Jörg Kuthe

qtXLS

Anleitung zur Verwendung der qtXLS Software

Stand: 17. April 2007

QT software GmbH
Konstanzer Str. 10
D-10707 Berlin
Germany

Telefon 030/9290087-0
Telefax 030/9290087-2

http://www.qtsoftware.de
eMail info@qtsoftware.de

Inhaltsverzeichnis

1. Einführung

Die qtXLS Library bietet dem Programmierer Routinen zum Lesen und Schreiben von Dateien im Microsoft Excel Format. Diese enden bekanntlich auf .xls. qtXLS basiert auf den von Microsoft bereitgestellten ODBC Treibern, die normalerweise bei der Installation von Excel auf einem PC unter Windows automatisch eingerichtet werden (vgl. Abb.).
Das Vorhandensein der Microsoft Excel ODBC Treiber ist eine der Grundvoraus- setzungen für das Funktionieren der qtXLS Routinen.
Sind diese Treiber auf einem PC nicht vorhanden, so können sie entweder durch Installation von Microsoft Excel oder die **Microsoft Data Access Components (MDAC)** bereitgestellt werden. Letztere dürften die kostengünstigere Alternative sein, da sie von Microsoft's WebSite kostenlos geladen werden können. Man findet sie am schnellsten mithilfe der Suchfunktion im "Download Center".

⇨ http://www.microsoft.com/downloads

Zum Zeitpunkt der Erstellung dieser Bedienungsanleitung waren die MDAC innerhalb der Download Kategorie "Drivers" bzw. "Treiber" zu finden. Sofern keine lizenzrechtlichen Einwände von Seiten des Herstellers bestehen, werden Sie den Verweis auf die Treiber auch auf der WebSeite von QT software vorfinden:

⇨ http://www.qtsoftware.de/vertrieb/db/qtxls.htm

Da ODBC die Grundlage von qtXLS ist, kommuniziert qtXLS mit dem Excel ODBC Treiber über die in Windows integrierten ODBC Funktionen und dort mithilfe der Structured Query Language (SQL). Dies ist für die Verwendung einiger der qtXLS Routinen relevant, da dort von der Mächtigkeit und den Möglichkeiten von SQL Gebrauch gemacht wird.
Da qtXLS auf SQL basiert und dies die Sprache ist, um auf relationale Datenbanken zuzugreifen, verwenden wir hier statt des Excel Begriffs "**Arbeitsblatt**" die Bezeichnung "**Tabelle**".

1.1 Funktionen und Beschränkungen

Mit qtXLS Routinen können

- Dateien im Excel Dateiformat angelegt werden,
- Tabellen (Arbeitsblätter) in diesen Dateien erzeugt werden,
- Daten in Tabellen geschrieben werden,
- Daten aus Tabellen gelesen werden und
- Informationen über Tabellen und Spalten ermittelt werden.

Da qtXLS auf den ODBC Treibern Microsofts basiert, ist qtXLS auch von deren Beschränkungen betroffen. Die wesentlichen Limitierungen sind:

- Zum Lesen von Excel Tabellen müssen sich die **Namen der Spalten in der ersten Zeile der Tabelle** befinden (vgl. Abb. 1). Beim Anlegen von Tabellen mittels qtXLS werden die Tabellen entsprechend eingerichtet, so daß diese Voraussetzung erfüllt ist (d.h. Tabellen, die mit qtXLS erzeugt wurden, sind auch damit lesbar). Das Schreiben erfolgt dann sukzessive in den Folgezeilen.

- Beim Schreiben in Excel Tabellen werden die neuen Daten stets hinzugefügt. Es ist nicht möglich gezielt in eine spezifizierte Zeile zu schreiben.

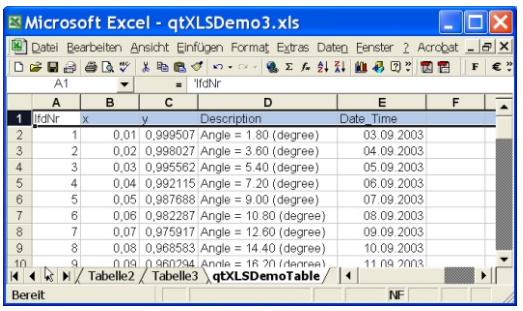

Abb. 1: Excel Tabelle qtXLSDemoTable (in qtXLSDemo3.xls, erzeugt mit qtXLSDemoWriteTable)

- Es kann nur zeilenweise geschrieben werden.
- Es können nur die Excel Datentypen NUMBER (Zahl), DATETIME (Datum und Zeit), TEXT (Text), CURRENCY (Währung) und LOGICAL (logische Werte) verwendet werden. Formeln oder sonstige Formate werden nicht unterstützt.
- Textformatierungen (Schrifttyp, Farbe etc.) sind nicht möglich.
- Namen von Spalten und Tabellen können aus fast allen in Excel gültigen Zeichen gebildet werden. Nicht verwendet werden sollten das Leerzeichen (ASCII 32 bzw. CHAR(32)) und das Ausrufezeichen (!). Auch von der Verwendung des Dollar-Zeichens ($) wird abgeraten, insbesondere in Tabellennamen.
- Namen von Spalten und Tabellen dürfen nicht mit SQL Schlüsselwörtern identisch sein (z.B. INSERT, TEXT, SELECT usw.).
- Maximale Länge von Spaltennamen: 63 Zeichen
- Maximale Länge von Tabellennamen: 255 Zeichen
- Die Excel ODBC Treiber unterstützen die Excel Versionen 3.0, 4.0, 5.0/7.0, 97, 2000 sowie spätere, soweit sie kompatibel sind (dies dürfte wohl grundsätzlich der Fall sein).
- Es kann möglich sein, daß ein Excel ODBC Treiber nur eine begrenzte Anzahl gleichzeitig geöffneter Excel Dateien erlaubt. Möglicherweise kann ein Excel ODBC Treiber auch nur den Zugriff auf eine einzige Datei gestatten.
- Tabellen können nicht gelöscht werden.
- Namen von Tabellen können nicht geändert werden.

2. Kurzübersicht qtXLS Routinen

Die qtXLS Funktionen befinden sich in einer Dynamic-Link-Library (DLL) namens **qtXLS.dll**. Nachfolgende Tabelle führt die Routinen namentlich auf und gruppiert sie:

Eine ausführliche Beschreibung der qtXLS Routinen befindet sich im Kapitel “Referenz”

Funktionsgruppe / qtXLS Routine	**Funktion**
Dateifunktionen	
qtXLSCreateEXCELFile	Excel Datei erzeugen
qtXLSOpenEXCELFile	Excel Datei öffnen
qtXLSCloseEXCELFile	Excel Datei schließen
Tabellenfunktionen	
qtXLSCreateTable	Tabelle anlegen und Spalten definieren
qtXLSReadRows	Zeilen in Tabelle lesen
qtXLSWriteRows	Zeilen in eine Tabelle schreiben
Informationsfunktionen	
qtXLSGetTableNames	Tabellennamen bestimmen
qtXLSDoesTableNameExist	Prüfen, ob Tabelle existiert
qtXLSGetColumnInfo	Informationen über Spalten ermitteln
qtXLSGetRowCount	Anzahl der Zeilen in einer Tabelle ermitteln
qtXLSGetNumericValue	Wert des Typs “Numeric” ermitteln
Fehlerfunktionen	
qtXLSGetErrorMessages	Fehlermeldungen abfragen
qtXLSSetErrorLevel	Fehlerbehandlung steuern
qtXLSSetErrorMessagesDisplay	Anzeige von Fehlermeldungen einrichten
Sonstige Funktionen	
qtXLSGetszStringLength	Länge eines mit ASCII 0 terminierten Textes bestimmen
qtSetLicence_qtXLS	Nutzung von qtXLS lizensieren (in qtXLSSetLicence_0611_######.f90)
qtXLSSetLicencePath	Pfad für Lizenzdatei angeben (für ältere und spezielle qtXLS Versionen)

Eine ausführliche Beschreibung der qtXLS Routinen befindet sich im Kapitel “Referenz”.

2.1 Verwendung von qtXLS in eigenen Programmen

Um die qtXLS Routinen in eigenen Programmen zu verwenden, wird ein sogenanntes **Binding** benötigt, daß aus einer Import-Library und weiteren Dateien - bspw. prä-kompilierten Fortran 90 MODULE-Dateien (enden auf .mod) bzw. ein C-Header Datei (qtXLS.h) - besteht. Im Kapitel "Kompilieren & Binden" wird auf diese compiler-spezifischen Dateien eingegangen.
Des weiteren wird zur unbeschränkten Nutzung der qtXLS Funktionen entweder eine **Lizenzdatei** benötigt (dies war bei allen früheren Versionen von qtXLS notwendig) oder die **Lizenzroutine**, die beim Kauf von qtXLS geliefert wird. Die Lizenzdatei hat die Form

```
L####-######.lic (z.B. L0611-570739.lic)
```

wobei # eine der Ziffern 0 - 9 repräsentiert. Die Lizenzdatei enthält Lizenzinformationen zur Lizenz und zum Lizenznehmer. Bei Weitergabe von qtXLS basierenden Programmen (.exe), war und ist für einige spezielle Varianten von qtXLS diese Lizenzdatei und die qtXLS.dll mitzuliefern. Ohne eine gültige Lizenzinformationen funktioniert die qtXLS.dll nur im Demonstrations-Modus, der im wesentlichen durch beschränkte Lese- und Schreibfunktionalität gekennzeichnet ist.

Alternativ steht Fortran Programmierern die Lizenzroutine zur Verfügung, um die Lizenzinformationen bereitzustellen. Der Dateiname lautet analog zur obigen Lizenzdatei

```
qtSetLicence_####_######.f90
(z.B. qtSetLicence_0611_570739.f90)
```

Diese Lizenzdatei enthält die Lizenzroutine:

```
SUBROUTINE qtSetLicence_qtXLS( iError )
```

Diese Routine ist vor dem Aufruf aller anderen qtXLS Routinen aufzurufen. Andernfalls läuft qtXLS nur im Demo-Modus (es sei denn, eine Lizenzdatei wird von qtXLS.dll gefunden).

2.2 Struktur von qtXLS-Applikationen

Programme, die qtXLS Routinen verwenden ("qtXLS-Applikationen"), besitzen eine Struktur, die dem gängigen Dateizugriff gleicht.
Bevor eine Excel Datei geschrieben oder gelesen werden kann, muß sie mit qtXLSCreateEXCELFile erzeugt oder mit qtXLSOpenEXCELFile geöffnet werden. Dabei wird ein "Handle", d.h. eine Nummer erzeugt, anhand derer im folgenden auf die Datei zugegriffen werden kann. Z.B. in Fortran codiert man:

```
szFileName = 'qtXLSDemo01.xls' // CHAR(0)
hDS = qtXLSCreateEXCELFile( szFileName )
```

In C/C++ sieht dies sehr ähnlich aus (die Null-Terminierung von Strings erfolgt automatisch):

```
szFileName = 'qtXLSDemo01.xls';
hDS = qtXLSCreateEXCELFile( szFileName );
```

Die Datei wird geschlossen mit

```
iRet = qtXLSCloseEXCELFile( hDS );
```

Diese Routine bildet auch den Abschluß einer qtXLS-Applikation.

Zwischen diesen beiden Aufrufen sind die Routinen für das Erzeugen von Tabellen und den Zugriff auf sie einzubetten.

Sämtliche INTERFACE Blöcke bzw. Funktions-Prototypen zu qtXLS-Funktionen sind in einem MODULE bzw. einer C-Header-Datei zu finden. Dieses ist wie für MODULEs üblich zu Beginn der Variablen-Deklarationen anzugeben. In Fortran schreibt man:

```
USE qtXLS
```

In C/C++ lautet die entsprechende Anweisung:

```
#include <qtXLS.h>
```

Das MODULE qtXLS bzw. die C-Header-Datei stellen auch die Konstanten (in Fortran: KINDs bzw. PARAMETERs) und Strukturen (in Fortran: TYPEs; in C/C++: struct) zur Verfügung, die von qtXLS Routinen verwendet werden. Das MODULE qtXLS existiert in compiler-spezifischen Varianten und wird als Prä-Kompilat (Datei mit Endung .mod) bereitgestellt (vgl. Kapitel "Kompilieren & Binden").

Nachfolgendes Beispielprogramm zeigt, wie man mit qtXLS eine Excel Datei namens "qtXLSDemo3.xls" erzeugt, dort eine Tabelle "qtXLSDemoTable" mit den Spaltenköpfen "lfdNr", "x", "y", "Description" und "Date_Time" anlegt. Das Programm verzichtet auf jegliche Fehlerbehandlung. Eine vollständige Version findet sich in der Datei qtXLSDemoWriteTable.f90, die im gleichnamigen Unterverzeichnis des Verzeichnisses "Examples" der qtXLS-Installation abgelegt ist. Das gleiche Beispiel in C/C++ findet sich in der Datei qtXLSDemoWriteTable.cpp.

```
PROGRAM qtXLSDemoWriteTable
  USE qtXLS
  IMPLICIT NONE
! Arrays with data to be exported.
  INTEGER, PARAMETER :: DIMArr = 50, NoColums = 5
  CHARACTER(256) szTextArr(DIMArr)
  INTEGER lfdNrArr(DIMArr)                    ! INTEGER*4
  REAL (qt_K_R8) xArr(DIMArr), yArr(DIMArr)! REAL*8
  TYPE (qT_TIMESTAMP_STRUCT) TSArr(DIMArr)  ! date &
! other variables                          ! time structure
  REAL (qt_K_R8) angle
  REAL (qt_K_R8), PARAMETER :: PI = 3.1415932654D0
  INTEGER dtValues(8), iError
! variables to be used by qtXLS routines
  INTEGER (qt_K_HANDLE) hDS
  INTEGER (qt_K_INT4) iRet, iRow, TNLen,NoRows
  CHARACTER (20) szFileName
  TYPE (qT_SQLColumn) tColumns(NoColums)
  CHARACTER (qt_I_MaxTableNameLEN) szTableName
  CHARACTER (1000) szTableDefinition

! to change the path of licence file, if provided:
! CALL qtXLSSetLicencePath( szPathName )
! better set the licence by

CALL qtSetLicence_QTXLS( iError )
IF ( iError /= 0 )  &
PRINT*, 'Invalid licence, program runs in demo mode.'

! Fill arrays with some values (the data we're
! going to export into an EXCEL file)
! -----------------------------------------
  DO iRow = 1, DIMArr
    lfdNrArr(iRow) = iRow
    xArr(iRow) = iRow * 0.01
    angle = xArr(iRow) * PI
    yArr(iRow) = COS(angle)
    WRITE(szTextArr(iRow),"('Angle = ', F0.2,    &
```

```
                                ' (degree)', A1)")  &
          angle * 180. / PI, CHAR(0)
    CALL CONTAINS_SetTSArr( iRow ) ! see CONTAINS below
  END DO

! Create "empty" EXCEL file
! -------------------------
  szFileName = 'qtXLSDemo3.xls' // CHAR(0)
  hDS = qtXLSCreateEXCELFile( szFileName )

! Create (empty) table
! --------------------
  szTableName = 'qtXLSDemoTable' // CHAR(0)
  TNLen = qtXLSGetszStringLength( szTableName )
! returns length of string (without terminating zero)

! Set up a command line containing the table name
! followed by a list of pairs of column names and
! column types (like NUMBER, DATETIME, TEXT,
! CURRENCY or LOGICAL).
  szTableDefinition = szTableName(1:TNLen)         &
      // ' (lfdNr NUMBER, x NUMBER, y NUMBER,'     &
      // 'Description TEXT, Date_Time DATETIME)'   &
      // CHAR(0)
  iRet = qtXLSCreateTable( hDS, szTableDefinition  )

! Set up columns
!   "lfdNr   x   y   Description   Date_Time"
! for export
!-------------------------------------------
! 1st column
  tColumns(1) % Name       = 'lfdNr'     ! column name
  tColumns(1) % ArrayAddr  = LOC(lfdNrArr)  ! address
  tColumns(1) % ArrayDim   = DIMArr      ! array dim.
  tColumns(1) % ArrayType  = qt_SQL_C_SLONG ! long INT
  tColumns(1) % LENArrElem = 4      ! array elem. size
  tColumns(1) % IndArrAddr = 0           ! must be 0
! and remaining columns (using the TYPE constructor
! function qT_SQLColumn)
  tColumns(2) = qT_SQLColumn('x', LOC(xArr), DIMArr, &
                             qt_SQL_C_DOUBLE, 8, 0)
  tColumns(3) = qT_SQLColumn('y', LOC(yArr), DIMArr, &
                              qt_SQL_C_DOUBLE, 8, 0)
  tColumns(4) = qT_SQLColumn('Description',          &
                             LOC(szTextArr), DIMArr,&
                             qt_SQL_C_CHAR,         &
                             LEN(szTextArr(1)), 0)
  tColumns(5) = qT_SQLColumn('Date_Time', LOC(TSArr),&
                             DIMArr,                &
                            qt_SQL_C_TIMESTAMP, 16,0)
  NoRows = DIMArr  ! export all values in the arrays

! Fill table with rows
! --------------------
  iRet = qtXLSWriteRows( hDS, szTableName, NoColums, &
                         NoRows, tColumns )
  PRINT*, 'Number of rows written: ', iRet

! DONE. Close file and qtXLS.
  iRet = qtXLSCloseEXCELFile( hDS )
  STOP

CONTAINS
  SUBROUTINE CONTAINS_SetTSArr( j )
  ! fill date & time array TSArr() with some
  ! date & time values (just to have some example)
    INTEGER j

    IF ( j == 1 ) THEN
      CALL DATE_AND_TIME( VALUES = dtValues )
```

```
      TSArr(j) % year = dtValues(1)
      TSArr(j) % month = dtValues(2)
      TSArr(j) % day  = dtValues(3)
      TSArr(j) % hour = dtValues(5)
      TSArr(j) % minute= dtValues(6)
      TSArr(j) % second= dtValues(7)
      TSArr(j) % fraction = dtValues(8)/10 ! hundredths
    ELSE
    ! increment date and time
      TSArr(j) = TSArr(j-1)

      TSArr(j) % day  = TSArr(j-1) % day + 1
      IF ( TSArr(j) % day > 28 ) THEN
        TSArr(j) % day = 1
        TSArr(j) % month = TSArr(j-1) % month + 1
        IF ( TSArr(j) % month > 12 ) THEN
          TSArr(j) % month = 1
          TSArr(j) % year = TSArr(j-1) % year + 1
        END IF
      END IF

      TSArr(j) % second= TSArr(j-1) % second + 1
      IF ( TSArr(j) % second > 59 ) THEN
        TSArr(j) % second = 1
        TSArr(j) % minute = TSArr(j-1) % minute + 1
        IF ( TSArr(j) % minute > 59 ) THEN
          TSArr(j) % minute = 1
          TSArr(j) % hour = MOD(TSArr(j-1) % hour, 24) +
1
        END IF
      END IF
    END IF

    RETURN
  END SUBROUTINE
END
```

3. Referenz

3.1 Das Fortran 90 MODULE qtXLS und die C-Header-Datei qtXLS.h

Sämtliche INTERFACEs bzw. Prototypen zu qtXLS Funktionen sind in einem Fortran 90 MODULE bzw. in einer C-Header-Datei zu finden (lediglich die Routine qtSetLicence_QTXLS(...) ist davon ausgenommen). Das MODULE ist zu Beginn der Fortran Variablen-Deklarationen anzugeben. In Fortran schreibt man:

```
USE qtXLS
```

und in C/C++, wie üblich:

```
#include <qtXLS.h>
```

Das MODULE qtXLS verweist intern auf ein MODULE namens qtXLSDeclarations, daß die Konstanten (KINDs bzw. PARAMETERs) und Strukturen (TYPEs) zur Verfügung stellt, die von qtXLS Routinen verwendet werden. **C/C++ Programmierern genügt die qtXLS Header-Datei, die alles notwendige enthält.**

