is evaluated in context of the line **"ABC"**.

□ Function - \$\$ExplodeOwner

The function **\$\$ExplodeOwner** changes the Interface (Requestor) Object to the Line owning the current exploded Part, and evaluates the given expression, i.e., Field and Variable Values, in the context of Interface Object.

Syntax

```
$$ExplodeOwner : <Expression>
```

Example:

```
[Line : Smp InvEntries]

  Fields : Name Field

  Local  : Field : Name Field : Set As : $StockItemName

  Explode : Smp Expl Part

[Part : Smp Expl Part]

  Lines : Smp Batch Allocations

  Repeat : Smp Batch Allocations : Batch Allocations

  Scroll : Vertical

[Line : Smp Batch Allocations]

  Fields : Name Field

  Local  : Field : Name Field : Set As : $$ExplodeOwner : #NameField
```

In this example, the field **NameField** is being evaluated in the context of the Line **Smp InvEntries**, which owns the current exploded part **Smp Expl Part**.

□ Function - \$\$PrevLine

When the line is repeating, we may require evaluating an expression in the context of the previous line. For example, we might require to fetch the field values stored in the previous line for an expression in the current line. The function **\$\$PrevLine** is used to change the Requestor to the Previous Line for expression evaluation.

Syntax

```
$$PrevLine : <Expression>
```



The function \$\$PrevLine not only changes the Interface (Requestor) Object context, but also changes the Data Object context to the Object associated with the Requestor.

Example:

```
[Line : PrevParticulars]
```

```
Explode : PrevParticulars ExpPart : $$PrevLine : +  
#PartyParticulars != #PartyParticulars
```

In this example, in case of repeated lines, where subtotals are required to be displayed or printed for the same party, we can explode a subtotal line after comparing the previous line's and the current line's Ledgers. If the field values are not the same, then the subtotal line is exploded.

4. Variable Framework with Compound Variables Introduced

Variables in TDL (Tally Definition Language) are entities which can hold values during the execution of a program. The values of these variables are initialized when they are created and can change during the entire execution of program. The Program can change the variable value by specifying expressions which are evaluated to set the values of the variables.

Variables are context-free structures which do not require any specific object context for manipulation. They are declared by name and can be operated using the same name. It is also possible to access and operate variables declared at the parent scope.

Variables are lightweight data structures, which are simple to operate and provide the capability of storing multiple values of the same type and different types as well. It is also possible to perform various manipulation operations like insert/update/delete/sort/find. These are mainly used to perform complex computations.

Variable can hold a single value, or more than one value of same type or different types. It can be declared at various scopes such as Report, Function and System Level.

4.1 Classification of Variables

The various types of variables in TDL are:

1. Simple Variable

Simple variables allow storage of a single value of the specified data type.

2. Simple Repeat Variables

The Simple Variable can hold method values of multiple objects of a collection based on an implicit index. This concept is used in Columnar Reports only, where the lines should be repeated vertically and the fields should be repeated horizontally.

3. Compound Variable

Compound Variables allow us to store values of different data types. This is achieved by making the variable itself compound, by allowing variable declaration inside itself. These sub variables are called member variables of the main variable.

A member variable can be a single instance or a list variable. A member variable can be a compound variable and can have members again, and therefore any hierarchy can be created.

Compound variables help grouping of related information together into one specification. In another terms, we can think about compound variables as an 'object'.

Following table shows the similarities between an object and a compound variable:

Object	Compound Variable
Can have methods	Can have Simple Variables as its members
Can have repeated methods (simple collections)	Can have a Simple List Variable as member
Can have collections (compound collections)	Can have Compound List Variable as its member
Cannot have objects under it directly	Can have Compound Variables as members

We can have a comparison between the internal Data Object '**Voucher**' and a Compound Variable '**CLV Emp**' to understand the similarities between an Object and Compound Variable.

For instance, the Compound Variable '**CLV Emp**' is defined as follows:

```
[Variable : CLV Emp]
```

```
Variable      : Name          : String
```

```
Variable      : Designation : String
```

```
Variable      : Age           : Number
```

```
Variable      : Salary       : Amount
```

```
List Variable : Contact Nos  : String
```

```
List Variable : Relatives
```

```
Variable      : Contact Address
```

;; Defining Compound Variable

```
[Variable : Relatives]
```

```
Variable : Name      : String
```

```
Variable : Age       : Number
```

```
Variable : Relation  : String
```

```
Variable : Salary    : Amount
```

;;Defining another compound variable

```
[Variable : Contact Address]
```

```
Variable : Street Name : String
```

```
Variable : City Name   : String
```


Object: Voucher	Compound Variable: CLV Emp
Object "Voucher" is having methods directly under it such as Date, Voucher Number, Narration, etc.	Compound Variable "CLV Emp" is having Simple Member Variables such as Name, Age, Salary, etc.
Voucher is having the repeated method BasicBuyerAddress (Simple Collection)	CLV Emp is having the Simple List Member Variable 'Contact Nos'
Voucher is having the collection "Inventory Entries" (Compound Collection).	CLV Emp is having the Compound List Member Variable 'Relatives'
Voucher object is not having another voucher (primary object) under it directly.	CLV Emp is having the another Compound Member Variable 'Contact Address'

4. List Variable

A variable at declaration time can be declared as a single instance or as a list. List variable is a container (data structure) variable and hence it is not defined. Variables can be declared as list.

List Variable can hold one or more variables which can be either a simple or a compound variable. Each of these is called Element Variable. Element Variable holds value as well as key, if specified. The key is optional, and hence without a key also, elements can be added to list variables. The value of key specified for each of the element variables must be unique.

- **Simple List Variable**

Simple Variable can be declared as a list. Simple List Variables can hold multiple values of single data type.

- **Compound List Variable**

Compound Variable can be declared as a list. Compound List Variables can hold multiple values of different data types.

4.2 'Variable' Definition and its Attributes

Definition - VARIABLE

A **Variable** definition is similar to any other definition. The behaviour of the variable is specified by the programmer via 'Variable' definition.

Syntax

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

A meaningful name which determines its purpose can be given as a variable name.

Attributes of 'Variable' Definition

Let us discuss the attributes of 'Variable' definition in detail.

- **Attribute - TYPE**

This attribute determines the Type of value that will be held by the variable. All the data types

supported by TDL such as String, Number, Date, etc., can be used to specify the variable data type. In the absence of this attribute, a variable assumes to be of the Type 'String' by default.

Syntax

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

Example:

```
[Variable : GroupNameVar]
Type : String
```

In this example, a variable which holds the data of Type 'String' is defined.

□ Attribute- DEFAULT

The default value of variables can be specified during definition, using **DEFAULT** attribute. It is the initial value assigned to the variable when it is instantiated / declared. We can also specify the default value during declaration / instantiation. The difference is that the default value specified using this attribute at definition time will be applicable to all instances of the variable declared (at any scope). Default value specified while declaration will apply only to the specific instance.



Declaration and scope will be covered in detail in the subsequent topics. The above explanation will be more clear after that.

Syntax

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

Example:

```
[Variable : GroupNameVar]
Type : String
Default : $$LocaleString : "SundryDebtors"
```

In this example, the default value for the variable is set as "Sundry Debtors".

□ Attribute - VOLATILE

If the **Volatile** attribute in Variable definition is set to **Yes**, then the variable is capable of retaining previous values from the caller scope. The default value of this attribute is **Yes**, i.e., if the variable by the same name is declared in the called Report/Function and the 'Volatile' attribute is set to "Yes", then in the called Report, it will assume the last value from the caller Report. The default value of the attribute 'Volatile' is always YES.

For better understanding, let us elaborate it further. When a variable is declared / instantiated, it assumes a default value. The default value which it assumes is controlled by the following factors:

1. If 'Volatile' is set to "Yes" for a variable in its definition which is instantiated / declared inside a function/report, and the variable by the same name exists in the parent scope, then it will take its default value from the Parent scope. If no variable by the same name exists in the parent scope, it will take the default value specified within the definition.

2. If the default value is specified within the declaration itself, it will assume that value. If a new report **Report2** is initiated, using a volatile variable **GroupNameVar**, from the current report **Report1**, the same variable in Report 2 will have the default value as the last value saved in Report 1. Within Report 2, the variable can assume a new value. Once the previous report **Report1** is returned back from **Report2**, the previous value of the variable will be restored. A classic example of this is a drill down **Trial Balance**.

Syntax

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

Example:

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

Volatile Attribute of **GroupNameVar** Variable is set to **Yes**, which means that 'GroupNameVar' can inherit values from one Report to another.

Variables defined at the function level are Non Volatile by default. They do not inherit the values from the caller scope.



Scope will be discussed in detail in the subsequent topics.

□ Attribute - PERSISTENT

This Attribute decides the retention periodicity of the variable, i.e., till when it will retain the value: i) till application termination, or ii) after application termination as well. Setting the attribute **Persistent** to **Yes**, means that the value saved during the last application session will be retained permanently in the system. When the next session of Tally is started, it will take its initial value from the value saved in the previous session, i.e., the latest value of the variable will be retained across the sessions. Please note that Variables declared at the system scope can only be persisted.

A List variable at System scope can also be persisted by specifying the 'Persistent' attribute for its element variable (whether it is simple/compound) within the definition. Inline variables even at system scope cannot be persisted. Inline variable declaration will be discussed in further topics.

Syntax

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

Example:

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

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



All the Persistent Variable Values are stored in a File Named TallySav.Cfg in the folder path specified for Tally Configuration file in F12 -> Data Configuration. Each time Tally is restarted, these variable values are accessed from this file.

□ Attribute - REPEAT

The attribute **Repeat** for a variable is used for its usage in Columnar Reports. It accepts Collection name and optional Method name, as parameters. Multiple values are stored in the variable based on an implicit Index. Method value of each object of the collection will have to be picked up and stored in the variable, based on implicit index. In case the method name is not specified, the variable name is considered as the method name and picked up from the collection.

Syntax

```
[Variable : <Variable Name>]
Repeat : <Collection Name> [:<Method Name>]
```

Where,

<Variable Name> is the name of the variable.

<Collection Name> can be any expression which evaluates to a Collection name.

<Method name> is the name of the method whose value needs to be picked up from each object of the collection. If not specified, the variable name is considered as the method name.

Example:

```
[Variable : SVCCurrentCompany] Volatile : Yes
Repeat : ##DSPRepeatCollection
```

Suppose 'DSPRepeatCollection' holds the value "List of Primary Companies". Method value 'SVCCurrentCompany' will be gathered from each object of the collection and stored in index 1, index2, and so on.



'Repeat' Attribute will be elaborated further under the topic "Implication of Repeat Variables in Columnar Report".

□ Attribute - VARIABLE

The attribute **Variable** is used to define the member variables (Simple/Compound) for a Compound Variable.

Syntax

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

Where,

<Variable Names> is the list of Simple or Compound Variables, separated by comma.

<Data Type> is used to specify the data type of Simple Variable. In case of Compound Variable, data type cannot be specified, as it consists of members belonging to various data types. If the data type is not mentioned, the primary variable definition is mandatory.

<Value> is the default/initial value provided for the variable.

Specifying **<Data Type>** and **<Value>** is optional. If data type is specified, then it is called inline declaration of variable. [We will learn about inline declarations and Compound Variables further].

Example:

```
[Variable : CLV Emp]

Variable : Name      : String

Variable : Age       : Number : 25

Variable : Salary    : Amount

Variable : Relatives
```

In this example, the simple variables Name, Age and Salary and the compound variable 'Relatives' are defined as members for the Compound Variable **CLV Emp**.

□ Attribute - LIST VARIABLE

The attribute **List Variable** is used to specify a list of Simple/Compound Variables.

Syntax

```
[Variable : <Variable Name>]
  List Variable : <Variable Names> [:< Data Type> [:<Value>]]
```

Where,

<Variable Names> is the list of Simple or Compound Variables, separated by comma.

<Data Type> is the data type of Simple Variable. In case of Compound Variable, data type cannot be specified, as it consists of members belonging to various data types.

<Value> denotes the no. of elements in the list. Specifying **<Data Type>** and **<Value>** is optional.

Example:

```
[Variable : CLV Emp]

Variable      : Name      : String

Variable      : Age       : Number

Variable      : Salary    : Amount

List Variable : City      : String : 3

List Variable : Relatives

[Variable : Relatives]
```

```

Variable : Name      : String
Variable : Age       : Number
Variable : Relation  : String
Variable : Salary    : Amount

```

In this example, in addition to simple variables, a simple list variable **City** and a compound list variable **Relatives** are defined as members using the attribute **List Variable**. A separate definition is required for the compound list variable **Relatives**, as it holds the multiple values of different data types.

4.3 Variable Declaration and Scope

Variables can be declared at various scopes. The availability of the variable within the definition under which it is declared is called as the scope. The lifetime of the variable will be within the scope. For example, if the scope of a particular variable is within a function, then the variable will last till the function is executing, and then it is destroyed.

Variables can be declared at System, Report and Function scopes. Let us have a detailed look on the variable scopes.

System Scope declaration

Variables declared at the system level will start their life when the application starts, and will be alive till the application's termination.

System variables are declared using a special [System: Variable] definition. The variables declared at system scope are accessible everywhere in the system.

Syntax

```

[System : Variable]
    Variable Name : <Initial Type Based Value>
                Or
    Variable      : <Variable Names> [:<Data Type>[:<Value>]]
    List Variable : <Variable Names> [:<Data Type>[:<Value>]]

```

Where,

<Initial Type Based Value> is the initial value specified to the variable.

The variables can be declared at the system scope by using the above. The usage of the attributes 'Variable' and 'List Variable' is same as described above in the "Variable Definition".

Example:

```

[System : Variable]
    BSVerticalFlag : No

```

The **BSVerticalFlag** Variable is declared in System Scope. Hence, this variable value being modified in a Report, is retained even after we quit and re-enter the report.

Report Scope declaration

Variables declared at **Report** definition are termed as having 'Report' Scope. These variables will exist till the life of the report. The variables declared at Report scope are accessible from the report itself and all the TDL elements which are executed from within this report such as another report, function, etc.

Report variables get their default value from definition specification, or from the declaration specification, or the values are inherited from the owner scope, if the variable is marked as Volatile.

Report allows two special attributes **SET** and **PRINT SET** to set / override the values of the variable during the startup of the report in Display / Print mode respectively.

'Form' definition also has a SET attribute, which overrides the variable's value during startup creation and subsequent re-creation of the form during any refresh / regeneration. We will study about these value specification attributes in detail under the topic "Manipulating Simple and Compound List Variables".

Syntax

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

```
Variable : <Variable Names>
```

Or

```
Variable : <Variable Names> [:<Data Type> [:<Value>]]
```

Or

```
List Variable : <Variable Names> [:<Data Type> [:<Value>]]
```

The variables can be declared at Report scope by using the above. The usage of attributes **Variable** and **List Variable** is same as described above in the "Variable definition".

Example:

```
[#Report : Balance Sheet]
```

```
Variable : Explode Flag
```

'Explode Flag' Variable is made local to the Report 'Balance Sheet' by associating it using the Report attribute '**Variable**'. This variable retains its value as long as we work with this Report. On exiting the Report, the variable is destroyed and the values are lost.

Function Scope declaration

Function (User Defined Function) also allows the variables to be declared at its scope. Function variables have lifetime till the end of execution of the function. Function variables can also be declared with default value. Function variables will never inherit the value from the parent context. This means that 'Volatile' attribute on function variables has no effect. Functions allow actions to change the values of the variables.

Function allows a special scope called STATIC. A static variable declared in a function is equivalent to a system variable, but can be accessed only within the defined function. Its initial value is set only during the first call to the function, and later it retains the value for further calls. Only simple or compound variables can be declared as static. List variables are not currently supported at 'Static' scope.

Syntax

```

Variable      : <Variable Names>
              Or
Variable      : <Variable Names> [:<Data Type> [:<Value>]]
              Or
List Variable : <Variable Names> [:<Data Type> [:<Value>]]
              Or
Static Variable : <Variable Names> [:<Data Type> [:<Value>]]

```

The variables can be declared at 'Function' scope by using the above. The usage of the attributes **Variable** and **List Variable** is the same as described above in the "Variable" definition.

Example:

```

[Function : FactorialOf]

Variable : Factorial

```

The Function '**FactorialOf**' requires variable '**Factorial**' for calculation within the Function.

Example:

```

[Function : Sample Function]

Static Variable : Sample Static Var : Number

```

The static variable '**Sample Static Var**' retains the value between successive calls to the Function 'Sample Function'.

Inline Declaration

Variables can also be defined (with limited behaviour) during declaration itself; so a separate definition would not be mandatory. This is called inline variable specification (i.e., during declaration itself, the variables are defined inline).

Only the DATA TYPE and the DEFAULT VALUE can be specified as the behaviour for inline variables. If the DATA TYPE is specified as a variable name (i.e., not an implicit data type key word such as String, Amount, etc.) or is left blank, it is treated as a pre-defined variable.

Persistence: Inline variables even at system scope cannot be persisted.

Declaring Simple Variable Inline

The '**Variable**' attribute allows declaring Simple Variable inline by specifying the data type. Initial value to the variable can also be specified optionally.

Syntax

```

Variable : <Variable Names> [:<Data Type> [:<Value>]]

```

Where,

<Variable Names> is a list of Simple Variables, separated by comma.

<Data Type> is the data type of the Simple Variable.

<Value> is the default/initial value provided for the variables, and this value specification is optional.

Example:

```
[Report : Cust Group Report]
```

```
Variable : VarGroupName1, VarGroupName2 : String : "Sundry Debtors"
```

In this example, the Simple Variables 'VarGroupName1' and 'VarGroupName2' of type 'String' are declared in a Report; hence, the following separate variable definitions are not required, which will help to reduce the coding complexity.

```
[Variable : VarGroupName1]
```

```
Type : String
```

```
[Variable : VarGroupName2]
```

```
Type : String
```

Declaring Simple List Variable Inline

'List Variable' attribute allows declaring Simple List Variable inline by specifying the Data Type. If the default value is specified, it is treated as the count to initialize the list with the specified elements by default.

Syntax

```
List Variable : <Variable Names> [: <Data Type> [: <Value>]]
```

Where,

<Variable Names> is a list of Simple Variables, separated by comma.

<Data Type> is the data type of the Simple Variable.

<Value> is treated as the count to initialize the list with the specified elements by default. The number of elements can be specified only for an index-based list.

Example:

```
[System : Variable]
```

```
List Variable : VarGroupName1, VarGroupName2 : String : 10
```

In this example, the variables 'VarGroupName1' and 'VarGroupName2' of 'String' data type are declared as inline simple list variables at System level, and each variable will have 10 elements by default.

Declaring Compound List Variable Inline

For Compound List Variables, **definition is mandatory**. They cannot be declared inline.

4.4 Using Modifiers with Variables

Variable allows static modifiers such as **Add/Delete/Change** and Dynamic modifier 'Local'.

Static Modification

Add/Delete/change modifiers can be used on variables to change the behaviour.

Example:

```
[#Variable : SV From Date]

Delete : Default
```

Locally modifying variables

When different reports require the same Compound Variable, and some modifications are required specific to respective reports, like adding additional members (local to the report); this is possible through the Dynamic Modifier 'Local'.

Example:

In this example, a Compound Variable CLVEMP is defined as shown:

```
[Variable : CLV Emp]

Variable      : Name      : String
Variable      : Designation : String
Variable      : Age       : Number
Variable      : Salary     : Amount
List Variable : Contact Nos : String
List Variable : Relatives
Variable      : Contact Address
```

:: Defining Compound List Variable

```
[Variable : Relatives]

Variable : Name      : String
Variable : Age       : Number
Variable : Relation  : String
Variable : Salary    : Amount
```

:: Defining another compound variable

```
[Variable : Contact Address]

Variable : Street Name : String
Variable : City Name   : String
```

In 'Employee Report1', the variable is declared and no modifications are required locally.

```
[Report : Employee Report1]

Variable : CLV EMP
```

In 'Employee Report2', the same variable is declared but locally one member variable is added and one existing member variable is deleted.

```
[Report : Employee Report2]
```

```
Variable : CLV EMP
```

```
Local      : Variable : CLV EMP : Add      : Variable : Qualification : String
```

```
Local      : Variable : CLV EMP : Delete  : Variable : Age
```

Also, member variables can be localized within a compound variable. This provides the ability to re-use a compound structure defined earlier and do any local modifications, as required.

Example:

```
[Variable : CLVEMP]
```

```
Variable : Contact Address
```

```
Local      : Variable : Contact Address : Add : Variable : State : String
```

4.5 List Variable Manipulations

Simple and Compound List variables support various data manipulation operations such as Adding/Deleting/Expanding List elements, Value Specifications, Retrieving values from the list elements, Searching and Sorting, Populating List Variable from a Collection, etc. New Actions and Functions specific to List Variables have been introduced for these manipulations. Before looking into these manipulations, let us understand the concept of Key, Index and Variable Path Specification using Dotted Notation Syntax.

Concept

1. Key

List variables can hold multiple values of variable types using a string based 'Key' specification. 'Key' is of type String, by default. We can specify a different data type for a key only in scenarios where we require key-based sorting. It is optional to specify key value while adding values to the list variable. The TDL Programmer has to explicitly specify the key value. Key is unique for all elements in the list. If an element is added with duplicate key, the existing element is overwritten. It is advisable to use a key only if we require frequent access to elements of the list based on key.

2. Index

An element of the list can be accessed via 'Index'. Index of an element is the location/position of the variable from the first element in the current sorting order. Even if we have specified keys for elements of a list, index is generated internally. It is always possible to access each element in the list by specifying the index within square brackets [] in the dotted notation syntax. This is explained below. Index can be negative as well. In that case, it is possible to access the elements in the reverse order of entry.

3. Variable Path Specification using Dotted Notation Syntax

We aware that in Tally.ERP 9, method value of any object including its sub-collections to any level can be accessed or modified with dotted notation syntax. The behaviour of the symbol prefix \$ was enhanced to access the method value of any object, and an action MODIFY OBJECT was introduced to modify multiple values of any object.

Compound Variables allow us to store values of different data types. A member variable can be a single instance or a list variable. A member variable can be a compound variable and can have members again, and thus, any hierarchy can be created. In short, it is similar to a Data Object. Hence, all the attributes and actions which operate the Variable, have now been enhanced to take extended variable path syntax, i.e., the variable path can be specified using dotted notation syntax. The syntax can be used to fetch any value from any member within the hierarchy. This syntax is applicable wherever we need to specify either the variable identifier or access the value of the variable. In case of value access the operator **##** is used. Value access using operator **##** has been discussed in detail in the topic **Index Based Retrieval using ## Operator**.

Syntax

```
<Element Variable Specification>.<Member Variable Specification>. +
<Simple Member Value specification>
```

Where,

<Element Variable Specification> can be a Compound Variable or Compound List Variable [Index Expression].

<Member Variable Specification> can be a Compound Variable Member or Compound List Member Variable [Index Expression].

<Simple Member Value Specification> refers to the name of the simple member in the specified path.

<Index Expression> is an expression evaluating to a number. Suffixing a variable with index refers to an Element Variable. It can be positive or negative. Negative index denotes reverse access.

Example: 1

Consider the compound variable defined below:-

```
[Variable : CLV Emp]

Variable      : Name : String

Variable      : Age  : Number

Variable      : Salary : Amount

List Variable : Relatives

[Variable : Relatives]

Variable : Name      : String

Variable : Age       : Number

Variable : Relation  : String

Variable : Salary    : Amount
```

The same is declared at System Scope, and hence can be accessed anywhere in the system.

```
[System : Variable]
```

```
List Variable : CLV Emp
```

Example: 2

Suppose we want to set the value of a simple variable 'Employee Name', which is declared at Report Level:

```
[Report : Employee Report]
```

```
Variable : Employee Name : String
```

```
SET      : Employee Name : ##CLVEMP[1].Name
```

The variable **Employee Name** will be set with the value of the member "Name" of the first element of the Compound List Variable "CLVEMP".

Example: 3

In case the age of first relative of the second employee needs to be displayed, the following statement would be used in the field in a report.

```
[Field : RelAge]
```

```
Set As : ##CLVEMP[2].Relatives[1].age
```

The value specification attributes and actions, with the enhanced variable path specification, will be discussed in detail in the further topics.

List Variable Manipulations – A Detailed Look

Let us have a detailed look on List Variable manipulations with examples:-

Adding/Deleting/Expanding Elements

1. Adding Elements to the List Variable

□ Action - LIST ADD

The Action LIST ADD is used on a list variable to add an element to the list variable based on KEY. This is mandatory before we set value into the element. KEY is compulsory in this case. Key is unique for all elements in the list. If an element is added with duplicate key, then the existing element is overwritten.

Syntax

```
LIST ADD : <List Variable Specification> : <Key Formula>
          [:<Value Formula> [:<Member Specification>]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be any expression which evaluates to a unique string value.

<Value Formula> can be any expression which returns a value. It sets the initial value of the element variable, and is optional.

<Member Specification> is required only if the value needs to be added to a specific member of a Compound List Variable. If member specification is not provided, the first member variable is considered for the value.



The actions `LIST APPEND` and `LIST SET` are aliases for the action `LIST ADD`.

To add multiple values dynamically to the List variable, we can use `LIST ADD` within a looping construct like `While`, `Walk Collection`, etc.

Example:

Adding elements to Simple List Variable using `LIST ADD`

1. Adding an element to the Simple List Variable `SLV Emp` with a Key

```
LIST ADD : SLV Emp : "E001"
```

2. Adding an element to the Simple List Variable `SLV Emp` with a Key and a value

```
LIST ADD : SLV Emp : "E001" : "Kumar"
```

3. Adding an element to the Simple List Variable `SLV Emp` with a Key and a value, and subsequently overriding a value corresponding to a particular key

```
LIST ADD : SLV Emp : "E001" : "Kumar"
```

```
LIST SET : SLV Emp : "E001" : "Keshav"
```

The value corresponding to the Key 'E001' is changed to Keshav

Adding Elements to Compound List Variable using `LIST ADD`

A Compound Variable `CLV Emp` is defined, which stores employee details such as Name, Age, Salary, etc., and the details of the Relatives.

```
[Variable : CLV Emp]
;;simple member variable
Variable : Name      : String
;;simple member variable
Variable : Age       : Number
;;simple member variable
Variable : Salary    : Amount
;;compound list member variable
List Variable : Relatives
;; Compound Variable is defined here
[Variable : Relatives]
Variable : Name      : String
```

```
Variable : Age      : Number
```

```
Variable : Relation : String
```

```
Variable : Salary   : Amount
```

The same is declared at the System Scope; hence, can be accessed anywhere in the system.

