```
Attribute VB_Name = "BitBoard32"
Option Explicit

' bitboard 32 bit with 2 long variables
Public Declare Sub CopyMemory _
                Lib "kernel32" _
                Alias "RtlMoveMemory" (ByVal Destination As Long, _
                                       ByVal Source As Long, _
                                       ByVal Length As Long)



Public Const MIN_INTEGER  As Integer = -32768
Public Const MAX_INTEGER  As Integer = 32767

Public Const BitL_0 As Long = &H1&
Public Const BitL_1 As Long = &H2&
Public Const BitL_2 As Long = &H4&
Public Const BitL_3 As Long = &H8&
Public Const BitL_4 As Long = &H10&
Public Const BitL_5 As Long = &H20&
Public Const BitL_6 As Long = &H40&
Public Const BitL_7 As Long = &H80&
Public Const BitL_8 As Long = &H100&
Public Const BitL_9 As Long = &H200&
Public Const BitL_10 As Long = &H400&
Public Const BitL_11 As Long = &H800&
Public Const BitL_12 As Long = &H1000&
Public Const BitL_13 As Long = &H2000&
Public Const BitL_14 As Long = &H4000&
Public Const BitL_15 As Long = &H8000&
Public Const BitL_16 As Long = &H10000
Public Const BitL_17 As Long = &H20000
Public Const BitL_18 As Long = &H40000
Public Const BitL_19 As Long = &H80000
Public Const BitL_20 As Long = &H100000
Public Const BitL_21 As Long = &H200000
Public Const BitL_22 As Long = &H400000
Public Const BitL_23 As Long = &H800000
Public Const BitL_24 As Long = &H1000000
Public Const BitL_25 As Long = &H2000000
Public Const BitL_26 As Long = &H4000000
Public Const BitL_27 As Long = &H8000000
Public Const BitL_28 As Long = &H10000000
Public Const BitL_29 As Long = &H20000000
Public Const BitL_30 As Long = &H40000000
Public Const BitL_31 As Long = &H80000000

Public Const RANK1_L = BitL_0 Or BitL_1 Or BitL_2 Or BitL_3 Or BitL_4 Or BitL_5 Or BitL_6 Or BitL_7
Public Const RANK2_L = BitL_8 Or BitL_9 Or BitL_10 Or BitL_11 Or BitL_12 Or BitL_13 Or BitL_14 Or BitL_15
Public Const RANK3_L = BitL_16 Or BitL_17 Or BitL_18 Or BitL_19 Or BitL_20 Or BitL_21 Or BitL_22 Or BitL_23
Public Const RANK4_L = BitL_24 Or BitL_25 Or BitL_26 Or BitL_27 Or BitL_28 Or BitL_29 Or BitL_30 Or BitL_31


Public Type TBit64   ' emulate 64 bit, use 4x16 bit (positive values only)
  i0 As Long
  i1 As Long
End Type

Public Type TInt16x2
  i0 As Integer
  i1 As Integer
End Type

```

```vb
Public Type TInt16x4
  i0 As Integer
  i1 As Integer
  i2 As Integer
  i3 As Integer
End Type

Public Type TByte8x8
  i0 As Byte
  i1 As Byte
  i2 As Byte
  i3 As Byte
  i4 As Byte
  i5 As Byte
  i6 As Byte
  i7 As Byte
End Type

'------------------------------------------------------------------------------------------

Public FILEA_BB As TBit64, FILEB_BB As TBit64, FILEC_BB As TBit64, FILED_BB As TBit64,
 FILEE_BB As TBit64, FILEF_BB As TBit64, FILEG_BB As TBit64, FILEH_BB As TBit64
Public RANK1_BB As TBit64, RANK2_BB As TBit64, RANK3_BB As TBit64, RANK4_BB As TBit64,
 RANK5_BB As TBit64, RANK6_BB As TBit64, RANK7_BB As TBit64, RANK8_BB As TBit64

Public Bit32Pos(31) As Long
Public Pop16Cnt(MIN_INTEGER To MAX_INTEGER) As Byte ' Max -int to +int
Public Int16x4 As TInt16x4
Public Int16x2 As TInt16x2
Public Byte8x8 As TByte8x8

Public Bit8Pos(7) As Integer

Public PiecesBB(WCOL, PT_ALL_PIECES) As TBit64, AllPiecesBB As TBit64, PiecesByPtBB(
PT_ALL_PIECES) As TBit64, ColBB(WCOL) As TBit64, AttackedByBB(WCOL, PT_ALL_PIECES) As
TBit64, AttackedBy2BB(WCOL) As TBit64
Public SquareBB(MAX_BOARD) As TBit64
Public EmptyBB As TBit64
Public FileBB(8) As TBit64
Public RankBB(8) As TBit64
Public AdjacentFilesBB(8) As TBit64
Public ForwardRanksBB(WCOL, 8) As TBit64
Public ForwardFileBB(WCOL, MAX_BOARD) As TBit64
Public PawnAttackSpanBB(WCOL, MAX_BOARD) As TBit64
Public PawnAttackSpanAllBB(WCOL) As TBit64
Public PassedPawnMaskBB(WCOL, MAX_BOARD) As TBit64
Public OutpostRanksBB(WCOL) As TBit64
Public SqToBit(MAX_BOARD) As Long
Public BitToSq(63) As Long
Public PawnAttacksFromSqBB(WCOL, SQ_H8) As TBit64
Public PseudoAttacksFromSqBB(PIECE_TYPE_NB, MAX_BOARD) As TBit64
Public AttackFromToBB(MAX_BOARD, MAX_BOARD) As TBit64
Public BetweenBB(MAX_BOARD, MAX_BOARD) As TBit64
Public KingRingBB(WCOL) As TBit64
Public LowRanksBB(WCOL) As TBit64
Public CampBB(WCOL) As TBit64
Public CenterBB As TBit64
Public CenterFilesBB As TBit64
Public DarkSquaresBB As TBit64
Public LSB16(MIN_INTEGER To MAX_INTEGER) As Integer
Public RSB16(MIN_INTEGER To MAX_INTEGER) As Integer

Public BB0 As TBit64, BB1 As TBit64

'------------------------------------------------------------------


Public Sub Init32BitBoards()
```

```
    Dim i As Long, j As Long, k As Long, SqBB As Long

    For i = 0 To 7: Bit8Pos(i) = 2 ^ i: Next
    For i = 0 To 31: Bit32Pos(i) = BitMask32(i): Next

    For j = MIN_INTEGER To MAX_INTEGER
      Pop16Cnt(j) = Pop16CountFkt(j)
      LSB16(j) = -1
      For i = 0 To 15
         If CBool(j And Bit32Pos(i)) Then LSB16(j) = i: Exit For
      Next
      RSB16(j) = -1
      For i = 15 To 0 Step -1
         If CBool(j And Bit32Pos(i)) Then RSB16(j) = i: Exit For
      Next
    Next

    SqBB = 0
    For i = 0 To 119
      SqToBit(i) = -1
      If Board(i) <> FRAME Then
          SqToBit(i) = SqBB
          BitToSq(SqBB) = i

          '--- set ranks
          Select Case Rank(i)
          Case 1: SetBit64 RANK1_BB, SqBB
          Case 2: SetBit64 RANK2_BB, SqBB
          Case 3: SetBit64 RANK3_BB, SqBB
          Case 4: SetBit64 RANK4_BB, SqBB
          Case 5: SetBit64 RANK5_BB, SqBB
          Case 6: SetBit64 RANK6_BB, SqBB
          Case 7: SetBit64 RANK7_BB, SqBB
          Case 8: SetBit64 RANK8_BB, SqBB
          End Select

          '--- set Files
          Select Case File(i)
          Case 1: SetBit64 FILEA_BB, SqBB
          Case 2: SetBit64 FILEB_BB, SqBB
          Case 3: SetBit64 FILEC_BB, SqBB
          Case 4: SetBit64 FILED_BB, SqBB
          Case 5: SetBit64 FILEE_BB, SqBB
          Case 6: SetBit64 FILEF_BB, SqBB
          Case 7: SetBit64 FILEG_BB, SqBB
          Case 8: SetBit64 FILEH_BB, SqBB
          End Select

        SetBit64 SquareBB(i), SqBB
        If ColorSq(i) = BCOL Then SetBit64 DarkSquaresBB, SqBB
        '
        SqBB = SqBB + 1
      End If
    Next i

    FileBB(FILE_A) = FILEA_BB
    FileBB(FILE_B) = FILEB_BB
    FileBB(FILE_C) = FILEC_BB
    FileBB(FILE_D) = FILED_BB
    FileBB(FILE_E) = FILEE_BB
    FileBB(FILE_F) = FILEF_BB
    FileBB(FILE_G) = FILEG_BB
    FileBB(FILE_H) = FILEH_BB

    RankBB(1) = RANK1_BB
    RankBB(2) = RANK2_BB
    RankBB(3) = RANK3_BB
    RankBB(4) = RANK4_BB
```

```
 RankBB(5) = RANK5_BB
 RankBB(6) = RANK6_BB
 RankBB(7) = RANK7_BB
 RankBB(8) = RANK8_BB

 OR64 LowRanksBB(WCOL), RANK2_BB, RANK3_BB
 OR64 LowRanksBB(BCOL), RANK6_BB, RANK7_BB
 OR64 CenterFilesBB, FILED_BB, FILEE_BB
 OR64 BB0, RANK4_BB, RANK5_BB
 AND64 CenterBB, CenterFilesBB, BB0
 OR64 OutpostRanksBB(WCOL), RANK4_BB, RANK5_BB: OR64 OutpostRanksBB(WCOL), OutpostRanksBB(WCOL), RANK6_BB
 OR64 OutpostRanksBB(BCOL), RANK3_BB, RANK4_BB: OR64 OutpostRanksBB(BCOL), OutpostRanksBB(BCOL), RANK5_BB

 AdjacentFilesBB(FILE_A) = FILEB_BB
 OR64 AdjacentFilesBB(FILE_B), FILEA_BB, FILEC_BB
 OR64 AdjacentFilesBB(FILE_C), FILEA_BB, FILED_BB
 OR64 AdjacentFilesBB(FILE_D), FILEA_BB, FILEE_BB
 OR64 AdjacentFilesBB(FILE_E), FILEA_BB, FILEF_BB
 OR64 AdjacentFilesBB(FILE_F), FILEA_BB, FILEG_BB
 OR64 AdjacentFilesBB(FILE_G), FILEA_BB, FILEH_BB
 AdjacentFilesBB(FILE_H) = FILEG_BB

 ForwardRanksBB(WCOL, 7) = RANK8_BB
 OR64 ForwardRanksBB(WCOL, 6), ForwardRanksBB(WCOL, 7), RANK7_BB
 OR64 ForwardRanksBB(WCOL, 5), ForwardRanksBB(WCOL, 6), RANK6_BB
 OR64 ForwardRanksBB(WCOL, 4), ForwardRanksBB(WCOL, 5), RANK5_BB
 OR64 ForwardRanksBB(WCOL, 3), ForwardRanksBB(WCOL, 4), RANK4_BB
 OR64 ForwardRanksBB(WCOL, 2), ForwardRanksBB(WCOL, 3), RANK3_BB
 OR64 ForwardRanksBB(WCOL, 1), ForwardRanksBB(WCOL, 2), RANK2_BB

 ForwardRanksBB(BCOL, 2) = RANK1_BB
 OR64 ForwardRanksBB(BCOL, 3), ForwardRanksBB(BCOL, 2), RANK2_BB
 OR64 ForwardRanksBB(BCOL, 4), ForwardRanksBB(BCOL, 3), RANK3_BB
 OR64 ForwardRanksBB(BCOL, 5), ForwardRanksBB(BCOL, 4), RANK4_BB
 OR64 ForwardRanksBB(BCOL, 6), ForwardRanksBB(BCOL, 5), RANK5_BB
 OR64 ForwardRanksBB(BCOL, 7), ForwardRanksBB(BCOL, 6), RANK6_BB
 OR64 ForwardRanksBB(BCOL, 8), ForwardRanksBB(BCOL, 7), RANK7_BB

 CampBB(WCOL) = ForwardRanksBB(BCOL, 6)
 CampBB(BCOL) = ForwardRanksBB(WCOL, 3)

 ' Init SqFrom attacks
   Dim d As Long, Col As Long, Offset As Long

For Col = BCOL To WCOL
 For i = SQ_A1 To SQ_H8
   If Board(i) <> FRAME Then
       ' Pawn attacks
       If Col = WCOL Then j = i + 9 Else j = i - 9
       If Board(j) <> FRAME Then
          SetBit64 PawnAttacksFromSqBB(Col, i), SqToBit(j)
       End If
       If Col = WCOL Then j = i + 11 Else j = i - 11
       If Board(j) <> FRAME Then
          SetBit64 PawnAttacksFromSqBB(Col, i), SqToBit(j)
       End If

       AND64 ForwardFileBB(Col, i), ForwardRanksBB(Col, Rank(i)), FileBB(File(i))
       AND64 PawnAttackSpanBB(Col, i), ForwardRanksBB(Col, Rank(i)), AdjacentFilesBB(File(i))
       OR64 PassedPawnMaskBB(Col, i), ForwardFileBB(Col, i), PawnAttackSpanBB(Col, i)

       If Col = WCOL Then ' same for black
         ' King/Knight attacks
         For d = 0 To 7
           Offset = QueenOffsets(d)
```

```
            j = i + Offset
            If Board(j) <> FRAME Then
              SetBit64 PseudoAttacksFromSqBB(PT_KING, i), SqToBit(j)  ' King

              '
              Do While Board(j) <> FRAME
                SetBit64 PseudoAttacksFromSqBB(PT_QUEEN, i), SqToBit(j)  ' Queen

                If Board(j) <> FRAME Then
                  If j <> i + Offset Then
                    SetBit64 BetweenBB(i, j), SqToBit(j - Offset)   ' between 2 squares, current square
                    SetOR64 BetweenBB(i, j), BetweenBB(i, j - Offset)  ' previous squares in line
                  End If
                  AttackFromToBB(i, j) = BetweenBB(i, j): SetBit64 AttackFromToBB(i, j), SqToBit(j) ' includes target square
                End If

                If d < 4 Then
                  SetBit64 PseudoAttacksFromSqBB(PT_ROOK, i), SqToBit(j)  ' Rook
                Else
                  SetBit64 PseudoAttacksFromSqBB(PT_BISHOP, i), SqToBit(j)  ' Bishop
                End If
                j = j + Offset
              Loop
            End If

            '---
            j = i + KnightOffsets(d)
            If Board(j) <> FRAME Then
              SetBit64 PseudoAttacksFromSqBB(PT_KNIGHT, i), SqToBit(j) ' Knight
            End If
          Next d
        End If
     End If
   Next i
 Next Col
End Sub

Function BitMask32(ByVal BitPos As Long) As Long ' 32 bit
  'If BitPos < 0 Or BitPos > 31 Then Err.Raise 6 ' overflow
  If BitPos < 31 Then
   BitMask32 = 2 ^ BitPos
  Else
   BitMask32 = BitL_31
  End If
End Function

Public Function Pop16CountFkt(ByVal x As Long) As Long
  ' for positive values only
  Pop16CountFkt = 0: If x = 0 Then Exit Function
  If x < 0 Then Pop16CountFkt = Pop16CountFkt + 1: x = x And Not &H8000
  Do While x > 0
    Pop16CountFkt = Pop16CountFkt + 1: x = x And (x - 1)
  Loop
End Function

Public Sub AND64(Result As TBit64, Op1 As TBit64, Op2 As TBit64)
  Result.i0 = Op1.i0 And Op2.i0: Result.i1 = Op1.i1 And Op2.i1
End Sub

Public Sub SetAND64(Op1 As TBit64, Op2 As TBit64) ' returns Op1
  Op1.i0 = Op1.i0 And Op2.i0: Op1.i1 = Op1.i1 And Op2.i1
End Sub

Public Sub SetANDNOT64(Op1 As TBit64, Op2 As TBit64) ' returns Op1
  Op1.i0 = Op1.i0 And Not Op2.i0: Op1.i1 = Op1.i1 And Not Op2.i1
```

```
End Sub

Public Sub OR64(Result As TBit64, Op1 As TBit64, Op2 As TBit64)
  Result.i0 = Op1.i0 Or Op2.i0: Result.i1 = Op1.i1 Or Op2.i1
End Sub

Public Sub SetOR64(Op1 As TBit64, Op2 As TBit64) ' returns Op1
  Op1.i0 = Op1.i0 Or Op2.i0: Op1.i1 = Op1.i1 Or Op2.i1
End Sub

Public Sub XOr64(Result As TBit64, Op1 As TBit64, Op2 As TBit64)
  Result.i0 = Op1.i0 Xor Op2.i0: Result.i1 = Op1.i1 Xor Op2.i1
End Sub

Public Sub ANDNOT64(Result As TBit64, Op1 As TBit64, Op2 As TBit64)
  Result.i0 = Op1.i0 And Not Op2.i0: Result.i1 = Op1.i1 And Not Op2.i1
End Sub

Public Sub SetNOT64(Result As TBit64, Op1 As TBit64)
  Result.i0 = Not Op1.i0: Result.i1 = Not Op1.i1
End Sub

Public Sub Set64(Result As TBit64, Op1 As TBit64)
  Result.i0 = Op1.i0: Result.i1 = Op1.i1  ' much faster then  Result=Op1 !!!!
End Sub

Public Function EQUAL64(Op1 As TBit64, Op2 As TBit64) As Boolean
  If Op1.i0 = Op2.i0 Then
    If Op1.i1 = Op2.i1 Then EQUAL64 = True Else EQUAL64 = False
  Else
    EQUAL64 = False
  End If
End Function

Public Sub Clear64(Op1 As TBit64)
  Op1.i0 = 0: Op1.i1 = 0
End Sub

Public Function ShiftDown64(Op1 As TBit64) As TBit64
   ' shift right 8 bits
' LSet Byte8x8 = Op1
' Byte8x8.i0 = Byte8x8.i1
' Byte8x8.i1 = Byte8x8.i2
' Byte8x8.i2 = Byte8x8.i3
' Byte8x8.i3 = Byte8x8.i4
' Byte8x8.i4 = Byte8x8.i5
' Byte8x8.i5 = Byte8x8.i6
' Byte8x8.i6 = Byte8x8.i7
' Byte8x8.i7 = 0
' LSet ShiftDown64 = Byte8x8

   ' i1
   If Op1.i1 And BitL_31 Then
     ShiftDown64.i1 = (((Op1.i1 And Not BitL_31) \ &H100&) Or BitL_23) And Not RANK4_L
          ' shift 8 bits down (=&H100& ),remove rank 4 and add sign bit 31 as bit 23
   Else
     ShiftDown64.i1 = (Op1.i1 \ &H100&) And Not RANK4_L
   End If

   ' Copy RANK5 to RANK4 > copy to i0 bits 0-6 (= &H7F&) of rank1 and shift 24 bits (=&H1000000) up
   If Op1.i1 And BitL_7 Then
     ShiftDown64.i0 = ((Op1.i1 And &H7F&) * &H1000000) Or BitL_31 ' copy bit 7 from I1 to I0
     sign bit
   Else
     ShiftDown64.i0 = (Op1.i1 And &H7F&) * &H1000000
   End If

   ' i0
   If Op1.i0 And BitL_31 Then
     ShiftDown64.i0 = ShiftDown64.i0 Or ((((Op1.i0 And Not BitL_31) \ &H100&) Or BitL_23) And Not RANK4_L)   ' shift 8 bits down (=&H100& ),remove rank 4 and add sign bit 31 as bit 23
   Else
     ShiftDown64.i0 = ShiftDown64.i0 Or ((Op1.i0 \ &H100&) And Not RANK4_L)
   End If
   'ShowLBB ShiftDown64
End Function

Public Function ShiftUp64(Op1 As TBit64) As TBit64
  ' shift left 8 bits
' LSet Byte8x8 = Op1
' Byte8x8.i7 = Byte8x8.i6
' Byte8x8.i6 = Byte8x8.i5
' Byte8x8.i5 = Byte8x8.i4
' Byte8x8.i4 = Byte8x8.i3
' Byte8x8.i3 = Byte8x8.i2
' Byte8x8.i2 = Byte8x8.i1
' Byte8x8.i1 = Byte8x8.i0
' Byte8x8.i0 = 0
' LSet ShiftUp64 = Byte8x8

  ' i0
  If Op1.i0 And BitL_23 Then
    ShiftUp64.i0 = ((Op1.i0 And Not RANK4_L And Not BitL_23) * &H100& Or BitL_31)
  Else
    ShiftUp64.i0 = (Op1.i0 And Not RANK4_L) * &H100&
  End If
  ' i1
  If Op1.i1 And BitL_23 Then
    ShiftUp64.i1 = ((Op1.i1 And Not RANK4_L And Not BitL_23) * &H100&) Or BitL_31
  Else
    ShiftUp64.i1 = (Op1.i1 And Not RANK4_L) * &H100&
  End If
  ' Copy RANK5 to RANK4 > copy to i1 bits 24-30 (= &H7F000000) of rank4  and shift 24 bits (=&H1000000) down
  If Op1.i0 And BitL_31 Then
    ShiftUp64.i1 = ShiftUp64.i1 Or ((Op1.i0 And &H7F000000) \ &H1000000) Or BitL_7 ' copy sign bit 31 from I0 to I1 bit 7
  Else
    ShiftUp64.i1 = ShiftUp64.i1 Or ((Op1.i0 And &H7F000000) \ &H1000000)
  End If
End Function

Public Function ShiftLeft64(Op1 As TBit64) As TBit64
  ShiftLeft64.i0 = Op1.i0 And Not FILEA_BB.i0: ShiftLeft64.i1 = Op1.i1 And Not FILEA_BB.i1 ' remove file A
  ShiftLeft64.i0 = ShiftLeft64.i0 \ &H2& And Not BitL_31
  ShiftLeft64.i1 = ShiftLeft64.i1 \ &H2& And Not BitL_31
End Function

Public Function ShiftRight64(Op1 As TBit64) As TBit64
  ShiftRight64.i0 = Op1.i0 And Not FILEH_BB.i0: ShiftRight64.i1 = Op1.i1 And Not FILEH_BB.i1 ' remove file H
  If ShiftRight64.i0 And BitL_30 Then
    ShiftRight64.i0 = ((ShiftRight64.i0 And Not BitL_30) * &H2&) Or BitL_31 ' move bit30 to bit31 else overflow
  Else
    ShiftRight64.i0 = (ShiftRight64.i0 And &HFFFFFFFF) * &H2&
  End If
  If ShiftRight64.i1 And BitL_30 Then
    ShiftRight64.i1 = ((ShiftRight64.i1 And Not BitL_30) * &H2&) Or BitL_31
  Else
    ShiftRight64.i1 = (ShiftRight64.i1 And &HFFFFFFFF) * &H2&
  End If
End Function

Public Function ShiftUpOrDown64(ByVal UpDown As Long, Op1 As TBit64) As TBit64
  If UpDown = SQ_UP Then
```

```
      ShiftUpOrDown64 = ShiftUp64(Op1)
    ElseIf UpDown = SQ_DOWN Then
      ShiftUpOrDown64 = ShiftDown64(Op1)
    End If
  End Function


  Public Sub SetBit64(Op1 As TBit64, ByVal BitPos As Long)
    ' bitPos 0 to 63
    Debug.Assert BitPos >= 0 And BitPos < 64
    'If BitPos < 0 Then Exit Sub
    'If BitPos > 63 Then Exit Sub

    If BitPos < 32 Then
      If BitPos = 31 Then Op1.i0 = Op1.i0 Or BitL_31 Else Op1.i0 = Op1.i0 Or Bit32Pos(
      BitPos)
    Else
      BitPos = BitPos - 32
      If BitPos = 31 Then Op1.i1 = Op1.i1 Or BitL_31 Else Op1.i1 = Op1.i1 Or Bit32Pos(
      BitPos)
    End If
  End Sub

  Public Function IsSetBit64(Op1 As TBit64, ByVal BitPos As Long) As Boolean
    ' bitPos 0 to 63
    Debug.Assert BitPos >= 0 And BitPos < 64
    'If BitPos < 0 Then Exit Sub
    'If BitPos > 63 Then Exit Sub

    If BitPos < 32 Then
      IsSetBit64 = CBool(Op1.i0 And Bit32Pos(BitPos))
    Else
      IsSetBit64 = CBool(Op1.i1 And Bit32Pos(BitPos - 32))
    End If
  End Function

  Public Function IsSet64(Op1 As TBit64) As Boolean
    If Op1.i0 <> 0 Then IsSet64 = True: Exit Function
    If Op1.i1 <> 0 Then IsSet64 = True: Exit Function
    IsSet64 = False
  End Function


  Public Sub ClearBit64(Op1 As TBit64, ByVal BitPos As Long)
    ' bitPos 0 to 63
    Debug.Assert BitPos < 64
    If BitPos < 32 Then
      Op1.i0 = Op1.i0 And Not Bit32Pos(BitPos)
    Else
      Op1.i1 = Op1.i1 And Not Bit32Pos(BitPos - 32)
    End If
  End Sub


  Public Function PopCnt64(Op1 As TBit64) As Long
    LSet Int16x4 = Op1
    PopCnt64 = Pop16Cnt(Int16x4.i0) + Pop16Cnt(Int16x4.i1) + Pop16Cnt(Int16x4.i2) +
    Pop16Cnt(Int16x4.i3)
  End Function

  Public Sub ShowBB(bb As TBit64)
   Dim i As Long, s As String
   Debug.Print
   Debug.Print " ------------------"
   s = ""
   For i = 63 To 0 Step -1
     If (i + 1) Mod 8 = 0 And s <> "" Then
       Debug.Print CStr((i + 9) \ 8) & "|" & s & "|"
```

```vb
      s = ""
    End If
    If IsSetBit64(bb, ByVal i) Then s = " X" & s Else s = " ." & s
  Next
  Debug.Print CStr((i + 9) \ 8) & "|" & s & "|"
  Debug.Print " -----------------"
  Debug.Print "   A B C D E F G H"
  Debug.Print

End Sub

Public Function Lsb64(Op1 As TBit64) As Long
 ' returns position of first bit set
 Lsb64 = -1
 If Op1.i0 <> 0 Then
   LSet Int16x4 = Op1
   Lsb64 = LSB16(Int16x4.i0): If Lsb64 >= 0 Then Exit Function
   Lsb64 = LSB16(Int16x4.i1): If Lsb64 >= 0 Then Lsb64 = Lsb64 + 16: Exit Function
 ElseIf Op1.i1 <> 0 Then
   LSet Int16x4 = Op1
   Lsb64 = LSB16(Int16x4.i2): If Lsb64 >= 0 Then Lsb64 = Lsb64 + 32: Exit Function
   Lsb64 = LSB16(Int16x4.i3): If Lsb64 >= 0 Then Lsb64 = Lsb64 + 48
 End If
End Function


Public Function Rsb64(Op1 As TBit64) As Long
 ' returns position of last bit set
 Rsb64 = -1
 If Op1.i1 <> 0 Then
   LSet Int16x4 = Op1
   Rsb64 = RSB16(Int16x4.i3): If Rsb64 >= 0 Then Rsb64 = Rsb64 + 48: Exit Function
   Rsb64 = RSB16(Int16x4.i2): If Rsb64 >= 0 Then Rsb64 = Rsb64 + 32: Exit Function
 ElseIf Op1.i0 <> 0 Then
   LSet Int16x4 = Op1
   Rsb64 = RSB16(Int16x4.i1): If Rsb64 >= 0 Then Rsb64 = Rsb64 + 16: Exit Function
   Rsb64 = RSB16(Int16x4.i0)
 End If
End Function

Public Function PopLsb64(Op1 As TBit64) As Long
  PopLsb64 = Lsb64(Op1)
  If PopLsb64 >= 0 Then ClearBit64 Op1, PopLsb64
End Function


Public Function PawnAttacksBB(ByRef Col As enumColor, Op1 As TBit64) As TBit64
  If Col = WCOL Then
    PawnAttacksBB = ShiftUp64(Op1)
    OR64 PawnAttacksBB, ShiftLeft64(PawnAttacksBB), ShiftRight64(PawnAttacksBB)
  ElseIf Col = BCOL Then
    PawnAttacksBB = ShiftDown64(Op1)
    OR64 PawnAttacksBB, ShiftLeft64(PawnAttacksBB), ShiftRight64(PawnAttacksBB)
  End If
End Function

Public Function AttacksBoardBB(ByVal pt As enumPieceType, ByVal sq As Long) As TBit64
      Dim LastTarget As Long, Target As Long, Offset As Long, d As Long, DirStart As
      Long, DirEnd As Long
      AttacksBoardBB = EmptyBB

      Select Case pt
      Case PT_ROOK: DirStart = 0: DirEnd = 3
      Case PT_BISHOP: DirStart = 4: DirEnd = 7
      Case PT_QUEEN: DirStart = 0: DirEnd = 7
      Case Else
        Exit Function
      End Select

      For d = DirStart To DirEnd
        Offset = QueenOffsets(d): Target = sq + Offset: LastTarget = sq
        Do While Board(Target) <> FRAME
          LastTarget = Target
          If Board(Target) >= NO_PIECE Then Exit Do
          Target = Target + Offset
        Loop
        If sq <> LastTarget Then
          If MaxDistance(sq, LastTarget) = 1 Then
            SetBit64 AttacksBoardBB, SqToBit(LastTarget)
          Else
            ' --- add bitboards for direction
            AttacksBoardBB.i0 = AttacksBoardBB.i0 Or AttackFromToBB(sq, LastTarget).i0
            : AttacksBoardBB.i1 = AttacksBoardBB.i1 Or AttackFromToBB(sq, LastTarget).
            i1
          End If
        End If
      Next
End Function

Public Function AttacksBB(ByVal pt As enumPieceType, ByVal sq As Long, occupied As
TBit64) As TBit64
      Dim LastTarget As Long, Target As Long, Offset As Long, d As Long, DirStart As
      Long, DirEnd As Long

      AttacksBB = EmptyBB

      Select Case pt
      Case PT_ROOK: DirStart = 0: DirEnd = 3
      Case PT_BISHOP: DirStart = 4: DirEnd = 7
      Case PT_QUEEN: DirStart = 0: DirEnd = 7
      Case Else
        Exit Function
      End Select

      For d = DirStart To DirEnd
        Offset = QueenOffsets(d): Target = sq + Offset: LastTarget = sq

        Do While Board(Target) <> FRAME
          LastTarget = Target: If IsSetBit64(occupied, SqToBit(Target)) Then Exit Do
          Target = Target + Offset
        Loop
        If sq <> LastTarget Then SetOR64 AttacksBB, AttackFromToBB(sq, LastTarget) '---
        add bitboards for direction
      Next
End Function

Public Function MoreThanOne(op1BB As TBit64) As Boolean
  MoreThanOne = CBool(PopCnt64(op1BB) > 1)
End Function

Public Function FrontMostSq(Us As enumColor, Op1 As TBit64) As Long
 ' returns first square board position relative for color
 If Us = WCOL Then FrontMostSq = Rsb64(Op1) Else FrontMostSq = Lsb64(Op1)
 If FrontMostSq >= 0 Then FrontMostSq = BitToSq(FrontMostSq) Else FrontMostSq = 0
End Function


Public Function BackMostSq(Us As enumColor, Op1 As TBit64) As Long
 ' returns first square board position relative for color
 If Us = WCOL Then BackMostSq = Lsb64(Op1) Else BackMostSq = Rsb64(Op1)
 If BackMostSq >= 0 Then BackMostSq = BitToSq(BackMostSq) Else BackMostSq = 0
End Function

Public Sub Or64To(Op1 As TBit64, Op2 As TBit64, Result As TBit64)
  Result.i0 = Op1.i0 Or Op2.i0: Result.i1 = Op1.i1 Or Op2.i1
End Sub
```

```


'---------------------------------------------------------
Public Sub SetMove(m1 As TMOVE, m2 As TMOVE)
 With m1
  .Captured = m2.Captured
  .CapturedNumber = m2.CapturedNumber
  .Castle = m2.Castle
  .EnPassant = m2.EnPassant
  .From = m2.From
  .IsChecking = m2.IsChecking
  .IsLegal = m2.IsLegal
  .OrderValue = m2.OrderValue
  .Piece = m2.Piece
  .Promoted = m2.Promoted
  .SeeValue = m2.SeeValue
  .Target = m2.Target
 End With
End Sub

Public Sub SwapMove(m1 As TMOVE, m2 As TMOVE)
 Dim t As TMOVE
 With t
  .Captured = m2.Captured: m2.Captured = m1.Captured: m1.Captured = .Captured
  .CapturedNumber = m2.CapturedNumber: m2.CapturedNumber = m1.CapturedNumber: m1.CapturedNumber = .CapturedNumber
  .Castle = m2.Castle: m2.Castle = m1.Castle: m1.Castle = .Castle
  .EnPassant = m2.EnPassant: m2.EnPassant = m1.EnPassant: m1.EnPassant = .EnPassant
  .From = m2.From: m2.From = m1.From: m1.From = .From
  .IsChecking = m2.IsChecking: m2.IsChecking = m1.IsChecking: m1.IsChecking = .IsChecking
  .IsLegal = m2.IsLegal: m2.IsLegal = m1.IsLegal: m1.IsLegal = .IsLegal
  .OrderValue = m2.OrderValue: m2.OrderValue = m1.OrderValue: m1.OrderValue = .OrderValue
  .Piece = m2.Piece: m2.Piece = m1.Piece: m1.Piece = .Piece
  .Promoted = m2.Promoted: m2.Promoted = m1.Promoted: m1.Promoted = .Promoted
  .SeeValue = m2.SeeValue: m2.SeeValue = m1.SeeValue: m1.SeeValue = .SeeValue
  .Target = m2.Target: m2.Target = m1.Target: m1.Target = .Target
 End With

End Sub

Public Sub ClearMove(m1 As TMOVE)
  With m1
    .From = 0: .Target = 0: .Piece = NO_PIECE: .Castle = NO_CASTLE: .Promoted = 0: .Captured = NO_PIECE: .CapturedNumber = 0
    .EnPassant = 0: .IsChecking = False: .IsLegal = False: .OrderValue = 0: .SeeValue = UNKNOWN_SCORE
  End With
End Sub


Public Function Test64()
 Dim b As TBit64, bb As TBit64, i As Long, t As TBit64, x As Long
 InitEngine

 Init32BitBoards

'----
Dim StartTime As Single, EndTime As Single, y As Long, z As Long, sq As Long, j As Long
Dim m1 As TMOVE, m2 As TMOVE, m3 As TMOVE


StartTime = Timer

b = EmptyBB: x = Len(m1)
t.i0 = 123
```

```
m1.From = 2: m2.From = 12: m3.Target = 34
For i = 1 To 50000000

    SetMove m1, EmptyMove '2x schneller
    SetMove m2, m3
    SetMove m3, m1

 ' m1 = EmptyMove
 ' m2 = m3
 ' m3 = m1

    '1. ---
   'x = 23 + i Mod 2
    'SetBit64 b, x
   'If x < 32& Then If x = 31 Then b.i0 = b.i0 Or BitL_31 Else b.i0 = b.i0 Or Bit32Pos(x) Else If x = 63 Then b.i1 = b.i1
   Or BitL_31 Else b.i1 = b.i1 Or Bit32Pos(x - 32)

   'x = x + 20
   'SetBit64 b, x
   'If x < 32& Then If x = 31 Then b.i0 = b.i0 Or BitL_31 Else b.i0 = b.i0 Or Bit32Pos(x) Else If x = 63 Then b.i1 = b.i1
   Or BitL_31 Else b.i1 = b.i1 Or Bit32Pos(x - 32)



   'For j = 1 To 7
   '  bb.i0 = AttackedByBB(1, j).i0: bb.i1 = AttackedByBB(1, j).i1
   '  b.i0 = bb.i0: b.i1 = bb.i1
      'b.i0 = bb.i0 Or t.i0: b.i1 = bb.i1 Or t.i1


      'bb = AttackedByBB(1, j)
      'Set64 b, bb
      'b = bb
      'OR64 b, bb, t
   'Next
   '------------------------------------------------
    'z = 31 + i Mod 2
   '2.---
   'AttackedByBB(1, PT_PAWN).i0 = AttackedByBB(1, PT_PAWN).i0 Or SquareBB(z).i0: AttackedByBB(1,
   PT_PAWN).i1 = AttackedByBB(1, PT_PAWN).i1 Or SquareBB(z).i1
   'SetOR64 AttackedByBB(1, PT_PAWN), SquareBB(z)

   '3.---
   'bb.i0 = b.i0 Or SquareBB(z).i0: bb.i1 = b.i1 Or SquareBB(z).i1
   ' bb = Or64(b, SquareBB(z)) ' 24,6
   'Or64To b, SquareBB(z), bb ' 3,7
   'Or64ToP b, SquareBB(z)    ' 2,6


Next

EndTime = Timer
Debug.Print Format$(EndTime - StartTime, "0.000")
Debug.Print y
MsgBox Format$(EndTime - StartTime, "0.000") & "           " & bb.i0 & bb.i1 & x &
m1.Target & m2.Target & b.i1 & t.i1 & b.i0 & AttackedByBB(1, PT_PAWN).i0

End Function
Attribute VB_Name = "basBoard"
'==================================================
'= basBoard:
'= Board structure and move generation
'==================================================
Option Explicit
' Index in array Board(119):   A1=21, A8=28, H1=91, H8=98
' frame needed for move generation (max knight move distance = 2+1 squares)
'  110  -- -- -- -- -- -- -- -- -- --  119
'  100  -- -- -- -- -- -- -- -- -- --  109
'  90  -- A8  B8  C8  D8  E8  F8  G8  H8  --   99
'  80  -- A7  B7  C7  D7  E7  F7  G7  H7  --   89
'  70  -- A6  B6  C6  D6  E6  F6  G6  H6  --   79
'  60  -- A5  B5  C5  D5  E5  F5  G5  H5  --   69
'  50  -- A4  B4  C4  D4  E4  F4  G4  H4  --   59
'  40  -- A3  B3  C3  D3  E3  F3  G3  H3  --   49
'  30  -- A2  B2  C2  D2  E2  F2  G2  H2  --   39
'  20  -- A1  B1  C1  D1  E1  F1  G1  H1  --   29
'  10  -- -- -- -- -- -- -- -- -- --   19
'   0  -- -- -- -- -- -- -- -- -- --    9
'
Public Board(MAX_BOARD)                            As Long ' Game board for all moves
Public NumPieces                                   As Long  '--- Current number of pieces at ply 0 in Pieces list
Public Pieces(32)                                  As Long  '--- List of pieces: pointer to board position (Captured pieces ares set to zero during search)
Public Squares(MAX_BOARD)                          As Long  '--- Squares on board: pointer to pieces list (Captured pieces ares set to zero during search)
Public ColorSq(MAX_BOARD)                          As Long  '--- Squares color: COL_WHITE or COL_BLACK
Public PieceCnt(16)                                As Long  ' number of pieces per piece type and color
Public SameXRay(MAX_BOARD, MAX_BOARD)              As Boolean ' are two squares on same rank or file or diagonal?
