Jörg Kuthe

qtXLS

Instructions for the Use of the qtXLS Software

Revision date: 18th of 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

Contents

1. Introduction

The qtXLS Library offers the programmer routines for reading and writing Microsoft Excel formatted files. Their names usually end on .xls. qtXLS is based on the ODBC drivers provided by Microsoft which are usually set up to a PC automatically at the installation from Excel under Windows (cf. illus.1). The existence of the Microsoft Excel ODBC drivers is one of the prerequisites for the functioning of the qtXLS routines. If these drivers aren't existing on a PC, then they can be provided by either installation of Microsoft Excel or the **Microsoft data Access Components** (**MDAC**). The latter should be the more economical alternative since they can be loaded by Microsoft's web site free of charge. One finds them the fastest with the help of the search function in the "download center".

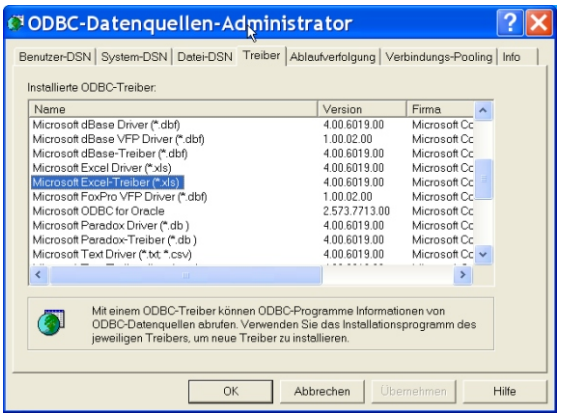

Fig. 1: ODBC Microsoft Excel Driver

⇨ http://www.microsoft.com/downloads

At the time of the writing of these operating instructions the MDAC could be found inside the download category "Drivers". Provided that there aren't license legal objections on the part of the manufacturer, you will find the reference to the drivers also on the web page of *QT software*:

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

Since ODBC is the basis of qtXLS, qtXLS communicates with the Excel ODBC driver about the ODBC functions integrated in Windows and there with the help of the Structured Query Language (SQL). This is relevant for the use of some of the qtXLS routines since they use the facilities of SQL.

1.1 Functions and Restrictions

With qtXLS routines you can

- create files in the Excel file format,
- create tables within these files,
- write data to tables,
- read data from tables and
- obtain information about tables and columns.

Since qtXLS is based on the ODBC drivers of Microsoft, qtXLS is also affected by their restrictions. The essential limitations are:

- For reading Excel tables the **names of the columns must be in the first row of the table** (cf. illus.2). When laying out tables by means of qtXLS the tables are set up correspondingly so that this prerequisite is filled, (i.e. tables which were produced with qtXLS are also thus readable). Writing is then carried out in the following rows.

- Writing into Excel tables the new data are always added. It is not possible to write to a specific row in the table.
- It can be written only in rows (not column wise).
- The Excel data types NUMBER, DATETIME (date and time), TEXT, CURRENCY and LOGICAL only can be used. Formulas or other formats are not supported.

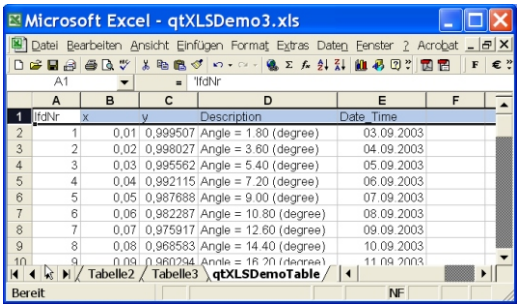

Fig. 2: Excel table qtXLSDemoTable (in qtXLSDemo3.xls, created by qtXLSDemoWriteTable)

- Text formattings (font type, color etc.) are not possible.
- Names of columns and tables can consist out of almost any valid Excel characters. The blank characters (ASCII 32 or CHAR(32)) and the exclamation mark (!) shouldn't be used. Also don't use the dollar sign ($) particularly in table names.
- Names of columns and tables must not be identical with SQL keywords (e.g. INSERT, TEXT, SELECT etc.).
- Maximum length of column names: 63 characters
- Maximum length of table names: 255 characters
- The Excel ODBC drivers support the Excel versions 3.0 4.0, 5.0/7.0, 97, 2000 as well as later as far as they are compatible (this might be the case well, in principle).
- It can be possible that an Excel ODBC driver permits that only a limited number or Excel files are open at the same time. It may also be possible that an Excel ODBC driver allows only the access to one single file.

2. Short Survey of the qtXLS Routines

The qtXLS functions are in a Dynamic-Link-Library (DLL) named **qtXLS.dll**. A following table lists the routines by name and groups them:

Function Group / qtXLS Routine	Function
File Functions	
qtXLSCreateEXCELFile	Create Excel file
qtXLSOpenEXCELFile	Open Excel file
qtXLSCloseEXCELFile	Close Excel file
Table Functions	
qtXLSCreateTable	Create table and define columns
qtXLSReadRows	Read table rows
qtXLSWriteRows	Write table rows
Information Functions	
qtXLSGetTableNames	Get table names
qtXLSDoesTableNameExist	Check if table exist
qtXLSGetColumnInfo	Get information about columns
qtXLSGetRowCount	Count the number of rows in a table
qtXLSGetNumericValue	Get the value of a “Numeric” type
Error Handling	
qtXLSGetErrorMessages	Get error messages
qtXLSSetErrorLevel	Control error handling
qtXLSSetErrorMessagesDisplay	Set up display of error messages
Other Functions	
qtXLSGetszStringLength	Get length of a null-terminated string
qtSet Li cence_qtXLS	Authorize use of qtXLS (see the file qtXLSSet Licence_0611_#####.f90)
qtXLSSetLicencePath	Set path for license file

A detailed description of the qtXLS routines is in the chapter "Reference".

2.1 Use of qtXLS in Your Programs

To use the qtXLS routines in your programs a so-called **binding** is needed that consists of an import library and additional files, e.g. Fortran 90 module files (end on .mod) or a pre-compiled C header file (qtXLS.h). In the chapter "Compile & Link" these compiler specific files are discussed in detail. Furthermore either a license file or a licence routine which is delivered by *QT software* at the purchase of qtXLS is needed for the unrestricted use of the qtXLS functions. The license file has the form

```
L####-######.lic (e.g. L0611-570739.lic)
```

with # representing a digit 0-9.

The license file contains license information about the license and about the licensee.

Alternatively, a license routine is provided (for Fortran programmers only):

```
qtSetLicence_####_######.f90
(for example: qtSetLicence_0611_570739.f90)
```

This file contains the routine

```
SUBROUTINE qtSetLicence_qtXLS( iError )
```

which has to be called prior any other qtXLS routine. Otherwise qtXLS runs in demonstration mode which is indicated by a limited reading and writing functionality.

2.2 Structure of qtXLS Applications

Programs which use qtXLS routines ("qtXLS applications") have a structure which is similar to the common file access. Before an Excel file can be written or read, it must be created calling routine qtXLSCreateEXCELFile or opened with qtXLSOpenEXCELFile. This creates a handle, i.e. a number with which the file can be accessed. One codes, e.g. in Fortran:

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

This looks very similar in C/C++ (the “zero termination” of strings is carried out automatically):

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

The file is closed by

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

This routine also forms the end of a qtXLS application.

Between these two calls (of qtXLSCreateEXCELFile or qtXLSOpenEXCELFile and qtXLSCloseEXCELFile) the other routines have to be embedded for producing tables and to access them.

All INTERFACE blocks or function prototypes to qtXLS functions can be found in a Fortran MODULE or a C header file. As usual for MODULEs, these have to be named at the beginning of the declaration part of your program. One writes in Fortran:

```
USE qtXLS
```

The corresponding instruction is in C/C++:

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

The qtXLS MODULE and the C header file also provide the constants (in Fortran: KINDs and PARAMETERs) and structures (in Fortran: TYPEa; in C/C++: struct) which are used by qtXLS routines. qtXLS exists in compiler specific variants and the modules are provided as pre-compiled (file names ending .mod) (cf. chapter "Compile & Link").

