Kompilieren und Debuggen

Autor & Copyright: Dipl.-Ing. Harald Nahrstedt

Version: 2016 / 2019 / 2021 / 365

Erstellungsdatum: 1.03.2024

Überarbeitung:

Beschreibung:

VBA ist eine interpretierende Programmiersprache. Zwar wird der VBA-Code vorkompiliert, um Strukturen für syntaktische Überprüfungen aufzubauen, ein Kompilieren bis hin zu einem Maschinencode ist aber nicht möglich. Wer VBA nutzt, sollte den durch Makros üblichen Ad-hoc-Programmierstil ablegen und seine VBA-Projekte genauso konzipieren wie andere Softwareprojekte.

Anwendungs-Datei:

1 Compiler vs. Interpreter

Prozeduren in einer höheren Programmiersprache müssen zur Ausführung in Maschinencode-Anweisungen übersetzt werden, damit sie vom jeweiligen Computer verstanden werden. Dazu gibt es zwei Möglichkeiten, die Verwendung eines Compilers oder eines Interpreters (Bild 1).

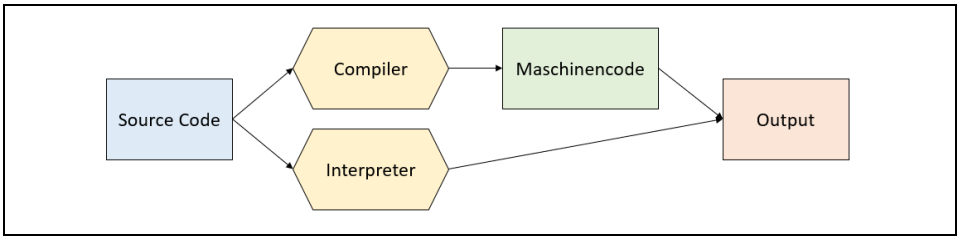

Bild 1. Compiler vs. Interpreter

Der Compiler wandelt den Programmcode vor der Ausführung in Maschinencode um, während der Interpreter bei der Ausführung einzelne Anweisungen im Programmcode nacheinander in Maschinencode umwandelt. Der compilierte Maschinencode wird dadurch schneller ausgeführt. Während der Compiler alle auftretenden Fehler nach der Umsetzung anzeigt, werden beim Interpreter die Fehler je Anweisung ausgegeben. Compiler arbeiten nach dem Übersetzungs-Linking-Loading-Modell, während Interpreter die Interpretationsmethode nutzen.

VBA ist eine interpretierende Programmiersprache. Zwar wird der VBA-Code vorkompiliert, um Strukturen für syntaktische Überprüfungen aufzubauen, ein Kompilieren bis hin zu einem Maschinencode ist aber nicht möglich. Wer VBA nutzt, sollte den durch Makros üblichen Ad-hoc-Programmierstil ablegen und seine VBA-Projekte genauso konzipieren wie andere Softwareprojekte.

2 Optionen zur Kompilierung

Schon bei der Eingabe von VBA-Code überprüft der Visual Basic-Editor den Code auf Syntaxfehler. Allerdings erst, wenn die Zeile gewechselt wird. Wie die Syntax überprüft werden soll, kann unter Register *Extras / Optionen* des VBA-Editors unter Register *Allgemein* ausgewählt werden (Bild 2).

Die Option *Bei Bedarf* legt fest, ob ein Projekt vor dem Start vollständig kompiliert wird, oder nur bei Bedarf. Mit der kompilierten Version kann die Anwendung schneller gestartet werden.

Die Option *Im Hintergrund* legt fest, dass Leerlaufzeiten zur Kompilierung des Projekts im Hintergrund verwendet werden soll. Dadurch kann die Ausführungsgeschwindigkeit während der Laufzeit verbessert werden. Die Option ist nur verfügbar, wenn die Option *Bei Bedarf* ebenfalls aktiviert wurde.