3.2 Das Fortran 90 MODULE qtXLSDeclarations

Auf das MODULE qtXLSDeclarations wird intern im MODULE qtXLS Bezug genommen. Es enthält die Konstanten (KINDs bzw. PARAMETERs) und Strukturen (TYPEs) die von qtXLS Routinen verwendet werden. Der Quellcode befindet sich in der Datei qtXLSDeclarations.f90, die im Verzeichnis "Bindings" der qtXLS Installation abgelegt ist. Das MODULE steht im Quellcode zur Verfügung, um Fortran Programmierern die Verwendung der Konstanten und Strukturen zu erleichtern. Eine explizite Einbindung (mittels USE) ist normalerweise nicht notwendig.

```
MODULE qtXLSDeclarations
! KINDs
    INTEGER, PARAMETER :: qt_K_INT1 = SELECTED_INT_KIND(2)  ! 1 Byte Integer
    INTEGER, PARAMETER :: qt_K_INT2 = SELECTED_INT_KIND(3)  ! 2 Byte Integer
    INTEGER, PARAMETER :: qt_K_INT4 = SELECTED_INT_KIND(9)  ! 4 Byte Integer
    INTEGER, PARAMETER :: qt_K_R4 = SELECTED_REAL_KIND(5,36)  ! 4 Byte REAL
    INTEGER, PARAMETER :: qt_K_R8 = SELECTED_REAL_KIND(15,307) !8 Byte REAL

    INTEGER, PARAMETER :: qt_K_LP = qt_K_INT4

    INTEGER, PARAMETER :: qt_K_SMALLINT = qt_K_INT2
    INTEGER, PARAMETER :: qt_K_UINTEGER = qt_K_INT4
    INTEGER, PARAMETER :: qt_K_INTEGER = qt_K_INT4
    INTEGER, PARAMETER :: qt_K_HANDLE = qt_K_INT4
    INTEGER, PARAMETER :: qt_K_RETURN = qt_K_SMALLINT

! Constants
    INTEGER, PARAMETER :: qt_I_MaxPathLEN = 1024
    INTEGER, PARAMETER :: qt_I_MaxTableNameLEN = 256
    INTEGER, PARAMETER :: qt_I_MaxColumneNameLEN = 64
    INTEGER, PARAMETER :: qt_I_MaxStatementLEN = 20480
    INTEGER, PARAMETER :: qt_I_SQLDataTypeLEN = 42

   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_CHAR = 1
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_NUMERIC = 2
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_DECIMAL = 3
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_INTEGER = 4
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_SMALLINT = 5
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_FLOAT = 6
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_REAL = 7
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_DOUBLE = 8
```

Das MODULE qtXLSDeclarations (Auszug aus der Datei qtXLSDeclarations.f90) - Fortsetzung s.u.

```
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_DATETIME = 9
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_VARCHAR = 12
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_DATE = 91
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_TIME = 92
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_TIMESTAMP = 93
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_DATE = 9
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_INTERVAL = 10
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TIME = 10
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TIMESTAMP = 11
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_LONGVARCHAR = -1
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_BINARY = -2
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_VARBINARY = -3
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_LONGVARBINARY = -4
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_BIGINT = -5
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TINYINT = -6
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_BIT = -7
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_GUID = -11

    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_CHAR = qt_SQL_CHAR
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_LONG = qt_SQL_INTEGER
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_SHORT = qt_SQL_SMALLINT
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_FLOAT = qt_SQL_REAL
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_DOUBLE = qt_SQL_DOUBLE
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_NUMERIC = qt_SQL_NUMERIC
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_DEFAULT = 99
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_SIGNED_OFFSET = -20
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_UNSIGNED_OFFSET = -22
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_DATE = qt_SQL_DATE
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TIME = qt_SQL_TIME
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TIMESTAMP = qt_SQL_TIMESTAMP
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TYPE_DATE = qt_SQL_TYPE_DATE
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TYPE_TIME = qt_SQL_TYPE_TIME
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TYPE_TIMESTAMP =   &
                                          qt_SQL_TYPE_TIMESTAMP
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_BINARY = qt_SQL_BINARY
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_BIT = qt_SQL_BIT
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_SBIGINT =    &
                                      qt_SQL_BIGINT+qt_SQL_SIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_UBIGINT =    &
                                    qt_SQL_BIGINT+qt_SQL_UNSIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_TINYINT = qt_SQL_TINYINT
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_SLONG =    &
                                      qt_SQL_C_LONG+qt_SQL_SIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_SSHORT =   &
                                     qt_SQL_C_SHORT+qt_SQL_SIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_STINYINT =   &
                                     qt_SQL_TINYINT+qt_SQL_SIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_ULONG =    &
                                     qt_SQL_C_LONG+qt_SQL_UNSIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_USHORT =   &
                                     qt_SQL_C_SHORT+qt_SQL_UNSIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_UTINYINT =   &
                                     qt_SQL_TINYINT+qt_SQL_UNSIGNED_OFFSET
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_BOOKMARK = qt_SQL_C_ULONG
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_GUID = qt_SQL_GUID
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_NULL = 0
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_MIN = qt_SQL_BIT
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_TYPE_MAX = qt_SQL_VARCHAR
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_C_VARBOOKMARK= qt_SQL_C_BINARY
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_NO_ROW_NUMBER = -1
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_NO_COLUMN_NUMBER = -1
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_ROW_NUMBER_UNKNOWN = -2
    INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_COLUMN_NUMBER_UNKNOWN = -2

! Error codes
    INTEGER, PARAMETER :: qtERRORBase = 70000
    INTEGER, PARAMETER :: qtERRORNotZeroTerminated    = qtERRORBase + 1
    INTEGER, PARAMETER :: qtERRORAllocHandleFailed    = qtERRORBase + 2
    INTEGER, PARAMETER :: qtERRORSQLFunctionFailed    = qtERRORBase + 3
    INTEGER, PARAMETER :: qtERRORConnectFailed     = qtERRORBase + 4
    INTEGER, PARAMETER :: qtERRORInsufficientDimension  = qtERRORBase + 5
    INTEGER, PARAMETER :: qtERRORNameNotSpecified   = qtERRORBase + 6
    INTEGER, PARAMETER :: qtERRORInvalid         = qtERRORBase + 7
    INTEGER, PARAMETER :: qtERRORExecDirectFailed   = qtERRORBase + 8
    INTEGER, PARAMETER :: qtERRORInsufficientSize   = qtERRORBase + 9
    INTEGER, PARAMETER :: qtERRORNotSupported      = qtERRORBase + 10
    !INTEGER, PARAMETER ::  = qtERRORBase + 1

    INTEGER, PARAMETER :: qtERRORUnknown = qtERRORBase + 200 ! the last one

! TYPEs
    TYPE qT_ColumnInfo
      SEQUENCE
      CHARACTER (qt_I_MaxColumneNameLEN) szName      ! column name
      INTEGER (qt_K_SMALLINT)            SQLDataType ! ODBC SQL data type
                  ! (e.g. qt_SQL_TYPE_DATE or qt_SQL_INTERVAL_YEAR_TO_MONTH
      CHARACTER (qt_I_SQLDataTypeLEN)    TypeName    ! column type name
                     ! (datasource dependent; for example, CHAR, VARCHAR,
                     !  MONEY, LONG VARBINAR, or CHAR ( ) FOR BIT DATA.
      INTEGER (qt_K_INT4)                MaxLen
    END TYPE

    TYPE qT_SQLColumn
      SEQUENCE
      CHARACTER (qt_I_MaxColumneNameLEN) Name    ! column name
      INTEGER (qt_K_LP)                  ArrayAddr ! LOC(array for results)
      INTEGER (qt_K_INT4)                ArrayDim  ! ArrayDimension
```

Das MODULE qtXLSDeclarations (Auszug aus der Datei qtXLSDeclarations.f90) - Fortsetzung

```
    INTEGER (qt_K_INT4)               ArrayType ! type of array
                  ! (qt_SQL_C_...: qt_SQL_C_SSHORT, qt_SQL_C_SLONG,
                  ! qt_SQL_C_DOUBLE, qt_SQL_C_FLOAT, qt_SQL_C_CHAR,
                  ! qt_SQL_C_DATE, qt_SQL_C_BIT ...). This usually also
                  ! determines the length of a single array element
                  ! (in bytes), but not for strings.
    INTEGER (qt_K_INT4)                LENArrElem! length of single array
                        ! element in characters in case of string arrays
    INTEGER (qt_K_LP)                  IndArrAddr! not used, should be 0
  END TYPE

  TYPE qT_DATE_STRUCT
    SEQUENCE
    INTEGER (qt_K_SMALLINT ) year
    INTEGER (qt_K_SMALLINT ) month
    INTEGER (qt_K_SMALLINT ) day
  END TYPE

  TYPE  qT_TIME_STRUCT
    SEQUENCE
    INTEGER (qt_K_SMALLINT)  hour
    INTEGER (qt_K_SMALLINT)  minute
    INTEGER (qt_K_SMALLINT)  second
  END TYPE

  TYPE  qT_TIMESTAMP_STRUCT
    SEQUENCE
    INTEGER (qt_K_SMALLINT)  year
    INTEGER (qt_K_SMALLINT)  month
    INTEGER (qt_K_SMALLINT)  day
    INTEGER (qt_K_SMALLINT)  hour
    INTEGER (qt_K_SMALLINT)  minute
    INTEGER (qt_K_SMALLINT)  second
    INTEGER (qt_K_UINTEGER)  fraction
  END TYPE

  INTEGER, PARAMETER :: qt_SQL_MAX_NUMERIC_LEN = 16

  TYPE  qT_NUMERIC_STRUCT
    SEQUENCE
    INTEGER (qt_K_INT1) precision
    INTEGER (qt_K_INT1) scale
    INTEGER (qt_K_INT1) sign               ! 1 if positive, 0 if negative
    INTEGER (qt_K_INT1) val(qt_SQL_MAX_NUMERIC_LEN)
  END TYPE
!-------------------------------------------------------------------------
! © Copyright QT software GmbH, Germany, 2007. All rights reserved.
END MODULE
```

Das MODULE qtXLSDeclarations (Auszug aus der Datei qtXLSDeclarations.f90)

Einige essentielle Konstanten und Strukturen werden in den folgenden Kapiteln erläutert.

3.3 Die C-Header-Datei qtXLS.h

Die C-Header-Datei qtXLS.h enthält die für die Verwendung in C/C++ Programmen notwendigen Konstanten (KINDs bzw. PARAMETERs), Strukturen und Funktionsprototypen. Diese werden in den folgenden Kapiteln erläutert.

```
//============================================================================
// qtXLS.h  -  Header file for qtXLS
//----------------------------------------------------------------------------
#ifndef _QTXLS_
#define _QTXLS_

#ifndef  _WINDOWS_
#include <windows.h>
#endif

/// MODULE qtXLSDeclarations

// Constants
const long  qt_I_MaxPathLEN = 1024;
const long  qt_I_MaxTableNameLEN   = 256;
const long  qt_I_MaxColumneNameLEN = 64;
const long  qt_I_MaxStatementLEN   = 20480;
const long  qt_I_SQLDataTypeLEN = 42;      /* max. length of SQL Data Type
names like VARCHAR, NUMBER etc. */

// KINDs
typedef char  qt_K_INT1;
typedef short   qt_K_INT2;
```

Die C-Header-Datei (Auszug aus der Datei qtXLS.h) - Fortsetzung s.u.

```
typedef long  qt_K_INT4;
typedef float   qt_K_R4;
typedef doubleqt_K_R8;

typedef longqt_K_LP;

typedef short   qt_K_SMALLINT;
typedef long  qt_K_UINTEGER;
typedef long  qt_K_INTEGER;
typedef long  qt_K_HANDLE;
typedef short   qt_K_RETURN;

typedef char type_charTableNamesRecord[qt_I_MaxTableNameLEN];

#define qt_SQL_CHAR   1
#define  qt_SQL_NUMERIC   2
#define  qt_SQL_DECIMAL   3
#define  qt_SQL_INTEGER   4
#define  qt_SQL_SMALLINT   5
#define  qt_SQL_FLOAT   6
#define  qt_SQL_REAL   7
#define  qt_SQL_DOUBLE   8
#define  qt_SQL_DATETIME   9
#define  qt_SQL_VARCHAR   12,
#define  qt_SQL_TYPE_DATE   91
#define  qt_SQL_TYPE_TIME   92
#define  qt_SQL_TYPE_TIMESTAMP   93,
#define  qt_SQL_DATE   9
#define  qt_SQL_INTERVAL   10
#define  qt_SQL_TIME   10
#define  qt_SQL_TIMESTAMP   11
#define  qt_SQL_LONGVARCHAR   -1
#define  qt_SQL_BINARY   -2
#define  qt_SQL_VARBINARY   -3
#define  qt_SQL_LONGVARBINARY   -4,
#define  qt_SQL_BIGINT   -5
#define  qt_SQL_TINYINT   -6
#define  qt_SQL_BIT   -7
#define  qt_SQL_GUID   -11

#define  qt_SQL_C_CHAR   qt_SQL_CHAR
#define  qt_SQL_C_LONG   qt_SQL_INTEGER
#define  qt_SQL_C_SHORT   qt_SQL_SMALLINT
#define  qt_SQL_C_FLOAT   qt_SQL_REAL
#define  qt_SQL_C_DOUBLE   qt_SQL_DOUBLE
#define  qt_SQL_C_NUMERIC   qt_SQL_NUMERIC
#define  qt_SQL_C_DEFAULT   99
#define  qt_SQL_SIGNED_OFFSET   -20
#define  qt_SQL_UNSIGNED_OFFSET   -22
#define  qt_SQL_C_DATE   qt_SQL_DATE
#define  qt_SQL_C_TIME   qt_SQL_TIME
#define  qt_SQL_C_TIMESTAMP   qt_SQL_TIMESTAMP
#define  qt_SQL_C_TYPE_DATE   qt_SQL_TYPE_DATE
#define  qt_SQL_C_TYPE_TIME   qt_SQL_TYPE_TIME
#define  qt_SQL_C_TYPE_TIMESTAMP   qt_SQL_TYPE_TIMESTAMP
#define  qt_SQL_C_BINARY   qt_SQL_BINARY
#define  qt_SQL_C_BIT   qt_SQL_BIT
#define  qt_SQL_C_SBIGINT   (qt_SQL_BIGINT+qt_SQL_SIGNED_OFFSET)
#define  qt_SQL_C_UBIGINT   (qt_SQL_BIGINT+qt_SQL_UNSIGNED_OFFSET)
#define  qt_SQL_C_TINYINT   qt_SQL_TINYINT
#define  qt_SQL_C_SLONG   (qt_SQL_C_LONG+qt_SQL_SIGNED_OFFSET)
#define  qt_SQL_C_SSHORT   (qt_SQL_C_SHORT+qt_SQL_SIGNED_OFFSET)
#define  qt_SQL_C_STINYINT   (qt_SQL_TINYINT+qt_SQL_SIGNED_OFFSET)
#define  qt_SQL_C_ULONG   (qt_SQL_C_LONG+qt_SQL_UNSIGNED_OFFSET)
#define  qt_SQL_C_USHORT   (qt_SQL_C_SHORT+qt_SQL_UNSIGNED_OFFSET)
#define  qt_SQL_C_UTINYINT   (qt_SQL_TINYINT+qt_SQL_UNSIGNED_OFFSET)
#define  qt_SQL_C_BOOKMARK   qt_SQL_C_ULONG
#define  qt_SQL_C_GUID   qt_SQL_GUID
#define  qt_SQL_TYPE_NULL   0
#define  qt_SQL_TYPE_MIN   qt_SQL_BIT
#define  qt_SQL_TYPE_MAX   qt_SQL_VARCHAR
#define  qt_SQL_C_VARBOOKMARK   qt_SQL_C_BINARY
#define  qt_SQL_NO_ROW_NUMBER   -1
#define  qt_SQL_NO_COLUMN_NUMBER   -1
#define  qt_SQL_ROW_NUMBER_UNKNOWN   -2
#define  qt_SQL_COLUMN_NUMBER_UNKNOWN   -2

// Error codes
const int  qtERRORBase  = 70000;
const int  qtERRORNotZeroTerminated     = qtERRORBase + 1;
const int  qtERRORAllocHandleFailed     = qtERRORBase + 2;
const int  qtERRORSQLFunctionFailed     =  qtERRORBase + 3;
const int  qtERRORConnectFailed       =  qtERRORBase + 4;
const int  qtERRORInsufficientDimension = qtERRORBase + 5;
const int  qtERRORNameNotSpecified    = qtERRORBase + 6;
const int  qtERRORInvalid           = qtERRORBase + 7;
const int  qtERRORExecDirectFailed    = qtERRORBase + 8;
const int  qtERRORInsufficientSize    = qtERRORBase + 9;
const int  qtERRORNotSupported        = qtERRORBase + 10;
// const int  qtERRORBase + 1;

const int  qtERRORUnknown = qtERRORBase + 200;   // the last error code

// STRUCTURES
struct qT_ColumnInfo {
   char    szName[qt_I_MaxColumneNameLEN];  // column name (zero terminated)
```

Die C-Header-Datei (Auszug aus der Datei qtXLS.h) - Fortsetzung

```
    qt_K_INT2 SQLDataType;                      // ODBC SQL data type
    char    TypeName[qt_I_SQLDataTypeLEN]; // column type name
    qt_K_INT4 MaxLen;
};

struct qT_SQLColumn {
    char    Name[qt_I_MaxColumneNameLEN ]; // column name
    qt_K_LP   ArrayAddr;                   // LOC(array for results)
    qt_K_INT4 ArrayDim;                    // ArrayDimension
    qt_K_INT4 ArrayType;                   // type of array
    qt_K_LP   IndArrAddr;                  // now: not used, should be 0
};

struct qT_DATE_STRUCT {
    qt_K_SMALLINT year;
    qt_K_SMALLINT month;
    qt_K_SMALLINT day;
};

struct qT_TIME_STRUCT {
    qt_K_SMALLINT hour;
    qt_K_SMALLINT minute;
    qt_K_SMALLINT second;
};

struct qT_TIMESTAMP_STRUCT {
    qt_K_SMALLINT year;
    qt_K_SMALLINT month;
    qt_K_SMALLINT day;
    qt_K_SMALLINT hour;
    qt_K_SMALLINT minute;
    qt_K_SMALLINT second;
    qt_K_UINTEGER fraction;
};

#define qt_SQL_MAX_NUMERIC_LEN   16

struct qT_NUMERIC_STRUCT {
    qt_K_INT1 precision;
    qt_K_INT1 scale;
    qt_K_INT1 sign;                  // 1 if positive, 0 if negative
    qt_K_INT1 val[qt_SQL_MAX_NUMERIC_LEN];
};

//------------------------------------------------------------------------------

// qtXLS PROTOTYPES / IMPORTED ROUTINES
/* Declare the routines imported from the qtXLS.dll.
   The "C" attribute prevents C++ name mangling  Remove it
   if the file type is .c
*/
#ifdef __cplusplus
extern "C"
#endif
{

#define qtXLSCreateEXCELFile  qtXLScr
qt_K_HANDLE WINAPI qtXLSCreateEXCELFile( LPSTR szFileName );

#define qtXLSOpenEXCELFile  qtXLSop
qt_K_HANDLE WINAPI qtXLSOpenEXCELFile( LPSTR szFileName );

#define qtXLSCloseEXCELFile  qtXLScl
qt_K_INTEGER WINAPI qtXLSCloseEXCELFile( qt_K_HANDLE hDS );

#define qtXLSCreateTable  qtXLSct
qt_K_INTEGER WINAPI qtXLSCreateTable( qt_K_HANDLE hDS,
                           LPSTR szTableDefinition  );

#define qtXLSGetTableNames  qtXLStn
VOID WINAPI qtXLSGetTableNames( qt_K_HANDLE hDS,
                        qt_K_INT4 iDIMcTableNames,
                        //char *cTableNames[][qt_I_MaxTableNameLEN],
                        //char *type_charTableNamesRecord[],
                        char *cTableNames,
                        qt_K_INT4 *iCountTableNames,
                        qt_K_INT4 *iError );

#define qtXLSGetColumnInfo  qtXLSci
VOID WINAPI qtXLSGetColumnInfo( qt_K_HANDLE hDS,
                        LPSTR szTableName,
                        qt_K_INT4 iDIMColumnInfo,
                        qT_ColumnInfo *ColumnInfo,
                        qt_K_INT4 *iCountColumnInfo,
                        qt_K_INT4 *iError );

#define qtXLSGetRowCount  qtXLSrc
qt_K_INTEGER WINAPI qtXLSGetRowCount( qt_K_HANDLE hDS,
                           LPSTR szTableName );

#define qtXLSReadRows  qtXLSrr
qt_K_INTEGER WINAPI qtXLSReadRows( qt_K_HANDLE hDS,
                         LPSTR szTableName,
                         qt_K_INTEGER iNoColums,
                         qt_K_INTEGER iNoRows,
                         qT_SQLColumn *tColumns,
                         LPSTR szCondition,
```

Die C-Header-Datei (Auszug aus der Datei qtXLS.h) - Fortsetzung

```
                              LPSTR szOrderBy);

#define qtXLSWriteRows  qtXLSwr
qt_K_INTEGER WINAPI qtXLSWriteRows( qt_K_HANDLE hDS,
                              LPSTR szTableName,
                              qt_K_INTEGER iNoColums,
                              qt_K_INTEGER iNoRows,
                              qT_SQLColumn *tColumns );

#define qtXLSGetErrorMessages  qtXLSem
qt_K_INTEGER WINAPI qtXLSGetErrorMessages( LPSTR szErrorMessages,
                                 qt_K_INTEGER iBufSizeErrorMessages );

#define qtXLSSetErrorLevel  qtXLSel
VOID WINAPI qtXLSSetErrorLevel( qt_K_INTEGER iErrorLevel  );

#define qtXLSCheckSQLReturnCode  qtXLScs
VOID WINAPI qtXLSCheckSQLReturnCode( qt_K_HANDLE hDS,
                              qt_K_HANDLE hBef,
                              qt_K_RETURN iSQLRetCode );

#define qtXLSSetErrorMessagesDisplay  qtXLSmd
VOID WINAPI qtXLSSetErrorMessagesDisplay( qt_K_INTEGER iDisplayErrorMessages
);

#define qtXLSGetNumericValue  qtXLSnv
qt_K_R8 WINAPI qtXLSGetNumericValue( qT_NUMERIC_STRUCT *NMVal );

#define qtXLSDoesTableNameExist  qtXLStx
qt_K_INTEGER WINAPI qtXLSDoesTableNameExist( qt_K_HANDLE hDS, LPSTR
szTableName );

#define qtXLSSetLicencePath  qtXLSlp
VOID WINAPI qtXLSSetLicencePath( LPSTR szPathName );

#define qtXLSGetszStringLength  qtXLSsl
qt_K_INTEGER WINAPI qtXLSGetszStringLength( LPSTR szString );

}
#endif // _QTXLS_
//-----------------------------------------------------------------------------
//(C) Copyright QT software GmbH, Germany. All rights reserved.
//=============================================================================
```

Die C-Header-Datei (Auszug aus der Datei qtXLS.h)

3.4 Grundsätzliches zum Aufruf der qtXLS Routinen

Einige grundsätzliche Prinzipien finden Anwendung bei der Namensgebung von Routinen und ihren Argumenten, was die Umsetzung bei der Programmierung erleichtern und die Fehlerwahrscheinlichkeit verringern soll.

