UiPath VBA Guide

Contents

[Introduction 2](#_Toc72887335)

[Check UiPath Activities First 2](#_Toc72887336)

[How to Create VBA 2](#_Toc72887337)

[Set Up the Office Environment 2](#_Toc72887338)

[Start with Macro Recording 2](#_Toc72887339)

[Edit in the Visual Basic Editor 3](#_Toc72887340)

[VBA Best Practices 3](#_Toc72887341)

[Error Handling 3](#_Toc72887342)

[Making VBA Code Dynamic 4](#_Toc72887343)

[Making VBA Easy to Read 4](#_Toc72887344)

[Making VBA Reusable 5](#_Toc72887345)

[Invoking VBA in UiPath 5](#_Toc72887346)

[Excel 6](#_Toc72887347)

[Sourcing a Macro From a File 6](#_Toc72887348)

[Running an Existing Macro 6](#_Toc72887349)

[Word 6](#_Toc72887350)

[Sourcing a Macro From a File 6](#_Toc72887351)

[Running an Existing Macro 6](#_Toc72887352)

[Interaction Between Office Files 7](#_Toc72887353)

[Excel –> Excel, Word –> Word 7](#_Toc72887354)

[Excel –> Word, Word –> Excel 7](#_Toc72887355)

# Introduction

When developing a bot that uses Office, most of the time it’s sufficient to simply use UiPath activities instead of using VBA. And using UiPath activities instead of VBA is generally better practice, since this makes it easier for people who don’t know VBA to debug the bot if it fails. But there are certain situations where VBA should be used instead of UiPath activities:

1. If one or more parts of the process that use Office CANNOT be developed using UiPath activities
2. If every part of the process that uses Office CAN be developed using UiPath activities, but doing so would require tons of activities, resulting in huge workflows that run slower

At the time of writing this document, at least one automation is being developed that falls in category 1 (Service Delivery > Sales Management Reports), and at least two automations are being developed that fall in category 2 (Service Desk > Amalgamated, Hill International). For any automations that are good candidates for VBA, this guide serves as a quick reference and introduction to learning VBA and incorporating it in UiPath.

# Check UiPath Activities First

Before starting to write out the VBA, it helps to check UiPath’s packages to see if an activity exists that accomplishes what you need to do in Office. If such an activity exists, it saves time to use it instead of creating VBA. These UiPath packages are good places to look for activities (recommendations in **bold**):

* Excel: **UiPath.Excel.Activities**, BalaReva.EasyExcel.Activities, BalaReva.Excel.Activities
* Word: UiPath.Word.Activities, **BalaReva.Word.Activities**
* PowerPoint: **UiPath.Presentations.Activities**, BalaReva.EasyPowerPoint.Activities, BalaReva.PowerPointActivities

If using VBA, there are several activities that will run your VBA code for you. For more about this, see the [Invoking VBA in UiPath](#_Invoking_VBA_in) section.

# How to Create VBA

## Set Up the Office Environment

You’ll need access to the Developer tab, so if it does not appear in the Office application you’re using, see [here](https://support.microsoft.com/en-us/topic/show-the-developer-tab-e1192344-5e56-4d45-931b-e5fd9bea2d45) to show the tab in the ribbon.

For each Office application used in your bot that requires VBA, remember to go to Developer > Macro Security and confirm that “Trust access to the VBA project object model” is **checked** (both on your local machine, and any test/prod VMs that the bot will run on).

## Start with Macro Recording

The easiest and quickest way to start writing VBA code is by recording a macro. Simply record the steps that the bot will go through using the macro recorder, and VBA code will be automatically generated.

Note that only Excel and Word have macro recorders available.

The code generated from the macro recorder is a great starting point, but this code will need to be edited to follow best practices, and to allow for integration with UiPath.

## Edit in the Visual Basic Editor

To edit a macro, you can [open the macro](https://support.microsoft.com/en-us/office/edit-a-macro-ed9e8c3d-58fd-47a1-83eb-bdee680376bb) in the Visual Basic editor. This editor features what you would expect from a code editor: Intellisense, breakpoints, an object browser, a locals window, etc. Here you can make code changes and run your code to see what it does to your file.

One annoying quirk this editor has is that every time a compile error occurs, a popup warning you about the error stops you in your tracks. This popup gets annoying quickly, but can thankfully be [turned off](https://stackoverflow.com/questions/11560934/when-editing-microsoft-office-vba-how-can-i-disable-the-popup-compile-error-m) so that compile errors are simply highlighted red.

## VBA Best Practices

There are several best practices to keep in mind when adding VBA to your UiPath automation:

### Error Handling

Error handling is incredibly important when using VBA in UiPath because if VBA code fails, a popup will appear displaying the error and asking the user to debug the code or end the execution. If a bot encounters this popup, the bot will be stuck in place and eventually fail, and we won’t know what the actual error was because of the popup.