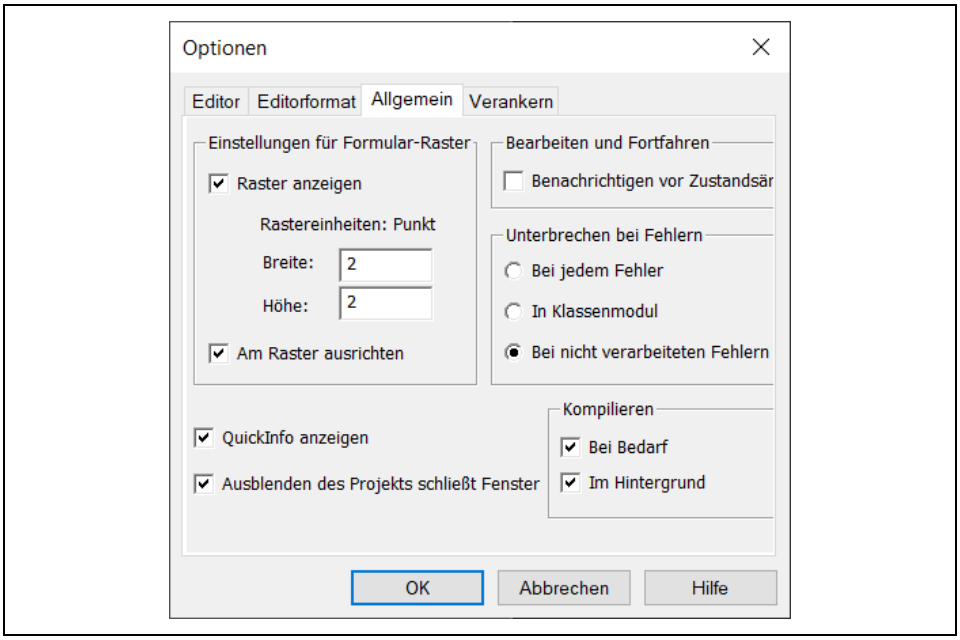

Bild 2. Optionen zum VBA-Editor

3 Kompilieren

Neben den Syntaxfehlern gibt es auch Fehler, die erst zur Laufzeit des Codes auftreten. Um den Code vollständig zu prüfen, kann er kompiliert werden. Dazu gibt es ein Register *Debuggen* im VBA-Editor (Bild 3).

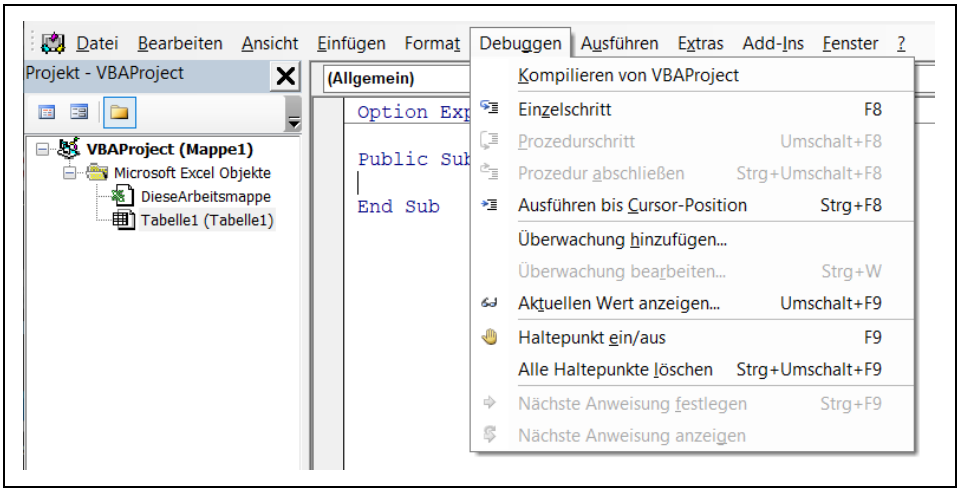

Bild 3. Register Debuggen im VBA-Editor

Unter Register *Debuggen* wird mit *Kompilieren von VBAProject* der Kompiliervorgang gestartet. Als Beispiel wird in einer Prozedur die Variable *sText* deklariert, in der Ausführung durch einen Schreibfehler aber die Variable *sTest* benutzt. Der Kompiliervorgang liefert dann einen Fehlerhinweis (Bild 4).