The following example program shows how to create a table named "qtXLSDemoTable", which consists of 5 columns named "lfdNr", "x", "y", "Description" and "Date_Time". The program does not handle any errors that might occur. A complete version is found in the file

qtXLSDemoWriteTable.f90 which is found in the qtXLS installation in the subdirectory “qtXLSDemoWriteTable” of the directory "Examples" . The same example in C/C++ is found in the file 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)
! 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. Reference

3.1 The Fortran 90 MODULE qtXLS and the C Header File qtXLS.h

All interfaces or prototypes to qtXLS functions can be found in a Fortran 90 MODULE and in a C header file. The MODULE is to be specified at the beginning of the declarations section of your Fortran program. One writes in Fortran:

```
USE qtXLS
```

and in C/C++:

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

One MODULE named qtXLSDeclarations is referenced within the MODULE qtXLS. It provides the constants and structures which are used by qtXLS routines. For C/C++ programmers qtXLS header file is supplied which contains everything necessary.

3.2 The Fortran 90 MODULE qtXLSDeclarations

As said before qtXLSDeclarations is referred to internally in the MODULE qtXLS. It provides the constants (KINDs and PARAMETERs) and structures (TYPEs) which are used by qtXLS routines. The source code is in the file qtXLSDeclarations.f90 which can be found in the subdirectory "Bindings" of the qtXLS installation on your harddisk. The MODULE is at the disposal in the source code to make the use of the constants and structures easier for Fortran programmers. An explicit integration (by means of USE) usually isn't necessary.

Some essential constants and structures are explained in the following chapters.

```
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
```

The MODULE qtXLSDeclarations (excerpt from the file qtXLSDeclarations.f90) - to be continued

```
   INTEGER (qt_K_INT2), PARAMETER :: qt_SQL_DOUBLE = 8
   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
```

The MODULE qtXLSDeclarations (excerpt from the file qtXLSDeclarations.f90) - to be continued

```
  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
    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, Munich, Germany.
! All rights reserved.
END MODULE
```

The MODULE qtXLSDeclarations (excerpt from the file qtXLSDeclarations.f90).

3.3 The C Header File qtXLS.h

The C header file qtXLS.h contains the constants necessary for the use in C/C++ programs, structures and function prototypes. These are explained in the following sections.

```
//=============================================================================
// 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;
typedef long  qt_K_INT4;
typedef float   qt_K_R4;
```

The C header file (excerpt from the file qtXLS.h) - to be continued.

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

The C header file (excerpt from the file qtXLS.h) - to be continued.

```
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,
                         LPSTR szOrderBy);
```

The C header file (excerpt from the file qtXLS.h) - to be continued.

```
#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, Berlin, Germany. All rights reserved.
//===========================================================================
```

The C header file (excerpt from the file qtXLS.h).

3.4 Fundamentals to the Call of qtXLS Routines

Some fundamental principles are, applied by the naming of routines and their arguments. This shall make the programming easier and reduce the error probability.

3.4.1 Use of KINDs and Defined Types

The qtXLS routines use arguments whose data types are defined by KIND definitions in Fortran and in C/C++ by typedef. One correspondingly declares the variables. E.g. in C/C++:

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

Or in Fortran:

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

The correct use of the types of arguments is checked with the prototype or INTERFACE declarations which are integrated in the C header file or in the MODULE qtXLS (the compiler should show syntax faults in the case of inconsistencies).

3.4.2 Names of Constants

When naming constants (PARAMETERs) the following scheme was used:

The C/C++ variable types or Fortran KIND constants start with the prefix "qt_K_".

The names of constants that represent error codes start with "qtERROR".

All other names of constants start with "qt_" followed by one or several capital letters which indicate the the type of constant (e.g., I marks an INTEGER, C a character).The variable type is detached from the real name by "_". So "qt_I_SQLDataTypeLEN" marks one INTEGER constant with the real name "SQLDataTypeLEN".

3.4.3 Naming of Routine Arguments

The C function prototypes and INTERFACEs of the qtXLS routines use prefixes at the description of arguments which are put in front of the "eloquent" argument names in lower case letters. E.g.:

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

or

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

The prefixes "h" and "sz" are put in front of the arguments here. The prefixes shall remind the programmer to use the indicated data type. The following table lists these prefixes, their meaning and their data types (KINDs).

Prefix	Meaning	C/C++ Data Type	Fortran Data Type or KIND
i	long “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 or String/Text	char	CHARACTER (*)
t	TYPE or Structure	struct	TYPE (...)

3.4.4 Lengths of Names

The limitations of names of paths, tables and columns are by the following declarations - in C/C++:

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

and in Fortran:

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

It has to be taken into account here, that most of names are followed by the character "ASCII 0" (in Fortran: CHAR (0)) and the actually usable text length therefore is reduced by one character. A column name can therefore be only 63 characters long (qt_I_MaxColumneNameLEN - 1). This should be known to C/C++ programmers.

3.4.5 Zero-Terminated Strings

All qtXLS routines arguments whose names start with "sz", have to be filled with "zero-terminated strings". One adds the "ASCII 0" to the text (in Fortran put an CHAR(0) at the end of the string). This indicates the end of the character string. E.g. in Fortran:

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

This is usually carried out in C/C+ automatically. To determine the length of a text specified this way, Fortran programmers are provided with a function:

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

C/C++ programmers are also able to use this function, however, they presumably prefer to use strlen(). qtXLSGetszStringLength delivers the "operative" length, i.e. without terminating zero. Example in Fortran:

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

and in C/C++

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

return 11 in iLen.

3.4.6 Structures (TYPEs bzw. structURES)

Some of the structures declared in the C header file qtXLS.h or in the MODULE qtXLSDeclarations require some explanation.

3.4.6.1 TYPE and struct qT_ColumnInfo

A structure is used for the information about columns named qT_ColumnInfo. 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
```

and in C/C++:

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

The component **szName** contains the name of a column (zero terminated).

The SQL data type of the column is defined by the member SQLDataType. At the call of the routine qtXLSGetColumnInfo (...), values are returned which correspond to the constants (PARAMETERs) declared in the module qtXLSDeclarations or in qtXLS.h with names "qt_SQL_".

The plain text of the SQL column type is found in the component **TypeName**. This can be: DATETIME, NUMBER, VARCHAR, CURRENCY, LOGICAL, NUMERIC.

The length of the data type or the size of the "column buffer" delivers the component **MaxLen**. In the case of the type DATETIME the number of characters is returned which is necessary to contain the value when converted to text.
In the case of the type NUMERIC MaxLen returns either the total number of digits or the maximum number of possible bits in the column.

3.4.6.2 TYPE and struct qT_SQLColumn

To be able to read or write rows of an Excel table the routines qtXLSReadRows(...) and qtXLSWriteRows(...) must be informed where the data can be found. This is carried out via the structure 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;
};
```

and 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
```

The component **Name** indicates the column name. The string can be zero terminated. If it isn't zero terminated, the component must be padded with blanks.

The component **ArrayAddr** has to be filled with the memory address of the variable or the array to which values are written to or read from. In C/C++ one finds this in the common way (depends on the declaration of the array). Depending on the Fortran compiler used one determines the address with the help of an INTRINSIC FUNCTION (since the Fortran standard doesn't provide any general function one must look for compiler specific functions). E.g.:

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

The size (number of bytes or characters) of the array defined before is to be specified in the component **ArrayDim**.

So that the driver knows of which type the array is, one indicates this in the component **ArrayType**.

The length of an array element is specified (in bytes or characters) by **LENArrElem**.

The following table lists the possible type specifications and corresponding C/C++ or Fortran data types as well as their element lengths (in byte).

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

The component **IndArrAddr** isn't used and must be set to 0.

The following example shows a typical specification for a column named 'lfdNr'. The values of the column to be read or written are to be found in the

array lfdNrArr of size DIMArr. The structure variable of TYPE qT_SQLColumn is named 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 the column definition can also be programmed by the help of a TYPE constructor. E.g.:

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

The order of the components has to be taken into account here!

3.4.6.3 TYPE and struct qT_TIMESTAMP_STRUCT

Date and time values are specified by a structure of TYPE qT_TIMESTAMP_STRUCT. 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;
};
```

All components with the exception of the last one are known. The component **fraction** holds the time in nano-seconds (10^{-9}s; time range: 0 ns to 999999999 ns).