3.4.1 Verwendung von KINDs bzw. vorgefertigten Datentypen

Die qtXLS Routinen verwenden Argumente, deren Datentypen in C/C++ über typedef und in Fortran über eine KIND Spezifikation bestimmt sind. Entsprechend deklariert man die Variablen. Z.B. in C/C++:

```
#include <qtXLS.h>
.
.
qt_K_HANDLE hDS;
```

Bzw. in Fortran:

```
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
```

Die korrekte Verwendung der Typen von Argumenten wird anhand der Prototypen bzw. INTERFACE Deklarationen, die in der C-Header-Datei bzw. im MODULE qtXLS integriert sind, geprüft (der Compiler sollte Syntaxfehler im Falle von Inkonsistenzen anzeigen).

3.4.2 Namen von Konstanten

Bei der Bezeichnung von Konstanten (PARAMETERs) wurde folgendes Schema verwandt:

Die C/C++ Variablentypen bzw. Fortran KIND Konstanten beginnen mit dem Präfix "qt_K_".

Die Konstanten, die Fehlercodes darstellen beginnen, mit "qtERROR".

Alle anderen Konstanten beginnen mit "qt_" gefolgt von einem oder mehreren Großbuchstaben, die den Typ der Konstante angeben. Die Typangabe wird vom eigentlichen Namen durch "_" abgetrennt. So bezeichnet "qt_I_SQLDataTypeLEN" eine INTEGER Konstante mit dem eigentlichen Namen "SQLDataTypeLEN".

3.4.3 Namensgebung von Routinen-Argumenten

Die C-Funktionsprototypen bzw. INTERFACEs der qtXLS Routinen verwenden bei der Bezeichnung von Argumenten Präfixe, die in Kleinbuchstaben den "sprechenden" Argumenten-Namen vorangestellt sind. Z.B.:

```
qt_K_INTEGER qtXLSCreateTable( qt_K_HANDLE hDS,
                               char* szTableDefinition );
```

bzw.

```
INTEGER FUNCTION qtXLSCreateTable( hDS,    &
                                   szTableDefinition )
```

Hier sind den Argumenten die Präfixe "h" und "sz" vorangestellt. Die Präfixe sollen den Programmierer an den zu verwenden Variablentyp erinnern. Nachfolgende Tabelle führt die verwendeten Präfixe und ihre Bedeutung bzw. den Variablentyp (KIND) auf.

Präfix	Bedeutung	C/C++ Variablentyp	Fortran Variablentyp bzw. KIND
i	langer "Integer"	qt_K_INT4 oder qt_K_INTEGER	INTEGER (qt_K_INT4)
h	Handle	qt_K_HANDLE	INTEGER (qt_K_HANDLE)
sz	string, zero terminated	char	CHARACTER (*)
c	CHARACTER bzw. Text	char	CHARACTER (*)
t	TYPE bzw. Struktur	struct	TYPE (...)

3.4.4 Namenslängen

Die Beschränkungen von Namensangaben von Pfaden, Tabellen und Spalten sind vorgegeben durch - in C/C++:

```
const long  qt_I_MaxPathLEN = 1024;
const long  qt_I_MaxTableNameLEN  = 256;
const long  qt_I_MaxColumneNameLEN = 64;
```

bzw. in Fortran:

```
INTEGER, PARAMETER :: qt_I_MaxPathLEN = 1024
INTEGER, PARAMETER :: qt_I_MaxTableNameLEN = 256
INTEGER, PARAMETER :: qt_I_MaxColumneNameLEN = 64
```

Hier ist zu beachten, daß Namensangaben meist durch das Zeichen "ASCII 0" (in Fortran: CHAR(0)) terminiert werden und sich somit die tatsächlich verwendbare Textlänge um ein Zeichen reduziert. Ein Spaltenname kann demzufolge nur 63 Zeichen lang sein (qt_I_MaxColumneNameLEN - 1). C/C++ Programmierern dürfte dies bekannt sein.

3.4.5 Null-terminierte Strings

Alle qtXLS Routinen Argumente deren Namen mit "sz" beginnen, sind mit "zero-terminated Strings" zu füllen. Hierzu fügt man dem Text ein "ASCII 0" (in Fortran: CHAR(0)) hinzu, welches das Ende der Zeichenkette anzeigt. Z.B. in Fortran:

```
szTableName = 'Koordinaten' // CHAR(0)
```

In C/C+ erfolgt dies normalerweise automatisch. Um die Länge eines derart spezifizierten Textes zu bestimmen, steht Fortran Programmierern die

```
INTEGER FUNCTION qtXLSGetszStringLength( szString )
```

zu Verfügung. C/C++ Programmierer können diese Funktion auch verwenden, werden aber vermutlich strlen() bevorzugen. qtXLSGetszStringLength() liefert die "operative" Länge, d.h. ohne terminierende Null zurück. Beispiel, in Fortran:

```
iLen = qtXLSGetszStringLength( szTableName )
```

und in C/C++

```
//              123456789 1
szTableName = 'Koordinaten';
iLen = strlen( szTableName );
```

liefern in iLen den Wert 11 zurück.

3.4.6 Strukturen (TYPEs bzw. structURES)

Einige der in der C-Header-Datei qtXLS.h bzw. im MODULE qtXLSDeclarations deklarierten Strukturen bedürfen der Erläuterung.

3.4.6.1 TYPE bzw. struct qT_ColumnInfo

Zur Information über Spalten wird eine Struktur namens qT_ColumnInfo verwandt. In Fortran:

```
TYPE qT_ColumnInfo
   SEQUENCE
   CHARACTER (qt_I_MaxColumneNameLEN)  szName
   INTEGER (qt_K_SMALLINT)             SQLDataType
   CHARACTER (qt_I_SQLDataTypeLEN)     TypeName
   INTEGER (qt_K_INT4)                 MaxLen
  END TYPE
```

und in C/C++:

```
struct qT_ColumnInfo {
  char        szName[qt_I_MaxColumneNameLEN];
  qt_K_INT2   SQLDataType;
  char        TypeName[qt_I_SQLDataTypeLEN];
  qt_K_INT4   MaxLen;
};
```

Die Komponente **szName** enthält den Namen einer Spalte (null-terminiert).

Der SQL Datentyp der Spalte wird in **SQLDataType** angegeben. Beim Aufruf der Routine qtXLSGetColumnInfo(...) liefert dieser Konstanten zurück, die den im MODULE qtXLSDeclarations bzw. in qtXLS.h deklarierten Parametern mit Namen "qt_SQL_" entsprechen.

Der Klartext des SQL Spaltentyps findet sich in der Komponente **TypeName**. Dies kann bspw. sein: DATETIME, NUMBER, VARCHAR, CURRENCY, LOGICAL, NUMERIC.

Die Länge der Datentyps bzw. die Größe des "Spaltenpuffers" liefert die Komponente **MaxLen**.
Im Falle des Typs DATETIME wird in ihr die Anzahl der Zeichen zurückgegeben, die notwendig ist, den Wert in Textform konvertiert darzustellen.
Im Falle des Typs NUMERIC liefert MaxLen entweder die Gesamtanzahl der Dezimalstellen oder die maximal mögliche Anzahl Bits der Spalte zurück.

3.4.6.2 TYPE bzw. struct qT_SQLColumn

Um Zeilen einer Excel Tabelle lesen oder beschreiben zu können, muß den Routinen qtXLSReadRows(...) bzw. qtXLSWriteRows(...) mitgeteilt werden, wo die Daten abgelegt werden bzw. wo sie zu finden sind. Dies erfolgt über die Struktur qT_SQLColumn. In C/C++:

```
struct qT_SQLColumn {
  char        Name[qt_I_MaxColumneNameLEN ];
  qt_K_LP     ArrayAddr;
  qt_K_INT4   ArrayDim;
  qt_K_INT4   ArrayType;
  qt_K_INT4   LENArrElem;
  qt_K_LP     IndArrAddr;
};
```

und in Fortran:

```
TYPE qT_SQLColumn
  SEQUENCE
  CHARACTER (qt_I_MaxColumneNameLEN)  Name
  INTEGER (qt_K_LP)                  ArrayAddr
  INTEGER (qt_K_INT4)                ArrayDim
  INTEGER (qt_K_INT4)                ArrayType
  INTEGER (qt_K_INT4)                LENArrElem
  INTEGER (qt_K_LP)                  IndArrAddr
END TYPE
```

Die Komponente **Name** gibt den Spaltennamen an. Der String kann null-terminiert sein. Ist er nicht null-terminiert, muß die Komponente mit Leerzeichen aufgefüllt sein ("blank padded").

Die Komponente **ArrayAddr** ist mit der Speicheradresse der Variablen bzw. des Feldes zu besetzen, in das Werte geschrieben bzw. von dem Werte gelesen werden. In C/C++ ermittelt man diese auf die jeweils angebrachte Weise (hängt von der Deklaration des Feldes ab). Je nach Fortran-Compiler bestimmt man die Adresse mithilfe einer INTRINSIC FUNCTION (da der Fortran Standard dafür keine generelle Funktion vorsieht, muß man auf compiler-spezifische Funktionen zurückgreifen). Z.B.:

```
USE qtXLS
TYPE (qT_SQLColumn) tColumn
REAL (qt_K_R8) xVector (1000)
  .
tColumn % ArrayAddr = LOC(xVector)
! LOC() returns address of array xVector
```

Die Größe des zuvor spezifizierten Feldes (array) ist in der Komponente **ArrayDim** anzugeben (Anzahl in Bytes).

Damit der Treiber weiß, von welchem Typ das Feld ist, gibt man dies in der Komponente **ArrayType** an.

Die Länge eines Feldelements (in Byte) wird in **LENArrElem** angegeben.

Nachfolgende Tabelle listet die möglichen Typangaben und zugehörigen C/C++ bzw. Fortran Variablentypen sowie ihre Elementlängen (in Byte) auf.

Variablentyp (C/C++ bzw. Fortran)	Typangabe in ArrayType	Angabe in LENArrElem
qt_K_INT2 bzw. INTEGER (qt_K_INT2)	qt_SQL_C_SSHORT	2
qt_K_INT4 bzw. INTEGER (qt_K_INT4)	qt_SQL_C_SLONG	4
qt_K_R4 bzw. REAL (qt_K_R4)	qt_SQL_C_FLOAT	4
qt_K_R8 bzw. REAL (qt_K_R8)	qt_SQL_C_DOUBLE	8
char bzw. CHARACTER (*)	qt_SQL_C_CHAR	sizeof() bzw. LEN(...)
struct qT_TIMESTAMP_STRUCT bzw. TYPE (qT_TIMESTAMP_STRUCT)	qt_SQL_C_DATE	16
BOOLEAN bzw. LOGICAL	qt_SQL_C_BIT	1

Die Komponente **IndArrAddr** wird nicht benutzt und muß mit 0 besetzt werden.

Nachfolgendes Beispiel zeigt eine typische Spezifikation für eine Spalte namens 'lfdNr'. Die zu lesenden oder zu schreibenden Werte der Spalte werden in dem Feld lfdNrArr() der Größe DIMArr abgelegt. Die Strukturvariable des TYPE qT_SQLColumn heißt tColumns. In C:

```
const qt_K_INTEGER  DIMArr = 50, NoColumns = 5;
char *sName = "lfdNr";
qt_K_INTEGER *lfdNrArr;
struct qT_SQLColumn *tColumns[NoColumns];

// create arrays
lfdNrArr = new qt_K_INTEGER [DIMArr ];
tColumns[0] = (qT_SQLColumn *) calloc(NoColumns,
sizeof(qT_SQLColumn));

strcpy(tColumn[0]->Name, sName);
tColumn[0]->ArrayAddr  = (qt_K_LP)lfdNrArr;
tColumn[0]->ArrayDim   = DIMArr ;
tColumn[0]->ArrayType  = qt_SQL_C_SLONG;
tColumn[0]->LENArrElem = 4;
tColumn[0]->IndArrAddr = 0;
```

In Fortran:

```
INTEGER, PARAMETER :: DIMArr = 100, NoColumns = 5
INTEGER (qt_K_INT4) lfdNrArr(DIMArr)
TYPE (qT_SQLColumn) tColumns(NoColumns)

tColumns(1) % Name        = 'lfdNr'
tColumns(1) % ArrayAddr   = LOC(lfdNrArr)
tColumns(1) % ArrayDim    = DIMArr
tColumns(1) % ArrayType   = qt_SQL_C_SLONG
tColumns(1) % LENArrElem  = 4
tColumns(1) % IndArrAddr  = 0
```

In Fortran kann die Spaltendefinition mithilfe des TYPE Konstruktors auch wie folgt codiert werden:

```
tColumns(1) = qT_SQLColumn( 'lfdNr', LOC(lfdNrArr),  &
                        DIMArr, qt_SQL_C_SLONG, 4, 0 )
```

Hier ist die Reihenfolge der Komponenten zu beachten!

3.4.6.3 TYPE bzw. struct qT_TIMESTAMP_STRUCT

Datums- und Zeitangaben in Spalten werden in einer Struktur vom Typ qT_TIMESTAMP_STRUCT angegeben. In Fortran:

```
TYPE  qT_TIMESTAMP_STRUCT
  SEQUENCE
  INTEGER (qt_K_SMALLINT) year
  INTEGER (qt_K_SMALLINT) month
  INTEGER (qt_K_SMALLINT) day
  INTEGER (qt_K_SMALLINT) hour
  INTEGER (qt_K_SMALLINT) minute
  INTEGER (qt_K_SMALLINT) second
  INTEGER (qt_K_UINTEGER) fraction
END TYPE
```

In C/C++:

```
struct qT_TIMESTAMP_STRUCT {
   qt_K_SMALLINT   year;
   qt_K_SMALLINT   month;
   qt_K_SMALLINT   day;
   qt_K_SMALLINT   hour;
   qt_K_SMALLINT   minute;
```

```
    qt_K_SMALLINT   second;
    qt_K_UINTEGER   fraction;
};
```

Alle Komponenten mit Ausnahme der letzten sind selbsterklärend. Die Komponente **fraction** gibt die Zeit in Nano-Sekunden (10^{-9}s) an (Wertebereich: 0 ns bis 999999999 ns).

3.4.7 Fehlercodes und Fehlerbehandlung

Tritt ein Fehler in einer qtXLS Routine auf, so wird der Fehlerzustand entweder durch einen Funktionsrückgabewert oder durch ein explizites Argument ("iError") angezeigt. Die Bedeutung des "aktuellen" Fehlercode kann durch Aufruf der Routine qtXLSGetErrorMessages(...) abgefragt werden. Er wird automatisch angezeigt, wenn der "Fehleranzeigemodus" mithilfe von qtXLSSetErrorMessagesDisplay(...) eingeschaltet worden ist.
Ein Fehler kann entweder durch fehlerhafte Argumentspezifikation zustande kommen oder durch unzureichende interne Puffergrößen oder durch einen Fehler, der im Excel ODBC Treiber auftritt. Die qtXLS spezifischen Fehlercodes sind im MODULE qtXLSDeclarations bzw. in der C-Header-Datei qtXLS.h deklariert. Nachfolgende Tabelle listet sie auf.

qtXLS Error Code	Bedeutung
qtERRORAllocHandleFailed	Ein interner Speicherbereich konnte nicht angelegt und somit kein Handle ermittelt werden.
qtERRORConnectFailed	Es konnte keine Verbindung zum Excel ODBC Treiber aufgebaut werden.
qtERRORExecDirectFailed	Der (interne) ODBC ExecDirect Befehl konnte nicht ausgeführt werden.
qtERRORInsufficientDimension	Die Dimension eines als Argument angegebenen Feldes (array) ist zu klein.
qtERRORInsufficientSize	Ein interner Puffer ist zu klein.
qtERRORInvalid	a) Der bei der Initialisierung von qtXLS überprüfte Lizenzcode ist ungültig. b) Ein Argument enthält einen ungültigen Wert. c) Eine Längenangabe fehlt (betr. qT_SQLColumn). d) Ein ungültiger Datentyp wurde angegeben (betr. qT_SQLColumn). e) Eine Addressangabe fehlt (betr. qT_SQLColumn). f) Die Dimension eines Feldes wurde fehlerhaft angegeben (betr. qT_SQLColumn).
qtERRORNameNotSpecified	Ein Name wurde nicht angegeben.
qtERRORNotSupported	Ein Datentyp wird nicht unterstützt (betr. qT_SQLColumn).
qtERRORNotZeroTerminated	Ein Text wurde nicht null-terminiert.
qtERRORSQLFunctionFailed	Ein Fehler ist bei der Ausführung einer Excel ODBC Treiber Funktion aufgetreten.
qtERRORUnknown	Unbekannter Fehler.

Normalerweise bricht eine qtXLS Routine im Fehlerfall ab und kehrt zum aufrufenden Programm zurück. Man kann aber durch Heraufsetzen des "Error Level" durch Aufruf (in Fortran)

```
CALL qtXLSSetErrorLevel( 1 )
```

bzw. in C/C++

```
qtXLSSetErrorLevel( 1 );
```

dafür sorgen, daß trotz internem Fehler versucht wird, einen Prozeß weiterzuführen, sofern dies möglich ist.

3.5 Beschreibung der qtXLS Routinen

qtSetLicence_qtXLS - Setze qtXLS Lizenz

C/C++:

```
void qtSetLicence_QTXLS( long iError );
```

Fortran:

```
SUBROUTINE qtSetLicence_qtXLS( iError )
INTEGER, INTENT(OUT) :: iError
```

Die Routine qtSetLicence_qtXLS(...) wird im Quellcode geliefert und enthält Informationen zum Lizenznehmer und den daraus abgeleiteten Lizenzcode.
Die Routine liefert in iError einen Fehlercode zurück. Ist dieser ungleich 0, liegt keine gültige Lizenz vor und qtXLS arbeitet dann nur noch im Demo-Modus.

Interna

qtSetLicence_qtXLS(...) verwendet intern das MODULE qtCompilerModule_QTXLS. Dieses compiler-spezifische MODULE ist einigen Compilern beim Compilieren des der Routine zugrundeliegenden Datei (qtSetLicence_####_######.f90) anzugeben. Vgl. Beispielprogramme.

qtXLSCloseEXCELFile - Schließe Excel Datei

C/C++:

```
qt_K_INTEGER qtXLSCloseEXCELFile(
                                qt_K_HANDLE hDS );
```

Fortran:

```
INTEGER FUNCTION qtXLSCloseEXCELFile( hDS )
INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
```

Die Routine qtXLSCloseEXCELFile(...) schließt eine Excel Datei, die zuvor mit qtXLSCreateEXCELFile erzeugt oder mit qtXLSOpenEXCELFile geöffnet worden ist. Das von beiden Routinen erzeugte Argument "Handle" (hDS) ist zu übergeben.