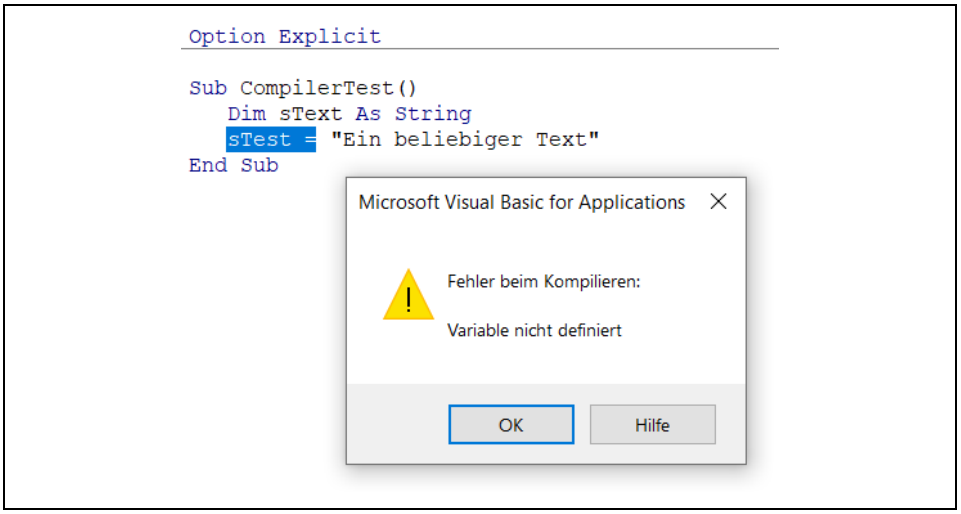

Bild 4. Fehlermeldung beim Kompiliervorgang

Ein Kompiliervorgang kann auch schrittweise erfolgen. Dazu wird der Cursor in die zu testende Prozedur gesetzt und unter Register *Debuggen* die Methode *Einzelschritt* aufgerufen. Es wird dann die erste Anweisung abgearbeitet (Bild 5).

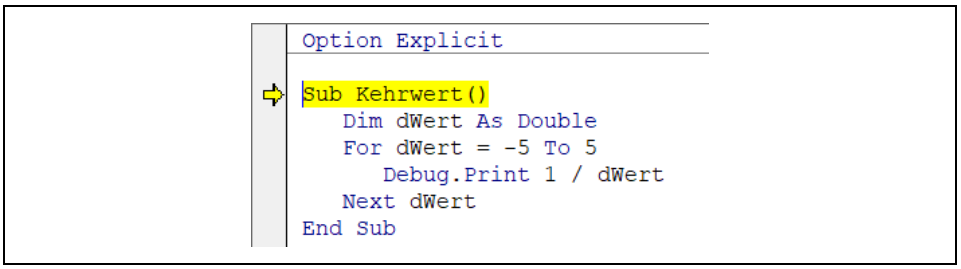

Bild 5. Schrittweise Kompilierung

Durch erneuten Aufruf von Einzelschritt oder mit der Taste F8 wird die nächste Anweisung aufgerufen. Dieses Vorgehen wird als *Debuggen* bezeichnet.

4 Debuggen

Statt des Aufrufs von *Einzelschritt* kann das Debugging auch mit der F8-Taste beginnen. Mit der F5-Taste wird der Kompiliervorgang ohne Unterbrechung, falls kein Fehler auftritt, bis zum Ende durchgeführt. Wird das Debugging direkt mit der F5-Taste gestartet, dann können vorher im Quellcode mit der F9-Taste oder durch Klicken des

Mauszeigers auf den linken grauen Rand Haltesymbole gesetzt werden. Der Kompiliervorgang stoppt an diesen Symbolen und kann mit den Tasten F8 oder F5 fortgesetzt werden. Beim Stopp an einem Haltpunkt können Inhalte von Varibalen über den Mauszeiger abgefragt werden (Bild 6).

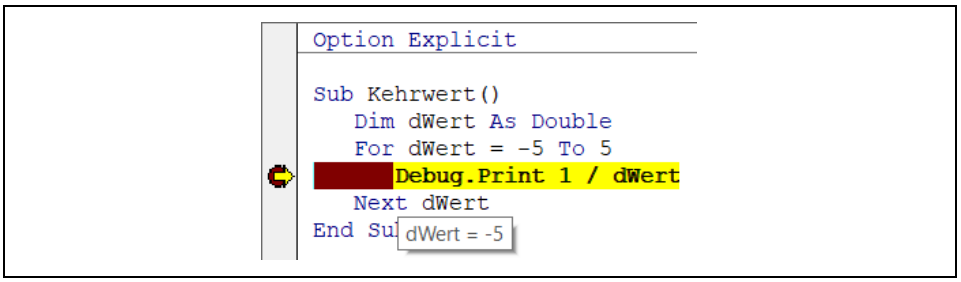

Bild 6. Inhalt einer Variablen beim Debugging

Statt einem Haltepunkt, kann der Cursor auch an eine Anweisung in der Prozedur gesetzt werden, an der die Ausführung anhalten soll. Mit der Methode *Ausführen bis Cursor-Position* stoppt die Ausführung dann an der bestimmten Stelle.

5 Laufzeitfehler

Trotzdem können immer noch Fehler zur Laufzeit auftreten. Im folgenden Beispiel soll der Kehrwert eines eingegebenen Wertes bestimmt werden. Wird dabei statt einem Wert z. B. ein Text eingegeben, dann gibt es Fehlerhinweis auf unverträgliche Typen (Bild 7).