3.4.7 Error Codes and Error Handling

If an error occurs in a qtXLS routine, then the error condition is shown either by a function return value or by an explicit argument ("iError"). The meaning of the "current" error code can be questioned by call of the routine qtXLSGetErrorMessages(...) . It becomes displayed automatically if the "error display mode" has (...) been switched on with the help of routine qtXLSSetErrorMessagesDisplay(...). An error can be caused by a faulty argument specification, or either by insufficient internal buffer sizes or by an error which appears in the Excel ODBC driver. The qtXLS of specific error code are declared in the MODULE qtXLSDeclarations or in the C header file qtXLS.h. A following table lists them.

qtXLS Error Code	Meaning
qtERRORAllocHandleFailed	Internal memory could not be allocated and therefore no handle could be obtained.
qtERRORConnectFailed	No connection to the Excel ODBC driver could be set up.
qtERRORExecDirectFailed	The (internal function) ODBC ExecDirect could not be executed.
qtERRORInsufficientDimension	The dimension of an array given as an argument is too small.
qtERRORInsufficientSize	An internal buffer is too small.
qtERRORInvalid	a) The license code checked by qtXLS at the initialization is invalid are b) An argument contains an invalid value. c) A length specification is missing (re qT_SQLColumn). d) An invalid data type was indicated (re qT_SQLColumn). d) An address specification is missing (re qT_SQLColumn). e) The dimension of an array was specified faultily (re qT_SQLColumn).
qtERRORNameNotSpecified	A name wasn't given.
qtERRORNotSupported	A data type isn't supported (re qT_SQLColumn).
qtERRORNotZeroTerminated	A text didn't become zero terminated.
qtERRORSQLFunctionFailed	An error occurred when executing an Excel ODBC driver function.
qtERRORUnknown	Unknown error.

A qtXLS routine usually breaks off in the fault case and returns to the calling program. However, one can increase the "Error level" by calling (in Fortran)

```
CALL qtXLSSetErrorLevel( 1 )
```

or in C/C++

```
qtXLSSetErrorLevel( 1 );
```

and try to continue a process despite an internal fault provided that this is possible.

3.5 Description of the qtXLS Routines

qtSetLicence_qtXLS - Set qtXLS License

C/C++:

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

Fortran:

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

The routine qtSetLicence_qtXLS is supplied in source form and contains information about the licensee and the licence code.
The routine returns an error code. If this not equal 0, no valid license was found, and qtXLS will operate in demo mode only.

Interna

qtSetLicence_qtXLS(...) internally uses the MODULE qtCompilerModule_QTXLS. This is a compiler specific MODULE. With some compilers its path has to be specified when compiling the file qtSetLicence_0611_######.f90. Compare. example programs.

qtXLSCloseEXCELFile - Close Excel File

C/C++:

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

Fortran:

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

An Excel file which has been created with qtXLSCreateEXCELFile(...) or opened with qtXLSOpenEXCELFile(...) before is closed by the routine qtXLSCloseEXCELFile(...). The argument “Handle” (hDS) returned by either of the two routines has to be submitted.

Interna

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

qtXLSCreateEXCELFile - Create Excel File

C/C++:

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

Fortran:

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

The routine qtXLSCreateEXCELFile(...) creates an Excel file whose name has to be given in the argument szFileName (zero terminated).
The file name in szFileName can contain a path and must be terminated by ASCII 0. The file name should sensibly end on .xls (Excel files are normally "registered" under this ending).
If the file specified by szFileName already exists, it is overwritten (and is then "empty" at first). The file otherwise is laid out newly. As the function's value, a handle will be returned , which has to be used in all following calls of qtXLS routines. Zero (0) is given back in the case of an error and its reason can be inquired by qtXLSGetErrorMessages(...).

C/C++ Example

```
#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 Example

```
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(...) initializes qtXLS and creates a connection to the Excel file with the help of the Excel ODBC driver ("Connect"). Internally, memory will be allocated which will be released again when qtXLSCloseEXCELFile(...) is called.

qtXLSCreateTable - Create Table

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
```

An Excel table (also called "sheet") will be created by qtXLSCreateTable(...) in a file which has previously been opened by either qtXLSOpenEXCELFile(...) or qtXLSCreateEXCELFile(...). Both routines return a handle (hDS) which has to specified on input of qtXLSCreateTable(...).

Table Definition

The table definition consist of the table name and the column names as well as their type specifications. It has to defined according to the following scheme:

```
"table name (column name 1 column type 1, column name
2 column type 2, ...)" // CHAR(0)
```

This way up to 255 columns can be laid out (internal restriction of Excel). At the name specification all valid Excel characters can theoretically be used. One is well advised, however, to confine oneself to the "readable" US ASCII set (ASCII 32 through 127) and furthermore not to use blanks (ASCII 32), no exclamation marks (!) and no dollar signs ($) in the name. In addition, you must pay attention that these names coincide with SQL keywords or SQL statements (e.g. SELECT, INSERT, TEXT, NUMBER, MONEY etc.). For the detail of the column type the following identifiers are possible: CURRENCY, DATE TIME, LOGICAL, NUMBER and TEXT.

If the table could be laid out successfully, qtXLSCreateTable (...) returns 0 as its function value. Otherwise an error code is returned whose meaning can be inquired with a call to the routine qtXLSGetErrorMessages(...).

C/C++ Example

```
#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 Example

```
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 - Check if Table exists

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
```

To check whether a table already exists within an Excel file which is identified by its handle (hDS) call qtXLSDoesTableNameExist(...) and specify the table's name in szTableName (zero terminated). The function returns:

1 : if the table exists.

0 : if the table does not exist.

-1 : if an error occurred (internally).

If an error occurs, its reason can be found out by a call to the routine qtXLSGetErrorMessages(...).

C/C++ Example

```
#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 Example

```
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 - Get Column Information

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
```

Information about the columns can be obtained for a table whose name has to be given in szTableName (zero terminated) with the help of qtXLSGetColumnInfo(...). The Excel file whose table information is questioned on is identified through the handle hDS (see qtXLSOpenEXCELFile(...) or qtXLSCreateEXCELFile(...)). The column information is returned in the structure tColumnInfo. The structure must contain iDIMColumnInfo elements. If more columns are in the table than tColumnInfo has elements, the routine returns the error code iError =

qtERRORInsufficientDimension. Then, the required dimension (i.e. the number of columns) of structure field tColumnInfo is found in iCountColumnInfo. If no error occurs (iError = 0), iCountColumnInfo also returns the number of columns in the table.

The column information in tColumnInfo(*j*) consists of the name of the column *j* (zero terminated or padded with blanks), its SQL data type (identified by a number), the type name in plain text (DATETIME, NUMBER, VARCHAR, CURRENCY, LOGICAL or NUMERIC) and the length of the column or the column buffer (in bytes). In C/C++ the structure is defined as follows (cf. qtXLS.h):

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

And 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
```

Details to this type TYPE qT_ColumnInfo is found in the chapter (3.4.6.1) bearing the same name.