Interna

qtXLSCloseEXCELFile(...) trennt die Verbindung zum Excel ODBC Treiber und gibt, falls notwendig, den durch zuvor genannte Routinen allokierten Speicher wieder frei.

qtXLSCreateEXCELFile - Erzeuge Excel Datei

C/C++:

```
qt_K_HANDLE qtXLSCreateEXCELFile(
                                char *szFileName );
```

Fortran:

```
FUNCTION qtXLSCreateEXCELFile( szFileName )
INTEGER (qt_K_HANDLE) :: qtXLSCreateEXCELFile
CHARACTER (*), INTENT(IN) :: szFileName
```

Die Routine qtXLSCreateEXCELFile(...) erzeugt eine Excel Datei, deren Name im Argument szFileName anzugeben ist (null-terminiert).
Der Dateiname in szFileName kann eine Pfadangabe enthalten und muß am Ende mit ASCII 0 terminiert sein. Sinnvollerweise sollte der Dateiname auf .xls enden (Excel Dateien sind üblicherweise unter dieser Endung "registriert").
Existiert die Datei, die in szFileName angegeben ist, bereits, wird sie überschrieben (und ist dann zunächst "leer"). Ansonsten wird die Datei neu angelegt. Als Funktionswert wird ein "Handle" zurückgegeben, das in allen folgenden Aufrufen von qtXLS Routinen zu verwenden ist.
Im Fehlerfall wird 0 zurückgegeben, und der Fehlercode bzw. seine Bedeutung kann über qtXLSGetErrorMessages(...) abgefragt werden.

C/C++ Beispiel

```
#include <qtXLS.h>

int main(void);
{
qt_K_HANDLE hDS;
char *szFileName = "DataExport.xls";

hDS = qtXLSCreateEXCELFile( szFileName );
if ( hDS == 0 )
 printf("Error. File could not be created.");
else
 printf("Excel File has been created successfully.");
```

Fortran Beispiel

```
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
CHARACTER (100) :: szFileName

szFileName = 'DataExport.xls' // CHAR(0)
hDS = qtXLSCreateEXCELFile( szFileName )
IF ( hDS == 0 ) THEN
 PRINT*, 'Error. File could not be created.'
ELSE
 PRINT*, 'Excel File has been created successfully.'
END IF
```

Interna

qtXLSCreateEXCELFile(...) initialisiert qtXLS und stellt eine Verbindung zur Excel Datei vermittels des Excel ODBC Treibers her ("Connect") und allokiert dazu intern Speicher, der mittels qtXLSCloseEXCELFile(...) wieder freizugeben ist.

qtXLSCreateTable - Erzeuge Tabelle

C/C++:

```
qt_K_INTEGER qtXLSCreateTable(
                                qt_K_HANDLE hDS,
                      char *szTableDefinition  );
```

Fortran:

```
FUNCTION qtXLSCreateTable(              hDS, &
                               szTableDefinition )
INTEGER :: qtXLSCreateTable
INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
CHARACTER (*), INTENT(IN) :: szTableDefinition
```

Eine Excel Tabelle (bzw. Arbeitsblatt; auch "sheet" genannt) wird in einer zuvor mit qtXLSCreateEXCELFile(...) oder qtXLSOpenEXCELFile(...) geöffneten Datei mithilfe von qtXLSCreateTable(...) erzeugt. Diese Datei wird durch das von diesen beiden Routinen zurückgelieferte Handle (hDS) identifiziert. Die Tabelle ist über das Argument szTableDefinition zu definieren.

Tabellendefinition

Die Tabellendefinition besteht aus der Angabe des Tabellennamens und der Spaltennamen sowie ihren Typen in der Form:

"*Tabellenname* (*Spaltenname1 Spaltentyp1*, *Spaltenname2 Spaltentyp2*, ...)" // CHAR(0)

Bis zu 255 Spalten können auf diese Weise angelegt werden (interne Beschränkung von Excel). Bei der Namensangabe können theoretisch alle gültigen Excel Zeichen verwendet werden. Man tut jedoch gut daran sich auf den "lesbaren" US ASCII Satz (ASCII 32 bis 127) zu beschränken und darüber hinaus keine Leerzeichen (ASCII 32), kein Ausrufezeichen (!) und keine Dollar-Zeichen in den Namen zu verwenden. Außerdem muß darauf geachtet werden, daß die verwendeten Namen nicht mit SQL Schlüsselwörtern bzw. Befehlen (bspw. SELECT, INSERT, TEXT, NUMBER, MONEY etc.) übereinstimmen.
Für die Angabe des Spaltentyps kommen die nachfolgenden Bezeichner in Frage: CURRENCY, DATETIME, LOGICAL, NUMBER und TEXT.

Konnte die Tabelle erfolgreich angelegt werden, gibt qtXLSCreateTable(...) 0 als Funktionswert zurück. Ansonsten wird ein Fehlercode zurückgegeben, dessen Bedeutung mit qtXLSGetErrorMessages(...) abgefragt werden kann.

C/C++ Beispiel

```
#include <qtXLS.h>

int main(void);
{
  qt_K_HANDLE hDS;
```

```
 char *szTableDefinition = ”Coordinates (lfdNr
     NUMBER, x NUMBER, y NUMBER, Description TEXT,
     Date_Time DATETIME)”;

 hDS = qtXLSCreateTable( hDS, szTableDefinition );
 if ( hDS == 0 )
  printf(”Error. Excel sheet could not be created.”);
 else
  printf(”Excel table created with success.”);

}
```

Fortran Beispiel

```
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
CHARACTER (1000) szTableDefinition
INTEGER iRet

szTableDefinition = 'Coordinates ('              &
    // 'lfdNr NUMBER, x NUMBER, y NUMBER, '     &
    // 'Description TEXT, Date_Time DATETIME)'  &
    // CHAR(0)
iRet = qtXLSCreateTable( hDS, szTableDefinition )
IF ( iRet /= 0 ) THEN
 PRINT*, 'Error. Excel sheet could not be created.'
ELSE
 PRINT*, 'Excel table has been created successfully.'
END IF
```

qtXLSDoesTableNameExist - Prüfe, ob Tabelle existiert

C/C++:

```
qt_K_INTEGER qtXLSDoesTableNameExist(
                                qt_K_HANDLE hDS,
                            char *szTableName );
```

Fortran:

```
FUNCTION qtXLSDoesTableNameExist(      hDS, &
                                  szTableName )

LOGICAL :: qtXLSDoesTableNameExist

INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS

CHARACTER (*), INTENT(IN) :: szTableName
```

Mittels qtXLSDoesTableNameExist(...) läßt sich feststellen, ob eine in der durch das Handle hDS identifizierte Excel Datei eine Tabelle (bzw. ein Arbeitsblatt) enthält, deren Namen in szTableName angegeben ist (null-terminiert). Die Funktion gibt zurück:

- 1 : wenn die Tabelle existiert.
- 0 : wenn die Tabelle nicht vorhanden ist.
- -1 : wenn intern ein Fehler aufgetreten ist.

Im Fehlerfall kann der Fehlercode bzw. seine Bedeutung über qtXLSGetErrorMessages(...) abgefragt werden.

C/C++ Beispiel

```
#include <qtXLS.h>

int main(void);
{
```

```
  qt_K_HANDLE hDS;
  char *szTableName = ”Coordinates”;
  int iRet;

  iRet = qtXLSDoesTableNameExist( hDS, szTableName );
  switch ( iRet )
  {
    case  -1 :
      printf(”An error occurred.\n");
      break;
    case  0 :
      printf(”The Excel sheet does not exist.\n");
      break;
    case  1 :
      printf(”The Excel sheet already exists.\n");
      break;
  }
}
```

Fortran Beispiel

```
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
CHARACTER (qt_I_MaxTableNameLEN) szTableName
INTEGER iRet

szTableName = 'Coordinates' // CHAR(0)
iRet = qtXLSDoesTableNameExist( hDS, szTableName )
SELECT CASE ( iRet )
  CASE ( -1 )
    PRINT*, ‘An error occurred.’
  CASE ( 0 )
    PRINT*, ‘The Excel sheet does not exist.’
  CASE ( 1 )
    PRINT*, ‘The Excel sheet already exists.’
END SELECT
```

qtXLSGetColumnInfo - Hole Spalteninformation

C/C++:

```
void qtXLSGetColumnInfo(     qt_K_HANDLE hDS,
                          char* szTableName,
                    qt_K_INT4 iDIMColumnInfo,
                  qT_ColumnInfo *ColumnInfo,
                qt_K_INT4 *iCountColumnInfo,
                        qt_K_INT4 *iError );
```

Fortran:

```
SUBROUTINE qtXLSGetColumnInfo(          hDS, &
                             szTableName, &
                          iDIMColumnInfo, &
                             tColumnInfo, &
                       iCountColumnInfo, &
                                iError )

INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
CHARACTER (*), INTENT(IN)         :: szTableName
INTEGER (qt_K_INT4), INTENT(IN)   :: iDIMColumnInfo
TYPE (qT_ColumnInfo), INTENT(OUT) :: tColumnInfo(  &
                                     iDIMColumnInfo)
INTEGER (qt_K_INT4), INTENT(OUT)  :: iCountColumnInfo
INTEGER (qt_K_INT4), INTENT(OUT)  :: iError
```

Mithilfe von qtXLSGetColumnInfo(...) kann Information über die Spalten einer Tabelle, deren Name in szTableName anzugeben ist (null-terminiert), beschafft werden. Die Excel Datei, deren Tabelleninformation abgefragt wird, wird durch das Handle hDS identifiziert (siehe qtXLSCreateEXCELFile(...) bzw. qtXLSOpenEXCELFile(...)). Die Spalteninformation wird in dem Strukturfeld tColumnInfo() zurückgegeben. Das Feld muß iDIMColumnInfo Elemente enthalten. Befinden sich in der Tabelle mehr Spalten als tColumnInfo() Elemente besitzt, liefert die Routine den Fehlercode iError = qtERRORInsufficientDimension zurück. Die benötigte Dimension (d.h. die Anzahl Spalten) des Strukturfelds tColumnInfo() findet sich dann in iCountColumnInfo. Wenn kein Fehler auftritt (iError = 0), gibt iCountColumnInfo ebenfalls die Anzahl der Spalten in der Tabelle zurück.
Die Spalteninformation in tColumnInfo(*j*) besteht aus dem Namen der Spalte *j* (null-terminiert oder mit Leerzeichen aufgefüllt), deren SQL Datentyp (repräsentiert durch eine Zahl), der Typbezeichnung im Klartext (DATETIME, NUMBER, VARCHAR, CURRENCY, LOGICAL oder NUMERIC) und der Länge der Spalte bzw. des Spaltenpuffers (in Byte). In C/C++ ist die Struktur wie folgt definiert (vgl. qtXLS.h):

```
struct qT_ColumnInfo {
  char        szName[qt_I_MaxColumneNameLEN];
  qt_K_INT2   SQLDataType;
  char        TypeName[qt_I_SQLDataTypeLEN];
  qt_K_INT4   MaxLen;
};
```

Und in Fortran:

```
TYPE qT_ColumnInfo
  SEQUENCE
  CHARACTER (qt_I_MaxColumneNameLEN)  szName
  INTEGER (qt_K_SMALLINT)             SQLDataType
  CHARACTER (qt_I_SQLDataTypeLEN)     TypeName
  INTEGER (qt_K_INT4)                 MaxLen
END TYPE
```

Näheres zum TYPE qT_ColumnInfo findet sich im gleichnamigen Kapitel 3.4.6.1.

■ C/C++ Beispiel

```
// see demo program qtXLSDemoListTablenames.cpp
// for another example
#include <stdio.h>
#include <qtXLS.h>

int main(void)
{
  qt_K_HANDLE hDS;
  char *szFileName;
  char *szTableName;
  qt_K_INT4 iError;
  qt_K_INTEGER iRet, indCol,
               iDIMColumnInfo, iCountColumnInfo;
  qT_ColumnInfo *ColumnInfo;

  szFileName = "Weather20030416.xls";
  hDS = qtXLSOpenEXCELFile( szFileName );

  szTableName = 'Temperatures'
  // column names
  indCol = -1;
  iCountColumnInfo = 0;
  iDIMColumnInfo = 0;
  while ( indCol < iCountColumnInfo )
```

```
 {
  qtXLSGetColumnInfo( hDS,
                      szTableName,
                      iDIMColumnInfo,
                      ColumnInfo,
                      &iCountColumnInfo,
                      &iError );
  if ( iError == qtERRORInsufficientDimension )
  {
   iDIMColumnInfo = iCountColumnInfo;
   ColumnInfo = new qT_ColumnInfo[iDIMColumnInfo];
  }
  else if ( iError != 0 )
  {
   printf("   Error (qtXLSGetColumnInfo),
          iError = %d\n", iError);
   break;
  }

  if ( indCol >= 0 )
  {
   printf("         %s   %d   %s   %d\n",
            ColumnInfo[indCol].szName,
            ColumnInfo[indCol].SQLDataType,
            ColumnInfo[indCol].TypeName,
            ColumnInfo[indCol].MaxLen);
  }
  indCol = indCol + 1;
 } // while ( indCol < iCountColumnInfo )
 delete ColumnInfo;

 iRet = qtXLSCloseEXCELFile( hDS );
 return iRet;
}
```

Fortran Beispiel

```
! see also demo program qtXLSDemoListTablenames.f90
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
CHARACTER (qt_I_MaxTableNameLEN) szTableName
INTEGER (qt_K_INT4) iDIMColumnInfo,                  &
                    iCountColumnInfo,iError
INTEGER iRet, iLen, jLen, indCol
TYPE(qT_ColumnInfo), ALLOCATABLE :: tColumnInfo(:)

hDS = qtXLSOpenEXCELFile( 'Weather20030416.xls'      &
                          // CHAR(0) )
szTableName = 'Temperatures' // CHAR(0)
iDIMColumnInfo = 0  ! causes qtXLSGetColumnInfo(...)
                    ! to return number of columns
indCol = 0; iCountColumnInfo = 1
DO WHILE ( indCol < iCountColumnInfo )
 CALL qtXLSGetColumnInfo( hDS, cTableNames(ind),     &
                          iDIMColumnInfo, tColumnInfo, &
                          iCountColumnInfo, iError )
 IF ( iError == qtERRORInsufficientDimension ) THEN
  iDIMColumnInfo = iCountColumnInfo
  ALLOCATE(tColumnInfo(iDIMColumnInfo))
 ELSE IF ( iError /= 0 ) THEN
  PRINT*, 'Error (qtXLSGetColumnInfo), iError =',   &
          iError
  EXIT
 END IF
 indCol = indCol + 1
 IF ( indCol > 0 ) THEN
  iLen = qtXLSGetszStringLength(                     &
                       tColumnInfo(indCol) % szName )
  jLen = qtXLSGetszStringLength(                     &
                      tColumnInfo(indCol) % TypeName )
```

```
   PRINT*,  tColumnInfo(indCol) % szName(1:iLen),     &
            tColumnInfo(indCol) % SQLDataType,        &
            tColumnInfo(indCol) % TypeName(1:jLen),   &
            tColumnInfo(indCol) % MaxLen
  END IF
END DO
DEALLOCATE(tColumnInfo)
```

qtXLSGetErrorMessages - Hole Fehlermeldungen

C/C++:

```
qt_K_INTEGER qtXLSGetErrorMessages(
                          char *szErrorMessages,
        qt_K_INTEGER iBufSizeErrorMessages );
```

Fortran:

```
FUNCTION qtXLSGetErrorMessages(              &
                               szErrorMessages, &
                          iBufSizeErrorMessages )
INTEGER qtXLSGetErrorMessages
CHARACTER (*), INTENT(OUT) :: szErrorMessages
INTEGER, INTENT(IN)        :: iBufSizeErrorMessages
```

Tritt während der Ausführung einer qtXLS Routine ein Fehler auf, wird die Fehlernummer intern gespeichert und die zugehörige Fehlermeldung kann dann mittels qtXLSGetErrorMessages(...) ermittelt werden. Die Routine gibt im Argument szErrorMessages die Fehlermeldung null-terminiert zurück. Die Größe dieses Puffers muß beim Aufruf in iBufSizeErrorMessages angegeben werden. Ein Fehlertextpuffer mit Platz für ca. 2000 Zeichen dürfte für die meisten Fälle genügen.
qtXLSGetErrorMessages(...) gibt als Funktionswert zurück:

- 0 : entweder weil kein intern gespeicherter Fehlercode vorlag oder wenn zu einem intern gespeicherten Fehlercode die Fehlermeldung vollständig bestimmt werden konnte.
- -1 : wenn bei der Ausführung von qtXLSGetErrorMessages(...) ein Fehler aufgetreten ist.
- >0 : wenn der Puffer szErrorMessages zu klein ist. Der zurückgegebene Wert gibt dann die notwendige Größe des Puffers (in Zeichen) an.

C/C++ Beispiel

```
#include <qtXLS.h>


char *szErrorMessages;
int iBufSizeErrorMessages, iRet;

iBufSizeErrorMessages = 1000;
szErrorMessages = new char [iBufSizeErrorMessages];
iRet = qtXLSGetErrorMessages( szErrorMessages,
                              iBufSizeErrorMessages );
if ( iRet > 0 )
{
 printf("Insufficient length of szErrorMessages.");
 printf(" Required length: %d\n', iRet);
}
else if ( iRet == -1 )
 printf("Error performing qtXLSGetErrorMessages\n.");
```

```
else
  printf(”Error messages: %s\n”,szErrorMessages);
```

Fortran Beispiel

```
USE qtXLS
CHARACTER (1000) szErrorMessages
INTEGER iBufSizeErrorMessages, iRet

iBufSizeErrorMessages = LEN( szErrorMessages )
iRet = qtXLSGetErrorMessages( szErrorMessages,       &
                              iBufSizeErrorMessages )
IF ( iRet > 0 ) THEN
  PRINT*, 'Insufficient length of szErrorMessages.'
  PRINT*, 'Required length:', iRet
ELSE IF ( iRet == -1 ) THEN
  PRINT*, 'Error performing qtXLSGetErrorMessages.'
ELSE
  PRINT*, 'Error messages: ',szErrorMessages
END IF
```

qtXLSGetNumericValue - Bestimme Gleitkommazahl

C/C++:

```
qt_K_R8 qtXLSGetNumericValue(
                    qT_NUMERIC_STRUCT *NMVal );
```

Fortran:

```
FUNCTION qtXLSGetNumericValue( NMVal )
REAL (qt_K_R8) qtXLSGetNumericValue
TYPE (qT_NUMERIC_STRUCT), INTENT (IN) :: NMVal
```

Werden aus einer Excel Tabelle Spalten vom Typ NUMERIC gelesen, so kommt man nicht umhin, die Werte in einen elementaren Gleitkommatyp umzuwandeln, der von Fortran, C etc. unterstützt wird. Der NUMERIC Typ wird beim Lesen der Excel Tabelle in einer Struktur qT_NUMERIC_STRUCT abgelegt. In C/C++ ist sie wie folgt definiert:

```
struct qT_NUMERIC_STRUCT {
  qt_K_INT1   precision;
  qt_K_INT1   scale;
  qt_K_INT1   sign;
  qt_K_INT1   val[qt_SQL_MAX_NUMERIC_LEN];
};
```

Und in Fortran:

```
TYPE  qT_NUMERIC_STRUCT
  SEQUENCE
  INTEGER (qt_K_INT1) precision
  INTEGER (qt_K_INT1) scale
  INTEGER (qt_K_INT1) sign
  INTEGER (qt_K_INT1) val(qt_SQL_MAX_NUMERIC_LEN)
END TYPE
```

Die Funktion qtXLSGetNumericValue(...) wandelt den Wert dann in eine 8-Byte Gleitkommazahl (Typ double bzw. REAL*8) um und gibt ihn zurück.

qtXLSGetszStringLength - Bestimme Länge eines szString

C/C++:

```
qt_K_INTEGER qtXLSGetszStringLength(
                                char *szString );
```

Fortran:

```
FUNCTION qtXLSGetszStringLength( szString )
INTEGER qtXLSGetszStringLength
CHARACTER (*), INTENT(IN) :: szString
```

Die Funktion qtXLSGetszStringLength(...) bestimmt die operative Länge einer null-terminierten Zeichenkette (zero-terminated string). Zurückgegeben wird die Länge der Zeichenkette ohne die terminierende Null.