```
[System: Variable]
```

```
List Variable: CLV Emp
```

1. Adding an element to Compound List Variable CLV Emp with a Key

```
LIST ADD : CLVEmp : "E001"
```

2. Adding an element to Compound List Variable CLV Emp with a Key and a Value

```
LIST ADD : CLVEmp : "E001" : "Kumar"
```

Since member specification is not provided, the first member variable is considered for value.

3. Adding an element to Compound List Variable CLV Emp with a Key and a value with member specification

```
LIST ADD : CLVEmp : "E001" : 25 : Age
```

Since member specification is provided, member variable 'Age' is considered.

4. Adding an element to the Compound List Member of a Compound List Variable with a Key and a value with member specification

```
LIST ADD : CLVEmp[1].Relatives : "R001" : "Prem" : Name
```

In this example, we are adding an element to the Compound List Variable "Relatives" and the member variable 'Name' is considered for the value. 'Relatives' is a Compound List Member variable of the Compound List Variable CLVEMP.



The values are hard coded in the examples for explanation purpose. The above Simple and Compound List Variable examples are used to explain further list variable manipulations.

□ Action - LIST ADD EX

This action is used on a list variable to add an element to the list variable without **KEY**.

Syntax

```
LIST ADD EX : <List Variable Specification> [:<Value Formula> +
           [:<Member Specification>]]
```

Where,

< List Variable Specification> is the Simple List / Compound List Variable specification.

<Key Formula> can be any expression which evaluates to a unique string value.

<Value Formula> can be any expression which returns a value. It sets the initial value of the element variable, and is optional.

<Member Specification> is required only if the value needs to be added to a specific member of a Compound List Variable. If member specification is not provided, the first member variable is considered for the value.



Action `LIST APPENDEX` is an alias for the action `LIST ADDEX`.

Adding elements to Simple List Variable using LIST ADD EX

1. Adding an element to Simple List Variable SLV Emp

```
LIST ADD EX : SLV Emp
```

2. Adding an element to Simple List Variable SLV Emp, with Value

```
LIST ADD EX : SLV Emp : "Kumar"
```

Adding elements to Compound List Variable using LIST ADD EX

1. Adding an element to Compound List Variable CLV Emp

```
LIST ADD EX : CLV Emp
```

2. Adding an element to Compound List Variable CLV Emp, with value

```
LIST ADD EX : CLV Emp : "Kumar"
```

Here, since member specification is not provided, first member variable is considered for value.

3. Adding an element to Compound List Variable CLV Emp, with value and member specification

```
LIST ADDEX : CLV Emp : 25 : Age
```

Here, member specification is provided, hence member variable 'Age' is considered for the value.

4. Adding an element to the Compound List Member variable of a Compound List Variable with value and member specification

```
LIST ADDEX : CLVEmp[1].Relatives : "Prem" : Name
```

In this example, we are adding an element to the Compound List Variable "Relatives" and the member variable 'Name' is considered for the value. 'Relatives' is a Compound List Member variable of the Compound List Variable CLVEMP.

2. Deleting Elements from the List Variable

□ Action - LIST DELETE

The Action LIST DELETE is used to delete an element from the list based on Key. Key formula is optional. If not specified, all the elements in the list are deleted.

Syntax

```
LIST DELETE : <List Variable Specification> [ : <Key Formula>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be any expression which evaluates to a unique string value. It is optional



Action LIST REMOVE is an alias for the action LIST DELETE.

Deleting elements from Simple List Variable using LIST DELETE

- Deleting a single element from a Simple List Variable

```
LIST DELETE : SLV Emp : "E001"
```

The element identified by key 'E001' will be deleted from the Simple List Variable **SLV Emp**.

- Deleting all elements from a Simple List Variable

```
LIST DELETE : SLV Emp
```

Since key formula is not specified, all elements from simple list variable **SLV Emp** will be deleted.

Deleting elements from a Compound List Variable using LIST DELETE

- Deleting an element from a Compound List Variable

```
LIST DELETE : CLV Emp : "E001"
```

The element identified by key '**E001**' will be deleted from the Compound List Variable "CLV Emp".

- Deleting all elements from a Compound List Variable

```
LIST DELETE : CLV Emp
```

Since key formula isn't specified, all elements from compound list variable **CLV Emp** are deleted.

- **Action - LIST DELETE EX**

This action is used to delete an element from the list based on index. INDEX formula is optional. If not specified, all the elements in the list are deleted. A negative index denotes reverse position.

Syntax

```
LIST DELETE EX : <List Variable Specification> [:<Index Formula>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Index Formula> can be any expression which evaluates to an index number. It is optional.



Action LIST REMOVE EX is an alias for the action LIST DELETE EX.

Deleting Elements from Simple List Variable using LIST DELETE EX

- Deleting a single element from a Simple List Variable

```
LIST DELETE EX : SLVEmp :2
```

The element identified by index number '2' will be deleted from Simple List Variable **SLV Emp**.

- Deleting all elements from a Simple List Variable

```
LIST DELETE EX : SLVEmp
```

Since index formula is not specified, all elements from Simple List Variable **SLV Emp** are deleted.

Deleting elements from a Compound List Variable using LIST DELETE EX

- Deleting an element from a Compound List Variable

```
LIST DELETE EX : CLVEmp : 10
```

The element identified by index '10' will be deleted from the Compound List Variable **CLV Emp**.

- Deleting all elements from a Compound List Variable

```
LIST DELETE EX : CLVEMP
```

Since index formula isn't specified, all elements of compound list variable **CLV EMP** are deleted.

3. Expanding Elements in the List Variable

- **Action - LIST EXPAND**

The Action LIST EXPAND is used to create the specified number of blank elements and insert them into the list. All these elements are created without a key. If key specification is required for each element, then either LIST FILL or a loop can be used to add elements individually.

Syntax

```
LIST EXPAND : <List Variable Specification> : <Count Formula>
```

Where,

<List Variable Specification> is the Simple List or Compound List variable specification.

<Count Formula> can be any expression which evaluates to a number.

Example:

Expanding Simple List Variable using LIST EXPAND

```
LIST EXPAND : SLVEMP : 10
```

Here, count formula is 10. Hence, 10 blank elements are added to Simple List Variable 'SLVEMP'.

Expanding Compound List Variable using LIST EXPAND

```
LIST EXPAND : CLVEMP : 5
```

Here, count formula is 5. Thus, 5 blank elements are added to the Compound List Variable 'CLVEMP'.

```
LIST EXPAND : CLVEMP[1].Relatives : 10
```

Here, count formula is 10. Hence, 10 blank elements are added to Compound List Variable 'Relatives'. 'Relatives' is a Compound List Member variable of the Compound List Variable 'CLVEMP'.

Value Specifications

The value for the Simple/List Variables (Simple & Compound) can be specified using Attributes at Report and Form Level, and using Actions in User Defined Functions.

Value specification at Report Level

The attributes SET and PRINTSET are used to specify the variable values at Report Level.

□ Attribute - SET

The Report attribute SET can be used to specify a variable name and its value, which will be set during the startup of the report.

Syntax

```
SET : <Variable Specification> : <Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be any expression, which evaluates to a value for the variable of the specified data type.

Example:

;; Setting value to a Simple Variable

```
SET : Var : "ABC"
```

;; Setting value to a Simple List Variable element

```
SET : ListVar[1] : "XYZ"
```

;; Setting value to Compound List Variable element's member

```
SET : CLVEMP[1].Name : "Kumar"
```

□ Attribute - PRINT SET

The Report attribute **Print Set** is similar to the SET attribute but sets the value of the variables to the specified value when the report is started in Print mode.

Syntax

```
PRINT SET : <Variable Specification> : <Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be any expression which evaluates to a value for the variable of the specified data type.

Example:

;; Setting value to a Simple Variable

```
PRINTSET : Var : "ABC"
```

;; Setting value to a Simple List Variable element

```
PRINTSET : ListVar[1] : "XYZ"
```

;; Setting value to Compound List Variable element's member

```
PRINTSET : CLVEMP[1].Name : "Kumar"
```

Value specification at Form Level

□ Attribute - SET

The Form attribute SET is similar to the Report attribute SET, the difference being that while the report sets the value once in its lifetime, the form SET is executed during every regeneration/refresh of the report.

Syntax

```
SET : <Variable Specification> : <Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be any expression, which evaluates to a value for the variable of the specified data type.

Example:

;; Setting value to a Simple Variable

```
SET : Var : "ABC"
```

;; Setting value to a Simple List Variable element

```
SET : ListVar[1] : "XYZ"
```

;; Setting value to Compound List Variable element's member

```
SET : CLVEMP[1].Name : "Kumar"
```

Value specification at Function Level

Actions SET, MULTISSET, EXCHANGE, INCREMENT and DECREMENT are used.

□ Action - SET

Values of variables can be set / updated via the SET action. This action is available as a global action, and can be used within a function also.

List variables and compound variables cannot have values; they can have only element/member variables inside them, respectively. If SET action is used on compound variables, the value will be set to the FIRST member variable. If the first member variable is again compound, the program would search for the first non-compound leaf member and set the value.

For list variables, the value is treated as the count, and the list is expanded by the number of elements provided in the expression.

Syntax

```
SET : <Variable Specification> : <Value Expression>
```

Where,

<Variable Specification> is the variable path specification.

<Value Expression> can be any expression which evaluates to a value for the variable of the specified data type.

Example:

;; Setting value to a Simple Variable

```
SET : Var : "ABC"
```

;; Setting value to a Simple List Variable element

```
SET : SLVEMP[1] : "XYZ"
```

;; Setting value to Compound List Variable element's member

```
SET : CLVEMP[1].Name: "Kumar"
```

□ **Action - MULTISSET**

The action MULTI SET is used to set the values of compound member variables in one call. All member specifications are relative to the compound variable specification given.

Syntax

```
MULTI SET : <CompoundVariable Specification> + : <Member Specification : Value> [, <Member Specification : Value>, ...]
```

Where,

<Compound Variable Specification> is the Compound Variable specification.

<Member Specification : Value> is the list of name-value pairs for setting member values.

Example: 1

```
MULTISET : CLVEMP[1] : Name : "Vimal",Age : 26, Salary :($$AsAmount : 10000)
```

All member variables of 1st element of Compound List Variable **CLVEMP** are set with MULTISSET.

Example: 2

```
MULTISET : CLVEMP[1].Relatives[1] : Name : "Hari", Age : 20, +
Relation:"Brother"
```

Here, all member variables for the first element of the Compound List Variable Relatives are set.

Relatives is a Compound List Member variable of the Compound List Variable **CLVEMP**.

□ **Action - EXCHANGE**

This action is used to swap the values of two variables, provided both belong to the same data type. This cannot be done for Simple List or Compound List as a whole. However, values of elements of Simple List and Compound List member variables having same data type can be exchanged.

Syntax

```
EXCHANGE : <First Variable Specification> : <Second Variable + Specification>
```

Where,

<First Variable Specification> is the simple variable specification.

<Second Variable Specification> is the simple variable specification.

Exchanging value of a Simple Variable with another Simple Variable

```
EXCHANGE : EmpVarOld : EmpVarNew
```

Both the variables are of 'String' data type. The value of the variable **EmpVarOld** is exchanged with that of the variable **EmpVarNew** on execution of the action.

Exchanging value of an element of Simple List Variable with that of another Simple List Variable

```
EXCHANGE : SlvEmpOld[1] : SlvEmpNew[1]
```

The value of the first element of **SlvEmpOld** is exchanged with that of the first element of **SlvEmpNew**. Both the Simple List Variables are of 'String' data type

Exchanging value of a Simple variable with a member variable of a Compound List variable

```
EXCHANGE : EMP Salary : CLVEmp[1].Salary
```

The value of a variable **Emp Salary** is exchanged with that of the member variable 'Salary' of the Compound List Variable **CLVEmp**. Both the simple variables are of 'String' data type.

□ Action - INCREMENT

INCREMENT is a special action provided in 'Function' scope to increment values of the variable. This is supported only on simple variables of type Number.

Syntax

```
INCREMENT : <Simple Variable Specification> [:<NumIncrement Expression>]
```

Where,

<Simple Variable Specification> is the simple variable specification.

<NumIncrement Expression> is an expression which evaluates to a number. Based on this, the variable value is incremented. It is optional. If not specified, variable value is incremented by 1.

Example:

```
INCREMENT : Counter ;; Incrementing the variable value by 1
```

```
INCR : Counter : 2 ;; Incrementing the variable value by 2
```



Action INCR is an alias for the action INCREMENT.

□ Action - DECREMENT

Decrement is a special action provided in 'Function' scope to decrement values of the variable. It is supported only on simple variables of type Number.

Syntax

```
DECREMENT : <Simple Variable Specification> [:< NumIncrementExpression>]
```

Where,

<SimpleVarSpecification> is the simple variable specification.

<NumIncrementExpression> is an expression evaluating to a no., based on which, the variable value is decremented. It is optional. If not specified, the variable value is decremented by 1.



Action DECR is an alias for the action DECREMENT.

Example:

;; Decrementing the variable value by 1

```
DECREMENT : Counter
```

;; Decrementing the variable value by 2

```
DECR      : Counter : 2
```

Value Modification at Field Level

- **Attribute - MODIFIES**

The Field attribute 'Modifies' is used to modify the value of the variable.

Syntax

```
Modifies : <Variable Specification> [:<Logical Flag>]
```

Where,

<Variable Specification> is the variable path specification.

<Logical Flag> can be a logical value TRUE/FALSE. TRUE would set the value after the field's acceptance, while FALSE will set it during the acceptance of the report having the field.

Example:

```
[Field : EMP Age]
```

```
Modifies : EMPAgeVar : Yes
```

Here, value of the variable EMPAgeVar will be modified with the value stored/keyed in the field EMP Age after the field's acceptance.

Retrieving value from List

- **Function - \$\$ListValue**

It is used to retrieve the value of an element in the list for a given key. If the list is of compound variables, an optional member specification can be given to extract value of a specific member.

Syntax

```

$$ListValue : <List Variable Specification> : <Key Formula> +
                [:<Member Specification>]
  
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be any expression which evaluates to a string value.

<Member Specification> is required only if the value needs to be extracted from a specific member of a Compound List Variable.

Example:**Retrieving value from Simple List Variable using \$\$ListValue**

```

$$ListValue : SLVEMP : "E001"
  
```

In this example, the function returns the value of the element identified by the key 'E001' from the simple list variable 'SLV Emp'.

```

$$ListValue : SLVEMP : ##KeyVar
  
```

In this example, the variable 'KeyVar' holds the key value. The function returns the value of the element identified by the key from the simple list variable 'SLV Emp'.

Retrieving value from Compound List Variable using \$\$ListValue

```

$$ListValue : CLVEmp : ##KeyVar : Age
  
```

In this example, the variable 'KeyVar' holds the key value. The function returns the identified Compound List Variable element's member variable value. In this case, the member specification has been specified as 'Age'.

- **Function - \$\$ListValueEx**

The Function **\$\$ListValueEx** returns the value of an element at the specified index in the list.

Syntax

```

$$ListValueEx : <List Variable Specification>:<Index Formula> +
                [:< Member Specification>]
  
```

Where,

<List Variable Specification> is the Simple or Compound List Variable specification.

<Index Formula> can be any expression which evaluates to an index number.

<Member Specification> is required only if the value needs to be extracted from a specific member of a Compound List Variable.

Example:**Retrieving value from Simple List Variable using \$\$ListValueEx**

```

$$ListValueEx : SLVEmp : ##IndexVar
  
```

In this example, the variable 'IndexVar' holds the index value. The function returns the value of the element identified by the index from the simple list variable 'SLV Emp'.

Retrieving value from Compound List Variable using \$\$ListValueEx

```
$$ListValueEx : CLVEmp : ##IndexVar : Age
```

Here, variable 'KeyVar' holds the index value. The function returns the identified Compound List Variable element's member variable value. Here, the member specified is 'Age'.

□ Index Based Retrieval using ## Operator

The operator ## is used to access the value of the variable. It also allows dotted notation syntax to access variables/member variables/element variables of a list at any level.

When ## is used on a compound variable (without path specification), it returns the value of the first member variable, by default. Similarly, on a list variable, it returns the no. of items in the list.

Syntax

```
##<Element Variable Specification>.<Member Variable Specification>.+
  <Simple Member Value specification>
```

Where,

<Element Variable Specification> can be a Compound Variable or Compound List Variable [Index Expression].

<Member Variable Specification> can be a Compound Variable Member or Compound List Member Variable [Index Expression].

<Simple Member Value Specification> refers to the name of a simple member in specified path.

<Index Expression> is an expression that evaluates to a no. Suffixing a variable with index refers to an element variable. It can be positive or negative. Negative index denotes reverse access.

Example:

Retrieving Value from Simple List Variable using ## Operator

```
SET : TempVar : ##SLVEMP[3]
```

Value of element in SLVEMP, identified by the index '3', will be set to the variable 'TempVar'.

Retrieving Value from Compound List Variable using ##Operator

```
LOG : ##CLVEmp[2].Relatives[1].Name
```

Here, we are retrieving value of the identified Compound List Variable (Relatives) element's member variable value. 'Relatives' is a member variable of the Compound List Variable CLVEMP.

Looping Construct – For In/For Each

The FOR IN loop is used to iterate over the values in the list variable. The number of iterations depends on the number of items in the list variable.

Syntax

```
FOR IN : <Iterator Variable> : <List Variable Name >
.
.
END FOR
```

Where,

<Iterator Variable> is the name of the variable which holds the Key value in every iteration.

<List Variable Name> is the name of the Simple List or Compound List Variable.

This construct will walk only the elements in the list which are having a key. Since the iterator variable is filled with a key for each element, all elements which do not have a key are ignored. This is useful to walk keyed list variable elements in the current sorting order. If the element does not have a key, then other loops like WHILE, FOR, etc., can be used and the elements can be operated via index.

Example:

Iterating the Simple List Variable Values

```
FOR IN : KeyVar : SLV Emp
    LOG : $$ListValue : SLVEmp : ##KeyVar
END FOR
```

Here, the iterator variable “KeyVar” holds the Key value in every occurrence of the iteration. In every iteration, the value of the element identified by the key is logged using the function **\$\$ListValue**.

Iterating the Compound List Variable Values

```
FOR IN : KeyVar : CLV Emp
    LOG : $$ListValue : CLVEmp : ##KeyVar : Age
END FOR
```

Here, the iterator variable “KeyVar” holds the Key value in every iteration. In every iteration, the value of the member “Age” of the element of “CLVEMP” identified by the key is logged using the function **\$\$ListValue**.



The looping construct FOR EACH is an alias for the looping construct FOR IN.

List Variable Specific Functions

□ Function - \$\$ListKey

The function **\$\$ListKey** returns the corresponding key for the given index.

Syntax

```
$$ListKey : <List Variable Specification> : <Index Specification>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Index Specification> can be any expression which evaluates to a number.

Example:**Retrieving key from a Simple List Variable using \$\$ListKey**

```
01 : LOG : $$ListKey : SLVEMP : 2
```

In this example, the function `$$ListKey` retrieves the Key of the second element of the Simple List Variable 'SLVEMP'.

Retrieving key from a Compound List Variable using \$\$ListKey

```
02 : LOG : $$ListKey : CLVEmp[1].Relatives : 1
```

Here, key of first element of Compound List Variable 'Relatives' is retrieved. 'Relatives' is a member of Compound List Variable 'CLVEMP'.

- **Function - \$\$ListIndex**

The function `$$ListIndex` returns the Corresponding index for the given Key.

Syntax

```
$$ListIndex : <List Variable Specification> : <Key Specification>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Specification> can be any expression which evaluates to a string value.

Example:**Retrieving index from a Simple List Variable using \$\$ListIndex**

```
01 : LOG : $$ListIndex : SLVEMP : E001
```

Here, index of the element identified by the key value 'E001' is retrieved from 'SLVEMP'.

Retrieving index from a Compound List Variable using \$\$ListIndex

```
02 : LOG : $$ListIndex : CLVEmp : E001
```

Here, index value of the element identified by the key value 'E001' is retrieved from 'CLVEMP'.

- **Function - \$\$ListCount**

The function `$$ListCount` retrieves the number of items in the list.

Syntax

```
$$ListCount : <List Variable Specification>
```

Where,

<ListVariable Specification> is the Simple List or Compound List Variable specification.

Example:

```
01 : LOG : $$ListCount : SLVEMP
```

```
02 : LOG : $$ListCount : CLVEMP
```

- **Function - \$\$ListFind**

It is used to check if a given key exists in the list or not. It returns a logical flag as a result.

Syntax

```
$$ListFind : <List Variable Specification> : <Key Formula>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Key Formula> can be any expression which evaluates to a string value.

Example:

```
01 : LOG : $$ListFind : SLVEMP : E001
```

```
02 : LOG : $$ListFind : CLVEMP : E001
```

▣ **Function - \$\$ListValueFind**

This function can be used to check if a given value exists in the list. If a given list has more than one same value, the index can be used to retrieve the n'th matching value.

Syntax

```
$$ListValueFind : <List Variable Specification> : < Occurance +  
Specification> : <Value Formula> [:<Member Specification>]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Occurance Specification> can be any expression which evaluates to a number.

<Value Formula> can be any expression which evaluates to a value.

<Member Specification> can be specified if the list element is compound. It is optional.

Example:

;; Finding value from the Simple List Variable

```
01 : LOG : $$ListValueFind : SLVEMP: 1 : RAMESH
```

;;Finding value from the Compound List Variable with member specification

```
03 : LOG : $$ListValueFind : CLVEmp : 1 : PRIYA : Name
```

The function will return YES if the value exists in the list, else it will return NO.

Populating a List from a Collection

▣ **Action - LIST FILL**

It is used to fill a list from a collection instead of using the looping constructs. The specified collection is walked and the key formula and value formula is evaluated in the context of each object to create list elements.

Syntax

```
LIST FILL : <List Variable Specification> : <CollectionName> +  
[:<Key Formula> [:<Value Formula> [:<Member Specification>]]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Collection Name> is the name of the collection from which the values need to be fetched to fill the list variable.

<Key Formula> can be any expression which evaluates to string value. It is optional.

<Value Formula> can be any expression which returns a value. The data type of the value must be same as that of the List variable. Value formula is optional. If not specified, only KEY is set for each added element.

<Member Specification> can be given if the list contains a compound variable



If both key and value are not specified, blank elements are added to the list.

Example:

Populating a Simple List Variable from a Collection

```
LIST FILL : SLV Emp : Employees : $Name : $Name
```

All the employee names from the collection 'Employees' will be added to the Simple List Variable, once the action LIST Fill is executed.

Populating a Compound List Variable from a Collection

```
LIST FILL : CLV Emp : Employees : $Name : $Name
```

In this example, all the employee names from the collection 'Employees' will be added to the first member variable, as there is no member specification.

```
LIST FILL : CLV Emp : Employees : $Name : $Designation: Designation
```

In this example, Designations of all the employees from the collection 'Employees' will be added to the member variable 'Designation'.

```
LIST FILL : CLV EMP[1].Relatives:Employees : $Name : $SpouseName : Name
```

Spouse name of all employees from the collection 'Employees' will be added to member variable 'Name' of Compound List Variable 'Relatives'. 'Relatives' is a member variable of 'CLVEMP'.

Sorting of List Elements

Initially, when the list variable is created, it is sorted on the order of insertion. TDL provides the facility to sort the values in the list variable based either on key or on value. The following actions allow changing the sort order:

- List Key Sort
- List Value Sort
- List Reset Sort

□ Action - LIST KEY SORT

The action LIST KEY SORT allows the user to sort the elements of the list based on the key.

Keys are by default of type 'String'; so, the absence of key data type specification will consider key data type as String while sorting. The user can override this by specifying a key data type. Keys are optional for elements. All elements in the list may not have a key. In such cases, comparisons of elements would be done based on the insertion order.

Syntax

```
LIST KEY SORT : <List Variable Specification>
                [:<Ascending/DescendingFlag> [:<Key Datatype>]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Ascending/DescendingFlag> can be YES/NO. YES is used to sort the list in ascending order and NO for descending. If the flag is not specified, then the default order is ascending.

<Key Data Type> can be String, Number, etc. It is optional.



The action LIST SORT is an alias for the action LIST KEY SORT.

Example:

Sorting Simple List based on Key

```
LIST KEY SORT : SLVEmp : Yes : String                ;;Ascending Order
LIST KEY SORT : SLVEmp : No  : String                ;;Descending Order
```

Sorting Compound List based on Key

```
LIST KEY SORT : CLVEmp : Yes : String                ;;Ascending Order
LIST KEY SORT : CLVEmp[1].Relatives : No : String ;;Descending Order
```

□ Action - LIST VALUE SORT

The action LIST VALUE SORT allows the user to sort the elements of the list based on value. The data are sorted as per the data type specified for the list variable in case of simple list, and the member specification data type in case of compound list. If a compound list is chosen and member specification is not specified, then the list is sorted by value of the first member variable.

If duplicate values are in the list, the key data type passed is considered to sort by key, and then in absence of key, insertion order is used.

Syntax

```
LIST VALUE SORT : <List Variable Specification> [:<Ascending/Descending +
                Flag> [:<Key Datatype> [:<Member Specification>]]]
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

<Ascending/DescendingFlag> can be YES/NO. YES is used to sort the list in ascending order and NO for descending. If the flag is not specified, then the default order is ascending.

<Key Data Type> can be String, Number, etc. It is optional.

<Member Specification> is the member specification in case of compound list. If not specified, the list is sorted by the value of first member variable.

Example:

Sorting Simple List based on Value

;;Ascending Order

```
LIST VALUE SORT : SLVEmp : Yes : String
```

;;Descending Order

```
LIST VALUE SORT : SLVEmp : No : String
```

Sorting Compound List based on Value

;;Ascending Order

```
LIST VALUE SORT : CLVEmp : Yes : String
```

;;Descending Order

```
LIST VALUE SORT : CLVEmp[1].Relatives : No : String
```

□ Action - LIST RESET SORT

This action resets the sorting method of the list and brings it back to the insertion order.

Syntax

```
LIST RESET SORT : <List Variable Specification>
```

Where,

<List Variable Specification> is the Simple List or Compound List Variable specification.

Example:

```
LIST RESET SORT : SLVEMP LIST RESET SORT : CLVEMP
```

4.6 Some Common Functions Used

□ Function - \$\$IsSysNameVar

This function checks if the variable has a value which is a SysName like 'Not Applicable', 'End of List', etc. In case of repeated variables, if any one value is a non-sysname, it returns FALSE.

Syntax

```
$$IsSysNameVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
$$IsSysNameVar : EmpVar
```

Here, \$\$IsSysNameVar returns YES if variable **EmpVar** has Sysname as value, else returns NO.

□ Function - **\$\$IsDefaultVar**

This function determines if the content of the variable has a “Default” or blank as the value. This function is applicable only for Simple variables. In case of simple repeated variable, if any one value is non-default, then this is not a default variable, and the function returns NO.

Syntax

```
$$IsDefaultVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Field : DefaultVar]
```

```
Set as : $$IsDefaultVar : SVValuationMethod
```

\$\$IsDefaultVar returns YES if value of **SVValuationMethod** is blank or Default, else returns NO.

□ Function - **\$\$IsActualsVar**

This function checks if the content of the variable is blank or sysname or “ACTUALS”.