'Public SameRookRay(MAX_BOARD, MAX_BOARD)        As Boolean ' are two squares on same rank or file or diagonal?
'Public SameBishopRay(MAX_BOARD, MAX_BOARD)       As Boolean ' are two squares on same rank or file or diagonal?
Public DirOffset(MAX_BOARD, MAX_BOARD)             As Integer ' direction offset from sq1 to sq 2
Public bWhiteToMove                                As Boolean  '--- side to move , false if black to move, often used
Public bCompIsWhite                                As Boolean
Public CastleFlag                                  As enumCastleFlag
Public WhiteCastled                                As enumCastleFlag
Public BlackCastled                                As enumCastleFlag
Public WPromotions(5)                              As Long  '--- list of promotion pieces
Public BPromotions(5)                              As Long
Public WKingLoc                                    As Long  ' white king location
Public BKingLoc                                    As Long  ' black king location
Public PieceType(16)                               As Long  ' sample: maps black pawn and white pawn pieces to PT_PAWN
Public PieceColor(16)                              As Long  ' white / Black
Public Ply                                         As Long  ' current ply
Public Fifty                                       As Long  ' counter for fifty move draw rule : 100 half moves
Public arFiftyMove(499)                            As Long  ' fifty counter for ply
Public Rank(MAX_BOARD)                             As Long  ' Rank from white view
Public RankB(MAX_BOARD)                            As Long  ' Rank from black view  1 - 8
Public RelativeSq(COL_WHITE, MAX_BOARD)            As Long  ' sq from black view  1 - 8
Public File(MAX_BOARD)                             As Long  ' file on board 1 - 8
Public SqBetween(MAX_BOARD, MAX_BOARD, MAX_BOARD) As Boolean ' (sq,sq1,sq2) is sq between sq1 and sq2?
'--- For faster move generation
Public WhitePiecesStart                            As Long ' used for access to PieceList
Public WhitePiecesEnd                              As Long ' used for access to PieceList
Public BlackPiecesStart                            As Long ' used for access to PieceList
Public BlackPiecesEnd                              As Long ' used for access to PieceList
Public WNonPawnPieces                              As Long ' counts pieces
Public BNonPawnPieces                              As Long ' counts pieces
'--- SEE data ( static exchange evaluation )
Dim PieceList(0 To 32)                             As Long, Cnt As Long
Dim SwapList(0 To 32)                              As Long, slIndex As Long
Dim Blocker(1 To 32)                               As Long, Block As Long
'-------------------------------
Public StartupBoard(MAX_BOARD)                     As Long ' Start Position used for copy to current board
Public Moved(MAX_BOARD)                            As Long ' Track for moved pieces (castle checks + eval)
Public KingCheckW(MAX_BOARD)                             As Integer ' for fast checking moves
detection, integer for faster ERASE
Public KingCheckB(MAX_BOARD)                             As Integer ' for fast checking moves detection
' Offsets of directions - for move generation
Public DirectionOffset(7)                                   As Long
Public KnightOffsets(7)                                  As Long
Public BishopOffsets(3)                                  As Long
Public RookOffsets(3)                                    As Long
Public OppositeDir(-11 To 11)                            As Long
Public EpPosArr(0 To MAX_DEPTH)                          As Long
Public MaxDistance(0 To SQ_H8, 0 To SQ_H8)               As Long ' max distance between two fields
Private bGenCapturesOnly                                 As Boolean ' generate QSearch -catures only
Private bGenQsChecks                                     As Boolean ' generate QSearch checks
'---------------------------------

'-------------------------------------------------------------------
' GenerateMoves()
' ===============
' Generates all Pseudo-legal move for a position. Check for legal moves later with CheckLegal
' if bCapturesOnly then only captures and promotions are generated.
'  if MovePickerDat(Ply).GenerateQSChecksCnt then checks are generated too. For QSearch first ply only.
'-------------------------------------------------------------------
Public Function GenerateMoves(ByVal Ply As Long, _
                              ByVal bCapturesOnly As Boolean, _
                              NumMoves As Long) As Long
  Dim From As Long, i As Long
  '--- Init special board with king checking positions for fast detection of checking moves
  If bWhiteToMove Then FillKingCheckB Else FillKingCheckW

  bGenCapturesOnly = bCapturesOnly: NumMoves = 0
  bGenQsChecks = (MovePickerDat(Ply).GenerateQSChecksCnt > 0)
  If bWhiteToMove Then

    For i = WhitePiecesStart To WhitePiecesEnd
      From = Pieces(i)
      Debug.Assert (From >= SQ_A1 And From <= SQ_H8) Or From = 0 ' from=0 if piece was
      captured during search

      Select Case Board(From)
        Case NO_PIECE, FRAME
        Case WPAWN
          ' note: FRAME has Bit 1 not set - like BCOL: PieceColor() cannot be used here, returns NO_COL for
          EP piece
          If ((Board(From + 11) And 1) = BCOL) Then If Board(From + 11) <> FRAME Then
          TryMoveWPawn Ply, NumMoves, From, From + 11 ' capture right side
          If ((Board(From + 9) And 1) = BCOL) Then If Board(From + 9) <> FRAME Then
          TryMoveWPawn Ply, NumMoves, From, From + 9 ' capture left side
          If Board(From + 10) = NO_PIECE Then ' one row up
            If Rank(From) = 2 Then If Board(From + 20) = NO_PIECE Then TryMoveWPawn
            Ply, NumMoves, From, From + 20 ' two rows up
            TryMoveWPawn Ply, NumMoves, From, From + 10 ' one row up
          End If
        Case WKNIGHT
          TryMoveListKnight Ply, NumMoves, From
        Case WBISHOP
          TryMoveSliderList Ply, NumMoves, From, PT_BISHOP
        Case WROOK
          TryMoveSliderList Ply, NumMoves, From, PT_ROOK
        Case WQUEEN
          TryMoveSliderList Ply, NumMoves, From, PT_QUEEN
        Case WKING
          TryMoveListKing Ply, NumMoves, From
          ' Check castling
          If From = WKING_START Then
            If Moved(WKING_START) = 0 Then
              'o-o
              If Moved(SQ_H1) = 0 And Board(SQ_H1) = WROOK Then
                If Board(SQ_F1) = NO_PIECE And Board(SQ_G1) = NO_PIECE Then
            CastleFlag = WHITEOO
            TryCastleMove Ply, NumMoves, From, From + 2
            CastleFlag = NO_CASTLE
           End If
          End If
          'o-o-o
          If Moved(SQ_A1) = 0 And Board(SQ_A1) = WROOK Then
           If Board(SQ_D1) = NO_PIECE And Board(SQ_C1) = NO_PIECE And Board(SQ_B1) = NO_PIECE Then
            CastleFlag = WHITEOOO
            TryCastleMove Ply, NumMoves, From, From - 2
            CastleFlag = NO_CASTLE
           End If
          End If
         End If
        End If
      End Select

     Next

    Else

     For i = BlackPiecesStart To BlackPiecesEnd
      From = Pieces(i)
      Debug.Assert (From >= SQ_A1 And From <= SQ_H8) Or From = 0

      Select Case Board(From)
       Case NO_PIECE, FRAME
       Case BPAWN
        ' note: NO_PIECE has Bit 1 set like WCOL
        If ((Board(From - 11) And 1) = WCOL) And Board(From - 11) <> NO_PIECE Then TryMoveBPawn Ply, NumMoves, From, From - 11
        If ((Board(From - 9) And 1) = WCOL) And Board(From - 9) <> NO_PIECE Then TryMoveBPawn Ply, NumMoves, From, From - 9
        If Board(From - 10) = NO_PIECE Then
         If Rank(From) = 7 Then If Board(From - 20) = NO_PIECE Then TryMoveBPawn Ply, NumMoves, From, From - 20
         TryMoveBPawn Ply, NumMoves, From, From - 10
        End If
       Case BKNIGHT
        TryMoveListKnight Ply, NumMoves, From
       Case BBISHOP
        TryMoveSliderList Ply, NumMoves, From, PT_BISHOP
       Case BROOK
        TryMoveSliderList Ply, NumMoves, From, PT_ROOK
       Case BQUEEN
        TryMoveSliderList Ply, NumMoves, From, PT_QUEEN
       Case BKING
        TryMoveListKing Ply, NumMoves, From
        ' Check castling
        If From = BKING_START Then
         If Moved(BKING_START) = 0 Then
          'o-o
          If Moved(SQ_H8) = 0 And Board(SQ_H8) = BROOK Then
           If Board(SQ_F8) = NO_PIECE And Board(SQ_G8) = NO_PIECE Then
            CastleFlag = BLACKOO
            TryCastleMove Ply, NumMoves, From, From + 2
            CastleFlag = NO_CASTLE
           End If
          End If
          'o-o-o
          If Moved(SQ_A8) = 0 And Board(SQ_A8) = BROOK Then
           If Board(SQ_D8) = NO_PIECE And Board(SQ_C8) = NO_PIECE And Board(SQ_B8) = NO_PIECE Then
            CastleFlag = BLACKOOO
            TryCastleMove Ply, NumMoves, From, From - 2
            CastleFlag = NO_CASTLE
           End If
```

```
              End If
            End If
          End If
      End Select

    Next

  End If
  GenerateMoves = NumMoves ' return move count
End Function

Private Function TryMoveWPawn(ByVal Ply As Long, _
                              NumMoves As Long, _
                              ByVal From As Long, _
                              ByVal Target As Long) As Boolean
  If Board(Target) = FRAME Then Exit Function
  Dim PieceFrom As Long, PieceTarget As Long, bDoCheckMove As Boolean
  PieceFrom = Board(From): PieceTarget = Board(Target)
  Debug.Assert PieceTarget <> FRAME

  If Rank(From) = 7 Then
      ' White Promotion
      Dim PromotePiece As Long
      For PromotePiece = 1 To 4 ' for each promotion piece type
        With Moves(Ply, NumMoves)
         .From = From: .Target = Target: .Captured = PieceTarget: .EnPassant = 0: .
         Castle = NO_CASTLE: .Promoted = WPromotions(PromotePiece): .Piece = .Promoted
         : .IsChecking = False: .IsLegal = False: .SeeValue = VALUE_NONE: .OrderValue
         = 0
        End With
        NumMoves = NumMoves + 1
      Next
  Else
    With Moves(Ply, NumMoves)
      Select Case PieceTarget
      Case BEP_PIECE
        .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
        IsChecking = False: .Castle = NO_CASTLE: .Captured = PieceTarget: .
        CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        .EnPassant = ENPASSANT_CAPTURE: NumMoves = NumMoves + 1
      Case NO_PIECE, WEP_PIECE ' WEP_PIECE should not appear
        '--- Normal move, not a capture, promotion ---
        bDoCheckMove = False
        '--- in QSearch: Generate checking moves only for first QSearch ply
        If bGenCapturesOnly And bGenQsChecks Then If IsCheckingMove(PieceFrom, From,
        Target, 0, 0) Then bDoCheckMove = True
        If Not bGenCapturesOnly Or bDoCheckMove Then
          '---Normal move, not generated in QSearch (exception: when in check)
          .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
          EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber
           = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
          If Target - From = 20 Then .EnPassant = ENPASSANT_WMOVE
          .IsChecking = bDoCheckMove: NumMoves = NumMoves + 1
        End If
      Case FRAME
      Case Else
        ' Normal capture.
        .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
        IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured =
        PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .
        OrderValue = 0
        NumMoves = NumMoves + 1
      End Select
    End With
  End If

End Function
```

```
Private Function TryMoveBPawn(ByVal Ply As Long, _
                              NumMoves As Long, _
                              ByVal From As Long, _
                              ByVal Target As Long) As Boolean
  If Board(Target) = FRAME Then Exit Function
  Dim PieceFrom As Long, PieceTarget As Long
  PieceFrom = Board(From): PieceTarget = Board(Target)
  Debug.Assert PieceTarget <> FRAME

  If Rank(From) = 2 Then
      ' Black Promotion
      Dim PromotePiece As Long
      For PromotePiece = 1 To 4
        With Moves(Ply, NumMoves)
         .From = From: .Target = Target: .Captured = PieceTarget: .EnPassant = 0: .
         Castle = NO_CASTLE: .Promoted = BPromotions(PromotePiece): .Piece = .Promoted
         : .IsChecking = False: .IsLegal = False: .SeeValue = VALUE_NONE: .OrderValue
         = 0
        End With
        NumMoves = NumMoves + 1
      Next
  Else
    With Moves(Ply, NumMoves)
      Select Case PieceTarget
      Case WEP_PIECE
        .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
        IsChecking = False: .Castle = NO_CASTLE: .Captured = PieceTarget: .
        CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        .EnPassant = ENPASSANT_CAPTURE: NumMoves = NumMoves + 1
      Case NO_PIECE, BEP_PIECE ' BEP_PIECE should not appear
        '--- Normal move, not a capture, promotion ---
        Dim bDoCheckMove As Boolean
        bDoCheckMove = False
        '--- in QSearch: Generate checking moves only for first QSearch ply
        If bGenCapturesOnly And bGenQsChecks Then If IsCheckingMove(PieceFrom, From,
        Target, 0, 0) Then bDoCheckMove = True
        If Not bGenCapturesOnly Or bDoCheckMove Then
          '---Normal move, not generated in QSearch (exception: when in check)
         .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False:  .
         EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber
         = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
          If Target - From = -20 Then .EnPassant = ENPASSANT_BMOVE
          .IsChecking = bDoCheckMove: NumMoves = NumMoves + 1
        End If
      Case FRAME
      Case Else
        ' Normal capture.
        .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
        IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured =
        PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .
        OrderValue = 0
        NumMoves = NumMoves + 1
      End Select
    End With
  End If
End Function

Private Function TryMoveListKnight(ByVal Ply As Long, _
                                   NumMoves As Long, _
                                   ByVal From As Long) As Boolean
  '--- Knights only moves
  Dim Target As Long, ActDir As Long, PieceFrom As Long, PieceTarget As Long,
  bDoCheckMove As Boolean, PieceCol As Long
  PieceFrom = Board(From): PieceCol = (PieceFrom And 1)

  For ActDir = 0 To 7
    Target = From + KnightOffsets(ActDir): PieceTarget = Board(Target)
    Select Case PieceTarget
```

```vb
    Case NO_PIECE, WEP_PIECE, BEP_PIECE
      '--- Normal move, not a capture, castle, promotion ---
      '--- in QSearch: Generate checking moves only for first QSearch ply
      If bGenCapturesOnly And bGenQsChecks Then bDoCheckMove = IsCheckingMove(PieceFrom, From, Target, 0, 0)
      If Not bGenCapturesOnly Or bDoCheckMove Then
        '---Normal move, not generated in QSearch (exception: when in check)
        With Moves(Ply, NumMoves)
          .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .IsChecking = bDoCheckMove: .EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        End With
        NumMoves = NumMoves + 1
      End If
    Case FRAME ' go on with next direction
    Case Else
      ' Captures
      If PieceCol <> (PieceTarget And 1) Then ' Capture of own piece not allowed
        With Moves(Ply, NumMoves)
          .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        End With
        NumMoves = NumMoves + 1
      End If
    End Select
  Next ActDir

End Function

Private Function TryMoveListKing(ByVal Ply As Long, _
                                 NumMoves As Long, _
                                 ByVal From As Long) As Boolean
  '--- Kings only
  Dim Target As Long, ActDir As Long, PieceFrom As Long, PieceTarget As Long, PieceCol As Long

  PieceFrom = Board(From): PieceCol = (PieceFrom And 1)

  For ActDir = 0 To 7
    Target = From + DirectionOffset(ActDir): PieceTarget = Board(Target)
    Select Case PieceTarget
    Case NO_PIECE, WEP_PIECE, BEP_PIECE
      '--- Normal move, not a capture, castle, not generated in QSearch (exception: when in check)---
      If Not bGenCapturesOnly Then
        '---Normal move, not generated in QSearch (exception: when in check)
        With Moves(Ply, NumMoves)
          .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        End With
        NumMoves = NumMoves + 1
      End If
    Case FRAME ' go on with next direction
    Case Else
      ' Captures
      If PieceCol <> (PieceTarget And 1) Then ' Capture of own piece not allowed
        With Moves(Ply, NumMoves)
          .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured = PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
        End With
        NumMoves = NumMoves + 1
      End If
    End Select
  Next ActDir

End Function

Private Function TryCastleMove(ByVal Ply As Long, _
                               NumMoves As Long, _
                               ByVal From As Long, _
                               ByVal Target As Long) As Boolean
  If Board(Target) = FRAME Then Exit Function
  Dim CurrentMove As TMOVE, PieceFrom As Long, PieceTarget As Long
  PieceFrom = Board(From): PieceTarget = Board(Target): TryCastleMove = False
  If CastleFlag <> NO_CASTLE Then
    If Not bGenCapturesOnly Then
      CurrentMove.From = From
      CurrentMove.Target = Target
      CurrentMove.Piece = PieceFrom
      CurrentMove.Captured = PieceTarget
      CurrentMove.EnPassant = 0
      CurrentMove.Castle = CastleFlag
      CurrentMove.Promoted = 0: CurrentMove.IsChecking = False
      CurrentMove.SeeValue = VALUE_NONE
      CastleFlag = NO_CASTLE
      SetMove Moves(Ply, NumMoves), CurrentMove
      NumMoves = NumMoves + 1
      TryCastleMove = True
    End If
  End If
End Function

Private Sub TryMoveSliderList(ByVal Ply As Long, _
                              NumMoves As Long, _
                              ByVal From As Long, _
                              ByVal PieceType As Long)
  Dim Target As Long, ActDir As Long, Offset As Long
  Dim PieceFrom As Long, PieceTarget As Long, bDoCheckMove As Boolean, DirStart As
  Long, DirEnd As Long, PieceCol As Long

  PieceFrom = Board(From): PieceCol = (PieceFrom And 1)

  Select Case PieceType ' get move directions
    Case PT_ROOK: DirStart = 0: DirEnd = 3 ' Rook
    Case PT_BISHOP: DirStart = 4: DirEnd = 7 ' Bishop
    Case Else: DirStart = 0: DirEnd = 7 ' Queen
  End Select

  For ActDir = DirStart To DirEnd ' for all possible directions
      Offset = DirectionOffset(ActDir): Target = From + Offset
      Do While Board(Target) <> FRAME '--- Slide loop
         PieceTarget = Board(Target)
         If PieceTarget < NO_PIECE Then ' Captures , not EnPassant
           If PieceCol <> (PieceTarget And 1) Then ' Capture of own piece not allowed, color in last
           bit of piece (even/uneven)
             ' Capture: add move to list
             With Moves(Ply, NumMoves)
               .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .
               IsChecking = False: .EnPassant = 0: .Castle = NO_CASTLE: .Captured =
               PieceTarget: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE:
                .OrderValue = 0
             End With
             NumMoves = NumMoves + 1
           End If
           Exit Do '<<< end for this direction
         End If
         '--- Normal move, not a capture, castle, promotion ---
         '--- in QSearch: Generate checking moves only for first QSearch ply
         If bGenCapturesOnly And bGenQsChecks Then bDoCheckMove = IsCheckingMove(
         PieceFrom, From, Target, 0, 0) Else bDoCheckMove = False
```

```
        If Not bGenCapturesOnly Or bDoCheckMove Then '---Normal move, not generated in QSearch (exception: when in check)
          With Moves(Ply, NumMoves) ' add move to list
            .From = From: .Target = Target: .Piece = PieceFrom: .IsLegal = False: .IsChecking = bDoCheckMove: .EnPassant = 0: .Castle = NO_CASTLE: .Captured = NO_PIECE: .CapturedNumber = 0: .Promoted = 0: .SeeValue = VALUE_NONE: .OrderValue = 0
          End With
          NumMoves = NumMoves + 1
        End If
        Target = Target + Offset
      Loop
    Next ActDir
End Sub

Public Function CheckLegalNotInCheck(mMove As TMOVE) As Boolean
  ' fast check for legal move: not for castling and when in check
  Dim Offset As Long, Target As Long, Piece As Long
  CheckLegalNotInCheck = False
  If mMove.From < SQ_A1 Then Exit Function

  If bWhiteToMove Then
    If mMove.Piece = BKING Then
      If IsAttackedByW(mMove.Target) Then Exit Function
    Else
      If Not SameXRay(mMove.From, BKingLoc) Then CheckLegalNotInCheck = True: Exit Function
      If SqBetween(mMove.From, BKingLoc, mMove.Target) Then CheckLegalNotInCheck = True: Exit Function

      Offset = DirOffset(BKingLoc, mMove.From): Target = mMove.From + Offset: Piece = Board(Target)
      Do While Piece <> FRAME
        If Piece < NO_PIECE Then
          Select Case Abs(Offset)
          Case 1, 10:
            If Piece = WROOK Or Piece = WQUEEN Then
              Exit Do ' still to check other direction
            Else
              CheckLegalNotInCheck = True: Exit Function
            End If
          Case 9, 11:
            If Piece = WBISHOP Or Piece = WQUEEN Then
              Exit Do ' still to check other direction
            Else
              CheckLegalNotInCheck = True: Exit Function
            End If
          Case Else
            CheckLegalNotInCheck = True: Exit Function
          End Select
        End If
        Target = Target + Offset: Piece = Board(Target)
      Loop

      If Piece <> FRAME Then
        '--- possible pinner found. check if there are other piece in direction to king
        Offset = -Offset: Target = mMove.From + Offset: Piece = Board(Target)
        Do While Piece <> FRAME
          If Piece < NO_PIECE Then
            If Piece = BKING Then
              CheckLegalNotInCheck = False: Exit Function
            Else
              CheckLegalNotInCheck = True: Exit Function
            End If
          End If
          Target = Target + Offset: Piece = Board(Target)
        Loop
      End If
```

```
        CheckLegalNotInCheck = True: Exit Function

      End If
    Else
      If mMove.Piece = WKING Then
        If IsAttackedByB(mMove.Target) Then Exit Function
      Else
        If Not SameXRay(mMove.From, WKingLoc) Then CheckLegalNotInCheck = True: Exit
        Function
        If SqBetween(mMove.From, WKingLoc, mMove.Target) Then CheckLegalNotInCheck =
        True: Exit Function

        Offset = DirOffset(WKingLoc, mMove.From): Target = mMove.From + Offset: Piece =
        Board(Target)
        Do While Piece <> FRAME
          If Piece < NO_PIECE Then
            Select Case Abs(Offset)
            Case 1, 10:
              If Piece = BROOK Or Piece = BQUEEN Then
                Exit Do ' still to check other direction
              Else
                CheckLegalNotInCheck = True: Exit Function
              End If
            Case 9, 11:
              If Piece = BBISHOP Or Piece = BQUEEN Then
                Exit Do ' still to check other direction
              Else
                CheckLegalNotInCheck = True: Exit Function
              End If
            Case Else
              CheckLegalNotInCheck = True: Exit Function
            End Select
          End If
          Target = Target + Offset: Piece = Board(Target)
        Loop

        If Piece <> FRAME Then
          '--- possible pinner found. check if there are other piece in direction to king
          Offset = -Offset: Target = mMove.From + Offset: Piece = Board(Target)
          Do While Piece <> FRAME
            If Piece < NO_PIECE Then
              If Piece = WKING Then
                CheckLegalNotInCheck = False: Exit Function
              Else
                CheckLegalNotInCheck = True: Exit Function
              End If
            End If
            Target = Target + Offset: Piece = Board(Target)
          Loop
        End If
        CheckLegalNotInCheck = True: Exit Function
      End If
    End If
    CheckLegalNotInCheck = CheckLegal(mMove)
  End Function

  '---------------------------------------------------------------------------
  '- CheckLegal() - Legal move?
  '---------------------------------------------------------------------------
  Public Function CheckLegal(mMove As TMOVE) As Boolean
    CheckLegal = False
    If mMove.From < SQ_A1 Then Exit Function
    If mMove.Castle = NO_CASTLE Then
      If bWhiteToMove Then
        If IsAttackedByW(BKingLoc) Then Exit Function ' BKing mate?
      Else
        If IsAttackedByB(WKingLoc) Then Exit Function ' WKing mate?
      End If
    Else

      ' Castling
      Select Case mMove.Castle
        Case WHITEOO:
          If IsAttackedByB(WKING_START) Then Exit Function
          If IsAttackedByB(WKING_START + 1) Then Exit Function
          If IsAttackedByB(WKING_START + 2) Then Exit Function
        Case WHITEOOO:
          If IsAttackedByB(WKING_START) Then Exit Function
          If IsAttackedByB(WKING_START - 1) Then Exit Function
          If IsAttackedByB(WKING_START - 2) Then Exit Function
        Case BLACKOO:
          If IsAttackedByW(BKING_START) Then Exit Function
          If IsAttackedByW(BKING_START + 1) Then Exit Function
          If IsAttackedByW(BKING_START + 2) Then Exit Function
        Case BLACKOOO:
          If IsAttackedByW(BKING_START) Then Exit Function
          If IsAttackedByW(BKING_START - 1) Then Exit Function
          If IsAttackedByW(BKING_START - 2) Then Exit Function
      End Select

    End If
    CheckLegal = True
  End Function

  '---------------------------------------------------------------------------
  '- CheckEvasionLegal() - Legal move? in check before
  '---------------------------------------------------------------------------
  Public Function CheckEvasionLegal() As Boolean
    If bWhiteToMove Then
      CheckEvasionLegal = Not IsAttackedByW(BKingLoc)  ' Black king mate?
    Else
      CheckEvasionLegal = Not IsAttackedByB(WKingLoc)  ' White king mate?