There is a way to address this however:

1. In VBA, add code to handle the error
2. In VBA, return the error details as a string
3. In UiPath, manually convert the returned error details to a string
4. In UiPath depending on the value of the error string,
   1. If the error string is "", continue on with the bot
   2. Else throw a custom system exception and include the error string

**Function Main() As String**

**On Error GoTo Handle**

**Main = ""**

**' This is where your VBA code will go**

**Exit Function**

**Handle:**

**Main = "Main.vb threw an error. Num: " & Err.Number & ", Source: " \_**

**& Err.Source & ", Description: " & Err.Description**

**End Function**

UiPath accepts macro output as an Object, so output will need to be first converted to a String. Then you can throw a custom system exception. For example, suppose the returned error is stored in vbaErrorObj:

* ASSIGN vbaErrorString = CStr(vbaErrorObj)
* IF vbaErrorString.Equals("") THEN continue with the bot
* ELSE Throw System.Exception(vbaErrorString)

### Making VBA Code Dynamic

When recording VBA code, the code that the macro recorder returns is often based on fixed references. So the code will need to be further modified to make the bot dynamic. For example, suppose that the goal of a bot is simply to bold all the cells in a certain Excel sheet:

|  |  |
| --- | --- |
| Name | Age |
| Andrew | 24 |

Using the above data to record the VBA code, the code returned from the macro recorder would look something like this:

**Range("A1:B2").Select**

**Selection.Font.Bold = True**

But in the future this Excel sheet could have more than just 2 rows. So the above code would need to be

**Range("A1").CurrentRegion.Select**

**Selection.Font.Bold = True**

to account for all the rows in the sheet. And this can be further trimmed down to be

**Range("A1").CurrentRegion.Font.Bold = True**

This saves a line of code and removes the need for the VBA script to Select the range. It’s good practice to only Select cells or ranges when needed.

If you want to execute multiple steps on the same object, use a variable:

**Dim r As Range**

**Set r = Range("A1").CurrentRegion**

**With r**

**.Font.Bold = True**

**.Font.Size = 12**

**.Style = "Currency"**

**End With**

### Making VBA Easy to Read

Documentation and readability of code is especially important for VBA because some may not know VBA and how to debug it if issues occur. Here are some best practices to make VBA code easier to read:

* Adding comments to code is critical to let readers know what the code does. To add a comment, simply add an apostrophe to a line, and everything after the apostrophe is commented out
* To improve readability, if a single VBA statement is very large, separate it into multiple lines. There is a quirk of VBA where if a statement extends to the next line, you need to use an underscore to indicate that the next line is the part of the same statement:

**Main = "Main.vb threw an error. Num: " & Err.Number & ", Source: " \_**

**& Err.Source & ", Description: " & Err.Description**

### Making VBA Reusable

Reusability is an important part of coding in any language. Not only can VBA code be reused several places in the same automation, but if the code is robust enough, it can be reused in other automations.

To make code reusable, you can either write a Sub or a Function. Sub procedures do not return anything, but functions do:

* Sub <Name>(<Parameters>)
* Function <Name>(<Parameters>) As <Type>
* Parameters are optional, and are <ArgName1 as Type1, ArgName2 as Type2, etc>
  + For example, b As Boolean, r As Range, s As String

Helper subs and functions should be placed in a separate file, e.g. **Helpers.vb**. These functions can then be sourced into a **Main.vb** file and called within the “Main” method.

For example, suppose that the following helper function GreaterThanTen is saved in **Helpers.vb**. This function takes an integer x and returns x > 10 as a Boolean value:

**Function GreaterThanTen(x As Integer) As Boolean**

**GreaterThanTen = x > 10**

**End Function**

Then the Main method of **Main.vb** would source this function within the current Excel file and execute it using the following:

**Dim b As Boolean ' Stores the result of GreaterThanTen**

**' Source all functions from Helpers.vb**

**ActiveWorkbook.VBProject.VBComponents.Import "<FILEPATH>\Helpers.vb"**

**' Call the GreaterThanTen function, and pass in 15**

**b = Application.Run "GreaterThanTen", 15**

**MsgBox(b) ' Display the result of GreaterThanTen (True since 15 > 10)**

If instead of just one Main file, the automation requires multiple main files, then name them according to what they do, e.g. **FormatReport.vb** or **MakePivotTable.vb**. The **Helpers.vb** file can then be sourced by any main file that needs it. This is similar to how you would make UiPath sequences reusable.

# Invoking VBA in UiPath

There are two ways that UiPath can invoke VBA on an Office file:

1. **Source VBA macros from a text file into the Office file and run the macros**
2. **Run macros that already exist on a macro-enabled file (docm, xlsm, etc)**

Using method 1 is generally better practice, since it promotes code reusability and simplifies debugging, but there may be situations where it makes more sense to use method 2.

This section shows which UiPath activities to use for methods 1 and 2, for both Excel and Word.

