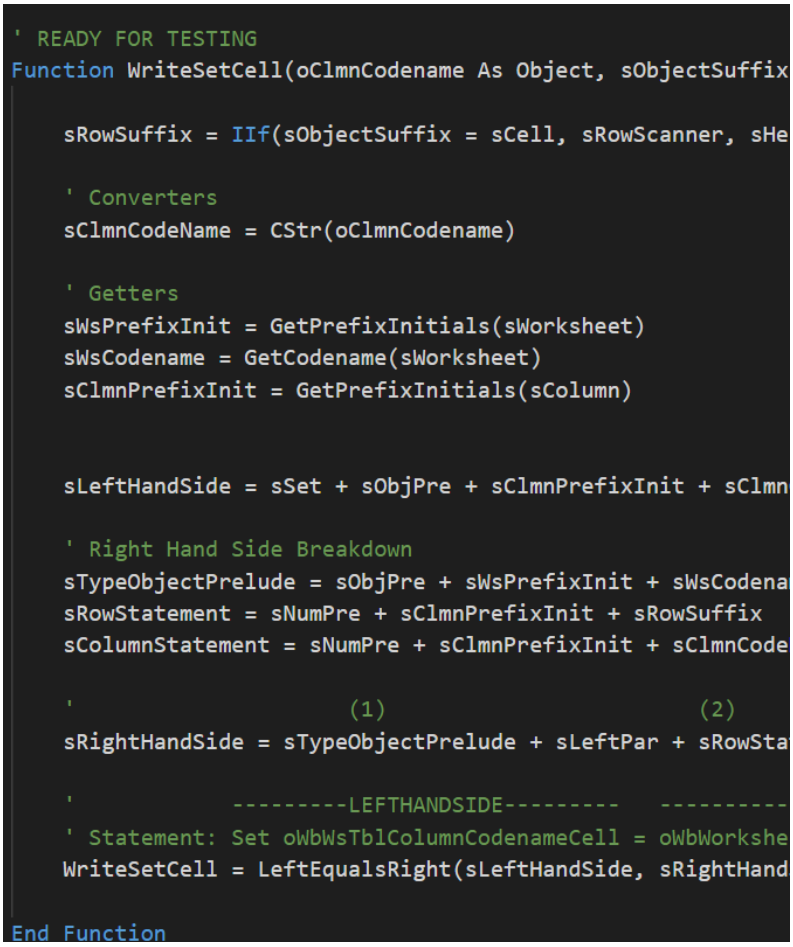


```
' READY FOR TESTING
Function WriteSetCell(oClmnCodename As Object, sObjectSuffix

    sRowSuffix = IIf(sObjectSuffix = sCell, sRowScanner, sHe

    ' Converters
    sClmnCodeName = CStr(oClmnCodename)

    ' Getters
    sWsPrefixInit = GetPrefixInitials(sWorksheet)
    sWsCodename = GetCodename(sWorksheet)
    sClmnPrefixInit = GetPrefixInitials(sColumn)


    sLeftHandSide = sSet + sObjPre + sClmnPrefixInit + sClmn

    ' Right Hand Side Breakdown
    sTypeObjectPrelude = sObjPre + sWsPrefixInit + sWsCodena
    sRowStatement = sNumPre + sClmnPrefixInit + sRowSuffix
    sColumnStatement = sNumPre + sClmnPrefixInit + sClmnCode

    '                    (1)                          (2)
    sRightHandSide = sTypeObjectPrelude + sLeftPar + sRowSta

    '            ---------LEFTHANDSIDE---------    ----------
    ' Statement: Set oWbWsTblColumnCodenameCell = oWbWorkshe
    WriteSetCell = LeftEqualsRight(sLeftHandSide, sRightHand

End Function
```

VBA BASELINE CODE GENERATION SYSTEM

A standardized way of generating VBA code that is easy to read, using VBA and Excel

ABSTRACT

VBA projects can be daunting with tables and databases, but they do not have to be. By building a strong baseline foundation through a systematic way of labeling everything, you can spend more time writing more meaningful work to get the job done faster.

Forrest Baird

VBA & Excel Developer

Table of Contents

Introduction

During my time as a Data Engineer in a previous role, my role was centered around harnessing the power of Microsoft products to automate as much workload in the digital spaces as possible. I had used Microsoft Visual Basic for Applications (VBA) in college for engineering coursework but only to solve complex mathematic problems.

As I was writing code in VBA, I was starting to notice there was a pattern to my writing. Rather than start every project from scratch, I decided to embark on a project to write the baseline metacode for each of my projects, starting with declarations and setters, which usually take the bulk of the time to write from scratch. Most of my VBA projects required creating tables for databases in Excel to push data, generate workloads, create reports for decision-making.