    End If
  End Function

  '---------------------------------------------------------------------------
  '- IsAttacked() - piece attacked?  Also used for checking legal move
  '---------------------------------------------------------------------------
  'Public Function IsAttacked(ByVal Location As Long, _
  '                    ByVal AttackByColor As enumColor) As Boolean
  '  If AttackByColor = COL_WHITE Then
  '    IsAttacked = IsAttackedByW(Location)
  '  Else
  '    IsAttacked = IsAttackedByB(Location)
  '  End If
  'End Function

  '---------------------------------------------------------------------------
  '- IsAttackedByW() - square attacked by white ?  Also used for checking legal move
  '---------------------------------------------------------------------------
  Public Function IsAttackedByW(ByVal Location As Long) As Boolean
    Dim i          As Long, Target As Long, Offset As Long, Piece As Long
    Dim OppQRCnt As Long, OppQBCnt As Long
    IsAttackedByW = True
    OppQRCnt = PieceCnt(WQUEEN) + PieceCnt(WROOK): OppQBCnt = PieceCnt(WQUEEN) + PieceCnt(WBISHOP)

    ' vertical+horizontal: Queen, Rook, King
    For i = 0 To 3
      Offset = DirectionOffset(i): Target = Location + Offset: Piece = Board(Target)
      If Piece <> FRAME Then
        If Piece = WKING Then Exit Function
        If OppQRCnt > 0 Then

          Do While Piece <> FRAME
            If Piece < NO_PIECE Then If Piece = WROOK Or Piece = WQUEEN Then Exit
            Function Else Exit Do
            Target = Target + Offset: Piece = Board(Target)
          Loop

        End If
      End If
    Next

    ' diagonal: Queen, Bishop, Pawn, King
    For i = 4 To 7
      Offset = DirectionOffset(i): Target = Location + Offset: Piece = Board(Target)
      If Piece <> FRAME Then
        If Piece = WPAWN Then
          If ((i = 5) Or (i = 7)) Then Exit Function
        ElseIf Piece = WKING Then Exit Function
        ElseIf OppQBCnt <> 0 Then

          Do While Piece <> FRAME
            If Piece < NO_PIECE Then If Piece = WBISHOP Or Piece = WQUEEN Then Exit
            Function Else Exit Do
            Target = Target + Offset: Piece = Board(Target)
          Loop

        End If
      End If
    Next

    If PieceCnt(WKNIGHT) > 0 Then
      For i = 0 To 7
        If Board(Location + KnightOffsets(i)) = WKNIGHT Then Exit Function ' Knight
      Next
    End If
    IsAttackedByW = False
  End Function

  '---------------------------------------------------------------------------
  '- IsAttackedByB() - square attacked by black ?  Also used for checking legal move
  '---------------------------------------------------------------------------
  Public Function IsAttackedByB(ByVal Location As Long) As Boolean
    Dim i        As Long, Target As Long, Offset As Long, Piece As Long
    Dim OppQRCnt As Long, OppQBCnt As Long
    IsAttackedByB = True
    OppQRCnt = PieceCnt(BQUEEN) + PieceCnt(BROOK): OppQBCnt = PieceCnt(BQUEEN) +
    PieceCnt(BBISHOP)

    ' vertical+horizontal: Queen, Rook, King
    For i = 0 To 3
      Offset = DirectionOffset(i): Target = Location + Offset: Piece = Board(Target)
      If Piece <> FRAME Then
        If Piece = BKING Then Exit Function
        If OppQRCnt > 0 Then

          Do While Piece <> FRAME
            If Piece < NO_PIECE Then If Piece = BROOK Or Piece = BQUEEN Then Exit
            Function Else Exit Do
            Target = Target + Offset: Piece = Board(Target)
          Loop

        End If
      End If
    Next

    ' diagonal: Queen, Bishop, Pawn, King
    For i = 4 To 7
      Offset = DirectionOffset(i): Target = Location + Offset: Piece = Board(Target)
      If Piece <> FRAME Then
        If Piece = BPAWN Then
          If ((i = 4) Or (i = 6)) Then Exit Function
        ElseIf Piece = BKING Then Exit Function
        ElseIf OppQBCnt <> 0 Then

          Do While Piece <> FRAME
            If Piece < NO_PIECE Then If Piece = BBISHOP Or Piece = BQUEEN Then Exit Function Else Exit Do
            Target = Target + Offset: Piece = Board(Target)
          Loop

        End If
      End If
    Next

    If PieceCnt(BKNIGHT) > 0 Then
      For i = 0 To 7
        If Board(Location + KnightOffsets(i)) = BKNIGHT Then Exit Function ' Knight
      Next
    End If
    IsAttackedByB = False
  End Function

  Public Sub PlayMove(mMove As TMOVE)
    '--- Play move in game
    Dim From      As Long, Target As Long
    Dim EnPassant As Long, Castle As Long, PromoteTo As Long
    Dim i         As Long

    With mMove
      From = .From
      Target = .Target
      EnPassant = .EnPassant
      Castle = .Castle
      PromoteTo = .Promoted
    End With

    ' Init EnPassant fields
    For i = SQ_A3 To SQ_A6
      If (Board(i) = WEP_PIECE) Then Board(i) = NO_PIECE
    Next

    For i = SQ_A6 To SQ_H6
      If (Board(i) = BEP_PIECE) Then Board(i) = NO_PIECE
    Next

    ' 50 move draw rule
    If Board(From) = WPAWN Or Board(From) = BPAWN Or Board(Target) < NO_PIECE Or PromoteTo <> 0 Then
      Fifty = 0
    Else
      Fifty = Fifty + 1
    End If
    PliesFromNull = PliesFromNull + 1
    bWhiteToMove = Not bWhiteToMove

    Select Case Castle
      Case NO_CASTLE
      Case WHITEOO
        Board(Target) = Board(From)
        Board(From) = NO_PIECE
        Board(SQ_H1) = NO_PIECE
        Board(SQ_F1) = WROOK
        Moved(Target) = Moved(Target) + 1
        Moved(From) = Moved(From) + 1
        Moved(SQ_H1) = Moved(SQ_H1) + 1
        Moved(SQ_F1) = Moved(SQ_F1) + 1
        WhiteCastled = WHITEOO
        WKingLoc = Target
        InitPieceSquares
```

```
        Exit Sub
      Case WHITEOOO
        Board(Target) = Board(From)
        Board(From) = NO_PIECE
        Board(SQ_A1) = NO_PIECE
        Board(SQ_D1) = WROOK
        Moved(Target) = Moved(Target) + 1
        Moved(From) = Moved(From) + 1
        Moved(SQ_A1) = Moved(SQ_A1) + 1
        Moved(SQ_D1) = Moved(SQ_D1) + 1
        WhiteCastled = WHITEOOO
        WKingLoc = Target
        InitPieceSquares
        Exit Sub
      Case BLACKOO
        Board(Target) = Board(From)
        Board(From) = NO_PIECE
        Board(SQ_H8) = NO_PIECE
        Board(SQ_F8) = BROOK
        Moved(Target) = Moved(Target) + 1
        Moved(From) = Moved(From) + 1
        Moved(SQ_H8) = Moved(SQ_H8) + 1
        Moved(SQ_F8) = Moved(SQ_F8) + 1
        BlackCastled = BLACKOO
        BKingLoc = Target
        InitPieceSquares
        Exit Sub
      Case BLACKOOO
        Board(Target) = Board(From)
        Board(From) = NO_PIECE
        Board(SQ_A8) = NO_PIECE
        Board(SQ_D8) = BROOK
        Moved(Target) = Moved(Target) + 1
        Moved(From) = Moved(From) + 1
        Moved(SQ_A8) = Moved(SQ_A8) + 1
        Moved(SQ_D8) = Moved(SQ_D8) + 1
        BlackCastled = BLACKOOO
        BKingLoc = Target
        InitPieceSquares
        Exit Sub
    End Select

    ' en passant
    If EnPassant = ENPASSANT_CAPTURE And (Board(From) And 1) <> 0 Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(Target - 10) = NO_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      Moved(Target - 10) = Moved(Target - 10) + 1
      InitPieceSquares
      Exit Sub
    End If
    If EnPassant = ENPASSANT_CAPTURE Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(Target + 10) = NO_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      Moved(Target + 10) = Moved(Target + 10) + 1
      InitPieceSquares
      Exit Sub
    End If
    If Board(From) = BPAWN And Rank(From) = 7 And Target = From - 20 Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(From - 10) = BEP_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      InitPieceSquares
      Exit Sub
    End If
    If Board(From) = BPAWN And Board(Target) = WEP_PIECE Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(Target + 10) = NO_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      Moved(Target + 10) = Moved(Target + 10) + 1
      InitPieceSquares
      Exit Sub
    End If
    If Board(From) = WPAWN And Rank(From) = 2 And Target = From + 20 Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(From + 10) = WEP_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      InitPieceSquares
      Exit Sub
    End If
    If Board(From) = WPAWN And Board(Target) = BEP_PIECE Then
      Board(Target) = Board(From)
      Board(From) = NO_PIECE
      Board(Target - 10) = NO_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      Moved(Target - 10) = Moved(Target - 10) + 1
      InitPieceSquares
      Exit Sub
    End If
    ' Promotion
    If PromoteTo <> 0 Then
      Board(Target) = PromoteTo
      Board(From) = NO_PIECE
      Moved(Target) = Moved(Target) + 1
      Moved(From) = Moved(From) + 1
      InitPieceSquares
      Exit Sub
    End If
    ' Normal move
    If Board(From) = WKING Then
      WKingLoc = Target
    ElseIf Board(From) = BKING Then
      BKingLoc = Target
    End If
    Board(Target) = Board(From)
    Board(From) = NO_PIECE
    Moved(Target) = Moved(Target) + 1
    Moved(From) = Moved(From) + 1
    InitPieceSquares
  End Sub

  Public Sub MakeMove(mMove As TMOVE)
    '--- Do move on board
    Dim From      As Long, Target As Long
    Dim Captured  As Long, EnPassant As Long
    Dim Promoted  As Long, Castle As Long
    Dim PieceFrom As Long

    With mMove
      From = .From: Target = .Target: Captured = .Captured: EnPassant = .EnPassant:
      Promoted = .Promoted: Castle = .Castle
    End With

    PieceFrom = Board(From)
    Board(From) = NO_PIECE: Moved(From) = Moved(From) + 1
    mMove.CapturedNumber = Squares(Target)
    Pieces(Squares(From)) = Target: Pieces(Squares(Target)) = 0
    Squares(Target) = Squares(From): Squares(From) = 0
    arFiftyMove(Ply) = Fifty: PliesFromNull = PliesFromNull + 1
    If PieceFrom = WPAWN Or PieceFrom = BPAWN Or Board(Target) < NO_PIECE Or Promoted <>
     0 Then Fifty = 0 Else Fifty = Fifty + 1

    ' En Passant
    EpPosArr(Ply + 1) = 0
    If EnPassant <> 0 Then
      If EnPassant = ENPASSANT_WMOVE Then
        Board(From + 10) = WEP_PIECE
        EpPosArr(Ply + 1) = From + 10
      ElseIf EnPassant = ENPASSANT_BMOVE Then
        Board(From - 10) = BEP_PIECE
        EpPosArr(Ply + 1) = From - 10
      End If
      If EnPassant = ENPASSANT_CAPTURE Then '--- EP capture move
        If PieceFrom = WPAWN Then
          Board(Target) = PieceFrom
          Board(Target - 10) = NO_PIECE: PieceCntMinus BPAWN
          mMove.CapturedNumber = Squares(Target - 10)
          Pieces(Squares(Target - 10)) = 0: Squares(Target - 10) = 0
        ElseIf PieceFrom = BPAWN Then
          Board(Target) = PieceFrom
          Board(Target + 10) = NO_PIECE: PieceCntMinus WPAWN
          mMove.CapturedNumber = Squares(Target + 10)
          Pieces(Squares(Target + 10)) = 0: Squares(Target + 10) = 0
        End If
        GoTo lblExit
      End If
    End If
    'Castle: additional rook move here, King later as normal move
    If Castle <> NO_CASTLE Then

      Select Case Castle
        Case WHITEOO
          WhiteCastled = WHITEOO
          Board(SQ_H1) = NO_PIECE: Moved(SQ_H1) = Moved(SQ_H1) + 1
          Board(SQ_F1) = WROOK: Moved(SQ_F1) = Moved(SQ_F1) + 1
          Pieces(Squares(SQ_H1)) = SQ_F1: Squares(SQ_F1) = Squares(SQ_H1): Squares(SQ_H1
          ) = 0
          Board(SQ_G1) = WKING: Moved(SQ_G1) = Moved(SQ_G1) + 1: WKingLoc = SQ_G1
          GoTo lblExit
        Case WHITEOOO
          WhiteCastled = WHITEOOO
          Board(SQ_A1) = NO_PIECE: Moved(SQ_A1) = Moved(SQ_A1) + 1
          Board(SQ_D1) = WROOK: Moved(SQ_D1) = Moved(SQ_D1) + 1
          Pieces(Squares(SQ_A1)) = SQ_D1: Squares(SQ_D1) = Squares(SQ_A1): Squares(SQ_A1
          ) = 0
          Board(SQ_C1) = WKING: Moved(SQ_C1) = Moved(SQ_C1) + 1: WKingLoc = SQ_C1
          GoTo lblExit
        Case BLACKOO
          BlackCastled = BLACKOO
          Board(SQ_H8) = NO_PIECE: Moved(SQ_H8) = Moved(SQ_H8) + 1
          Board(SQ_F8) = BROOK: Moved(SQ_F8) = Moved(SQ_F8) + 1
          Pieces(Squares(SQ_H8)) = SQ_F8: Squares(SQ_F8) = Squares(SQ_H8): Squares(SQ_H8
          ) = 0
          Board(SQ_G8) = BKING: Moved(SQ_G8) = Moved(SQ_G8) + 1: BKingLoc = SQ_G8
          GoTo lblExit
        Case BLACKOOO
          BlackCastled = BLACKOOO
          Board(SQ_A8) = NO_PIECE: Moved(SQ_A8) = Moved(SQ_A8) + 1
          Board(SQ_D8) = BROOK: Moved(SQ_D8) = Moved(SQ_D8) + 1
          Pieces(Squares(SQ_A8)) = SQ_D8: Squares(SQ_D8) = Squares(SQ_A8): Squares(SQ_A8
          ) = 0
          Board(SQ_C8) = BKING: Moved(SQ_C8) = Moved(SQ_C8) + 1: BKingLoc = SQ_C8
          GoTo lblExit
        End Select

    End If
    If Promoted <> 0 Then
      PieceCntPlus Promoted
      Board(Target) = Promoted
      PieceCntMinus PieceFrom
      Moved(Target) = Moved(Target) + 1
    Else

      '--- normal move
      Select Case PieceFrom
        Case WKING: WKingLoc = Target
        Case BKING: BKingLoc = Target
      End Select

      Board(Target) = PieceFrom: Moved(Target) = Moved(Target) + 1
    End If
    If Captured > 0 Then If Captured < NO_PIECE Then PieceCntMinus Captured
  lblExit:
    bWhiteToMove = Not bWhiteToMove
  End Sub

  Public Sub UnmakeMove(mMove As TMOVE)
    ' take back this move on board
    Dim From     As Long, Target As Long
    Dim Captured As Long, EnPassant As Long, CapturedNumber As Long
    Dim Promoted As Long, Castle As Long, PieceTarget As Long

    With mMove
      From = .From: Target = .Target: Captured = .Captured
      EnPassant = .EnPassant: Promoted = .Promoted: Castle = .Castle: CapturedNumber = .
      CapturedNumber
    End With

    PieceTarget = Board(Target)
    Squares(From) = Squares(Target): Squares(Target) = CapturedNumber
    Pieces(Squares(Target)) = Target: Pieces(Squares(From)) = From
    Fifty = arFiftyMove(Ply)
    If Castle <> NO_CASTLE Then

      Select Case Castle
        Case WHITEOO
          WhiteCastled = NO_CASTLE
          Board(SQ_F1) = NO_PIECE: Moved(SQ_F1) = Moved(SQ_F1) - 1
          Board(SQ_H1) = WROOK: Moved(SQ_H1) = Moved(SQ_H1) - 1
          Squares(SQ_H1) = Squares(SQ_F1): Squares(SQ_F1) = 0: Pieces(Squares(SQ_H1)) =
          SQ_H1
          Board(SQ_E1) = WKING: Moved(SQ_E1) = Moved(SQ_E1) - 1: WKingLoc = SQ_E1
          Board(SQ_G1) = NO_PIECE: Moved(SQ_G1) = Moved(SQ_G1) - 1
          GoTo lblExit
        Case WHITEOOO
          WhiteCastled = NO_CASTLE
          Board(SQ_D1) = NO_PIECE: Moved(SQ_D1) = Moved(SQ_D1) - 1
          Board(SQ_A1) = WROOK: Moved(SQ_A1) = Moved(SQ_A1) - 1
          Squares(SQ_A1) = Squares(SQ_D1): Squares(SQ_D1) = 0: Pieces(Squares(SQ_A1)) =
          SQ_A1
          Board(SQ_E1) = WKING: Moved(SQ_E1) = Moved(SQ_E1) - 1: WKingLoc = SQ_E1
          Board(SQ_C1) = NO_PIECE: Moved(SQ_C1) = Moved(SQ_C1) - 1
          GoTo lblExit
        Case BLACKOO
          BlackCastled = NO_CASTLE
          Board(SQ_F8) = NO_PIECE: Moved(SQ_F8) = Moved(SQ_F8) - 1
          Board(SQ_H8) = BROOK: Moved(SQ_H8) = Moved(SQ_H8) - 1
          Squares(SQ_H8) = Squares(SQ_F8): Squares(SQ_F8) = 0: Pieces(Squares(SQ_H8)) =
          SQ_H8
          Board(SQ_E8) = BKING: Moved(SQ_E8) = Moved(SQ_E8) - 1: BKingLoc = SQ_E8
          Board(SQ_G8) = NO_PIECE: Moved(SQ_G8) = Moved(SQ_G8) - 1
          GoTo lblExit
        Case BLACKOOO
          BlackCastled = NO_CASTLE
          Board(SQ_D8) = NO_PIECE: Moved(SQ_D8) = Moved(SQ_D8) - 1
          Board(SQ_A8) = BROOK: Moved(SQ_A8) = Moved(SQ_A8) - 1
          Squares(SQ_A8) = Squares(SQ_D8): Squares(SQ_D8) = 0: Pieces(Squares(SQ_A8)) =
          SQ_A8
          Board(SQ_E8) = BKING: Moved(SQ_E8) = Moved(SQ_E8) - 1: BKingLoc = SQ_E8
          Board(SQ_C8) = NO_PIECE: Moved(SQ_C8) = Moved(SQ_C8) - 1
          GoTo lblExit
      End Select

    End If
    If EnPassant <> 0 Then
      If EnPassant = ENPASSANT_WMOVE Then
        Board(From + 10) = NO_PIECE
      ElseIf EnPassant = ENPASSANT_BMOVE Then
        Board(From - 10) = NO_PIECE
      End If
      If EnPassant = ENPASSANT_CAPTURE Then
        If PieceTarget = WPAWN Then
          Board(From) = PieceTarget
          Board(Target) = NO_PIECE
          Board(Target - 10) = BPAWN: PieceCntPlus BPAWN
          Squares(Target - 10) = CapturedNumber
          Pieces(CapturedNumber) = Target - 10
          Squares(Target) = 0
        ElseIf PieceTarget = BPAWN Then
          Board(From) = PieceTarget
          Board(Target) = NO_PIECE
          Board(Target + 10) = WPAWN: PieceCntPlus WPAWN
          Squares(Target + 10) = CapturedNumber
          Pieces(CapturedNumber) = Target + 10
          Squares(Target) = 0
        End If
        Moved(From) = Moved(From) - 1
        GoTo lblExit
      End If
    End If
    If Promoted <> 0 Then
      If (Promoted And 1) = WCOL Then
        Board(From) = WPAWN: PieceCntPlus WPAWN
        PieceCntMinus Board(Target)
        Board(Target) = Captured
        Moved(From) = Moved(From) - 1
        Moved(Target) = Moved(Target) - 1
      Else
        Board(From) = BPAWN: PieceCntPlus BPAWN
        PieceCntMinus Board(Target)
        Board(Target) = Captured
        Moved(From) = Moved(From) - 1
        Moved(Target) = Moved(Target) - 1
      End If
    Else

      '--- normal move
      Select Case PieceTarget
        Case WKING: WKingLoc = From
        Case BKING: BKingLoc = From
      End Select

      Board(From) = PieceTarget: Moved(From) = Moved(From) - 1
      Board(Target) = Captured: Moved(Target) = Moved(Target) - 1
    End If
    If Captured > 0 Then If Captured < NO_PIECE Then PieceCntPlus Captured
  lblExit:
    PliesFromNull = PliesFromNull - 1
```

```vb

  bWhiteToMove = Not bWhiteToMove ' switch side to move

End Sub

'-----------------------------------------------------------------------
' InitPieceSquares: Init tables for pieces and squares
' Squares(board location) points to piece in Pieces() list
' Pieces(piece num) points to board location
'-----------------------------------------------------------------------
Public Sub InitPieceSquares()
  Dim i As Long, PT As Long
  NumPieces = 0
  Pieces(0) = 0
  Erase PieceCnt()
  Erase Squares()
  Erase Pieces()
  WNonPawnPieces = 0: BNonPawnPieces = 0
  '--- White --
  WhitePiecesStart = 1

  For PT = PT_PAWN To PT_KING ' sort by piece type
    For i = SQ_A1 To SQ_H8
      If (Board(i) <> FRAME And Board(i) < NO_PIECE And (Board(i) And 1) = WCOL) And
      PieceType(Board(i)) = PT Then
        NumPieces = NumPieces + 1: Pieces(NumPieces) = i: Squares(i) = NumPieces
        PieceCntPlus Board(i)

        Select Case Board(i)
          Case WKING: WKingLoc = i
        End Select

      End If
    Next
  Next

  WhitePiecesEnd = NumPieces
  '--- Black ---
  BlackPiecesStart = NumPieces + 1

  For PT = PT_PAWN To PT_KING
    For i = SQ_A1 To SQ_H8
      If (Board(i) <> FRAME And Board(i) < NO_PIECE And (Board(i) And 1) = BCOL) And
      PieceType(Board(i)) = PT Then
        NumPieces = NumPieces + 1: Pieces(NumPieces) = i: Squares(i) = NumPieces
        PieceCntPlus Board(i)

        Select Case Board(i)
          Case BKING: BKingLoc = i
        End Select

      End If
    Next
  Next

  BlackPiecesEnd = NumPieces
  ResetMaterial
End Sub

Public Sub PieceCntPlus(ByVal Piece As Long)
  If Piece > FRAME And Piece < NO_PIECE Then
    PieceCnt(Piece) = PieceCnt(Piece) + 1
    If Piece > BPAWN And Piece < WKING Then ' King not counted
      If CBool(Piece And 1) Then WNonPawnPieces = WNonPawnPieces + 1 Else
      BNonPawnPieces = BNonPawnPieces + 1
    End If
  End If
End Sub
```

```

Public Sub PieceCntMinus(ByVal Piece As Long)
  If Piece > FRAME And Piece < NO_PIECE Then
    PieceCnt(Piece) = PieceCnt(Piece) - 1
    If Piece > BPAWN And Piece < WKING Then
      If CBool(Piece And 1) Then WNonPawnPieces = WNonPawnPieces - 1 Else
      BNonPawnPieces = BNonPawnPieces - 1
    End If
  End If
  Debug.Assert PieceCnt(Piece) >= 0
End Sub

'------------------------------------------------------------------------
'InCheck() Color to move in check?
'------------------------------------------------------------------------
Public Function InCheck() As Boolean
  If bWhiteToMove Then
    InCheck = IsAttackedByB(WKingLoc)
  Else
    InCheck = IsAttackedByW(BKingLoc)
  End If
End Function

'Public Function OppInCheck() As Boolean
'  If Not bWhiteToMove Then
'    OppInCheck = IsAttackedByB(WKingLoc)
'  Else
'    OppInCheck = IsAttackedByW(BKingLoc)
'  End If
'End Function

Public Function LocCoord(ByVal Square As Long) As String
  LocCoord = UCase$(Chr$(File(Square) + 96) & Rank(Square))
End Function

'------------------------------------------------------------------------
' Board File character to number  A => 1
'------------------------------------------------------------------------
Public Function FileRev(ByVal sFile As String) As Long
  FileRev = Asc(LCase$(sFile)) - 96
End Function

'------------------------------------------------------------------------
'RankRev() - Board Rank number to square number Rank 2 = 30
'------------------------------------------------------------------------
Public Function RankRev(ByVal sRank As String) As Long
  RankRev = (Val(sRank) + 1) * 10
End Function

Public Function RelativeRank(ByVal Col As enumColor, ByVal sq As Long) As Long
  If Col = COL_WHITE Then
    RelativeRank = Rank(sq)
  Else
    RelativeRank = (9 - Rank(sq))
  End If
End Function

'------------------------------------------------------------------------
'CompToCoord(): Convert internal move to text output
'------------------------------------------------------------------------
Public Function CompToCoord(CompMove As TMOVE) As String
  Dim sCoordMove As String
  If CompMove.From = 0 Then CompToCoord = "": Exit Function
  sCoordMove = Chr$(File(CompMove.From) + 96) & Rank(CompMove.From) & Chr$(File(
  CompMove.Target) + 96) & Rank(CompMove.Target)
  If CompMove.Promoted <> 0 Then

    Select Case CompMove.Promoted
```

```vb
      Case WKNIGHT, BKNIGHT
        sCoordMove = sCoordMove & "n"
      Case WROOK, BROOK
        sCoordMove = sCoordMove & "r"
      Case WBISHOP, BBISHOP
        sCoordMove = sCoordMove & "b"
      Case WQUEEN, BQUEEN
        sCoordMove = sCoordMove & "q"
    End Select

  End If
  CompToCoord = sCoordMove
End Function

Public Function TextToMove(ByVal sMoveText As String) As TMOVE
  ' format "b7b8q"
  TextToMove = EmptyMove
  sMoveText = Trim(Replace(sMoveText, "-", ""))
  TextToMove.From = CoordToLoc(Left$(sMoveText, 2))
  TextToMove.Piece = Board(TextToMove.From)
  TextToMove.Target = CoordToLoc(Mid$(sMoveText, 3, 2))
  TextToMove.Captured = Board(TextToMove.Target)

  Select Case LCase(Mid$(sMoveText, 5, 1))
    Case "q":
      If PieceColor(TextToMove.Piece) = COL_WHITE Then TextToMove.Promoted = WQUEEN Else TextToMove.Promoted = BQUEEN
    Case "r":
      If PieceColor(TextToMove.Piece) = COL_WHITE Then TextToMove.Promoted = WROOK Else TextToMove.Promoted = BROOK
    Case "b":
      If PieceColor(TextToMove.Piece) = COL_WHITE Then TextToMove.Promoted = WBISHOP Else TextToMove.Promoted = BBISHOP
    Case "n":
      If PieceColor(TextToMove.Piece) = COL_WHITE Then TextToMove.Promoted = WKNIGHT Else TextToMove.Promoted = BKNIGHT
    Case Else
      TextToMove.Promoted = 0
  End Select

End Function


Public Sub RemoveEpPiece()
  ' Remove EP from Previous Move
  If EpPosArr(Ply) > 0 Then Board(EpPosArr(Ply)) = NO_PIECE
End Sub

Public Sub ResetEpPiece()
  ' Reset EP from Previous Move
  If EpPosArr(Ply) > 0 Then
    Select Case Rank(EpPosArr(Ply))
      Case 3
        Board(EpPosArr(Ply)) = WEP_PIECE
      Case 6
        Board(EpPosArr(Ply)) = BEP_PIECE
    End Select
  End If
End Sub

Public Sub CleanEpPieces()
  Dim i As Long

  For i = SQ_A1 To SQ_H8
    If Board(i) = WEP_PIECE Or Board(BEP_PIECE) Then Board(i) = NO_PIECE
  Next

End Sub
```

```vb

'Public Function Alpha2Piece(ByVal sPiece As String, ByVal bWhiteToMove As Boolean) As Long
'  Dim a As Long
'
'  Select Case LCase(sPiece)
'    Case "n"
'      a = WKNIGHT
'    Case "b"
'      a = WBISHOP
'    Case "r"
'      a = WROOK
'    Case "q"
'      a = WQUEEN
'  End Select
'
'  If a > 0 And Not bWhiteToMove Then a = a + 1
'  Alpha2Piece = a
'End Function

Public Function Piece2Alpha(ByVal iPiece As Long) As String

  Select Case iPiece
    Case WPAWN
      Piece2Alpha = "P"
    Case BPAWN
      Piece2Alpha = "p"
    Case WKNIGHT
      Piece2Alpha = "N"
    Case BKNIGHT
      Piece2Alpha = "n"
    Case WBISHOP
      Piece2Alpha = "B"
    Case BBISHOP
      Piece2Alpha = "b"
    Case WROOK
      Piece2Alpha = "R"
    Case BROOK
      Piece2Alpha = "r"
    Case WQUEEN
      Piece2Alpha = "Q"
    Case BQUEEN
      Piece2Alpha = "q"
    Case WKING
      Piece2Alpha = "K"
    Case BKING
      Piece2Alpha = "k"
    Case Else
      Piece2Alpha = "."
  End Select

End Function

'---------------------------------------------------------------------------
'PrintPos() - board position in ASCII table
'---------------------------------------------------------------------------
Public Function PrintPos() As String
  Dim a      As Long, b As Long, c As Long
  Dim sBoard As String
  sBoard = vbCrLf
  If True Then ' Not bCompIsWhite Then  'punto di vista del B (engine e' N)
    sBoard = sBoard & " -------------------" & vbCrLf
    For a = 1 To 8
      sBoard = sBoard & (9 - a) & "|  "

      For b = 1 To 8
        c = 100 - (a * 10) + b
        sBoard = sBoard & Piece2Alpha(Board(c)) & " "
      Next
      sBoard = sBoard & "| " & vbCrLf
    Next

  Else

    For a = 1 To 8
      sBoard = sBoard & a & vbTab

      For b = 1 To 8
        c = 10 + (a * 10) - b
        sBoard = sBoard & Piece2Alpha(Board(c)) & " "
      Next

      sBoard = sBoard & vbCrLf
    Next

  End If
 sBoard = sBoard & " ------------------" & vbCrLf
  sBoard = sBoard & " " & vbTab & " A B C D E F G H" & vbCrLf
  PrintPos = sBoard
End Function

Public Function MoveText(CompMove As TMOVE) As String
  ' Returns move string for data type TMove
  ' Sample: ComPMove.from= 22: CompMove.target=24: MsgBox CompMove > "a2a4"
  Dim sCoordMove As String
  If CompMove.From = 0 Then MoveText = "      ": Exit Function
  sCoordMove = Chr$(File(CompMove.From) + 96) & Rank(CompMove.From)
  If CompMove.Captured <> NO_PIECE And CompMove.Captured > 0 Then sCoordMove = sCoordMove & "x"
  sCoordMove = sCoordMove & Chr$(File(CompMove.Target) + 96) & Rank(CompMove.Target)
  If CompMove.IsChecking Then sCoordMove = sCoordMove & "+"
  If CompMove.Promoted <> 0 Then

    Select Case CompMove.Promoted
      Case WKNIGHT, BKNIGHT
        sCoordMove = sCoordMove & "n"
      Case WROOK, BROOK
        sCoordMove = sCoordMove & "r"
      Case WBISHOP, BBISHOP
        sCoordMove = sCoordMove & "b"
      Case WQUEEN, BQUEEN
        sCoordMove = sCoordMove & "q"
    End Select

  End If
  MoveText = sCoordMove
End Function

Public Function GUIMoveText(CompMove As TMOVE) As String
  If UCIMode Or bWbPvInUciFormat Then
    GUIMoveText = UCIMoveText(CompMove)
  Else
    GUIMoveText = MoveText(CompMove)
  End If
End Function

Public Function UCIMoveText(CompMove As TMOVE) As String
  'UCI: no x for captrue or + for check
  ' Returns move string for data type TMove
  ' Sample: ComPMove.from= 22: CompMove.target=24: MsgBox CompMove > "a2a4"
  Dim sCoordMove As String
  If CompMove.From = 0 Then UCIMoveText = "      ": Exit Function
  sCoordMove = Chr$(File(CompMove.From) + 96) & Rank(CompMove.From)
  sCoordMove = sCoordMove & Chr$(File(CompMove.Target) + 96) & Rank(CompMove.Target)
  If CompMove.Promoted <> 0 Then

```

```
      Select Case CompMove.Promoted
        Case WKNIGHT, BKNIGHT
          sCoordMove = sCoordMove & "n"
        Case WROOK, BROOK
          sCoordMove = sCoordMove & "r"
        Case WBISHOP, BBISHOP
          sCoordMove = sCoordMove & "b"
        Case WQUEEN, BQUEEN
          sCoordMove = sCoordMove & "q"
      End Select

    End If
    UCIMoveText = sCoordMove
  End Function

  Public Function PSQT64(pDestW() As TScore, pDestB() As TScore, ParamArray pSrc())
    ' Read piece square table as parameter list into array
    ' SF tables are symmetric so file A-D is flipped to E-F
    Dim i As Long, sq As Long, x As Long, y As Long, x2 As Long, y2 As Long, MG As Long,
     EG As Long
    Erase pDestW(): Erase pDestB()

    ' Source table is for file A-D, rank 1-8 > Flip for E-F
    For i = 0 To 31
      MG = pSrc(i * 2): EG = pSrc(i * 2 + 1)
      ' White
      x = i Mod 4: y = i \ 4: sq = 21 + x + y * 10
      pDestW(sq).MG = MG: pDestW(sq).EG = EG
      '   Debug.Print x, y, sq, pDestW(sq).MG, pDestW(sq).EG
      ' flip to E-F
      x2 = 7 - x: y2 = y: sq = 21 + x2 + y2 * 10
      pDestW(sq).MG = MG: pDestW(sq).EG = EG
      '   Debug.Print x2, y2, sq, pDestW(sq).MG, pDestW(sq).EG
      ' Black
      x2 = x: y2 = 7 - y: sq = 21 + x2 + y2 * 10
      pDestB(sq).MG = MG: pDestB(sq).EG = EG
      '   Debug.Print x2, y2, sq, pDestB(sq).MG, pDestB(sq).EG
      x2 = 7 - x: y2 = 7 - y: sq = 21 + x2 + y2 * 10
      pDestB(sq).MG = MG: pDestB(sq).EG = EG
      '   Debug.Print x2, y2, sq, pDestB(sq).MG, pDestB(sq).EG
    Next

  End Function

  Public Sub InitRankFile()
    Dim i As Long

    For i = 1 To MAX_BOARD
      Rank(i) = (i \ 10) - 1
      RankB(i) = 9 - Rank(i)
      File(i) = i Mod 10
      RelativeSq(COL_WHITE, i) = i
      RelativeSq(COL_BLACK, i) = SQ_A1 - 1 + File(i) + (8 - Rank(i)) * 10
    Next

  End Sub

  '------------------------------------------------------------------------
  ' AttackedCnt() - ROOK+QUEEN , BISHOP+QUEEN  added
  ' AttackedCnt attacks + DEFENDER
  '------------------------------------------------------------------------
  'Public Function AttackedCnt(ByVal Location As Long, ByVal Color As enumColor) As Long
  '  Dim i As Long, Target As Long
  '  AttackedCnt = 0
  '
  ' ' Orthogonal = index 0-3
  '  For i = 0 To 3
  '    Target = Location + DirectionOffset(i)
```

```
'   If Color = COL_BLACK Then
'     If Board(Target) = BKING Then
'       AttackedCnt = AttackedCnt + 1
'     Else
'
'       Do While Board(Target) <> FRAME
'         If Board(Target) = BROOK Or Board(Target) = BQUEEN Then
'           AttackedCnt = AttackedCnt + 1
'         ElseIf Board(Target) = WROOK Or Board(Target) = WQUEEN Then
'           AttackedCnt = AttackedCnt - 1
'         ElseIf Board(Target) < NO_PIECE Then ' other pieces
'           Exit Do
'         End If
'         Target = Target + DirectionOffset(i)
'       Loop
'
'     End If
'   Else
'     If Board(Target) = WKING Then
'       AttackedCnt = AttackedCnt + 1
'     Else
'
'       Do While Board(Target) <> FRAME
'         If Board(Target) = WROOK Or Board(Target) = WQUEEN Then
'           AttackedCnt = AttackedCnt + 1
'         ElseIf Board(Target) = BROOK Or Board(Target) = BQUEEN Then
'           AttackedCnt = AttackedCnt - 1
'         ElseIf Board(Target) < NO_PIECE Then ' other pieces
'           Exit Do
'         End If
'         Target = Target + DirectionOffset(i)
'       Loop
'
'     End If
'   End If
' Next
'
' ' Diagonal = index 4 to 7
' For i = 4 To 7
'   Target = Location + DirectionOffset(i)
'   If Color = COL_BLACK Then
'     If Board(Target) = BKING Then
'       AttackedCnt = AttackedCnt + 1
'     Else
'       If Board(Target) = BPAWN And ((i = 4) Or (i = 6)) Then
'         AttackedCnt = AttackedCnt + 1
'         Target = Location + DirectionOffset(i)
'       End If
'
'       Do While Board(Target) <> FRAME
'         If Board(Target) = BBISHOP Or Board(Target) = BQUEEN Then
'           AttackedCnt = AttackedCnt + 1
'         ElseIf Board(Target) = WBISHOP Or Board(Target) = WQUEEN Then
'           AttackedCnt = AttackedCnt - 1
'         ElseIf Board(Target) < NO_PIECE Then
'           Exit Do
'         End If
'         Target = Target + DirectionOffset(i)
'       Loop
'
'     End If
'   Else
'     If Board(Target) = WKING Then
'       AttackedCnt = AttackedCnt + 1
'     Else
'       If Board(Target) = WPAWN And ((i = 5) Or (i = 7)) Then
'         AttackedCnt = AttackedCnt + 1
'         Target = Location + DirectionOffset(i)
'    End If
'
'    Do While Board(Target) <> FRAME
'     If Board(Target) = WBISHOP Or Board(Target) = WQUEEN Then
'      AttackedCnt = AttackedCnt + 1
'     ElseIf Board(Target) = BBISHOP Or Board(Target) = BQUEEN Then
'      AttackedCnt = AttackedCnt - 1
'     ElseIf Board(Target) < NO_PIECE Then
'      Exit Do
'     End If
'     Target = Target + DirectionOffset(i)
'    Loop
'
'   End If
'  End If
' Next
'
' ' Knight moves
' For i = 0 To 7
'  Target = Location + KnightOffsets(i)
'  If Color = COL_BLACK Then
'   If Board(Target) = BKNIGHT Then AttackedCnt = AttackedCnt + 1
'   If Board(Target) = WKNIGHT Then AttackedCnt = AttackedCnt - 1
'  Else
'   If Board(Target) = WKNIGHT Then AttackedCnt = AttackedCnt + 1
'   If Board(Target) = BKNIGHT Then AttackedCnt = AttackedCnt - 1
'  End If
' Next
'
'End Function

Public Sub InitMaxDistance()
  ' Max distance x or y
  Dim i As Long, j As Long
  Dim d As Long, v As Long

  For i = SQ_A1 To SQ_H8
    For j = SQ_A1 To SQ_H8
      v = Abs(Rank(i) - Rank(j))
      d = Abs(File(i) - File(j))
      If d > v Then v = d
      MaxDistance(i, j) = v
    Next j
  Next i

End Sub

Public Sub InitSqBetween()
  ' InitSqBetween(sq,Sq1,Sq2) : sq between sq1 and sq2
  Dim i As Long, dir1 As Long, Dir2 As Long, sq As Long, sq1 As Long, sq2 As Long

  For sq = SQ_A1 To SQ_H8
    If File(sq) >= 1 And File(sq) <= 8 And Rank(sq) >= 1 And Rank(sq) <= 8 Then

      For i = 0 To 7
        dir1 = DirectionOffset(i)
        Dir2 = OppositeDir(dir1)
        sq1 = sq + dir1

        Do While File(sq1) >= 1 And File(sq1) <= 8 And Rank(sq1) >= 1 And Rank(sq1) <= 8
          sq2 = sq + Dir2

          Do While File(sq2) >= 1 And File(sq2) <= 8 And Rank(sq2) >= 1 And Rank(sq2) <= 8
            SqBetween(sq, sq1, sq2) = True
            sq2 = sq2 + Dir2
          Loop
          sq1 = sq1 + dir1
        Loop

      Next i

    End If
  Next sq

End Sub

'Public Function TotalPieceValue() As Long
'  Dim i As Long
'  TotalPieceValue = 0
'
'  For i = 1 To NumPieces
'    TotalPieceValue = TotalPieceValue + PieceAbsValue(Board(Pieces(i)))
'  Next
'
'End Function

Public Function ResetMaterial() As Long
  Dim i As Long
  ResetMaterial = 0

  For i = 1 To NumPieces
    Material = Material + PieceScore(Board(Pieces(i)))
  Next

End Function

Public Sub FillKingCheckW()
  '--- Fill special board to speed up detection of checking moves in OrderMoves
  '--- direction to white king is set for queen directions and knights
  Dim i As Long, Target As Long, Offset As Long
  Erase KingCheckW()

  For i = 0 To 7
    Offset = DirectionOffset(i): Target = WKingLoc + Offset

    Do While Board(Target) <> FRAME '- not color critical: Opp piece can be captured, own piece can
    move away
      KingCheckW(Target) = Offset: If Board(Target) < NO_PIECE Then Exit Do Else
      Target = Target + Offset
    Loop

    Target = WKingLoc + KnightOffsets(i): If Board(Target) <> FRAME Then KingCheckW(
    Target) = KnightOffsets(i)
  Next

End Sub

Public Sub FillKingCheckB()
  '--- Fill special board to speed up detection of checking moves in OrderMoves
  '--- direction to black king is set for queen directions and knights
  Dim i As Long, Target As Long, Offset As Long
  Erase KingCheckB()

  For i = 0 To 7
    Offset = DirectionOffset(i): Target = BKingLoc + Offset

    Do While Board(Target) <> FRAME
      KingCheckB(Target) = Offset: If Board(Target) < NO_PIECE Then Exit Do Else
      Target = Target + Offset
    Loop

    Target = BKingLoc + KnightOffsets(i): If Board(Target) <> FRAME Then KingCheckB(
    Target) = KnightOffsets(i)
  Next

End Sub

Public Function IsBlockingMove(ThreatM As TMOVE, BlockM As TMOVE) As Boolean
  ' BlockM blocks TreatM ?