C/C++ Example

```
// 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 Example

```
! 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 - Get Error Messages

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
```

If an error occurs during the execution of a qtXLS routine , then the error number is stored internally and the accompanying error message can be investigated by means of qtXLSGetErrorMessages(...). The routine returns the error message zero terminated in the argument szErrorMessages. The size of this buffer must be indicated at the call in iBufSizeErrorMessages. An error text buffer with space for approx. 2000 characters might suffice in most cases. qtXLSGetErrorMessages(...) returns as its function value:

- 0 : either because there wasn't an internally stored error code or if the error message could be determined completely for an internally stored error code.
- -1 : if an error has occurred at the execution of qtXLSGetErrorMessages(...).
- >0 : if the buffer szErrorMessages is too small. Then the value returned specifies the necessary size of the buffer (in characters).

C/C++ Example

```
#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 Example

```
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 - Get Numerical Value

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
```

If columns of the type NUMERIC are read from an Excel table, their values have to be converted into one of those elementary floating point number types supported by Fortran, C etc.. The NUMERIC type is stored in a structure qT_NUMERIC_STRUCT. It is defined in C/C++ as follows:

```
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];
};
```

And 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
```

The function qtXLSGetNumericValue(...) then returns the value converted into a 8 byte floating point number (Type double bzw. REAL*8).

qtXLSGetszStringLength - Get Length of a zero terminated String

C/C++:

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

Fortran:

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

The operative length of a zero terminated string of characters (zero-terminated string) is determined by the function

qtXLSGetszStringLength(...). The length of the character string is returned without the terminating zero.

Fortran Example

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

qtXLSGetTableNames - Get Table Names

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
```

The names of all tables of an Excel file which is identified by the argument hDS (see qtXLSCreateEXCELFile(...) or qtXLSOpenEXCELFile(...)), can be determined by a call to routine qtXLSGetTableNames(...).
The names are returned in the CHARACTER string array szTableNames zero terminated and padded with blanks (pay attention to the length declaration qt_I_MaxTableNameLEN).The array is dimensioned with iDIMTableNames. In C/C++ you have to to define an array with fixed string lengths qt_I_MaxTableNameLEN (for example: char szTableNames[iDIMTableNames][qt_I_MaxTableNameLEN];
The number of available or found tables in the Excel file is given back in iCountTableNames. The last argument iError returns 0 if no fault has appeared at the execution of the routine. It otherwise contains an error code (qtERROR ...). If iError returns the error code qtERRORInsufficientDimension, iCountTableNames contains the required dimension of the array szTableNames(), i.e. the number of tables.
The table names given back in szTableNames have mostly one “$” as the last character (for example "Coordinates $"), even if the table name was defined without the “$” character in a call of qtXLSCreateTable(...). This is a peculiarity of the Excel ODBC driver. Unfortunately it wasn't possible to receive further information about the backgrounds of this behaviour. Tests yielded that the name of table created by means of qtXLSCreateTable(...) is recognized with and also without "$". So one can continue safely to use

the table names found out using qtXLSGetTableNames(...) in the following calls of qtXLS routines without having to remove the dollar sign at the end.

C/C++ Example

```
#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 Example

```
! 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 - Count Rows in a Table

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
```

The number of rows in a table whose name was given in szTableName (zero terminated) is returned by the function qtXLSGetRowCount(...). The argument hDS identifies the Excel file (see qtXLSCreateEXCELFile(...) or qtXLSOpenEXCELFile(...)) in which the table is to be found. If a fault appears at the execution of qtXLSGetRowCount(...), the value -1 is given back. The meaning of the error can be questioned by call of the routine qtXLSGetErrorMessages.

C/C++ Example

```
#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 Example

```
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 - Open Excel File

C/C++:

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

Fortran:

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

An Excel file whose name is given in szFileName is opened by means of qtXLSOpenEXCELFile(...). The file name in szFileName can contain a path specification and must be terminated with ASCII 0. If the file exists and could be opened successfully, the function returns a handle which is to be used in all following calls of qtXLS routines.
In the fault case, 0 is returned and the error code or his meaning can be determined by a call to routine qtXLSGetErrorMessages(...).

C/C++ Example

```
#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 Example

```
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(...) initializes qtXLS and creates a connection to the Excel file by means of the Excel ODBC driver ("Connect"). Internally memory is allocated, which is freed by a call to routine qtXLSCloseEXCELFile(...).

qtXLSReadRows - Read Rows

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
```

qtXLSReadRows(...) reads rows in a table whose name is specified by szTableName (zero terminated). The table belongs to an Excel file which is identified by its handle hDS (see qtXLSCreateEXCELFile(...) or qtXLSOpenEXCELFile(...)). The number of rows to be read is to be given in the argument iNoRows. If all rows shall be read, set iNoRows = -1.
Which columns are read is determined by a definition in the argument tColumns(). Its dimension iNoColums determines the number of columns to be read. tColumns() contains the names of the columns to be read, the specification in which array the values shall be stored, the variable's type of this array (for example qt_SQL_C_DOUBLE for a double and REAL*8 array, respectively), its dimension and the length of a single array element (in bytes or characters). The structure definition is, 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;
};
```

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 to the TYPE qT_SQLColumn conveys the chapter bearing the same name (3.4.6.2).

■ Specifications for tColumns - Column Type and Array Type

The dimension (in the member ArrayDim) of the array given by the component ArrayAddr arises from the number of rows to be read (therefore at least iNoRows). The choice of the array in which the read column values shall be put, is usually determined by the Excel column type. Here the assignment of an Excel or ODBC column type to the corresponding variable type of the array (ArrayType) is important. A following table shows the relations.

Column Type	Variable Type of the Array	SQL Array Type
CURRENCY	struct qT_NUMERIC_STRUCT or TYPE (qT_NUMERIC_STRUCT)	qt_SQL_C_NUMERIC
DATETIME	struct qT_TIMESTAMP_STRUCT or TYPE (qT_TIMESTAMP_STRUCT)	qt_SQL_C_TIMESTAMP
LOGICAL	in C/C++ 1-Byte integer or in Fortran LOGICAL or INTEGER(1)	qt_SQL_C_BIT
NUMBER	float, double, int, short oder long or REAL or INTEGER	qt_SQL_C_FLOAT qt_SQL_C_DOUBLE qt_SQL_C_SHORT qt_SQL_C_LONG
TEXT	char or CHARACTER	SQL_C_CHAR

For reading a text column one therefore uses an array of the type char or CHARACTER and indicates in C/C++ for the ArrayType

```
tColumns[iColumnNo]->ArrayType = SQL_C_CHAR;
// or (dependent on the declaration of tColumns)
tColumns[iColumnNo].ArrayType = SQL_C_CHAR;
```

or in Fortran:

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

It is also possible to use the conversion facility of the ODBC driver, which may be able to convert data from one column type (CURRENCY, DATETIME, LOGICAL, NUMBER or TEXT) in another column variable type according to the table shown above. This procedure is recommended

particularly at the column types CURRENCY and NUMBER, to which advantageously a REAL array is "binded" via tColumns() (component ArrayType = qt_SQL_C_DOUBLE or qt_SQL_C_FLOAT). An INTEGER array is possible too (component ArrayType = qt_SQL_C_LONG or qt_SQL_C_SHORT), provided one knows, that the column contains integer numbers in a valid range of INTEGER values only. For all Excel and ODBC column types the binding of a CHARACTER array (component ArrayType = qt_SQL_C_CHAR) should be possible (i.e., specifications of numbers, date and time values are converted to strings).

See also the description of qtXLSWriteRows(...).

■ Search Conditions and Sort Order

Which rows shall be read can be restricted by a condition specified in szCondition. The zero terminated string holds the condition conforming to SQL rules. Its form therefore is

```
"WHERE Search Condition"
```

The "*Search Condition*" contains the column names and the restrictions to be applied. Several conditions have to be connected by SQL operators like AND or OR. An example:

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

Furthermore the sort order of the values returned in the arrays given by tColumns() can be specified. The argument szOrderBy holds a zero terminated string (SQL conforming) which determines the sort order. Its form is:

```
"ORDER BY column name [ASC | DESC]"
```

Here, several column names have to be separated by a comma. ASC and DESC determine if the sort will be ascending (default) or descending. Example in Fortran:

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

In C/C++:

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

The maximum length of the strings in szCondition or szOrderBy should not exceed the value of the parameter qt_I_MaxStatementLEN.

■ Return Value of the Function

The number of read rows is returned as the function value.

If a fault appears, qtXLSReadRows(...) returns -1. The meaning of the error can be inquired by call of the routine qtXLSGetErrorMessages(...).

■ C/C++ Example