## Excel

### Sourcing a Macro From a File

Use the Invoke VBA activity in UiPath.Excel.Activities. This activity is only valid within an Excel Application Scope. Specify the file path where the VBA code is stored, which method in the code file should be executed, and if necessary, any input parameters and return values.

### Running an Existing Macro

Use the Execute Macro activity in UiPath.Excel.Activities. This activity is only valid within an Excel Application Scope. Enter the macro name, and if necessary, any input parameters and return values.

## Word

### Sourcing a Macro From a File

Unfortunately, UiPath’s Invoke VBA activity only works for Excel files, so sourcing a macro into a Word file and running it is less straightforward. But it is still possible to invoke VBA on a Word file, and surprisingly enough, the best way to do this is using the Invoke VBA activity.

We can run the Invoke VBA activity using a blank Excel file. Then, in our VBA code, we can create a Word object. With this object we can open our desired Word document, source macros into it, and run them.

Suppose that we wanted to source a macro in a Word document to change the font size of the entire document. First, create this macro and store it in **Helpers.vb**:

**Sub ChangeFontSize(n As Integer)**

**Selection.WholeStory**

**Selection.Font.Size = n**

**End Sub**

Then the Main method of **Main.vb** would source this function within the Word document and execute it using the following code:

**' Create the Word object and open the desired Word document**

**Dim objWord As Object**

**Set objWord = CreateObject("Word.Application")**

**objWord.Documents.Open("<DOCPATH>")**

**' Source all functions from Helpers.vb, and change the doc's font size to 12**

**objWord.ActiveDocument.VBProject.VBComponents.Import** **"<FILEPATH>\Helpers.vb"**

**objWord.Run "ChangeFontSize", 12**

**' Save and close the document, and quit the instance of Word**

**objWord.ActiveDocument.Save**

**objWord.ActiveDocument.Close**

**objWord.Quit**

### Running an Existing Macro

Use the Execute Macro activity in BalaReva.Word.Activities. This activity is only valid within a Word Application Scope. Enter the macro name, and if necessary, any input parameters and return values.

## Interaction Between Office Files

There may be times when using VBA across multiple Office files is necessary. For example, you may need to move data from one Excel file to another, or paste a chart from Excel into Word.

### Excel –> Excel, Word –> Word

When the two files that interact have the same type, the VBA is a bit easier. Excel –> Excel is much more common than Word –> Word.

For Excel –> Excel, it's as simple as creating a Workbook variable and opening up a workbook in it:

**' <EXCELPATH> is the path of the other Excel file you want to use**

**Dim wb As Workbook**

**Set wb = Workbooks.Open("<EXCELPATH>")**

When using VBA variables to open another workbook, refer to the original workbook using ThisWorkbook, and use your variable (wb) to refer to the other Excel file. For example, suppose you wanted to copy a cell from the other workbook to your original workbook:

**' Copy the value of cell A1 from the other workbook to the original**

**ThisWorkbook.Sheets("Sheet1").Range("A1").Value = \_**

**wb.Sheets("Sheet1").Range("A1").Value**

Word –> Word interaction is much less common, but it's done in the exact same way as Excel –> Excel:

**Dim doc As Document**

**Set doc = Documents.Open("<FILEPATH>")**

**' Instead of ThisWorkbook and wb, you'll use ThisDocument and doc**

### Excel –> Word, Word –> Excel

When the two files that interact have different filetypes, we can simply create an Excel object in our Word script, or a Word object in our Excel script:

**' Create an Excel object in Word**

**Dim objExcel As Object**

**Set objExcel = CreateObject("Excel.Application")**

**objExcel.Workbooks.Open("<FILEPATH>")**

**' Or create a Word object in Excel**

**Dim objWord As Object**

**Set objWord = CreateObject("Word.Application")**

**objWord.Documents.Open("<DOCPATH>")**

For example, suppose you wanted to copy data from Excel and paste it into Word in a certain place:

**' Create the Word object in Excel**

**Dim objWord As Object**

**Set objWord = CreateObject("Word.Application")**

**objWord.Documents.Open("<DOCPATH>")**

**' Copy the Excel data to the clipboard**

**Range("A1").CurrentRegion.Copy**

**' Paste the data in the Word doc. One way to control where the data is pasted**

**' is to use bookmarks. Here the bookmark is "b1"**

**objWord.ActiveDocument.Bookmarks("b1").Range.PasteExcelTable \_**

**LinkedToExcel:=False, WordFormatting:=False, RTF:=False**

**' Clear the clipboard (a good practice after copy-pasting in VBA)**

**Application.CutCopyMode=False**

When using Excel or Word object variables, use **ActiveDocument** and **ActiveWorkbook** instead of **ThisDocument** and **ThisWorkbook**.

Moving contents from Word –> Excel is done in the same way as Excel –> Word, but it's incredibly unlikely that you would ever need to move from Word –> Excel.