  IsBlockingMove = False
  If MaxDistance(ThreatM.From, ThreatM.Target) <= 1 Then Exit Function
  If ThreatM.Piece = WKNIGHT Or ThreatM.Piece = BKNIGHT Then Exit Function
  If BlockM.Piece = WKING Or BlockM.Piece = BKING Then Exit Function
  If SqBetween(BlockM.Target, ThreatM.From, ThreatM.Target) Then IsBlockingMove = True
End Function

'Public Function SeeSign(Move As TMOVE) As Long
'  SeeSign = 0
'  ' Early return if SEE cannot be negative because captured piece value
'  ' is not less then capturing one. Note that king moves always return
'  ' here
'  If Move.Castle > 0 Or Move.Target = 0 Or Move.Piece = NO_PIECE Or Board(Move.Target) = FRAME Then Exit Function
'  If PieceType(Move.Piece) = PT_KING Then SeeSign = VALUE_KNOWN_WIN: Exit Function   ' King move always good because legal checked before
'  If Move.SeeValue = VALUE_NONE Then
'    If PieceAbsValue(Move.Piece) + MAX_SEE_DIFF <= PieceAbsValue(Move.Captured) Then SeeSign = VALUE_KNOWN_WIN: Exit Function ' winning or equal  move
'    ' Calculate exchange score
'    Move.SeeValue = GetSEE(Move) ' Returned for future use
'  End If
'  SeeSign = Move.SeeValue
'End Function
'
'Public Function BadSEEMove(Move As TMOVE) As Boolean
'  BadSEEMove = False
'  If Move.Castle > 0 Or Move.Target = 0 Or Move.Piece = NO_PIECE Or Board(Move.Target) = FRAME Then Exit Function
'  If PieceType(Move.Piece) = PT_KING Then Exit Function   ' King move always good because legal checked before
'  If Move.SeeValue = VALUE_NONE Then
'    If PieceAbsValue(Move.Piece) + MAX_SEE_DIFF <= PieceAbsValue(Move.Captured) Then Exit Function ' winning or equal  move
'    Move.SeeValue = GetSEE(Move) ' Returned for future use
'  End If
'  BadSEEMove = (Move.SeeValue < -MAX_SEE_DIFF)
'End Function

'Public Function GoodSEEMove(Move As TMOVE) As Boolean
'  GoodSEEMove = True
'  If Move.Castle > 0 Or Move.Target = 0 Or Move.Piece = NO_PIECE Or Board(Move.Target) = FRAME Then Exit Function
'  If PieceType(Move.Piece) = PT_KING Then Exit Function   ' King move always good because legal checked before
'  If Move.SeeValue = VALUE_NONE Then
'    If PieceAbsValue(Move.Piece) + MAX_SEE_DIFF <= PieceAbsValue(Move.Captured) Then Exit Function ' winning or equal move
'    Move.SeeValue = GetSEE(Move) ' Returned for future use
'  End If
'  GoodSEEMove = (Move.SeeValue >= -MAX_SEE_DIFF)
'End Function

Public Function SEEGreaterOrEqual(Move As TMOVE, ByVal Score As Long) As Boolean
  '--- Optimized call of Static Exchange Evaluation (SEE): True if SEE greater or equal given Score
  SEEGreaterOrEqual = True
  If Move.Castle > 0 Or Move.Target = 0 Or Move.Piece = NO_PIECE Or Board(Move.Target)
   = FRAME Then Exit Function
  If PieceAbsValue(Move.Captured) < Score Then SEEGreaterOrEqual = False: Exit
  Function  ' only for positice 'score' values
  If PieceType(Move.Piece) = PT_KING Then Exit Function    ' King move always good because
  legal checked before
  If Move.SeeValue = VALUE_NONE Then
    If PieceAbsValue(Move.Captured) - PieceAbsValue(Move.Piece) >= Score -
```

```
      MAX_SEE_DIFF Then Exit Function ' winning or equal move
      Move.SeeValue = GetSEE(Move) ' Returned for future use
    End If
    SEEGreaterOrEqual = (Move.SeeValue >= Score - MAX_SEE_DIFF) ' MAX_SEE_DIFF to handle
    bishop equal knight
  End Function

  Public Function GetSEE(Move As TMOVE) As Long
    ' Returns piece score win for AttackColor ( positive for white or black).
    Dim i                As Long, From As Long, MoveTo As Long, Target As Long
    Dim CapturedVal      As Long, PieceMoved As Boolean
    Dim SideToMove       As enumColor, SideNotToMove As enumColor
    Dim NumAttackers(2) As Long, CurrSgn As Long, MinValIndex As Long, Piece As Long,
    Offset As Long
    '----
    GetSEE = 0
    If PieceType(Move.Piece) = PT_KING Then GetSEE = PieceAbsValue(Move.Captured): Exit
    Function
    If Move.Castle <> NO_CASTLE Then Exit Function
    From = Move.From: MoveTo = Move.Target: PieceMoved = CBool(Board(From) = NO_PIECE)
    If Not PieceMoved Then
      'If PinnedPieceDir(From, MoveTo, PieceColor(PieceMoved)) <> 0 Then GetSEE = -100000: Exit Function
      Piece = Board(From): Board(From) = NO_PIECE ' Remove piece to open sliding xrays
      If Move.EnPassant = ENPASSANT_CAPTURE Then  ' remove captured pawn not on target field
        If Piece = WPAWN Then Board(MoveTo + SQ_DOWN) = NO_PIECE Else Board(MoveTo +
        SQ_UP) = NO_PIECE
      End If
    Else
      Piece = Board(MoveTo)
    End If
    Cnt = 0 ' Counter for PieceList array of attackers (both sides)
    Erase Blocker  ' Array to manage blocker of sliding pieces: -1: is blocked, >0: is blocking,index of blocked
    piece, 0:not blocked/blocking

    ' Find attackers
    For i = 0 To 3 ' horizontal+vertical
      Block = 0: Offset = DirectionOffset(i): Target = MoveTo + Offset
      If Board(Target) = BKING Or Board(Target) = WKING Then
        Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target))
      Else

        Do While Board(Target) <> FRAME
          Select Case Board(Target)
            Case BROOK, BQUEEN, WROOK, WQUEEN
              Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target))
              If Block > 0 Then Blocker(Block) = Cnt: Blocker(Cnt) = -1 '- 1. point to blocked
              piece index; 2. -1 = blocked
              Block = Cnt
            Case NO_PIECE, WEP_PIECE, BEP_PIECE
              '-- Continue
            Case Else
              Exit Do ' other piece
          End Select

          Target = Target + Offset
        Loop

      End If
    Next

    For i = 4 To 7 ' diagonal
      Block = 0: Offset = DirectionOffset(i): Target = MoveTo + Offset

      Select Case Board(Target)
        Case BKING, WKING
          Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target))
          GoTo lblContinue
        Case WPAWN
```

```
          If i = 5 Or i = 7 Then Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target
          )): Block = Cnt: Target = Target + Offset
        Case BPAWN
          If i = 4 Or i = 6 Then Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target
          )): Block = Cnt: Target = Target + Offset
      End Select

      Do While Board(Target) <> FRAME
        Select Case Board(Target)
          Case BBISHOP, BQUEEN, WBISHOP, WQUEEN
            Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(Target))
            If Block > 0 Then Blocker(Block) = Cnt: Blocker(Cnt) = -1 '- 1. point to blocked
            piece index; 2. -1 = blocked
            Block = Cnt
          Case NO_PIECE, WEP_PIECE, BEP_PIECE
            '-- Continue
          Case Else
            Exit Do ' other piece
        End Select
        Target = Target + Offset
      Loop

  lblContinue:
    Next

    ' Knights
    If PieceCnt(WKNIGHT) + PieceCnt(BKNIGHT) > 0 Then
      For i = 0 To 7
        Select Case Board(MoveTo + KnightOffsets(i))
          Case WKNIGHT, BKNIGHT: Cnt = Cnt + 1: PieceList(Cnt) = PieceScore(Board(MoveTo
           + KnightOffsets(i)))
        End Select
      Next
    End If

    '---<<< End of collecting attackers ---
    ' Count Attackers for each color (non blocked only)
    For i = 1 To Cnt
      If PieceList(i) > 0 And Blocker(i) >= 0 Then NumAttackers(COL_WHITE) =
      NumAttackers(COL_WHITE) + 1 Else NumAttackers(COL_BLACK) = NumAttackers(COL_BLACK)
       + 1
    Next

    ' Init swap list
    SwapList(0) = PieceAbsValue(Move.Captured)
    slIndex = 1
    SideToMove = PieceColor(Move.Piece)
    ' Switch side
    SideNotToMove = SideToMove: If SideToMove = COL_WHITE Then SideToMove = COL_BLACK
    Else SideToMove = COL_WHITE
    ' If the opponent has no attackers we are finished
    If NumAttackers(SideToMove) = 0 Then
      GoTo lblEndSEE
    End If
    If SideToMove = COL_WHITE Then CurrSgn = 1 Else CurrSgn = -1
    '---- CALCULATE SEE ---
    CapturedVal = PieceAbsValue(Move.Piece)

    Do
      SwapList(slIndex) = -SwapList(slIndex - 1) + CapturedVal
      ' find least valuable attacker (min value)
      CapturedVal = 99999
      MinValIndex = -1

      For i = 1 To Cnt
        If PieceList(i) <> 0 Then If Sgn(PieceList(i)) = CurrSgn Then If Blocker(i) >= 0
         Then If Abs(PieceList(i)) < CapturedVal Then CapturedVal = Abs(PieceList(i)):
        MinValIndex = i
      Next

      If MinValIndex > 0 Then
        If Blocker(MinValIndex) > 0 Then ' unblock other sliding piece?
          Blocker(Blocker(MinValIndex)) = 0
          'Increase attack number
          If PieceList(Blocker(MinValIndex)) > 0 Then NumAttackers(COL_WHITE) =
          NumAttackers(COL_WHITE) + 1 Else NumAttackers(COL_BLACK) = NumAttackers(
          COL_BLACK) + 1
        End If
        PieceList(MinValIndex) = 0 ' Remove from list by setting piece value to zero
      End If
      If CapturedVal = 5000 Then ' King
        If NumAttackers(SideNotToMove) = 0 Then slIndex = slIndex + 1
        Exit Do ' King
      End If
      If CapturedVal = 99999 Then Exit Do
      NumAttackers(SideToMove) = NumAttackers(SideToMove) - 1
      CurrSgn = -CurrSgn: SideNotToMove = SideToMove: If SideToMove = COL_WHITE Then
      SideToMove = COL_BLACK Else SideToMove = COL_WHITE
      slIndex = slIndex + 1
    Loop While NumAttackers(SideToMove) > 0

    '// Having built the swap list, we negamax through it to find the best
    ' // achievable score from the point of view of the side to move.
    slIndex = slIndex - 1

    Do While slIndex > 0
      'SwapList(slIndex - 1) = GetMin(-SwapList(slIndex), SwapList(slIndex - 1))
      If -SwapList(slIndex) < SwapList(slIndex - 1) Then SwapList(slIndex - 1) = -
      SwapList(slIndex)
      slIndex = slIndex - 1
    Loop

  lblEndSEE:
    If Not PieceMoved Then
      Board(From) = Piece
      If Move.EnPassant = ENPASSANT_CAPTURE Then  ' restore captured pawn not on target field
        If Piece = WPAWN Then Board(MoveTo + SQ_DOWN) = BPAWN Else Board(MoveTo + SQ_UP)
         = WPAWN
      End If
    End If
    GetSEE = SwapList(0)
  End Function

  Public Sub InitPieceColor()
    Dim Piece As Long, PieceCol As Long

    For Piece = 0 To 16
      If Piece < 1 Or Piece >= NO_PIECE Then
        PieceCol = COL_NOPIECE ' NO_PIECE, or EP-PIECE  or FRAME
      Else
        If (Piece And 1) = WCOL Then PieceCol = COL_WHITE Else PieceCol = COL_BLACK
      End If
      PieceColor(Piece) = PieceCol
    Next

  End Sub

  'Public Function SwitchColor(Color As enumColor) As enumColor
  '  If Color = COL_WHITE Then SwitchColor = COL_BLACK Else SwitchColor = COL_WHITE
  'End Function

  Public Sub InitSameXRay()
    Dim i As Long, j As Long

    For i = SQ_A1 To SQ_H8
      If File(i) >= 1 And File(i) <= 8 And Rank(i) >= 1 And Rank(i) <= 8 Then
      DirOffset(i, j) = 0
      For j = SQ_A1 To SQ_H8
        If File(j) >= 1 And File(j) <= 8 And Rank(j) >= 1 And Rank(j) <= 8 Then
          If File(i) = File(j) Then
            SameXRay(i, j) = True
            If i < j Then DirOffset(i, j) = 10 Else If i > j Then DirOffset(i, j) = -10
          ElseIf Rank(i) = Rank(j) Then
            SameXRay(i, j) = True
            If i < j Then DirOffset(i, j) = 1 Else If i > j Then DirOffset(i, j) = -1
          ElseIf Abs(File(i) - File(j)) = Abs(Rank(i) - Rank(j)) Then
            SameXRay(i, j) = True
            If Abs(j - i) Mod 9 = 0 Then
              If i < j Then DirOffset(i, j) = 9 Else If i > j Then DirOffset(i, j) = -9
            Else
              If i < j Then DirOffset(i, j) = 11 Else If i > j Then DirOffset(i, j) = -11
            End If
          Else
            SameXRay(i, j) = False
          End If
        End If

       ' If Abs(DirOffset(i, j)) = 10 Or Abs(DirOffset(i, j)) = 1 Then SameRookRay(i, j) = True
       ' If Abs(DirOffset(i, j)) = 9 Or Abs(DirOffset(i, j)) = 11 Then SameBishopRay(i, j) = True
      Next

    End If
  Next

End Sub

Public Function IsCheckingMove(ByVal PieceFrom As Long, _
                               ByVal From As Long, _
                               ByVal Target As Long, _
                               ByVal Promoted As Long, ByVal EnPassant As Long) As Boolean
  ' is this a checking move?
  ' array KingCheckW/B must be set before with function FillKingCheckW / FillKingCheckB (fast detection logic)
  Dim bFound As Boolean, Offset As Long, SlidePos As Long, EpSquare As Long
  bFound = False: EpSquare = 0

  ' --------------- White piece moves -----------------------------------------------------
  If (PieceFrom And 1) = WCOL Then
    If Promoted > 0 Then
      PieceFrom = Promoted: If SqBetween(From, BKingLoc, Target) Then Target = From '--- to get KingCheck array offset
    ElseIf EnPassant = ENPASSANT_CAPTURE Then
      EpSquare = Target + SQ_DOWN
    ElseIf PieceFrom = WKING Then
      ' Castling check?
      If From = WKING_START Then
        If Target = SQ_G1 Then '00
          Target = SQ_F1: PieceFrom = WROOK
        ElseIf Target = SQ_C1 Then '000
          Target = SQ_D1: PieceFrom = WROOK
        End If
      End If
    End If

    If KingCheckB(From) = 0 Then If KingCheckB(Target) = 0 Then If KingCheckB(EpSquare) = 0 Then IsCheckingMove = False: Exit Function

    Select Case KingCheckB(Target)
      Case 0:  ' ignore
      Case -9, -11:
        If PieceFrom = WPAWN Then
```

```
          If MaxDistance(Target, BKingLoc) = 1 Then bFound = True
        ElseIf PieceFrom = WQUEEN Or PieceFrom = WBISHOP Then
          bFound = True
        End If
      Case 9, 11: If PieceFrom = WQUEEN Or PieceFrom = WBISHOP Then bFound = True
      Case 1, -1, 10, -10: If PieceFrom = WQUEEN Or PieceFrom = WROOK Then bFound = True
      Case 8, -8, 12, -12, 19, -19, 21, -21: If PieceFrom = WKNIGHT Then bFound = True
    End Select

    If Not bFound Then
      '--- Sliding Check?  also continue loop for EnPassant square
      Offset = KingCheckB(From): SlidePos = From
      Do
        Select Case Abs(Offset)
          Case 0, 8, 12, 19, 21: 'empty or Knight> ignore
          Case Else
            If SqBetween(SlidePos, BKingLoc, Target) Then '--- ignore if move in same direction towards king
              ' ignore
            ElseIf SqBetween(Target, BKingLoc, SlidePos) Then  '--- ignore if move in same direction towards king
              ' ignore
            Else
              Select Case Abs(Offset)  ' check needed?
                Case 1, 10: If PieceCnt(WROOK) + PieceCnt(WQUEEN) + Promoted = 0 Then GoTo lblNextWSq
                Case 9, 11: If PieceCnt(WBISHOP) + PieceCnt(WQUEEN) + Promoted = 0 Then GoTo lblNextWSq
              End Select
              Do ' search for piece or border
                SlidePos = SlidePos + Offset
                Select Case Board(SlidePos)
                  Case NO_PIECE, WEP_PIECE, BEP_PIECE: ' - go on
                  Case FRAME: Exit Do
                  Case WQUEEN: bFound = True
                    Exit Do
                  Case WROOK: If Abs(Offset) = 10 Or Abs(Offset) = 1 Then bFound = True
                    Exit Do
                  Case WBISHOP: If Abs(Offset) = 9 Or Abs(Offset) = 11 Then bFound = True
                    Exit Do
                  Case Else
                    Exit Do
                End Select
              Loop
            End If
          End Select
          If bFound Then Exit Do
lblNextWSq:
          If EpSquare = 0 Then Exit Do
          '--- additonal EP - Check
          If SqBetween(EpSquare, BKingLoc, From) Then ' King, EpSquare, attacker in same row
            ' Fix for position "8/8/8/3kPpR1/8/8/8/4K3 w - f6 0 1" Enpassant e5xf6ep/ changed2023' debug.print printpos, LocCoord(from),LocCoord(target),LocCoord(EpSquare), KingCheckB(EpSquare)
            Offset = KingCheckB(EpSquare): If Offset <> 0 Then SlidePos = From
          ElseIf SqBetween(From, BKingLoc, EpSquare) Then
            ' Fix for position : 2. case "8/8/8/1k1pP1R1/8/8/8/4K3 w - d6 0 1"" Enpassant d5xc6ep/ changed2023
            Offset = KingCheckB(From): If Offset <> 0 Then SlidePos = EpSquare
          ElseIf KingCheckB(EpSquare) = 0 Then
            Exit Do
          Else
            Offset = KingCheckB(EpSquare): If Offset <> 0 Then SlidePos = EpSquare Else Exit Do ' do a second loop behind EpSquare
          End If
          EpSquare = 0
      Loop '----- search for slider check

    End If

' -------------- Black piece moves ----------------------------------------
  ElseIf (PieceFrom And 1) = BCOL Then
    If Promoted > 0 Then
      PieceFrom = Promoted: If SqBetween(From, WKingLoc, Target) Then Target = From
      '--- to get KingCheck array offset
    ElseIf EnPassant = ENPASSANT_CAPTURE Then
      EpSquare = Target + SQ_UP
    ElseIf PieceFrom = BKING Then
      ' Castling check?
      If From = BKING_START Then
        If Target = SQ_G8 Then '00
          Target = SQ_F8: PieceFrom = BROOK
        ElseIf Target = SQ_C8 Then '000
          Target = SQ_D8: PieceFrom = BROOK
        End If
      End If
    End If
    If KingCheckW(From) = 0 Then If KingCheckW(Target) = 0 Then If KingCheckW(EpSquare
    ) = 0 Then IsCheckingMove = False: Exit Function

    Select Case KingCheckW(Target)
      Case 0:  ' ignore
      Case 9, 11:
        If PieceFrom = BPAWN Then
          If MaxDistance(Target, WKingLoc) = 1 Then bFound = True
        ElseIf PieceFrom = BQUEEN Or PieceFrom = BBISHOP Then
          bFound = True
        End If
      Case -9, -11: If PieceFrom = BQUEEN Or PieceFrom = BBISHOP Then bFound = True
      Case 1, -1, 10, -10: If PieceFrom = BQUEEN Or PieceFrom = BROOK Then bFound =
      True
      Case 8, -8, 12, -12, 19, -19, 21, -21: If PieceFrom = BKNIGHT Then bFound = True
    End Select

    If Not bFound Then
      '--- Sliding Check? also continue loop for EnPassant square
      Offset = KingCheckW(From): SlidePos = From
      Do
        Select Case Abs(Offset)
          Case 0, 8, 12, 19, 21: 'empty or Knight> ignore
          Case Else
            If SqBetween(SlidePos, WKingLoc, Target) Then '--- ignore if move in same direction
            towards king
              ' ignore
            ElseIf SqBetween(Target, WKingLoc, SlidePos) Then  '--- ignore if move in same
            direction towards king
              ' ignore
            Else

              Select Case Abs(Offset)  ' check needed?
                Case 1, 10: If PieceCnt(BROOK) + PieceCnt(BQUEEN) + Promoted = 0 Then
                GoTo lblNextBSq
                Case 9, 11: If PieceCnt(BBISHOP) + PieceCnt(BQUEEN) + Promoted = 0
                Then GoTo lblNextBSq
              End Select

              Do
                SlidePos = SlidePos + Offset
                Select Case Board(SlidePos)
                  Case NO_PIECE, WEP_PIECE, BEP_PIECE: '- go on
                  Case FRAME: Exit Do
                  Case BQUEEN: bFound = True
                    Exit Do
                  Case BROOK: If Abs(Offset) = 10 Or Abs(Offset) = 1 Then bFound =
                  True
```

```vb
          Exit Do
         Case BBISHOP: If Abs(Offset) = 9 Or Abs(Offset) = 11 Then bFound = True
          Exit Do
         Case Else
          Exit Do
        End Select
       Loop
      End If
     End Select
     If bFound Then Exit Do
lblNextBSq:
     If EpSquare = 0 Then Exit Do
     '--- additonial EP - Check
     If SqBetween(EpSquare, WKingLoc, From) Then ' King, EpSquare, attacker in same row
      Offset = KingCheckW(EpSquare): If Offset <> 0 Then SlidePos = From
     ElseIf SqBetween(From, WKingLoc, EpSquare) Then
      Offset = KingCheckW(From): If Offset <> 0 Then SlidePos = EpSquare
     ElseIf KingCheckW(EpSquare) = 0 Then
      Exit Do
     Else
      Offset = KingCheckW(EpSquare): If Offset <> 0 Then SlidePos = EpSquare Else Exit Do ' do a second loop behind EpSquare
     End If
     EpSquare = 0
   Loop '----- search for slider check
  End If
 End If
 IsCheckingMove = bFound
 'If bFound And EnPassant = ENPASSANT_CAPTURE And Target <> 74 Then Stop
End Function

Public Sub InitBoardColors()
 Dim x As Long, y As Long, ColSq  As Long, IsWhite As Boolean

 For y = 1 To 8
  IsWhite = CBool((y And 1) = 0)

  For x = 1 To 8
   If IsWhite Then ColSq = COL_WHITE Else ColSq = COL_BLACK
   ColorSq(20 + x + (y - 1) * 10) = ColSq
   IsWhite = Not IsWhite
  Next
 Next

End Sub

Public Function CoordToLoc(ByVal isCoord As String) As Long
 ' "A1" => 21  ( board array index )
 If Len(isCoord) = 2 Then
  CoordToLoc = 10 + Asc(Left$(LCase$(isCoord), 1)) - 96 + Val(Mid$(isCoord, 2)) * 10
 Else
  CoordToLoc = 0
 End If
End Function

Public Function MovesEqual(m1 As TMOVE, m2 As TMOVE) As Boolean
 MovesEqual = False 'same moves?
 If m1.From = m2.From Then If m1.Target = m2.Target Then If m1.Piece = m2.Piece Then If m1.Promoted = m2.Promoted Then MovesEqual = True
End Function

Public Function WCanCastleOO() As Boolean
 ' not checked for attacked squares
 WCanCastleOO = False
 If Moved(WKING_START) = 0 Then If Moved(SQ_H1) = 0 Then If Board(SQ_H1) = WROOK Then If Board(SQ_F1) = NO_PIECE And Board(SQ_G1) = NO_PIECE Then WCanCastleOO = True
End Function
```

```
Public Function WCanCastleOOO() As Boolean
  ' not checked for attacked squares
  WCanCastleOOO = False
  If Moved(WKING_START) = 0 Then If Moved(SQ_A1) = 0 Then If Board(SQ_A1) = WROOK Then
   If Board(SQ_B1) = NO_PIECE And Board(SQ_C1) = NO_PIECE And Board(SQ_D1) = NO_PIECE
  Then WCanCastleOOO = True
End Function

Public Function BCanCastleOO() As Boolean
  ' not checked for attacked squares
  BCanCastleOO = False
  If Moved(BKING_START) = 0 Then If Moved(SQ_H8) = 0 Then If Board(SQ_H8) = BROOK Then
   If Board(SQ_F8) = NO_PIECE And Board(SQ_G8) = NO_PIECE Then BCanCastleOO = True
End Function

Public Function BCanCastleOOO() As Boolean
  ' not checked for attacked squares
  BCanCastleOOO = False
  If Moved(BKING_START) = 0 Then If Moved(SQ_A8) = 0 Then If Board(SQ_A8) = BROOK Then
   If Board(SQ_B8) = NO_PIECE And Board(SQ_C8) = NO_PIECE And Board(SQ_D8) = NO_PIECE
  Then BCanCastleOOO = True
End Function


Public Function GetMoveFromSAN(ByVal isSAN As String) As TMOVE
  ' read Standard Algebraic Notation like "Rexd1"
  Dim SANMove As TMOVE, FileFrom As Long, RankFrom As Long
  GetMoveFromSAN = EmptyMove
  isSAN = Trim$(isSAN)
  isSAN = Replace(isSAN, "x", "")
  isSAN = Replace(isSAN, "+", "")
  isSAN = Replace(isSAN, "-", "")
  isSAN = Replace(isSAN, "=", "")
  isSAN = Replace(isSAN, "#", "")
  isSAN = Replace(isSAN, "e.p.", "")
  If isSAN = "" Then Exit Function
  SANMove = EmptyMove

  ' Get piece type
  Select Case Left$(isSAN, 1)
    Case "K": isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WKING Else
    SANMove.Piece = BKING
    Case "O", "o": 'Castle
      isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WKING Else
      SANMove.Piece = BKING
      If UCase$(Left$(isSAN, 2)) = "OO" Then
        If bWhiteToMove Then
          SANMove.From = SQ_E1: SANMove.Target = SQ_G1: SANMove.Castle = WHITEOO
        Else
          SANMove.From = SQ_E8: SANMove.Target = SQ_G8: SANMove.Castle = BLACKOO
        End If
      ElseIf UCase$(Left$(isSAN, 3)) = "OOO" Then
        If bWhiteToMove Then
          SANMove.From = SQ_E8: SANMove.Target = SQ_C8: SANMove.Castle = WHITEOOO
        Else
          SANMove.From = SQ_E8: SANMove.Target = SQ_G8: SANMove.Castle = BLACKOOO
        End If
      Else
        Exit Function
      End If
      GoTo lblTestMoves
    Case "B": isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WBISHOP
    Else SANMove.Piece = BBISHOP
    Case "N": isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WKNIGHT
    Else SANMove.Piece = BKNIGHT
    Case "R": isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WROOK Else
    SANMove.Piece = BROOK
```

```
      Case "Q": isSAN = Mid$(isSAN, 2): If bWhiteToMove Then SANMove.Piece = WQUEEN Else
       SANMove.Piece = BQUEEN
      Case "a" To "h": If bWhiteToMove Then SANMove.Piece = WPAWN Else SANMove.Piece =
      BPAWN
      Case Else
        Exit Function
    End Select

    ' d5 or ed5 or 1d5 or d8d5
    FileFrom = 0: RankFrom = 0
    If IsNumeric(Mid$(isSAN, 4, 1)) And IsNumeric(Mid$(isSAN, 2, 1)) Then
      ' d8d5
      SANMove.From = FileRev(Left$(isSAN, 1)) + RankRev(Mid$(isSAN, 2, 1))
      isSAN = Mid$(isSAN, 3)
    ElseIf IsNumeric(Mid$(isSAN, 3, 1)) Then
      ' ed5 or 1d5
      If IsNumeric(Left$(isSAN, 1)) Then
        RankFrom = RankRev(Left$(isSAN, 1))
      Else
        FileFrom = FileRev(Left$(isSAN, 1))
      End If
      isSAN = Mid$(isSAN, 2)
    End If
    ' Get target square
    SANMove.Target = FileRev(Left$(isSAN, 1)) + RankRev(Mid$(isSAN, 2, 1))
    isSAN = Trim$(Mid$(isSAN, 3))
    If isSAN <> "" Then ' Promote:  e8=Q

      Select Case Left$(isSAN, 1)
        Case "B": If bWhiteToMove Then SANMove.Promoted = WBISHOP Else SANMove.Promoted
        = BBISHOP
        Case "N": If bWhiteToMove Then SANMove.Promoted = WKNIGHT Else SANMove.Promoted
        = BKNIGHT
        Case "R": If bWhiteToMove Then SANMove.Promoted = WROOK Else SANMove.Promoted =
        BROOK
        Case "Q": If bWhiteToMove Then SANMove.Promoted = WQUEEN Else SANMove.Promoted =
         BQUEEN
      End Select

      If SANMove.Promoted > 0 Then SANMove.Piece = SANMove.Promoted
    End If
  lblTestMoves:
    Dim iNumMoves As Long, i As Long, bLegalInput As Boolean
    GenerateMoves Ply, False, iNumMoves

    ' find move
    For i = 0 To iNumMoves - 1
      If SANMove.Piece = Moves(Ply, i).Piece And SANMove.Target = Moves(Ply, i).Target
      Then
        If SANMove.From > 0 Then If SANMove.From <> Moves(Ply, i).From Then GoTo
        lblNextMove
        If FileFrom > 0 Then If FileFrom <> File(Moves(Ply, i).From) Then GoTo
        lblNextMove
        If RankFrom > 0 Then If RankFrom <> Rank(Moves(Ply, i).From) Then GoTo
        lblNextMove
        If SANMove.Promoted > 0 Then If SANMove.Promoted <> Moves(Ply, i).Promoted Then
        GoTo lblNextMove
        ' Ok, check if legal move
        RemoveEpPiece
        MakeMove Moves(Ply, i)
        If CheckLegal(Moves(Ply, i)) Then
          bLegalInput = True
          GetMoveFromSAN = Moves(Ply, i) ' found!
        End If
        UnmakeMove Moves(Ply, i)
        ResetEpPiece
      End If
  lblNextMove:
      If bLegalInput Then Exit For
    Next

End Function


''--- Bit functions ---
'' many lines of codes, but very fast
'Public Function BitsShiftLeft(ByVal Value As Long, ByVal ShiftCount As Long) As Long
'
'  '- Shifts the bits to the left the specified number of positions and returns the new value.
'  '- Bits "falling off" the left edge do not wrap around. Fill bits coming in from right are 0.
'  '- A shift left is effectively a multiplication by 2. Some common languages like C/C++ or Java have an operator for this job: "<<".
'  Select Case ShiftCount
'    Case 0&
'      BitsShiftLeft = Value
'    Case 1&
'      If Value And &H40000000 Then
'        BitsShiftLeft = (Value And &H3FFFFFFF) * &H2& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H3FFFFFFF) * &H2&
'      End If
'    Case 2&
'      If Value And &H20000000 Then
'        BitsShiftLeft = (Value And &H1FFFFFFF) * &H4& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H1FFFFFFF) * &H4&
'      End If
'    Case 3&
'      If Value And &H10000000 Then
'        BitsShiftLeft = (Value And &HFFFFFFF) * &H8& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &HFFFFFFF) * &H8&
'      End If
'    Case 4&
'      If Value And &H8000000 Then
'        BitsShiftLeft = (Value And &H7FFFFFF) * &H10& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H7FFFFFF) * &H10&
'      End If
'    Case 5&
'      If Value And &H4000000 Then
'        BitsShiftLeft = (Value And &H3FFFFFF) * &H20& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H3FFFFFF) * &H20&
'      End If
'    Case 6&
'      If Value And &H2000000 Then
'        BitsShiftLeft = (Value And &H1FFFFFF) * &H40& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H1FFFFFF) * &H40&
'      End If
'    Case 7&
'      If Value And &H1000000 Then
'        BitsShiftLeft = (Value And &HFFFFFF) * &H80& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &HFFFFFF) * &H80&
'      End If
'    Case 8&
'      If Value And &H800000 Then
'        BitsShiftLeft = (Value And &H7FFFFF) * &H100& Or &H80000000
'      Else
'        BitsShiftLeft = (Value And &H7FFFFF) * &H100&
'      End If
'    Case 9&
'      If Value And &H400000 Then
'        BitsShiftLeft = (Value And &H3FFFFF) * &H200& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H3FFFFF) * &H200&
'     End If
'   Case 10&
'     If Value And &H200000 Then
'       BitsShiftLeft = (Value And &H1FFFFF) * &H400& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H1FFFFF) * &H400&
'     End If
'   Case 11&
'     If Value And &H100000 Then
'       BitsShiftLeft = (Value And &HFFFFF) * &H800& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &HFFFFF) * &H800&
'     End If
'   Case 12&
'     If Value And &H80000 Then
'       BitsShiftLeft = (Value And &H7FFFF) * &H1000& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H7FFFF) * &H1000&
'     End If
'   Case 13&
'     If Value And &H40000 Then
'       BitsShiftLeft = (Value And &H3FFFF) * &H2000& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H3FFFF) * &H2000&
'     End If
'   Case 14&
'     If Value And &H20000 Then
'       BitsShiftLeft = (Value And &H1FFFF) * &H4000& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H1FFFF) * &H4000&
'     End If
'   Case 15&
'     If Value And &H10000 Then
'       BitsShiftLeft = (Value And &HFFFF&) * &H8000& Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &HFFFF&) * &H8000&
'     End If
'   Case 16&
'     If Value And &H8000& Then
'       BitsShiftLeft = (Value And &H7FFF&) * &H10000 Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H7FFF&) * &H10000
'     End If
'   Case 17&
'     If Value And &H4000& Then
'       BitsShiftLeft = (Value And &H3FFF&) * &H20000 Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H3FFF&) * &H20000
'     End If
'   Case 18&
'     If Value And &H2000& Then
'       BitsShiftLeft = (Value And &H1FFF&) * &H40000 Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H1FFF&) * &H40000
'     End If
'   Case 19&
'     If Value And &H1000& Then
'       BitsShiftLeft = (Value And &HFFF&) * &H80000 Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &HFFF&) * &H80000
'     End If
'   Case 20&
'     If Value And &H800& Then
'       BitsShiftLeft = (Value And &H7FF&) * &H100000 Or &H80000000
'     Else
'       BitsShiftLeft = (Value And &H7FF&) * &H100000
'   End If
'  Case 21&
'   If Value And &H400& Then
'    BitsShiftLeft = (Value And &H3FF&) * &H200000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H3FF&) * &H200000
'   End If
'  Case 22&
'   If Value And &H200& Then
'    BitsShiftLeft = (Value And &H1FF&) * &H400000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H1FF&) * &H400000
'   End If
'  Case 23&
'   If Value And &H100& Then
'    BitsShiftLeft = (Value And &HFF&) * &H800000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &HFF&) * &H800000
'   End If
'  Case 24&
'   If Value And &H80& Then
'    BitsShiftLeft = (Value And &H7F&) * &H1000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H7F&) * &H1000000
'   End If
'  Case 25&
'   If Value And &H40& Then
'    BitsShiftLeft = (Value And &H3F&) * &H2000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H3F&) * &H2000000
'   End If
'  Case 26&
'   If Value And &H20& Then
'    BitsShiftLeft = (Value And &H1F&) * &H4000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H1F&) * &H4000000
'   End If
'  Case 27&
'   If Value And &H10& Then
'    BitsShiftLeft = (Value And &HF&) * &H8000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &HF&) * &H8000000
'   End If
'  Case 28&
'   If Value And &H8& Then
'    BitsShiftLeft = (Value And &H7&) * &H10000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H7&) * &H10000000
'   End If
'  Case 29&
'   If Value And &H4& Then
'    BitsShiftLeft = (Value And &H3&) * &H20000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H3&) * &H20000000
'   End If
'  Case 30&
'   If Value And &H2& Then
'    BitsShiftLeft = (Value And &H1&) * &H40000000 Or &H80000000
'   Else
'    BitsShiftLeft = (Value And &H1&) * &H40000000
'   End If
'  Case 31&
'   If Value And &H1& Then
'    BitsShiftLeft = &H80000000
'   Else
'    BitsShiftLeft = &H0&
'   End If
' End Select
```

```
'
'End Function
'
'Public Function BitsShiftRight(ByVal Value As Long, ByVal ShiftCount As Long) As Long
'
'  ' Shifts the bits to the right the specified number of positions and returns the new value.
