### workbookProtection (Workbook Protection)

This element specifies options for protecting data in the workbook. Applications may use [workbook](workbook.docx) [protection](protection.docx) to prevent anyone from accidentally changing, moving, or deleting important data. This [protection](protection.docx) may be ignored by applications which choose not to support this optional [protection](protection.docx) mechanism.

[Note: Worksheet or [workbook](workbook.docx) element [protection](protection.docx) should not be confused with file security. It is not meant to make your [workbook](workbook.docx) safe from unintentional modification, and cannot protect it from malicious modification. end note]

|  |
| --- |
| Parent Elements |
| [workbook](workbook.docx) (§) |

|  |  |
| --- | --- |
| Attributes | Description |
| lockRevision (Lock Revisions) | Specifies a boolean value that indicates whether the [workbook](workbook.docx) is locked for revisions.  The possible values for this attribute are defined by the XML [Schema](Schema.docx) boolean datatype. |
| lockStructure (Lock Structure) | Specifies a boolean value that indicates whether structure of [workbook](workbook.docx) is locked.  A value of on, 1, or true indicates the structure of the [workbook](workbook.docx) is locked. [Worksheets](Worksheets.docx) in the [workbook](workbook.docx) can't be moved, deleted, hidden, unhidden, or renamed, and new worksheets can't be inserted.  A value of off, 0, or false indicates the structure of the [workbook](workbook.docx) is not locked.  The default value for this attribute is false.  The possible values for this attribute are defined by the XML [Schema](Schema.docx) boolean datatype. |
| lockWindows (Lock Windows) | Specifies a boolean value that indicates whether the windows that comprise the [workbook](workbook.docx) are locked.  A value of on, 1, or true indicates the [workbook](workbook.docx) windows are locked. Windows are the same size and position each time the [workbook](workbook.docx) is opened.  A value of off, 0, or false indicates the [workbook](workbook.docx) windows are not locked.  The default value for this attribute is false.  The possible values for this attribute are defined by the XML [Schema](Schema.docx) boolean datatype. |
| revisionsPassword (Revisions Password) | Specifies the hash of the password required for unlocking [revisions](revisions.docx) in this workbook. The hash is generated from an 8-bit wide character. 16-bit Unicode characters must be converted down to 8 bits before the hash is computed, using the following logic:  For SpreadsheetML password hash purposes, Unicode UTF-16 input code points are converted to an “ansi” single or double byte code [page](page.docx) from the following list:   |  |  |  | | --- | --- | --- | | 874 | windows-874 | ANSI/OEM Thai (same as 28605, ISO 8859-15); Thai (Windows) | | 932 | shift\_jis | ANSI/OEM Japanese; Japanese (Shift-JIS) | | 936 | gb2312 | ANSI/OEM Simplified Chinese (PRC, Singapore); Chinese Simplified (GB2312) | | 949 | ks\_c\_5601-1987 | ANSI/OEM Korean (Unified Hangul Code) | | 950 | big5 | ANSI/OEM Traditional Chinese (Taiwan; Hong Kong SAR, PRC); Chinese Traditional (Big5) | | 1250 | windows-1250 | ANSI Central European; Central European (Windows) | | 1251 | windows-1251 | ANSI Cyrillic; Cyrillic (Windows) | | 1252 | windows-1252 | ANSI Latin 1; Western European (Windows) | | 1253 | windows-1253 | ANSI Greek; Greek (Windows) | | 1254 | windows-1254 | ANSI Turkish; Turkish (Windows) | | 1255 | windows-1255 | ANSI Hebrew; Hebrew (Windows) | | 1256 | windows-1256 | ANSI Arabic; Arabic (Windows) | | 1257 | windows-1257 | ANSI Baltic; Baltic (Windows) | | 1258 | windows-1258 | ANSI/OEM Vietnamese; Vietnamese (Windows) |   Code points with no representation in the target code [page](page.docx) are replaced with Unicode character 0x3f (?).  The necessary mapping [tables](tables.docx) can be found at the following location: http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WindowsBestFit/ .  Code [pages](pages.docx) 932, 936, 949, and 950 are “Double Byte” code pages. The remainder of the “ANSI” code [pages](pages.docx) supported by windows are “Single Byte” code pages.  For single byte code [pages](pages.docx) each Unicode code point is replaced by a single byte or 0x3f if an appropriate character doesn’t exist in the code page.  