I have decided to share this with other fellow VBA Developers in hopes that they can use this as a tool for their own projects and even give suggestions to improve upon it. The generation system is one small piece of the puzzle that I think can be built upon to help developers who may be using VBA for Microsoft-based projects, whether they are legacy or modern.

Return to Table of Contents.

Variable Generation Naming Scheme

The user has a user interface to input the name of the workbooks, worksheets, tables, and columns associated with each table. Additionally, the user can put in the name of constants and variables with their associated values. When a string constant is produced, it is automatically enveloped in quotation marks.

The generator starts by writing in the declarations output page from a hierarchical top to bottom, starting with workbooks and ending with constants. Every declaration is written as follows:

dtPrefixCodenameSuffix

where,

- dt = Data type first letter,
- Prefix = The initials inherited from associated objects such as workbooks, worksheets, and tables,
- Codename = User generated codename,

- Suffix = Type of Object such as Header, Cell, or Column (Worksheet and Workbook omitted)

Each of these components are written in camelCase and are explained further below.

Return to Table of Contents.

Data Types

There are five distinct data types that can be generated through the Codebit Generator System:

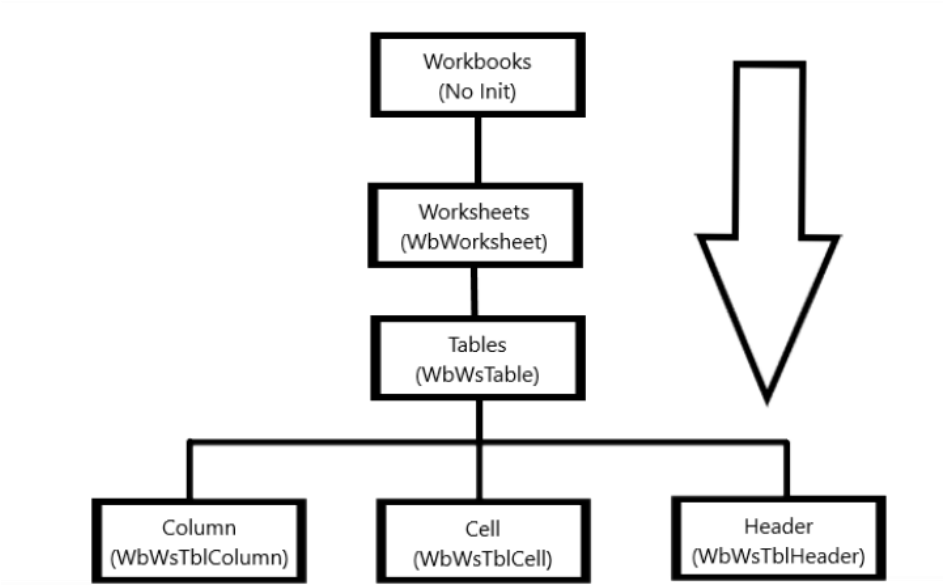
- b: For Boolean data types (true/false)
- i: For numeric data types (byte, integer, long, single, double)
- s: For string data types
- o: For object data types (Workbooks, Worksheets, Tables, Cells, Headers)
- v: For variant data types, usually when a function accepts inputs that can be objects, numbers, or strings and ultimately converted to a string.

Additionally, if there is an array declaration, the data type will be present first followed by *Arr*. There are plans to create specialized data type first letters later, such as for dates.

Return to Table of Contents.

Prefixes

Prefixes are initials that are generated to indirectly associate a form of inheritance. Prefixes are in the form of initials (usually the capital letters). Below is a diagram to help clarify how initials are created:



As seen from the diagram above, the further down the tree you go, more prefix initials are inherited to help keep track of variable names and their associations. Remember that every prefix initial conglomerate is preluded by a data type letter.

Initials are generated by taking the first letter of each letter in a parent's main name. The first letter of the initial is kept capitalized with subsequent letters converted to lowercase. If a parent's Main Name is singular, the first two letters are used. The user can ultimately change the initials in the object's generation table.

Example: Create the Name Column in the Recurring Purchases Table which belongs to the All Purchases Worksheet in the Financial Management System Workbook.

The code automatically generates it as such:

Public Const iFmsApRpNameColumn As Byte = …

Where the righthand side is declared in a fashion discussed in column declarations below.

Return to Table of Contents.

Workbook, Worksheet, and Table Declarations

Every workbook, worksheet, and table declaration are generated as an object using prefix initials described above. If the object is a constant, a string name declaration is also generated with a unique name from the inputs.