'  ' Bits "falling off" the right edge do not wrap around. Fill bits coming in from left match bit 31 (the sign bit): if bit 31 is 1 the fill bits will be 1 (see ShiftRightZ for the alternative zero-fill-in version).
'  ' A shift right is effectively a division by 2 (rounding downward, see Examples). Some common languages like C/C++ or Java have an operator for this job: ">>"
'  Select Case ShiftCount
'    Case 0&:  BitsShiftRight = Value
'    Case 1&:  BitsShiftRight = (Value And &HFFFFFFFE) \ &H2&
'    Case 2&:  BitsShiftRight = (Value And &HFFFFFFFC) \ &H4&
'    Case 3&:  BitsShiftRight = (Value And &HFFFFFFF8) \ &H8&
'    Case 4&:  BitsShiftRight = (Value And &HFFFFFFF0) \ &H10&
'    Case 5&:  BitsShiftRight = (Value And &HFFFFFFE0) \ &H20&
'    Case 6&:  BitsShiftRight = (Value And &HFFFFFFC0) \ &H40&
'    Case 7&:  BitsShiftRight = (Value And &HFFFFFF80) \ &H80&
'    Case 8&:  BitsShiftRight = (Value And &HFFFFFF00) \ &H100&
'    Case 9&:  BitsShiftRight = (Value And &HFFFFFE00) \ &H200&
'    Case 10&: BitsShiftRight = (Value And &HFFFFFC00) \ &H400&
'    Case 11&: BitsShiftRight = (Value And &HFFFFF800) \ &H800&
'    Case 12&: BitsShiftRight = (Value And &HFFFFF000) \ &H1000&
'    Case 13&: BitsShiftRight = (Value And &HFFFFE000) \ &H2000&
'    Case 14&: BitsShiftRight = (Value And &HFFFFC000) \ &H4000&
'    Case 15&: BitsShiftRight = (Value And &HFFFF8000) \ &H8000&
'    Case 16&: BitsShiftRight = (Value And &HFFFF0000) \ &H10000
'    Case 17&: BitsShiftRight = (Value And &HFFFE0000) \ &H20000
'    Case 18&: BitsShiftRight = (Value And &HFFFC0000) \ &H40000
'    Case 19&: BitsShiftRight = (Value And &HFFF80000) \ &H80000
'    Case 20&: BitsShiftRight = (Value And &HFFF00000) \ &H100000
'    Case 21&: BitsShiftRight = (Value And &HFFE00000) \ &H200000
'    Case 22&: BitsShiftRight = (Value And &HFFC00000) \ &H400000
'    Case 23&: BitsShiftRight = (Value And &HFF800000) \ &H800000
'    Case 24&: BitsShiftRight = (Value And &HFF000000) \ &H1000000
'    Case 25&: BitsShiftRight = (Value And &HFE000000) \ &H2000000
'    Case 26&: BitsShiftRight = (Value And &HFC000000) \ &H4000000
'    Case 27&: BitsShiftRight = (Value And &HF8000000) \ &H8000000
'    Case 28&: BitsShiftRight = (Value And &HF0000000) \ &H10000000
'    Case 29&: BitsShiftRight = (Value And &HE0000000) \ &H20000000
'    Case 30&: BitsShiftRight = (Value And &HC0000000) \ &H40000000
'    Case 31&: BitsShiftRight = CBool(Value And &H80000000)
'  End Select
'
'End Function
'
'Public Function BitsShiftRightZ(ByVal Value As Long, ByVal ShiftCount As Long) As Long
'  '- Shifts the bits to the right the specified number of positions and returns the new value.
'  '- Bits "falling off" the right edge do not wrap around. Fill bits coming in from left are 0 (zero, hence "ShiftRightZ", see ShiftRight for the alternative signbit-fill-in version)
'  If Value And &H80000000 Then
'
'    Select Case ShiftCount
'      Case 0&:  BitsShiftRightZ = Value
'      Case 1&:  BitsShiftRightZ = &H40000000 Or (Value And &H7FFFFFFF) \ &H2&
'      Case 2&:  BitsShiftRightZ = &H20000000 Or (Value And &H7FFFFFFF) \ &H4&
'      Case 3&:  BitsShiftRightZ = &H10000000 Or (Value And &H7FFFFFFF) \ &H8&
'      Case 4&:  BitsShiftRightZ = &H8000000 Or (Value And &H7FFFFFFF) \ &H10&
'      Case 5&:  BitsShiftRightZ = &H4000000 Or (Value And &H7FFFFFFF) \ &H20&
'      Case 6&:  BitsShiftRightZ = &H2000000 Or (Value And &H7FFFFFFF) \ &H40&
'      Case 7&:  BitsShiftRightZ = &H1000000 Or (Value And &H7FFFFFFF) \ &H80&
'      Case 8&:  BitsShiftRightZ = &H800000 Or (Value And &H7FFFFFFF) \ &H100&
'      Case 9&:  BitsShiftRightZ = &H400000 Or (Value And &H7FFFFFFF) \ &H200&
'      Case 10&: BitsShiftRightZ = &H200000 Or (Value And &H7FFFFFFF) \ &H400&
'      Case 11&: BitsShiftRightZ = &H100000 Or (Value And &H7FFFFFFF) \ &H800&
'      Case 12&: BitsShiftRightZ = &H80000 Or (Value And &H7FFFFFFF) \ &H1000&
'      Case 13&: BitsShiftRightZ = &H40000 Or (Value And &H7FFFFFFF) \ &H2000&
'     Case 14&: BitsShiftRightZ = &H20000 Or (Value And &H7FFFFFFF) \ &H4000&
'     Case 15&: BitsShiftRightZ = &H10000 Or (Value And &H7FFFFFFF) \ &H8000&
'     Case 16&: BitsShiftRightZ = &H8000& Or (Value And &H7FFFFFFF) \ &H10000
'     Case 17&: BitsShiftRightZ = &H4000& Or (Value And &H7FFFFFFF) \ &H20000
'     Case 18&: BitsShiftRightZ = &H2000& Or (Value And &H7FFFFFFF) \ &H40000
'     Case 19&: BitsShiftRightZ = &H1000& Or (Value And &H7FFFFFFF) \ &H80000
'     Case 20&: BitsShiftRightZ = &H800& Or (Value And &H7FFFFFFF) \ &H100000
'     Case 21&: BitsShiftRightZ = &H400& Or (Value And &H7FFFFFFF) \ &H200000
'     Case 22&: BitsShiftRightZ = &H200& Or (Value And &H7FFFFFFF) \ &H400000
'     Case 23&: BitsShiftRightZ = &H100& Or (Value And &H7FFFFFFF) \ &H800000
'     Case 24&: BitsShiftRightZ = &H80& Or (Value And &H7FFFFFFF) \ &H1000000
'     Case 25&: BitsShiftRightZ = &H40& Or (Value And &H7FFFFFFF) \ &H2000000
'     Case 26&: BitsShiftRightZ = &H20& Or (Value And &H7FFFFFFF) \ &H4000000
'     Case 27&: BitsShiftRightZ = &H10& Or (Value And &H7FFFFFFF) \ &H8000000
'     Case 28&: BitsShiftRightZ = &H8& Or (Value And &H7FFFFFFF) \ &H10000000
'     Case 29&: BitsShiftRightZ = &H4& Or (Value And &H7FFFFFFF) \ &H20000000
'     Case 30&: BitsShiftRightZ = &H2& Or (Value And &H7FFFFFFF) \ &H40000000
'     Case 31&: BitsShiftRightZ = &H1&
'   End Select
'
' Else
'
'   Select Case ShiftCount
'     Case 0&:  BitsShiftRightZ = Value
'     Case 1&:  BitsShiftRightZ = Value \ &H2&
'     Case 2&:  BitsShiftRightZ = Value \ &H4&
'     Case 3&:  BitsShiftRightZ = Value \ &H8&
'     Case 4&:  BitsShiftRightZ = Value \ &H10&
'     Case 5&:  BitsShiftRightZ = Value \ &H20&
'     Case 6&:  BitsShiftRightZ = Value \ &H40&
'     Case 7&:  BitsShiftRightZ = Value \ &H80&
'     Case 8&:  BitsShiftRightZ = Value \ &H100&
'     Case 9&:  BitsShiftRightZ = Value \ &H200&
'     Case 10&: BitsShiftRightZ = Value \ &H400&
'     Case 11&: BitsShiftRightZ = Value \ &H800&
'     Case 12&: BitsShiftRightZ = Value \ &H1000&
'     Case 13&: BitsShiftRightZ = Value \ &H2000&
'     Case 14&: BitsShiftRightZ = Value \ &H4000&
'     Case 15&: BitsShiftRightZ = Value \ &H8000&
'     Case 16&: BitsShiftRightZ = Value \ &H10000
'     Case 17&: BitsShiftRightZ = Value \ &H20000
'     Case 18&: BitsShiftRightZ = Value \ &H40000
'     Case 19&: BitsShiftRightZ = Value \ &H80000
'     Case 20&: BitsShiftRightZ = Value \ &H100000
'     Case 21&: BitsShiftRightZ = Value \ &H200000
'     Case 22&: BitsShiftRightZ = Value \ &H400000
'     Case 23&: BitsShiftRightZ = Value \ &H800000
'     Case 24&: BitsShiftRightZ = Value \ &H1000000
'     Case 25&: BitsShiftRightZ = Value \ &H2000000
'     Case 26&: BitsShiftRightZ = Value \ &H4000000
'     Case 27&: BitsShiftRightZ = Value \ &H8000000
'     Case 28&: BitsShiftRightZ = Value \ &H10000000
'     Case 29&: BitsShiftRightZ = Value \ &H20000000
'     Case 30&: BitsShiftRightZ = Value \ &H40000000
'     Case 31&: BitsShiftRightZ = &H0&
'   End Select
'
' End If
'End Function



'Public Function PopCount(ByVal x As Long) As Long
' ' for positive values only
' Debug.Assert x >= 0
'
' PopCount = 0
' Do While x > 0
'   PopCount = PopCount + 1: x = x And (x - 1)
' Loop
'End Function
'
'Public Function And64(Op1 As TBit64, Op2 As TBit64) As TBit64
' And64.i0 = Op1.i0 And Op2.i0
' And64.i1 = Op1.i1 And Op2.i1
' And64.i2 = Op1.i2 And Op2.i2
' And64.i3 = Op1.i3 And Op2.i3
'End Function
'
'Public Function Or64(Op1 As TBit64, Op2 As TBit64) As TBit64
' Or64.i0 = Op1.i0 Or Op2.i0
' Or64.i1 = Op1.i1 Or Op2.i1
' Or64.i2 = Op1.i2 Or Op2.i2
' Or64.i3 = Op1.i3 Or Op2.i3
'End Function
'
'Public Function Xor64(Op1 As TBit64, Op2 As TBit64) As TBit64
' Xor64.i0 = Op1.i0 Xor Op2.i0
' Xor64.i1 = Op1.i1 Xor Op2.i1
' Xor64.i2 = Op1.i2 Xor Op2.i2
' Xor64.i3 = Op1.i3 Xor Op2.i3
'End Function
'
'Public Sub Clear64(Op1 As TBit64)
' Op1.i0 = 0
' Op1.i1 = 0
' Op1.i2 = 0
' Op1.i3 = 0
'End Sub
'
'Public Function PopCnt64(Op1 As TBit64) As Long
' PopCnt64 = PopCount(Op1.i0) + PopCount(Op1.i1) + PopCount(Op1.i2) + PopCount(Op1.i3)
'End Function
'
Public Sub SetMove(m1 As TMOVE, m2 As TMOVE)
 ' assign m2 to m1. 3x faster than Move1 = Move 2  !
 With m1
  .Captured = m2.Captured: .CapturedNumber = m2.CapturedNumber: .Castle = m2.Castle: .
  EnPassant = m2.EnPassant
  .From = m2.From: .IsChecking = m2.IsChecking: .IsLegal = m2.IsLegal: .OrderValue =
  m2.OrderValue: .Piece = m2.Piece
  .Promoted = m2.Promoted: .SeeValue = m2.SeeValue: .Target = m2.Target
 End With
End Sub

Public Sub SwapMove(m1 As TMOVE, m2 As TMOVE)
 Dim l As Long, b As Boolean
 With m2
  l = .Captured: .Captured = m1.Captured: m1.Captured = l
  l = .CapturedNumber: .CapturedNumber = m1.CapturedNumber: m1.CapturedNumber = l
  l = .Castle: .Castle = m1.Castle: m1.Castle = l
  l = .EnPassant: .EnPassant = m1.EnPassant: m1.EnPassant = l
  l = .From: .From = m1.From: m1.From = l
  b = .IsChecking: .IsChecking = m1.IsChecking: m1.IsChecking = b
  b = .IsLegal: .IsLegal = m1.IsLegal: m1.IsLegal = b
  l = .OrderValue: .OrderValue = m1.OrderValue: m1.OrderValue = l
  l = .Piece: .Piece = m1.Piece: m1.Piece = l
  l = .Promoted: .Promoted = m1.Promoted: m1.Promoted = l
  l = .SeeValue: .SeeValue = m1.SeeValue: m1.SeeValue = l
  l = .Target: .Target = m1.Target: m1.Target = l
 End With
End Sub

Public Sub ClearMove(m1 As TMOVE)
  ' 2x faster than Move1 = EmptyMove  !
  With m1
    .From = 0: .Target = 0: .Piece = NO_PIECE: .Castle = NO_CASTLE: .Promoted = 0: .
    Captured = NO_PIECE: .CapturedNumber = 0
    .EnPassant = 0: .IsChecking = False: .IsLegal = False: .OrderValue = 0: .SeeValue
    = VALUE_NONE
  End With
End Sub

'Public Function WCastlingRight() As Long
'   If Moved(WKingLoc) = 0 Then
'     If Moved(SQ_H1) = 0 Then WCastlingRight = 1
'     If Moved(SQ_A1) = 0 Then WCastlingRight = WCastlingRight Or 2
'   Else
'     WCastlingRight = 0
'   End If
'End Function

'Public Function BCastlingRight() As Long
'   If Moved(BKingLoc) = 0 Then
'     If Moved(SQ_H8) = 0 Then BCastlingRight = 1
'     If Moved(SQ_A8) = 0 Then BCastlingRight = BCastlingRight Or 2
'   Else
'     BCastlingRight = 0
'   End If
'End Function


Attribute VB_Name = "basBook"
'==================================================
'= basBook:
'= chess opening book functions
'==================================================
Option Explicit
Public bUseBook       As Boolean

Public UCIBook() As String
Public UCIBookMax As Long, UCIBookCnt As Long
Public BookMovePossible As Boolean

'---------------------------------------------------------------------------
'ChooseBookMove()
'---------------------------------------------------------------------------
Public Function ChooseBookMove() As TMOVE
  ' game has to be started from startup position, FEN/EPD loaded position not supported
  Dim i                As Long
  Dim sPossibleMove    As String, sCoordMove As String
  Dim iNumMoves        As Long

  SetMove ChooseBookMove, EmptyMove

  sPossibleMove = GetUCIBookMove()

  ' check for legal move
  Ply = 1
  GenerateMoves Ply, False, iNumMoves

  For i = 0 To iNumMoves - 1
    sCoordMove = CompToCoord(Moves(Ply, i)) ' format "e4d5"
    If sCoordMove = sPossibleMove Then
      SetMove ChooseBookMove, Moves(Ply, i)
      Exit Function
    End If
  Next

End Function


'---------------------------------------------------------------------------
```

```
'InitBook()
'-----------------------------------------------------------------------
Public Function InitBook() As Boolean
 Static bInitBookDone As Boolean
 Static bUseBookOk As Boolean
 Dim sBookFile As String

 If bInitBookDone Then ' read only once
  InitBook = bUseBookOk
  Exit Function
 End If

 If pbMSExcelRunning Then ' set in SetVBAPathes
   InitBook = ReadExcelBook()
 End If
 If Not InitBook Then
   sBookFile = ReadINISetting(USE_BOOK_KEY, "CB_BOOK.TXT")
   If pbIsOfficeMode And Trim(sBookFile) = "" Then
     ' Always use default book if not set in INI file
     sBookFile = "CB_BOOK.TXT"
   End If
   InitBook = ReadUCIBook(sBookFile)
 End If
 bUseBookOk = InitBook
 bInitBookDone = True
End Function

'-----------------------------------------------------------------------
' MS Excel: read book from internal worksheet
'-----------------------------------------------------------------------
Public Function ReadExcelBook() As Boolean
  On Error GoTo lblError

   #If VBA_MODE = 1 Then
      ' read opening book lines from Excel worksheet CB_BOOK
      Dim Sheet As Object, lNum As Long, i As Long, sInp As String

      Set Sheet = ActiveWorkbook.Sheets("CB_BOOK")

      ReDim UCIBook(0)
      UCIBookMax = 0: UCIBookCnt = 0

      With Sheet
        lNum = .Cells(.Rows.Count, 1).End(xlUp).Row
        For i = 1 To lNum
          sInp = Trim$(.Cells(i, 1))
          If Left(sInp, 1) <> "#" And sInp <> "" Then '# : comment line
            UCIBookCnt = UCIBookCnt + 1: If UCIBookCnt > UCIBookMax Then UCIBookMax =
            UCIBookMax + 1000: ReDim Preserve UCIBook(UCIBookMax)
            UCIBook(UCIBookCnt) = sInp
          End If
        Next i
      End With ' sheet

      ReadExcelBook = (UCIBookCnt > 0)
      If ReadExcelBook Then
        SendCommand "opening book found in Excel sheet CB_BOOK. Lines found:" &
        UCIBookCnt
      End If
      Exit Function
   #End If
lblError:
      ReadExcelBook = False
End Function


Public Function GetUCIGameLine() As String
  Dim i As Long, h As Long, s As String, MoveCnt As Long, Cnt As Long

  GetUCIGameLine = ""

  Cnt = GameMovesCnt
  If Cnt = 0 Then Exit Function
  s = "": MoveCnt = 0

  For i = 1 To Cnt Step 2
    MoveCnt = MoveCnt + 1
    s = s & CompToCoord(arGameMoves(i))
    If i + 1 <= Cnt Then s = s & " " & CompToCoord(arGameMoves(i + 1)) & " "
  Next i
  GetUCIGameLine = Trim$(s)

End Function

Public Function ReadUCIBook(isFile As String) As Boolean
 ' Read PGN File
  Dim h As Long, sInp As String, sBookFile As String

  ReadUCIBook = False

  h = 10 'FreeFile()
  ReDim UCIBook(0)
  UCIBookMax = 0: UCIBookCnt = 0

  sBookFile = psEnginePath & "\" & isFile

  On Error GoTo lblError
  If Dir(sBookFile) = "" Or isFile = "" Then
    Dim sDefault As String
    If pbIsOfficeMode Then sDefault = "1" Else sDefault = "0"

    If ReadINISetting("USE_INTERNAL_BOOK", sDefault) = "1" Then
      InitInternalBook
      ReadUCIBook = True
      If pbIsOfficeMode Then
        SendCommand "internal opening book active"
      ElseIf UCIMode Then
        SendCommand "info string internal opening book active"
      End If
    End If
    Exit Function
  End If

  Open sBookFile For Input As #h

  Do Until EOF(h)
    Line Input #h, sInp: sInp = Trim(sInp)
    If Left(sInp, 1) <> "#" And sInp <> "" Then '# : comment line
      UCIBookCnt = UCIBookCnt + 1: If UCIBookCnt > UCIBookMax Then UCIBookMax =
      UCIBookMax + 1000: ReDim Preserve UCIBook(UCIBookMax)
      UCIBook(UCIBookCnt) = sInp
    End If
  Loop
  ReadUCIBook = (UCIBookCnt > 0)
  If ReadUCIBook Then
      If pbIsOfficeMode Then
        SendCommand "opening book found: " & isFile
      ElseIf UCIMode Then
        SendCommand "info string opening book found: " & isFile
      End If
  End If

  Close #h
  Exit Function
lblError:
  ReadUCIBook = False
```

```
End Function

Public Function GetUCIBookMove() As String
'--- input file ist sorted, lowercase, UCi format e4d5 (not e4xd5 or Bxd5)
' ---- create book file from PGN
' pgn-extract.exe -Wuci --notags --noresults -C -N -V --output book.txt test.pgn
' sort out.txt /o book.txt

Dim sUCIGame As String, sBookLine As String, r As Double
Dim i As Long, lStart As Long, lEnd As Long, lUCILen As Long, x As Long

GetUCIBookMove = ""
sUCIGame = GetUCIGameLine()

lUCILen = Len(sUCIGame)

If lUCILen >= 4 Then
  lStart = 0
  For i = 1 To UCIBookCnt
   If Left$(UCIBook(i), lUCILen) = sUCIGame Then
     lEnd = i: If lStart = 0 Then lStart = i
   End If
  Next
Else
  ' first game move
  lStart = 1: lEnd = UCIBookCnt
End If

' get a random move in the range found
Randomize
r = Rnd
If lEnd > lStart Then lStart = lStart + Int(((lEnd - lStart + 1) * r))
sBookLine = Trim$(Mid$(UCIBook(lStart), lUCILen + 1))
If Len(sBookLine) >= 4 Then
  sBookLine = Trim$(Left$(sBookLine, 4)) ' no promotion moves supported
  If Len(sBookLine) = 4 Then GetUCIBookMove = sBookLine
End If

'Debug.Print lStart, lEnd; r, GetUCIBookMove
End Function

Public Function InitInternalBook()
  ' Read internal book, just for fun - if external book is missing
  Dim BookArr As Variant 'extra array because Variant type needed for ARRAY()
  Dim i As Long
  BookArr = Array("a2a3 g7g6 g2g3 f8g7 f1g2", "a2a3 g8f6 g1f3 d7d5 d2d4", "b1c3 c7c5
  d2d4 c5d4 d1d4", "b1c3 c7c5 e2e3 g7g6 d2d4", "b1c3 d7d5 e2e4 d5d4 c3e2", _
                        "b2b3 d7d5 c1b2 c8g4 g2g3", "b2b3 e7e5 c1b2 b8c6 e2e3", "c2c4 b8c6
                        g2g3 e7e5 f1g2", "c2c4 c7c5 b1c3 b7b6 e2e3", "c2c4 c7c5 b1c3 b7b6
                        e2e4", _
                        "c2c4 e7e5 b1c3 b8c6 g2g3", "c2c4 e7e6 g1f3 g8f6 b2b3", "c2c4 e7e6
                        g2g3 g8f6 f1g2", "c2c4 f7f5 b1c3 g8f6 d2d3", "c2c4 f7f5 b1c3 g8f6
                        d2d4", _
                        "c2c4 g8f6 b1c3 e7e6 g1f3", "d2d4 d7d5 c2c4 c7c6 b1c3", "d2d4 d7d5
                        g1f3 g8f6 c2c4", "d2d4 d7d5 g1f3 g8f6 e2e3", "d2d4 d7d6 c1g5 b8d7
                        e2e4", _
                        "d2d4 d7d6 c1g5 f7f6 g5h4", "d2d4 d7d6 c1g5 g7g6 c2c4", "d2d4 d7d6
                        c2c3 g8f6 c1g5", "d2d4 d7d6 c2c4 e7e5 b1c3", "d2d4 d7d6 c2c4 f7f5
                        g2g3", _
                        "d2d4 d7d6 c2c4 g7g6 b1c3", "d2d4 d7d6 e2e4 c7c5 d4d5", "d2d4 d7d6
                        e2e4 e7e5 g1f3", "d2d4 d7d6 e2e4 g7g6 b1c3", "d2d4 d7d6 e2e4 g7g6
                        c2c4", _
                        "d2d4 d7d6 e2e4 g8f6 b1c3", "d2d4 g8f6 c2c4 e7e6 g1f3", "d2d4 g8f6
                        g1f3 g7g6 g2g3", "e2e4 b7b6 g2g3 c8b7 f1g2", "e2e4 b8c6 b1c3 e7e5
                        f1c4", _
                        "e2e4 b8c6 d2d4 e7e5 g1f3", "e2e4 b8c6 f1b5 g8f6 d2d3", "e2e4 b8c6
                        g1f3 d7d6 d2d4", "e2e4 c7c5 b1c3 a7a6 g2g4", "e2e4 c7c5 b1c3 b8c6
                        d2d3", _
```

```
                    "e2e4 c7c5 c2c3 e7e6 d2d4", "e2e4 c7c5 f2f4 d7d5 d2d3", "e2e4 c7c5
                    f2f4 d7d5 e4d5", "e2e4 c7c5 g1f3 a7a6 b1c3", "e2e4 c7c5 g1f3 d8c7
                    d2d4", _
                    "e2e4 c7c5 g1f3 e7e6 b1c3", "e2e4 c7c6 g1f3 d7d5 e4d5", "e2e4 d7d6
                    d2d4 g8f6 b1c3", "f2f4 b7b6 g1f3 c8b7 e2e3", "f2f4 c7c5 b2b3 g8f6
                    c1b2", _
                    "f2f4 d7d5 e2e3 g8f6 g1f3", "g1f3 c7c5 c2c3 g8f6 g2g3", "g1f3 c7c5
                    c2c4 b7b6 b1c3", "g1f3 d7d5 c2c4 c7c6 g2g3", "g1f3 d7d5 c2c4 d5c4
                    b1a3", _
                    "g2g3 d7d5 f1g2 c7c6 g1f3", "g2g3 d7d5 f1g2 e7e5 c2c3", "g2g3 e7e5
                    f1g2 d7d5 d2d3", "g2g3 g7g6 f1g2 f8g7 c2c4", "g2g3 g8f6 f1g2 e7e5
                    d2d3")
  UCIBookCnt = UBound(BookArr) + 1
  ReDim UCIBook(UCIBookCnt)
  For i = 1 To UCIBookCnt: UCIBook(i) = BookArr(i - 1): Next

End Function




Attribute VB_Name = "basChessBrainVB"
'==================================================
'= basChessBrainVB:
'= main program
'=
'= ChessBrainVB V4.00:
'=   by Roger Zuehlsdorf (Copyright 2023)
'=   based on LarsenVB by Luca Dormio (http://xoomer.virgilio.it/ludormio/download.htm) and Faile by Adrien M.
Regimbald
'=       and Stockfish by Marco Costalba, Joona Kiiski, Gary Linscott, Tord Romstad
'= start of program
'= init engine
'==================================================
Option Explicit
'DEBUGMODE: console input via VB form.   Else: Winbord interface
Public DebugMode                    As Boolean
'simulate standard input
'set in frmDebugMain.cmdFakeInput_Click
Public FakeInputState               As Boolean
Public FakeInput                    As String
Public MatchInfo                    As TMatchInfo
Public bXBoardMode                  As Boolean
Public iXBoardProtoVer              As Long        ' winboard protocol version
Public bForceMode                   As Boolean
Public bPostMode                    As Boolean
Public bAnalyzeMode                 As Boolean
Public bExitReceived                As Boolean
Public bAllowPonder                 As Boolean
Public ThisApp                      As Object
Public psAppName                    As String
Public Moves(MAX_DEPTH, MAX_MOVES)        As TMOVE ' Generated moves [ply,Move]
Public QuietsSearched(MAX_DEPTH, 65)      As TMOVE  ' Quiet moves for pruning conditions
Public CapturesSearched(MAX_DEPTH, 32)    As TMOVE  ' Quiet moves for pruning conditions
Public MovePickerDat(MAX_DEPTH)           As TMovePicker
Public GameMovesCnt                 As Long
Public arGameMoves(MAX_GAME_MOVES)  As TMOVE
Public GamePosHash(MAX_GAME_MOVES)  As THashKey
Public GUICheckIntervalNodes        As Long
Public MemoryMB                     As Long ' memory command
Public UCIMode                      As Boolean
Public pbMSExcelRunning             As Boolean


'-------------------------------------
' Main:  Start of program ChessBrainVB -
'-------------------------------------
Sub Main()
```

```
Dim sCmdList() As String
Dim i          As Long

'--- VBA_MODE constant is set in Excel/Word in VBAChessBrain project properties for conditional compiling
#If VBA_MODE = 1 Then
  '--- MS-OFFICE VBA ---
  pbIsOfficeMode = True
  GUICheckIntervalNodes = 1000  ' nodes until next check for GUI commands
  SetVBAPathes
#Else
  '--- VB6 ---
  pbIsOfficeMode = False
  pbMSExcelRunning = False
  GUICheckIntervalNodes = 5000
  psEnginePath = App.Path
  psAppName = App.EXEName
#End If
DebugMode = CBool(ReadINISetting("DEBUGMODE", "0") <> "0")
bWinboardTrace = CBool(ReadINISetting("COMMANDTRACE", "0") <> "0")
bThreadTrace = CBool(ReadINISetting("THREADTRACE", "0") <> "0")
bTimeTrace = CBool(ReadINISetting("TIMETRACE", "0") <> "0")
bEGTbBaseTrace = CBool(ReadINISetting("TBBASE_TRACE", "0") <> "0")
bWbPvInUciFormat = CBool(ReadINISetting("WB_PV_IN_UCI", "0") <> "0")
InitTranslate
' set main threadnum=-1
SetThreads 1
'
'--- command line options
'
If Command$ <> "" Then
  sCmdList = Split(LCase(Command$))

  For i = 0 To UBound(sCmdList)
    If bWinboardTrace Then WriteTrace "Command: " & sCmdList(i) & " " & Now()
    If Left$(Trim$(sCmdList(i)), 6) = "thread" Then
      #If VBA_MODE = 0 Then
        ' Parameter for helper threads : "threat1" .. "threat8"
        ThreadNum = Val("0" & Trim$(Mid$((Trim$(sCmdList(i))), 7)))
        ThreadNum = GetMax(1, ThreadNum): NoOfThreads = ThreadNum + 1
        If bThreadTrace Then WriteTrace "Command: ThreadNum = " & ThreadNum & " / " & Now()
        App.Title = "ChessBrainVB_T" & Trim$(CStr(ThreadNum))
      #End If
    Else

      Select Case Trim$(sCmdList(i))
        Case "xboard", "/xboard", "-xboard"
          bXBoardMode = True
        Case "log", "/log", "-log"
          bLogMode = True
          bLogPV = CBool(Val(ReadINISetting(LOG_PV_KEY, "0")))
        Case "/?", "-?", "?"