Syntax

```
$$IsActualsVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
$$IsActualsVar : SVBudget
```

YES is returned if the value of variable **SVBudget** is Blank or Sysname or “ACTUALS”, else NO.

□ Function - **\$\$IsCurrentVar**

This function checks if the content of the variable is Blank or Sysname or “Stock in hand”.

Syntax

```
$$IsCurrentVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
$$IsCurrentVar : DSPOrderCombo
```

YES is returned if value of **DSPOrderCombo** is Blank or Sysname or Stock-In-Hand, else NO.

□ Function - **\$\$ExecVar**

This function returns the value of a variable in the parent report chain.

Syntax

```
$$ExecVar : <Variable Specification>
```


Where,

<Variable Specification> is the simple variable path specification.

Example:

```
$$ExecVar : DSPShowMonthly
```

Function \$\$ExecVar returns the value of the variable DSPShowMonthly from the parent report.

□ **Function - \$\$FieldVar**

This function returns the value of the field which is acting as a variable with the specified name.

Syntax

```
$$FieldVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Collection : GodownChildOfGodownName]
```

```
    Type      : Godown
```

```
    Child of  : $$FieldVar : DSPGodownName
```

In this example, \$\$FieldVar is used to fetch the value of the variable DSPGodownName whose value is modified in a field. This value becomes the value for the 'ChildOf' attribute.

□ **Function - \$\$ParentFieldVar**

This function gets the field variable value from its parent report.

Syntax

```
$$ParentFieldVar : <Variable Specification>
```

Where,

<Variable Specification> is the simple variable path specification.

Example:

```
[Field : ParentFieldVar]
```

```
    Set as   : $$ParentFieldVar : SVStockItem
```

Here, the function returns field variable value from its parent report for the variable "SVStockItem".

4.7 Field Acting as a Variable

The 'Variable' attribute in a 'Field' Definition is used to make the field behave as a variable, with the specified name. The variable need not be defined as it inherits data type from the field itself. Field can act as a simple variable only, since it can hold only simple value.

Syntax

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

```
    Variable : <Variable Name>
```

Where,

<Field Name> is the name of the field.

<Variable Name> is the name of the variable.

Example:

```
[Field : EmployeeName]
```

```
Variable : EmpNameVar
```

4.8 Implication of Repeat Variables in Columnar Report

The report in which a number of columns can be added or deleted as per the user inputs, is referred to as Columnar Report. In a Columnar Report, Lines are repeated vertically and Fields are repeated horizontally. The Columnar Report can be a:

- **MultiColumn Report** - Column can be repeated based on the user inputs.
- **AutoColumn Report** - Multiple columns can be repeated based on the user input, on the single click of a button.
- **Automatic Auto Columns** - Report can be started with predefined multiple columns without user intervention.

The Attribute 'Repeat' – Variable, Report and Line

Let us see the implications of Repeat Attribute of Variable / Report / Line Definitions in context of Columnar Reports.

1. 'Repeat' Attribute of Variable definition

Please refer to the topic "Variable Definition and Attributes".

2. 'Repeat' Attribute of Report definition

The **Repeat** Attribute of 'Report' definition is used specifically in Columnar Reports. When we specify 'Repeat' attribute with a variable name, the report becomes a Columnar Report and the number of columns depends upon the values stored in the variable. Only simple variables can be repeated. Also, a report can have more than one variables repeated. In such cases, the number of columns in the report depends on the maximum value a Repeat Variable holds.

The 'Repeat' attribute of the report is declaration cum repeat specification; so a separate declaration is not required. Even if a declaration is done using 'Variable' attribute, 'Repeat' is considered as a repeat specification.

Syntax

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

Where,

<Report Name> is the name of the Report.

<Variable Names> is the comma-separated list of variables.

3. Repeat Attribute of Line Definition

The 'Repeat' Attribute of 'Line' Definition is used to repeat the Field horizontally in columns.

Syntax

```
[Line : <Line Name>] Repeat : <Field Name>
```

Where,

<Line Name> is the name of the line.

<Field Name> is the name of the Field which needs to be repeated.

Example:

Let us look into the usage of 'Repeat' Attribute at Variable/Report/Line Definitions in designing the Columnar Stock Item-wise Customer-wise Sales Report.

In this report, Stock Item names should be repeated vertically and Customer/Party names horizontally. The columns should be automatically available when the report is started.

Repeat Attribute of Variable Definition

```
[Variable : PName]

    Type      : String

    Repeat    : ##DSPRepeatCollection
```

The variable 'DSPRepeatCollection' holds the collection name 'CFBK Party'. This collection contains a method name 'PName'. In this case, the variable 'PName' would be filled with the method value from each object of the collection "CFBK Party".

```
[Collection : CFBK Party]

    Source Collection : CFBK Voucher

    Walk              : Inventory Entries

    By                : PName : $PartyLedgerName

    Aggr Compute      : BilledQty : SUM: $BilledQty

    Filter            : NonEmptyQty
```

Variable 'PName' holds multiple values based on implicit index. Method value of each object of collection 'CFBK Party' is picked up and stored in the variable's 1st index, 2nd index, and so on.

'Repeat' Attribute of Report Definition

```
[Report : CFBK Rep]

    Use          : DSP Template

    Form         : CFBK Rep

    Variable    : DoSetAutoColumn, PName Repeat : PName

    Set         : DoSetAutoColumn : Yes

    Set         : DSPRepeatCollection : "CFBK Party"

    Set         : SVFromDate : $$MonthStart : ##SVCurrentDate

    Set         : SVToDate   : $$MonthEnd   : ##SVCurrentDate
```

The attribute “Repeat” determines that it is a Columnar Report. The number of columns depends on the number of values available in the variable “PName”.

Repeat Attribute of Line Definition

```
[Line : CFBK Rep Details]
```

```
Fields : CFBK Rep Name, CFBK Rep Party, CFBK Rep Col Total
```

```
Repeat : CFBK Rep Party
```

```
Total : CFBK Rep Party
```

Field ‘CFBK Rep Party’ is repeated based on the no. of values of variable (NumSets). So, those many numbers of instances of the field are created. Each field will have an implicit index number (starting from 1). This implicit index is used to evaluate expressions in the context of the field.

Common Functions used with Columnar Reports

□ Function - \$\$NumSets

It returns the number of columns in the report. It does not take any parameter. If the report is an auto report or sub report, it returns the number of columns in the parent of the auto/sub report. Number of set is the maximum number of values a repeated variable can hold in that report.

Syntax

```
$$NumSets
```

Example:

```
[Field : CFBK Rep Col Total]
```

```
Use : Qty Primary Field
```

```
Set as : $$Total : CFBKRepParty
```

```
Border : Thin Left
```

```
Invisible : $$Numsets=1
```

In this example, the ‘Total’ column will be invisible if there is only one column in the report.

□ Function - \$\$LowValue

This function can be used to get the lowest value in a set of values in the repeated variable.

Syntax

```
$$LowValue : <Variable Specification>
```

Where,

<Variable Specification> is a simple variable specification.

Example:

Let us suppose that the Repeat Variables in a Columnar Report are **SVFromDate** and **SVToDate**. Consider the following Field Definition in the same report:

```
[Field : VariableLowValue]
```

```
Use      : Name Field
```

```
Set as   : $$LowValue : SVFromDate
```

\$\$LowValue returns the lowest value in a set of values in the repeat variable **SVFromDate**.

▫ **Function - \$\$HighValue**

This function can be used to get the highest value in a set of values in the repeated variable.

Syntax

```
$$HighValue : <Variable Specification>
```

Where,

<Variable Specification> is a simple variable specification.

Example:

Suppose that the Repeat Variables in a Columnar Report are **SVFromDate** and **SVToDate**. Consider the following Field definition in the same report:

```
[Field : VariableHighValue]
```

```
Use      : Name Field
```

```
Set as   : $$HighValue : SVToDate
```

\$\$HighValue returns the highest value in a set of values in the repeat variable **SVToDate**.

▫ **Function - \$\$IsCommon**

This function is used with repeated variable to check if all the values in the repeat set are same.

Syntax

```
$$IsCommon : <Variable Specification>
```

Where,

<Variable Specification> is a simple variable specification.

Example:

Suppose the Repeat Variable in a columnar report is **SVCurrentCompany**. Consider the following Field Definition in the same report:

```
[Field : VariableIsCommon]
```

```
Use      : Logical Field
```

```
Set as   : $$IsCommon : SVCurrentCompany
```

\$\$IsCommon returns YES if all values in **SVCurrentCompany** are same, otherwise returns NO.

▫ **Function - \$\$VarRangeValue**

This function gets a list of variable values, separated by the specified separator character. If no separator character is specified, comma (,) is taken as the separator character by default.

Syntax

```

$$VarRangeValue : <Variable Specification> [:<Separator Character>
[:<Start Position> [:<End Position>]]]

```

Where,

<Variable Specification> is the simple variable specification.

<Separator Character> is the separator character.

<Start Position> is the index which denotes the starting position.

<End Position> is the index which denotes the ending position.



Specifying Start and End Positions is optional. If not specified, the function will return all the values of the specified Repeat variable separated by comma(,)

If Start and End Positions are specified, the function will return the values of repeat variable within the Specified index Range. Again, specifying End Position is optional. If the End Position is not specified, the function will return the entire set of values from the starting position.

Example:

```

$$VarRangeValue : SVFromDate

```

In this example, the function returns the entire set of values of the Repeat Variable SVFromDate.

```

$$VarRangeValue : SVFromDate:",":1:5

```

Here, the function returns the value of specified index range (1 to 5) of the Repeat Variable SVFromDate

```

$$VarRangeValue : SVFromDate:",":3

```

The entire set of values from Starting Index position of the repeat variable SVFromDate are returned.

4.9 Variables Usage and Behaviour in Auto Report

A report can be marked as an auto report via **AUTO** attribute, which indicates the system that the report cannot instantiate its own variables. It will inherit variables from parent scope. It is mainly used for configuration reports which require modifying configuration variables of parent report.

Syntax

```

[Report : <Report Name>] Auto : <Logical Value>

```

Where,

<Report Name> is the name of the report.

<Logical Value> can be YES / NO. The default value is NO.

Example:

```

[Report : Voucher Configuration]

```

```
Auto : Yes
```

```
Title : $$LocaleString : "Voucher Configuration"
```

This is a default configuration report marked as Auto report, to modify variables of parent report. A report can be launched in 'Auto' mode using Actions **Modify Variable** and **Modify System**

Actions MODIFY VARIABLE and MODIFY SYSTEM

□ Action - MODIFY VARIABLE

It launches the given report in 'auto' mode. Since the launched report is in 'auto' mode, it cannot have its own instance of variables and any modification would affect the parent context.

Syntax

```
MODIFY VARIABLE : <Report Name>
```

Where,

<Report Name> is the name of the report which is to be launched in 'Auto Mode'.

Example:

```
[Button : F2 Change Period]
```

```
Key : F2
```

```
Action : Modify Variables : Change Period
```

```
Title : $$LocaleString : "Period"
```

The Action 'Modify Variable' launches the report 'Change Period' in 'Auto' Mode. The report is having two fields SVFromDate and SVToDate

```
[Field : SVFromDate]
```

```
Use : Short Date Field
```

```
Modifies : SVFromDate
```

```
Variable : SVFromDate
```

```
[Field : SVToDate]
```

```
Use : Short Date Field
```

```
Format : Short Date, End : #SVFromDate
```

```
Modifies : SVToDate
```

```
Variable : SVToDate
```

The variable value changes would affect the parent report context only (i.e., it will affect values of the variables **SVFromDate** and **SVToDate**, which are associated to the report, from which the report **Change Period** is launched in Auto Mode).

□ Action - MODIFY SYSTEM

The action **MODIFY SYSTEM** launches the given report in 'auto' mode. Even if the report is called under some other report context, this action makes the new report to get the system context and thereby modify the system scope variables.

Syntax

```
MODIFY SYSTEM : <Report Name>
```

Where,

<Report Name> is the name of the report which is to be launched in 'Auto Mode'.

Example:

```
[Button : Change System Period]
```

```
Key : Alt+F2
```

```
Action : Modify System : Change Menu Period
```

```
Title : $$LocaleString : "Period"
```

The Action 'Modify System' has launched the report 'Change Menu Period' in 'Auto' Mode. The report is having two fields SVFromDate and SVToDate

```
[Field : SVFromDate]
```

```
Use : Short Date Field
```

```
Modifies : SVFromDate
```

```
Variable : SVFromDate
```

```
[Field : SVToDate]
```

```
Use : Short Date Field
```

```
Format : Short Date, End : #SVFromDate
```

```
Modifies : SVToDate
```

```
Variable : SVToDate
```

The value changes would affect the variables at system scope, as the report is launched using the Action 'Modify System'.

4.10 Repeat Line with Optional Collection

We are aware that the 'Repeat' Attribute of a Part is used to Repeat a line over a Collection.

Existing Syntax

Syntax:

```
[Part : <Part Name>]
```

```
Repeat : <Line Name> : <Collection>
```


Where,

<Part Name> is the name of the part.

<Line Name> is the name of the line to be repeated.

<Collection> is the name of the collection on which the line is repeated. This was mandatory prior to this release. In this case, the same line will be repeated for each object of the collection. Each line will be associated with an Object of the collection. Report created in Create/Alter/Display mode will either store method values into the object or fetch method values from the Object. Any expression evaluation within this line will happen with an object in context.

With the introduction of List Variable (Simple/Compound), there will be a requirement to store values into the Variable by accepting user inputs and also to display or use it for expression evaluation. Since Variables are Context free structures there is no need to associate element variables with the line. For this purpose the 'Repeat' Attribute of the part has been enhanced to have the collection as Optional. Now, it is possible to Repeat a Line with or without a Collection. In cases where the collection is not specified, the number of lines to be repeated is unknown. Hence, specifying the SET attribute is mandatory. In case of Edit, SET can be optional if 'Break On' is specified.

New Enhanced Syntax

Syntax

```
[Part : <Part Name>]
  Repeat : <Line Name> [: <Collection>]
```

Where,

<Part Name> is the name of the part.

<Line Name> is the name of the line to be repeated.

<Collection> is the name of the collection on which the line is repeated. It is now OPTIONAL.

Storing Values into List Variables

With this enhancement, values can be added to List Variable (Simple/Compound) dynamically by accepting user inputs by repeating a line without a Collection. Multiple lines can be added dynamically or a fixed number of lines can be added as per user requirement, while repeating the line.

Example:

To accept the values from a user to the Simple List Variables SLVEMP, a report is opened in 'Create' Mode. Let us look into the 'Part' Definition:

```
[Part : SLV List Values]

  Lines   : SLV List Title, SLV List Values

  Repeat  : SLV List Values

  BreakOn : $$IsEmpty : #SLVAlias

  Scroll  : Vertical
```

Here, the line is repeated without a collection and it will break if the field value 'SLV Alias' is empty. Let us look into the Field Definitions:

```
[Line : SLV List Values]

  Fields : SLV Alias, SLV Name

  [Field : SLV Alias]

    Use : Name Field

  [Field : SLV Name]

    Use      : Name Field

    Delete   : Key

    Add      : Key : SLV List Key

    Inactive: $$IsEmpty:#SLVAlias

[Key: SLV List Key]

  Key      : Enter

  Action List : Field Accept, SLV List Add

[Key : SLV List Add]

  Key      : Enter

  Action : LIST ADD : SLVEMP : #SLVAlias : #SLVName
```

Values are added to the List Variable "SLVEMP" using the Action "LIST ADD". Similarly, user inputs can be added / altered dynamically to the Compound List Variable also.

Retrieving Values from List Variables

In the previous example, we had stored values into a Simple List Variable "SLVEMP". Let us suppose that the values need to be retrieved from a Simple List Variable SLVEMP and displayed in a report.

This report "SLV List Values with Key Display" is opened in 'Display' mode. Let us look into the code snippet of the Part definition:

```
[Part : SLVList ValuesDisplay]

  Lines      : SLV List DisplayTitle, SLV List DisplayValues

  Repeat     : SLV List DisplayValues

  Set        : $$ListCount : SLVEmp

  Scroll     : Vertical

  CommonBorder : Yes
```

In Part level, the number of lines is fixed using the Attribute 'SET', based on the number of elements in the Simple List Variable "SLVEmp".

```
[Line : SLV List DisplayValues]

  Fields : SLV Alias, SLV Name

  [Field : SLV Alias]

    Use      : Name Field

    Set as : $$ListKey:SLVEMP:$$Line

  [Field : SLV Name]

    Use      : Name Field

    Set as : $$ListValue : SLVEMP : #SLVAlias
```

Key and Value from the Simple List Variable "SLVEMP" are retrieved using the functions \$\$ListKey and \$\$ListValue at the field level. Similarly, the values can be retrieved from a Compound List Variable also.

4.11 Variables in Collection

The inline variables can be declared at the Collection using the Attributes Source Var, Compute Var and Filter Var. In case of Simple Collection, during the evaluation, only current objects are available. Whereas in case of Aggregate/Summary collection, during the evaluation, the following three sets of objects are available:

Source Objects: Objects of the collection specified in the 'Source Collection' attribute

Current Objects: Objects of the last collection specified in the Walk path

Aggregate Objects: Objects obtained after performing the grouping and aggregation

There are scenarios where some calculation is to be evaluated based on the source object or the current object value and the filtration is done based on the value evaluated with respect to the final objects before populating the collection. In these cases, to evaluate the value based on the changing object context is tiresome, and sometimes impossible as well.

The collection level variables provide Object-Context Free processing. The values of these inline variables are evaluated before populating the collection.

The sequence of evaluation of collection attributes has been changed to support attributes ComputeVar, Source Var and Filter Var. The variables defined using the attributes Source Var and ComputeVar can be referred to in the collection attributes By, Aggr Compute and Compute. The variable defined by 'Filter Var' can be referred to in the collection attribute 'Filter'. The value of these variables can be accessed from anywhere, while evaluating the current collection objects.

Attributes SOURCE VAR, COMPUTE VAR and FILTER VAR

□ Attribute - Source Var

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

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression, which evaluates to a value of 'Variable' data type.

Example:

```
Source Var : Log Var: Logical : No
```

The value of the variable 'LogVar' is set to NO.

- **Attribute - Compute Var**

The attribute 'Compute Var' evaluates the value of the variable based on the current objects.

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression which evaluates to a value of 'Variable' data type.

Example:

```
Compute Var : IName : String : if ##LogVar then $StockItemName else +
                                     ##LogVar
```

- **Attribute - Filter Var**

The attribute 'Filter Var' evaluates the value of the variable based on the objects available in the collection after the evaluation of the attributes 'Fetch' and 'Compute'.

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression which evaluates to a value of 'Variable' data type.

Example:

```
Filter Var : Fin Obj Var : Logical : $$Number:$BilledQty > 100
```

4.12 Using Variable as a Data Source for Collections

Collection attribute 'Data Source' has been enhanced to support 'Variable' as a data source. Now, variable element(s) can be gathered as objects in collection and their respective simple member variables will be available as methods. Member List Variables will be treated as sub-collections.

Syntax

```
Data Source : <Type> : <Identity> [:<Encoding>]
```

Where,

<Type> is the type of data source, i.e., File XML, HTTP XML, Report, Parent Report, Variable.

<Identity> can be file path/ scope keywords/ variable specification, based on type of data source.

<Encoding> can be ASCII or UNICODE. It is applicable for data types File XML and HTTP XML.

Example:

Simple List Variable as Data Source

```
[Collection : LV List Collection]
```

```
Data Source : Variable : SLVEmp
```

The elements of the Simple List Variable 'SLVEmp' will be available as objects in the collection 'LV List Collection'. Let us suppose that a Line is repeated over the collection 'LV List Collection'. The value can be retrieved in the field level as shown below:

```
[Field : SLVEmp Field]
```

```
Use      : Name Field
```

```
Set as  : $SLVEmp
```

Compound List Variable as Data Source

```
[Collection : CV List Collection]
```

```
Data Source : Variable : CLVEmp
```

The elements of the Compound List Variable CLVEmp will be available as objects in the collection CV List Collection. It is used as a Source Collection in the following Summary Collection:

```
[Collection : CV List SummaryCollection1]
```

```
Source Collection : CV List Collection
```

```
Walk              : Relatives
```

```
By                : Relation : $Relation
```

```
Aggr Compute     : MaxAge   : Max : $Age
```

```
Aggr Compute     : MinAge   : Min : $Age
```

```
Aggr Compute     : TotSal   : Sum : $Salary
```

Here, we are walking to the sub-collection 'Relatives' and performing grouping and aggregation.

4.13 Variables in Remoting

In a Tally.NET Environment, where Tally at the remote end sends request to the Tally Company at the Server, all client requests must contain the dependencies, based on which data is gathered. In other words, any request sent to the server must accompany the values configured at the client to extract data from the server. For example, a Collection of Ledgers falling under a user selected group must accompany the request sent to the server. Hence, the request to the server must contain the Variable value which signifies the Group name.

Only the configuration information which is relevant to the data to be fetched from the Server needs to be sent to the Server, and not the ones which are User Interface related, like Show Vertical Balance Sheet, Show Percentages, etc.

When a Collection is sent to the Server, all the dependencies, i.e., variable values, are enclosed within the requests **automatically**.

Example: 1

```
[Collection : User Ledger Coll]
    Type      : Ledger
    Child of  : ##UserSelectedGroup
```

While sending this collection to the server, the value for the variable **UserSelectedGroup** is also passed to the server automatically and the server responds accordingly.

Example: 2

```
[Collection : Emp Coll]
    Type      : Cost Centre
    Filter    : EmpSpouseName

[System : Formula]
    EmpSpouseName : $SpouseName = ##CLVEMP[1].Relatives[1].Name
```

Value of CLVEMP[1].Relatives[1].Name will be enclosed within the request to the server.

In some cases, variable values will not be remoted automatically like Child Of : \$FuncName, which in turn returns the variable value through the Function. Such variables need to be remoted using an adhoc 'Compute' within the collection. This 'Compute' is required to set a manual dependency on the variable and hence, consider it while sending request to Server. Consider the following example:

```
[Collection : User Ledger Coll]
    Type      : Ledger
    Child of  : $$UserFunc

[Function : UserFunc]
    00 : RETURN : ##FuncVar
```

In this example, the function **UserFunc** returns the value through the variable 'FuncVar'. Hence, the variable 'FuncVar' needs to be remotod using an adhoc 'Compute' as follows:

```
[Collection : User Ledger Coll]

Type      : Ledger

Child of  : $$UserFunc

Compute   : FuncVar : ##FuncVar
```

4.14 Use Case – Report Configuration

Scenario

ABC Company Limited, which is into trading business, is using Tally.ERP 9. It deals with purchase and sale of computers, printers, etc. The company management likes to view the Stock Summary Report in various dimensions. Hence, every time, they need to set configurations for the report and view it. They want to have multiple configurations for the same report and set it at one time.

Requirement Statement

By default, in Tally, the user has to set the configurations in Stock Summary Report as per the requirement every time. The requirement can be customized using the Compound List Variable.

Functional Demo

The solution has been developed using Compound Variables and User Defined Functions. Before looking into the design logic, we will have a functional demo.

A new stock summary report is created for demonstration purpose and the same is available as a part of "TDL Language Enhancements" sample TDLs. The TDL is enabled in Tally.ERP 9.

Saving Multiple Configurations

Gateway of Tally → TDL Language Enhancements → What's New → Release 1.8 → Variable Framework → Stock Summary → F12. Set the required configuration.