```
// 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 Example

```
! 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 - Set Error Level

C/C++:

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

Fortran:

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

.

A qtXLS routine usually stops its execution in the case of an error and returns to the calling program. One can take care by increasing the error level from 0 (default) to 1 that a qtXLS routine continues despite the occurrence of an internal error, provided this is possible. For example, this can be the case if data could not be transmitted completely (for example, because a buffer being too small).
The routine qtXLSSetErrorLevel(...) can be called at any time to change the error treatment with either one of the two values (0 or 1).

qtXLSSetErrorMessagesDisplay - Set Error Display Modus

C/C++:

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

Fortran:

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

When testing qtXLS applications a steady error checking with an automatic error indication can be very helpful. Calling

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

or in Fortran

```
CALL qtXLSSetErrorMessagesDisplay( 1 )
```

the error display modus is activated in the qtXLS routines. This means, if an error occurs in a qtXLS routine, in most cases an error message is displayed in a dialog window. This is to be closed by the user by pressing an 'OK button, which also causes the program to continue execution. (cf. illus. 7).

Fig. 7: Error message due to qtXLSSetErrorMessagesDisplay(1)

This behavior is switched off accordingly to the presetting or one can be turned off through

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

or in Fortran by

```
CALL qtXLSSetErrorMessagesDisplay( 0 )
```

The error messages are identically with those returned by qtXLSGetErrorMessages(...).

qtXLSSetLicencePath - Set Licence Path

C/C++:

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

Fortran:

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

If the usage of qtXLS has not be authorised by

```
CALL qtSetLicence_qtXLS( iError )
```

it is tried at the initialization of qtXLS to read a licence file which is delivered by QT software at the purchase of a qtXLS licence. This licence file is

usually searched in the the directory where the qtXLS.dll is located. This presetting can be changed by putting the licence file path in szPathName (zero terminated string). szPathName must contain a valid path specification.
The routine qtXLSSetLicencePath(...) must be called before all other qtXLS routines.

C/C++ Example

```
#include <qtXLS.h>

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

Fortran Example

```
USE qtXLS

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

qtXLSWriteRows - Write Rows

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)
```

In a table specified by szTableName (name is zero terminated) of an Excel file identified through hDS (see qtXLSCreateEXCELFile(...) or qtXLSOpenEXCELFile(...)), iNoRows rows can be added with the help of qtXLSWriteRows(...). Only those columns of the rows written are occupied with data, which are defined in the area tColumns(). The dimension of the field tColumns() has to be given in iNoColums.
tColumns() contains the names of the columns of the table, the specification in which array to find the values to be exported, which variable's type this array has (for example qt_SQL_C_DOUBLE for a double and REAL*8 array), its dimension and the length of an array element (in characters or bytes). The structure is defined in C/C++ as follows:

```
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;
};
```

And 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 to the TYPE qT_SQLColumn conveys the chapter bearing the same name (3.4.6.2).

The dimension (in the component ArrayDim) of the array given by the component ArrayAddr arises from the number of rows to be written (at least therefore iNoRows). The assignment of an Excel or ODBC column type to the corresponding type of the array as well as the SQL ArrayType to be used for this variable shows the following table.

Column Type	Variable Type of the Array	SQL Array Type
CURRENCY	struct qT_NUMERIC_STRUCT or TYPE (qT_NUMERIC_STRUCT)	qt_SQL_C_NUMERIC
DATETIME	struct qT_TIMESTAMP_STRUCT or TYPE (qT_TIMESTAMP_STRUCT)	qt_SQL_C_TIMESTAMP
LOGICAL	in C/C++ 1-Byte integer or in Fortran LOGICAL or INTEGER(1)	qt_SQL_C_BIT
NUMBER	float, double, int, short or long or REAL or INTEGER	qt_SQL_C_FLOAT qt_SQL_C_DOUBLE qt_SQL_C_SHORT qt_SQL_C_LONG
TEXT	char or CHARACTER	SQL_C_CHAR

For writing a text column (column type = TEXT) one therefore uses an array of the type char or CHARACTER and specifies for the ArrayType in C/C++

```
tColumns[iColumnNo]->ArrayType = SQL_C_CHAR;
// or (dependent on the declaration of tColumns)
tColumns[iColumnNo].ArrayType = SQL_C_CHAR;
```

and in Fortran:

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

The assignments between variable type and column type in the above table aren't mandatory, though. The Excel ODBC driver is capable of conversions. For example, one wants to write a column of the type CURRENCY, it is recommended to use an double or REAL*8 array of type qt_K_R8 or REAL (qt_K_R8) instead of an array of TYPE (qT_NUMERIC_STRUCT). Example in Fortran:

```
! Definition of a column of type 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
```

Or in C/C++

```
! Definition of a column of type 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
```

See also description of routine qtXLSReadRows(...).

The rows are always appended to the table. It is not possible to specify a particular row of the table into which one would like to write.
If more columns in the table are available than are defined in tColumns(), these columns are marked with a value provided by Excel, i.e. as "not set" (reading these columns does not return a value).
If successfully (iError = 0), the routine returns the number of rows written. It should be identical with the number specified on entry in iNoRows.
In case of an error, a function value of -1 is returned and iError then contains the error code.

■ C/C++ Example

```
#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 Example

```
! 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. Compile & Link

4.1 General Notes

Programs (.exe), that are based on qtXLS, require the Dynamic-Link-Library qtXLS.dll at runtime. If usage of qtXLS is not authorised by calling the routine qtSetLicence_qtXLS, a license file is necessary for the unrestricted use of the qtXLS functions (form L ####-######.lic) otherwise the qtXLS.dll can be used only in demonstration mode.
Since the qtXLS routines use functions of the Microsoft Excel ODBC driver, such must be installed (cf. chapter "introduction").
To develop qtXLS based programs (.exe) different bindings are available for various compilers. These consist of partly necessary import library for the qtXLS.dll as well as other files, (e.g. pre-compiled Fortran 90 MODULE files - ending on .mod - or header files (.h)). Furthermore are files (for exampe batch files) available in these bindings which serve the "assembly" of the enclosed qtXLS demo programs. In the following sections it is described how the different bindings are used in concert with the compiler system they are made for.

Generally, **4 byte (long) INTEGERs are used** (this is usually the presetting for all compilers).

With Absoft ProFortran for Windows

The binding for the use with Absoft Pro Fortran for Windows (v10.0 and compatible) is to be found in the directory

qtXLS\Bindings\ProFortran

of the installation. It consists of the files

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

Compile & Link

For compiling a qtXLS based program the MODULE files

```
QTXLS.MOD
```

```
QTXLSDECLARATIONS.MOD
```

QTCOMPILERMODULE_QTXLS.MOD

are needed. They have to be located in a directory the compiler has access to. If necessary, the MODULE path has to be set using the compiler option -p <pathname>.

In the development environment "Developer Tools Interface" the path is to be given in the dialog "Module File Path(s)" (see following illustration).

Press the button “Set Module Path(s)...” in the register card "F95" of dialog “Project Options”.

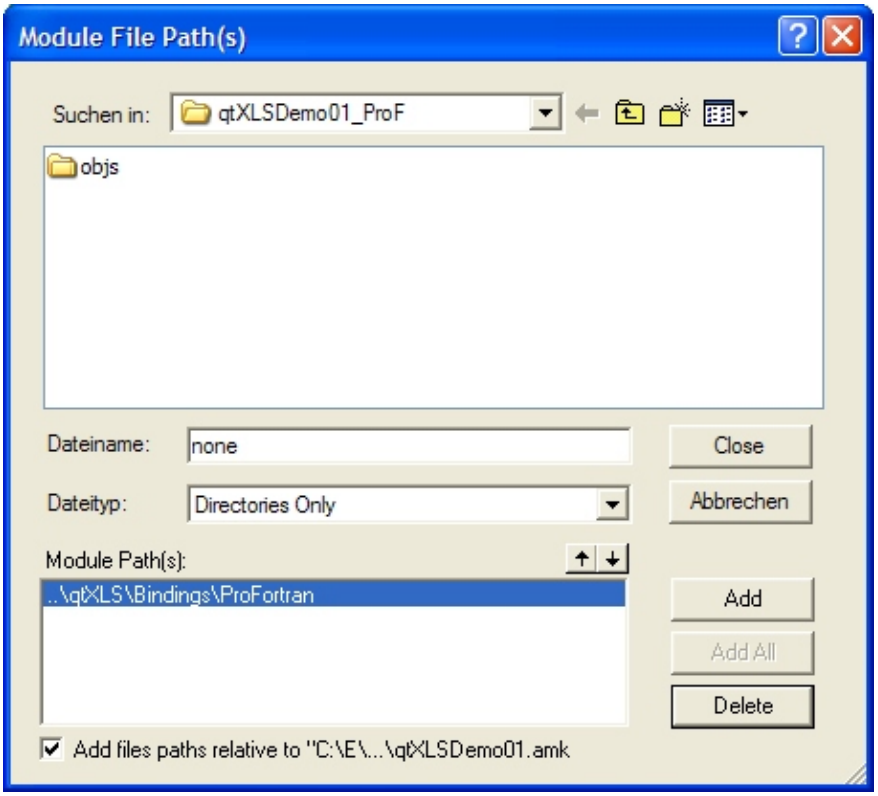

Fig.10: Dialog “Module File Path(s)” with specificaton of the MODULE path

When linking, the libraries

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

have to be specified. In the development environmen add these files to the project (see adjoining figure).