Fortran Beispiel

```
USE qtXLS
CHARACTER (8) szStr8
INTEGER iLen
!         1234567
szStr8 = 'Example' // CHAR(0)
iLen = qtXLSGetszStringLength( szStr8 )
! iLen = 7, now.
```

qtXLSGetTableNames - Bestimme Tabellennamen

C/C++:

```
void qtXLSGetTableNames(     qt_K_HANDLE hDS,
                        qt_K_INT4 iDIMcTableNames,
                              char *cTableNames,
                     qt_K_INT4 *iCountTableNames,
                              qt_K_INT4 *iError );
```

Fortran:

```
SUBROUTINE qtXLSGetTableNames(          hDS, &
                                iDIMTableNames, &
                                  szTableNames, &
                               iCountTableNames, &
                                        iError )
INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
INTEGER (qt_K_INT4), INTENT(IN)   :: iDIMTableNames
CHARACTER (*), INTENT(OUT)        ::                &
                          szTableNames(iDIMTableNames)
INTEGER (qt_K_INT4), INTENT(OUT)  :: iCountTableNames
INTEGER (qt_K_INT4), INTENT(OUT)  :: iError
```

Die Namen aller Tabellen einer Excel Datei, die über das Argument hDS identifiziert werden (siehe qtXLSCreateEXCELFile(...) oder qtXLSOpenEXCELFile(...)), können durch Aufruf von qtXLSGetTableNames(...) ermittelt werden.
Die Namen werden null-terminiert und mit Leerzeichen aufgefüllt im CHARACTER Feld szTableNames() abgelegt (man beachte die Längendeklaration qt_I_MaxTableNameLEN!). Das Feld ist mit iDIMTableNames dimensioniert. In C/C++ ist dazu ein Array aus Strings

mit fester Längenvorgabe qt_I_MaxTableNameLEN zu definieren (etwa: char szTableNames[iDIMTableNames][qt_I_MaxTableNameLEN];).
Die Anzahl der vorhandenen bzw. gefundenen Tabellen in der Excel Datei wird in iCountTableNames zurückgegeben. Das letzte Argument iError gibt 0 zurück, wenn kein Fehler bei der Durchführung der Routine aufgetreten ist. Ansonsten enthält sie einen Fehlercode (qtERROR...). Wenn iError den Fehlercode qtERRORInsufficientDimension zurückgibt, enthält iCountTableNames die erforderliche Dimension des Felds szTableNames(), d.h. die Anzahl der Tabellen.
Die in szTableNames() zurückgegebenen Tabellennamen besitzen als letztes Zeichen zumeist ein $ (bspw. “Coordinates$”), selbst dann, wenn die Tabelle mithilfe von qtXLSCreateTable(...) angelegt wurde und dort der Name ohne $ angegeben wurde. Dies ist eine Eigenart des Excel ODBC Treibers, zu deren Hintergründen leider keine weitere Information zu erhalten war. Tests ergaben, das der Name einer mithilfe von qtXLSCreateTable(...) erzeugten Tabelle später mit und auch ohne “$” erkannt wird. Man kann also die über qtXLSGetTableNames(...) ermittelten Tabellennamen in folgenden Aufrufen von qtXLS Routinen getrost weiterverwenden - ohne zuvor das Dollar-Zeichen am Ende entfernen zu müssen.

C/C++ Beispiel

```
#include <stdio.h>
#include <qtXLS.h>

int main(void)
{
  char *szFileName = "qtXLSDemo1.xls";
  qt_K_HANDLE hDS;
  qt_K_INT4 iDIMcTableNames;
  char *cTableNames, *cTableNamesInd;  // -> string
array with elements of length qt_I_MaxTableNameLEN
  qt_K_INT4 iCountTableNames, iError;
  qt_K_INTEGER iRet, ind;

  hDS = qtXLSOpenEXCELFile( szFileName );

  // first call to qtXLSGetTableNames to obtain
  // the number of tables in the file.
  iDIMcTableNames = 0;
M100:
  if ( iDIMcTableNames > 0 )
    cTableNames = ((char *) calloc(iDIMcTableNames,
                                   qt_I_MaxTableNameLEN));
  qtXLSGetTableNames( hDS, iDIMcTableNames,
                      cTableNames,
                      &iCountTableNames,
                      &iError );
  if ( iError == qtERRORInsufficientDimension )
  {
    if ( iCountTableNames == 0 )
      printf("No tables could be found.\n");
    else
    { /*  create string array cTableNames (each element
          holds a string up to [qt_I_MaxTableNameLEN]
          characters) */
      iDIMcTableNames = iCountTableNames;
      cTableNames = ((char *) calloc(iDIMcTableNames,
                                     qt_I_MaxTableNameLEN));
      goto M100;
    }
  }
  else if ( iError == 0 )
  {
    printf("   Table names:");
```

```
   for (ind = 0; ind < iCountTableNames; ind++)
   {
    cTableNamesInd = cTableNames
                   + ind * qt_I_MaxTableNameLEN;
    printf("      %s\n", cTableNamesInd);
   } // for (ind = 0, ind < iCountTableNames, ind++)
  } // if ( iError == 0 )
  else
     printf("   Error (qtXLSGetTableNames),
            iError = %d\n", iError);

  iRet = qtXLSCloseEXCELFile( hDS );
  return 0;
}
```

Fortran Beispiel

```
! see also demo program qtXLSDemoListTablenames.f90
USE qtXLS
INTEGER (qt_K_HANDLE) hDS
INTEGER (qt_K_INT4) iDIMTableNames
CHARACTER (qt_I_MaxTableNameLEN), ALLOCATABLE ::     &
                                        szTableNames(:)
INTEGER (qt_K_INT4) iCountTableNames, iError
INTEGER iRet, ind, iLen

hDS = qtXLSOpenEXCELFile( 'Data021.xls'C )

iDIMTableNames = 0    ! to obtain the number of tables
CALL qtXLSGetTableNames( hDS, iDIMTableNames,        &
               szTableNames,iCountTableNames, iError )
IF ( iError == qtERRORInsufficientDimension ) THEN
 iDIMTableNames = iCountTableNames
 ALLOCATE(szTableNames(iDIMTableNames))
ELSE
 iRet = qtXLSCloseEXCELFile( hDS )
 STOP
END IF

CALL qtXLSGetTableNames( hDS, iDIMTableNames,        &
               szTableNames,iCountTableNames, iError )
IF ( iError == 0 ) THEN
 PRINT*, 'Table names:'
 DO ind = 1, iCountTableNames
  iLen = qtXLSGetszStringLength( szTableNames(ind) )
  PRINT*, szTableNames(ind)(1:iLen)
 END DO
ELSE
 PRINT*, 'Error performing qtXLSGetTableNames:',     &
         iError
END IF
```

qtXLSGetRowCount - Bestimme Zeilenanzahl

C/C++:

```
qt_K_INTEGER qtXLSGetRowCount(
                                qt_K_HANDLE hDS,
                              char *szTableName );
```

Fortran:

```
FUNCTION qtXLSGetRowCount( hDS, szTableName )
INTEGER qtXLSGetRowCount
INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
CHARACTER (*), INTENT(IN) :: szTableName
```

Die Anzahl der Zeilen in einer Tabelle, deren Name in szTableName (null-terminiert) angegeben wurde, gibt die Funktion qtXLSGetRowCount(...) zurück. Das Argument hDS identifiziert die Excel Datei (siehe qtXLSCreateEXCELFile(...) oder qtXLSOpenEXCELFile(...)), in der sich die Tabelle befindet.
Falls ein Fehler bei der Ausführung von qtXLSGetRowCount(...) auftritt, wird der Wert -1 zurückgegeben. Die Bedeutung des aufgetretenen Fehlers kann durch Aufruf der Routine qtXLSGetErrorMessages(...) abgefragt werden.

C/C++ Beispiel

```
#include <stdio.h>
#include <qtXLS.h>


char *szTableName = "Coordinates";
qt_K_HANDLE hDS;
int noRows;

noRows = qtXLSGetRowCount( hDS, szTableName );
if ( noRows == -1 )
 printf("Error calling qtXLSGetRowCount(...).\n");
else
 printf("Count rows: %d.\n", noRows);
```

Fortran Beispiel

```
USE qtXLS
INTEGER (qt_K_HANDLE) hDS
CHARACTER (qt_I_MaxTableNameLEN) :: szTableName
INTEGER noRows

szTableName = 'Coordinates' // CHAR(0)
noRows= qtXLSGetRowCount( hDS, szTableName )
IF ( noRows == -1 ) THEN
 PRINT*, 'Error calling qtXLSGetRowCount(...)'
ELSE
 PRINT*, 'Count rows:', noRows
END IF
```

qtXLSOpenEXCELFile - Öffne Excel Datei

C/C++:

```
qt_K_HANDLE qtXLSOpenEXCELFile(
                              char *szFileName );
```

Fortran:

```
FUNCTION qtXLSOpenEXCELFile( szFileName )
INTEGER (qt_K_HANDLE) qtXLSOpenEXCELFile
CHARACTER (*), INTENT(IN) :: szFileName
```

Eine Excel Datei, deren Namen in szFileName angegeben ist, wird mittels qtXLSOpenEXCELFile(...) geöffnet. Der Dateiname in szFileName kann eine Pfadangabe enthalten und muß am Ende mit ASCII 0 terminiert sein. Die Funktion liefert, falls die Datei existiert und geöffnet werden konnte, ein Handle zurück, das in allen folgenden Aufrufen von qtXLS Routinen zu verwenden ist.
Im Fehlerfall wird 0 zurückgegeben, und der Fehlercode bzw. seine Bedeutung kann über qtXLSGetErrorMessages(...) abgefragt werden.

C/C++ Beispiel

```
#include <stdio.h>
#include <qtXLS.h>

qt_K_HANDLE hDS;
char *szFileName

szFileName = "DataExport.xls";
hDS = qtXLSOpenEXCELFile( szFileName );
if ( hDS == 0 )
 printf(”Error. File could not be opened.”);
else
 printf(”Excel File has been opened successfully.”);
```

Fortran Beispiel

```
USE qtXLS
INTEGER (qt_K_HANDLE) :: hDS
CHARACTER (100) :: szFileName

szFileName = 'DataExport.xls' // CHAR(0)
hDS = qtXLSOpenEXCELFile( szFileName )
IF ( hDS == 0 ) THEN
 PRINT*, 'Error. File could not be opened.'
ELSE
 PRINT*, 'Excel File has been opened successfully.'
END IF
```

Interna

qtXLSOpenEXCELFile(...) initialisiert qtXLS und stellt eine Verbindung zur Excel Datei vermittels des Excel ODBC Treibers her (”Connect”) und allokiert dazu intern Speicher, der mittels qtXLSCloseEXCELFile(...) wieder freizugeben ist.

qtXLSReadRows - Lies Zeilen

C/C++:

```
qt_K_INTEGER qtXLSReadRows(  qt_K_HANDLE hDS,
                           char *szTableName,
                     qt_K_INTEGER iNoColums,
                       qt_K_INTEGER iNoRows,
                     qT_SQLColumn *tColumns,
                           char *szCondition,
                            char *szOrderBy);
```

Fortran:

```
FUNCTION qtXLSReadRows(                 hDS, &
                                  szTableName, &
                                    iNoColums, &
                                      iNoRows, &
                                     tColumns, &
                                  szCondition, &
                                     szOrderBy )

INTEGER qtXLSReadRows
INTEGER (qt_K_HANDLE), INTENT(IN) :: hDS
CHARACTER (*), INTENT(IN) :: szTableName
INTEGER, INTENT(IN) :: iNoColums, iNoRows
TYPE (qT_SQLColumn), INTENT(IN) :: tColumns(iNoColums)
```

```
CHARACTER (*), INTENT(IN) :: szCondition
CHARACTER (*), INTENT(IN) :: szOrderBy
```

Die Zeilen der durch szTableName angegebenen Tabelle (null-terminierter Name) in einer durch das Handle hDS spezifizierten Excel Datei (siehe qtXLSCreateEXCELFile(...) oder qtXLSOpenEXCELFile(...)) können mithilfe von qtXLSReadRows(...) gelesen werden.
Die Anzahl der zu lesenden Zeilen wird in iNoRows vorgegeben. Sollen alle Zeilen gelesen werden, ist iNoRows = -1 zu setzen.
Welche Spalten gelesen werden wird durch eine Definition im Argument tColumns() vorgegeben.
Seine Felddimension iNoColums bestimmt die Anzahl der zu lesenden Spalten.
tColumns() enthält die Namen der zu lesenden Spalten, in welchem Feld (array) die Werte abzuspeichern sind, welchen Variablentyp dieses Feld besitzt (z.B. qt_SQL_C_DOUBLE für ein double bzw. REAL*8 Feld), seine Dimension und die Länge eines Feldelements (in Zeichen bzw. Byte).
Die Strukturdefinition, in C/C++ lautet:

```
struct qT_SQLColumn {
  char        Name[qt_I_MaxColumneNameLEN ];
  qt_K_LP     ArrayAddr;
  qt_K_INT4   ArrayDim;
  qt_K_INT4   ArrayType;
  qt_K_INT4   LENArrElem;
  qt_K_LP     IndArrAddr;
};
```

In Fortran:

```
TYPE qT_SQLColumn
  SEQUENCE
  CHARACTER (qt_I_MaxColumneNameLEN)  Name
  INTEGER (qt_K_LP)                   ArrayAddr
  INTEGER (qt_K_INT4)                 ArrayDim
  INTEGER (qt_K_INT4)                 ArrayType
  INTEGER (qt_K_INT4)                 LENArrElem
  INTEGER (qt_K_LP)                   IndArrAddr
END TYPE
```

Details zum TYPE qT_SQLColumn vermittelt das gleichnamige Kapitel (3.4.6.2).

■ Besetzung von tColumns - Spaltentyp und Feldtyp

Die Dimension (in der Komponente ArrayDim) des durch die Komponente ArrayAddr angegebenen Feldes ergibt sich aus der Anzahl der zu lesenden Zeilen (also mindestens iNoRows).
Die Auswahl des Felds, in dem die gelesenen Spaltenwerte abgelegt werden sollen, wird durch den Excel Spaltentyp bedingt vorgegeben (nicht zwingend!). Hier spielt die Zuordnung eines Excel bzw. ODBC Spaltentyps zum korrespondierenden Variablentyp des Feldes sowie dem für diesen Variablentyp zu verwendenden ArrayType eine Rolle. Nachfolgende Tabelle zeigt die Zusammenhänge .

Spaltentyp	Variablentyp des Feld	Feldtyp bzw. ArrayType
CURRENCY	struct qT_NUMERIC_STRUCT bzw. TYPE (qT_NUMERIC_STRUCT)	qt_SQL_C_NUMERIC

DATETIME	struct qT_TIMESTAMP_STRUCT bzw. TYPE (qT_TIMESTAMP_STRUCT)	qt_SQL_C_TIMESTAMP
LOGICAL	in C/C++ 1-Byte Ganzzahl bzw. in Fortran LOGICAL oder INTEGER(1)	qt_SQL_C_BIT
NUMBER	float, double, int, short oder long bzw. REAL oder INTEGER	qt_SQL_C_FLOAT qt_SQL_C_DOUBLE qt_SQL_C_SHORT qt_SQL_C_LONG
TEXT	char bzw. CHARACTER	SQL_C_CHAR

Zum Lesen einer Textspalte (Spaltentyp = TEXT) verwendet man demzufolge ein Feld vom Typ char bzw. CHARACTER und gibt in C/C++ für den ArrayType an

```
tColumns[SpaltenNr]->ArrayType = SQL_C_CHAR;
// oder auch (abhängig von Deklaration von tColumns)
tColumns[SpaltenNr].ArrayType = SQL_C_CHAR;
```

oder in Fortran:

```
tColumns(SpaltenNr) % ArrayType = SQL_C_CHAR
```

Man kann aber auch die Konvertierungsmöglichkeiten des ODBC Treibers nutzen, der mitunter in der Lage ist, einen Spaltentyp (CURRENCY, DATETIME, LOGICAL, NUMBER oder TEXT) in einen nicht gemäß obiger Tabelle korrespondierenden Feldtyp umzuwandeln. Diese Vorgehensweise empfiehlt sich insbesondere bei den Spaltentypen CURRENCY und NUMBER, an die man vorteilhafterweise ein REAL Feld in tColumns() "bindet" (Komponente ArrayType = qt_SQL_C_DOUBLE bzw. qt_SQL_C_FLOAT). Auch ein INTEGER Feld wäre möglich (Komponente ArrayType = qt_SQL_C_LONG bzw. qt_SQL_C_SHORT), so man weiß, daß die Spalte nur ganze Zahlen im zulässigen Wertebereich für INTEGER enthält. Auch die Anbindung eines CHARACTER Feldes (Komponente ArrayType = qt_SQL_C_CHAR) sollte für alle Excel bzw. ODBC Spaltentypen möglich sein (d.h. Zahlen, Datumsangaben etc. werden in Strings konvertiert und zurückgegeben).

Man vergleiche hierzu auch die Beschreibung zu qtXLSWriteRows(...) .

■ Suchbedingungen und Sortierordnung

Welche Zeilen gelesen werden sollen, kann über szCondition vorgegeben werden. Der null-terminierte String gibt einen SQL konformen Ausdruck vor der als Bedingung beim Lesen appliziert wird. Seine Form lautet demzufolge

"WHERE *Suchbedingung*"

wobei die Suchbedingung Spaltennamen und für sie geltende Bedingungen enthält. Mehrere Bedingungen sind durch SQL Operatoren wie AND oder OR zu verknüpfen. Ein Beispiel

```
szCondition = 'WHERE temperature > 200 '            &
                  'AND pressure < 10000' // CHAR(0)
```

Des weiteren kann die Sortierordnung der zurückgegebenen Werte in den durch tColumns() angegebenen Feldern angegeben werden. Im Argument szOrderBy ist dazu null-terminiert ein String anzugeben - der SQL konform - folgendes Aussehen hat:

```
"ORDER BY Spaltenname [ASC | DESC]"
```

wobei auch mehrere Spaltennamen durch Komma abgetrennt angegeben werden können. ASC bzw. DESC geben vor ob aufsteigend (ascending; Voreinstellung) oder absteigend (descending) sortiert werden soll. Beispiel in Fortran:

```
szOrderBy = 'ORDER BY measTime, measPressure DESC'   &
          // CHAR(0)
```

In C/C++:

```
szOrderBy = "ORDER BY measTime, measPressure DESC";
```

Die maximale Länge der Strings in szCondition bzw. szOrderBy sollte den Wert des PARAMETERs qt_I_MaxStatementLEN nicht überschreiten.

■ Rückgabewert der Funktion

Die Anzahl der gelesenen Zeilen wird als Funktionswert zurückgeben.

Tritt ein Fehler auf, gibt qtXLSReadRows(...) den Wert -1 zurück. Der Fehlercode bzw. seine Bedeutung kann dann über qtXLSGetErrorMessages(...) abgefragt werden.