What's New in Release 1.8

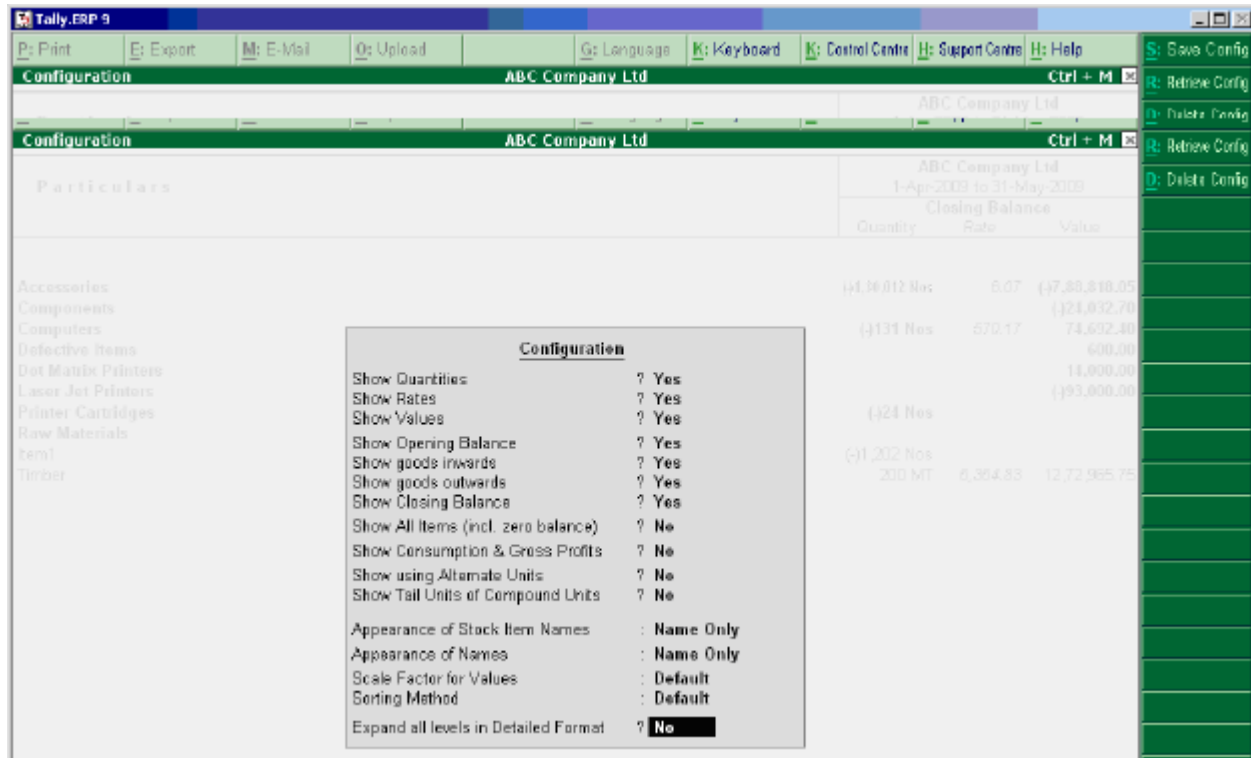


Figure 4. Setting required Configurations

The above configuration has to be saved using the button **Alt+S** (Save Config). Enter the Configuration Name and accept it as shown in the following figure:

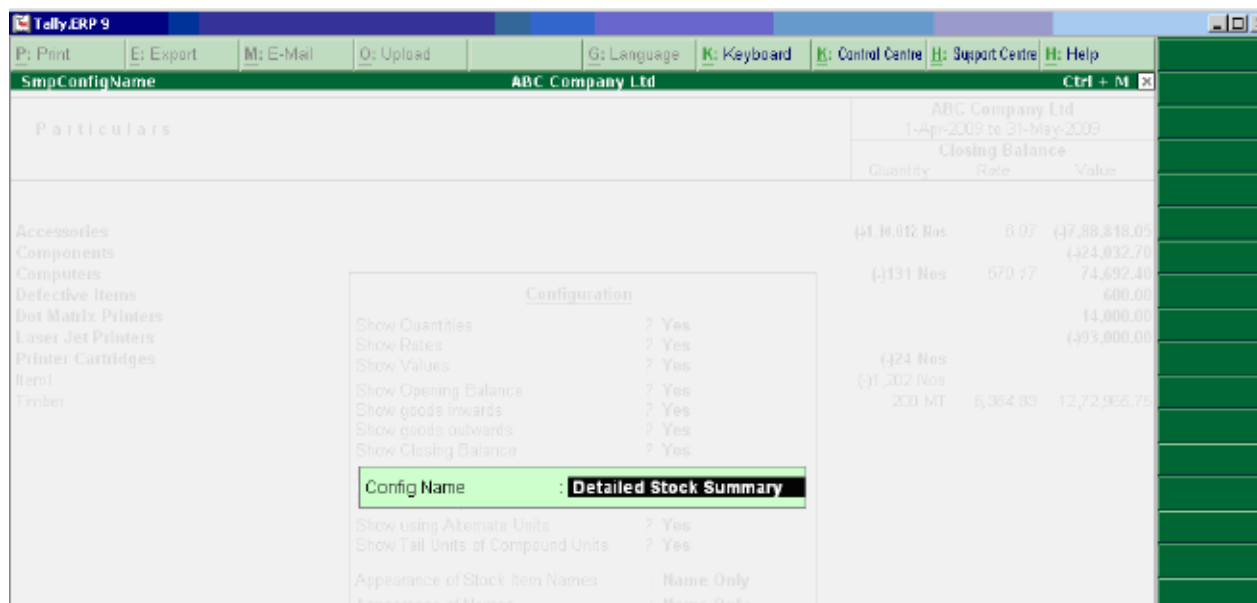


Figure 5. Saving the Configuration with a suitable name

Similarly, we can save another configuration for the same report, as shown in the following figure:

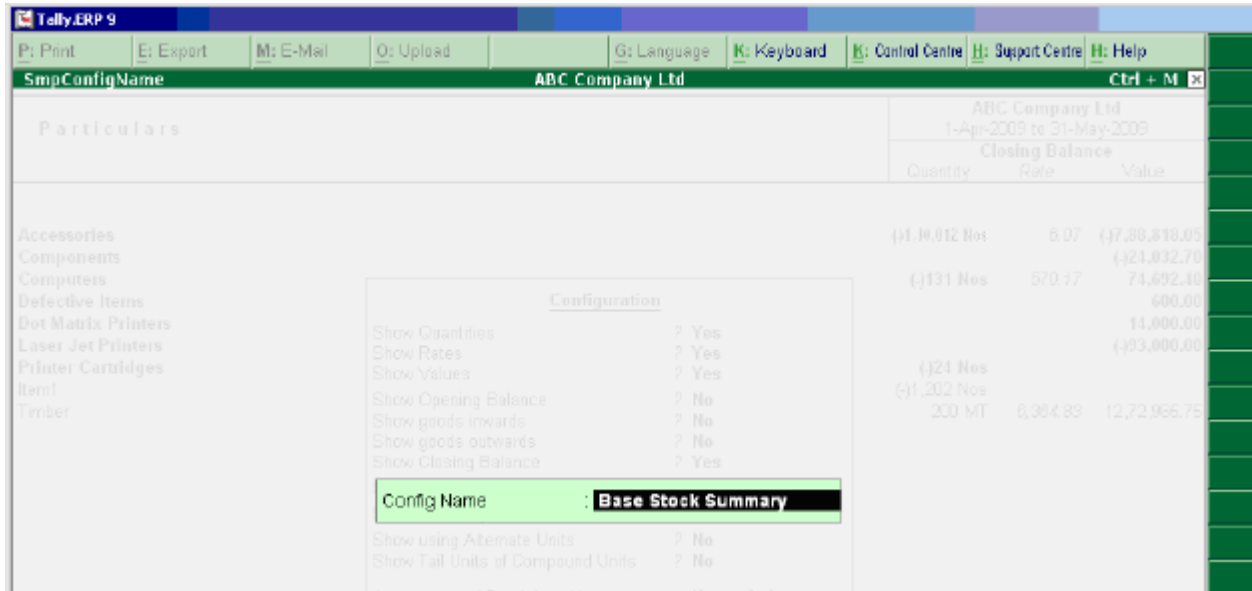


Figure 6. Saving another Configuration

Retrieving Configuration to view the Report in Different Dimensions

Gateway of Tally → TDL Language Enhancements → What's New → Release 1.8 → Variable Framework → Stock Summary → F12 → Alt+R (Retrieve Config). Select the Required Configuration and press **Enter**.

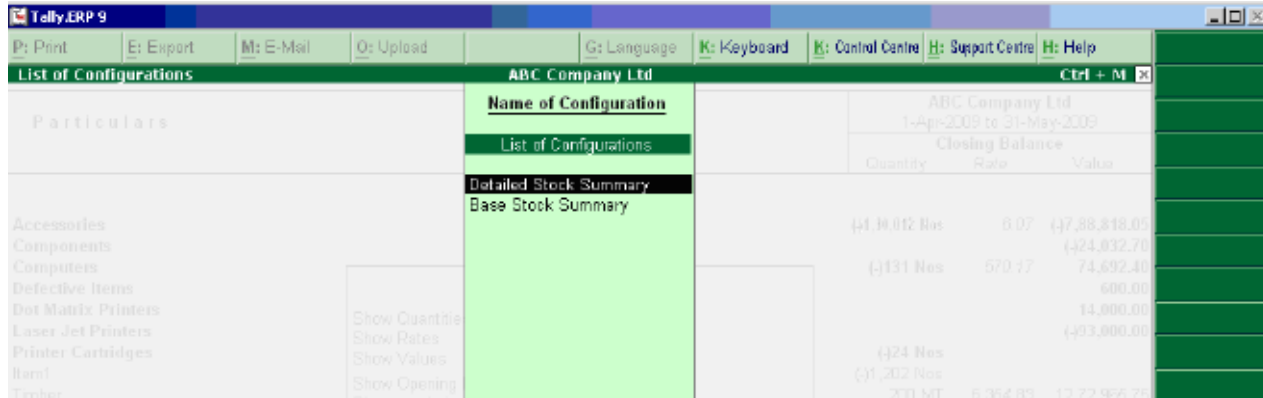


Figure 7. Retrieving and Selecting the Required Configuration

What's New in Release 1.8

The configuration will be set automatically as shown in the following figure:

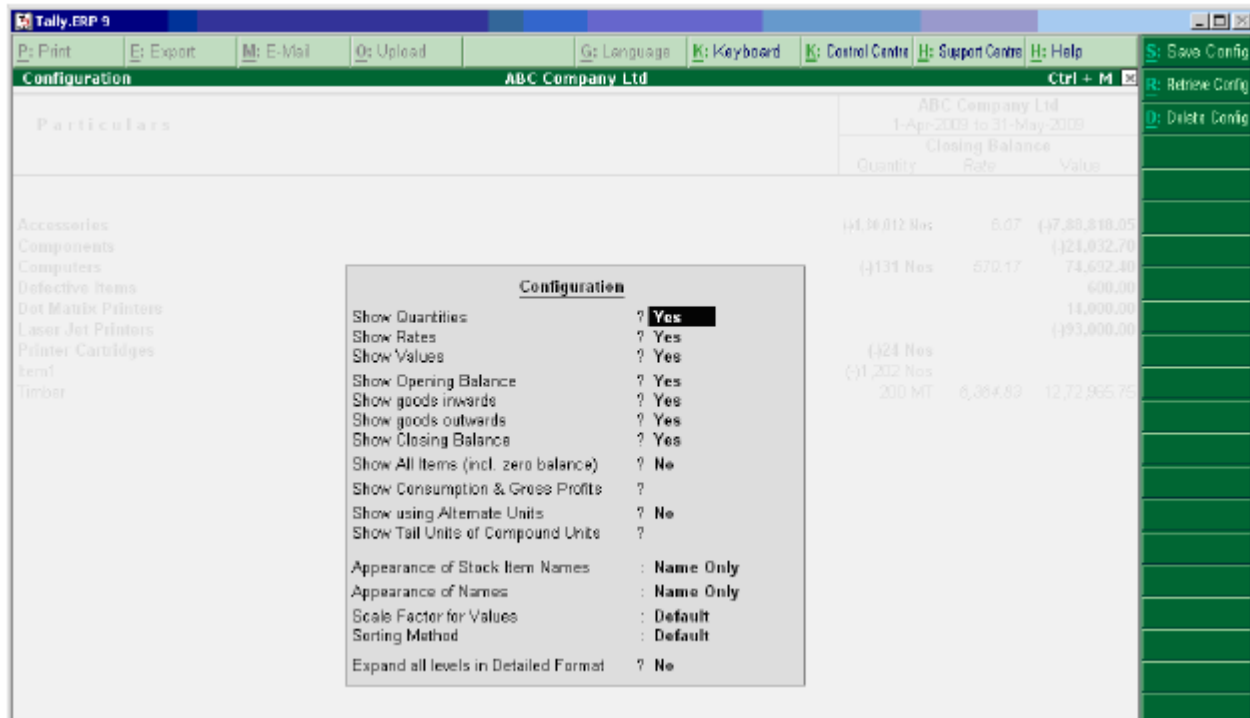


Figure 8. Applying the selected Configuration

Accept the screen to view the Report:

ABC Company Ltd												
1-Apr-2009 to 31-May-2009												
Particulars	Opening Balance			Inwards			Outwards			Closing Balance		
	Quantity	Rate	Value	Quantity	Rate	Value	Quantity	Rate	Value	Quantity	Rate	Value
Accessories	349,210 Nos	0.32	15,792.76	(-100 Nos	8.82	(-882.00	88,202 Nos	12.77	10,30,714.00	6,130,002 Nos	6.07	(7,88,818.00
Components			2,573.22						20,662.00			(-24,032.70
Computers	(-150 Nos	18,239.64	9,14,982.14				81 Nos	10,678.70	8,64,975.00	(-131 Nos	570.17	74,692.40
Defective Items			600.00						1,000.00			600.00
Dot Matrix Printers			14,000.00									14,000.00
Laser Jet Printers			17,400.00						1,44,000.00			(-93,000.00
Printer Cartridges							24 Nos	1,350.00	32,400.00	(-24 Nos		
Item							1,202 Nos	100.00	1,20,200.00	(-1,202 Nos		
Timber	200 MT	6,364.83	12,72,965.75							200 MT	6,364.83	12,72,965.75
Grand Total			22,38,313.87	(-100 Nos		(-882.00	82,008 Nos		22,13,961.00			4,56,407.40

Figure 9. Report configured as per the selection

To view the same Report with another configuration, Press **F12** → **Alt+R**, select the required configuration, and press **Enter**.

What's New in Release 1.8

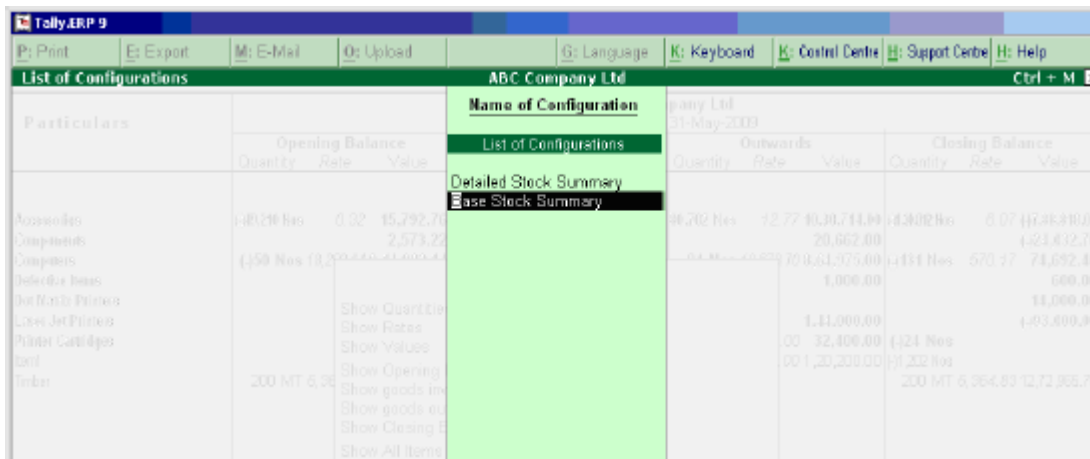


Figure 10. Retrieving and Selecting another Configuration

The configuration will be set automatically as shown in the following figure:

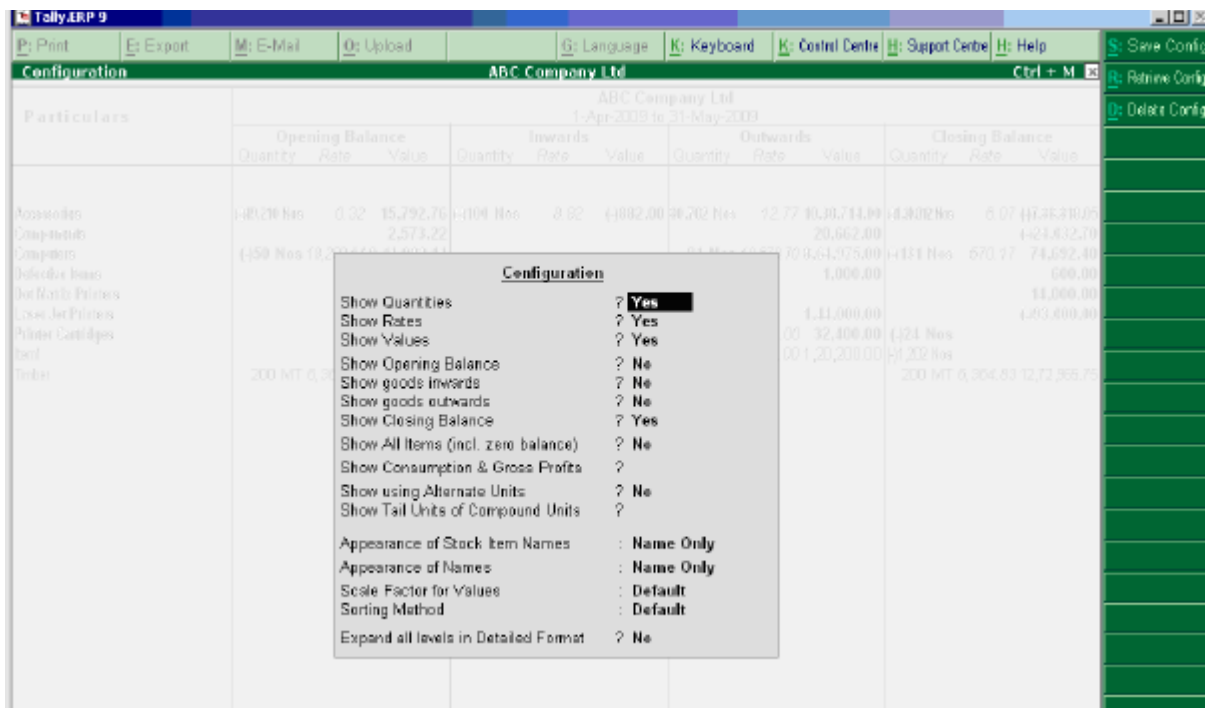


Figure 11. Applying the selected Configurations

Accept the Screen to view the Report.

Particulars	ABC Company Ltd 1-Apr-2009 to 31-May-2009		
	Quantity	Rate	Value
Accessories	(1,00,012 Nos	6.07	(17,88,818.05
Components			(24,032.70
Computers	(131 Nos	570.17	74,692.40
Defective Items			609.00
Dot Matrix Printers			14,009.00
Laser Jet Printers			(93,009.00
Printer Cartridges	(24 Nos		
Item1	(1,202 Nos		
Timber	200 MT	6,364.83	12,72,965.75

Figure 12. Report configured as per the selection

Deleting the Configuration

Gateway of Tally → TDL Language Enhancements → What's New → Release 1.8 → Variable Framework → Stock Summary → F12 → Alt+D (Delete Config).

Particulars	ABC Company Ltd 1-Apr-2009 to 31-May-2009		
	Quantity	Rate	Value
Accessories	(1,00,012 Nos	6.07	(17,88,818.05
Components			(24,032.70
Computers	(131 Nos	570.17	74,692.40
Defective Items			609.00
Dot Matrix Printers			14,009.00
Laser Jet Printers			(93,009.00
Printer Cartridges	(24 Nos		
Item1	(1,202 Nos		
Timber	200 MT	6,364.83	12,72,965.75

Figure 13. Selecting the configuration name for deletion

Select the Configuration to be deleted and press **Enter**.

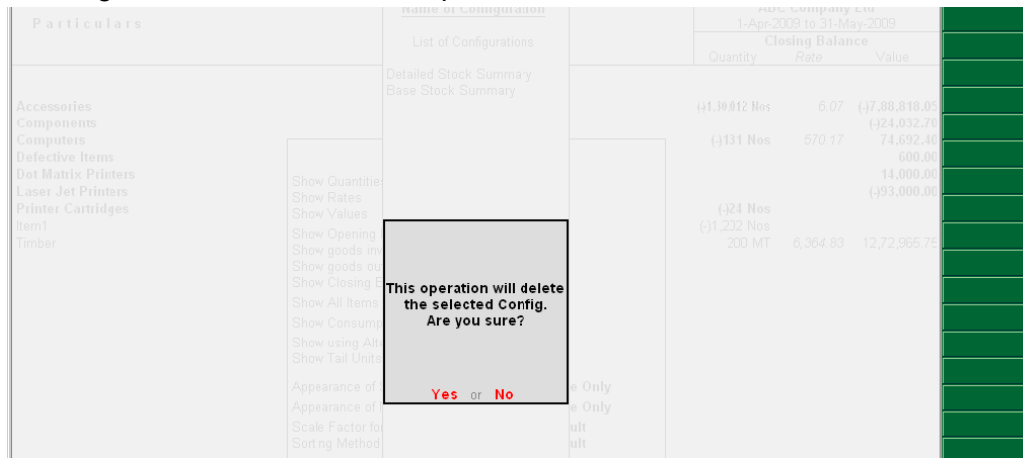


Figure 14. Confirmation before deleting the selected configuration

Accept it to delete the configuration. Press **Alt+R**. The report is not displaying the configuration “Detailed Stock Summary” (as shown in the next figure).

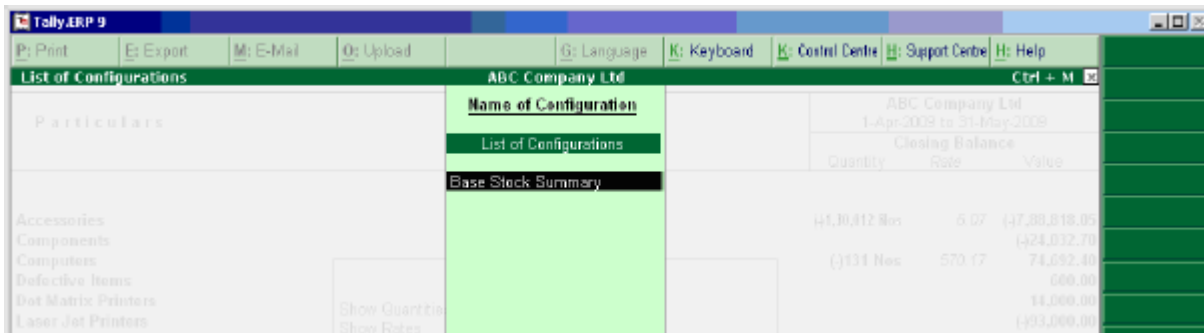


Figure 15. Report of Configuration List post deletion

Solution Development

The solution (Setting different configurations for the same report, saving and retrieving them as and when required) was possible by using Compound List Variable.

The steps followed to achieve the requirement are:-

1. Defining Compound Variables with the required members having Persistence behaviour

```
[Variable: Smp Save Config]

Variable      : Smp Report Name      : String
Variable      : Smp Company Name     : String
Variable      : Smp User Name        : String
List Variable  : Smp Config
Persistent    : Yes

[Variable: Smp Config]

Variable      : Smp Config Name      : String

:: Stock Summary

Variable      : Smp Show Qty          : Logical
Variable      : Smp Show Rate        : Logical
Variable      : Smp Show Value       : Logical

Variable      : Smp Show Open Bal     : Logical
Variable      : Smp Show Goods Inw   : Logical
Variable      : Smp Show Goods Outw  : Logical
Variable      : Smp Show Clos Bal    : Logical

Variable      : Smp Show All Items    : Logical
Variable      : Smp Show Cons GP     : Logical
Variable      : Smp Show Alt Units    : Logical
Variable      : Smp Show Tail Units   : Logical
Variable      : Smp Appear Item Name  : String
```

Figure 16. Defining Compound Variables

Declaring Compound List in System Scope

```
[System: Variable]

List Variable  : Smp Save Config
Variable       : Smp Delete Config : Logical
```

Figure 17. Declaring Compound List Variable in System Scope

2. Adding Relevant Buttons

```

[#Form: DSP Configure]

    Add      : Buttons      : Smp Save Config, Smp Retrieve Config

[Button: Smp Save Config]

    Key      : Alt + S
    Action   : CALL        : Smp Save Config
    Title    : "Save Config"

[Button: Smp Retrieve Config]

    Key      : Alt + R
    Action   : CALL        : Smp Retrieve Config
    Title    : "Retrieve Config"
    
```

Figure 18. Adding relevant buttons

3. When the user chooses to save a configuration,

- Add an element with current Report Name as Key

```

20 : IF      : NOT ##ListVarExists
30 : LIST ADD : SmpSaveConfig      : ##SmpBaseReport
40 : ENDIF
    
```

Figure 19. Adding an element to the list variable

- Add an element within the above element with Config Name (specified by user) as Key.

```

80 : IF      : NOT ##ListVarExists
90 : LIST ADD: SmpSaveConfig[##ListVarIndex].SmpConfig      : ##UserConfigName
    
```

Figure 20. Adding a sub-element

- Set the **variable values** in the current Configuration Screen to the respective **Members** within the above sub element.

```

00 : SET      : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpConfigName      : ##UserConfigName
10 : SET      : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowQty        : ##SSShowQty
20 : SET      : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowDate       : ##SSShowDate
30 : SET      : SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowValue      : ##SSShowValue
    
```

Figure 21. Setting the member variable values

4. When the user chooses to retrieve a configuration,
 - Display a Report showing the list of available Configurations in a Table.

```
[Report: SmpListofConfigs]      ;/ Report to display list of configs

Form      : SmpListofConfigs
Title     : "List of Configurations"
Auto      : Yes

[Field: SmpListofConfigs]

Use       : Name Field
Set As    : ##SmpSaveConfig[##ListVarIndex].SmpConfig[$$Line].SmpConfigName
```

Figure 22. Report to display list of configurations

On selecting the desired configuration, retrieve the saved values from the compound variable and set the values to the respective configuration variables.

```
00 : SET      : ListVarSubIndex : ##UserConfigIndex
10 : SET      : DSPShowQty     : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowQty
20 : SET      : DSPShowRate    : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowRate
30 : SET      : DSPShowValue   : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowValue
40 : SET      : DSPShowOpening : ##SmpSaveConfig[##ListVarIndex].SmpConfig[##ListVarSubIndex].SmpShowOpenBal
```

Figure 23. Applying selected configuration values to the Report variables

5. When the user chooses to delete a configuration,
 - Display a Report showing the list of available Configurations in a Table.
 - On selecting the desired configuration, delete saved values from the compound variable.

```
Function: Smp_Delete_Variable_Values]

00 : SET      : ListVarSubIndex : ##UserConfigIndex
10 : QUERY BOX : "This operation will delete \nthe selected Config. \nAre you sure?":Yes:No
20 : IF       : ##LastResult
30 :   LIST DELETE : SmpSaveConfig[##ListVarIndex].SmpConfig:##SmpSaveConfig[##ListVarIndex].SmpConfig[##L
40 : ENDF
```

Figure 24. Deleting the selected configuration from the List Variable

TDL Capabilities Used

- User Defined Functions
- Compound List Variable



Code Snippets have been extracted from the working solution provided with the Samples.

5. Licensing Binding Mechanism

Nowadays, it is a common practice to have multiple applications for various business operations at different branches/ locations and then integrate their data and/or reports, as and when required. Tally being the most common and popular product across all industries, many Third Party Applications look forward to integrate their applications with Tally.

To ensure a secure environment, Third Party Applications need to build a robust licensing mechanism in order to validate the users of their application, which may be time consuming and costly. Alternatively, they can opt to use the robust licensing mechanism already built in Tally and stitch it together with the Tally Application.

License Information like Tally Serial Number, Account Email ID, etc., can be retrieved from Tally and validated with the current instance of an external application. In order to use the Tally licensing mechanism, Third Party Applications need to send various XML Requests to Tally running at a predefined IP Address and a Port. On receiving the XML Request in Tally understandable format, Tally responds with the required information, data or Report requested.



For any further information on XML Formats, please refer to the documents and samples available at www.tallysolutions.com in the path Developer Network -> Tally Technology -> Integration Capabilities.

The various approaches for retrieving License Information from Tally that can be followed by Third Party Applications have been broadly classified based on the desired level of security, ranging from simple to the most complex one.

The Approaches that can be used by Third Party Applications to retrieve License Information from Tally, based on the level of Security desired, are as follows:

5.1 License Info Retrieval using Open XML

This approach is one of the simplest approaches with minimal security wherein the Third Party Applications will be able to send an XML Request to invoke platform functions in Tally to retrieve the required License Information. This is a less secured environment, as the license data returned will be available as an Open XML

In Tally, a platform function **\$\$LicenseInfo** is available which accepts a parameter to determine the type of License details required and returns the value accordingly. For example, **\$\$LicenseInfo:SerialNumber** returns the Serial Number of the running copy of Tally.