          MsgBox "arguments:  -xboard ,  -log"
        Case ""
        Case Else
          MsgBox "Wrong argument " & vbLf & Command$, vbExclamation
      End Select

    End If
  Next
End If

If ThreadNum <= 0 Then
  OpenCommHandles ' enable GUI communication > main thread
  SendCommand "ChessBrainVB by Roger Zuehlsdorf"
End If

#If VBA_MODE <> 0 Then
```

```
      InitEngine
      frmChessX.Show
      Exit Sub
    #End If
    #If DEBUG_MODE <> 0 Then
      ' Simulate Xboard using input of debug form
      bXBoardMode = True
      InitEngine
      If ThreadNum <= 0 Then
        frmDebugMain.Show   ' --- Show debug form
      End If

      '-------------------------------------------------
      MainLoop   '--- Wait for winboard commands from debug form
      '-------------------------------------------------

      Exit Sub
    #End If
    #If DEBUG_MODE = 0 And VBA_MODE = 0 Then
      If Not bXBoardMode And Trim(ReadINISetting("WINBOARD", "")) = "" Then
        bXBoardMode = CBool(Trim(ReadINISetting("XBOARD_MODE", "1")) = "1")
      End If
      If bXBoardMode Then
        '------------------------------------------
        '---  normal winboard/uci mode without form
        '------------------------------------------
        InitEngine
        '------------>>> loop for new external commands <<<-------------------
        MainLoop    '--- Wait for winboard/ uci commands
        '------------<<< loop for new external commands <<<-------------------
        '<<<
      Else
        ' init winboard path
        frmMain.Show   '--- Show main form
      End If
    #End If
  End Sub

  '---------------------------------------------------------------------------
  ' InitEngine()
  '---------------------------------------------------------------------------
  Public Sub InitEngine()
    iXBoardProtoVer = 1
    '-----------------------------
    '--- init arrays
    '-----------------------------
    Erase PVLength()
    Erase PV()
    Erase History()
    Erase CaptureHistory()
    Erase CounterMove()
    Erase ContinuationHistory()
    'InitContHist  ' if filled with specific start values

    Erase Pieces()
    Erase Squares()
    Erase Killer()
    Erase Board()
    Erase Moved()
    Erase MovesList()
    Erase arGameMoves()
    Erase GamePosHash()

    InitPieceColor

    '-----------------------------------
    '--- move offsets  ---
    '-----------------------------------
```

```
' 0-3: Orthogonal (Queen+Rook), 4-7=diagonal (Queen+Bishop)
ReadIntArr DirectionOffset(), 10, -10, 1, -1, 11, -11, 9, -9
ReadIntArr KnightOffsets(), 8, 19, 21, 12, -8, -19, -21, -12
ReadIntArr BishopOffsets(), 9, 11, -9, -11
ReadIntArr RookOffsets(), 1, -1, 10, -10
OppositeDir(1) = -1: OppositeDir(-1) = 1: OppositeDir(10) = -10: OppositeDir(-10) = 10
OppositeDir(11) = -11: OppositeDir(-11) = 11: OppositeDir(9) = -9: OppositeDir(-9) = 9

ReadIntArr WPromotions(), 0, WQUEEN, WROOK, WKNIGHT, WBISHOP
ReadIntArr BPromotions(), 0, BQUEEN, BROOK, BKNIGHT, BBISHOP
ReadIntArr PieceType, 0, PT_PAWN, PT_PAWN, PT_KNIGHT, PT_KNIGHT, PT_BISHOP, PT_BISHOP, PT_ROOK, PT_ROOK, PT_QUEEN, PT_QUEEN, PT_KING, PT_KING, NO_PIECE_TYPE, PT_PAWN, PT_PAWN
InitRankFile ' must be before InitMaxDistance
InitBoardColors
InitMaxDistance
InitSqBetween
InitSameXRay
InitAttackBitCnt
bAllowPonder = False

' setup empty move
With EmptyMove
  .From = 0: .Target = 0: .Piece = NO_PIECE: .Castle = NO_CASTLE: .Promoted = 0: .Captured = NO_PIECE: .CapturedNumber = 0
  .EnPassant = 0: .IsChecking = False: .IsLegal = False: .OrderValue = 0: .SeeValue = VALUE_NONE
End With

'--------------------------------------------
'--- startup board
'--------------------------------------------
ReadIntArr StartupBoard(), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, WROOK, WKNIGHT, WBISHOP, WQUEEN, WKING, WBISHOP, WKNIGHT, WROOK, 0, 0, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, WPAWN, 0, 0, 13, 13, 13, 13, 13, 13, 13, 13, 0, 0, 13, 13, 13, 13, 13, 13, 13, 13, 0, 0, 13, 13, 13, 13, 13, 13, 13, 13, 0, 0, 13, 13, 13, 13, 13, 13, 13, 13, 0, 0, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, BPAWN, 0, 0, BROOK, BKNIGHT, BBISHOP, BQUEEN, BKING, BBISHOP, BKNIGHT, BROOK, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

'----------------------------------------------------------
'--- Piece square table: bonus for piece position on board ---
'----------------------------------------------------------
' ( FILE A-D: Pairs MG,EG :  A(MG,EG),B(MG,EG),...
'--- Pawn piece square table
PSQT64 PsqtWP, PsqtBP, 0, 0, 0, 0, 0, 0, 0, 0, -11, 7, 6, -4, 7, 8, 3, -2, -18, -4, -2, -5, 19, 5, 24, 4, -17, 3, -9, 3, 20, -8, 35, -3, -6, 8, 5, 9, 3, 7, 21, -6, -6, 8, -8, -5, -6, 2, -2, 4, -4, 3, 20, -9, -8, 1, -4, 18, 0, 0, 0, 0, 0, 0, 0, 0
'--- Knight piece square table
PSQT64 PsqtWN, PsqtBN, -161, -105, -96, -82, -80, -46, -73, -14, -83, -69, -43, -54, -21, -17, -10, 9, -71, -50, -22, -39, 0, -7, 9, 28, -25, -41, 18, -25, 43, 6, 47, 38, -26, -46, 16, -25, 38, 3, 50, 40, -11, -54, 37, -38, 56, -7, 65, 27, -63, -65, -19, -50, 5, -24, 14, 13, -195, -109, -67, -89, -42, -50, -29, -13
'--- Bishop piece square table
PSQT64 PsqtWB, PsqtBB, -44, -58, -13, -31, -25, -37, -34, -19, -20, -34, 20, -9, 12, -14, 1, 4, -9, -23, 27, 0, 21, -3, 11, 16, -11, -26, 28, -3, 21, -5, 10, 16, -11, -26, 27, -4, 16, -7, 9, 14, -17, -24, 16, -2, 12, 0, 2, 13, -23, -34, 17, -10, 6, -12, -2, 6, -35, -55, -11, -32, -19, -36, -29, -17
'--- Rook piece square table
PSQT64 PsqtWR, PsqtBR, -25, 0, -16, 0, -16, 0, -9, 0, -21, 0, -8, 0, -3, 0, 0, 0, -21, 0, -9, 0, -4, 0, 2, 0, -22, 0, -6, 0, -1, 0, 2, 0, -22, 0, -7, 0, 0, 0, 1, 0, -21, 0, -7, 0, 0, 0, 2, 0, -12, 0, 4, 0, 8, 0, 12, 0, -23, 0, -15, 0, -11, 0, -5, 0
'--- Queen piece square table
PSQT64 PsqtWQ, PsqtBQ, 0, -71, -4, -56, -3, -42, -1, -29, -4, -56, 6, -30, 9, -21, 8, -5, -2, -39, 6, -17, 9, -8, 9, 5, -1, -29, 8, -5, 10, 9, 7, 19, -3, -27, 9, -5, 8, 10, 7, 21, -2, -40, 6, -16, 8, -10, 10, 3, -2, -55, 7, -30, 7, -21, 6, -6, -1, -74,
 -4, -55, -1, -43, 0, -30
'--- King piece square table
PSQT64 PsqtWK, PsqtBK, 267, 0, 320, 48, 270, 75, 195, 84, 264, 43, 304, 92, 238, 143
, 180, 132, 200, 83, 245, 138, 176, 167, 110, 165, 177, 106, 185, 169, 148, 169, 110
, 179, 149, 108, 177, 163, 115, 200, 66, 203, 118, 95, 159, 155, 84, 176, 41, 174,
87, 50, 128, 99, 63, 122, 20, 139, 63, 9, 88, 55, 47, 80, 0, 90
FillPieceSquareVal
'---  Mobility bonus for number of attacked squares not occupied by friendly pieces (pairs: MG,EG, MG,EG)
' Knights
ReadScoreArr MobilityN, -75, -76, -56, -54, -9, -26, -2, -10, 6, 5, 15, 11, 22, 26,
30, 28, 36, 29
' Bishops
ReadScoreArr MobilityB, -48, -58, -21, -19, 16, -2, 26, 12, 37, 22, 51, 42, 54, 54,
63, 58, 65, 63, 71, 70, 79, 74, 81, 86, 92, 90, 97, 94
' Rooks
ReadScoreArr MobilityR, -56, -78, -25, -18, -11, 26, -5, 55, -4, 70, -1, 81, 8, 109,
 14, 120, 21, 128, 23, 143, 31, 154, 32, 160, 43, 165, 49, 168, 59, 169
' Queens
ReadScoreArr MobilityQ, -40, -35, -25, -12, 2, 7, 4, 19, 14, 37, 24, 55, 25, 62, 40,
 76, 43, 79, 47, 87, 54, 94, 56, 102, 60, 111, 70, 116, 72, 118, 73, 122, 75, 128,
77, 130, 85, 133, 94, 136, 99, 140, 108, 157, 112, 158, 113, 161, 118, 174, 119, 177
, 123, 191, 128, 199
'SF6: Threat by pawn (pairs MG/EG: NOPIECE,PAWN,KNIGHT (176,139), BISHOP, ROOK, QUEEN
'SF6: Outpost (Pair MG/EG )[0, 1=supported by pawn]
ReadScoreArr ReachableOutpostKnight, 22, 6, 36, 12
ReadScoreArr ReachableOutpostBishop, 9, 2, 15, 5
ReadScoreArr OutpostBonusKnight, 44, 12, 66, 18
ReadScoreArr OutpostBonusBishop, 18, 4, 28, 8
'SF6: King Attack Weights by attacker { 0, 0, 7, 5, 4, 1 }  NO_PIECE_TYPE, PAWN, KNIGHT, BISHOP, ROOK,
QUEEN, KING,
' SF values not clear: why queen is 1 and knight is 7 ?!? More attack fields in total for queen?
KingAttackWeights(PT_PAWN) = 5: KingAttackWeights(PT_KNIGHT) = 80: KingAttackWeights
(PT_BISHOP) = 56: KingAttackWeights(PT_ROOK) = 45: KingAttackWeights(PT_QUEEN) = 12
' Pawn eval
' Isolated pawn penalty by opposed flag
ReadScoreArr IsolatedPenalty(), 27, 30, 13, 18
ReadScoreArr BackwardPenalty(), 40, 26, 24, 12 ' not opposed /  opposed
SetScoreVal DoubledPenalty, 18, 38
ReadScoreArr LeverBonus(), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 17, 16, 33, 32, 0, 0, 0, 0
ReadIntArr PassedDanger(), 0, 0, 0, 0, 3, 6, 12, 21
ReadScoreArr PassedPawnRankBonus(), 0, 0, 0, 0, 7, 10, -12, 26, 3, 31, 42, 63, 178,
167, 279, 244
ReadScoreArr PassedPawnFileBonus(), 0, 0, 17, 3, 0, 10, 1, -23, -16, -20, _
                                       -17, -8, 3, -1, -8, 4, 17, 9
ReadScoreArr KingProtector(), 0, 0, 0, 0, -3, -5, -4, -3, -3, 0, -1, 1 ' for N,B,R,Q
ReadIntArr QueenMinorsImbalance(), 31, -8, -15, -25, -5
ReadIntArr CaptPruneMargin(), 0, -238, -262, -244, -252, -241, -228
' King safety eval
' Weakness of our pawn shelter in front of the king by [distance from edge][rank]
ReadIntArr2 ShelterWeakness(), 1, 0, 100, 10, 46, 82, 87, 86, 98 ' 1 = ArrIndex, 0: fill
Array(0)
ReadIntArr2 ShelterWeakness(), 2, 0, 116, 4, 28, 87, 94, 108, 104
ReadIntArr2 ShelterWeakness(), 3, 0, 109, 1, 59, 87, 62, 91, 116
ReadIntArr2 ShelterWeakness(), 4, 0, 75, 12, 43, 59, 90, 84, 112
' Danger of enemy pawns moving toward our king by [type][distance from edge][rank]
' BlockedByKing
ReadIntArr3 StormDanger(), 1, 1, 0, 0, -290, -274, 57, 41
ReadIntArr3 StormDanger(), 1, 2, 0, 0, 60, 144, 39, 13
ReadIntArr3 StormDanger(), 1, 3, 0, 0, 65, 141, 41, 34
ReadIntArr3 StormDanger(), 1, 4, 0, 0, 53, 127, 56, 14
' Unopposed
ReadIntArr3 StormDanger(), 2, 1, 0, 4, 73, 132, 46, 31
ReadIntArr3 StormDanger(), 2, 2, 0, 1, 64, 143, 26, 13
ReadIntArr3 StormDanger(), 2, 3, 0, 1, 47, 110, 44, 24
ReadIntArr3 StormDanger(), 2, 4, 0, 0, 72, 127, 50, 31
' BlockedByPawn
ReadIntArr3 StormDanger(), 3, 1, 0, 0, 0, 79, 23, 1
ReadIntArr3 StormDanger(), 3, 2, 0, 0, 0, 148, 27, 2
```

```
    ReadIntArr3 StormDanger(), 3, 3, 0, 0, 0, 161, 16, 1
    ReadIntArr3 StormDanger(), 3, 4, 0, 0, 0, 171, 22, 15
    ' Unblocked
    ReadIntArr3 StormDanger(), 4, 1, 0, 22, 45, 104, 62, 6
    ReadIntArr3 StormDanger(), 4, 2, 0, 31, 30, 99, 39, 19
    ReadIntArr3 StormDanger(), 4, 3, 0, 23, 29, 96, 41, 15
    ReadIntArr3 StormDanger(), 4, 4, 0, 21, 23, 116, 41, 15
    '--- Endgame helper tables: Tables used to drive a piece towards or away from another piece
    ReadIntArr PushClose(), 0, 0, 100, 80, 60, 40, 20, 10
    ReadIntArr PushAway(), 0, 5, 20, 40, 60, 80, 90, 100
    ReadIntArr PushToEdges(), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    , 0, 100, 90, 80, 70, 70, 80, 90, 100, 0, 0, 90, 70, 60, 50, 50, 60, 70, 90, 0, 0,
    80, 60, 40, 30, 30, 40, 60, 80, 0, 0, 70, 50, 30, 20, 20, 30, 50, 70, 0, 0, 70, 50,
    30, 20, 20, 30, 50, 70, 0, 0, 80, 60, 40, 30, 30, 40, 60, 80, 0, 0, 90, 70, 60, 50,
    50, 60, 70, 90, 0, 0, 100, 90, 80, 70, 70, 80, 90, 100
    ReadIntArr KRPPKRP_SFactor(), 0, 0, 9, 10, 14, 21, 44, 0, 0
    ' Threats
    ReadScoreArr ThreatByMinor, 0, 0, 0, 33, 45, 43, 46, 47, 72, 107, 48, 118 'Minor on
    Defended
    ReadScoreArr ThreatByRook, 0, 0, 0, 25, 40, 62, 40, 59, 0, 34, 35, 48 'Major on Defended
    ReadScoreArr ThreatBySafePawn, 0, 0, 0, 0, 176, 139, 141, 127, 217, 218, 203, 215
    SetScoreVal ThreatByRank, 16, 3
    SetScoreVal ThreatenedByHangingPawn, 71, 61
    SetScoreVal KingOnOneBonus, 3, 62
    SetScoreVal KingOnManyBonus, 9, 138
    SetScoreVal Hanging, 48, 27 ' Hanging piece penalty
    SetScoreVal Overload, 10, 5 ' attacked opp pieces defended onyl once
    SetScoreVal WeakUnopposedPawn, 5, 25 ' weak pawn when opp has Q/R
    SetScoreVal SafeCheck, 20, 20
    SetScoreVal OtherCheck, 10, 10
    SetScoreVal PawnlessFlank, 20, 80
    SetScoreVal ThreatByAttackOnQueen, 43, 19
    ' Thread Skip values for depth/move
    ReadIntArr SkipSize, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4
    ReadIntArr SkipPhase, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 6, 7
    'Material Imbalance
    InitImbalance
    ' Init EPD table
    InitEPDTable
    bUseBook = InitBook
    ' Init Hash
    InitZobrist
    ' Endgame tablebase access (via online web service or fathom.exe)
    InitTableBases
    ' Init game
    InitGame
  End Sub

  '---------------------------------------------------------------------------
  ' MainLoop() - main program loop
  '
  ' contains two functions
  ' ParseCommand:  parse for new input from winboard: setup board,time control, ...
  '
  ' StartEngine:  if computer to move:  execute commands (calculate moves)
  '---------------------------------------------------------------------------
  Public Sub MainLoop()
    Dim sInput As String
    ThreadCommand = ""

    Do
      StartEngine ' returns with no action if computer not to move
      If PollCommand Then   ' Something new ?
        sInput = ReadCommand ' Get it
        If sInput <> "" Then ParseCommand sInput ' Examine it
      Else
        If Not DebugMode Then
          Sleep 10 ' do not use more CPU than needed when waiting
        End If
      End If
      DoEvents
      If ThreadNum > 0 Then CheckThreadTermination True
    Loop

  End Sub

  '-------------------------------------------------------------------------
  ' ParseCommand() - parse winboard input
  '
  ' a command list like "xboard\nnew\nrandom\nlevel 40 5 0\nhard" is splitted
  '-------------------------------------------------------------------------
  Public Sub ParseCommand(ByVal sCommand As String)
    Dim bLegalInput As Boolean
    Dim i           As Long, c As Long, x As Long, s As String, sSearch As String
    Dim PlayerMove  As TMOVE, sCoordMove As String
    Dim iNumMoves   As Long
    Dim sCurrentCmd As String
    Dim sCmdList()  As String
    Dim sInput()    As String
    Dim Hashkey     As THashKey
    If Trim$(sCommand) = "" Then Exit Sub
    sCommand = Replace(sCommand, vbCr, vbLf) 'Fix per DDInterfaceEngine:
    If Right$(sCommand, 1) <> vbLf Then sCommand = sCommand & vbLf
    sCmdList = Split(sCommand, vbLf)

    For c = 0 To UBound(sCmdList) - 1        'ignore vbLf
      sCurrentCmd = sCmdList(c)
      If sCurrentCmd = "" Then GoTo NextCmd
      If bWinboardTrace Then WriteTrace "Command: " & sCurrentCmd & " " & Now()
      If Trim$(sCurrentCmd) = "uci" Then
        '--- send UCI options
        UCIMode = True
        #If VBA_MODE = 1 Then
          SendCommand "id name ChessBrainVB" ' App object not defined
        #Else
          SendCommand "id name ChessBrainVB V" & Trim(App.Major) & "." & Trim(App.Minor)
           & Trim(App.Revision)
        #End If
        SendCommand ConvertID()
        SendCommand "option name Threads type spin default 1 min 1 max " & CStr(
        MAX_THREADS)
        SendCommand "option name Hash type spin default 128 min 1 max " & CStr(
        MAX_HASHSIZE_MB)
        SendCommand "option name Clear Hash type button"
       ' SendCommand "option name SyzygyPieceSet type spin default 5 min 0 max 6"
       ' SendCommand "option name SyzygyPath type string default <empty>"
       ' SendCommand "option name SyzygyMaxPly type spin default 3 min 1 max 6"
        SendCommand "uciok"
        UCISyzygyPath = ""
        UCISyzygyMaxPieceSet = -1
        UCISyzygyMaxPly = -1
        GoTo NextCmd
      End If
      If UCIMode Then
        '--- get UCI command
        sCurrentCmd = Trim$(sCurrentCmd)
        If sCurrentCmd = "ucinewgame" Or sCurrentCmd = "position startpos" Then
          If bWinboardTrace Then WriteTrace "UCI: " & sCurrentCmd & " " & Now()
          InitGame
          GoTo NextCmd
        ElseIf sCurrentCmd = "stop" Or sCurrentCmd = "ponderhit" Then
          bForceMode = False
          bTimeExit = True
          GoTo NextCmd
        ElseIf sCurrentCmd = "quit" Then
          ExitProgram
          End
        End If
        sSearch = "setoption name Hash value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          ' UCI hash memory size
          MemoryMB = Val("0" & Val(Mid$(sCurrentCmd, Len(sSearch) + 1)))
          If bWinboardTrace Then WriteTrace "UCI: hash memory size: " & sCurrentCmd & "
          " & Now()
          GoTo NextCmd
        End If
        sSearch = "setoption name Threads value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          ' number of threads/cores
         If Not pbIsOfficeMode Then
          If CBool(ReadINISetting("THREADS_IGNORE_GUI", "0") = "0") Then
            x = Val("0" & Val(Mid$(sCurrentCmd, Len(sSearch) + 1)))
            SetThreads x
            If bThreadTrace Then WriteTrace "Command:" & LCase(Command$)
          End If
         End If
          If bWinboardTrace Then WriteTrace "UCI: Threads: " & sCurrentCmd & " " & Now()
          GoTo NextCmd
        End If
        sSearch = "setoption name Contempt value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          ' contempt score in centi pawns for draw
          x = Val("0" & Val(Mid$(sCurrentCmd, Len(sSearch) + 1)))
          GoTo NextCmd
        End If
        sSearch = "setoption name Clear Hash"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          If NoOfThreads < 2 Then InitHash
          GoTo NextCmd
        End If
        sSearch = "setoption name SyzygyPieceSet value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          x = Val("0" & Val(Mid$(sCurrentCmd, Len(sSearch) + 1)))
          UCISyzygyMaxPieceSet = x
          If bEGTbBaseTrace Then WriteTrace "UCI SyzygyPieceSet= " & x
          GoTo NextCmd
        End If
        sSearch = "setoption name SyzygyPath value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          s = Trim$(Mid$(sCurrentCmd, Len(sSearch) + 1))
          If Right$(s, 1) = "\" Then s = Left$(s, Len(s) - 1) ' Remove right \
          UCISyzygyPath = s
          If bEGTbBaseTrace Then WriteTrace "UCI SyzygyPath= " & s
          InitTableBases
          If EGTBasesEnabled Then SendCommand "info string Tablebases found"
          GoTo NextCmd
        End If
        sSearch = "setoption name SyzygyMaxPly value"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          x = Val("0" & Val(Mid$(sCurrentCmd, Len(sSearch) + 1)))
          UCISyzygyMaxPly = x
          If bEGTbBaseTrace Then WriteTrace "UCI UCISyzygyMaxPly= " & x
          GoTo NextCmd
        End If
        sSearch = "setoption name Ponder"
        If Left$(sCurrentCmd, Len(sSearch)) = sSearch Then
          ' Ponder: ignore until implemented
          GoTo NextCmd
        End If
        If Left$(sCurrentCmd, Len("isready")) = "isready" Then
          'DoEvents
          If CBool(ReadINISetting("THREADS_IGNORE_GUI", "0") = "0") Then
            SendCommand "info string " & CStr(NoOfThreads) & IIf(NoOfThreads = 1, "
            core", " cores")
          Else
            SendCommand "info string " & CStr(NoOfThreads) & IIf(NoOfThreads = 1, " core", " cores (set in INI file)")
          End If
          If bWinboardTrace Then WriteTrace "UCI: " & sCurrentCmd & " " & Now()
          SendCommand "readyok"
          GoTo NextCmd
        End If
        If Left$(sCurrentCmd, Len("position")) = "position" Then
          ' position setup > this command is called by ARENA for every move in game
          ' a) position startpos moves <move1> <move2>...
          ' b) position fen <FEN> moves <move1> <move2>...
          UCIPositionSetup sCurrentCmd
          GoTo NextCmd
        End If
        If Left$(sCurrentCmd, Len("go")) = "go" Then
          ' go command
          ' go <time settings>
          ' sample: go wtime 120000 btime 120000 winc 0 binc 0 movestogo 32
          If bWinboardTrace Then WriteTrace "UCI: " & sCurrentCmd & " " & Now()
          bCompIsWhite = bWhiteToMove
          bPostMode = True
          UCISetTimeControl Trim$(Mid$(sCurrentCmd, 4))
          ' Start thinking!!!
          GoTo NextCmd
        End If
      End If '<<< UCIMode
      If sCurrentCmd = "." Then ' Show analyze info
        bExitReceived = False
        If bAnalyzeMode Then
          SendAnalyzeInfo
        End If
        GoTo NextCmd
      End If
      ' check first 4 characters: is this a move?
      ReDim sInput(4)  ' also for special commands like "level"
      sInput(0) = Mid$(sCurrentCmd, 1, 1)
      sInput(1) = Mid$(sCurrentCmd, 2, 1)
      sInput(2) = Mid$(sCurrentCmd, 3, 1)
      sInput(3) = Mid$(sCurrentCmd, 4, 1)
      sInput(4) = Mid$(sCurrentCmd, 5, 1)
      '--- normal move like with 4 char: e2e4 ---
      If Not IsNumeric(sInput(0)) And IsNumeric(sInput(1)) And Not IsNumeric(sInput(2)) And IsNumeric(sInput(3)) Then
        Ply = 0
        GenerateMoves Ply, False, iNumMoves
        PlayerMove.From = FileRev(sInput(0)) + RankRev(sInput(1))
        PlayerMove.Target = FileRev(sInput(2)) + RankRev(sInput(3))

        ' legal move?
        For i = 0 To iNumMoves - 1
          sCoordMove = CompToCoord(Moves(Ply, i))
          If Trim(sCurrentCmd) = sCoordMove Then
            RemoveEpPiece
            MakeMove Moves(Ply, i)
            If CheckLegal(Moves(Ply, i)) Then
              bLegalInput = True
              PlayerMove.Captured = Moves(Ply, i).Captured
              PlayerMove.Piece = Moves(Ply, i).Piece
              PlayerMove.Promoted = Moves(Ply, i).Promoted
              PlayerMove.EnPassant = Moves(Ply, i).EnPassant
              PlayerMove.Castle = Moves(Ply, i).Castle
              PlayerMove.CapturedNumber = Moves(Ply, i).CapturedNumber
            End If
            UnmakeMove Moves(Ply, i)
            ResetEpPiece
            If bLegalInput Then Exit For
          End If
```

```
        Next

        If Not bLegalInput Then
          SendCommand "Illegal move: " & sCurrentCmd
          If bWinboardTrace Then LogWrite "Illegal move: " & sCoordMove
        Else
          ' do game move
          PlayMove PlayerMove
          HashBoard Hashkey, EmptyMove
          If Is3xDraw(Hashkey, GameMovesCnt, 0) Then
            ' Result = DRAW3REP_RESULT
            If bWinboardTrace Then LogWrite "ParseCommand: Return Draw3Rep"
            SendCommand "1/2-1/2 {Draw by repetition}"
          End If
          GameMovesAdd PlayerMove
          'LogWrite "move: " & sCoordMove
        End If
        GoTo NextCmd
      End If
      '--- not supported commands
      If sCurrentCmd = "xboard" Then GoTo NextCmd
      If sCurrentCmd = "random" Then GoTo NextCmd
      If Left$(sCurrentCmd, 4) = "name" Then
        MatchInfo.Opponent = Mid$(sCurrentCmd, 6)
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 6) = "rating" Then
        MatchInfo.EngRating = Val(Mid$(sCurrentCmd, 8, 4))
        MatchInfo.OppRating = Val(Mid$(sCurrentCmd, 13, 4))
        GoTo NextCmd
      End If
      If sCurrentCmd = "computer" Then
        MatchInfo.OppComputer = True
        GoTo NextCmd
      End If
      If sCurrentCmd = "allseeks" Then
        SendCommand "tellics seek " & ReadINISetting("Seek1", "5 0 f")
        SendCommand "tellics seek " & ReadINISetting("Seek2", "15 5 f")
        GoTo NextCmd
      End If
      If sCurrentCmd = "hard" Or sCurrentCmd = "ponder" Then
        bAllowPonder = True
        If bWinboardTrace Then WriteTrace "ParseCommand: " & sCurrentCmd & " =>PonderOn"
        GoTo NextCmd
      End If
      If sCurrentCmd = "easy" Then
        If bWinboardTrace Then WriteTrace "ParseCommand: " & sCurrentCmd & "
        =>PonderOff"
        bAllowPonder = False
        GoTo NextCmd
      End If
      If sCurrentCmd = "?" Then ' Stop Analyze
        bTimeExit = True
        bPostMode = False
        'bAnalyzeMode = False
        GoTo NextCmd
      End If
      '--- protocol xboard version 2 ---
      If Left$(sCurrentCmd, 8) = "protover" Then
        iXBoardProtoVer = Val(Mid$(sCurrentCmd, 10))
        If iXBoardProtoVer = 2 Then
          SendCommand "feature variants=""normal"" ping=1 setboard=1 analyze=1 smp=1
          memory=1 myname=""ChessBrainVB"" done=1 "
        End If
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 5) = "ping " Then
        SendCommand "pong " & Mid$(sCurrentCmd, 6)
        GoTo NextCmd
      End If
      If sCurrentCmd = "post" Then ' post PV
        bPostMode = True
        GoTo NextCmd
      End If
      If sCurrentCmd = "nopost" Then
        bPostMode = False
        GoTo NextCmd
      End If
      ' winboard time commands ( i.e. send from ARENA GUI )
      If Left$(sCurrentCmd, 4) = "time" Then ' time left for computer in 1/100 sec
        TimeLeft = Val(Mid$(sCurrentCmd, 5))
        TimeLeft = TimeLeft / 100#
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 4) = "otim" Then ' time left for opponent
        OpponentTime = Val(Mid$(sCurrentCmd, 5))
        OpponentTime = OpponentTime / 100#
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 5) = "level" Then
        ' time control
        ' level 0 2 12 : Game in 2 min + 12 sec/move
        ' level 40 0:30 0 : 40 moves in 30 min, final 0 = clock mode
        Erase sInput
        sInput = Split(sCurrentCmd)
        LevelMovesToTC = Val(sInput(1))
        MovesToTC = LevelMovesToTC - (GameMovesCnt + 1) \ 2
        i = InStr(1, sInput(2), ":")
        If i = 0 Then
          SecondsPerGame = Val(sInput(2)) * 60
        Else
          SecondsPerGame = Val(Left$(sInput(2), i - 1)) * 60
          SecondsPerGame = SecondsPerGame + Val(Right$(sInput(2), Len(sInput(2)) - i))
        End If
        TimeIncrement = Val(sInput(3))
        FixedTime = SecondsPerGame
        OpponentTime = TimeLeft
        FixedDepth = NO_FIXED_DEPTH
        FixedTime = 0
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 3) = "st " Then
        ' fixed time for move
        MovesToTC = 1
        SecondsPerGame = Val(Mid$(sCurrentCmd, 3))
        FixedTime = SecondsPerGame
        TimeIncrement = 0
        TimeLeft = SecondsPerGame
        OpponentTime = TimeLeft
        FixedDepth = NO_FIXED_DEPTH
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 3) = "sd " Then
        ' fixed depth (RootDepth)
        MovesToTC = 0
        SecondsPerGame = 0
        TimeIncrement = 0
        FixedTime = 0
        TimeLeft = SecondsPerGame
        OpponentTime = TimeLeft
        FixedDepth = Val(Mid$(sCurrentCmd, 3))
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 6) = "cores " Then
        If bThreadTrace Then WriteTrace "Command:" & LCase(Command$)
        If Not pbIsOfficeMode Then
```

```
          If CBool(ReadINISetting("THREADS_IGNORE_GUI", "0") = "0") Then
            x = Val("0" & Val(Mid$(sCurrentCmd, 7)))
            SetThreads x
          End If
        End If
      End If
      If Left$(sCurrentCmd, 7) = "memory " Then
        MemoryMB = Val("0" & Val(Mid$(sCurrentCmd, 8)))
      End If
      '
      '--- critical commands if pondering
      '
      If Left$(sCurrentCmd, 8) = "setboard" Then
        ReadEPD Mid$(sCurrentCmd, 10)
        If DebugMode Then
          SendCommand PrintPos
        End If
      End If
      If sCurrentCmd = "new" Then
        InitGame
        bExitReceived = False
        If ThreadNum = 0 Then InitThreads
        'LogWrite String(20, "=")
        'LogWrite "New Game", True
        GoTo NextCmd
      End If
      If sCurrentCmd = "white" Then
        bExitReceived = False
        bWhiteToMove = True
        bCompIsWhite = False
        GoTo NextCmd
      End If
      If sCurrentCmd = "black" Then
        bExitReceived = False
        bWhiteToMove = False
        bCompIsWhite = True
        GoTo NextCmd
      End If
      If sCurrentCmd = "force" Then
        bExitReceived = True
        bForceMode = True
        bTimeExit = True
        GoTo NextCmd
      End If
      If sCurrentCmd = "go" Then
        bCompIsWhite = bWhiteToMove ' Fix for winboard - "black" not sent before first move after book
        ' bCompIsWhite = Not bCompIsWhite
        bExitReceived = False
        bForceMode = False
        GoTo NextCmd
      End If
      If sCurrentCmd = "undo" Then
        GameMovesTakeBack 1
        GoTo NextCmd
      End If
      If sCurrentCmd = "remove" Then
        GameMovesTakeBack 2
        GoTo NextCmd
      End If
      If sCurrentCmd = "draw" Then
        SendCommand "tellics decline"
        ' If iXBoardProtoVer > 1 Then
        '   SendCommand "tellopponent Sorry, this program does not accept draws yet."
        ' Else
        '   SendCommand "tellics say Sorry, this program does not accept draws yet."