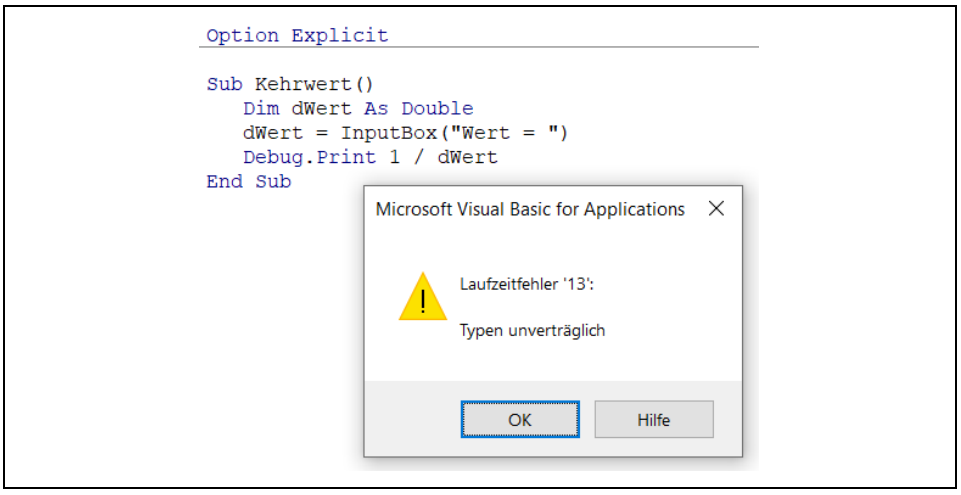

Bild 7. Laufzeitfehler 13

Auch die Eingabe von 0 führt zu einem Laufzeitfehler (Bild 8).

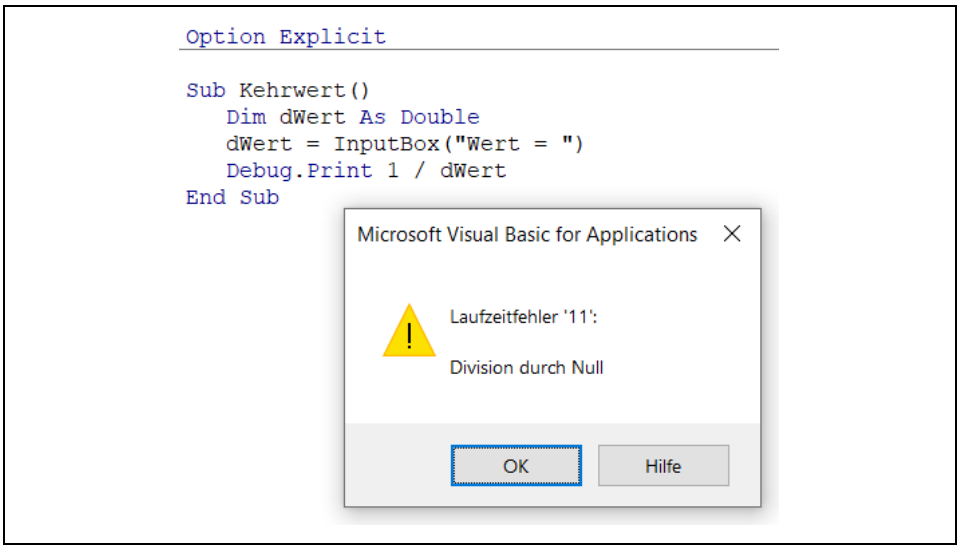

Bild 8. Laufzeitfehler 11

Die Ausführung wird dann unterbrochen und mit Bestätigung der Fehlermeldung durch OK öffnet sich der VBA-Editor und die Anweisung mit dem Fehler gelb markiert. Der VBA-Editor befindet sich noch im Laufzeitmodus, sodass die Inhalte von Variablen durch den Mauszeiger geprüft werden können.

6 Bedingte Kompilierung

Mit Anweisungen zur bedingten Kompilierung kann ein Codeblock aktiviert, bzw. deaktiviert werden. So lassen sich alternative Codezeilen in einer Prozedur auf ihr Laufzeitverhalten testen. Selbst fehlerhafte Anweisungen können so im Sourcecode verbleiben.

Bedingte Kompilierungsstrukturen werden mit der *If-Then-Else*-Anweisung erstellt, denen das #-Zeichen vorangestellt wird. Außerdem können konstante Werte deklariert werden, denen ebenfalls das #-Zeichen vorangestellt wird. Die folgende Prozedur ermittelt die Laufzeit einer *For-Next*- und einer *Do-Loop*-Schleife, die beide jeweils eine Million-Durchläufe haben. Welche Schleife zum Einsatz kommt, bestimmt die Konstante iSwitch.