Following is the list of parameters allowed for the Function **\$\$LicenseInfo**:

Parameters permissible for LicenseInfo	Return Type	Description
SerialNumber	Number	Serial Number
AccountID	String	Account ID
SiteID	String	Site ID
AdminEmailID	String	Admin Email ID
IsAdmin	Logical	Whether the System logged in user is Administrator or not
IsIndian	Logical	Whether the country is India or not
IsSilver	Logical	Whether the Product flavour is Silver or not
IsGold	Logical	Whether the Product flavour is Gold or not
IsEducationalMode	Logical	Whether the Product is running in Educational mode
IsLicensedMode	Logical	Whether Product is running in Licensed mode
LicServerDate	Date	License Server Date
LicServerTime	String	License Server Time
LicServerDateTime	String	License Server Date & Time

The following XML Request is required to fetch Tally Serial Number:

```

<!-- XML Request -->
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <TALLYREQUEST>EXPORT</TALLYREQUEST>
    <TYPE>FUNCTION</TYPE>
    <!-- Platform Function Name in Tally.ERP 9 -->
    <ID>$$LicenseInfo</ID>
  </HEADER>
  <BODY>

```

```

    <DESC>
      <FUNCPARAMLIST>
        <!-- Parameter for the function $$LicenseInfo -->
        <PARAM>Serial Number</PARAM>
      </FUNCPARAMLIST>
    </DESC>
  </BODY>
</ENVELOPE>

```

The previous XML Request fetches the following XML Response:

```

<!-- XML Response -->
<ENVELOPE>
  <HEADER>
    <VERSION>1</VERSION>
    <STATUS>1</STATUS>
  </HEADER>
  <BODY>
    <DESC>
    </DESC>
    <DATA>
      <RESULT TYPE="Long">790003089</RESULT>
    </DATA>
  </BODY>
</ENVELOPE>

```

In this response received from Tally, Serial Number is retrieved within the **RESULT** Tag.

Similarly, to fetch the Account ID of the current Tally Application, replace the Parameter **Serial Number** with **Account ID** within Param Tag in the XML Request.

5.2 License Info Retrieval using Encoding Procedure built in a TCP

This approach is a slightly better approach than the previous one, since the Response received here is encoded using some encoding mechanism built within TDL.

The Third Party Application will send a Validation String within the XML Request. At Tally's End, the validation string and the required License Info will be encoded using the encoding mechanism built within TDL. The converted Strings will then be sent back within the XML Response to the Third Party Applications, which will decode the strings at their end.

Following needs to be made available for this approach to be executed:

At Tally End

A TDL needs to be written containing the encryption mechanism to encrypt a string.

Following is an example of String encryption in Tally using TDL Function:

;; TDL Function to Encrypt an input String, by reversing it

```
[Function : StrEnc]

    Parameter : pStringtoReverse : String

    Variable  : ReverseString      : String

    00 : FOR RANGE : IteratorVar : Number : ($$StringLength: +
        ##pStringtoReverse - 1) : 0: 1

    10 : SET : ReverseString : ##ReverseString + $$StringPart:+
        ##pStringToReverse : ##IteratorVar:1

    20 : END FOR

    30 : RETURN : ##ReverseString
```

This was a simple example of String encryption in Tally. Similarly, much robust encryption mechanisms can be built in TDL and used in Third Party Applications.

Report having a string variable, and triggering the encrypt function with string variable as a parameter, returning the encrypted value within the required XML Tags.

;; TDL Report to invoke the above Function

```
[Report : Sec XML Request2]

    Form : Sec XML Request

;; Variable for received String

    Variable : EncString: String

    [Form : Sec XML Request]

        Parts : Sec XML Request

            [Part : Sec XML Request]

                Lines : Sec XML Req SerialNo, Sec XML Req EncString

                Scroll : Vertical
```

```
XMLTAG : "TALLYLICENSEINFO"
```

;; Serial Number of Tally

```
[Line : Sec XML Req SerialNo]

Fields : Name Field

Local : Field : Name Field : Set As : $$StrEnc:@@LicSlNo

Local : Field : Name Field : XMLTAG : "SerialNumber"
```

;; To Encrypt the received String

```
[Line : Sec XML Req EncString]

Fields : Name Field

Local : Field : Name Field : Set As : $$StrEnc:##EncString

Local : Field : Name Field : XMLTAG : "EncryptedString"
```

On receiving the XML Request, the report is executed and both the Serial Number and the String received within the XML Request are encrypted and sent back to Third Party Applications.

At Third Party Application End

An XML Request to trigger the Tally Report with request String to be encrypted. Following XML Request triggers the previous Report associated with Tally:

```
<!-- XML Request -->
<ENVELOPE>

  <HEADER>

    <VERSION>1</VERSION>

    <TALLYREQUEST>Export</TALLYREQUEST>

    <TYPE>Data</TYPE>

    <ID>Sec XML Request2</ID>

  </HEADER>

  <BODY>

    <DESC>

      <STATICVARIABLES>

        <SVEXPORTFORMAT>$$SysName:XML</SVEXPORTFORMAT>

        <EncString>Keshav</EncString>

      </STATICVARIABLES>
```

```
</DESC>
```

```
</BODY>
```

```
</ENVELOPE>
```

Sec XML Request2 is the TDL Report which is requested and variables **SVExportFormat** (format in which response is required) and **EncString** (Variable Name specified in TDL Report for string to be encrypted) are enclosed within the XML Request.

The following response is received from Tally on sending the above request

```
<!-- XML Response -->
```

```
<ENVELOPE>
```

```
<TALLYLICENSEINFO>
```

```
<SERIALNUMBER>980300097</SERIALNUMBER>
```

```
<ENCRYPTEDSTRING>vahsek</ENCRYPTEDSTRING>
```

```
</TALLYLICENSEINFO>
```

In this response, Serial No. and String sent as request are returned encrypted, i.e., reversed from Tally. On receiving the response, the Third Party Application needs to decrypt the Serial Number as well as String and validate the current instance. It is a much secure environment as the response is in encrypted form.

5.3 License Info Retrieval using Encryption Functions provided within Tally

This Approach is similar to the previous approach except that it uses an inbuilt Platform Function to encrypt the string. In Tally, the validation string and the required License Info can be encrypted using the function **\$\$EncryptStr** provided within the platform. The encrypted Strings will be sent back within the XML response to the Third Party Application. The Third Party Application will decrypt the Strings at their end using the standard DLL shipped by Tally for decryption.

XML Request is similar to the Request in the previous approach, except that:

- ❑ An additional variable value Password must be specified with the XMLTag **Password** (Variable Name used in TDL Report for Password), and
- ❑ The requested Report triggers the platform function **\$\$EncryptStr** for encryption mechanism.

A supporting DLL File **EncryptDecrypt.DLL** is provided along with Sample Files to decrypt the Encrypted String in Tally, using the Function **\$\$DecryptStr** available in DLL. This Function accepts 4 parameters viz.,

- ❑ Input String to be decrypted
- ❑ Password specified while encoding an XML request
- ❑ Output String Variable to hold the decrypted return Value
- ❑ Output String Buffer Length



The above DLL can be copied either to the local path of the Third Party Application or to the Windows System Directory.

On decrypting the above string, the Third Party Application can validate the returned String and Serial Number, and continue if the validation is successful.

5.4 License Info Retrieval using Encryption Algorithms built using Third Party DLLs

This Approach is the most secured approach, wherein an external DLL is written to encrypt the given string. The Third Party Application will send a Validation String within the XML Request. At Tally's End, the validation string and the required License Info will be encrypted using an External DLL, which can have its own Encryption Routines.

Tally uses the function **\$\$CallDllFunction** to trigger the DLL written for encryption and returns the encrypted strings to the Third party Application within the XML Response. At Third Party Application End, decryption algorithms will be required, which can again be provided inside the same DLL used for encryption.

▫ **Function - \$\$CallDllFunction**

The Platform Function **CallDllFunction** is used to trigger the function enclosed within an external DLL (*written in C++/VC++*)

Syntax

```
$$CallDllFunction : <DLL Name> : <Function Name> : <Param 1> : +
                  <Param 2> :...<Param N>
```

Where,

<DLL Name> is any DLL written in C++/VC++,

<Function Name> is a Function available in the DLL,

<Param 1 to N> are arguments, which depend upon the number of parameters needed by the Function designed.



DLL must exist in the Tally Application folder or Windows System Folder.

XML Request for this approach is similar to the Request in the previous approach, except that

- The requested Report triggers the function written within DLL for encryption mechanism using **CallDllFunction**.

Subsequently, the Third Party Application can decrypt the encrypted String and the Serial Number using the decrypt function within the same DLL or any other DLL.

What's New in Release 1.61

1. Narrowing Table Search

The current search capability on a Table allows the user to highlight a particular set of items based on the search text entered in the field. The text is searched from the beginning of the item names in the list and is applicable to the first column only.

In a scenario where there are large number of items in the list/table, it is impossible for the user to remember the starting characters of the item names. He may remember only a part of the item name which he requires to search. Even after the relevant items are searched and highlighted, all the items are displayed, which is not required.

The latest enhancement in TDL allows the user to search a text from any part of the item name which appears in the list. The table keeps on narrowing down and displaying only those items which fulfil the search criteria. It is also possible now to specify whether the search criteria should be applicable on first column or all columns of the table.

1.1 Field Attribute – Table Search

A new field attribute called 'Table Search' has been introduced to achieve the above capability.

Syntax

```
[Field : <Field name>]
    Table Search : <Enable reducing table search> : <Apply search to all +
                                                    columns>
```

Where,

<Enable reducing table search> is a logical value (YES/NO), to specify whether we want to enable the reducing of search or not.

<Apply search to all columns> is a logical value (YES/NO), to specify whether the search criteria should apply to all columns of the table or not.



We can also use expressions in attribute values which evaluate to logical values.

1.2 Function - \$\$TableNumItems

A new function \$\$TableNumItems has been introduced which returns the number of items in the list/table.

Example:

```
[Collection : RTS Ledger]
```

```
Type      : Ledger
Format    : $Name
Format    : $Parent
Format    : $ClosingBalance
```

```
[Field : Reducing Table Search GT 100]
```

```
Use       : Name Field
Table     : RTS Ledger
Show Table : Always
Table Search : $$TableNumItems > 100 : Yes
```

In this example, the field 'Reducing Table Search GT 100' is displaying the table 'RTS Ledger', which has three columns 'Name', 'Parent' and 'Closing Balance'. The attribute 'Table Search' evaluates the first value to YES, only when the number of items in the table exceeds 100, i.e., reducing search will be enabled if this criteria is met. The second attribute value is set to YES, i.e., the search criteria will apply to all columns in the table.

1.3 Functionality Achieved

Using the above capability, it has been possible to deliver the functionality of applying the above search technique to all the tables available in the default product. This will of course be based on the configuration settings selected by the user.

1.4 Use Cases

1. Company search based on the Company ID.
2. Ledger search based on parent Group name available in other column in a table.
3. While selecting the Stock item Name in a voucher, the user can now narrow the search, based on the UOM and make his selection of item based on the closing balance available for that UOM.

What's New in Release 1.6

In this release, there have been enhancements in User Defined Functions, Collections and Actions. We will see in depth the changes for the Actions - **Print**, **Upload**, **Export** and **Mail**. It is now possible to program the configurations for these Actions. This breakthrough capability has enabled to deliver the mass mailing feature in the product Tally.ERP 9.

Collection attribute **Keep Source** is enhanced to accept a new value, i.e., **Keep Source : ()**. This has been done with the aim to improve the performance. The Loop Collection capability has paved the way for displaying and operating on Multi-Company Data, along with ease of programming.

The TDL language has been enriched with more and more procedural capabilities by introducing the Function **\$\$LoopIndex** and Looping construct **FOR RANGE**. There have been some changes in the Action **NEW OBJECT** as well.

With the introduction of the function **\$\$SysInfo**, it is now possible to retrieve all system-related information consistently using a single function.

1 General Enhancements

1.1 Programmable Configuration for Actions – Print, Export, Mail, Upload

In Tally.ERP 9, the Actions **Print**, **Export**, **Mail** and **Upload** depend upon various parameters like Printer Name, File Name, Email To, etc. Prior to execution of these actions, the relevant parameters are captured in a Configuration Report. These parameters are persisted as system variables, so that the next time, these can be considered as default settings.

There are situations when multiple reports are being printed or mass mailing is being done in a sequence. Subsequent to each Print or Email Action, if a configuration report is popped up for user inputs, this interrupts the flow, thereby requiring a dedicated person to monitor the process, which is time-consuming too. This issue has been addressed in the recent enhancements in Tally.ERP 9, where the configuration report can be suppressed by specifying a logical parameter. Also, the variables can be set prior to invoking the desired action. Before exploring the new enhancements, let us see the existing behaviour of the actions Print, Email, Export and Upload.

Existing behavior of Actions – Print, Export, Mail, Upload

Presently, in Tally.ERP 9, whenever any of these actions is invoked, a common Configuration report **SVPrintConfiguration** is displayed to accept the user inputs. The user provides the details in the configuration screen, based on the action being executed. The action gets executed based on the values provided in the configuration report.

The *existing syntax* of these actions was:

Syntax

<Action Name> : <Report Name>

Where,

<Action Name> can be any of **Mail**, **Upload**, **Print** or **Export**.

<Report Name> is the name of the Report.

For successful execution of these actions, along with the Report Name, additional action-specific parameters are also required. These action specific parameters are passed by setting the values of variables through the configuration report - 'SVPrintConfiguration'.

The default configuration report 'SVPrintConfiguration' is invoked only when the Report specified does not contain the **Print** attribute in its own definition. The 'Print' attribute allows the user to specify his own configuration settings, whenever any of these Actions is invoked.

Example:

```
Mail : Balance Sheet
```

The action **Mail** needs information regarding the To e-mail ID, From e-mail ID, CC e-mail ID, Email server name, etc., which are provided through the Configuration report.

For example, a report needs to be mailed to multiple e-mail IDs in one go. Currently, for every mail, the configuration screen is displayed, and every time, the user has to manually provide To email ID, From email ID, etc. So, whenever any of the above actions is executed, a configuration report is displayed, which requires user inputs. In some scenarios, this behaviour is not desirable. Configuration settings can be specified once, and the user should be able to use it multiple times.

The action syntax *has been enhanced* to avoid the display of configuration screen repeatedly.

Changes in the Actions for Programming Configurations

The global actions Print, Export, Mail and Upload have been enhanced to suppress the Configuration Screen. These actions now accept an additional logical parameter. Based on the value of the logical parameter, the configuration report is suppressed.

The **new enhanced syntax** of these actions is:

Syntax

```
<Action Name> : <Report Name> : <Logical Value>
```

Where,

<Action Name> can be any of **Mail, Upload, Print** or **Export**.

<Report Name> is name of the Report.

<Logical Value> can be **TRUE, FALSE, YES** or **NO**.

With the new syntax, it is possible to configure the values of the report only once and then mail it to the specified e-mail addresses, without repeated display of the configuration report.

Example:

```
10 : MAIL : Ledger Outstandings : TRUE
```

As the Configuration Report is not displayed, the values of the mail action specific variables like 'SVPrintFileName', 'SVOutputName', etc., must be specified for the successful execution of these actions.

Following are the action-specific variables and their acceptable values:

The Configuration Variables – Action Specific

The action-specific Variables can be classified into **four** categories based on their usage.

Common Variables

SVOutputType - The value of this variable is one of the predefined button type keywords like Print Button, Export Button, Upload Button and Mail Button. The variables' value is used by the functions \$\$InMailAction, \$\$InPrintAction, \$\$InUploadAction and \$\$InExportAction to determine the execution of the correct option in the form 'SVPrintConfiguration'. For example, if the value of 'SVOutputType' is 'Print Button', then the optional form 'SV PrintConfig' in the report 'SVPrintConfiguration' is executed.

SVPrintFileName - This variable accepts the output location as value. The value of this variable is specific to each action. The usage of each action is explained in detail, along with the action.

SVExportFormat - The value of this variable is the name of the format to be used with the actions Mail, Upload and Export. The values are SDF, ASCII, HTML, EXCEL, XML, AnsiSDF, AnsiASCII, AnsiXML, AnsiHTML and AnsiExcel, which are set using \$\$SysName.

Example:

```
01 : SET : SVExportFormat : $$SysName:Excel
```

SVExcelExportUpdateBook - This is a logical value and can be used only if the 'Excel' format is used. If the value is set to YES, then the existing file is overwritten; otherwise, it will ask for "Overwrite" confirmation.

SVBrowserWidth, SVBrowserHeight - These variables are used to set the width and height of the page when the format is HTML.

Variables Specific to Action – Print

As soon as the user executes a Print action, the following screen is displayed:

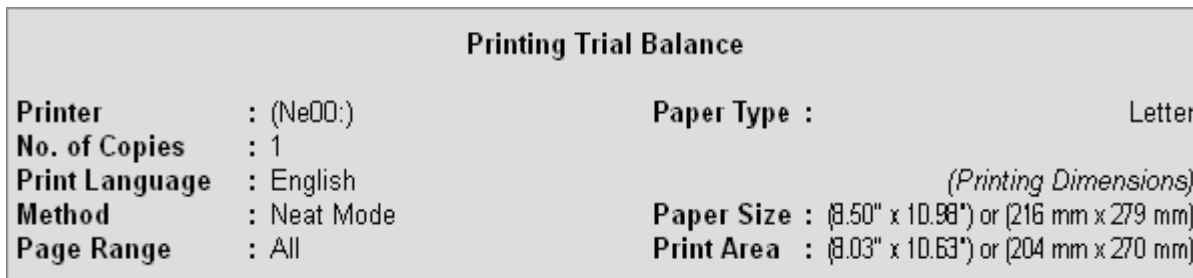


Figure 1. Print Screen

This screen captures the user inputs such as the "Printer Name", "No. of Copies" to be printed, etc. The various action specific variables required by the 'Print' action are modified, based on the user inputs. Following variables are used by Print action:

SVPrintMode - This variable used by 'Print' action accepts the printing mode as the value. The Print mode can be 'Neat', 'DMP' or 'Draft', which are system names. Default mode is **Neat** mode.

SVPrintFileName - This variable is applicable for 'Print' action, only if 'Print To File' option is selected by the user, while printing in DOT Matrix or Quick/Draft format. In this case, the variable 'SVPrintFileName' accepts filename using function \$\$MakeExportName, to add right extensions.

SVPrintToFile - This is a logical value which determines if the print output should be saved in a file. If the value is TRUE, the output is saved in the file specified in the variable 'SVPrintFileName'. If the value is FALSE, then the variable 'SVPrinterName' must contain a valid printer name.

SVPrinterName - It accepts a printer name as a value for printing. The default value is taken from the system settings available in Control Panel for Printers and Faxes.

SVPreview - This is a logical variable and is applicable only for 'Print' action with 'Neat' mode format. If the value is set to YES, then the preview of the report is displayed. Otherwise, the report is printed, without displaying the preview.

SVPrintCopies - It is applicable only for 'Print' action. It accepts a number to print multiple copies.

SVPrePrinted - The variable is applicable only for 'Print' action, and it specifies whether a pre-printed stationary or plain paper is to be used for printing.

SVPrintRange - It is applicable only for Print action. It determines the range of pages to be printed.

SV Draft Split Names - It accepts a logical value to determine if the long names should be split into multiple lines.

SV Draft Split Numbers - It accepts a logical value to determine if the long numbers should be split into multiple lines.

SVPrintStartPageNo - It is applicable only for Print action. It allows to specify starting page no.

SVPosMode - It determines if POS mode is to be used. The default value of this variable is NO.

Variables Specific to Action – Export

The following screen is displayed when the user executes 'Export' action.

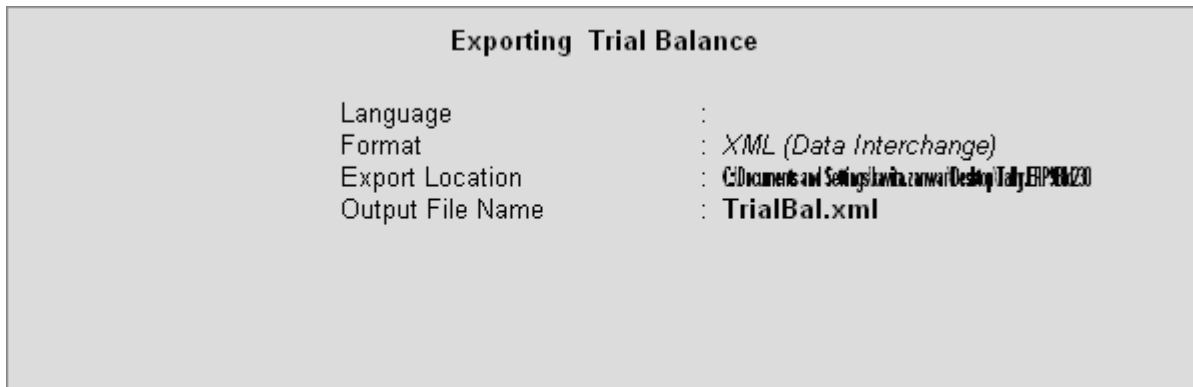


Figure 2. Export screen

This screen captures the user inputs such as "Export Format", "Output File Name", etc. The action 'Export' uses the variables 'SVExportFormat', 'SVPrintFileName', etc.

SVPrintFileName - For the Export action, the value of this variable is the output file name. The path can be specified directly, or the function \$\$MakeExportName can be used to create the output path. The function \$\$MakeExportName suffixes the extension based on the export format, if only the file name is passed as a parameter.

Syntax

`$$MakeExportName : <String Formula> : <Export format>`

Where,

<String Formula> is a string formula which evaluates to the path\filename.

<Export format> is the name of the format which has to be used while exporting.

Example:

`$$MakeExportName : "C:\Tally.ERP\abc.xls" : Excel`

Variables Specific to Action – Mail

When the user executes 'Mail' action, following screen is displayed to capture the mailing details:

Mailing Trial Balance	
E-Mail Server : smtp.gmail.com <i>(Name:Port, Default Port is 25)</i> From : <i>ABC Company Ltd</i> From E-Mail Address : <i>contact@abc.com</i> Authentication User Name: <i>(Only if required)</i> Password : Use SSL : No Format : <i>HTML (Web-Publishing)</i> Resolution : <i>1024 x 768</i>	To E-Mail Address : CC To (if any) : Subject : Trial Balance Additional Text (if any) : Information sent : As Attachment

Figure 3. Mail Screen

For successful execution of 'Mail' action, user has to enter the above details. The URL is then created using function `$$MakeMailName`, and the value is stored in variable 'SVPrintFileName'.

SVPrintFileName - This variable accepts the URL location as value. Function `$$MakeMailName` is used to construct the URL. The mail is sent to specified mail addresses using the given server.

Syntax

`$$MakeMailName : <ToAddress> : <SMTP Sever name> : <From Address>: +
 <CC Address> : <Subject>:<Username> : <Password> :
 <Use SSL flag>`

Where,

<To Address> is the e-mail id of the receiver.

<SMTP Server Name> is the name of the server from which the mail is sent.

<From Address> is the sender's e-mail id.

<CC Address> is the email-id where the copy of the mail is to be sent.

<Subject> is the subject of the mail.

<User Name> is the user id on the secured server.

<Password> is the password for the user id on the secured server.

<Use SSL Flag> can be **TRUE/FALSE OR YES/NO**. If the **Use SSL** flag is set to TRUE, then the Username and Password must be specified, i.e., they can't be empty.

Example:

```

    $$MakeMailName : "abc"+ "<" + " abc@abc.com" + ">" smtp.gmail.com" : +
    "abc@gmail.com" : "" : "Your outstandingpayment" : +
    abc@gmail.com   : abc123 : True
    
```

Variables Specific to Action – Upload

Following screen is displayed to capture details of the folder where the report is to be uploaded:

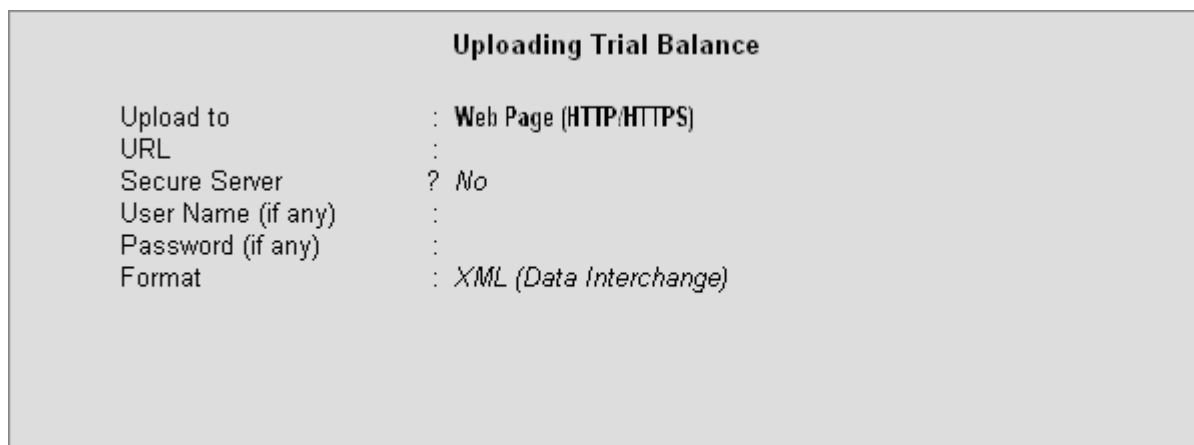


Figure 4. Upload Screen

Based on information entered by the user, the URL of the upload site is created using function **\$\$MakeHTTPName** or **\$\$MakeFTPName**, and the value is stored in variable 'SVPrintFileName'.

SVPrintFileName - It accepts the URL of upload site. The URL is constructed using the functions **\$\$MakeHTTPName** or **\$\$MakeFTPName**, depending on protocol selected by the user for upload. Function **\$\$MakeFTPName** is used for creating the file transfer protocol, based on specifications.

Syntax

```

    $$MakeFTPName : <FtpServer> : <FtpUser> : <FtpPassword> : <FtpPath>
    
```

Where,

<FtpServer> is the FTP server name.

<FtpUser> is the FTP user name.

<FtpPassword> is the FTP password.

<FtpPath> is the full path of the folder on the FTP server.

Example:


```
$$MakeFTPName : "ftp://ftp.microsoft.com" : "" : "" : "dbook.xml"
```

Function **\$\$MakeHTTPName** is used for creating the Hyper Text Transfer Protocol for the specified security features.

Syntax

```
$$MakeHTTPName : <HttpUrl> : <HttpIsSecure> : <HttpUserName> : +
                <HttpPassword> : <CompanyName>
```

Where,

<HttpUrl> is the HTTP URL name.

<HttpIsSecure> is a logical attribute which checks whether the HTTP is secure or not.

<HttpUserName> is the HTTP user name.

<HttpPassword> is the HTTP password.

<CompanyName> is the name of the Company.

Example:

```
$$MakeHTTPName : "https://www.abc.com" : Yes : "guestuser" : "pswd99" :
                "ABC Company Ltd"
```

Use Case Scenario:

Report "Bill-wise details" is to be mailed to each party with their respective Bill Details. However, mails to all parties should be sent at one key stroke, without the e-mail configuration screen popping up multiple times. As of now, a user has to manually enter email IDs for each ledger.