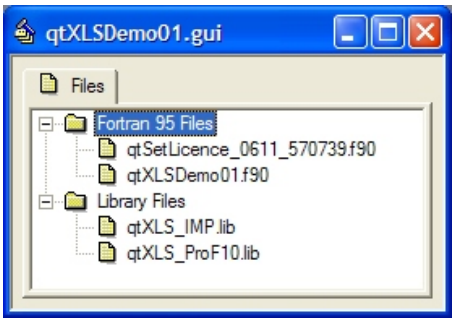

Fig. 8: Files in a qtXLS project

Provided all files mentioned above are in the same directory, F95 has to be called as follows (here for a console application):

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

The binding for F95 contains a batch file

```
clProF10.bat
```

which you can copy and modify according to your needs. The batch file is called from another one named

```
BuildDemosWithProF10.bat
```

in order to build the qtXLS demonstration programs.

With Compaq Visual Fortran

The binding for use with Compaq Visual Fortran (abbreviated CVF) v6.6 is found in a directory named

qtXLS\Bindings\CVF

of the installation. For version 6.1 of CVF it is found in the directory

qtXLS\Bindings\CVF61

Both bindings consist of files named

```
qtXLS_CVF.lib
qtXLS_CVF.exp
QTFOREXCELMODULE.MOD
QTXLS.MOD
QTXLSDECLARATIONS.MOD
qtCompilerModule_QTXLS_VF.obj
```

and a subdirectory

qtXLS\Bindings\CVF\BuildDemosWithCVF

and

qtXLS\Bindings\CVF61\BuildDemosWithCVF

respectively, in which a workspace file

`BuildDemosWithCVF.dsw`

is located. This can be loaded (by a "double click" in Windows File Explorer) into the development environment (IDE) of CVF. The workspace contains four projects (stored in .dsp files)

`qtXLSDemoCreateTable`

`qtXLSDemoListTablenames`

`qtXLSDemoReadTable`

`qtXLSDemoWriteTable`

which serve to create the qtXLS demonstration programs (cf. illus.).

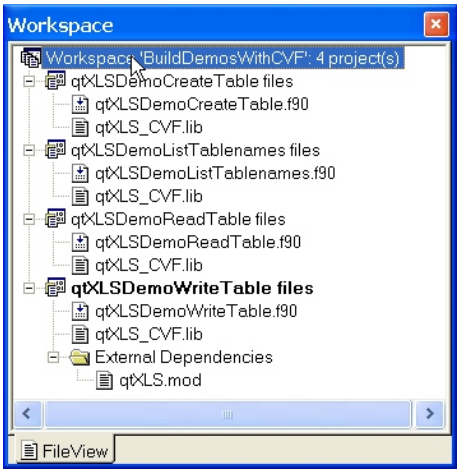

Fig. 11: CVF Workspace "BuildDemosWithCVF"

Compile & Link

Furthermore these projects demonstrate how the qtXLS Library and the MODULEs files have to be bound.

The compiler has to be informed in the entry field "INCLUDE and USE Paths" of the register card "Fortran" (of the "Project Settings" dialog), category "Preprocessor", where the module files

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

are located (cf. illus.).

The linker needs to know where the import library

`qtXLS_CVF.lib`

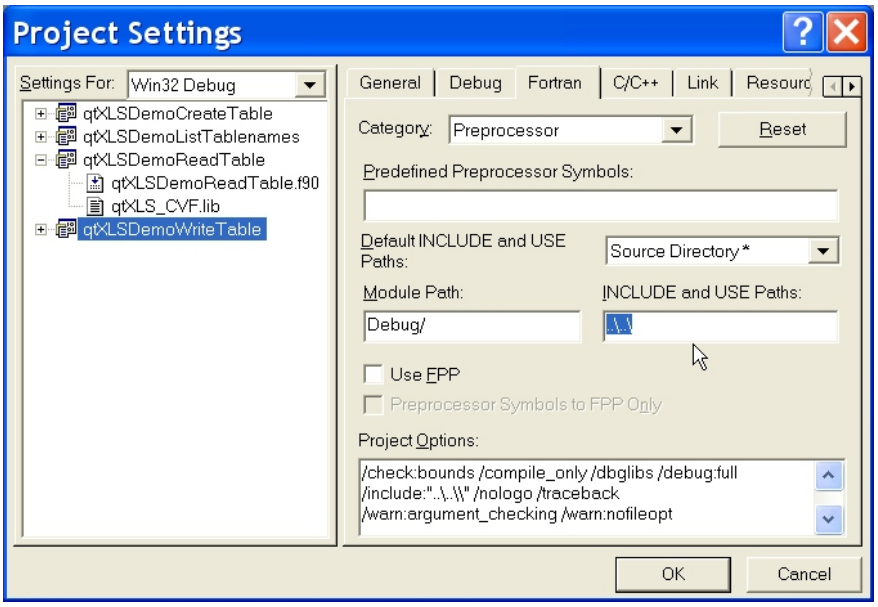

Fig. 13: Project Settings with entry "INCLUDE and USE Paths"

is to be found. Here it suffices to add the library to the project files (cf. the above illus.11). A simple click on the "Build" button (or in the menu "Build | Build ..exe . ") then starts creating the executables.

With Intel Visual Fortran

The binding for use with Intel Visual Fortran (abbreviated IVF) is found in a directory named

qtXLS\Bindings\IVF

of the installation. The binding consists of files named

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

and a subdirectory

qtXLS\Bindings\IVF\BuildDemosWithIVF

in which a solution file

```
BuildDemosWithIVF.sln
```

is located. This can be loaded (by a "double click" in Windows File Explorer) into Visual Studio, the development environment (IDE) of IVF. The solution contains four projects (stored in .vfproj files)

```
qtXLSDemoCreateTable
qtXLSDemoListTablenames
qtXLSDemoReadTable
qtXLSDemoWriteTable
```

which serve to create the qtXLS demonstration programs.

Compile & Link

Furthermore these projects demonstrate as the qtXLS Library and the MODULE files have to be bound.

The compiler has to be informed in the entry field "Additional Include Directories" of the register card "Fortran" (of the "Project Properties" dialog), category "Preprocessor", where the module

```
QTXLS.MOD
```

```
QTXLSDECLARATIONS.MOD
```

files are located.

The linker needs to know, where the import library

```
qtXLS_IVF.lib
```

is to be found. Here it suffices to add the library to the project files. Then, a simple click on the "Build Solution" button (or in the menu "Build | Build Solution") is all you need to create the executables.

Debugging & Testing

To test the demo programs within Visual Studio or to run them in the debugger, you have to provide the correct settings for the working directory of the debugger (cf. illus).