For double byte code [pages](pages.docx), each Unicode code point is replaced by either a single byte, or a two byte sequence, depending on the input character, or 0x3f if an appropriate character doesn’t exist in the code page. In our [tables](tables.docx) the target is a single byte sequence if the most significant byte is 0x00, otherwise it is a double byte sequence, with the lead byte being the most significant byte.  To convert, first check if conversion is being done to a single or double byte code [page](page.docx) and load the appropriate WCTABLE code [page](page.docx) table.  For each input WCHAR, look up the code point in the WCTABLE. There are 3 possibilities: Not found, single byte, or double byte.  If the input WCHAR is not found, append 0x3f and continue to the next WCHAR.  If the result is a single byte, check to make sure the entry in the MBTABLE matches the input. If it matches, append the single byte to the output. If it does not match, append 0x3f to the output.  If the result is a double byte,check to make sure the entry in the DBCSENTRY [table](table.docx) for the appropriate lead byte matches the input WCHAR. If it matches, append the lead byte and trail byte to the output. If it does not match, append 0x3f to the output.  The following pueudocode describes how this conversion should be done:  int WideCharToMultiByte(WCHAR\* wszInput, byte\* szOutput) {  // Remember output start so we can return length  byte\* szOutputStart = szOutput;   // Ask the system for the current ANSI code [page](page.docx), which  // on windows is a system setting.  int iCodePage = GetCurrentAnsiCodePage();  // Load Code Page [Tables](Tables.docx)  // This will depend on how the code [pages](pages.docx) are represented on  // the target machine. TABLECLASS represents some abstract  // representation of this structure here.  TABLECLASS pTables = LoadCodePageTables(iCodePage);  bool bDoubleByte = false;  if (iCodePage == 932 ||   iCodePage == 936 ||  iCodePage == 949 ||  iCodePage == 950)  bDoubleByte = true;   while (\*wszInput != 0)  {  if (bDoubleByte)   szOutput = AppendDoubleByte(pTables, \*wszInput, szOutput);  else  szOutput = AppendSingleByte(pTables, \*wszInput, szOutput);   // Read next input WCHAR  wszInput++;  }   // Null terminate the output  \*szOutput = 0;   // Return output length  return szOutput – szOutputStart; }  byte\* AppendSingleByte(TABLECLASS pTables, WCHAR wcIn, byte\* szOutput) {  // Look up byte that we want to append.  byte bOut = pTables->LookUpSingleByte(wcIn);   // Make sure that bOut matches the input, otherwise use ?  // (ie: no best fit behavior allowed)  if (wcIn != pTables->LookUpWideChar(bOut))  bOut = 0x3f;   \*szOutput = bOut;  szOutput++;  return szOutput; }  byte\* AppendDoubleByte(TABLECLASS pTables, WCHAR wcIn, byte\* szOutput) {  // Look up bytes that we want to append.  UINT16 bytesOut = pTables->LookUpDoubleByte(wcIn);   // See if it is a single or double byte sequence  if (bytesOut & 0xFF00)  {  // It is a double byte sequence  // Make sure that bytesOut matches the input, otherwise use ?  // (ie: no best fit behavior allowed)  if (wcIn != pTables->LookUpWideChar(bytesOut))  {  // Use ?, it will be added below  bytesOut = 0x003f;  }  else  {  // It matched, use the lead byte we found  // trail byte will be added below  \*szOutput = bytesOut >> 8;  szOutput++;  }  else  {  // It is a single byte sequence  // Make sure that bytesOut matches the input, otherwise use ?  // (ie: no best fit behavior allowed)  if (wcIn != pTables->LookUpWideChar(bytesOut & 0xFF))  bytesOut = 0x003f;  }   // Add the single or trail byte  \*szOutput = bytesOut & 0xFF;  szOutput++;   return szOutput; }  class pTables {  // Construction depends on how you choose to store & load the  // [table](table.docx) files   byte LookUpSingleByte(WCHAR wcIn)  {  // How you access the [table](table.docx) depends on your storage mechanism.  // Look up the line in WCTABLE where the first column matches wcIn,  // and then return the byte value from the second column.  if (exists WCTABLE{wcIn})  return WCTABLE{wcIn}.SecondColumn;   // If it doesn’t exist, return ?  return 0x3f;  }   UINT16 LookUpDoubleByte(WCHAR wcIn)  {  // How you access the [table](table.docx) depends on your storage mechanism.  // Look up the line in WCTABLE where the first column matches wcIn,  // and then return the double byte value from the second column.  