■ C/C++ Beispiel

```
// see also demo program qtXLSDemoReadTable.cpp

#include <stdio.h>
#include <qtXLS.h>

void set_qT_SQLColumn( qT_SQLColumn *tColumn,
                       char* sName,
                       void *ArrayAddr,
                       qt_K_INT4 iArrayDim,
                       qt_K_INT4 iArrayType,
                       qt_K_INT4 iLENArrElem )
{
  // tColumn->Name = sName; // column name
  strcpy(tColumn->Name, sName); // column name
  tColumn->ArrayAddr = (qt_K_LP)ArrayAddr;  // address
  tColumn->ArrayDim  = iArrayDim;    // array dimension
  tColumn->ArrayType = iArrayType;   // type of array
  tColumn->LENArrElem = iLENArrElem;    // size
  tColumn->IndArrAddr = 0;      // reserved

  return;
}

int main(void)
{
// Arrays for data to be read.
  const qt_K_INTEGER  NoColumns = 5;
  char *szTextArr;         // for array of strings[256]
  qt_K_INTEGER *lfdNrArr; // for INTEGER*4 array
  qt_K_R8 *xArr, *yArr;   // for REAL*8 arrays
  struct qT_TIMESTAMP_STRUCT *TSArr;  // for array of
                           // date & time structures
// variables to be used by qtXLS routines
  qt_K_HANDLE hDS;
  qt_K_INT4 iError, iRet, iRow, NoRows, ind;
  char *szFileName;
  struct qT_SQLColumn *tColumns[NoColumns];
  char *szTableName;
  char *szCondition;
  char *szOrderBy;

// open EXCEL file
// ---------------
```

```
 szFileName = "qtXLSDemo4.xls";
 hDS = qtXLSOpenEXCELFile( szFileName );

// get row count of table
// ----------------------
 szTableName = "qtXLSDemoTable";
 NoRows = qtXLSGetRowCount( hDS, szTableName );

 if ( NoRows <= 0 )
 {
  printf("Table is empty. No rows to read.\n");
  goto M900;   // to the "Exit"
 }
// allocate arrays for result set
 //szTextArr[NoRows] = new char [NoRows][256];
 szTextArr = (char *) calloc(NoRows, 256);
 lfdNrArr = new qt_K_INTEGER [NoRows];
 xArr = new qt_K_R8 [NoRows];
 yArr = new qt_K_R8 [NoRows];
 TSArr = new qT_TIMESTAMP_STRUCT [NoRows];

// set up columns for import
//  "lfdNr   x   y   Description   Date_Time"
//  ----------------------------------------
 // create array of structure tColumns
 tColumns[0] = (qT_SQLColumn *)
             calloc(NoColumns, sizeof(qT_SQLColumn));
 for (ind = 1; ind < NoColumns; ind++)
  tColumns[ind] = tColumns[ind-1] + 1;
                    // means: + sizeof(qT_SQLColumn);
 set_qT_SQLColumn( tColumns[0],"lfdNr", lfdNrArr,
                   NoRows, qt_SQL_C_SLONG, 4 );
 set_qT_SQLColumn( tColumns[1], "x", xArr, NoRows,
                  qt_SQL_C_DOUBLE, 8 );
 set_qT_SQLColumn( tColumns[2], "y", yArr, NoRows,
                   qt_SQL_C_DOUBLE, 8 );
 set_qT_SQLColumn( tColumns[3], "Description",
                   szTextArr, NoRows, qt_SQL_C_CHAR,
                   256);
 set_qT_SQLColumn( tColumns[4], "Date_Time", TSArr,
                  NoRows, qt_SQL_C_TIMESTAMP, 16);

// read rows
// ---------
 // a condition & a sort order (SQL syntax)
 szCondition = "WHERE x > 0.4 AND x <= 0.5";
 szOrderBy = "ORDER BY x DESC";
            // sort order: by column "x", descending

 NoRows = qtXLSReadRows( hDS,
                         szTableName,
                         NoColumns,
                         -1,
                         tColumns[0],
                         szCondition,
                         szOrderBy );
// spec. of tColumns[0] causes qtXLSReadRows() to
// receive the starting address of array tColumns.
 if ( NoRows < 0 )
  printf("An error occurred.\n");
 else
  printf("%d rows read:\n", NoRows);

// Exit
M910:
 delete szTextArr; delete lfdNrArr; delete xArr;
 delete yArr; delete TSArr; delete *tColumns;

M900:
 iRet = qtXLSCloseEXCELFile( hDS );
```

```
  return 0;
}
```

Fortran Beispiel

```
! see also demo program qtXLSDemoReadTable.f90
USE qtXLS
! Arrays for data to be read.
INTEGER, PARAMETER ::  NoColums = 5
CHARACTER(256), ALLOCATABLE :: szTextArr(:)
INTEGER, ALLOCATABLE :: lfdNrArr(:)
REAL (qt_K_R8), ALLOCATABLE :: xArr(:), yArr(:)
TYPE (qT_TIMESTAMP_STRUCT), ALLOCATABLE :: TSArr(:)
! variables to be used by qtXLS routines
INTEGER (qt_K_HANDLE) hDS
INTEGER (qt_K_INT4) iError, iRet, iLen, iRow, NoRows
CHARACTER (20) szFileName
TYPE (qT_SQLColumn) tColumns(NoColums)
CHARACTER (qt_I_MaxTableNameLEN) szTableName
CHARACTER (qt_I_MaxStatementLEN) szCondition
CHARACTER (qt_I_MaxStatementLEN) szOrderBy

! open EXCEL file
szFileName = 'qtXLSDemo4.xls' // CHAR(0)
hDS = qtXLSOpenEXCELFile( szFileName )

! get row count of table
szTableName = 'qtXLSDemoTable' // CHAR(0)
NoRows = qtXLSGetRowCount( hDS, szTableName )

IF ( NoRows == 0 ) THEN
  PRINT*, 'Table qtXLSDemoTable is empty.'
  GOTO 900  ! to the "Exit"
END IF

! allocate arrays for result set
ALLOCATE( szTextArr(NoRows), &
          lfdNrArr(NoRows),  &
          xArr(NoRows),      &
          yArr(NoRows),      &
          TSArr(NoRows) )
! set up columns
!     "lfdNr  x  y  Description  Date_Time"
! for import
! 1st column
tColumns(1) % Name      = 'lfdNr'        ! column name
tColumns(1) % ArrayAddr = LOC(lfdNrArr)  ! array addr.
tColumns(1) % ArrayDim  = NoRows         ! array dim.
tColumns(1) % ArrayType = qt_SQL_C_SLONG ! INTEGER
tColumns(1) % LENArrElem= 4              ! elem. size
tColumns(1) % IndArrAddr = 0             ! must be 0
! and remaining columns
tColumns(2) = qT_SQLColumn('x', LOC(xArr), NoRows,   &
                           qt_SQL_C_DOUBLE, 8, 0)
tColumns(3) = qT_SQLColumn('y', LOC(yArr), NoRows,   &
                           qt_SQL_C_DOUBLE, 8, 0)
tColumns(4) = qT_SQLColumn('Description',            &
                           LOC(szTextArr), NoRows,   &
                           qt_SQL_C_CHAR,            &
                           LEN(szTextArr(1)), 0)
tColumns(5) = qT_SQLColumn('Date_Time', LOC(TSArr),  &
                           NoRows,                   &
                           qt_SQL_C_TIMESTAMP, 16, 0)
! read all rows
! spec.: condition and the sort order (SQL syntax)
szCondition = 'WHERE x > 0.4 AND x <= 0.5' // CHAR(0)
szOrderBy = 'ORDER BY x DESC' // CHAR(0)

NoRows = qtXLSReadRows( hDS, szTableName, NoColums,  &
               -1, tColumns,szCondition, szOrderBy )
```

```
IF ( NoRows < 0 ) THEN
  PRINT*, 'An error occurred.'
ELSE
  PRINT*, NoRows, ' rows read.'
END IF

! Exit
DEALLOCATE(szTextArr, lfdNrArr, xArr, yArr, TSArr)
900 CONTINUE
iRet = qtXLSCloseEXCELFile( hDS )
```

qtXLSSetErrorLevel - Setze Fehlerbehandlungsstufe

C/C++:

```
void qtXLSSetErrorLevel(
                  qt_K_INTEGER iErrorLevel  );
```

Fortran:

```
SUBROUTINE qtXLSSetErrorLevel( iErrorLevel )
INTEGER, INTENT(IN) :: iErrorLevel
```

Normalerweise bricht eine qtXLS Routine im Fehlerfall ab und kehrt zum aufrufenden Programm zurück. Man kann durch Heraufsetzen der Fehlerbehandlungsstufe (Error Level) von 0 (Voreinstellung) auf 1 dafür sorgen, daß trotz internem Fehler versucht wird, einen Prozeß weiterzuführen, sofern dies möglich ist. Dies kann bspw. beim Lesen einer Tabelle der Fall sein, wenn Daten unvollständig übertragen werden (bspw. weil ein Puffer zu klein ist). Die Routine qtXLSSetErrorLevel(...) kann jederzeit mit einem der beiden Werte (0 oder 1) gerufen werden, um die Fehlerbehandlung zu ändern.

qtXLSSetErrorMessagesDisplay - Setze Fehleranzeigemodus

C/C++:

```
void qtXLSSetErrorMessagesDisplay(
        qt_K_INTEGER iDisplayErrorMessages );
```

Fortran:

```
SUBROUTINE qtXLSSetErrorMessagesDisplay(    &
                    iDisplayErrorMessages )
INTEGER, INTENT(IN) :: iDisplayErrorMessages
```

Beim Testen von qtXLS-Applikationen kann eine stetige Fehlerprüfung mit automatischer Fehleranzeige sehr hilfreich sein. Durch

```
qtXLSSetErrorMessagesDisplay( 1 );  // in C/C++
```

bzw.

```
CALL qtXLSSetErrorMessagesDisplay( 1 )  ! Fortran
```

wird in den qtXLS Routinen dieser Fehleranzeigemodus aktiviert. D.h., falls in einer Routine ein Fehler auftritt, wird (in den meisten Fällen) eine Fehlermeldung in einem Dialogfenster angezeigt, daß der Benutzer dann

durch Drücken der 'OK'-Taste zu quittieren hat, um den Programmlauf fortzusetzen (vgl. Abb. 6).

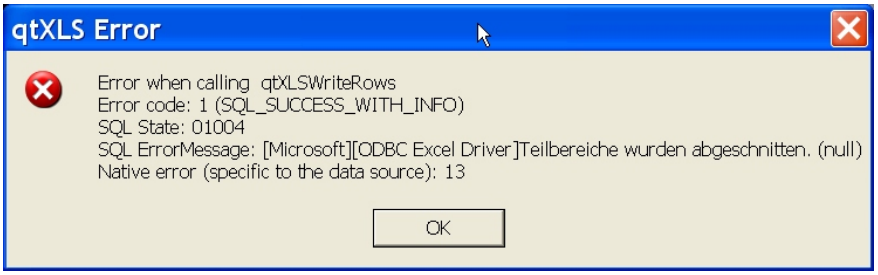

Abb. 6: Fehlermeldung, veranlaßt durch qtXLSSetErrorMessagesDisplay(1)

Dieses Verhalten ist voreinstellungsgemäß abgeschaltet oder man kann es durch

```
qtXLSSetErrorMessagesDisplay( 0 );  // in C/C++
```

bzw.

```
CALL qtXLSSetErrorMessagesDisplay( 0 )
```

ausschalten.
Die Fehlermeldungen sind identisch mit denen, die qtXLSGetErrorMessages(...) zurückliefert.

qtXLSSetLicencePath - Setze Lizenzdateipfad

C/C++:

```
void qtXLSSetLicencePath( char *szPathName );
```

Fortran:

```
SUBROUTINE qtXLSSetLicencePath( szPathName )
CHARACTER (*), INTENT(IN) :: szPathName
```

Sofern die qtXLS Lizenz nicht mittels Aufruf von

```
CALL qtSetLicence_qtXLS( iError )
```

"gesetzt" wurde, wird bei der Initialisierung von qtXLS versucht eine Lizenzdatei zu lesen, die (früher) beim Kauf einer qtXLS-Lizenz von QT software geliefert wurde. Diese Lizenzdatei wird normalerweise in dem Pfad gesucht, in dem sich die qtXLS. dll befindet. Diese Voreinstellung läßt sich durch Setzen des Lizenzdateipfads in szPathName (null-terminierter String) verändern. szPathName muß eine gültige Verzeichnisangabe enthalten (ohne den Namen der Lizenzdatei).
Die Routine qtXLSSetLicencePath(...) muß vor allen anderen qtXLS Routinen gerufen werden (besser ist aber die Routine qtSetLicence_qtXLS(...) zu verwenden - und man erspart sich die Weitergabe der Lizenzdatei).

C/C++ Beispiel

```
#include <qtXLS.h>

char* szLicPath = "C:\Program Files\Licencefiles\";
qtXLSSetLicencePath( szLicPath );
```

Fortran Beispiel

```
USE qtXLS
```

```
CALL qtXLSSetLicencePath(                              &
       'C:\Program Files\Licencefiles\' // CHAR(0) )
```

qtXLSWriteRows - Schreibe Zeilen

C/C++:

```
qt_K_INTEGER qtXLSWriteRows( qt_K_HANDLE hDS,
                             LPSTR szTableName,
                          qt_K_INTEGER iNoColums,
                            qt_K_INTEGER iNoRows,
                        qT_SQLColumn *tColumns );
```

Fortran:

```
FUNCTION qtXLSWriteRows(                    hDS, &
                                       szTableName, &
                                         iNoColums, &
                                           iNoRows, &
                                           tColumns )

INTEGER qtXLSWriteRows
INTEGER (SQLHANDLE), INTENT(IN) :: hDS
CHARACTER (*), INTENT(IN) :: szTableName
INTEGER, INTENT(IN) :: iNoColums, iNoRows
TYPE (qT_SQLColumn), INTENT(IN) :: tColumns(iNoColums)
```

Einer durch szTableName angegebenen Tabelle (Name null-terminiert) der Excel Datei, die durch das Handle hDS (siehe qtXLSCreateEXCELFile(...) bzw. qtXLSOpenEXCELFile(...)) identifiziert wird, kann mithilfe von qtXLSWriteRows(...) die im Argument iNoRows genannte Anzahl Zeilen hinzugefügt werden. Es werden dann die Spalten mit Werten besetzt, die im Feld tColumns() bezeichnet sind. Die Dimension des Felds tColumns() ist in iNoColums anzugeben.
tColumns() enthält die Namen der Spalten der Tabelle, die Angabe in welchem Feld (array) die zu exportierenden Werte zu finden sind, welchen Variablentyp dieses Feld besitzt (z.B. qt_SQL_C_DOUBLE für ein double bzw. REAL*8 Feld), seine Dimension und die Länge eines Feldelements (in Zeichen bzw. Byte). In C/C++ ist die Struktur folgendermaßen definiert:

```
struct qT_SQLColumn {
  char        Name[qt_I_MaxColumneNameLEN ];
  qt_K_LP     ArrayAddr;
  qt_K_INT4   ArrayDim;
  qt_K_INT4   ArrayType;
  qt_K_INT4   LENArrElem;
  qt_K_LP     IndArrAddr;
};
```

Und in Fortran:

```
TYPE qT_SQLColumn
  SEQUENCE
  CHARACTER (qt_I_MaxColumneNameLEN)  Name
  INTEGER (qt_K_LP)                   ArrayAddr
  INTEGER (qt_K_INT4)                 ArrayDim
  INTEGER (qt_K_INT4)                 ArrayType
  INTEGER (qt_K_INT4)                 LENArrElem
  INTEGER (qt_K_LP)                   IndArrAddr
END TYPE
```

Details zum TYPE qT_SQLColumn vermittelt das gleichnamige Kapitel (3.4.6.2).
Die Dimension (in der Komponente ArrayDim) des durch die Komponente

ArrayAddr angegebenen Feldes ergibt sich aus der Anzahl der zu schreibenden Zeilen (also mindestens iNoRows).
Die Zuordnung eines Excel bzw. ODBC Spaltentyps zum korrespondierenden Variablentyp des Feldes sowie dem für diesen Variablentyp zu verwendenden ArrayType zeigt folgende Tabelle.

Spaltentyp	Variablentyp des Feld	Feldtyp bzw. ArrayType
CURRENCY	struct qT_NUMERIC_STRUCT bzw. TYPE (qT_NUMERIC_STRUCT)	qt_SQL_C_NUMERIC
DATETIME	struct qT_TIMESTAMP_STRUCT bzw. TYPE (qT_TIMESTAMP_STRUCT)	qt_SQL_C_TIMESTAMP
LOGICAL	in C/C++ 1-Byte Ganzzahl bzw. in Fortran LOGICAL oder INTEGER(1)	qt_SQL_C_BIT
NUMBER	float, double, int, short oder long bzw. REAL oder INTEGER	qt_SQL_C_FLOAT qt_SQL_C_DOUBLE qt_SQL_C_SHORT qt_SQL_C_LONG
TEXT	char bzw. CHARACTER	SQL_C_CHAR

Zum Schreiben einer Textspalte (Spaltentyp = TEXT) verwendet man demzufolge ein Feld vom Typ char bzw. CHARACTER und gibt in C/C++ für den ArrayType an

```
tColumns[SpaltenNr]->ArrayType = SQL_C_CHAR;
// oder auch (abhängig von Deklaration von tColumns)
tColumns[SpaltenNr].ArrayType = SQL_C_CHAR;
```

oder in Fortran:

```
tColumns(SpaltenNr) % ArrayType = SQL_C_CHAR
```

Die Zuordnungen zwischen Variablentyp und Spaltentyp in obiger Tabelle sind allerdings nicht zwingend. Der Excel ODBC Treiber ist zu Konvertierungen fähig. Will man bspw. eine Spalte vom Typ CURRENCY schreiben, empfiehlt es sich statt eines Feldes vom TYPE (qT_NUMERIC_STRUCT), eines vom Typ qt_K_R8 bzw. REAL (qt_K_R8) zu verwenden (also ein 8-Byte REAL bzw. double). Beispiel in Fortran:

```
! Definition einer Spalte vom Typ CURRENCY
REAL (qt_K_R8) r8AmountUSD(100)
tColumns(1) % Name = 'AmountInUSD'
tColumns(1) % ArrayAddr = LOC(r8AmountUSD)
tColumns(1) % ArrayDim = 100
tColumns(1) % ArrayType = qt_SQL_C_DOUBLE
tColumns(1) % LENArrElem = 8  ! 8 byte REAL
tColumns(1) % IndArrAddr = 0  ! must be 0
```

Oder in C/C++

```
! Definition einer Spalte vom Typ CURRENCY
qt_K_R8 r8AmountUSD[100];
tColumns[1].Name = 'AmountInUSD';
tColumns[1].ArrayAddr = &r8AmountUSD);
tColumns[1].ArrayDim = 100;
tColumns[1].ArrayType = qt_SQL_C_DOUBLE;
tColumns[1].LENArrElem = 8;   // 8 byte double
tColumns[1].IndArrAddr = 0;   // must be 0
```

Man vergleiche hierzu auch die Beschreibung zu qtXLSReadRows(...) .

Die Zeilen werden stets nur an die Tabelle angehängt. Es ist nicht möglich eine spezielle Zeile der Tabelle anzugeben, in die man schreiben möchte. Sind mehr Spalten in der Tabelle vorhanden, als in tColumns() definiert, werden diese Spalten mit einem von Excel vorgegebenen Wert markiert, d.h. als "nicht besetzt" (das Lesen dieser Spalten liefert keinen Wert zurück).
Die Routine liefert im Erfolgsfall (iError = 0) die Anzahl der geschriebenen Zeilen zurück. Sie sollte identisch sein mit der vorgegebenen Anzahl in iNoRows.
Im Fehlerfall wird als Funktionswert -1 zurückgegeben, und iError enthält dann den Fehlercode.