Solution: Following steps need to be implemented:

Step 1: Create Function

```
[Function : FuncEmailingOutstanding]
```

```
Variable : LedgerName : String
```

Step 2: Create Local Formulae for enhanced readability of code

```
Local Formula : FromAddress : "abc" + "<abc@abc.com>"
```

```
Local Formula : ToAddress   : if $$IsEmpty : $Email then "abc@abc.com"
                               else      $Email
```

```
Local Formula : Subject     : ##LedgerName + "(Bill-wise Details)"
```

Step 3: Set the values of common variables used in 'Mail' action

```
03a : SET : SVExportFormat : $$SysName : HTML
```

Step 4: Walk the Ledger Collection to retrieve the email-id of the ledger

```
01 : WALK COLLECTION      : SDLedger
```

Step 5: Set the values of the variables used in 'Mail' action

```

02 : SET : LedgerName      : $Name

03b : SET : SVMailEmbedImage : @@AsAttach

03c : SET : ExplodeFlag     : "Detailed"

03d : SET : SVPrintFileName  : $$MakeMailName : @ToAddress : +
      "smtp,gmail.com" : @FromAddress : "admin@tallysolutions.com" : +
      @Subject : "" : "" : FALSE

```

Step 6: Call the action

```

04 : MAIL : Ledger Outstandings : TRUE

;; TRUE is meant to suppress the configuration report

06 : END WALK

|

|

08 : RETURN

```

2 Collection Enhancements

In Collection definition, attributes **Keep Source** and **Collection** have been enhanced. The collection attribute 'Keep Source' has been enhanced to accept a new value, i.e., Keep Source: '()' to make the data available at Primary Object, which can be a Menu, Report or Function.

The Collection definition can now use a new capability to loop one collection for each object of another collection. This functionality has been introduced by enhancing the 'Collection' attribute.

2.1 'Collection' Attribute Value - Keep Source: ().

Attribute 'Keep Source' accepts various values used to specify the In memory source retention of the collection. Specifications like ., .., Yes, No, etc., were used earlier for this. The source collection was retained along with data object associated with the User Interface object in the current User Interface object hierarchy, as per specification. The newly introduced specification "()." is used to keep the source collection with the parent UI object, which is either Report or Function.

The dotted notation depends on the interface object hierarchy. If there are recursive explodes in a report, then it is difficult to use the dotted notation when the data is to be kept at Report or Function level. The new value Keep Source : (). has been introduced to overcome this issue.

Keep Source : (). signifies that the collection data has to be kept available at primary level, which can be Menu or Report or Function. So, now 'Keep Source' attribute accepts the following values:

- Keep Source: **NO** –The source collection data is not kept in memory.
- Keep Source: **YES** – The source collection data is kept in the object associated with the current interface object.

- Keep Source: (). –The source collection data is kept in the data object associated with the primary owner interface object, i.e., Menu or Function or Report.
- Keep Source: – Each dot in the notation is used to traverse upward in the owner chain by the number of dots specified, till a primary interface object is found.

In scenarios where the data is to be kept at Primary interface object, the application developer can directly use Keep Source : (). without worrying about the interface object hierarchy.

Example:

```

|
[Part : TB Report KeepSrc Part]

  Lines      : TB Report KeepSrc Title, TB Report KeepSrc Details

  Bottom Lines : TB Report KeepSrc Total

  Repeat      : TB Report KeepSrc Details:TB Report KeepSrc GroupsPri

```

The line repeats on collection '*TB Report KeepSrc GroupsPri*', which displays all the groups belonging to Primary group. The line then explodes to display the subgroups.

```

[Line : TB Report KeepSrc Details]

  Explode      : TB Report KeepSrc Group Explosion : $$KeyExplode

```

In the part '*TB Report KeepSrc Group Explosion*', if the object is Group, then once again the line explodes to display the sub-groups or the ledgers belonging to the current sub group.

```

[Part : TB Report KeepSrc Group Explosion]

  Lines : TB Report KeepSrc Details Explosion

  Repeat : TB Report KeepSrc Details Explosion:TB Report KeepSrc SubGrp

  Scroll : Vertical

[Line : TB Report KeepSrc Details Explosion]

  Explode : TB Report KeepSrc Group Explosion : $$KeyExplode

  Explode : TB Report KeepSrc Ledger Explosion : $$KeyExplode

  Indent : If $$IsGroup Then 2*$$ExplodeLevel Else 3* $$ExplodeLevel

  Local  : Field : Default : Delete: Border

```

The part '*TB Report KeepSrc Group Explosion*' is exploded recursively. So, it is useful to keep the data at the primary interface object level.

The collections '*TB Report KeepSrc GroupsPri*' and '*TB Report KeepSrc SubGrp*' both use the same source collection '*TB Report KeepSrcGroups*'. The collections are defined as follows:

```

[Collection : TB Report KeepSrcGroups]

```

```

Type      : Group
Fetch     : Name, Parent, Closing Balance

[Collection : TB Report KeepSrc GroupsPri]

Source Collection : TB Report KeepSrc Groups
Filter           : PrimaryGrp
By              : Name: $Name
Compute        : Parent: $Parent
Keep Source     : ().

[Collection : TB Report KeepSrc SubGrp]

Source Collection : TB Report KeepSrc Groups
Filter           : SubGrp
By              : Name : $Name
Compute        : Parent : $Parent
Keep Source     : ().

[Collection : TB Report KeepSrc Ledgers]

Type      : Ledger
Child Of  : #MyGroupName1
Filter    : Zero Filter
Fetch     : Name ,Parent, Closing Balance

[System : Formula]

Zero Filter : $ClosingBalance > 0 AND NOT $$IsLedgerProfit

PrimaryGrp : $$IsSysNameEqual : Primary : $Parent

SubGrp     : $Parent = #MyGroupName1
    
```

2.2 Attribute 'Collection' change – Loop Collection

The data processing artefact of TDL, i.e., 'Collection', provides extensive capabilities to gather data from various data sources. The TDL application developers are aware that the source can be report, parent report, collection/s and external data sources like excel, XML, etc.

It is possible to gather the data from multiple collections in one collection. Till this enhancement, the collection didn't directly support the capability to gather data dynamically from multiple collections based on the object context of another collection. The functionality was achieved

using 'Filter' and 'Child Of' attributes of the 'Collection' definition. Programming using these was tedious and time consuming, and increased the code complexity as well. The new enhancement has simplified the TDL code development.

The 'Collection' attribute of 'Collection' definition has been enhanced to repeat and combine the same collection based on the number and context of objects in another collection. The present syntax of 'Collection' attribute allows us to combine and collate the data from all the collections specified as a comma-separated list, provided the number, order and data type of methods are the same in each of the collection specified in the list.

Existing Syntax

The *existing syntax* of 'Collection' attribute is as below:

Syntax

```
[Collection : <Collection Name>]
    Collection : <List of Data Collection>
```

Where,

<Collection Name> is the name of the collection.

<List of Data Collection> is the comma-separated list of data collections.

Example:

```
[Collection : GroupLedColl]
    Collection : TestGroups, TestLedgers
```

In this example, the Collection 'GroupLedColl' will contain the union of objects from both the collections 'TestGroups' and 'TestLedgers'.

The **Collection** attribute has been enhanced to dynamically gather the data collections in the context of each object of another collection. It now accepts two additional optional parameters.

New Enhanced Syntax

The *new enhanced syntax* of the 'Collection' attribute is as given below:

Syntax

```
[Collection : <Collection Name>]
    Collection : <List of Data Collection> [:<Loop Coll Name>]
                [:+<Condition for Loop Coll>]
```

Where,

<Collection Name> is the name of the collection.

<List of Data Collection> is the comma-separated list of data collections.

<Loop Coll Name> is the name of the loop collection, and is optional.

<Condition for Loop Coll> is optional. The condition is evaluated on each object of the Loop Collection.

The attribute 'Collection' has been now enhanced to repeat a collection over objects of another collection. Optionally, a condition can be specified. This condition applies to the collection on

which looping is desired. The collection on which the repetition is done is referred to as Loop Collection. The collection names in the list of collections are referred to as Data Collections. The data is populated in the resultant collection from the list of data collections.

Each data collection is gathered once for each object in the loop collection. If there are n objects in the loop collection, the data collections are gathered n times in the context of each object in the loop collection. In the resultant collection, the objects are delivered as TDL Objects. This makes it mandatory to fetch the required methods in the Data Collection.

Example:

```
[Collection : VchOfMultipleLedgers]

    Collection : VchOfLedger : LedgerListColl : ($Parent starting with "Sundry")

[Collection : VchOfLedger]

    Type      : Vouchers : Ledger

    Child of  : $Name

    Fetch    : VoucherNumber, Date, Amount
```

The collection **VchOfLedger** is the data collection. It is mandatory to fetch the required methods 'Voucher Number', 'Data' and 'Amount', in order for them to be available in the resultant collection.

It can be observed that the method \$name of loop collection **LedgerListColl** is used in 'ChildOf' attribute directly. This is because while the evaluation of 'ChildOf' attribute, the loop collection object context is available. If we are referring to the methods of the loop collection directly in the attributes SOURCE VAR, COMPUTE VAR, COMPUTE, AGGR COMPUTE, FILTER and FILTER

VAR, we cannot do so. This is because while evaluating these attributes, the loop collection object context is not available. In order to make these methods available in the Data collection, the following function has been introduced.

New Function – \$\$LoopCollObj

- **Function - \$\$LoopCollObj**

The function \$\$LoopCollObj has been introduced to refer to any method of the Loop Collection objects in the Data Collection. The Data Collection can use this function for the evaluation of expressions.

Syntax

```
$$LoopCollObj : <Method name from Loop Coll Obj>
```

Where,

<Method name from Loop Coll Obj> is name of method from the object of the loop collection.

Example:

A collection is created to gather all the vouchers of all the loaded companies as follows:

```
[Collection : Vouchers of Multiple Companies]

    Collection : VchCollection : Company Collection
```

```
Sort : Default : $Date, $LedgerName
```

Objects in collection 'Vouchers of Multiple Companies' are sorted based on date and ledger name

```
[Collection : VchCollection]
```

```
Type      : Voucher
```

```
Fetch     : Date, Vouchernumber, VoucherTypeName, Amount,
           MasterID, + LedgerName
```

```
Compute  : Owner Company : $$LoopCollObj : $Name
```

Let us examine the Data Collection definition "VchCollection". When the attribute 'Compute' is evaluated, the Loop collection object context is not available here. So, to refer to the Company Name, the function \$\$LoopCollObj is mandatory.

```
[Collection : Company Collection]
```

```
Type     : Company
```

```
Fetch    : Name
```

Use Case

Scenario: A Stock Summary Report for Group Company.

The consolidated stock summary report of all the members of a group company. The member companies of the group company can have different Unit of Measures and Currency.

Solution: The report displays stock item name, unit of measurement and combined closing balance of all members of the group company, assuming that the base currency is same. At part level, the line is repeated on the aggregate/summary collection *GrpCmpSSRepeatColl* as follows:

```
[Part : GrpCmpSSPart]
```

```
Line      : GrpCmpSSLineTitle, GrpCmpSSLineDetails
```

```
Repeat    : GrpCmpSSLineDetails : GrpCmpSSRepeatColl
```

```
Scroll    : Vertical
```

```
Common Border : Yes
```

The summary collection is defined as follows:

```
[Collection : GrpCmpSSRepeatColl]
```

```
Source Collection : GrpCmpSSLoopCollection By : StkName : $Name
```

```
By              : UOM: $BaseUnits
```

```
Aggr Compute    : ClBal : SUM : $ClosingBalance Sort : Default :
                 $StkName, $UOM
```

Since the member companies may have different UOM, the grouping is done on the same. If the UOMs are same then the 'ClosingBalance' is aggregated, else the Items are displayed as separate line items with respective UOMs.



We cannot perform aggregation directly on the resultant collection (which is created using data and loop collection). If required to do so, the same has to be used as a source collection for the aggregate/summary collection.

The source collection '*GrpCmpSSLoopCollection*' is defined as follows:

```
[Collection : GrpCmpSSLoopCollection]

    Collection : StkColl : GrpCmpColl
```

The data collection '*StkColl*' is gathered for each object of the loop collection '*GrpCmpColl*'. The collections are defined as follows:

;; Data Collection

```
[Collection : StkColl]

    Type : Stock Item

    Fetch : Name,BaseUnits, ClosingBalance
```

;; Loop Collection

```
[Collection : GrpCmpColl]

    Type : Member List : Company

    Child Of : ##SVCurrentCompany
```

Assume that currently a group company **Grp ABC** is loaded with three member companies A, B and C. The Stock Items details in each company are shown in following table:

Company Name	Stock Item	Unit of Measure	Closing Balance
Company A	Item 1	Nos	500
	Item 2	Kg	500
	Item 3	Nos	500
Company B	Item 1	Nos	400
	Item 3	Nos	800
Company C	Item 1	Nos	300
	Item 2	Nos	700
	Item 3	Nos	500

The following table demonstrates the objects in each collection:

Objects in collection GrpCmpColl	Objects in collection Stk-Coll	Objects in collection GrpCmpSSLoopCollection	Objects in collection GrpCmpSSRepeat-Coll
3 - A, B, C	All stock items of First member company i.e. A	All stock items of all member companies	Sum of Closing balance is evaluated by grouping Stock Item name and Unit Of Measure
	Item 1 - 500 Item 2 - 500 Item 3 - 500	Item 1 - 500 Item 2 - 500 Item 3 - 500 Item 1 - 400 Item 3 - 800 Item 1 - 300 Item 2 - 700 Item 3 - 500	Item 1 - 1200 Nos Item 2 - 500 Kg Item 2 - 700 Nos Item 3 - 1800 Nos

Multi Column behavior with Multi-Company data

Various reports can be generated in Tally.ERP 9 relevant to the user's business requirement. All the reports are generated in context of SVCURRENTCOMPANY, SVFROMDATE and SVTODATE. In the multi column report, the collection is gathered for each column of the report. The code complexity has been reduced with the introduction of Loop Collection in TDL language.

When the Data collections are gathered in the context of Company as Loop collection; in the resultant collection, the object context is forcefully changed to current/owner/loaded company and the report is displayed.

Consider the following example to understand the Loop Collection behaviour of multi column report for multiple companies. Assume that there are three companies A, B and C. The company A has ledgers L1 and L2, B has ledgers L3 and L4, while C has ledgers L5 and L6. The currently loaded company is A and the loop collection "My Company" has objects as A, B and C.

The collection is constructed as follows:

```
[Collection : LedCmpColl]
    Collection : MyLed : My Company
[Collection : My Led]
    Type : Ledger
    Fetch : $Name, $ClosingBalance
[Collection : MyCompany]
```

Type : Company

When the multi column report is displayed for the first time, all the ledgers are associated to the current company A forcefully, and their closing balance is displayed in the column as follows:

Ledger Name	A Closing Balance
L1	100
L2	200
L3	300
L4	400
L5	500
L6	600

When an additional column is added for company B, the report is displayed as follows:

Ledger Name	A Closing Balance	B Closing Balance
L1	100	
L2	200	
L3		300
L4		400
L5		
L6		

For this column, the collection is gathered with the current company B in context.

As a result, the closing balances of ledgers belonging to companies A and B are available and are displayed in their respective company columns. As the company C context is not available, the closing balances of ledgers L5 and L6 are not displayed at all.

When the column for company C is added, the closing balances of ledgers L1, L2, L3, L4, L5 and L6 are displayed in the respective company columns as follows:

Ledger Name	A Closing Balance	B Closing Balance	C Closing Balance
L1	100		
L2	200		
L3		300	
L4		400	

L5			500
L6			600

Points for consideration during usage

The collection attribute 'Search Key' can be specified only in the Summary Collection and not in the Data/Loop/Source Collection

The Summary Collection using source collection created using loop collection concept, can only be referred from elsewhere using the function \$\$CollectionFieldByKey. The other functions like \$\$CollectionField, \$\$CollAmtTotal are at present not supported.

If the companies have different currencies and aggregation is done, then the resultant values for the masters would not be displayed.

In case the stock items of the companies have different units of measures, and aggregation is done on them, the stock item name having different UOM would not be displayed at all in the list.

2.3 Changes pertaining to Parameter Collection

The internal collection "Parameter Collection" was used earlier in TDL at two places:

When objects in the specified scope needed to be referred from a Report or Child Report.

XML Data response, after triggering the action HTTP Post, which is used either in Success/Error Report displayed for the user.

As the 'Data Source' attribute of the Collection has been enhanced to populate it using the objects in the specified scope from a current or Parent Report, the concept of Parameter Collection in this respect does not carry relevance any more. We need to use Data Source capability, instead of Parameter Collection in codes to be written in future.

For the existing TDLs which are already using Parameter Collection, a collection has been introduced in Default TDL, which uses Data Source with scope as selected. The TDLs which are using Parameter Collection for different scopes need to make desired changes in the code.

In context of HTTP Post, the usage remains as it is.

3 User Defined Functions Enhancements

The user-defined function can use a newly introduce looping construct which iterates for the given range of values, and the action **New Object** has been enhanced to accept a logical value.

3.1 New Looping Construct – FOR RANGE

As explained earlier, TDL allows different looping constructs for varied usage. The existing loop constructs allow us to loop on the objects in collection or on the tokenized string or condition based looping of a set of statement.

There are scenarios where the looping is to be performed for a range of values. For this, a new loop FOR RANGE has been introduced. The newly introduced loop construct allows to loop on a range of numbers or date. This loop can be used to repeat a loop for the given range of specified values. The range can either be incremental or decremental. The FOR RANGE loop can be used to get the Period Collection-like functionality.

Syntax

```
FOR RANGE : <Iterator Var> : <Data type> : <StartRangeExpr> : +
          <EndRangeExpr> [ :<Increment Expr>[:<DateRangeKeyword>]]
```

Where,

<Iterator Var Name> is the name of the variable used for the iteration. This variable is created implicitly.

<Data Type> can be Number or Date only.

<StartRangeExpr> is an expression which evaluates to number or date values. It refers to the starting value of the range.

<EndRangeExpr> is an expression which evaluates to number or date values. It refers to the end value of the range.

<Increment Expr> is an expression which evaluates to a number by which the <StartRange-Expr> value is incremented. It is optional, and the default value is 1.

<DateRangeKeyword> is optional, and only applicable if the data type is Date. The values can be any one of 'Day', 'Week', 'Month' and 'Year'.

Example:

```
|
01 : FOR RANGE : IteratorVar : Number : 2 : 10 : 2
02 : LIST ADD : EvenNo : ##IteratorVar
03 : END FOR
|
```

The values 2,4,6,8,10 are added in the List variable 'EvenNo', as the range of value is 2 to 10, and the value is incremented by 2 with each iteration.

Example:

The following code snippet is used to calculate the number of weeks elapsed between System date and Current Date set in Tally.

```
|
09 : FOR RANGE : IteratorVar : Date : ##SVCURRENTDATE : $$MACHINEDATE :
          1 : "Week"
10 : LOG : ##IteratorVar
20 : INCREMENT : Cnt
30 : END FOR
50 : LOG : "No of weeks Back Dated : "+$$STRING : ##Cnt
```

```
60 : RETURN: ##Cnt
```

```
|
```

Assume that the range for this is from 15 - Mar - 2009 to 30 - Mar - 2009. The values 15-Mar-2009, 22-Mar-2009 and 29-Mar-2009 are logged, as the increment is done by a 'Week'. So, there are three iterations of the loop. The number of weeks is logged using the counter.

Example:

```
|
```

```
09 : FOR RANGE : IteratorVar : Date : ##SVFromDate:##SVToDate : 1 : "Month"
```

```
10 : LOG : $$MonthEnd : ##IteratorVar
```

```
20 : END FOR
```

```
|
```

Assume that the range for this is from 1-Jan-2009 to 5-Mar-2009. The values 31-Jan-2009, 28-Feb-2009 and 31-Mar-2009 are logged.

3.2 New Function – \$\$LoopIndex

The TDL programming community is now aware about the enhancements that have been introduced in TDL language and are efficiently implementing the same in the programs.

A vast area of possible extensions is unlocked by the User Defined Functions in TDL. User defined functions gave the sequential control in the hands of the TDL programmers. Many actions and looping constructs have been introduced in User Defined Functions in TDL. During the sequential execution, the loops are used to iterate through a set of values. TDL allows nested loops as well.

There are scenarios where the loop counter is required for some evaluation. Presently, a variable is defined and then at the end of loop, its value is incremented. This variable can be used for some calculations, if required. To avoid this inline declaration of variable which is solely used as a counter, a new function \$\$Loop Index has been introduced.

The function \$\$LoopIndex gives the count of how many times the current loop is executed. In case of nested loops, the additional parameter <outer loop index number> can be used in the inner loop to get the current iteration count of the outer loop.

Syntax

```
$$LoopIndex [:<Outer Loop Index Expr>]
```

Where,

<Outer Loop Index Expr> can be any expression which evaluates to a number. It is optional, and the outer loop index number in the nested loop hierarchy from the inner most loop to the outer most loop. For the current loop, the value is 0 by default, for the parent loop 1, and so on.

Consider following example:

```
[Function : LoopIndex Test]
```

```
|
```

```

|
05 :WALK COLLECTION :.....
|
WHILE : .....
|
|
FOR : .....
    SET : Var: $$LoopIndex LOG : ##Var
|
END FOR
SET : Var1: $$LoopIndex:1
|
END WHILE
|
|
END WALK

```

The variable **Var** will hold the count of number of times the FOR loop is executed, while the variable Var1 will have the count of 'WALK Collection' loop execution.

3.3 Enhanced Action - NEW OBJECT

The action 'New Object' takes two parameters **Object Type** and **Object Identifier**. The syntax is:

Syntax

```
NEW OBJECT : <Object Type> : [:<Object Identifier>]
```

Where,

<Object Type> is the type of the object to be created,

<Object Identifier> is the unique identifier of the object. This parameter is optional. In case this is not specified, it creates a blank object of the specified type.

The actions Save Target/Alter Target/Create Target are used along with New Object for specific usage. There are three scenarios to consider for this:

1. In case a Blank Object is created using 'New Object' without specifying the Object Identifier, the actions 'Save Target' and 'Create Target' will work, while 'Alter Target' would fail.
2. In case an existing object is brought into context by specifying Object Identifier with 'New Object', the actions 'Save Target' and 'Alter Target' will work, while 'Create Target' would fail.



Save Target' saves the current Object, whether it is a blank new Object, or an existing Object for Alteration.

- When an Object Identifier is specified with 'New Object' and the object does not exist in the database, the Action 'Save Target' fails, as 'New Object' does not create a blank object.

In order to overcome the scenario (3), the Action 'New Object' has been enhanced to accept an additional parameter 'Force Create Flag' along with the Object Identifier. This forces the creation of an empty blank object, in case the Object with Identifier does not exist in the database.

Syntax

```
NEW OBJECT : <Object Type> : [:<Object Identifier>[:<Forced Create Flag>]]
```

Where,

<Object Type> is the type of the object to be created,

<Object Identifier> is the unique identifier of the object.

<Forced Create Flag> is an Optional Flag and is required only if <Object Identifier> is specified. If the Flag is set to TRUE, then if the object identified by <Object Identifier> exists, the object is altered; otherwise, a new empty object of the specified type is created.

Example:

```
|
01: NEW OBJECT : Group: ##EGroupName : TRUE
02: SET VALUE: Name: ##NGroupName
03: SAVE TARGET
|
```

If the ledger identified by *'##EGroupName'* exists in Tally database, then the objects are altered by the action SAVE TARGET; else, a new object is created as the Forced flag is set to 'YES'.

4 New Functions

A new function **\$\$SysInfo** has been introduced to get any system-related information.

4.1 Function - \$\$SysInfo

The TDL Platform has provided TDL Programmers with various functions that accept zero or more parameters, process them and return the appropriate result. Apart from the Object/ Data Manipulation, there is much system-related information that is required to be retrieved.

Functions like **\$\$MachineDate**, which returns the System Date, **\$\$MachineTime**, which returns the System time, etc., are now supported by platform. Few more system related functions like Machine Name, Windows User Name, IP Address, etc., are required by the TDL Programmers.

Such system-related information is bundled together into a single function **\$\$\$SysInfo**, designed to accept different parameters based on requirement, and subsequently return the desired result.

Syntax

```
$$$SysInfo : <Parameter>
```

Where,

<Parameter> can be any one of ApplicationPath, CurrentPath, SystemDate, SytemTime, SystemTimeHMS, SystemName, IsWindows, WindowsVersion, WindowsUser, IPAddress, MACAd- dress.

Example:

```
$$$SysInfo : MachineName
```

This will return the Machine Name, in which current copy of Tally is running.

Example of each parameter has been explained considering the following system details:

Application Path is **C:\Tally.ERP9**

Data Path is **C:\Tally.ERP9\Data**

System Date is **27-Sep-2009**

System Time is **18:27**

System Time in Hours, Minutes and Seconds is **18:27:36**

System Name is **TallySystem1**

Operating System is **Windows 7 / Windows XP / Windows 2000 / Windows Vista**

Version of Windows is **5.1 (2600)**

User logged into Windows is **Tally.User1**

Network IP Address is **192.168.1.17**

Network Adapter's MAC Address is **0720fhac027a**

List of Parameters with corresponding Result

ApplicationPath – Returns the Folder path, from where the current copy of Tally is executed.

Example:

```
$$$SysInfo : ApplicationPath returns C:\Tally.ERP9
```

CurrentPath – Returns the Data path configured in Tally.INI residing within the application path.

Example:

```
$$$SysInfo : CurrentPath returns C:\Tally.ERP9\Data
```

SystemDate – Returns the current System/ Machine Date.

Example:

```
$$$SysInfo : SystemDate returns 27-Sep-2009
```


SystemTime – Returns the current System/ Machine Time.

Example:

```
$$SysInfo : SystemTime returns 18:27
```

SystemTimeHMS – Returns the current System/ Machine Time in Hour Minute Second Format.

Example:

```
$$SysInfo : SystemTimeHMS returns 18:27:36
```

SystemName – Returns the current System/ Machine Name.

Example:

```
$$SysInfo : SystemName returns TallySystem1
```

IsWindows – Returns YES only if the current Operating System is Windows, else returns NO.

Example:

```
$$SysInfo : IsWindows returns Yes
```

WindowsVersion – Returns the current Windows Version with the Build Number.

Example:

```
$$SysInfo : WindowsVersion returns 5.1 (2600)
```

WindowsUser – Returns the Name of the User who has logged into the current Windows session.

Example:

```
$$SysInfo : WindowsUser returns Tally.User1
```

IPAddress – Returns the Network IP Address of the current system.

Example:

```
$$SysInfo : IPAddress returns 192.168.1.17
```

MACAddress – Returns the Network Adapter's Media Access Control Address of the current system.

Example:

```
$$SysInfo : MACAddress - 0720fhac027a
```

Corresponding Functions *ApplicationPath*, *CurrentPath*, *MachineDate*, *MachineTime*, *IsWindows*, *WindowsVersion*, *WindowsUser*, *IPAddress* and *MACAddress* are alternative functions available in default TDL. These functions are deprecated from platform.