Codeliste 1. Konstruktion einer bedingten Kompilierung

```
Sub RuntimeTest()
   #Const iSwitch = 1
   Dim sStart  As Single
   Dim lLoop   As Long

   sStart = Timer
   #If iSwitch = 1 Then
      For lLoop = 1 To 10000000
      Next lLoop
   #Else
      Do
```

```
         lLoop = lLoop + 1
      Loop Until lLoop = 10000000
   #End If
   Debug.Print Timer - sStart
End Sub
```

Mit iSwitch = 1 ergeben sich Laufzeiten zwischen 0,05 bis 0,09. Mit iSwitch = 2 liegen die Laufzeiten zwischen 0,17 bis 0,24 und sind damit langsamer.

VBA besitzt auch integrierte Konstante, die nicht deklariert werden müssen.

Tabelle 1. Integrierte Kompilerkonstanten

Konstante	Beschreibung
VBA6	Älteres Office
VBA7	Office ab Version 2010
Win32	Office unterstützt mindestens 32 Bit
Win64	Office unterstützt 64 Bit

Win32 gibt auch auf 64-Bit-Systemen den Wert *True* zurück, damit auch älterer Sourcecode, der früher zur Unterscheidung zwischen 16- und 21-Bit Versionen notwendig war, lauffähig ist. Empfehlenswert ist es immer, die höhere Version als Bedingung zu nutzen.

Im Gegensatz zur normalen *If-Then*-Anweisung in Prozeduren, kann die bedingte Kompileranweisung #If auch im Deklarationsbereich eines Moduls verwendet werden. Im folgenden Beispiel wird der Typ einer Variblen, abhängig von der Office-Version, deklariert.

Codeliste 2. Deklarationsteil zur Office-Version

```
Option Explicit

#If Win64 Then
   Public iloop As LongLong
#Else
   Public iloop As Long
#End If
```

Auch eine Kompilerkonstante, egal ob sie im Deklarationsteil oder in einer Prozedur steht, gilt ebenfalls im gesamten Modul. Sollen sie allerdings im gesamten Projekt gelten, dann müssen sie im VBA-Editor unter Register *Extras / Eigenschaften von VBAProject* im Feld *Argumente* eingetragen werden, getrennt durch Doppelpunkte (Bild 9).

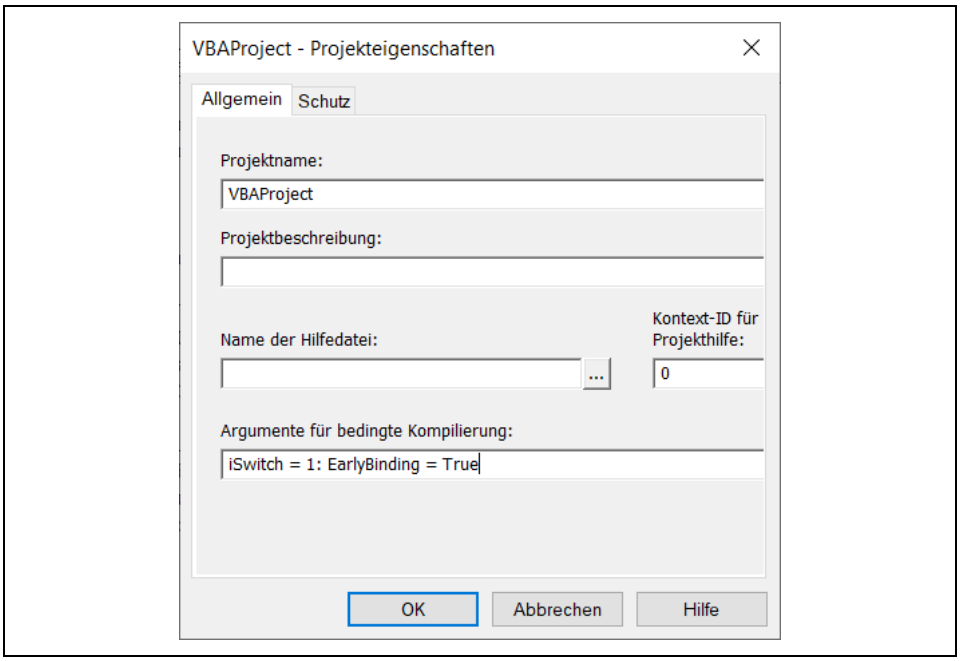

Bild 9. Argumente für bedingte Kompilierung im gesamten Projekt