■ C/C++ Beispiel

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <qtXLS.h>

const qt_K_INTEGER  DIMArr = 50, NoColumns = 4,
                    TEXTLen = 256;

void set_qT_SQLColumn( qT_SQLColumn *tColumn,
                       char* sName,
                       void *ArrayAddr,
                       qt_K_INT4 iArrayDim,
                       qt_K_INT4 iArrayType,
                       qt_K_INT4 iLENArrElem )
{
  // tColumn->Name = sName; // column name
  strcpy(tColumn->Name, sName); // column name
  tColumn->ArrayAddr = (qt_K_LP)ArrayAddr;  // address
  tColumn->ArrayDim  = iArrayDim;    // array dimension
  tColumn->ArrayType = iArrayType;   // type of array
  tColumn->LENArrElem = iLENArrElem;// size in bytes
  tColumn->IndArrAddr = 0;   // reserved, should be 0

  return;
}


int main(void)
{
// Arrays with data to be exported.
  char *szTextArr;         // [DIMArr][TEXTLen];
  qt_K_INTEGER *lfdNrArr; // long [DIMArr] arrays
  qt_K_R8 *xArr, *yArr;   // double [DIMArr] arrays
  qt_K_R8 angle;
  const qt_K_R8 PI = 3.1415932654;

// variables to be used by qtXLS routines
  qt_K_HANDLE hDS;
  qt_K_INT4 iRet, iRow, TNLen, NoRows, ind;
  char *szFileName;
  struct qT_SQLColumn *tColumns[NoColumns];
  char *szTableName;
  char *szTableDefinition;

  NoRows = DIMArr;
// allocate arrays for result set
  //szTextArr[NoRows] = new char [NoRows][TEXTLen];
  szTextArr =(char *) calloc(NoRows, TEXTLen);
  lfdNrArr = new qt_K_INTEGER [NoRows];
  xArr = new qt_K_R8 [NoRows];
  yArr = new qt_K_R8 [NoRows];
```

```
// Fill arrays with values (the data for export)
 for (ind = 0; ind < DIMArr; ind++)
 {
  iRow = ind + 1;
  lfdNrArr[ind] = iRow;
  xArr[ind] = iRow * 0.01;
  angle = xArr[ind] * PI;
  yArr[ind] = cos(angle);
  sprintf( szTextArr + ind * TEXTLen,
           "(Angle = , %.2f,  (degree)",
           angle * 180. / PI );
 }

// create "empty" EXCEL file
 szFileName = "qtXLSDemo3.xls";
 hDS = qtXLSCreateEXCELFile( szFileName );

// continue, if an error occurs (if possible)
 qtXLSSetErrorLevel( 1 );

// Create (empty) table
// --------------------
 szTableName = "qtXLSDemoTable";
 TNLen = strlen( szTableName );

 /*
    create table by setting up a command line
    containing the table name followed by a list of
    pairs of column names and column types (like
    NUMBER, DATETIME, TEXT, CURRENCY or LOGICAL).
 */
 szTableDefinition = new char [1000];
 strcpy(szTableDefinition, szTableName);
 strcpy(&szTableDefinition[TNLen],
    " (lfdNr NUMBER, x NUMBER, y NUMBER,
       Description TEXT, Date_Time DATETIME)");
 iRet = qtXLSCreateTable( hDS, szTableDefinition  );
  if ( iRet != 0) return -1;  // stop on error

// Set up columns
//  "lfdNr   x   y   Description   Date_Time"
// for export
// ----------------------------------------
// create array of structure tColumns
 tColumns[0] = (qT_SQLColumn *) calloc(NoColumns,
                                sizeof(qT_SQLColumn));
 for (ind = 1; ind < NoColumns; ind++)
  tColumns[ind] = tColumns[ind-1] + 1; /* means:
                           + sizeof(qT_SQLColumn) */

 set_qT_SQLColumn( tColumns[0], "lfdNr", lfdNrArr,
                   NoRows,qt_SQL_C_SLONG,4 );
 set_qT_SQLColumn( tColumns[1], "x", xArr, NoRows,
                   qt_SQL_C_DOUBLE, 8);
 set_qT_SQLColumn( tColumns[2], "y", yArr, NoRows,
                   qt_SQL_C_DOUBLE, 8);
 set_qT_SQLColumn( tColumns[3], "Description",
                   szTextArr, NoRows, qt_SQL_C_CHAR,
                   TEXTLen );

// Fill table with rows
// --------------------
 iRet = qtXLSWriteRows( hDS, szTableName, NoColumns,
                        NoRows, tColumns[0] );
 if ( iRet >= 0 )
  printf("Number of rows written: %d\n", iRet);
 else
  printf("Error; iError = %d\n", iRet);

 iRet = qtXLSCloseEXCELFile( hDS );
```

```
 return 0;
}
```

Fortran Beispiel

```
! see also demo program qtXLSDemoWriteTable.f90
USE qtXLS
IMPLICIT NONE

INTEGER, PARAMETER :: DIMArr = 50, NoColums = 4
CHARACTER(256) szTextArr(DIMArr)
INTEGER lfdNrArr(DIMArr)         ! INTEGER*4
REAL (qt_K_R8) xArr(DIMArr), yArr(DIMArr)   ! REAL*8

REAL (qt_K_R8) angle
REAL (qt_K_R8), PARAMETER :: PI = 3.1415932654D0

INTEGER (qt_K_HANDLE) hDS
INTEGER (qt_K_INT4) iRet, iRow, TNLen,NoRows
CHARACTER (20) szFileName
TYPE (qT_SQLColumn) tColumns(NoColums)
CHARACTER (qt_I_MaxTableNameLEN) szTableName
CHARACTER (1000) szTableDefinition

! Fill arrays with values (the data we're going to
export into an EXCEL file)
DO iRow = 1, DIMArr
 lfdNrArr(iRow) = iRow
 xArr(iRow) = iRow * 0.01
 angle = xArr(iRow) * PI
 yArr(iRow) = COS(angle)
 WRITE(szTextArr(iRow),"('Angle = ', F0.2,          &
        ' (degree)', A1)") angle * 180. / PI, CHAR(0)
END DO

! create "empty" EXCEL file
szFileName = 'qtXLSDemo3.xls' // CHAR(0)
hDS = qtXLSCreateEXCELFile( szFileName )

! Create (empty) table
szTableName = 'qtXLSDemoTable' // CHAR(0)
TNLen = qtXLSGetszStringLength( szTableName )

! create table by setting up a command line containing
! the table name followed by a list of pairs of column
! names and column types (like NUMBER, DATETIME, TEXT,
! CURRENCY or LOGICAL).
szTableDefinition = szTableName(1:TNLen)            &
          // ' (lfdNr NUMBER, x NUMBER, y NUMBER,'   &
          // ' Description TEXT)' // CHAR(0)
iRet = qtXLSCreateTable( hDS, szTableDefinition  )

! Set up columns
!    "lfdNr   x   y   Description"
! for export
! 1st column:
tColumns(1) % Name      = 'lfdNr'       ! column name
tColumns(1) % ArrayAddr = LOC(lfdNrArr) ! arr.address
tColumns(1) % ArrayDim  = DIMArr         ! array dim.
tColumns(1) % ArrayType = qt_SQL_C_SLONG ! INTEGER
tColumns(1) % LENArrElem= 4              ! elem. size
tColumns(1) % IndArrAddr= 0              ! must be 0
! and remaining columns (using the TYPE constructor
! function qT_SQLColumn)
tColumns(2) = qT_SQLColumn('x', LOC(xArr), DIMArr,   &
                           qt_SQL_C_DOUBLE, 8, 0)
tColumns(3) = qT_SQLColumn('y', LOC(yArr), DIMArr, &
                           qt_SQL_C_DOUBLE, 8, 0)
tColumns(4) = qT_SQLColumn('Description',           &
                           LOC(szTextArr), DIMArr, &
```

```
                                qt_SQL_C_CHAR,  &
                                LEN(szTextArr(1)), 0)
NoRows = DIMArr
! now, write rows
iRet = qtXLSWriteRows( hDS, szTableName, NoColums,   &
                                      NoRows, tColumns )
IF ( iRet >= 0 ) THEN
  PRINT*, 'Number of rows written: ', iRet
ELSE
  PRINT*, 'Error; iError = ', iRet
END IF

iRet = qtXLSCloseEXCELFile( hDS )
```

4. Kompilieren & Binden (compile & link)

4.1 Allgemeine Hinweise

Programme (.exe), die auf qtXLS basieren, benötigen zur Laufzeit die Dynamic-Link-Library qtXLS.dll Darüber hinaus war für die unbeschränkte Nutzung der qtXLS-Funktionen in früheren qtXLS Versionen eine Lizenzdatei notwendig (Form L####-######.lic), sonst konnte die qtXLS.dll nur im Demonstrationsmodus benutzt werden. Auch die aktuelle qtXLS Version unterstützt diese Eigenschaft noch. Besser ist es jedoch die Routine qtSetLicence_QTXLS(...) in Ihrem Programm aufzurufen und so die Nutzung von qtXLS in Ihrem Programm zu autorisieren.
Da die qtXLS Routinen Funktionen des Microsoft Excel ODBC Treiber verwenden, muß ein solcher installiert sein (vgl. Kapitel "Einführung").
Um qtXLS basierende Programme (.exe) zu entwickeln, sind für diverse Compiler verschiedene Bindings erhältlich, die die zum Teil notwendige Import-Library für die qtXLS.dll beinhalten sowie weitere Dateien - wie bspw. prä-komplilierte Fortran 90 MODULE-Dateien (enden auf .mod) oder Headerfiles (.h). Des weiteren sind in diesen Bindings Dateien vorhanden, die dem "Zusammenbau" der mitgelieferten qtXLS-Demoprogramme dienen.
Im den nachfolgenden Abschnitten werden die verschiedenen Bindings vorgestellt und die zum Kompilieren und Binden mit den jeweiligen Compilersystemen notwendigen Befehle beschrieben.
Grundsätzlich gilt für alle Compiler-Einstellungen, daß 4-Byte lange INTEGER verwandt werden (dies ist normalerweise auch die Voreinstellung).

Mit Absoft ProFortran for Windows

Das Binding zur Nutzung mit Absoft Pro Fortran for Windows (v10.0 und kompatible) befindet sich im Verzeichnis

qtXLS\Bindings\ProFortran

der Installation. Es besteht aus den Dateien

```
qtXLS_ProF10.lib
qtXLS_IMP.lib
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
BuildDemosWithProF10.bat
clProF10.bat
```

Compile & Link

Zum Übersetzen eines qtXLS basierenden Programms benötigt man die MODULE Dateien,

```
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.mod
```

die in einem beliebigen Verzeichnis abgelegt sein können, auf das der Compiler Zugriff hat. Ggf. ist über die Compiler-Option -p <pathname>

der Pfad anzugeben, wo sich diese .mod Dateien befinden.
In der Entwicklungsumgebung "Developer Tools Interface" gibt man den Pfad im Dialog "Module File Path(s)" an (vgl. nachfolgende Abb.). Um zu diesem zu gelangen, ist in den “Project Options”, Registerkarte “F95” die Taste “Set Module Path(s)...” zu drücken.

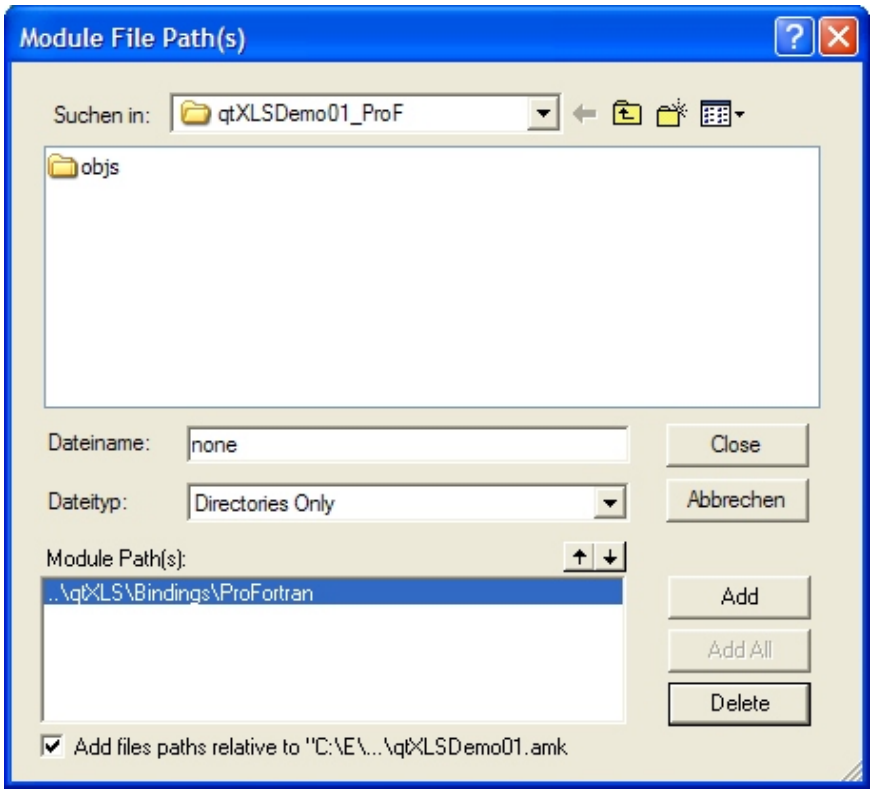

Abb.7: Dialog "Module File Path(s)" mit Angabe des MODULE Pfads

Zum Binden (Link) müssen die Libraries

```
qtXLS_ProF10.lib
qtXLS_IMP.lib
```

benannt werden. In der Entwicklungsumgebung von Absoft ProFortran sind diese dem Projekt hinzuzufügen (vgl. nebenstehende Abb.).

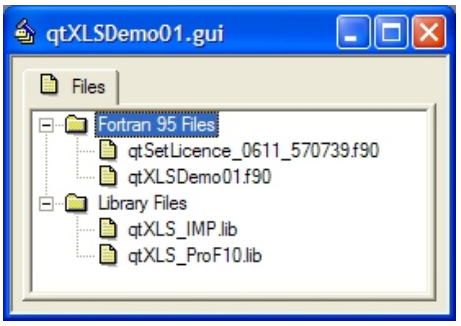

Abb.8: Dateien eines qtXLS-Projekts

Vorausgesetzt alle o.g. Dateien und die Quellecodedatei befinden sich im gleichen Verzeichnis, ist der Absoft Compiler F95 wie folgt aufzurufen (hier für Konsolenapplikation):

```
F95 <Dateiname>.f90 qtSetLicence_0611_######.obj
qtXLS_ProF10.lib qtXLS_IMP.lib -out:<Dateiname>.exe
```

Das Binding für den F95 enthält ein Stapeldatei

```
clProF10.bat
```

die vom Programmierer kopiert und seinen eigenen Bedürfnissen angepaßt werden mag. Sie wird auch von einer weiteren Stapeldatei, nämlich

```
BuildDemosWithProF10.bat
```

verwandt, um die qtXLS-Demoprogramme zu erstellen (sollte hier ein "Serialization Error" auftreten, sind die Befehle der Stapeldatei einzeln in der "DOS Box" aufzurufen).

Mit Compaq Visual Fortran

Das Binding zur Nutzung mit Compaq's Visual Fortran (kurz CVF) v6.6 befindet sich im Verzeichnis

qtXLS\Bindings\CVF

der Installation. Für die Version v6.1 des CVF ist es

im Verzeichnis

qtXLS\Bindings\CVF61

abgelegt. Beide Bindings bestehen aus den Dateien

```
qtXLS_CVF.lib
qtXLS_CVF.exp
QTFOREXCELMODULE.MOD
QTXLS.MOD
QTCOMPILERMODULE_QTXLS.MOD ! in CVF 6.1 Version nicht
enthalten
QTXLSDECLARATIONS.MOD
qtCompilerModule_QTXLS_VF.obj
```

sowie einem Unterverzeichnis

qtXLS\Bindings\CVF\BuildDemosWithCVF

bzw.

qtXLS\Bindings\CVF61\BuildDemosWithCVF

in dem sich jeweils ein Workspace Datei

```
BuildDemosWithCVF.dsw
```

die durch Aufruf ("Doppelklick im Windows Explorer") in die Entwicklungsumgebung von CVF geladen werden kann. Der Workspace enthält 4 Projekte (gespeichert in .dsp Dateien)

```
qtXLSDemoCreateTable
qtXLSDemoListTablenames
qtXLSDemoReadTable
qtXLSDemoWriteTable
```

die zum Erstellen der qtXLS-Demoprogramme dienen (vgl. nebenstehende Abb.).

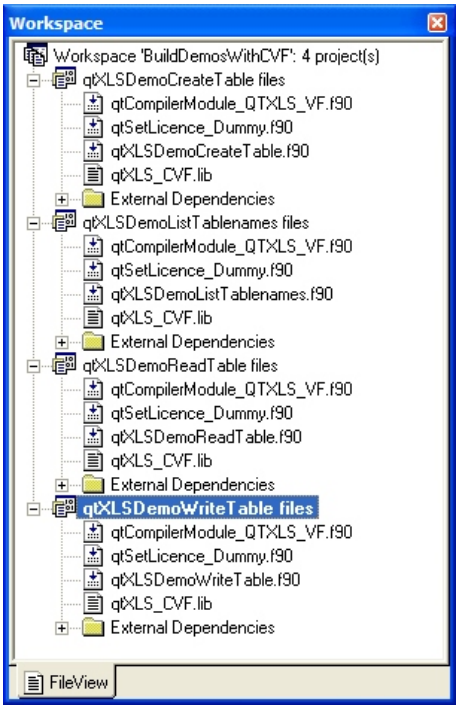

Abb. 9: CVF Workspace "BuildDemosWithCVF"

Compile & Link

Des weiteren demonstrieren diese Projekte, wie die qtXLS Library und die MODULE Dateien einzubinden sind.
Dem Compiler ist in der Registerkarte "Fortran" (in den "Project Settings"), Kategorie "Preprocessor", im Eingabefeld "INCLUDE and USE Paths" mitzuteilen, wo sich die MODULE Dateien

```
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTFOREXCELMODULE.MOD
QTCOMPILERMODULE_QTXLS.MOD
```

befinden (vgl. Abb. 10).

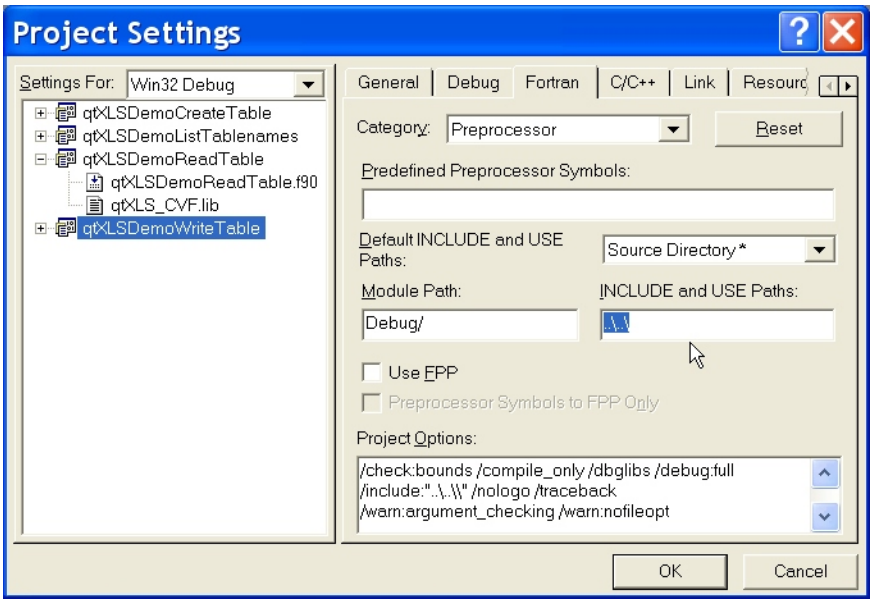

Abb. 10: Project Settings mit Eingabefeld "INCLUDE and USE Paths"

Der Linker muß wissen, wo er nach der Import-Library

```
qtXLS_CVF.lib
```

suchen muß. Hier genügt es die Library den Projektdateien hinzuzufügen (vgl. obige Abb. 9). Ein simpler Klick auf die "Build"-Taste (oder Aufruf über das Menü "Build | Build ..exe.") sorgt dann für alles weitere.

Mit Intel Visual Fortran

Das Binding zur Nutzung mit Intel Visual Fortran (kurz IVF) befindet sich im Verzeichnis

qtXLS\Bindings\IVF

der Installation. Es enthält die Dateien

```
qtXLS_IVF.lib
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
```

sowie in einem Unterverzeichnis

qtXLS\Bindings\IVF\BuildDemosWithIVF

die "Projektmappe"

```
BuildDemosWithIVF.sln
```

die durch Aufruf ("Doppelklick im Windows Explorer") in die Entwicklungsumgebung Visual Studio geladen werden kann. Die Projektmappe (Solution) enthält 4 Projekte (gespeichert in .vfproj Dateien)

```
qtXLSDemoCreateTable
qtXLSDemoListTablenames
qtXLSDemoReadTable
qtXLSDemoWriteTable
```

die zum Erstellen der qtXLS-Demoprogramme dienen.

Compile & Link

Des weiteren demonstrieren diese Projekte, wie die qtXLS Library und die MODULE Dateien einzubinden sind.
Dem Compiler ist in der Registerkarte “Fortran” (in den “Project Properties”), Kategorie “Preprocessor”, im Eingabefeld “Additional Include Directories” mitzuteilen, wo sich die MODULE Dateien

```
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
```

befinden.

Dadurch, daß die Import-Library qtXLS_IVF.lib als Projektdatei angegeben ist, wird sie vom Linker hinzugebunden.

Um die Demoprogramme zu erzeugen, genügt ein simpler Klick auf die “Build Solution”-Taste, oder der Aufruf über das Menü “Build | Build Solution” sorgt dann für alles weitere.

Debug & Testen

Um die Demoprogramme innerhalb von Visual Studio auszuprobieren oder den Ablauf im Debugger verfolgen zu können, muß man noch in der Entwicklungsumgebung das Arbeitsverzeichnis für den Debugger korrekt setzen (vgl. nachfolgende Abbildung).

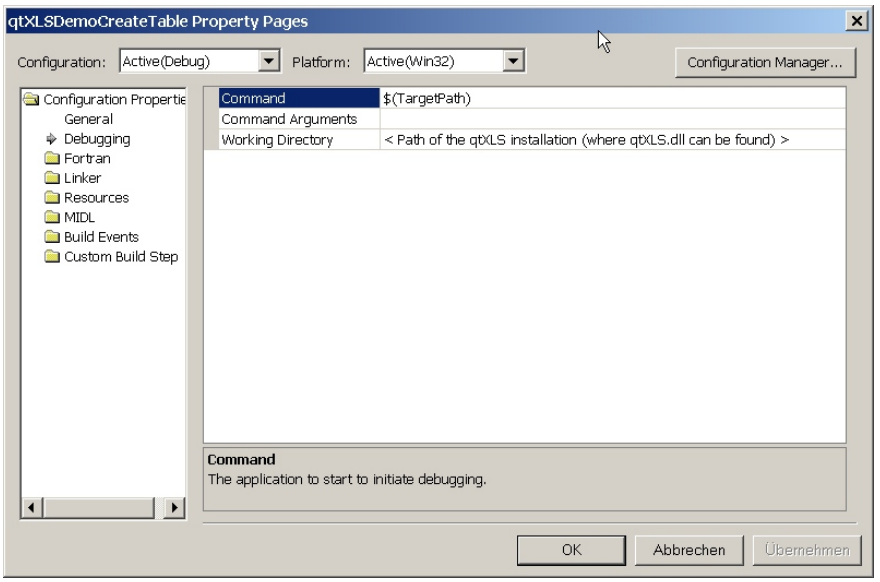

Abb. 11: Project Properties mit Eingabefeld “Working Directory”

Die Demoprogramme benötigen die qtXLS.dll, die sich im Installationsverzeichnis von qtXLS befindet. Daher muß dessen Verzeichnispfad unter “Working Directory” eingetragen werden. Leider scheint Visual Studio (VS 2003 & 2005) hier keine relative Pfadangabe zu erlauben (..\..\..\ würde genügen).

Mit Lahey/Fujitsu Fortran for Windows (LF95 v5.7)

Das Binding zur Nutzung mit Lahey/Fujitsu LF95 v5.7 (und ggf. höher) befindet sich im Verzeichnis

qtXLS\Bindings\LF9557

der Installation. Es besteht aus den Dateien