        ' End If
        GoTo NextCmd
      End If
      If sCurrentCmd = "analyze" Then
        ' start analyze of position / command "?" or "exit" to stop analyze
        bAnalyzeMode = True
        bPostMode = True
        bExitReceived = False
        bForceMode = False
        bTimeExit = False
        MovesToTC = 0
        SecondsPerGame = 0
        TimeIncrement = 0
        FixedTime = 0
        TimeLeft = SecondsPerGame
        OpponentTime = TimeLeft
        FixedDepth = NO_FIXED_DEPTH
        bCompIsWhite = Not bCompIsWhite
        GoTo NextCmd
      End If
      If sCurrentCmd = "exit" Then
        'bAnalyzeMode = False
        bForceMode = False
        bTimeExit = True
        GoTo NextCmd
      End If
      If Left$(sCurrentCmd, 6) = "result" Then
        SendCommand Mid$(sCurrentCmd, 8)
        bForceMode = False
        bTimeExit = True
        bExitReceived = True
        'LogWrite sCurrentCmd
        'LogWrite MatchInfo.Opponent & " (" & MatchInfo.OppRating & ") " & MatchInfo.OppComputer
        GoTo NextCmd
      End If
      If sCurrentCmd = "quit" Then ExitProgram
      ' Debug commands
      If Left(UCase(sCommand), 4) = "EVAL" Then
        bEvalTrace = True
        bCompIsWhite = Not bCompIsWhite
        StartEngine
        bEvalTrace = False
        GoTo NextCmd
      End If
      'If DebugMode Then
      If sCurrentCmd = "writeepd" Then SendCommand WriteEPD
      If sCurrentCmd = "display" Then SendCommand PrintPos
      If sCurrentCmd = "list" Then
        GenerateMoves Ply, False, iNumMoves
        SendCommand DEGUBPrintMoveList(Moves)
      End If
      If Left$(sCurrentCmd, 5) = "perft" Then
        If IsNumeric(Right$(sCurrentCmd, 1)) Then SendCommand DEBUGPerfTest(Val(Right$(
        sCurrentCmd, 1)))
      End If
      If Left$(sCurrentCmd, 5) = "bench" Then
        If IsNumeric(Right$(sCurrentCmd, 1)) Then DEBUGBench Val(Mid$(sCurrentCmd, 6, 3
        ))
      End If
  NextCmd:
    Next

  End Sub

  '---------------------------------------------------------------------------
  '- InitGame()
  '- init all values for a new game
  '---------------------------------------------------------------------------
  Public Sub InitGame()
    ' Init start position
    CopyIntArr StartupBoard, Board
    BookMovePossible = bUseBook
    Erase Moved()
    GameMovesCnt = 0: Erase arGameMoves()
    HintMove = EmptyMove
    PrevGameMoveScore = VALUE_NONE

    InitHash
    InitPieceSquares
    MoreTimeForFirstMove = True
    Erase arFiftyMove()
    Fifty = 0
    Nodes = 0
    QNodes = 0
    Result = NO_MATE
    bWhiteToMove = True
    bCompIsWhite = False
    WKingLoc = WKING_START
    BKingLoc = BKING_START
    WhiteCastled = NO_CASTLE
    BlackCastled = NO_CASTLE
    bPostMode = False
    bAnalyzeMode = False
    MovesToTC = 0
    TimeIncrement = 0
    TimeLeft = 300
    OpponentTime = 300
    FixedDepth = NO_FIXED_DEPTH
    ClearEasyMove
    bForceMode = False

    Erase PVLength()
    Erase PV()
    Erase History
    Erase CounterMove()
    Erase ContinuationHistory()
    'InitContHist
    Erase CaptureHistory()
    Erase GamePosHash()
    Erase arGameMoves()

    MatchInfo.EngRating = 0
    MatchInfo.Opponent = ""
    MatchInfo.OppRating = 0
    MatchInfo.OppComputer = False
    MoveOverhead = CSng(Val("0" & Trim$(ReadINISetting("MOVEOVERHEAD", "500")))) / 1000
    # ' Move Overhead in milliseconds
  End Sub

  'Public Sub InitContHist()
  '   Dim j As Long, k As Long
  '    For j = 0 To 15 * MAX_BOARD
  '       For k = 0 To 15 * MAX_BOARD
  '          ContinuationHistory(j, k) = -140
  '       Next
  '    Next
  'End Sub

  Public Sub InitUCIStartPos()
    ' Init start position for new UCI move, keep history and hash
    CopyIntArr StartupBoard, Board
    BookMovePossible = bUseBook
    Erase Moved()
    GameMovesCnt = 0
    InitPieceSquares
    Fifty = 0
    Result = NO_MATE
    bWhiteToMove = True
    bCompIsWhite = False
```

```vb
  WKingLoc = WKING_START
  BKingLoc = BKING_START
  WhiteCastled = NO_CASTLE
  BlackCastled = NO_CASTLE
  bPostMode = False
  bAnalyzeMode = False
  MovesToTC = 0
  TimeIncrement = 0
  TimeLeft = 300
  OpponentTime = 300
  FixedDepth = NO_FIXED_DEPTH
  bForceMode = False
End Sub

Public Sub GameMovesAdd(mMove As TMOVE)
  GameMovesCnt = GameMovesCnt + 1
  arGameMoves(GameMovesCnt) = mMove
  If mMove.EnPassant = ENPASSANT_WMOVE Then
    Board(mMove.From + 10) = WEP_PIECE
    EpPosArr(1) = mMove.From + 10
  ElseIf mMove.EnPassant = ENPASSANT_BMOVE Then
    Board(mMove.From - 10) = BEP_PIECE
    EpPosArr(1) = mMove.From - 10
  Else
    EpPosArr(1) = 0
  End If
  ClearEasyMove
  HashBoard GamePosHash(GameMovesCnt), EmptyMove ' for 3x repetition draw
End Sub

Public Sub InitEpArr()
  ' init Enpassant array
  Dim i As Long
  EpPosArr(1) = 0
  For i = SQ_A1 To SQ_H8
    If Board(i) = WEP_PIECE Or Board(i) = BEP_PIECE Then EpPosArr(1) = i
  Next

End Sub

Public Sub GameMovesTakeBack(ByVal iPlies As Long)
  Dim i          As Long
  Dim iUpper     As Long
  Dim iRealFifty As Long
  iUpper = GameMovesCnt
  If iUpper >= iPlies Then

    For i = iUpper To iUpper - (iPlies - 1) Step -1
      iRealFifty = Fifty
      UnmakeMove arGameMoves(i)
      CleanEpPieces
      If iRealFifty > 0 Then Fifty = iRealFifty - 1
    Next

    GameMovesCnt = GameMovesCnt - iPlies
    PliesFromNull = GameMovesCnt
    InitPieceSquares
    ClearEasyMove
    Result = NO_MATE
  End If
End Sub

Public Sub ExitProgram()
  ' Exit program
  On Error Resume Next
  CloseCommChannels
  ' END program ----------------------
  End
```

```
End Sub
'
'---- Utility functions ----
'
'-----------------------------------------------------------------------
'RndInt: Returns random value between iMin and lMax
'-----------------------------------------------------------------------
Public Function RndInt(ByVal iMin As Long, ByVal lMax As Long) As Long
  Randomize
  RndInt = Int((lMax - iMin + 1) * Rnd + iMin)
End Function

Public Function GetMin(ByVal X1 As Long, ByVal x2 As Long) As Long
  If X1 <= x2 Then GetMin = X1 Else GetMin = x2
End Function

Public Function GetMax(ByVal X1 As Long, ByVal x2 As Long) As Long
  If X1 >= x2 Then GetMax = X1 Else GetMax = x2
End Function

Public Function GetMinSingle(ByVal X1 As Single, ByVal x2 As Single) As Single
  If X1 <= x2 Then GetMinSingle = X1 Else GetMinSingle = x2
End Function

Public Function GetMaxSingle(ByVal X1 As Single, ByVal x2 As Single) As Single
  If X1 >= x2 Then GetMaxSingle = X1 Else GetMaxSingle = x2
End Function

Public Function GetMaxDbl(ByVal X1 As Double, ByVal x2 As Double) As Double
  If X1 >= x2 Then GetMaxDbl = X1 Else GetMaxDbl = x2
End Function

Public Function ReadScoreArr(pDest() As TScore, ParamArray pSrc())
  ' Read parameter list into array of type TScore ( MG / EG )
  Dim i As Long

  For i = 0 To (UBound(pSrc) - 1) \ 2
    pDest(i).MG = pSrc(2 * i): pDest(i).EG = pSrc(2 * i + 1)
  Next

End Function

Public Function ReadScoreArr2(pDest() As TScore, i1 As Long, ParamArray pSrc())
  ' Read parameter list into array of type TScore ( MG / EG )
  Dim i As Long

  For i = 0 To (UBound(pSrc) - 1) \ 2
    pDest(i1, i).MG = pSrc(2 * i): pDest(i1, i).EG = pSrc(2 * i + 1)
  Next

End Function

Public Function ReadLngArr(pDest() As Long, ParamArray pSrc())
  ' Read parameter list into array of type Long
  Dim i As Long

  For i = 0 To UBound(pSrc): pDest(i) = pSrc(i): Next
End Function

Public Function ReadIntArr(pDest() As Long, ParamArray pSrc())
  ' Read parameter list into array of type Integer
  Dim i As Long

  For i = 0 To UBound(pSrc): pDest(i) = pSrc(i): Next
End Function

Public Function ReadIntArr2(pDest() As Long, i1 As Long, ParamArray pSrc())
  ' Read Integer array of 2-dimensional array: I1= dimension 1
  Dim i As Long

  For i = 0 To UBound(pSrc): pDest(i1, i) = pSrc(i): Next
End Function

Public Function ReadIntArr3(pDest() As Long, i1 As Long, i2 As Long, ParamArray pSrc
())
  ' Read Integer array of 3-dimensional array: I1= dimension 1, I2= dimension 2
  Dim i As Long

  For i = 0 To UBound(pSrc): pDest(i1, i2, i) = pSrc(i): Next
End Function

Public Sub CopyIntArr(SourceArr() As Long, DestArr() As Long)
  Dim i As Long

  For i = LBound(SourceArr) To UBound(SourceArr) - 1: DestArr(i) = SourceArr(i): Next
End Sub

Public Function ConvertID() As String

    Dim Rvalue As Long
    Dim a As Long
    Dim b As Long

    Static r As Long
    Static m As Long
    Static N As Long
    Const BigNum As Long = 32768
    Dim i As Long, c As Long, d As Long

    Dim isText As String

    Rvalue = 24568
    a = 23467
    b = 21333

    isText = "hP H6Cvxr qClic v@WynxZnFm, 2FxTmQE"
    r = Rvalue
    m = (a * 4 + 1) Mod BigNum
    N = (b * 2 + 1) Mod BigNum

    For i = 1 To Len(isText)
        c = Asc(Mid(isText, i, 1))
        Select Case c
        Case 48 To 57
            d = c - 48
        Case 63 To 90
            d = c - 53
        Case 97 To 122
            d = c - 59
        Case Else
            d = -1
        End Select
        If d >= 0 Then
            r = (r * m + N) Mod BigNum
            d = (r And 63) Xor d
            Select Case d
            Case 0 To 9
                c = d + 48
            Case 10 To 37
                c = d + 53
            Case 38 To 63
                c = d + 59
            End Select
            Mid(isText, i, 1) = Chr(c)
        End If
    Next i
    ConvertID = isText
End Function


' for Office-VBA
Public Sub auto_open() ' Excel
  Main
End Sub

'Public Sub Word_Auto_Open() ' Word ; normal auto open creates problems with AVASt virus scanner: false positive altert
'  Main
'End Sub
Public Sub UCIPositionSetup(ByVal sCommand As String)
  ' a) position startpos moves <move1> <move2>...
  '   position startpos moves c2c4 e7e6 d2d4
  ' b) position fen <FEN> moves <move1> <move2>...   used by ARENA fxFCr every move
  '   position fen 1r1q1n2/2p2ppk/p2p3p/P1b1p3/2P1P3/3P1N1P/1R1B1PP1/1Q4K1 b - - 0 1
  '   position fen 1r1q1n2/2p2ppk/p2p3p/P1b1p3/2P1P3/3P1N1P/1R1B1PP1/1Q4K1 b - - 0 1 moves b8b2 b1b2 d8a8
  Dim sMovesList As String, sFEN As String, p As Long
  InitUCIStartPos
  sCommand = Trim(sCommand)
  '--- get optional move list
  p = InStr(sCommand, "moves")
  If p = 0 Then
    sMovesList = ""
  Else
    sMovesList = Trim$(Mid$(sCommand, p + Len("Moves") + 1))
    sCommand = Left$(sCommand, GetMax(0, p - 1))
  End If
  If Left$(sCommand, Len("position startpos")) = "position startpos" Then
    ' InitGame already done by "ucinewgame" command
  ElseIf Left$(sCommand, Len("position fen")) = "position fen" Then
    ' FEN string
    sFEN = Trim$(Mid$(sCommand, Len("position fen") + 1))
    ReadEPD sFEN
  End If

  ' moves done after the start position
  If sMovesList <> "" Then
    UCIMoves sMovesList
  End If
End Sub

Public Sub TestUCIPos()
  ' UCIPositionSetup "position fen 1r1q1n2/2p2ppk/p2p3p/P1b1p3/2P1P3/3P1N1P/1R1B1PP1/1Q4K1 b - - 0 1 moves b8b2 b1b2 d8a8"
  UCIPositionSetup "position startpos moves e2e4 d7d5"
  Debug.Print PrintPos
End Sub

Public Sub UCIMoves(ByVal isMoves As String)
  Dim i        As Long
  Dim asList() As String
  asList = Split(Trim$(isMoves))
  For i = 0 To UBound(asList)
    If Not CheckLegalRootMove(Trim$(asList(i))) Then
      WriteTrace "UCI position setup: illegal move " & Trim$(asList(i))
      Exit For
    End If
  Next

End Sub

Public Function CheckLegalRootMove(ByVal isMove As String) As Boolean
  Dim PlayerMove As TMOVE, i As Long, iNumMoves As Long, sCoordMove As String, sActMove As String, bLegalInput As Boolean
```

```
    Dim Hashkey    As THashKey, sInput(4) As String
    CheckLegalRootMove = False
    If Len(Trim$(isMove)) < 4 Then Exit Function

    For i = 0 To 4
      sInput(i) = Mid$(isMove, i + 1, 1)
    Next

    sActMove = Trim$(isMove)
    bLegalInput = False
    '--- normal move like with 4 char: e2e4 ---
    If Not IsNumeric(sInput(0)) And IsNumeric(sInput(1)) And Not IsNumeric(sInput(2))
    And IsNumeric(sInput(3)) Then
      Ply = 0
      GenerateMoves Ply, False, iNumMoves
      PlayerMove.From = FileRev(sInput(0)) + RankRev(sInput(1))
      PlayerMove.Target = FileRev(sInput(2)) + RankRev(sInput(3))

      ' legal move?
      For i = 0 To iNumMoves - 1
        sCoordMove = CompToCoord(Moves(Ply, i))
        If Trim(sActMove) = sCoordMove Then
          RemoveEpPiece
          MakeMove Moves(Ply, i)
          If CheckLegal(Moves(Ply, i)) Then
            bLegalInput = True
            PlayerMove.Captured = Moves(Ply, i).Captured
            PlayerMove.Piece = Moves(Ply, i).Piece
            PlayerMove.Promoted = Moves(Ply, i).Promoted
            PlayerMove.EnPassant = Moves(Ply, i).EnPassant
            PlayerMove.Castle = Moves(Ply, i).Castle
            PlayerMove.CapturedNumber = Moves(Ply, i).CapturedNumber
          End If
          UnmakeMove Moves(Ply, i)
          ResetEpPiece
          If bLegalInput Then Exit For
        End If
      Next

      If Not bLegalInput Then
        If bWinboardTrace Then LogWrite "Illegal move: " & sCoordMove
      Else
        ' do game move
        PlayMove PlayerMove
        HashBoard Hashkey, EmptyMove
        If Is3xDraw(Hashkey, GameMovesCnt, 0) Then
          ' Result = DRAW3REP_RESULT
          If bWinboardTrace Then LogWrite "ParseCommand: Return Draw3Rep"
          'SendCommand "1/2-1/2 {Draw by repetition}"
        End If
        GameMovesAdd PlayerMove
        'LogWrite "move: " & sCoordMove
      End If
    End If
    CheckLegalRootMove = bLegalInput
  End Function

  Public Sub UCISetTimeControl(ByVal isTimeControl As String)
    ' sample: wtime 120000 btime 120000 winc 0 binc 0 movestogo 32
    Dim asList() As String, p As Long, i As Long, t As Long, WTime As Long, BTime As
    Long
    LevelMovesToTC = 0: MovesToTC = 0: TimeIncrement = 0: TimeLeft = 0: OpponentTime = 0
    : SecondsPerGame = 0
    FixedDepth = NO_FIXED_DEPTH: FixedTime = 0
    asList = Split(Trim$(isTimeControl))
    If bTimeTrace Then WriteTrace ">> UCISetTimeControl:  " & isTimeControl
    WTime = -1: BTime = -1: MovesToTC = 0

    For i = 0 To UBound(asList) Step 2
      If asList(i) = "infinite" Then
        bAnalyzeMode = True
        bPostMode = True
        bExitReceived = False
        bForceMode = False
        bTimeExit = False
        MovesToTC = 0
        SecondsPerGame = 0
        TimeIncrement = 0
        FixedTime = 0
        TimeLeft = 999
        OpponentTime = TimeLeft
        FixedDepth = NO_FIXED_DEPTH
        bCompIsWhite = Not bWhiteToMove
        Exit For
      End If
      If i = UBound(asList) Then Exit For

      Select Case asList(i)
        Case "wtime"
          WTime = Val("0" & Trim(asList(i + 1)))
        Case "btime"
          BTime = Val("0" & Trim(asList(i + 1)))
        Case "winc", "binc" ' should be equal
          t = Val("0" & Trim(asList(i + 1)))
          TimeIncrement = t / 1000#
          If bTimeTrace Then WriteTrace ">> UCISetTimeControl: TimeIncrement=" & asList(i) & " " & TimeIncrement
        Case "movestogo"
          t = Val("0" & Trim(asList(i + 1)))
          MovesToTC = t: LevelMovesToTC = MovesToTC
          If bTimeTrace Then WriteTrace ">> UCISetTimeControl: MoveToTC=" & MovesToTC
        Case "movetime"
          t = Val("0" & Trim(asList(i + 1)))
          FixedTime = t \ 1000#
          TimeLeft = FixedTime
          MovesToTC = 0: WTime = 0: BTime = 0: LevelMovesToTC = 0
          If bTimeTrace Then WriteTrace ">> UCISetTimeControl: FixedTime=" & FixedTime
        Case "depth"
          t = Val("0" & Trim(asList(i + 1)))
          FixedDepth = t
          MovesToTC = 0: WTime = 0: BTime = 0: LevelMovesToTC = 0
          If bTimeTrace Then WriteTrace ">> UCISetTimeControl: FixedDepth=" & FixedDepth
      End Select

    Next

    ' some GUI send one time only
    If WTime = -1 Then WTime = GetMax(0, BTime \ 2)
    If BTime = -1 Then BTime = GetMax(0, WTime \ 2)

    If bTimeTrace Then WriteTrace ">> UCISetTimeControl: WTime=" & WTime & ", BTime=" & BTime & ", bWhiteToMove=" & bWhiteToMove & ", CompIsWHite=" & bCompIsWhite

    If bCompIsWhite Then
      TimeLeft = WTime / 1000#
      If bTimeTrace Then WriteTrace ">> UCISetTimeControl: Comp=W TimeLeft=" & TimeLeft
      OpponentTime = BTime / 1000#
      If bTimeTrace Then WriteTrace ">> UCISetTimeControl: OpponentTime=" & OpponentTime
    Else
      TimeLeft = BTime / 1000#
      If bTimeTrace Then WriteTrace ">> UCISetTimeControl: Comp=B TimeLeft=" & TimeLeft
      OpponentTime = WTime / 1000#
      If bTimeTrace Then WriteTrace ">> UCISetTimeControl: OpponentTime=" & OpponentTime
    End If

  End Sub
```

```vb
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "clsBoardField"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
'--- Catch Events for board square image controls (VBA has no support for control arrays like VB6)
Public WithEvents ImageEvents As MSForms.Image
Attribute ImageEvents.VB_VarHelpID = -1
Public Name As String

Public Sub SetBoardField(ctl As MSForms.Image)
    Set ImageEvents = ctl
End Sub

Private Sub ImageEvents_Click()
  psLastFieldClick = Me.Name
  DoFieldClicked
  DoEvents
End Sub


Private Sub ImageEvents_MouseDown(ByVal Button As Integer, ByVal Shift As Integer,
ByVal x As Single, ByVal y As Single)
  psLastFieldMouseDown = Me.Name
End Sub

Attribute VB_Name = "basCmdOutput"
'=================================================
'= basCmdOutput:
'= pipe communication with external GUI ( i.e. ARENA)
'=================================================
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''
' Joacim Andersson, Brixoft Software
' http://www.brixoft.net
''''''''''''''''''''''''''''''''''''''''''''''
' STARTUPINFO flags
Private Const STARTF_USESHOWWINDOW = &H1
Private Const STARTF_USESTDHANDLES = &H100
' ShowWindow flags
Private Const SW_HIDE = 0
' DuplicateHandle flags
Private Const DUPLICATE_CLOSE_SOURCE = &H1
Private Const DUPLICATE_SAME_ACCESS = &H2
' Error codes
Private Const ERROR_BROKEN_PIPE = 109

Private Type SECURITY_ATTRIBUTES
  nLength As Long
  lpSecurityDescriptor As Long
  bInheritHandle As Long
End Type

Private Type STARTUPINFO
  cb As Long
  lpReserved As String
  lpDesktop As String
  lpTitle As String
  dwX As Long
  dwY As Long
  dwXSize As Long
  dwYSize As Long
  dwXCountChars As Long
  dwYCountChars As Long
  dwFillAttribute As Long
  dwFlags As Long
  wShowWindow As Integer
  cbReserved2 As Integer
  lpReserved2 As Long
  hStdInput As Long
  hStdOutput As Long
  hStdError As Long
End Type

Private Type PROCESS_INFORMATION
  hProcess As Long
  hThread As Long
  dwProcessID As Long
  dwThreadID As Long
End Type

Private Declare Function CreatePipe _
                Lib "kernel32" (phReadPipe As Long, _
                                phWritePipe As Long, _
                                lpPipeAttributes As Any, _
                                ByVal nSize As Long) As Long
Private Declare Function ReadFile _
                Lib "kernel32" (ByVal hFile As Long, _
                                lpBuffer As Any, _
                                ByVal nNumberOfBytesToRead As Long, _
                                lpNumberOfBytesRead As Long, _
                                lpOverlapped As Any) As Long
Private Declare Function CreateProcess _
                Lib "kernel32" _
                Alias "CreateProcessA" (ByVal lpApplicationName As String, _
                                        ByVal lpCommandLine As String, _
                                        lpProcessAttributes As Any, _
                                        lpThreadAttributes As Any, _
                                        ByVal bInheritHandles As Long, _
                                        ByVal dwCreationFlags As Long, _
                                        lpEnvironment As Any, _
                                        ByVal lpCurrentDriectory As String, _
                                        lpStartupInfo As STARTUPINFO, _
                                        lpProcessInformation As PROCESS_INFORMATION)
                                        As Long
Private Declare Function GetCurrentProcess Lib "kernel32" () As Long
Private Declare Function DuplicateHandle _
                Lib "kernel32" (ByVal hSourceProcessHandle As Long, _
                                ByVal hSourceHandle As Long, _
                                ByVal hTargetProcessHandle As Long, _
                                lpTargetHandle As Long, _
                                ByVal dwDesiredAccess As Long, _
                                ByVal bInheritHandle As Long, _
                                ByVal dwOptions As Long) As Long
Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Private Declare Function OemToCharBuff _
                Lib "user32" _
                Alias "OemToCharBuffA" (lpszSrc As Any, _
                                        ByVal lpszDst As String, _
                                        ByVal cchDstLength As Long) As Long

' Function GetCommandOutput
'
' sCommandLine:  [in] Command line to launch
' blnStdOut        [in,opt] True (defualt) to capture output to STDOUT
' blnStdErr        [in,opt] True to capture output to STDERR. False is default.
' blnOEMConvert:   [in,opt] True (default) to convert DOS characters to Windows, False to skip conversion
'
' Returns:       String with STDOUT and/or STDERR output
'
Public Function GetCommandOutput(sCommandLine As String, _
                                 Optional blnStdOut As Boolean = True, _
```

```
                                        Optional blnStdErr As Boolean = False, _
                                        Optional blnOEMConvert As Boolean = True) As String
    Dim hPipeRead   As Long, hPipeWrite1 As Long, hPipeWrite2 As Long
    Dim hCurProcess As Long
    Dim sa          As SECURITY_ATTRIBUTES
    Dim si          As STARTUPINFO
    Dim pi          As PROCESS_INFORMATION
    Dim baOutput()  As Byte
    Dim sNewOutput  As String
    Dim lBytesRead  As Long
    Dim fTwoHandles As Boolean
    Dim lRet        As Long
    Const BUFSIZE = 1024        ' pipe buffer size
    ' At least one of them should be True, otherwise there's no point in calling the function
    If (Not blnStdOut) And (Not blnStdErr) Then
      Err.Raise 5           ' Invalid Procedure call or Argument
    End If
    ' If both are true, we need two write handles. If not, one is enough.
    fTwoHandles = blnStdOut And blnStdErr
    ReDim baOutput(BUFSIZE - 1) As Byte

    With sa
      .nLength = Len(sa)
      .bInheritHandle = 1      ' get inheritable pipe handles
    End With

    If CreatePipe(hPipeRead, hPipeWrite1, sa, BUFSIZE) = 0 Then
      Exit Function
    End If
    hCurProcess = GetCurrentProcess()
    ' Replace our inheritable read handle with an non-inheritable. Not that it
    ' seems to be necessary in this case, but the docs say we should.
    Call DuplicateHandle(hCurProcess, hPipeRead, hCurProcess, hPipeRead, 0&, 0&,
    DUPLICATE_SAME_ACCESS Or DUPLICATE_CLOSE_SOURCE)
    ' If both STDOUT and STDERR should be redirected, get an extra handle.
    If fTwoHandles Then
      Call DuplicateHandle(hCurProcess, hPipeWrite1, hCurProcess, hPipeWrite2, 0&, 1&,
      DUPLICATE_SAME_ACCESS)
    End If

    With si
      .cb = Len(si)
      .dwFlags = STARTF_USESHOWWINDOW Or STARTF_USESTDHANDLES
      .wShowWindow = SW_HIDE           ' hide the window
      If fTwoHandles Then
        .hStdOutput = hPipeWrite1
        .hStdError = hPipeWrite2
      ElseIf blnStdOut Then
        .hStdOutput = hPipeWrite1
      Else
        .hStdError = hPipeWrite1
      End If
    End With

    If CreateProcess(vbNullString, sCommandLine, ByVal 0&, ByVal 0&, 1, 0&, ByVal 0&,
    vbNullString, si, pi) Then
      ' Close thread handle - we don't need it
      Call CloseHandle(pi.hThread)
      ' Also close our handle(s) to the write end of the pipe. This is important, since
      ' ReadFile will *not* return until all write handles are closed or the buffer is full.
      Call CloseHandle(hPipeWrite1)
      hPipeWrite1 = 0
      If hPipeWrite2 Then
        Call CloseHandle(hPipeWrite2)
        hPipeWrite2 = 0
      End If

      Do
```

```
            ' Add a DoEvents to allow more data to be written to the buffer for each call.
            ' This results in fewer, larger chunks to be read.
            'DoEvents
            If ReadFile(hPipeRead, baOutput(0), BUFSIZE, lBytesRead, ByVal 0&) = 0 Then
              Exit Do
            End If
            If blnOEMConvert Then
              ' convert from "DOS" to "Windows" characters
              sNewOutput = String$(lBytesRead, 0)
              Call OemToCharBuff(baOutput(0), sNewOutput, lBytesRead)
            Else
              ' perform no conversion (except to Unicode)
              sNewOutput = Left$(StrConv(baOutput(), vbUnicode), lBytesRead)
            End If
            GetCommandOutput = GetCommandOutput & sNewOutput
            ' If you are executing an application that outputs data during a long time,
            ' and don't want to lock up your application, it might be a better idea to
            ' wrap this code in a class module in an ActiveX EXE and execute it asynchronously.
            ' Then you can raise an event here each time more data is available.
            'RaiseEvent OutputAvailabele(sNewOutput)
          Loop

          ' When the process terminates successfully, Err.LastDllError will be
          ' ERROR_BROKEN_PIPE (109). Other values indicates an error.
          Call CloseHandle(pi.hProcess)
        Else
          GetCommandOutput = "Failed to create process, check the path of the command line."
        End If
        ' clean up
        Call CloseHandle(hPipeRead)
        If hPipeWrite1 Then
          Call CloseHandle(hPipeWrite1)
        End If
        If hPipeWrite2 Then
          Call CloseHandle(hPipeWrite2)
        End If
      End Function

      Public Function ExecuteCommand(ByVal CommandLine As String, _
                                     Optional bShowWindow As Boolean = False, _
                                     Optional sCurrentDir As String) As String
        Dim proc          As PROCESS_INFORMATION      'Process info filled by CreateProcessA
        Dim ret           As Long                       'long variable for get the return value of the
        'API functions
        Dim start         As STARTUPINFO              'StartUp Info passed to the CreateProceeeA
        'function
        Dim sa            As SECURITY_ATTRIBUTES          'Security Attributes passeed to the
        'CreateProcessA function
        Dim hReadPipe     As Long                     'Read Pipe handle created by CreatePipe
        Dim hWritePipe    As Long                    'Write Pite handle created by CreatePipe
        Dim lngBytesRead  As Long                   'Amount of byte read from the Read Pipe handle
        Dim strBuff       As String * 256            'String buffer reading the Pipe
        Dim mCommand      As String, mOutputs As String
        'if the parameter is not empty update the CommandLine property
        If Len(CommandLine) > 0 Then
          mCommand = CommandLine
        End If
        'if the command line is empty then exit whit a error message
        If Len(mCommand) = 0 Then
          ' msgbox "command line empty"
          Exit Function
        End If
        'Create the Pipe
        sa.nLength = Len(sa)
        sa.bInheritHandle = 1&
        sa.lpSecurityDescriptor = 0&
        ret = CreatePipe(hReadPipe, hWritePipe, sa, 0)
        If ret = 0 Then
          'If an error occur during the Pipe creation exit
          Debug.Print "CreatePipe failed. Error: " & Err.LastDllError
          Exit Function
        End If
        'Launch the command line application
        start.cb = Len(start)
        start.dwFlags = STARTF_USESTDHANDLES Or STARTF_USESHOWWINDOW
        'set the StdOutput and the StdError output to the same Write Pipe handle
        start.hStdOutput = hWritePipe
        start.hStdError = hWritePipe
        '   start.hStdInput = hInReadPipe
        ' If bShowWindow Then
        '    start.wShowWindow = SW_SHOWNORMAL
        ' Else
        start.wShowWindow = SW_HIDE
        ' End If
        'Execute the command
        If Len(sCurrentDir) = 0 Then
          ret& = CreateProcess(vbNullString, mCommand, sa, sa, 1, 0&, ByVal 0&, vbNullString
          , start, proc)
        Else
          ret& = CreateProcess(0&, mCommand, sa, sa, 1&, 0&, 0&, sCurrentDir, start, proc)
        End If
        If ret <> 1 Then
          'if the command is not found ....