What's New in Release 1.52

In this release, major enhancements have taken place at the Collection level and in the User Defined Functions. Further sections talk in depth about the usage of **Data Source** attribute in Collection and the various **Looping Constructs** inside a Function.

Few generic built-in functions - **\$\$AccessObj**, **\$\$FirstObj** and **\$\$LastObj** have been introduced. Https client capability has been enhanced in Tally to exchange data with other applications securely. Https sites can be used for ftp upload, posting request and receiving data in collection.

1. Collection Enhancements - Attribute 'Data Source' enhanced

The TDL programmers are aware that data from various data sources can be gathered in a collection. Till the release 1.5, the data sources were Tally database, XML, HTTP, ODBC and DLL. After the multi-line selection capability was introduced, a report or a function could be launched from the current report based on the specified scope. The different scopes that could be specified were Selected lines, Current line, Unselected lines, etc.

Now, the objects can also be gathered from the report or parent report using the **Type** parameter for the Collection attribute **Data Source**. This new capability allows the access of specific objects of a report from anywhere; like Functions, Subsequent report or the Current report itself. The **Data Source** attribute of the collection has been enhanced to support these two data sources, in addition to the existing data sources. The collection can be created directly from the specified data source and can be displayed in a report.

Syntax

```
Data Source : <Type> : <Identity> [:<Encoding>]
```

Where,

<Type> specifies the type of data source, e.g., File Xml, HTTP XML, Report, Parent Report.

<Identity> can be the file path or the scope keywords. If the type is **File XML** or **HTTP XML**,

<identity> is the data source file path. If the type is **Report** or **Parent Report** then the scope keywords '*Selected Lines*', '*UnSelected Lines*', '*Current Line*', '*All Lines*', '*Line*' and '*Sorting Methods*' are used as identity.

<Encoding> can be ASCII or UNICODE. It is Optional. The default value is UNICODE. If the data source type is Report or Parent report, the encoding format is ignored.

1.1 Existing Data Source Types

Example: XML file as data source

```
[Collection : My XML Coll]
```

```
Data Source : File XML : "C:\MyFile.xml"
```

In this code snippet, the type of file is '*File XML*', as the data source is XML file. The encoding is **Unicode** by default, as it is not specified.

Example: HTTP as data source

```
[Collection : My XML Coll]
```

```
    Data Source : HTTP XML : "http:\\localhost\\MyFile.xml" : ASCII
```

In this code snippet, the type of file is '*HTTP XML*', as the data source is obtained through HTTP. The encoding of the file '*MyFile.XML*' is ASCII.

While specifying the URL, now the **https** site can be given in Collection attributes **Remote URL** and **Data Source**.

1.2 Data Source Types Introduced

Example: Report as data source

```
[Collection : My Report Coll]
```

```
    Data Source : Report : Selected Lines
```

The selected objects from the current report in which the collection is accessed, is the data source for the collection '*MY Report Coll*'.

Example: Parent Report as data source

```
[Collection : My Parent RepColl2]
```

```
    Data Source : Parent Report : UnSelected Lines
```

The objects associated with all the unselected lines from the parent report are gathered for the collection '*My Parent RepColl2*'.

The objects of the report with the given scope can be accessed from the report and functions which are called from the report.

2. Enhancements in User Defined Functions

In this release, 'Dynamic action support' has been provided for simple actions inside a function and the looping construct **Walk Collection** has been enhanced. New looping constructs **FOR COLLECTION** and **FOR TOKEN** have been introduced as well.

There are scenarios when the collection name is to be obtained from an expression while performing a Walk. **WALK COLLECTION** has been enhanced to provide this functionality. **FOR COLLECTION** loop has been introduced to walk the collection for a specific value. **FOR TOKEN** loop walks on the tokens within a string separated by a specified character.

2.1 Attribute 'Walk Collection' Enhanced

The 'Walk Collection' attribute has been enhanced to accept Collection Name as an expression and an additional logical parameter. For example, now the collection name can be passed as parameter to the function while executing it.

Syntax

```
    Walk Collection : <Expression> [:<Rev Flag>]
```

Where,

<Expression> can be any expression which evaluates to a collection name.

<Rev Flag> can be any expression which evaluates to a logical value. If it is **True**, then the collection objects are traversed in reverse order. This parameter is optional. The Default value is **False**.

Example:

```
[Function : Test Function]

    Parameter : parmcoll
    |
    |
    05 : WALK COLLECTION : ##parmColl : Yes
```

The collection name is passed as parameter to the function '*Test function*' and is walked in reverse order.

The code snippet to call the function '*Test function*' from a key is as follows:

```
[Key : DC Call Function]

    Key      : Enter

    Action : CALL : Test Function : ##CollName
```

The collection name is passed through the variable '*CollName*'.

2.2 Dynamic Actions

Prior to Release 1.52, the dynamic action capability was available for global actions. It was possible to specify the Action Keyword and Action parameters as expressions. This allowed the programmer to execute actions based on dynamic evaluation of parameters. The 'Action' keyword can as well be evaluated dynamically.

The dynamic action capability is now introduced for simple actions inside a function. Expressions can be used to evaluate the name of the Action as well as Action parameters. The new action with Action keyword has been introduced to achieve this.

Syntax

```
Action : <Expression> : <Action Parameters>
```

Where,

<Expression> is any expression evaluating to a simple action name like LOG DISPLAY, etc.

<Action Parameters> are parameters required by the action passed through an expression.

Example:

```
[Function : ObjFunc]

    |
```

```

|
02 : ACTION : LOG : "$" + ##pObjMethod

```

Inside a function, a global action can be called using dynamic action capability. In this case, the expression specified in the dynamic action is evaluated in the context of the function, and then the global action is executed.

The context of function elements like Variables, Objects, etc., can be used while calling a global action dynamically. For example, the variable name or methods of an object can be passed as parameter while executing the dynamic action.

Example:

```
[Function : Dynamic Action Within Function]
```

```

Variable : DA Logical : Logical
|
|
40       : SET: DA Logical : Yes
50       : ACTION : Display : if ##DALogical then "Trial Balance" else +
                                         "Balance Sheet"

```

In function 'Dynamic Action Within Function', first the expression is evaluated and then based on the value, the report 'Trial Balance' is displayed.

2.3 Looping Constructs 'For Collection' and 'For Token' introduced

Two new looping constructs - **For Collection** and **For Token** have been introduced in user defined functions.

Looping Construct - FOR COLLECTION

When WALK COLLECTION is used inside a function, the object of collection is set as the current object in the context of iteration, i.e., the loop is executed for each object in the collection, making it as the current context.

The newly introduced FOR COLLECTION provides a context free walk, as the current object is not set as the current object context while looping. It loops on the collection for the specific value and returns the value in the iterator variable. The value of the iterator variable can be referred to by the actions inside the For Collection loop.

Syntax

```
FOR COLLECTION : <IteratorVar> : <CollExprn> [:<Value Exprn> : <Rev Flag>]
```

Where,

<Iterator Var> is the name of the variable which is used for the iteration. This variable is created implicitly.

<Coll Exprn> can be any expression which evaluates to a collection name.

<Value Exprn> can be any expression, whose value is returned in the iterator variable. If the value expression is not specified, the name of the object is returned.

<Rev Flag> can be any expression which evaluates to a logical value. If it is **True**, then the collection objects are traversed in reverse order. This parameter is optional. The Default value is **False**.

Example:

```
[Function : Test Function]
|
|
30 : FOR COLLECTION : i : Group : $ClosingBalance > 1000
40 : LOG : ##i
50 : END FOR
```

The value **Yes** is logged in the file 'TDLFunc.log' if the closing balance is greater than 1000, else **No**.

Looping Construct - FOR TOKEN

The looping construct FOR TOKEN is used to walk a String expression separated by a delimiter character. It loops on a String expression and returns one value at a time. The value is returned in the iterator variable.

Syntax

```
FOR TOKEN : <IteratorVar> : <SrcStringExprn> [:<DelimiterChar>]
```

Where,

<Iterator Var Name> is the name of the variable used for iteration. The variable is created implicitly.

<Src String Exprn> can be any string expression separated by a *<delimiter Char>*.

<Delimiter Char> can be any expression which evaluates to a character used for separating the string expression. It is optional. The default separator char is ':'.

Example:

```
[Function : Test Function]
|
|
01 : FOR TOKEN : TokenVar : "Tally : Shopper : Tally Developer" : ":"
02 : LOG : ##TokenVar
03 : END FOR
```

This code snippet will give the output as shown below:

Tally
 Shopper
 Tally Developer



The same List is considered in explaining the further examples.

3. New Functions

New functions \$\$AccessObj, \$\$FirstObj and \$\$LastObj have been introduced in this release.

3.1 Function - \$\$AccessObj

The capability to access data objects associated with Interface objects was introduced in Tally.ERP 9. The attribute 'Access Name' is used to specify name to 'Part' or 'Line' Definition. This name can be used to refer to the Data Object associated with the Part or the Line.

A new function \$\$AccessObj has been introduced to evaluate the specified formula in the context of the Interface object identified by the given definition type and access name.

Syntax

```

    $$AccessObj : <Definition Type> : <AccessNameFormula> :
                <Evaluation Formula>
    
```

Where,

<Definition Type> can be Part or Line.

<Access Name Formula> can be any formula which evaluates to a string.

<Evaluation Formula> is a formula which is evaluated in the context of the object identified by the definition type and the access name.

Example:

```

[Line : AccessObj]

    Fields : AccessObj AccessName : "A01"

    On      : Focus : Yes : CALL      : AccessObj

[Field : AccessObj]

    Set As : $Name

[Function : AccessObj]

    Variable : AccessObj : String

    00 : SET : AccessObj : $$AccessObj : Line : "A01" : $Name

    10 : LOG : ##AccessObj
    
```


The Line 'AccessObj' is identified by the access name 'AO1'. The access name is used while evaluating the value of \$Name.

3.2 Functions - \$\$FirstObj and \$\$LastObj

The objects of the collection are available in the context of repeat line or while performing a walk inside a function. The functions \$\$FirstObj and \$\$LastObj can be used to find the first or the last object of the collection respectively.

Function - \$\$FirstObj

The function **\$\$FirstObj** returns the value of the specified method for the first object of the collection.

Syntax

```
$$FirstObj : <MethodName>
```

Where,

<Method Name> is the name of a method of the current object in context.

Example:

```
40 : LOG : "First Object : " + $$FirstObj : $Name
```

\$\$FirstObj logs the name of the first object of the collection, which is used in **Walk Collection**.

Function - \$\$LastObj

The function **\$\$LastObj** returns the value of the specified method for the last object of the collection.

Syntax

```
$$LastObj : <MethodName>
```

Where,

<Method Name> is the name of a method of the current object in context.

Example:

```
50 : LOG : "Last Object: " + $$LastObj : $Name
```

The function **\$\$LastObj** logs the name of the last object of the collection, which is used in **Walk Collection**.

4. https URL support in Tally

Https client capability has been enhanced in Tally to exchange data with other applications securely.

Https sites can be used for ftp upload, post request and receiving the data in collection. Now, data can be uploaded to https site, or request to the https site can be sent, using the action HTTP Post. The URL for the https site can be specified while gathering data in a collection.

Example: Upload

In the upload configuration screen, the URL for https site can be given as shown:



Figure 1. Upload Configuration Screen

Example: Action HTTP Post

[Button : PostButton]

Key : Ctrl+K

Action : HTTP Post : @@MyURL : ASCII : HTTP Post ReqRep : +

HTTP Post Response Report1 : HTTP Post Response Report

[System : Formula]

MyURL : "https://www.testserver.co.in/CXMLResponse as per tally.php"

Example: In collection

[Collection : https Coll]

Remote URL : "https://www.testserver.co.in/TestXML.xml"

What's New in Release 1.5

1. Collection Enhancements

TDL supports the hierarchical database structure. While designing any report, the objects are first populated in the collection, before being displayed.

TDL also supports the concept of aggregate/summary collection, for creating summarized reports. In the aggregate collection, during evaluation, the 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 sometimes impossible as well.

The newly introduced concept of collection level variables provides Object-Context Free processing. The values of these in-line 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 values of these variables can be accessed from anywhere while evaluating the current collection objects. Sometimes, it is not possible to get the value of the object from the current object context. In such scenarios, these variables are used.

1.1 Collection Attributes - Source Var, Compute Var, Filter Var

Attribute - Source Var

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

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression formula, which evaluates to a value of 'Variable' data type.

Example:

```
Source Var : Log Var : Logical : No
```

The value of the variable **LogVar** is set to NO.

Attribute - Compute Var

The attribute **Compute Var** evaluates the value of the variable based on the sub-object of the source object.

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression formula which evaluates to a value of 'Variable' data type.

Example:

```
Compute Var : IName:String : if ##LogVar then $StockItemName else ##LogVar
```

Attribute - Filter Var

The attribute **Filter Var** evaluates the value of the variable based on the objects available in the collection, after the evaluation of attributes **Fetch** and **Compute**.

Syntax

```
Filter Var : MyFilVar : <Data Type> : <Formula>
```

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable.

<Formula> can be any expression formula which evaluates to a value of the 'Variable' data type.

Example:

```
Filter Var : Fin Obj Var : Logical : $$Number:$BilledQty > 100
```

1.2 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 the functions \$\$Owner, \$\$ReqObject, \$\$FilterValue, \$\$FilterAmtTotal, etc., can be reduced.

1.3 Usage of the Collection attributes - Source Var, Compute Var, Filter Var

In this section, the use cases where the collection attributes can be used, are explained.

Usage of 'Compute Var' in Simple Collection

When **Compute Var** is used in a simple collection, then before populating the objects in the collection, 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. **Type** attribute is evaluated, and the objects of the specified type are identified.
2. **Compute Var** is evaluated and the name of the first object, i.e., the 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 in the variable 'CmpVarColl'.
4. Steps 2 and 3 are repeated for each group in the collection 'Test ComVar'.
5. After computing the value of the method \$MyAmt for each group, the collection is populated with the objects.

The variable 'CmpVarColl' can also be referred in the **By**, **Aggr Compute** and **Filter** attributes of the collection.

Usage of Source Var, Compute Var and 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. Value of the variable **SrcVar** is evaluated, and referred in the **Filter** attribute of the collection 'CFBK Voucher'.
2. In the collection 'Smp Stock Item', the value of the 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 the variable **IName** is evaluated. If Source Variable **Src Var** contains "12", then the variable **IName** stores only the **StockItemName** method, else it stores **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 the method **\$BilledQty** is computed.
7. The variable **Fin Obj Var** retains a logical value, if the method **\$BilledQty** is greater than 100.
8. Based on the value of **Fin Obj Var**, 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 functionality of variables, and their 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 Variables**.
- System Level – commonly referred to as **Global variables**.
- Function Level – Local variables used inside User-defined functions.

The variable framework has been enhanced to support a new type of 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 an alias of the attribute **List Variable**.

Syntax

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

Where,

<Variable Name> is the name of the variable.

<Data Type> is the data type of the variable. It is Optional. If it is specified, a separate Variable definition is not required. If not specified, a variable definition with 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 functions for data manipulation, which will be explained in the following section.

List Variable Manipulation

List variable supports various data manipulation operations, which include:

1. Adding/Deleting Values – Actions **List Add** and **List Delete**
2. Populating List Var from a Collection – Action **List Fill**
3. Accessing List Variable values – Function **\$\$ListValue**
4. Sorting the values in the List Variable

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 values in a List variable. It adds a 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 overwritten. **LIST SET** is an alias for the action LIST ADD.

Syntax

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

Where,

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

<Key Formula> can be any expression formula which evaluates to a 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 the 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 the further examples.

To add multiple values dynamically in the list variable, looping constructs **WHILE**, **WALK COLLECTION**, etc., can be used. **LIST REMOVE** is an alias for LIST DELETE.

Action - LIST DELETE

The action **LIST DELETE** is used to delete values from the List variable. It allows to delete a single value at a time or all the values in one go.

Syntax

```
LIST DELETE : <List Var Name> [:<Key Formula>]
```

Where,

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

<Key Formula> can be any expression formula which evaluates to a 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, the list is reset.

Example: 1

```
LIST DELETE : TestFuncVar : "Office"
```

The value identified by the key 'Office' is deleted from the list variable 'Test Func Var'.

Example: 2

```
LIST DELETE : TestFuncVar
```


All the values in the list variable **TestFuncVar** are removed. The list variable is empty after the execution of the action.

Populating List variable from a collection

Instead of using looping constructs, multiple values from a collection can be added to the list variable using one statement. Action **LIST FILL** is used for the same.

Action - LIST FILL

It is used to add multiple values from a collection to the List Variable.

Syntax

```
LIST FILL : <List Var Name> : <Collection Name> : <Key Formula> :  
          <Value Formula>
```

Where,

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

<Collection Name> is the name of collection from which values are fetched to fill the list variable.

<Key Formula> can be any expression formula which evaluates to a string value.

<Value Formula> can be any expression formula which returns a value. The data type of the value must be the same as that of the List variable.

The action LIST FILL returns the number of items added to the list variable.

Example:

```
LIST FILL : TestFuncVar : Group : $Name : $Name
```

Accessing List variable values

To access values from a list variable, a function is to be used. TDL provides different functions to fetch the value from a list variable, identified by the given key.

Function - \$\$ListValue

\$\$ListValue returns the value identified by the given key in the list variable.

Syntax

```
$$ListValue : <List Var Name> : <Key Formula>
```

Where,

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

<Key Formula> can be any expression formula which evaluates to a string value.

Example:

```
$$ListValue : TestFuncVar : "Mobile"
```

In this example, the function returns the values identified by the key 'Mobile' from the list variable 'TestFuncVar', when the function is executed.

Sorting values in a List variable

By default, the values in the list variable are sorted in the order of entry. TDL provides the facility to sort the values in the list variable either by key or by value. The data type can be specified while

sorting based on key. Following actions allow to change the sort order:

- List Key Sort
- List Value Sort
- List Reset Sort

These actions accept three parameters. First parameter is the **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** OR **String**.

Date data type accepts any valid date format. If it is not of 'Date' data type and starts with a number or a decimal, then it is assumed as **Number**. If it is neither 'Date' nor 'Number', then it is 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 based on key value. If the data type specified while sorting the list is different than the original, then this action will temporarily convert the original data type to the specified data type while comparing the elements for sorting the list and the list will be sorted based on the new data type specified. The original list and the key data type remains as it is, on which a new sorting can be applied, based on some other data type, at any other point of time. **LIST SORT** is an alias of the action LIST KEY SORT.

Syntax

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

Where,

<List Var Name> is the name of the 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 is 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, 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 the items in the list variable is used for comparison.

```
LIST KEY SORT : Test Func Var : YES : Number
```

In this example, the action LIST KEY SORT will convert the key to ZERO (0) for all the list items while comparing, as all the keys are of type **String**. 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**, the list is sorted based on the key values, else it converts them to ZERO and sorts the list as per the order of insertion.

Action - LIST VALUE SORT

Action **LIST VALUES SORT** sorts the list items based on 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 VALUESORT : <List Var Name> [:<Asc/Desc flag> : <Key Data Type>]
```

Where,

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

<Asc/Desc> can be YES/NO. YES is used sort the list in ascending order and NO for descending. It is optional. If the flag is not specified, then the default order is ascending.

<Key Data Type> can be String, Number, etc. It is optional.

Example:

```
LIST VALUE SORT : Test Func Var : YES : String
```

The values in the list variable are sorted in ascending order of values.

Action - LIST RESET SORT

The action **LIST RESET SORT** retains the sorting back to the order of insertion.

Syntax

```
LIST RESET SORT : <List Var Name>
```

Where,

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

Example:

```
LIST RESET SORT : Test Func Var
```

Here, the action resets the sort order of the list variable 'Test Func Var' to the order of insertion.

2.2 Functions Used with List Variables

TDL supports some functions for general operations like finding the total number of items in a list, checking whether the last action was successful, etc.

Function - \$\$ListValue

As explained earlier in section 2.1, this function is used to access values from a list variable.

Function - \$\$ListCount

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

Syntax

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

Where,

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

Example:

```
$$ListCount : TestFuncVar
```

The action returns the number of items in the list variable 'Test func Var', when it is executed.

Function - \$\$ListFind

The function **ListFind** is used to search if the value belonging to a specific key is available in the list variable. If the key is found, \$\$ListFind returns TRUE, otherwise it returns FALSE.

Syntax

```
$$ListFind : <List Var Name> : <Key Formula>
```

Where,

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

<Key Formula> can be any expression formula, which evaluates to a string value.

Example:

```
$$ListFind : TestFuncVar : "Mobile"
```

It returns TRUE if the key '**Mobile**' is present in list variable '**Test func Var**', else returns FALSE.



*Function **\$\$LastResult** is used to check if the last executed action was successful.*

- *If the last action that was 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 was executed is LIST FILL, then \$\$LastResult returns the number of items inserted in the list variable.*

2.3 Constructs introduced in Functions for List Var**Looping Construct - FOR IN**

The FOR IN loop is supported to iterate the values in the list variable. The number of iterations depends on the number items in the list variable.

Syntax

```
FOR IN : <Iterator Var Name> : <List Var Name> ..END FOR
```

Where,

<Iterator Var Name> is the name of variable used for iteration. The variable is created implicitly.

<List Var Name> is the name of the 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**".

3.1 'Action' Keyword

The 'Action' keyword allows the programmer to execute actions based on dynamic evaluation of parameters. The syntax for specifying the same is as given below:

Syntax

```
Action : <Action Keyword Expression> : <Action Parameter Expression>
```

Where,

<Action> is the keyword **Action** to be used for Dynamic Actions usage.

<Action Keyword Expression> is an expression evaluating to an Action Keyword.

<Action Parameter Expression> is an expression evaluating to Action Parameters.

We can specify and initiate an Action from the following:

- Menu Item
- Key Definition
- In a User Defined Function

At present, the capability is valid for:

- Global Actions like Display, Alter, etc.
- Global Actions inside User Defined Functions

Example:

1. Dynamic Actions in Key/Button Definition

```
[Button : Test Button]
```

```
Key      : F6
```

```
Action : Action : Display : @@MyFor
```

*;; The Button **Test Button** initiates a dynamic Action which takes the parameter as a formula.*

```
[System : Formula]
```

```
MyFor   : if ##SVCcurrentCompany CONTAINS "ABC" Then "BalanceSheet" +
                                                else "TrialBalance"
```



Observe the usage of **Action** keyword twice in this example. The first usage is the **attribute** "Action" for 'Key' definition. The second is the **keyword** "Action" introduced specifically for executing Dynamic Actions.

1. 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 have been introduced - \$\$TgtObject and \$\$ContextKeyword.

4.1 Function – \$\$TgtObject

In TDL, normally all evaluation is done in context of the Context object. With the introduction of aggregate collections and user-defined functions, apart from the Requestor Object context and Source Object context, now the Target Object context is also available.

The object which is being populated or altered is referred to as the Target object. In simple collections, the Source Object and the Target Object are both the same. In case of aggregate collections and user-defined functions, the Target Object is different.

There are scenarios where the expression needs to be evaluated in the context of the Target object. In such cases, the function **\$\$TgtObject** can be used.

Function – \$\$TgtObject

The new Context Evaluation function \$\$TgtObject evaluates the expression in the context of the Target Object. Using \$\$TgtObject, values can be fetched from the target object without making the target object as the context object.

Syntax

```
$$TgtObject : <String Expression>
```

Where,

<String Expression> is the expression which will be evaluated in the context of Target Object.

Usage of \$\$TgtObject in User Defined Functions

In a user defined function, while setting the method values of target object, the expression needs to be evaluated in the context of the target object itself. **\$\$TgtObject** is used in this case.

Example:

Ledgers 'Party 1' and 'Party 2' are having some opening balance. The requirement is to add the opening balances of both ledgers and set the resultant value as the opening balance of Party 2.