```
qtXLS_LF9557.lib
QTXLS.MOD
QTXLSDECLARATIONS.MOD
BuildDemosWithLF9557.bat
clLF9557.bat
```

Compile & Link

Zum Übersetzen eines qtXLS basierenden Programms benötigt man die MODULE Dateien,

```
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
```

die in einem beliebigen Verzeichnis abgelegt sein können, auf das der Compiler Zugriff hat. Ggf. ist über die Compiler-Option -MOD <pathname> der Pfad anzugeben, wo sich diese .mod Dateien befinden.

Zum Binden (Link) muß die Library

```
qtXLS_LF9557.lib
```

angegeben werden.

Vorausgesetzt alle o.g. Dateien und die Quellecodedatei sind im gleichen Verzeichnis, ist der LF95 wie folgt aufzurufen:

```
LF95 <Dateiname>.f90 qtSetLicence_0611_######.f90
qtXLS_LF9557.lib
```

(####_###### ist durch die Lizenznummer zu ersetzen). Das Binding für den LF95 enthält ein Stapeldatei

```
clLF9557.bat
```

die vom Programmierer kopiert und seinen eigenen Bedürfnissen angepaßt werden mag. Sie wird auch von einer weiteren Stapeldatei, nämlich

```
BuildDemosWithLF9557.bat
```

verwandt, um die qtXLS-Demoprogramme zu erstellen.

Mit Microsoft Visual C++

Das Binding zur Nutzung mit Microsoft Visual C++ v6 oder v7 (kurz VC) befindet sich im Verzeichnis

qtXLS\Bindings\VC

der Installation. Es besteht aus den Dateien

```
qtXLS_IMP.lib
qtXLS_IMP.exp
qtXLS.h
```

sowie einem Unterverzeichnis

qtXLS\Bindings\VC\BuildDemosWithVC

in dem sich eine Workspace Datei (VC v6)

```
BuildDemosWithVC.dsw
```

bzw. eine Solution Datei (VC v7)

```
BuildDemosWithVC.sln
```

befindet, die durch Aufruf ("Doppelklick im Windows Explorer") in die Entwicklungsumgebung von VC geladen werden kann. Der Workspace bzw. die Projektmappe (Solution) enthält 4 Projekte (gespeichert in .dsp bzw. .vcproj Dateien)

```
qtXLSDemoCreateTable
qtXLSDemoListTablenames
qtXLSDemoReadTable
qtXLSDemoWriteTable
```

die zum Erstellen der qtXLS-Demoprogramme dienen.

Compile & Link

Des weiteren demonstrieren diese Projekte, wie die qtXLS Library einzubinden ist und der INCLUDE Pfad für die Header-Datei qtXLS.h zu setzen ist. Dem Compiler ist hierzu in der Registerkarte "C/C++" (in den "Project Settings"), Kategorie "Preprocessor", im Eingabefeld "Additional Include directories" mitzuteilen, wo sich die Header-Datei qtXLS.h befindet.

Der Linker muß wissen, wo er nach der Import-Library

```
qtXLS_IMP.lib
```

suchen muß. Hier genügt es die Library den Projektdateien hinzuzufügen. Ein simpler Klick auf die "Build"-Taste (oder Aufruf über das Menü "Build | Build ..exe.") sorgt dann für alles weitere. Vgl. auch die Beschreibung zu Compaq Visual Fortran bzw. Intel Visual Fortran (die Compiler verwenden die gleiche Entwicklungsumgebung wie VC v6 bzw v7).

Mit Salford bzw. Silverfrost FTN95 (Win32)

Das Binding zur Nutzung mit Salford's FTN95 befindet sich im Verzeichnis

qtXLS\Bindings\FTN95

der Installation. Es besteht aus den Dateien

```
qtXLS_FTN95.lib
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
BuildDemosWithFTN95.bat
clFTN95.bat
```

Compile & Link

Zum Übersetzen eines qtXLS basierenden Programms benötigt man die MODULE Dateien,

```
QTXLS.MOD
QTXLSDECLARATIONS.MOD
QTCOMPILERMODULE_QTXLS.MOD
```

die in einem beliebigen Verzeichnis abgelegt sein können, auf das der Compiler Zugriff hat. Ggf. ist über die Compiler-Option /MOD_PATH <pathname> der Pfad anzugeben, wo sich diese .mod Dateien befinden.

In der Entwicklungsumgebung des FTN95 (Plato) gibt man den MODULE Pfad im "Properties"-Dialog an (vgl. nachfolgende Abbildung).

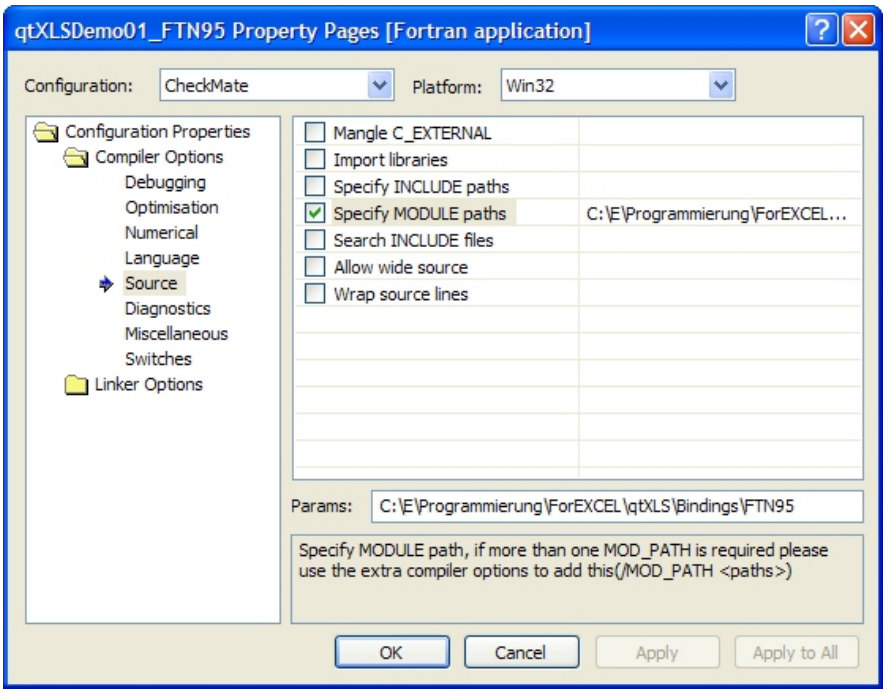

Abb. 13: MODULE Pfadangabe in Plato3 (Dialog "Properties")

Zum Binden (Link) muß sowohl die DLL

`qtXLS.dll`

als auch die Library

`qtXLS_FTN95.lib`

angegeben werden. In der Entwicklungsumgebung des FTN95 (Plato) werden die beiden Libraries unter References angegeben (vgl. nachfolgende Abbildung).

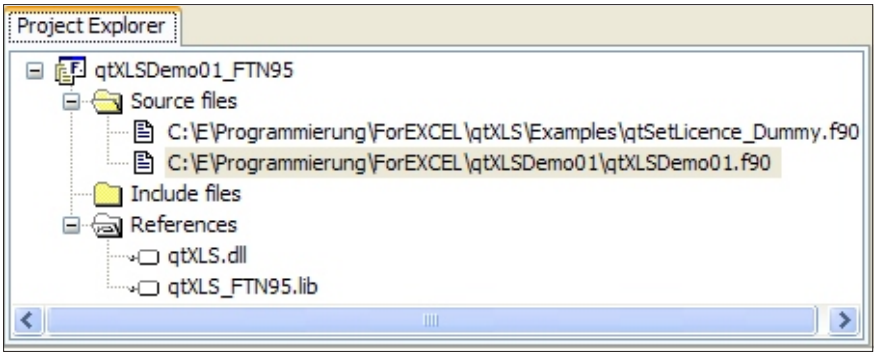

Abb. 12: Dateien im Project Explorer von Plato3

Vorausgesetzt alle o.g. Dateien und die Quellecodedatei sind im gleichen Verzeichnis, ist der FTN95 wie folgt aufzurufen:

```
FTN95 <Dateiname>.f90 qtSetLicence_####_######.f90
/LIBRARY qtXLS.dll /LIBRARY qtXLS_FTN95.lib /link
```

(`####_######` ist durch die Lizenznummer zu ersetzen). Das Binding für den FTN95 enthält ein Stapeldatei

`clFTN95.bat`

die vom Programmierer kopiert und seinen eigenen Bedürfnissen angepaßt werden mag. Sie wird auch von einer weiteren Stapeldatei, nämlich

`BuildDemosWithFTN95.bat`

verwandt, um die qtXLS-Demoprogramme zu erstellen. Unter Plato genügt ein Klick auf den Build-Button.

5. Inhalt und Aufbau der qtXLS Installation

qtXLS steht als komprimierte Datei bereit (im ZIP Format):

```
qtXLS.zip
```

Beim Entpacken in einem vom Programmierer beliebig auszuwählenden Verzeichnis auf der Festplatte ergibt sich eine Verzeichnisstruktur, ähnlich der nachfolgenden Abbildung.

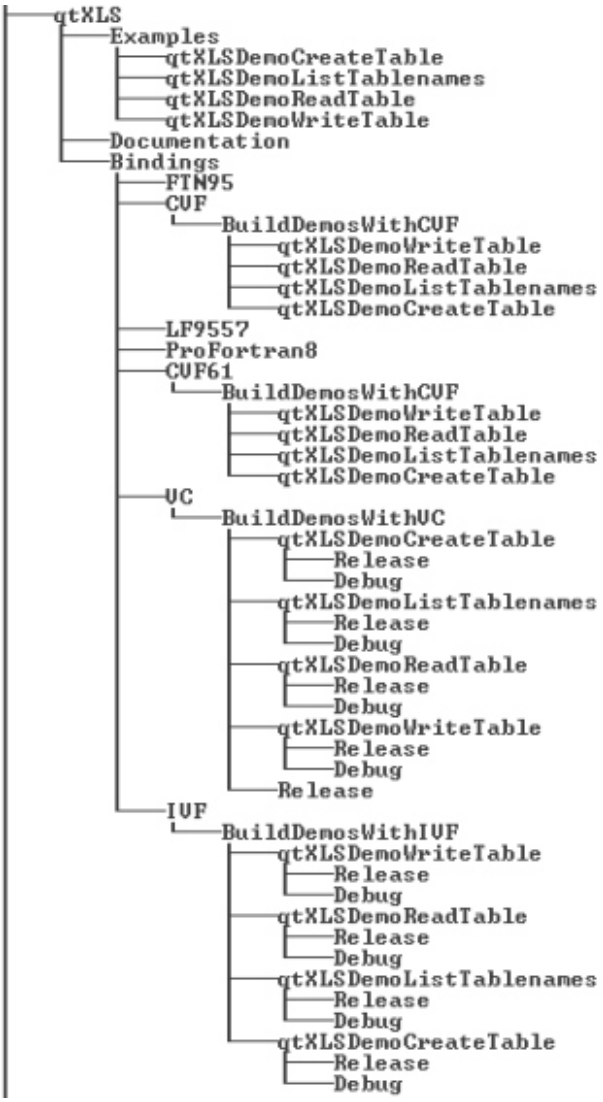

Abb. 14: Verzeichnisstruktur der qtXLS Installation

Im "Wurzelverzeichnis" der Installation befindet sich das Verzeichnis "qtXLS", in dem die

```
qtXLS.dll
```

liegt sowie die Excel Beispieldateien:

```
qtXLSDemo1.xls
qtXLSDemo2.xls
qtXLSDemo3.xls
qtXLSDemo4.xls
```

Im Verzeichnis "Bindings" sind die Libraries, die .h- und .mod-Dateien für die unterstützten Compiler untergebracht, sowie Stapeldateien (.bat) und andere compiler-spezifische "Make"-Dateien, die der Erzeugung der Beispielprogramme dienen, die im Verzeichnis "Examples" abgelegt sind.

6. Weitergabe von qtXLS-Applikationen

Programme (.exe), die auf qtXLS Funktionen zugreifen, benötigen für die uneingeschränkte Funktionalität die Datei

```
qtXLS.dll
```

die rechtmäßige Lizenznehmer (siehe Kapitel “Nutzungsbedingungen”) zusammen mit ihren qtXLS-Applikationen weitergeben dürfen. Wird in der Applikation die Nutzung nicht mittels der Routine qtSetLicence_qtXLS(...) autorisiert, ist noch die

```
Lizenzdatei (Form L####-######.lic mit # = 0 bis 9)
```

mit der qtXLS basierenden Applikation weiterzugeben (die Lizenzbedingungen erlauben dies). Die Weitergabe aller anderen Dateien der Software qtXLS ist nicht gestattet.

7. Systemvoraussetzungen

Um die qtXLS Software nutzen zu können, werden benötigt:

- PC mit Pentium Prozessor, Festplatte mit mindestens 15 MB freiem Speicherplatz, mindestens 32 MB RAM.
- Unterstützte Betriebssysteme: Microsoft Windows 98, Windows NT 4.0, Windows 2000, Windows XP und kompatible (hier sei darauf hingewiesen, daß diverse Microsoft Excel ODBC Treiber nur unter bestimmten Windows Betriebssystemen lauffähig sind und manche dieser Betriebssysteme von einigen Compilern nicht mehr unterstützt werden, so daß dann auch qtXLS für diese nicht mehr verfügbar ist).
- Microsoft Excel ODBC Treiber (vgl. Kapitel “Einführung”)
- Compilersystem: Fortran 90/95 oder C/C++ Compiler und Linker, wie im Kapitel “Kompilieren & Binden” aufgeführt, sowie kompatible.

8. Nutzungsbedingungen

§1. Eigentum und Eigentümer

Die in diesem Dokument beschriebenen Software, im nachfolgenden qtXLS Software genannt, besteht im wesentlichen aus der Datei qtXLS.dll, den im Kapitel “Kompilieren & Binden” aufgeführten compiler-spezifischen Libraries (Dateinamen enden auf .lib), prä-kompilierten MODULE Dateien (enden auf .mod), dieser qtXLS Dokumentation, den Quellcode-Dateien qtXLS.h und qtXLSDeclarations.f90 und weiteren in diesem Dokument genannte Dateien, deren Name mit den Buchstaben “qt” beginnt. Alle diese Dateien sind Eigentum des Autors Jörg Kuthe. Der Autor wird durch die QT software GmbH, Berlin - nachfolgend QT genannt - vertreten, die berechtigt ist, Nutzungsberechtigungen für qtXLS zu vergeben.
Das Urheberrecht an der qtXLS Software und dieser Dokumentation verbleiben beim Autor.

§2. Lizenznehmer und Lizenzdatei bzw. Lizenzroutine

Der Lizenznehmer ist der Erwerber der qtXLS Software sowie einer Lizenzdatei oder alternativ einer Lizenzroutine, die er von QT im Zuge des Kaufs der Nutzungslizenz erhalten und für die er den vereinbarten Kaufpreis vollständig entrichtet hat. Die Lizenzdatei (Form L####-######.lic) und die Lizenzroutine qtSetLicence_####-######.f90 (mit # = Ziffer 0 bis 9) enthalten identifizierende Daten des Lizenznehmers.

§3. Nutzungslizenz und Weitergabe von Bestandteilen der Software qtXLS

Die Nutzungslizenz besteht aus dem Recht des Lizenznehmers, die Software qtXLS zur Entwicklung von ausführbaren Programmen zu verwenden, d.h. damit ausführbare Dateien zu erstellen, deren Name auf .exe endet.
Die mit qtXLS erstellte ausführbare Datei (.exe) darf weder die Funktion eines Übersetzers (Compiler) noch die eines Programms zum Binden von kompilierten Dateien (Linker) enthalten.
Die Lizenz umfaßt außerdem die Berechtigung zur Weitergabe der qtXLS.dll sowie der Lizenzdatei (sofern bereitgestellt) des Lizenznehmers zusammen mit Programmen (.exe Dateien), die auf qtXLS Routinen aufrufen.
Die Weitergabe aller anderen Dateien der Software qtXLS ist nicht gestattet.
Der Lizenznehmer hat bei Weitergabe seiner auf qtXLS basierenden Programme den bzw. die Nutzer dieser Programme auf den Eigentumsrechte an der qtXLS.dll durch folgenden Text hinzuweisen:
"Die Datei qtXLS.dll ist Eigentum von Jörg Kuthe, vertreten durch die QT software GmbH, Deutschland. Die Nutzung der qtXLS.dll ist nur zusammen mit dem vom <qtXLS Lizenznehmer> gelieferten Programm <Programmname des qtXLS Lizenznehmers>.exe gestattet." Die in spitzen Klammern gefaßten Texte sind durch den Namen des Lizenznehmers bzw. des Programms zu ersetzen.

§4. Übertragung der Nutzungslizenz

Der Lizenznehmer kann nicht durch eine andere Person vertreten werden. Dies schließt insbesondere den Verleih der qtXLS Software aus.
Der Lizenznehmer darf die Nutzungslizenz veräußern, wenn er die Veräußerung bzw. Übertragung der Lizenz auf einen anderen Lizenznehmer schriftlich QT anzeigt. Die Übertragung muß die Bestätigung enthalten, daß der veräußernde Lizenznehmer seine Nutzungsrechte an der qtXLS Software aufgibt.
Der neue Lizenznehmer muß diesen Lizenzbedingungen schriftlich zustimmen und QT die zur Erstellung einer neuen Lizenzdatei notwendigen Daten liefern.

§5. Garantie

QT gewährleistet die Funktionsfähigkeit der Software qtXLS für den Zeitraum von 2 Jahren nach Erwerb der Nutzungslizenz. Im Fehlerfall steht es QT frei, entweder den entrichteten Kaufpreis an den Lizenznehmer zurückzuerstatten oder den beanstandeten Fehler nachzubessern.
Wird der Kaufpreis zurückerstattet, verliert der Lizenznehmer die Berechtigung die Software qtXLS weiter zu benutzen.
Beanstandungen bzw. Fehler sind durch ein Beispielprogramm durch den Lizenznehmer zu belegen.

§6. Nutzungsrisiko und Haftungsbeschränkungen

Der Lizenznehmer nutzt die Software qtXLS auf eigenes Risiko. QT haftet in maximaler Höhe des entrichteten Kaufpreises.

§7. Einverständniserklärung

Der Lizenznehmer erklärt durch den Erwerb der Nutzungslizenz sein Einverständnis mit diesen Nutzungsbedingungen.

9. Sonstige Hinweise

Der Autor und QT software GmbH erkennen die Rechte der Inhaber an den in diesem Dokument namentlich aufgeführten Markennamen, Warenzeichen und Produktnamen ohne Einschränkung an:
Excel ist ein Warenzeichen der Microsoft Corporation, U.S.A..
Windows ist ein Warenzeichen der Microsoft Corporation, U.S.A..
"ProFortran for Windows" ist ein Produkt der Absoft Corporation, U.S.A..
"Compaq Visual Fortran" ist ein Produkt der Hewlett-Packard Company, U.S.A..
"Intel Visual Fortran" ist ein Produkt der Intel Corporation, U.S.A..
"Lahey/Fujitsu Fortran 95 for Windows" ist ein Produkt der Firma Lahey Computer Systems, Inc., U.S.A..
"Salford FTN95" ist ein Produkt der Salford Software Ltd., U.K..
"Silverfrost FTN95" ist ein Produkt der Silverfrost Ltd., U.K..
"Excel", "Visual C++", "Visual Studio" und "Visual Basic" sind Produkte der Microsoft Corporation, U.S.A..

Index

M

N

O

P

Q

S

T

U

V

W

X

Z