Examples:

' WORKBOOKS

Public oCashflowGenerationSystem As Workbook: Public Const sCashflowGenerationSystem As String = "Filename.xlsm"

' WORKSHEETS

Public oCgsSummarySheet As Worksheet: Public Const sCgsSummarySheet As String = "Summary Sheet"

Public oCgsCategoriesSheet As Worksheet: Public Const sCgsCategoriesSheet As String = "Categories"

' TABLES

' Worksheet 01: Summary

Public oCgsSsOverviewTable As Object: Public Const sCgsSsOverviewTable As String = "sCgsSsOverviewTable"

' Worksheet 02: Categories

Public oCgsCsOverviewTable As Object: Public Const sCgsCsOverviewTable As String = "sCgsCsOverviewTable"

Public oCgsCsClothesTable As Object: Public Const sCgsCsClothesTable As String = "sCgsCsClothesTable"

Public oCgsCsSelectedTable As Object

As seen above, the Selected Table does not have a string clause afterwards because it is a variable. In the event that an object is a variable, a selector table is usually present to associate workbook, worksheet, and table properties appropriately.

Setter subs for constant objects with the associated declarations are created on the Setters page.

Return to Table of Contents.

Column Declarations

For maximum flexibility to change tables while working with databases, all column numbers are generated in a relative manner. Instead of associating each column with a fixed number, every column is written as a function of a column that precedes it, unless it is the lead column (leftmost) on the worksheet.

Additionally, if the column is a lead column of another table that comes to the right of the lead table, it is written as a function of the last column generated on the preceding table plus a Table Skip Factor. A visual example below with pseudocode written afterwards can help demonstrate this position:

Leading Table

Lead Column	Another Column	Final Column

(Space)

Another Table

Lead Column	Another Column	Final Column

Pseudocode (where Leading Table is *Lt*, and Another Table is *At*):

' Lead Table

Public Const iWbWsLtLeadColumn As Byte = 1

Public Const iWbWsLtAnotherColumn As Byte = iWbWsLtLeadColumn + 1

Public Const iWbWsLtFinalColumn As Byte = iWbWsLtAnotherColumn + 1

' Another Table

Public Const iWbWsAtLeadColumn As Byte = iWbWsLtFinalColumn + iTableSkipFactor

Public Const iWbWsAtAnotherColumn As Byte = iWbWsAtLeadColumn + 1

Public Const iWbWsAtFinalColumn As Byte = iWbWsAtAnotherColumn + 1

As shown above, the lead column is always given the value of 1, where the user can change it at will. The constant *iTableSkipFactor* is written in the declarations and is always assumed at two to allow one column of space between tables. The user can change it numerically or change the value of the constant. For nonleading columns, their values are always of the value of the Previous Column + 1.

The advantage for writing columns this way are as follows:

1. If a new column needs to be inserted on the fly, only the code for the adjacent columns would need to be fixed. All other columns positions will be recalculated automatically.
2. The naming scheme assures that when cells are set that they are set within the right table, worksheet, and workbook (for multiple workbook projects).
3. If the lead column is adjusted, everything else follows.

Return to Table of Contents.

RowScanners, Header Rows, and Initial Rows

RowScanners are used in these type of set ups to allow subs and functions to scan an entire table from beginning to end. The Header Row is user-determined as to where the table begins, and the Initial Row is always the row below it (HeaderRow + 1).

The pseudocode for the RowScanner set declarations are as follows:

Public Const iWbWsTblHeaderRow As Byte = …

Public Const iWbWsTblInitialRow As Byte = iWbWsTblHeaderRow + 1

Public iWbWsTblRowScanner As Integer

Public Const sWbWsTblRowScanner As String = "sWbWsTblRowScanner"

Where the HeaderRow's value is determined according to user input in the Tables table.

RowScanners are variables that allow a user to utilize an automatically generated or user-defined sub to go through a table to perform a task. The RowScanner is usually initialized to the InitialRow as follows:

iWbWsTblRowScanner = iWbWsTblInitialRow

Return to Table of Contents.

String RowScanner vs. Numeric RowScanner Variables

The code generates two different data types of RowScanners: numeric and strings. The strings are generated with a unique identifier that points to the associated numeric RowScanner. This method was developed intentionally to prevent another RowScanner with the same numeric value from being called by accident, often crashing subs due to numerical overflow, crashing the entire worksheet due to an infinite loop, and/or manipulating data in the wrong table.

Return to Table of Contents.

Future Work

While the Code Generation System is functional, I have plans soon to improve the system and documentation behind it.

1. Validation code to make sure all inputs are correct,
2. An additional column in tables to count columns and make sure each table has more than 1 column before generating code,
3. The ability to create unique iTableSkipFactors for each nonleading Table,
4. Create Getters as appropriate,
5. Create a bank of constants and variables to automatically populate the Input Interface Tables,
6. Create Functions that create appropriate headers as a function of object type (workbooks, worksheets, cells, etc.)
7. Update documentation to walk through each sub and module (comments are currently available) and create a visualization of the whole project.

8. Work on automatically generating arrays (in progress) to allow scans across columns.
9. Validation for constants that their associated values will work with their inputs (example: reject an input for 468 if it is a byte, which has a range from 0-255).

Return to Table of Contents.