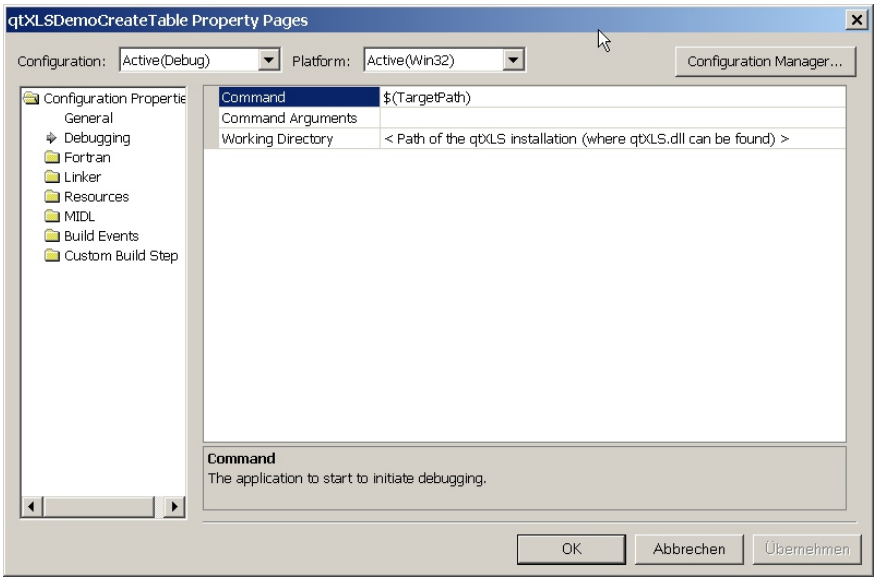

Fig. 14: Project Properties and entry field "Working Directory"

The demo programs require the qtXLS.dll which is located in the installation directory of qtXLS. Its path is to be entered in the field named "Working Directory". Unfortunately, it appears that Visual Studio does not allow to enter relative pathnames here (..\..\..\ would be sufficient).

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

The binding for the use with Lahey/Fujitsu LF95 (v5.7 and compatible) is to be found in the directory

qtXLS\Bindings\LF9557

of the installation. It consists of the files

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

Compile & Link

For compiling a qtXLS based program the MODULE files

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

are needed. They have to be located in a directory the compiler has access to. If necessary, the MODULE path has to be set using the compiler option -MOD <pathname>.

When linking, the library

```
qtXLS_LF9557.lib
```

have to be specified.

Provided all files mentioned above are in the same directory, LF95 has to be called as follows:

```
LF95 <file name>.f90 qtSetLicence_0611_######.f90
qtXLS_LF9557.lib
```

(###### ist to be replaced by the license number you have obtained). The binding for LF95 contains a batch file

```
clLF9557.bat
```

which you can copy and modify according to your needs. The batch file is called from another one named

```
BuildDemosWithLF9557.bat
```

in order to build the qtXLS demonstration programs.

With Microsoft Visual C++

The binding for the use with Microsoft Visual C++ (abbreviated VC) (v6.0 and compatible) is to be found in the directory

qtXLS\Bindings\VC6

of the installation. It consists of the files

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

and a subdirectory

qtXLS\Bindings\VC6\BuildDemosWithVC

in which the workspace file

```
BuildDemosWithVC.dsw
```

is found. This can be loaded (by a "double click" in Windows File Explorer) into the development environment (IDE) of VC. The workspace contains 4 projects (stored in .dsp files)

```
qtXLSDemoCreateTable
qtXLSDemoListTablenames
qtXLSDemoReadTable
qtXLSDemoWriteTable
```

which serve to create the qtXLS demonstration programs.

Compile & Link

Furthermore these projects demonstrate as the qtXLS Librarys and the MODULEs files have to be tied and how to set the path for header file qtXLS.h.

The compiler has to be informed in the entry field “Additional Include directories” of the register card "C/C++" (of the "Project Settings" dialog), category "Preprocessor", where the header file qtXLS.h resides.

The linker needs to know, where the import library

```
qtXLS_IMP.lib
```

is to be found. Here it suffices to add the library to the project files. A simple click on the "Build" button (or in the menu "Build | Build ..exe . ") then provides build everything else.

With Salford FTN95 or Silverfrost FTN95 (Win32)

The binding for the use with FTN95 is to be found in the directory

qtXLS\Bindings\FTN95

of the installation. It consists of the files

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

Compile & Link

For compiling a qtXLS based program the MODULE files

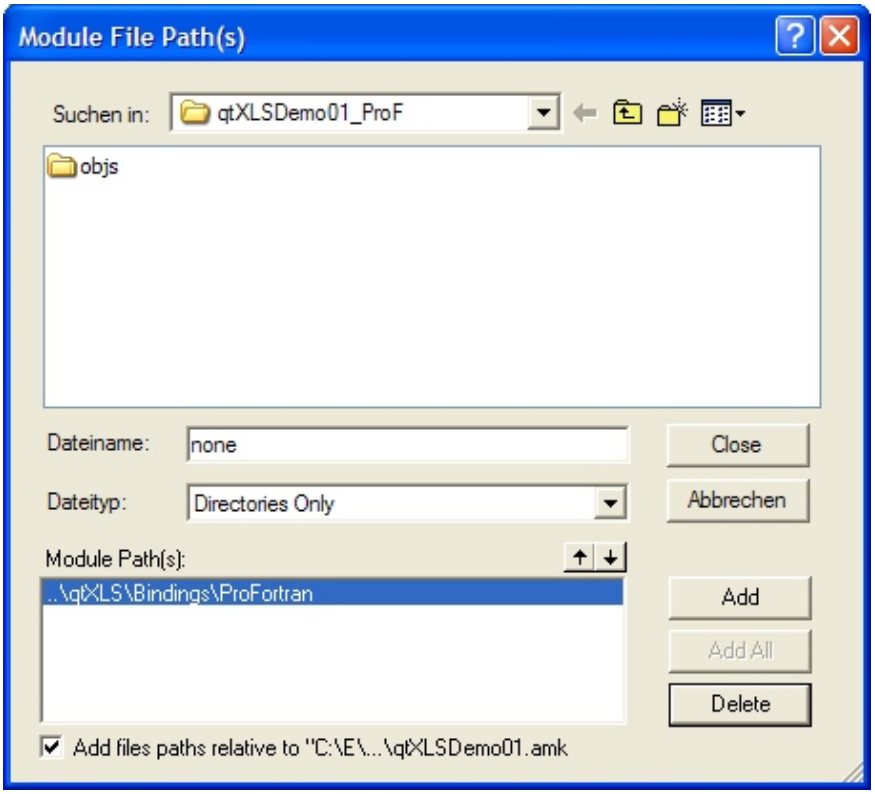

Fig. 15: MODULE Path in Plato3 (Dialog “Properties”)

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

are needed. They have to be located in a directory the compiler has access to. If necessary, the MODULE path has to be set using the compiler option /MOD_PATH <pathname>.
When using the development environment of FTN95 (Plato) the MODULE path has to be given in the "Properties" dialog (cf. illus.15).

When linking, the dynamic link library (DLL)

```
qtXLS.dll
```

and the import library

```
qtXLS_FTN95.lib
```

have to be specified. In Plato add these files to "References" (cf. illus. below).

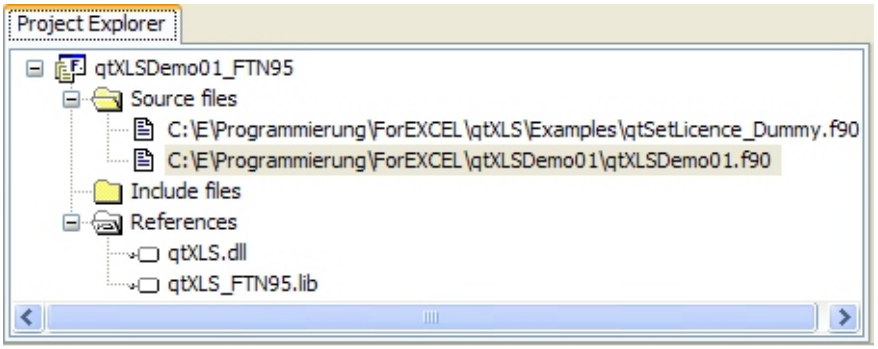

Fig. 16: Files in the Project Explorer of Plato3

Provided all files mentioned above are in the same directory, FTN95 has to be called as follows:

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

(####_###### has to be replaced by your license number). The binding for FTN95 contains a batch file

```
clFTN95.bat
```

which you can copy and modify according to your needs. The batch file is called from another one named

```
BuildDemosWithFTN95.bat
```

in order to build the qtXLS demonstration programs.