```
[Function : Sample Function]
```

```
Object : Ledger : "Party 1" : NEW OBJECT : Ledger : "Party 2" +
      : SET VALUE : OpeningBalance : $OpeningBalance +
      $$TgtObject : $OpeningBalance : ACCEPT ALTER
```

;; By prefixing \$\$TgtObject to the opening balance, the closing balance of the Target Object, i.e., Party 2, is retrieved.

Here, 'Party 1' is the Source object and 'Party 2' is the Target object. The opening balance of 'Party 2' is accessed using the expression **\$\$TgtObject:\$OpeningBalance**.

Usage of \$\$TgtObject in Collections

In simple collections, the source object and the target object are both the same. In case of aggregate collections and user defined functions, the target object is different.

The function **\$\$TgtObject** allows to access 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 the target object.

The function **\$\$TgtObject** is useful when the values are to be populated in collection, based on the values that have been computed earlier. In aggregate collections, 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 the item is sold and the maximum amount is received. The collection is defined as follows:

```
[Collection : Src Voucher]
```

```
Type      : Vouchers : VoucherType
ChildOf   : $$VchTypeSales
```

```
[Collection : Summ Voucher]
```

```
Source Collection : Src Voucher

Walk              : Inventory Entries

By               : ItemName : $StockItemName

Aggr Compute     : MaxDate : SUM : IF $$IsEmpty : $$TgtObject : $ItemDet +
                  OR $$TgtObject : $ItemDet < $Amount THEN $Date ELSE +
```

`$$TgtObject : $MaxDate Aggr Compute : ItemDet :`

`MAX : $Amount`

While creating a collection “Summ Voucher”, `$$TgtObject` is used to get the date on which the maximum sales amount is received for each stock item. `$ItemDet` gives the maximum amount received for individual item. In the condition checking part, if the evaluated `$ItemDet` is empty for the stock item or 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 the 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` can be 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 the **Report** definition, then by default, it returns the name of the Report.

Example:

`[Report : Context Keyword Function]`

`Form : Context Keyword Function`

`Title : "New Function Context Keyword"`

|

|

`[Field : Context Keyword Function]`


```
Use      : Name Field
Set As   : $$ContextKeyword
```

In this example, 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, will be returned.

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 a new number in fields using dynamic tables such as Tracking number, Order No, etc.

Syntax

```
Trigger : <Report Name> : <Trigger Condition>
```

Where,

<Report Name> is the name of the report which is invoked if **<Trigger Condition>** is True. The value entered in the Output 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 *FieldTrigger*, 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. This attribute allows to add values to the dynamic table through an expression or user-defined functions.

Attribute - Trigger Ex

The **Trigger Ex** attribute allows to add values to the dynamic table through an expression or user- defined function.

Syntax

```
TriggerEx : <Value-expression> : <Trigger Condition>
```

Where,

<Value Expression> is an expression/function which evaluates to a String if **<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** have been introduced to log the object, its method and collection contents.

6.1 Action - Log Object

The action **Log Object** has been introduced as a Global action. It accepts a filename as the 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 the file, along with the path in which the log is created. If no file name is specified, the contents of the object are logged in "**TDLfunc.log**" when logging is disabled, otherwise it logs into the **Calculator** pane.

<Overwrite Flag> is used to specify whether the contents should be appended or overwritten. The default value is **NO**, which appends the content in the file. If set to **YES**, the file is overwritten.

Example:

```
[Function : FuncLedExp]
|
Object : Ledger
|
10 : Log Object : LedgerObj.txt
```

6.2 Action - Log Target

The action **Log Target** is a function-specific action. It accepts filename as a parameter. In this file, the log of the object, its method and collection is created for the target object.

Syntax

```
Log Target [:<path\filename> [:<Overwrite Flag>]]
```

Where,

<path/filename> is optional. It accepts the name of the 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 into the **Calculator** pane.

<Overwrite Flag> is used to specify whether the contents should be appended or overwritten. The default value is **NO**, which appends the content in the file. If set to **YES**, the file is overwritten.

Example:

```
[Function : FuncLedExp]
|
05 : Set Target
|
10 : Log Target : LedgerObj.txt
```

7. Tally Command Line Parameters

While executing Tally, now the following command line parameters can also be given:

- /NOINITDL

This parameter will start Tally.ERP 9 without loading any **TDL** specified in the Tally.ini file.

Syntax

```
/NOINITDL
```

- /TDL

This parameter will start Tally.ERP 9 with 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 the TDL file, along with the path.

- /NOINILOAD

This parameter will start Tally.ERP 9 without loading any **Company** specified in the Tally.ini file.

Syntax

```
/NOINILOAD
```

- /LOAD

It starts Tally.ERP 9, with the specified company loaded, and can be specified multiple times.

Syntax

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

- /VARIABLE

This parameter allows to specify inline system variables of the specified data type, and can be specified multiple times.

Syntax

```
/VARIABLE : <Variable Name> : <Data Type>
```

Where,

<Variable Name> is the name of the inline variable. It must be unique.

<Data Type> is any of the primary data types.

- /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 any of the primary data types.

- /NOGUI

This parameter hides the GUI (Graphical User Interface) of Tally. It performs the specified ACTION without showing the Tally interface based on a non-GUI or GUI action. It starts Tally without showing the Tally window, performs the action and exits tally for non-GUI actions like executing a batch of job. If the action is a GUI action which invokes a report, menu or a message box, then the Tally window will be shown until the user quits.

- /ACTION

This parameter starts the Tally application with the specified action and it quits the Tally application when the user exits.

Syntax

```
/ACTION : <Action Name> [:<Action Parameter>]
```

Where,

<Action Name> is the name of any of the Global actions.

<Action Parameter> is optional. It has to be specified based on the action.

- **/PREACTION**

This parameter starts Tally, loads the company and executes the specified action before displaying the Main Menu of Tally.

Syntax

```
/PREACTION : <Action Name> [:<Action Parameter>]
```

Where,

<Action Name> is the name of any of the Global actions.

<Action Parameter> is optional. It has to be specified based on the action.

- **/POSTACTION**

This parameter starts Tally, loads the company and executes the specified action when the user quits Tally.

Syntax

```
/POSTACTION : <Action Name> [:<Action Parameter>]
```

Where,

<Action Name> is the name of any of the Global actions.

<Action Parameter> is optional. It has to be specified based on the action.



- i. Only one of the action parameters can be specified at a time.
- ii. The actions specified with **/PREACTION** and **/POSTACTION** are not executed 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 Tally.exe is available. The corresponding TDL file "BackUP.txt" for functions is available in the sample folder.

- **/NOINITDL & /TDL**

```
"C:\Tally.ERP 9\Tally.exe" /NOINITDL /LOAD:00009 "/TDL:C: \Tally.ERP 9
\TDL \SecurityTDL.txt" /TDL:MasterTDL.txt
```

This command expression ignores all the TDLs specified in Tally.ini file while loading Tally. It starts Tally application and loads the TDLs 'SecurityTDL.txt' and 'MasterTDL.txt'.

- **/NOINILOAD with /LOAD**

```
"C:\Tally.ERP 9 \Tally.exe" /NOINILOAD /LOAD:00009
```

This command expression ignores all the companies specified in Tally.ini file while loading Tally. It starts Tally application and loads the company identified by **00009**.

- /VARIABLE

```
"C:\Tally.ERP 9 \Tally.exe" /LOAD:00009 /VARIABLE : MyLogicalVar : Logical
```

This command expression starts the Tally application, and with a logical variable **MyLogicalVar**.

- /SETVAR and /ACTION

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

This command expression sets the value of the variable **ExplodeFlag** to **YES** and directly displays the **Trial Balance** report.

- /PREACTION

```
"C:\Tally.ERP9 \Tally.exe"/LOAD : 00009/PREACTION : CALL : BackupBeforeEntry
```

This command expression starts Tally application, loads the company identified by **00009** and calls the function "**BackUpBeforeEntry**" before displaying the main menu.

- /POSTACTION

```
"C:\Tally.ERP 9 \Tally.exe" /LOAD:00009 /POSTACTION : CALL : BackupOnExit
```

This command expression starts Tally application, loads the company **00009**, and calls the function **BackupOnExit**, when the user quits Tally.

- /NOGUI

```
"C:\Tally.ERP 9 \Tally.exe" /NOGUI /LOAD : 00009 /ACTION :
CALL : BackupSchedule
```

This command expression starts the Tally application, and executes the function **BackupSchedule**, without displaying the Tally window.

Appendix

Objects can be composed of methods and collection. The collection can be made up of objects which is again a combination of methods and collection and so on. This chain can go up to any number of levels. The following diagram represents the structure of an object in general.

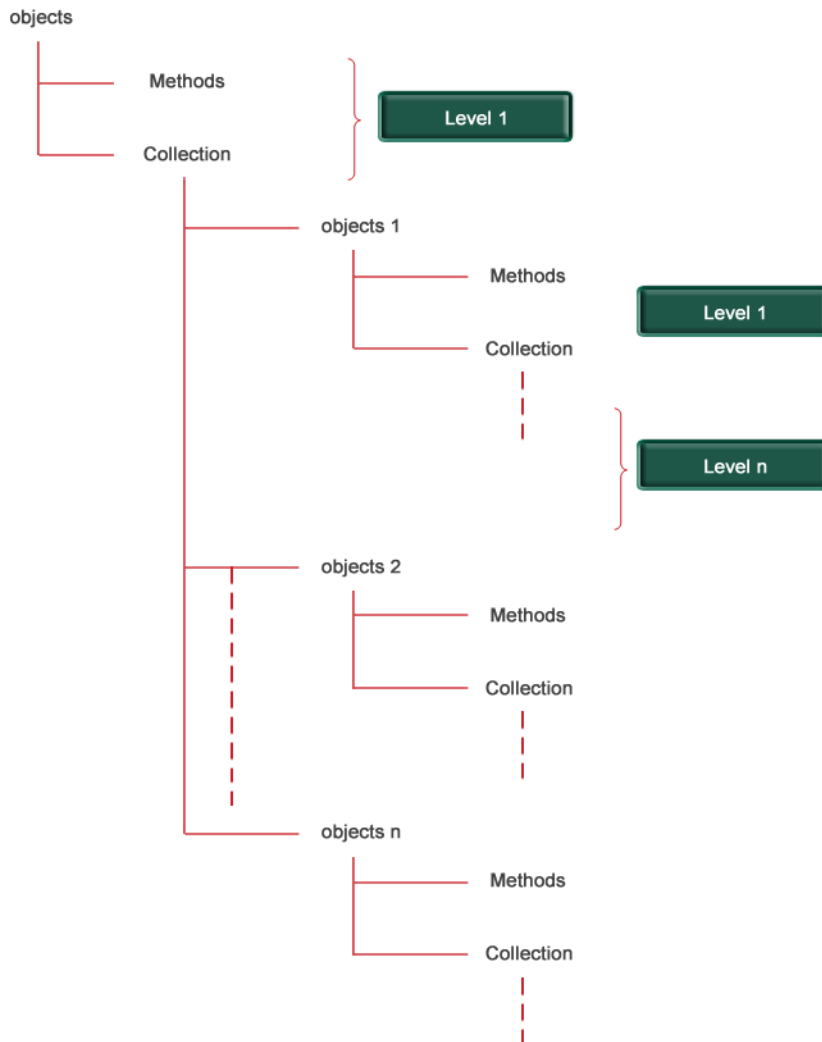


Figure 1 Object Structure

The detailed structure of the masters Company, Ledger, Group, Stock Item and transaction object Voucher is described in this section.

Company

Company objects contains various methods and collections. Some of the collections further contains sub-collections. Figure 2 shows the complete structure of Company objects. The availability of methods and collection depends on the features that are activated while creating the company or through F11 Features and F12 configuration settings.

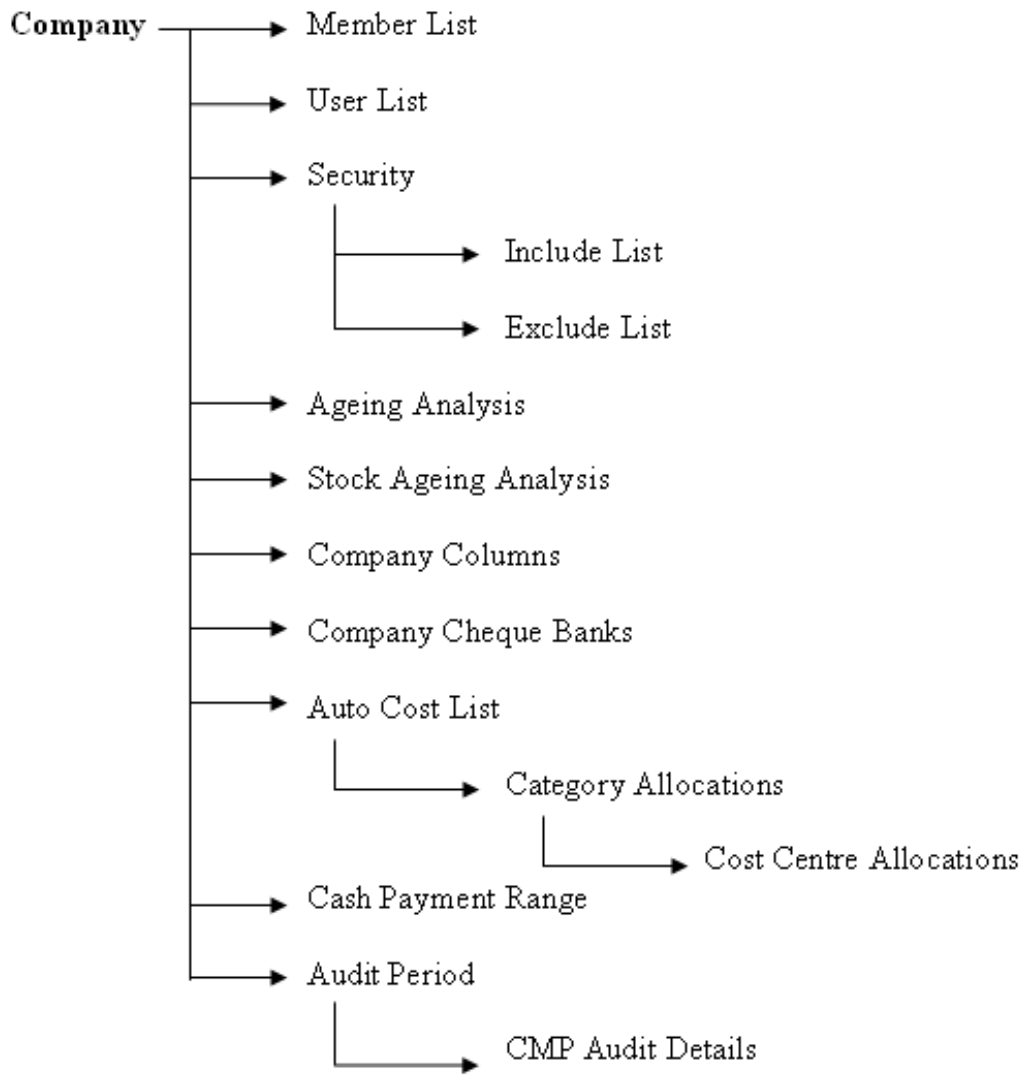


Figure 2 Structure of Company Object

The company object contains methods and collections at first level. Methods \$Name, \$Books- From, \$ExciseRange etc are available at Level 1. Some of the collections further contain sub-collection which in-turn contains a sub collection. For Example, the collection AutoCostList contains a sub - collection Category Allocations at Level 2. Category allocations again contain sub-collection Cost Centre allocations at Level 3.

Some methods and collections of Company Object:

Name	Type	Description
Name	Method	To fetch the name of Company
Address	Collection	Address of the Company
State Name	Method	To fetch the state name
Pincode	Method	To fetch the Pincode
Email	Method	To fetch the Email id
VATTINNumber	Method	To fetch the VAT No details

Address Collection

Name	Type	Description
Address	Method	To fetch the name of Company

Group

Group object contain methods \$Name, \$Parent, \$IsBillWiseOn, \$IsDeemedPositive, \$OverdueBills etc. and one sub-collection Language Name.



Figure 3 Group Object

Some methods and collections of Group Object:

Name	Type	Description
Language Name	Collection	Group Name in various languages
Parent	Method	Parent of current group name
OpeningBalance	Method	Opening Balance
ClosingBalance	Method	Closing Balance
DebitTotals	Method	DebitTotals
CreditTotals	Method	CreditTotals

Language Name Collection

Name	Type	Description
Name	Method	Ledger Name in selected language
Language	Method	Language Name
LanguageId	Method	Language ID

Ledger

Ledger objects contains methods \$Name, \$Parent, \$LedgerPhone etc. and collections Address, Bill allocations etc at Level 1. The features activated through F11 features and F12 configuration settings effectively decides the availability of methods and collection for Ledger Object.

The complete hierarchy of Ledger object is as shown in the following figure.

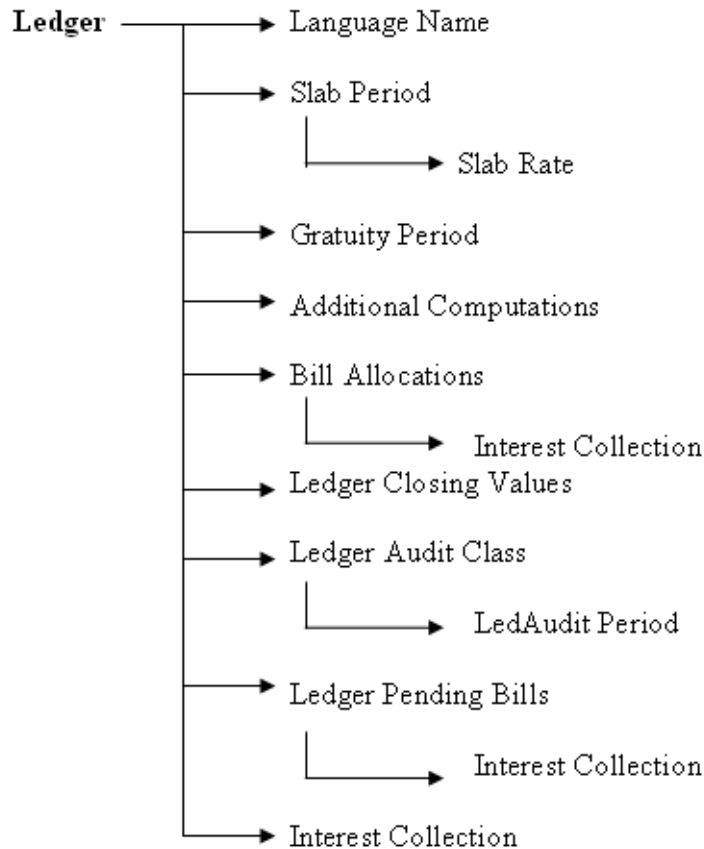


Figure 4 Ledger Object

The collection Bill Allocations won't be available if the option "Maintain Bill-Wise Details" is set to NO in F11 Accounting features.

Some methods and collections of Ledger object:

Name	Type	Description
Name	Method	Ledger name
Parent	Method	Parent group of ledger
Address	Collection	Address of the party
Mailing Name	Method	Ledger Mailing Name
Ledger Phone	Method	Phone number

Ledger Contact	Method	Contact person name
IsBillwiseOn	Method	Checks whether Billwise Details are required for the specified Ledger.
Bill Allocations	Collection	Opening Bill Details

Bill Allocations Collection

Name	Type	Description
BillDate	Method	Bill date
Name	Method	Bill name
OpeningBalance	Method	Opening balance of the bill

Stock Group

The Group Object contains many methods namely \$Parent, \$BaseUnits etc. and one sub - collection Language Name.



Figure 5 Stock Group Object

Some methods and collections of Stock Group Object:

Name	Type	Description
Name	Method	Name of Stock Group
Parent	Method	Name of Parent
Opening Balance	Method	Opening balance
Closing Balance	Method	Closing balance

Stock Item

The methods \$Name, \$BaseUnits, \$Description etc. and collections Language Name, Batch Allocations and Component List etc. belongs to the object stock Item. The features activated through F11 features and F12 configuration settings effectively decides the availability of methods and collection for Stock Item Object.

The complete hierarchy of Stock Item object is as shown in the following figure 6:

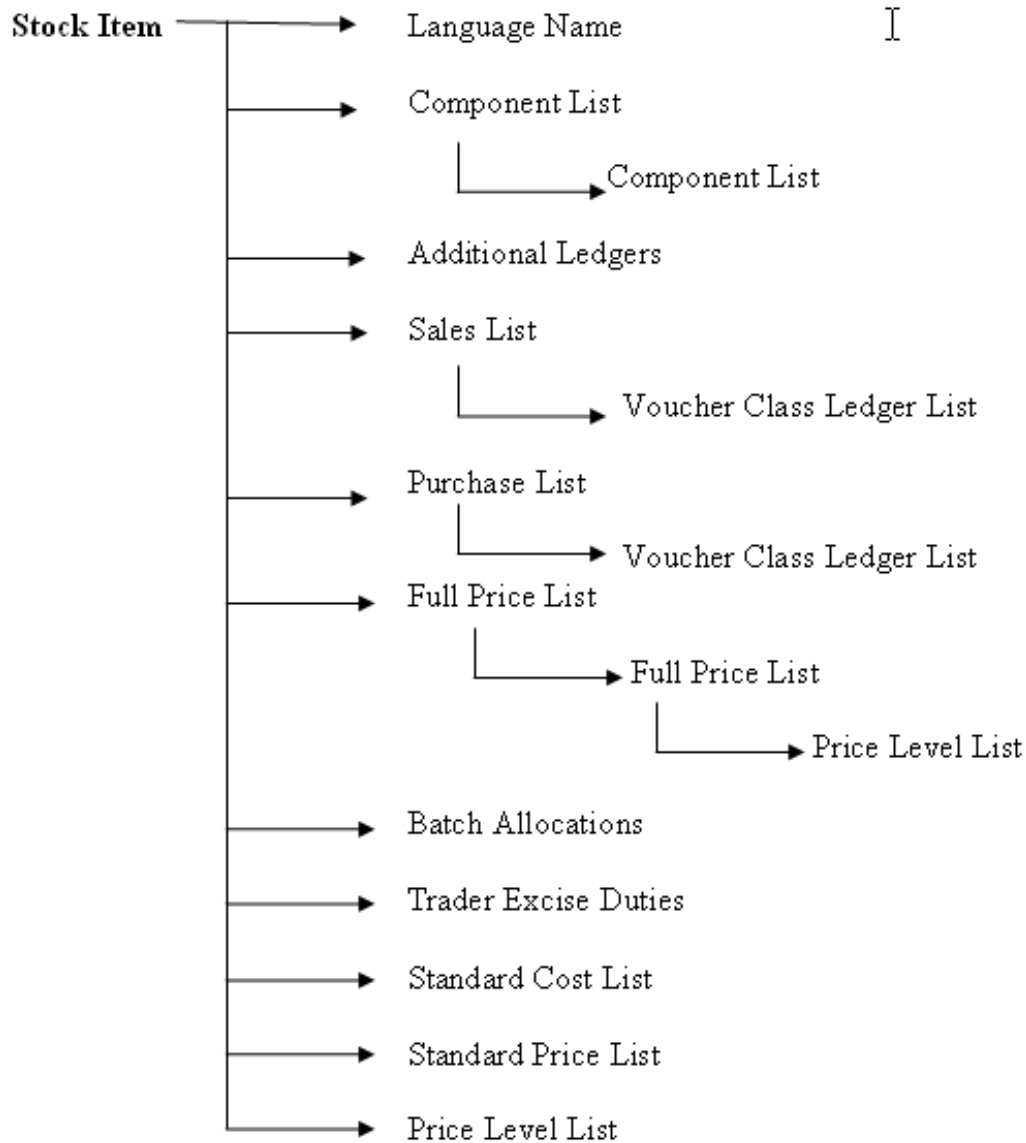


Figure 6 Stock Item Object

The collections Component List, Sales List, Purchase List and Full Price List further contain a sub-collection.

Some methods and collections of Stock Item object:

Name	Type	Description
Name	Method	Name of the Stock Item
Parent	Method	Parent name of the Stock Item

Category	Alloca-	Collection	Stock Item Category name
BaseUnits		Method	Stock Item Primary units
Description		Method	Description of Stock item
OpeningBalance		Method	Opening Balance in Quantity
ClosingBalance		Method	Closing Balance in Quantity
BatchAllocations		Collection	Opening Batch Details

BatchAllocations Collection

Name	Type	Description
BatchName	Method	To fetch the name of batch
GodownName	Method	Godown name
OpeningBalance	Method	Opening balance
ExpiryPeriod	Method	Expiry period



For the details of Category Allocations collection please refer Voucher object.

Voucher

Voucher object is the most complex object in TDL. There are so many methods and collections at Level 1 and most of the collections further have methods and sub-collection. The availability of methods and collection is based on the features activated through F11 features and F12 configuration settings.

The following figure shows the complete hierarchy of the Voucher object:

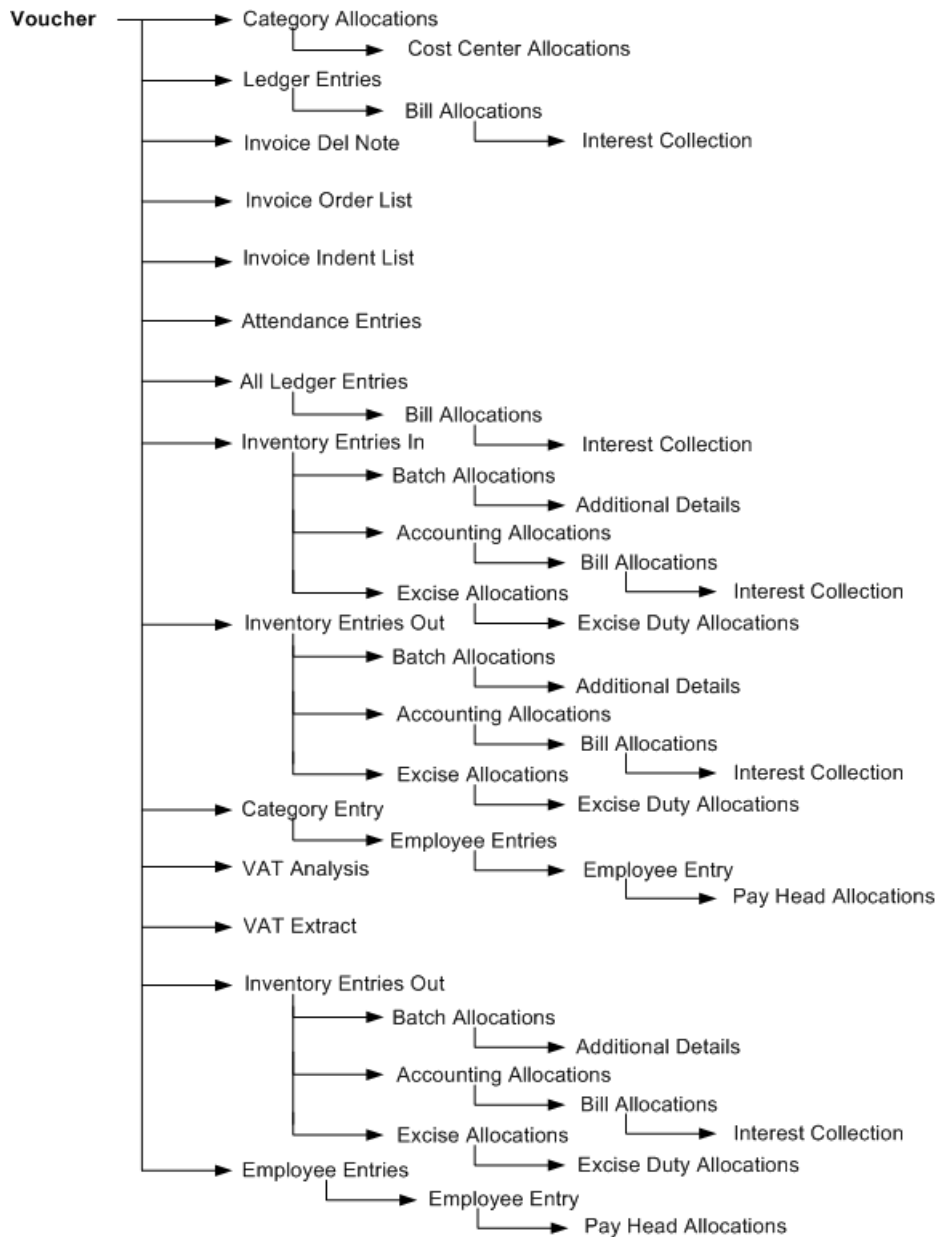


Figure 7 Voucher Object

The collection Ledger Entries, Inventory Entries, All Ledger Entries and All Inventory Entries Collections are widely used in the reports and for invoice customisation.

Some methods and collections of Voucher object:

Name	Type	Description
Date	Method	Voucher Date
VoucherNumber	Method	Voucher number
VoucherTypeName	Method	Name of the Voucher Type
PartyLedgerName	Method	Party Name in voucher
Narration	Method	Narration of the voucher
LedgerEntries	Collection	Ledgers involved in the transaction
InventoryEntries	Collection	Inventory details

LedgerEntries Collection

Name	Type	Description
LedgerName	Method	Ledger
Amount	Method	Amount
BillAllocations	Collection	Bill Details
CategoryAlloca-	Collection	Category Details



For the details of Bill Allocations details please refer Ledger object

InventoryEntries Collection

Name	Type	Description
StockItemName	Method	Name of the Stock Item sold to the party
BilledQty	Method	Quantity of the item sold to the party
Rate	Method	Rate of the Stock Item
Amount	Method	Amount
Batch Allocations	Collection	Batch details
UserDescription	Method	Description entered



For the details of Batch Allocations collection please refer Stock Item object

CategoryAllocations Collection

Name	Type	Description
Category	Method	Category Name
CostCentreAllocations	Collection	Cost Centre Details

CostCenterAllocations Collection

Name	Type	Description
Name	Method	Name of the Cost centre
Amount	Method	Amount