if (exists WCTABLE{wcIn})  return WCTABLE{wcIn}.SecondColumn;   // If it doesn’t exist, return ?  return 0x003f;  }   // Overload that looks up wide chars from single byte code points.  WCHAR LookUpWideChar(byte bIn)  {  // How you access the [table](table.docx) depends on your storage mechanism.  // Look up the line in MBTABLE where the first column matches bIn,  // and then return the WCHAR value from the second column.  if (exists MBTABLE{bIn})  return MBTABLE{bIn}.SecondColumn;   // If it doesn’t exist, return ?  return 0x003f;   }   // Overload that looks up wide chars from double byte code points  WCHAR LookUpWideChar(UINT16 bytesIn)  {  // How you access the [table](table.docx) depends on your storage mechanism.  // First find the DBCSTABLE where the LeadByte matches  // the lead (most significant) input byte.  if (exists DBCSTABLE{bytesIn >> 8))  {  DbcsTable = DBCSTABLE{bytesIn >> 8);   // Look up the line in DbcsTable where the first column  // matches the input trail (least significant) byte,  // and then return the WCHAR value from the second column.  if (exists DbcsTable{bytesIn & 0xFF})  return DbcsTable{bytesIn & 0xFF}.SecondColumn;  }   // Either the lead byte [table](table.docx) or specific trail byte  // doesn’t exist in the [table](table.docx), return ?  return 0x003f;   } }  The resulting value is hashed using the algorithm defined below.  [Note: An example algorithm to hash the resulting single-byte user input into the value stored is as follows:  // Function Input:  // szPassword: NULL terminated C-Style string  // cchPassword: The number of characters in szPassword (not including the NULL terminator)  WORD GetPasswordHash(const [CHAR](CHAR.docx) \*szPassword, int cchPassword) {  WORD wPasswordHash;  const [CHAR](CHAR.docx) \*pch;    wPasswordHash = 0;    if (cchPassword > 0)  {  pch = &szPassword[cchPassword];  while (pch-- != szPassword)  {  wPasswordHash = ((wPasswordHash >> 14) & 0x01) | ((wPasswordHash << 1) & 0x7fff);  wPasswordHash ^= \*pch;  }  wPasswordHash ^= (0x8000 | ('N' << 8) | 'K');  }    return(wPasswordHash);  }  end note]  The possible values for this attribute are defined by the [ST\_UnsignedShortHex](ST_UnsignedShortHex.docx) simple type (§). |
| workbookPassword (Workbook Password) | Specifies the hash of the password required for unlocking [revisions](revisions.docx) in this workbook. The hash is generated from an 8-bit wide character. 16-bit Unicode characters must be converted down to 8 bits before the hash is computed, using the following logic:  For SpreadsheetML password hash purposes, Unicode UTF-16 input code points are converted to an “ansi” single or double byte code [page](page.docx) using the logic defined in the preceding revisionsPassword attribute.  The resulting value is hashed using the algorithm defined below.  [Note: An example algorithm to hash the user input into the value stored is as follows:  // Function Input:  // szPassword: NULL terminated C-Style string  // cchPassword: The number of characters in szPassword (not including the NULL terminator)  WORD GetPasswordHash(const [CHAR](CHAR.docx) \*szPassword, int cchPassword) {  WORD wPasswordHash;  const [CHAR](CHAR.docx) \*pch;    wPasswordHash = 0;    if (cchPassword > 0)  {  pch = &szPassword[cchPassword];  while (pch-- != szPassword)  {  wPasswordHash = ((wPasswordHash >> 14) & 0x01) | ((wPasswordHash << 1) & 0x7fff);  wPasswordHash ^= \*pch;  }  wPasswordHash ^= (0x8000 | ('N' << 8) | 'K');  }    return(wPasswordHash);  }  end note]  The possible values for this attribute are defined by the [ST\_UnsignedShortHex](ST_UnsignedShortHex.docx) simple type (§). |

The following XML [Schema](Schema.docx) fragment defines the contents of this element:

<complexType [name](name.docx)="CT\_WorkbookProtection">

<attribute [name](name.docx)="workbookPassword" type="[ST\_UnsignedShortHex](ST_UnsignedShortHex.docx)" use="optional"/>

<attribute [name](name.docx)="revisionsPassword" type="[ST\_UnsignedShortHex](ST_UnsignedShortHex.docx)" use="optional"/>

<attribute name="lockStructure" type="xsd:boolean" use="optional" default="false"/>

<attribute name="lockWindows" type="xsd:boolean" use="optional" default="false"/>

<attribute name="lockRevision" type="xsd:boolean" use="optional" default="false"/>

</complexType>