5. Contents and Structure of the qtXLS Installation

qtXLS is provided in a packed file (in the ZIP format):

```
qtXLS.zip
```

When unpacking in a directory arbitrarily selected by the programmer, a directory structure is created as it shows a following illustration.

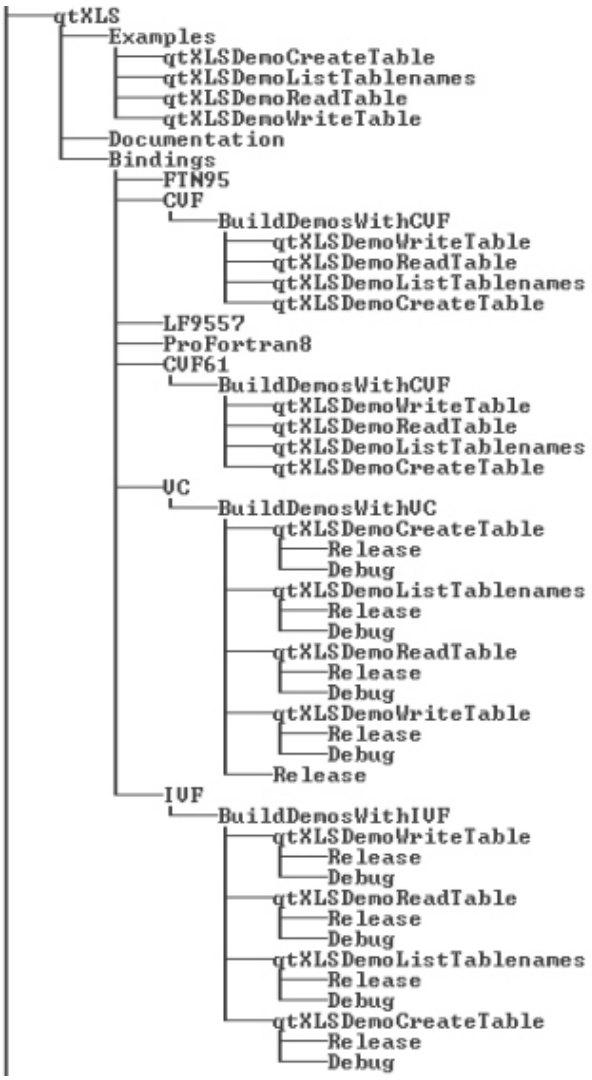

Fig. 17: Directory structure of an qtXLS installation

In the root directory of the installation the directory “qtXLS” is found. In this the

```
qtXLS.dll
```

is located, as well a the Excel example files:

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

In the directory "Bindings" the Libraries, the .h and .mod files for the supported compilers as well as batch files (.bat) and other compiler specific "Make" files which serve the production of the example programs, are put down. The example programs are found in the subdirectory “Examples”.

6. Passing on of qtXLS Applications

Programs (.exe) which access qtXLS functions require the file

```
qtXLS.dll
```

for full functionality. A licensee may pass on (see chapter "Use Conditions") this file with his application.
If usageof qtXLS is not authorised by call of routine qtSetLicence_qtXLS(...) a license file is needed to be passed on with the qtXLS application:

```
License file (format: L####-######.lic with # = 0
through 9)
```

The passing on of all other files of the software qtXLS is not allowed.

7. System Requirements

To be able to use the qtXLS software, the following is required:

- PC with a Pentium processor, hard disk with at least 15 MB memory available, a minimum of 32 MB RAM.
- Supported operating systems are: Microsoft Windows 95, Windows 98, Windows NT 4.0, Windows 2000, Windows XP and compatible (it is explicitely stated, that some Microsoft Excel ODBC drivers are compatible to specific Windows operating systems only).
- Microsoft Excel ODBC driver (see chapter “Introduction”)
- Compiler systems: Fortran 90/95 or C/C++ compilers and linkers listed in the chapter “Compile & Link”.

8. Licence Agreement - Legal Conditions to Use the qtXLS Software

§1. Property and Ownership

The software described in this document - called qtXLS software in the following, mainly consists of the files named qtXLS.dll and those mentioned in the chapter “Compile & Link”. In particular these are library files ending on the .lib, pre-compiled MODULE files (ending on .mod), this documentation, the source files qtXLS.h and qtXLSDeclarations.f90, and any other file mentioned in this document whose name start with the letters “qt”. All these files are property of the author Jörg Kuthe. The author is represented by QT software GmbH, Germany, called QT followingly, which is authorized to allocate use rights to qtXLS. The copyright remains at the qtXLS software and this documentation with the author.

§2. Licensee and Licence File and Licence Routine

The licensee is the user of the qtXLS software and the buyer of a licence, which he has obtained from QT in the course of the purchase of the qtXLS licence. The licence file (format L####-######.lic, with # = digit 0 through 9) and the licence routine (in the file

qtSetLicence_####-######.f90 with # = digits from 0 through 9) contain identifying data of the licensee.

§3. Licence and Passing on of Components of the qtXLS Software

The licence consists of the right of the licensee to use the qtXLS software for the development of programs, i.e. thus to create executable files whose name ends on .exe. The executable file created with qtXLS may neither have the function of a compiler nor that one of a linker.
In addition, the licence covers the right to the passing on of the qtXLS.dll as well as the licence file (L####-######.lic, if supplied) of the licensee together with programs (.exe files) which call qtXLS routines. The passing on of any other file of the qtXLS software is not allowed.
When passing on of programs based on qtXLS, the licensee has to point out to the user of the qtXLS based programs the ownership at the qtXLS.dll by the following text:

"***The file qtXLS.dll is property of Jörg Kuthe, represented by QT software GmbH, Germany. The use of the qtXLS.dll is permitted only together with the <program name of the qtXLS licensee>.exe provided by the <qtXLS licensee>.*** " Those words put in pointed brackets have to be replaced by the name of the licensee and his qtXLS based program.

§4. Transfer and Assignment of the Licence

The licensee cannot be represented by another person. This excludes particularly the rental of the qtXLS software. The licensee may sell the use licence if he reports the disposal or assignment of the licence to another licensee to QT in writing. The assignment must include the confirmation, that the selling licensee gives up his rights at the qtXLS software. The new licensee must agree to these licence conditions in writing and deliver to QT those data which are necessary for the preparation of a new licence file.

§5. Warranty

QT ensures the function ability of the qtXLS software for the period of 2 years after acquisition of the licence. In the fault case it is up to QT either to refund the paid price to the licensee or to fix the reported error in the qtXLS software. If the money paid for qtXLS is refunded, the licensee looses the right to use the software qtXLS. Complaints or faults have to be verified by the licensee by a program he provides.

§6. Use Risk and Limitations of Liability

The licensee uses the software qtXLS on risk of his own. QT is liable in maximum amount of the paid price for the qtXLS software.

§7. Declaration of Consent

The licensee gives his agreement to these conditions by the acquisition of the licence.

9. Other Notes

The author and QT software GmbH acknowledge the rights of the owners at the brand names, trademarks and products named in this document:
Excel is a trademark of Microsoft Corporation, U.S.A..
Windows is a trademark of Microsoft Corporation, U.S.A..

“ProFortran for Windows” is a product of Absoft Corporation, U.S.A..
“Compaq Visual Fortran” is a product of Hewlett-Packard Company, U.S.A..
“Intel Visual Fortran” is a product of Intel Corporation, U.S.A..
“Lahey/Fujitsu Fortran 95 for Windows” is a product of Lahey Computer Systems, Inc., U.S.A..
“Salford FTN95” is a product of Salford Software Ltd., U.K..
“Excel”, “Visual C++”, “Visual Studio 2003” and “Visual Basic” are products of Microsoft Corporation, U.S.A..

Index

M

N

O

P

Q

R