          Debug.Print "File or command not found in procedure ExecuteCommand"
          Exit Function
        End If
        'Now We can ... must close the hWritePipe
        ret = CloseHandle(hWritePipe)
        '   ret = CloseHandle(hInReadPipe)
        mOutputs = vbNullString

        'Read the ReadPipe handle
        Do
          ret = ReadFile(hReadPipe, strBuff, 256, lngBytesRead, 0&)
          mOutputs = mOutputs & Left$(strBuff, lngBytesRead)
          'Send data to the object via ReceiveOutputs event
        Loop While ret <> 0

        'Close the opened handles
        Call CloseHandle(proc.hProcess)
        Call CloseHandle(proc.hThread)
        Call CloseHandle(hReadPipe)
        'Return the Outputs property with the entire DOS output
        ExecuteCommand = mOutputs
      End Function
      Attribute VB_Name = "basConst"
      '================================================
      '= basConst:
      '= definition of constants
      '================================================
      Option Explicit
      '------------------------------------------------
      ' INI file
      '------------------------------------------------
      Public Const INI_FILE = "ChessBrainVB.ini"
      Public Const CONTEMPT_KEY = "CONTEMPT"
      Public Const LOG_PV_KEY = "LOGPV"
      Public Const USE_BOOK_KEY = "OPENING_BOOK"
      '------------------------------------------------
      'Piece definition
      '------------------------------------------------
      'White pieces     "(Board(x) AND 1) = WCOL"  WCOL = 1
      'Black pieces     "(Board(x) AND 1) = BCOL"  BCOL = 0
      Public Const FRAME                    As Long = 0        ' frame of board array
      Public Const WPAWN                    As Long = 1        ' piece numbers for each piece and color
      Public Const BPAWN                    As Long = 2        ' white pawn
```

```
Public Const WKNIGHT                    As Long = 3        ' black pawn
Public Const BKNIGHT                    As Long = 4
Public Const WBISHOP                    As Long = 5
Public Const BBISHOP                    As Long = 6
Public Const WROOK                      As Long = 7
Public Const BROOK                      As Long = 8
Public Const WQUEEN                     As Long = 9
Public Const BQUEEN                     As Long = 10
Public Const WKING                      As Long = 11
Public Const BKING                      As Long = 12
Public Const NO_PIECE                   As Long = 13     ' empty field
' skip 14, WEP-Piece needs bit 1 set for white color logic
Public Const WEP_PIECE                  As Long = 15    ' en passant dummy piece white
Public Const BEP_PIECE                  As Long = 16    ' en passant dummy piece black

Public Const ENPASSANT_WMOVE            As Long = 1     ' white pawn moves 2 rows > creates WEP_PIECE
Public Const ENPASSANT_BMOVE            As Long = 2     ' black pawn moves 2 rows > creates BEP_PIECE
Public Const ENPASSANT_CAPTURE          As Long = 3     ' en passant captures dummy piece WEP_PIECE or BEP_PIECE

'--- start positions
Public Const WKING_START                As Long = 25
Public Const BKING_START                As Long = 95
'--- Piece color ( (piece AND 1 )= WCOL => bit 1 set = White)
Public Const WCOL                       As Long = 1
Public Const BCOL                       As Long = 0
'--- Squares on board
Public Const SQ_A1                      As Long = 21, SQ_B1 As Long = 22, SQ_C1 As Long = 23, SQ_D1 As Long = 24, SQ_E1 As Long = 25, SQ_F1 As Long = 26, SQ_G1 As Long = 27, SQ_H1 As Long = 28
Public Const SQ_A2                      As Long = 31, SQ_B2 As Long = 32, SQ_C2 As Long = 33, SQ_D2 As Long = 34, SQ_E2 As Long = 35, SQ_F2 As Long = 36, SQ_G2 As Long = 37, SQ_H2 As Long = 38
Public Const SQ_A3                      As Long = 41, SQ_B3 As Long = 42, SQ_C3 As Long = 43, SQ_D3 As Long = 44, SQ_E3 As Long = 45, SQ_F3 As Long = 46, SQ_G3 As Long = 47, SQ_H3 As Long = 48
Public Const SQ_A4                      As Long = 51, SQ_B4 As Long = 52, SQ_C4 As Long = 53, SQ_D4 As Long = 54, SQ_E4 As Long = 55, SQ_F4 As Long = 56, SQ_G4 As Long = 57, SQ_H4 As Long = 58
Public Const SQ_A5                      As Long = 61, SQ_B5 As Long = 62, SQ_C5 As Long = 63, SQ_D5 As Long = 64, SQ_E5 As Long = 65, SQ_F5 As Long = 66, SQ_G5 As Long = 67, SQ_H5 As Long = 68
Public Const SQ_A6                      As Long = 71, SQ_B6 As Long = 72, SQ_C6 As Long = 73, SQ_D6 As Long = 74, SQ_E6 As Long = 75, SQ_F6 As Long = 76, SQ_G6 As Long = 77, SQ_H6 As Long = 78
Public Const SQ_A7                      As Long = 81, SQ_B7 As Long = 82, SQ_C7 As Long = 83, SQ_D7 As Long = 84, SQ_E7 As Long = 85, SQ_F7 As Long = 86, SQ_G7 As Long = 87, SQ_H7 As Long = 88
Public Const SQ_A8                      As Long = 91, SQ_B8 As Long = 92, SQ_C8 As Long = 93, SQ_D8 As Long = 94, SQ_E8 As Long = 95, SQ_F8 As Long = 96, SQ_G8 As Long = 97, SQ_H8 As Long = 98
'--- Move directions
Public Const SQ_UP                      As Long = 10
Public Const SQ_DOWN                    As Long = -10
Public Const SQ_RIGHT                   As Long = 1
Public Const SQ_LEFT                    As Long = -1
Public Const SQ_UP_RIGHT                As Long = 11
Public Const SQ_UP_LEFT                 As Long = 9
Public Const SQ_DOWN_RIGHT              As Long = -9
Public Const SQ_DOWN_LEFT               As Long = -11
'--- Files A-H
Public Const FILE_A                     As Long = 1, FILE_B As Long = 2, FILE_C As Long = 3, FILE_D As Long = 4, FILE_E As Long = 5, FILE_F As Long = 6, FILE_G As Long = 7, FILE_H As Long = 8

'--- Score values
```

```
Public Const MATE0                     As Long = 100000
Public Const MATE_IN_MAX_PLY           As Long = 100000 - 1000
Public Const VALUE_INFINITE            As Long = 111111
Public Const VALUE_NONE                As Long = 333333
Public Const VALUE_KNOWN_WIN           As Long = 10000

'-------------------------------------------------
' Array dimensions
'-------------------------------------------------
Public Const MAX_BOARD                 As Long = 119
Public Const MAX_MOVES                 As Long = 250 ' max moves for a position
Public Const MAX_GAME_MOVES            As Long = 999
Public Const MAX_PV                    As Long = 255
Public Const LIGHTNING_DEPTH           As Long = 3
Public Const MAX_DEPTH                 As Long = 150
Public Const NO_FIXED_DEPTH            As Long = 1000
Public Const PV_NODE                   As Boolean = True
Public Const NON_PV_NODE               As Boolean = False
Public Const QS_CHECKS                 As Boolean = True
Public Const QS_NO_CHECKS              As Boolean = False
Public Const GENERATE_ALL_MOVES        As Boolean = False

'-- Depth constants
Public Const DEPTH_ZERO                As Long = 0
Public Const DEPTH_QS_CHECKS           As Long = 0
Public Const DEPTH_QS_NO_CHECKS        As Long = -1
Public Const DEPTH_QS_RECAPTURES       As Long = -5
Public Const DEPTH_NONE                As Long = -6
Public Const DEPTH_MAX                 As Long = 50

'--- Move ordering value groups
Public Const MOVE_ORDER_QUIETS         As Long = -30000
Public Const MOVE_ORDER_BAD_CAPTURES As Long = -60000

'-------------------------------------------------
' Structure types
'-------------------------------------------------
Public Type TMOVE
  From             As Integer
  Target           As Integer
  Piece            As Integer
  Captured         As Integer
  EnPassant        As Integer
  Promoted         As Integer
  Castle           As Integer ' enumCastleFlag
  CapturedNumber   As Integer
  OrderValue       As Long
  SeeValue         As Long
  IsLegal          As Boolean
  IsChecking       As Boolean
End Type

Public Type TMatchInfo ' for future use in GUI
  EngRating   As Long
  Opponent    As String
  OppRating   As Long
  OppComputer As Boolean
End Type

Public Enum enumColor
  COL_WHITE = 1
  COL_BLACK = 0
  COL_NOPIECE = -1
End Enum

Public Enum enumPieceType
  NO_PIECE_TYPE = 0
  PT_PAWN = 1
```

```vb
  PT_KNIGHT = 2
  PT_BISHOP = 3
  PT_ROOK = 4
  PT_QUEEN = 5
  PT_KING = 6
  ALL_PIECES = 7
  PIECE_TYPE_NB = 8
End Enum

Public Type TMovePicker ' data fields for move picker logic
  CurrMoveNum As Long
  EndMoves As Long
  BestMove As TMOVE
  bBestMoveChecked As Boolean
  bBestMoveDone As Boolean  ' Moves are not generated before BestMove was tried
  PrevMove As TMOVE
  ThreatMove As TMOVE
  LegalMovesOutOfCheck As Long
  bMovesGenerated As Boolean
  bCapturesOnly As Boolean ' for QSearch
  GenerateQSChecksCnt As Long ' number of ply in QSearch where checks are generated
End Type

Public Type TScore ' final score = mg+eg scaled by game phase
  MG As Long ' Midgame score
  EG As Long ' Endgame score
End Type

Public Enum enumCastleFlag
  NO_CASTLE = 0
  WHITEOO = 1
  WHITEOOO = 2
  BLACKOO = 3
  BLACKOOO = 4
End Enum

Public Enum enumEndOfGame ' Game result
  NO_MATE = 0
  WHITE_WON = 1
  BLACK_WON = 2
  DRAW_RESULT = 3
  DRAW3REP_RESULT = 4
End Enum



Attribute VB_Name = "basDebug"
Option Explicit
'==================================================
'= basDebug:
'= Debug functions
'==================================================

Public TestStart As Long, TestEnd As Long

Public Function DEGUBPrintMoveList(MoveList() As TMOVE) As String
  Dim i         As Long
  Dim strMoves As String

  Do While Not MoveList(i).From = 0
    strMoves = strMoves & vbTab & MoveText(MoveList(i))
    i = i + 1
    If i Mod 3 = 0 Then strMoves = strMoves & vbCrLf
  Loop

  DEGUBPrintMoveList = strMoves
End Function

```

```
Public Sub DEBUGPerfTestSearch(ByVal iDepth As Long)
  Dim NumMoves As Long
  Dim i        As Long
  If iDepth = 0 Then Exit Sub
  Ply = Ply + 1
  GenerateMoves Ply, False, NumMoves

  For i = 0 To NumMoves - 1
    MakeMove Moves(Ply, i)
    If CheckLegal(Moves(Ply, i)) Then
      Nodes = Nodes + 1
      DEBUGPerfTestSearch iDepth - 1
    End If
    UnmakeMove Moves(Ply, i)
  Next

  Ply = Ply - 1
End Sub

Public Function DEBUGPerfTest(ByVal iDepth As Long) As String
  Dim strResult As String, StartTime As Single, EndTime As Single
  InitGame
  Ply = 1
  bWhiteToMove = True
  Nodes = 0
  StartTime = Timer
  DEBUGPerfTestSearch iDepth
  EndTime = Timer
  ' time for move generation
  strResult = "time: " & Format$(EndTime - StartTime, "0.00") & " nodes: "

  ' show correct move counts until depth 5
  Select Case iDepth
    Case 1
      strResult = strResult & Nodes & " (expected: 20)"
    Case 2
      strResult = strResult & Nodes - 20 & " (expected: 400)"
    Case 3
      strResult = strResult & Nodes - 400 - 20 & " (expected: 8902)"
    Case 4
      strResult = strResult & Nodes - 8902 - 400 - 20 & " (expected: 197281)"
    Case 5
      strResult = strResult & Nodes - 197281 - 8902 - 400 - 20 & " (expected: 
      4865609)"
  End Select

  DEBUGPerfTest = strResult
End Function

Public Sub DEBUGBench(ByVal iDepth As Long)
  ' ORIGINAL
  Dim i         As Long, StartTime As Single, EndTime As Single, x As Long, c As Long,
   s As String
  Dim arTime(2) As Single, EPD(20) As String
  '--- Test positions -----
  'EPD(1) = "r1b1kb1r/pppp1ppp/2n1pq2/8/2PP4/P1P2N2/4PPPP/R1BQKB1R w KQkq - 1 7 " ' SF6 problem: Too
  high eval until ply 7
  ' EPD(1) = "rn1q4/pbp2kp1/1p1ppn2/8/1PP5/P5Q1/3PPP1r/R1B1KBR1 b Q b3 0 11" ' too high KSafety eval
  'EPD(1) = "3r2k1/p1q1r2p/bppb2p1/6Qn/2NPp3/1PN1Pp1P/PB3PP1/2R3RK b - - 3 27 " '  King attack eval too high
  <<<
  ' EPD(1) = "r3k3/p2nbpp1/bpp1p3/3nP3/2NP3P/1PB4P/P1Q2PBq/R3RK2 w q - 1 20 " ' KS eval
  'EPD(1) = "r4r2/p1q1n1kp/2n1ppp1/8/3P2N1/3BPP2/2Q2P1P/R3K2R w KQ - 0 19 " ' Trapped knight h3/h4
  'EPD(1) = "r4rk1/1p2ppbp/pq1p2p1/3P4/1nP3n1/2N2N2/PP2QPPP/R1B2RK1 b - - 0 18 " ' Trapped knight a5
  'EPD(1) = "rnbq1rk1/ppp2pp1/8/2npP2Q/1P6/8/P1PN1PPP/R1B2RK1 b - b3 0 11"
  'EPD(1) = "rnbq1r2/ppp2ppk/8/2npP2Q/8/8/PPPN1PPP/R1B2RK1 b - - 1 10 "
  'EPD(1) = "3r1r1k/1b2b1p1/1p5p/2p1Pp2/q1B2P2/4P2P/1BR1Q2K/6R1 b - - 0 1 " ' Eval BEnch
  'EPD(1) = "8/p6p/4k1p1/3p4/2p4P/Pr3PK1/R5P1/8 b - - 1 41 " ' Passed Pawn eval
  'EPD(1) = "r2qr1k1/p3bppp/bpn2n2/2pp4/3P1B2/1PN2NP1/P3PPBP/2RQ1RK1 w - - 8 1 " ' SEE problem
```

```
    'EPD(1) = "r1b1k2r/1pp1q2p/p1n3p1/3QPp2/8/1BP3B1/P5PP/3R1RK1 w kq - 0 1 " ' WAC133
    'EPD(1) = "r3kbr1/1p3p1b/pq4Pp/3pp1n1/3PP1N1/PQ4pP/1P3P1B/R3KBR1 w Qq - 0 1  " 'Eval Test symmetric 2

    'EPD(1) = "rnbqkbnr/1pp2pp1/p6p/3pp3/3PP3/P6P/1PP2PP1/RNBQKBNR w KQkq - 0 1 " ' Eval Test symmetric 1
    'EPD(1) = "8/5K2/8/3N4/8/8/7k/8 w - - 0 4" 'endgame test
    'EPD(1) = "8/6R1/8/4k1K1/8/8/3r4/8 w - - 3 3 " ' draw test
    'EPD(1) = "r1bq3r/ppppR1p1/5n1k/3P4/6pP/3Q4/PP1N1PP1/5K1R w - - 0 1 " ' WAC138
    'EPD(1) = "8/7p/7k/8/1PK5/8/8/8 w - - 0 1  " ' endgame  pawn promote
    'EPD(1) = "8/8/8/6p1/6Pp/5k2/7K w - - 2 95 "  ' bug hanging movepicker => one legal move out of check
    'EPD(1) = "r2qk2r/pp1n1ppp/2p1p3/5b2/P2Pn3/BBP1P3/3N1PPP/R3QRK1 w kq - 0 14 " ' Eval ?
    'EPD(1) = "2r5/7K/k5P1/8/8/1p6/8/8 b - - 0 1 " ' Passed pawn test
    'EPD(1) = "3R4/p6r/8/1P2k3/2B5/8/4K3/8 w - - 50 103  " ' endgame king to pawn1p2PP2
    'EPD(1) = "r1b2rk1/p4ppp/1p1Qp3/4P2N/1P6/8/P3qPPP/3R1RK1 w - - 0 1 " ' WAC 288
    'EPD(1) = "8/8/8/Q7/8/2K3k1/7r/8 w - - 0 1 " ' KQKR
    'EPD(1) = "8/8/8/Q7/8/2K3k1/7p/8 w - - 0 1 " ' KQKP
    'EPD(1) = "8/8/8/5pk1/8/2KR4/8/8 w - - 0 1" ' KRKP
    'EPD(1) = "2qrr1n1/3b1kp1/2pBpn1p//p2P4/1BP5/P3Q1PP/4RRK1 w - - 0 1" ' ; e2h5 "BWTC.0031"
    ' EPD(1) = "5rk1/1pp3bp/3p2p1/2PPp3/1P2P3/2Q1B3/4q1PP/R5K1 b - - bm Bh6; id WAC.169"
    'EPD(1) = "8/7p/1R4pk/8/6PK/7P/1p6/1r6 b - - 3 1 " ' Passed pawn attacked by rook   SF6: mg:1.14 eg:2.24 cp
    'EPD(1) = "8/7p/1R4pk/8/6PK/7P/1pr5/8 b - - 0 1  " ' Passed pawn attacked by rook, blocked by own rook  SF6: 1.38 2.36
    'EPD(1) = "8/7p/3R2pk/8/1r4PK/7P/1p6/8 w - - 0 1 " ' Passed pawn defended by rook  SF6: 2.54  3.97
    ' EPD(1) = "r3r1k1/pbq2p2/4p2p/1p1nP2Q/2pR4/2P5/PPB2PPP/4R1K1 w - - 0 20 " ' Defend
    'EPD(1) = "r3r1k1/pbq2pp1/4p2B/1p1nP2Q/2pR4/2P5/PPB2PPP/4R1K1 b - - 0 19 " ' Attack f7f5 (g7xh6 bad)
    'EPD(1) = "r1bqkbnr/ppp2ppp/2np4/4p3/2B1P3/5N2/PPPP1PPP/RNBQK2R w KQkq - 2 4 " ' KSafety/Castle eval
    'EPD(1) = "rnbq1rkr/pppp1p1p/5n2/2b1p3/4P3/2NP4/PPP2PPP/R1BQKBNR b KQ - 2 1 " ' KSafety/Castle eval-Black
    ' EPD(1) = "6k1/6p1/8/8/8/4P2P/6K1 b - -" ' Test Endgame Tablebase acces in search for root
    'EPD(1) = "8/6k1/6p1/8/7r/3P1KP1/8/8 w - - 0 1 "  ' Test Endgame Tablebase acces in search for ply=1
    ' EPD(1) = "r3k2r/pb3pbp/2p1p3/1q2p3/2p5/6P1/1PQ1PPBP/R1BR2K1 w kq - 0 2 "
    'EPD(1) = "2r1r1k1/4bp1p/p2pp1pP/q3n1P1/Np1Nb3/1P2B3/P1PQ4/1K2RBR1 b - - 1 21 " ' e5f3 not found
    ' EPD(1) = "2r1r1k1/4bp1p/p2pp1pP/q5P1/Np2b3/1P2BN2/P1PQ4/1K2RBR1 b - - 0 21 "
    '   EPD(1) = "4r1k1/4bp1p/p2pp1pP/q5P1/Np2b3/1P2BN2/P1rQ4/1K2RBR1 w - - 0 22"  ' d2xc2 ok, d2d4 >Rc2c4 illegal move, IsCHecking no detected
    'EPD(1) = "r2r2k1/pb3p1p/1qn1p2Q/5p2/1p1P4/1NPB4/P4PPP/2R1R1K1 b - - 0 22 " ' KSafety test
    ' EPD(1) = "8/5pk1/1p4Pp/q6P/Q7/1P6/8/6K1 b - - 0 1 " ' ShelterStorm test
    'EPD(1) = "5k2/6b1/8/4N3/8/8/3P1K2/8 w - - 3 1  " ' Scale factor 1 pawn test
    ' EPD(1) = "r1b2r1k/p5pp/2nq4/Ppp1pp2/2Bn1N1Q/2B1R3/2P2PPP/R5K1 w - b6 0 2 " ' EnPassant test
    ' EPD(1) = "r1b2r1k/pp4pp/2nq4/P1p1pp2/2Bn1N1Q/2B1R3/2P2PPP/R5K1 b - - 1 1" ' EnPassant test2 move b7b5
    ' EPD(1) = "6k1/4Q1p1/7p/8/nn6/1p3R2/5PPP/6K1 w - - 1 1   " ' mate threat
    ' EPD(1) = "8/2pp4/3kPKP1/3P4/8/8/8/8 w - - 0 1 "
    'EPD(1) = "8/8/2k5/8/5K2/3R4/8/3qR3 w - - 0 1"  ' EGTB
    'EPD(1) = "8/3PK3/8/5p1k/8/8/8/8 b - - 0 w " ' EGTB test promotion
    'EPD(1) = "8/5PK1/8/2Q5/4P1k1/8/8/8 b - - 0 14 "
    'EPD(1) = "8/4k3/8/8/5P2/5K2/8/8 b - - 4 3 "  ' EGTB KPK
    'EPD(1) = "8/3k3K/7P/1r6/5p2/8/8/8 b - - 0 1 "
    'EPD(1) = "8/8/3R4/p3npk1/P3p2p/4P3/3K1PP1/r2B4 w - - 8 39 " ' EP capture mate bug
    'EPD(1) = "8/2b5/8/4kN2/1r4K1/6N1/8/8 w - - 0 1"  ' endgame scale factor no pawns
    'EPD(1) = "8/8/7k/p1P4p/P6P/7K/8/8 w - - 0 1"  ' passed pawn test 1 rank 5
    'EPD(1) = "8/8/7k/p1P4p/P6P/7K/8/2R5 w - - 0 1"  ' passed pawn test 2 defended from behind
    'EPD(1) = "8/8/7k/p1P4p/P6P/7K/8/2r5 w - - 0 1"  ' passed pawn test 3 attacked from behind
    'EPD(1) = "8/7r/7k/p1P4p/P6P/7K/8/2R5 w - - 0 1"  ' passed pawn test 4 defended from behind + attacked path
    'EPD(1) = "8/8/2P4k/p6p/P6P/7K/8/2R5 w - - 0 1"  ' passed pawn test 5 defended from behind rank 6
    'EPD(1) = "7k/5K1p/7P/8/8/8/8/8 b - - 1 1" ' no move draw
   ' EPD(1) = "r5k1/pp4pp/2pb3r/3p2q1/P1PP1nB1/1PB1P1PP/7K/R2Q2R1 b - - 0 27" ' KSafety

    'EPD(1) = "5rk1/pp4pp/2pb3r/3p2q1/P1PP4/1PB1P1PB/7K/R4QR1 b - - 2 29" '
    'EPD(1) = "6k1/4b1p1/5p1Q/1p2pP2/4P3/1P6/6PP/6rK w - - 0 41 " 'only one legalmove
    'EPD(1) = " /4b1p1/8/1p2p3/8/7p/5p2/6bK w - - 0 1 " 'no legal move

    'EPD(1) = "8/8/7k/8/8/6PP/3r3K w - - 1 1  " ' mated result : bestmove (none)

    'EPD(1) = "r5k1/pp4pp/2pb3r/3p2q1/P1PP4/1PB1P1PB/7K/R2Q2R1 b - - 0 28 "
     'EPD(1) = "6r1/2pq2pk/1p3p1p/1P1Pp2P/Q3P1P1/p1R3K1/P7/8 w - - 98 109 " ' fifty
    ' EPD(1) = "k7/8/P7/1K6/8/8/8/8 w - - 12 1 " ' endgame kpk"
   '  EPD(1) = "r1bq3r/1p1nbpk1/p2p1np1/P1pPp3/4P2p/2NBB2P/1PPQNPP1/R4RK1 b - - 1 14 " ' Tactic

    ' EPD(1) = "4kb1r/1pqb1ppp/p3p3/3pP3/2r2P2/2NQB3/PPP3PP/R4RK1 w k - 6 14" ' a2a3 lost
```

```
  '  EPD(1) = "4kb1r/1pqb1ppp/p3p3/3pP3/2r2P2/P1NQB3/1PP3PP/R4RK1 b k - 0 14 " ' d5d4 wins
  '  EPD(1) = "4kb1r/1pqb1ppp/p3p3/4P3/2rB1P2/P1NQ4/1PP3PP/R4RK1 b k - 0 15 " ' Txd4 wins
    ' EPD(1) = "4kb1r/1pqb1ppp/p3p3/4P3/3r1P2/P1NQ4/1PP3PP/R4RK1 w k - 0 16 " ' d3d4 lost
  ' EPD(1) = "4kb1r/1pqb1ppp/p3p3/4P3/2rp1P2/P1NQB3/1PP3PP/R4RK1 w k - 0 15 " ' e3xd4 lost

'  EPD(1) = "8/6k1/8/5P1P/6PK/3n4/8/8 w - - 0 81 "
'  EPD(1) = "r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 1" 'Qsearch test
'  EPD(1) = "1r4Bk/PPPPPp1P/8/7R/R6K/8/ppppppQ1/6B1 w - - 0 1 " ' QS
    '------ normal test ----
    EPD(1) = "1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - 0 1"
  ' EPD(1) = "4k3/6KP/8/8/6r1/8/7p/8 w - -" ' Endgame tablesbase test
  ' EPD(1) = "4k2K/7P/8/8/6r1/8/7p/8 b - - 1 1" 'EGTB

  'EPD(1) = "r3k2r/pp2pp1p/8/q2Pb3/2P5/4p3/B1Q2PPP/2R2RK1 w kq - bm c5;" ' Bug depth 100
  ' EPD(1) = "8/7p/5P1k/1p5P/5p2/2p1p3/P1P1P1P1/1K3Nb1 w - - bm Ng3" ' Bug Depth
  'EPD(1) = "3kB3/5K2/7p/3p4/3pn3/4NN2/8/1b4B1 w - - " ' crash max_ply
' EPD(1) = "8/8/8/4k3/3r2p1/3P4/3K4/8 b - - 0 7" ' EGTB

'EPD(1) = "1r1r4/1p4k1/p1b2R2/P1N2p1p/1P3QpP/3p2P1/5P1K/8 b - - 0 67" ' Draw bug
'EPD(1) = "1r1r4/1p6/p1b4Q/P1N1kp1p/1P4pP/3p2P1/5P1K/8 w - - 6 71 ' Draw fehler vor Fehlzug h6f4 -149305158
-1177636898"
' EPD(1) = "1r1r4/1p6/p1b5/P1N1kp1p/1P3QpP/3p2P1/5P1K/8 b - - 7 71" 'nach Fehlzug h6f4          -1641286943
1195148230
'EPD(1) = "1r1r4/1p6/p1b2k2/P1N2p1p/1P3QpP/3p2P1/5P1K/8 w - - 8 72" ' Draw fehler endstellung 3x
  '  EPD(1) = "8/8/8/8/4kp2/1R6/P2q1PPK/8 w - - 0 1"
  '  EPD(1) = "1r3k2/5pbQ/3q4/3PpBP1/5P2/p1B5/P1p5/1KR5 w - - 0 41" ' Kb1a1 bad

  'EPD(1) = "r1b3k1/p2p1nP1/2pqr3/1p2p1QP/2B1Pn2/1P6/P1PP4/1K4R1 w - - 2 3" ' g5d8
  '##########################################################

    'EPD(2) = "1rb2rk1/p3nppp/1p1qp3/3n2N1/2pP4/2P3P1/PP3PBP/R1BQR11K w - -"  'TEST 2
    'EPD(2) = "2k4B/bpp1qp2/p1b5/7p/1PN1n1p1/2Pr4/P5PP/R3QR1K b - - " ' g3g4 ; Ng3; e1d3not h2xg3! <<<<<<
    EPD(2) = "r1bqk1r1/1p1p1n2/p1n2pN1/2p1b2Q/2P1Pp2/1PN5/PB4PP/R4RK1 w q - - " ' f1xf4
    SF-Eval -0.5
    EPD(3) = "r1b2rk1/p2nq1p1/1pp1p2p/5p2/2PPp3/2Q1P3/PP1N1PPP/2R1KB1R w K - 0 13" '---
    quiet
    EPD(4) = "6k1/p1r5/4b1p1/R1pprp1p/7P/1P1BP3/P1P3P1/4R1K1 w - - 4 25" ' no advantage
    EPD(5) = "8/8/2R5/1p2qp1k/1P2r3/2PQ2P1/5K2/8 w - - 0 1" ' Endgame
    EPD(6) = "r7/pbk5/1pp5/4n1q1/2P5/1P6/P4BBQ/4R1K1 b - - 0 33" '
    EPD(7) = "r1bqk2r/p2p1pp1/1p2pn1p/n1pP2B1/1bP5/2N2N2/PPQ1PPPP/R3KB1R w KQkq - 0 9"
    '<<<<< AKT
    EPD(8) = "r3qb1k/1b4p1/p2pr2p/3n4/Pnp1N1N1/6RP/1B3PP1/1B1QR1K1 w - - 0 1" ' Nxh6
    SF-Eval +1,4
    '-----------------------------------------------------------------------------------
    DebugMode = True

    'iDepth = 8
    ' ReadGame "Drawbug2.txt"
    'bForceMode = False

'
'' setup UCI game string, see ARENA GUI protocol trace window
''--- 3x draw problem g8g7 d8f6
''UCIPositionSetup "position fen r1bqk2r/pp1nbppp/2p1p3/3n4/4N3/3P1NP1/PPP1QPBP/R1B1K2R w KQkq - 0 1
moves e1g1 e8g8 c2c4 d5f6 e4c3 e6e5 f3e5 d7e5 e2e5 f8e8 d3d4 e7b4 e5f4 b4c3 b2c3 c8e6 f1e1 e6c4 e1e8 d8e8
c1d2 e8d7 a2a4 f6d5 f4e4 a8e8 e4c2 d5f6 c2b2 g7g6 a4a5 f6g4 g2f3 g4f6 b2b4 d7f5 f3g2 f5d3 d2e1 d3e2 h2h3
a7a6 b4b1 e8e7 b1d1 e2e6 d1d2 g8g7 d2f4 c4d5 f2f3 e6c8 g3g4 d5b3 f4d6 f6d5 d6a3 b3c4 a3c5 e7e2 g2f1 c8e8
f1e2 e8e2 c5d6 e2f3 g4g5 f3f1 g1h2 h7h6 g5h6 g7h7 d6e5 f7f6 e5e4 d5f4 e4e7 h7h6 e7f8 h6h7 f8e7 h7g8 e7d8 g8f7
d8d7 f7f8 d7d8 f8g7 d8e7 c4f7 e7f6 g7g8 f6d8 g8g7 d8f6 g7g8 f6d8"
''UCIPositionSetup "position fen r1bq1rk1/ppp2ppp/5n2/2bp4/2NPP3/2P5/PP3PPP/RNBQK2R w KQ - 0 1 moves
c4e3 c5e7 e4e5 f6e4 f2f3 e4g5 f3f4 g5e4 b1d2 b7b6 f4f5 c8a6 d1f3 a6b7 d2e4 d5e4 f3g4 g8h8 e1f2 f8e8 h1d1 a7a5
f2g1 a5a4 a2a3 e7f8 g4f4 b6b5 a1b1 f7f6 e5e6 d8d6 f4g4 a8d8 b2b3 a4b3 b1b3 d6a6 b3b1 b7a8 g4e2 d8b8 e2f2
e8d8 e3c2 a6c6 c1d2 c6c4 c2e3 c4a4 d2e1 a4a3 f2e2 a3a6 e1g3 a6a5 b1a1 a5b6 g3f4 a8c6 d1b1 b8a8 a1a8 d8a8
h2h4 f8e7 h4h5 h7h6 e2c2 a8a7 c2d2 b6b8 c3c4 b5b4 c4c5 b8d8 d2b4 a7b7 b4b7 c6b7 b1b7 d8d4 b7c7 e7c5 e6e7
c5e7 c7e7 h8h7 e7c7 d4d2 g1f1 d2d3 f1e1 d3b5 e1f2 b5b2 c7c2 b2d4 f2g3 d4b4 f4c7 b4e1 g3f4 e1h4 g2g4 h4h1
c2f2 h1g1 f2g2 g1a1 c7b8 a1b1 b8d6 b1d3 d6c7 d3b1 g2c2 b1h1"
'UCIPositionSetup "position fen r1bq1rk1/ppp2ppp/5n2/2bp4/2NPP3/2P5/PP3PPP/RNBQK2R w KQ - 0 1 moves
c4e3 c5e7 e4e5 f6e4 f2f3 e4g5 f3f4 g5e4 b1d2 b7b6 f4f5 c8a6 d1f3 a6b7 d2e4 d5e4 f3g4 g8h8 e1f2 f8e8 h1d1 a7a5
```

```
f2g1 a5a4 a2a3 e7f8 g4f4 b6b5 a1b1 f7f6 e5e6 d8d6 f4g4 a8d8 b2b3 a4b3 b1b3 d6a6 b3b1 b7a8 g4e2 d8b8 e2f2
e8d8 e3c2 a6c6 c1d2 c6c4 c2e3 c4a4 d2e1 a4a3 f2e2 a3a6 e1g3 a6a5 b1a1 a5b6 g3f4 a8c6 d1b1 b8a8 a1a8 d8a8
h2h4 f8e7 h4h5 h7h6 e2c2 a8a7 c2d2 b6b8 c3c4 b5b4 c4c5 b8d8 d2b4 a7b7 b4b7 c6b7 b1b7 d8d4 b7c7 e7c5 e6e7
c5e7 c7e7 h8h7 e7c7 d4d2 g1f1 d2d3 f1e1 d3b5 e1f2 b5b2 c7c2 b2d4 f2g3 d4b4 f4c7 b4e1 g3f4 e1h4 g2g4 h4h1
c2f2 h1g1 f2g2 g1a1 c7b8 a1b1 b8d6 b1d3 d6c7 d3b1 g2c2 b1h1 c2f2 h1g1 f2g2 g1b1 g2h2 b1a1"
'     FixedDepth = 15: MovesToTC = 0: TimeLeft = 20: TimeIncrement = 10: bPostMode = True:  bCompIsWhite =
bWhiteToMove
'     '--- start computing --------------
'     StartEngine
'     Stop
' End

'
  For x = TestStart To TestEnd

      For i = 0 To 0  ' number of time measure runs  > 1x
        'For i = 0 To 2 ' number of time measure runs > 3x
        InitGame  ' Reset FixedDepth , Hash, History...
        ReadEPD EPD(x)  ' Reset FixedDepth

'##########

'ParseCommand "g7f6"
'Debug.Print Fifty
'ParseCommand "f4g5"
'Debug.Print Fifty
'ParseCommand "f6e5"
'Debug.Print Fifty

'ParseCommand "g5f4"
'Debug.Print Fifty
'ParseCommand "e5f6"
'Debug.Print Fifty

'ParseCommand "f4h6"
'ParseCommand "f6e5"
'
'ParseCommand "h6f4"
'ParseCommand "e5f6"


        If True Then
          If x = 3 Or x = 4 Or x = 5 Or x = 7 Then
            FixedDepth = iDepth + 1
          Else
            FixedDepth = iDepth
          End If
        ' Else
        ' FixedTime = 4
         MovesToTC = 0: TimeLeft = 20: TimeIncrement = 10
        End If
        If InStr(EPD(x), " w") > 0 Then
          bCompIsWhite = True    'False:
          bWhiteToMove = True    '---False
        Else
          bCompIsWhite = False ' True  'False:
          bWhiteToMove = False ' True  '---False
        End If
        ' ParseCommand "b7b5"
        bPostMode = True
        '   bPostMode = False
        'SendCommand PrintPos
        If False Then  ' Time based end of thinking
          FixedDepth = NO_FIXED_DEPTH
          LevelMovesToTC = 40
          MovesToTC = 0
          TimeLeft = 120
          TimeIncrement = 0
          GameMovesCnt = 119 ' plies, /2 for MoveCnt
        End If

        StartTime = Timer
        '--- start computing --------------
        StartEngine
        '
        EndTime = Timer

        arTime(i) = EndTime - StartTime
        If arTime(i) = 0 Then arTime(i) = 1
        bPostMode = True
        SendCommand vbCrLf & "time: " & Format$(arTime(i), "0.000") & " nod: " & Nodes &
         " qn: " & QNodes & "(DMax:" & QSDepthMax & ")" & " ev:" & EvalCnt & " sc: " &
        FinalScore & " EGTB:" & EGTBasesHitsCnt & " Ply:" & MaxPly & " " & s & vbCrLf

           ' Test Counter
        s = ""
        For c = 1 To 19
          If TestCnt(c) <> 0 Then s = s & CStr(c) & ":" & TestCnt(c) & ","
        Next c
        If s <> "" Then SendCommand "Counter: " & s
      Next

      If arTime(0) < arTime(1) Then
        If arTime(0) < arTime(2) Then
          i = 0
        Else
          i = 2
        End If
      ElseIf arTime(1) < arTime(2) Then
        i = 1
      Else
        i = 2
      End If
      ' count 3x
      If arTime(i) > 0 Then
        SendCommand "best time: " & Format$(arTime(i), "0.000") & " nps: " & Int(Nodes /
         arTime(i))
      Else
        SendCommand "best time: " & Format$(0, "0.000") & " nps: " & Nodes
      End If
      SendCommand "Hash usage:" & Format((CDbl(HashUsage) / CDbl(HashSize)) * 100#,
      "0.00")
      SendCommand "--------------------"
    Next x

  End Sub

  Public Sub WriteDebug(s As String)
    Debug.Print s
  End Sub

  Public Sub DMoves()
    ' Debug: print current move line
    Dim i As Long, s As String
    s = CStr(RootDepth) & "/" & CStr(Ply) & ">"

    For i = 1 To Ply - 1
      s = s & CStr(i) & ":" & MoveText(MovesList(i)) & "/"
    Next

    Debug.Print s
    DoEvents
  End Sub

  Public Function SearchMovesList() As String
    ' print current move line
    Dim i As Long, s As String
    s = ""

    For i = 1 To Ply - 1
      s = s & MoveText(MovesList(i)) & " "
    Next
    s = Trim(s)
    SearchMovesList = s
  End Function


  Public Sub DEBUGLoadGame(ByVal iDepth As Long)
    ' ORIGINAL
    Dim i          As Long, StartTime As Single, EndTime As Single, x As Long, c As Long,
     s As String
    Dim arTime(2) As Single
    iDepth = 8
    DEBUGReadGame "bug001.txt"
    bForceMode = False

    For i = 0 To 0 ' number of time measure runs > 1x
      FixedDepth = iDepth
      bCompIsWhite = False ' True 'False:
      bWhiteToMove = False ' True '---False
      bPostMode = True
      StartTime = Timer
      StartEngine
      EndTime = Timer
      arTime(i) = EndTime - StartTime
      If arTime(i) = 0 Then arTime(i) = 1
      bPostMode = True
      SendCommand vbCrLf & "time: " & Format$(arTime(i), "0.000") & " nod: " & Nodes &
      " qn: " & QNodes & " ev:" & EvalCnt & " sc: " & EvalSFTo100(FinalScore) & " Ply:"
      & MaxPly & " " & s & vbCrLf
    Next

    SendCommand "--------------------"
  End Sub

  Public Sub DEBUGReadGame(sFile As String)
    ' Read PGN File
    Dim h              As Long, s As String, m As Long, sInp As String, m1 As String, m2
    As String
    InitGame
    bForceMode = True
    h = 10 'FreeFile()
    Open sFile For Input As #h

    Do Until EOF(h)
      Line Input #h, sInp
      sInp = Trim(sInp) & "   "
      s = Trim(sInp)
      'Debug.Print s
      m1 = Trim(Left(s, 4))
      If Len(m1) = 4 Then
        ParseCommand m1 & vbLf
      End If
    Loop

    Close #h
  End Sub
  VERSION 5.00
  Begin VB.Form frmDebugMain
     Caption         =   "ChessBrainVB debug console"
     ClientHeight    =   9960
     ClientLeft      =   60
     ClientTop       =   345
     ClientWidth     =   14715
     Icon             =   "DebugMain.frx":0000
     LinkTopic        =   "Form1"
     ScaleHeight      =   9960
     ScaleWidth       =   14715
     StartUpPosition  =   2  'CenterScreen
     Begin VB.CommandButton cmdRunUCI
        Caption         =   "Calc UCI-Pos"
        Height          =   330
        Left            =   4200
        TabIndex        =   10
        Top             =   600
        Width           =   1305
     End
     Begin VB.TextBox txtUciPosition
        Height          =   285
        Left            =   5640
        TabIndex        =   9
        Text            =   $"DebugMain.frx":0442
        Top             =   600
        Width           =   8535
     End
     Begin VB.CommandButton cmdThink
        Caption         =   "Think"
        Height          =   330
        Left            =   1800
        TabIndex        =   8
        Top             =   600
        Width           =   1425
     End
     Begin VB.CommandButton cmdNewgame
        Caption         =   "New game"
        Height          =   330
        Left            =   120
        TabIndex        =   7
        Top             =   600
        Width           =   1425
     End
     Begin VB.CommandButton cmdTx
        Caption         =   "f1xf4"
        Height          =   330
        Left            =   13200
        TabIndex        =   6
        Top             =   240
        Width           =   945
     End
     Begin VB.CommandButton cmdT2
        Caption         =   "g4xh6"
        Height          =   330
        Left            =   11880
        TabIndex        =   5
        Top             =   240
        Width           =   945
     End
     Begin VB.CommandButton cmdTest1
        Caption         =   "e3e8+ M8"
        Height          =   330
        Left            =   10440
        TabIndex        =   4
        Top             =   240
        Width           =   1065
     End
     Begin VB.CommandButton cmdFakeInput
        Caption         =   "Send"
        Height          =   330
        Left            =   9000
        TabIndex        =   1
        Top             =   240
        Width           =   1065
     End
     Begin VB.ComboBox cboFakeInput
        Height          =   315
        Left            =   105
        TabIndex        =   0
        Top             =   240
        Width           =   8796
     End
     Begin VB.TextBox txtIO
        BackColor       =   &H00E0E0E0&
        BeginProperty Font
           Name            =   "Courier New"
           Size            =   8.25
           Charset         =   0
           Weight          =   400
           Underline       =   0   'False
           Italic          =   0   'False
           Strikethrough   =   0   'False
        EndProperty
        ForeColor       =   &H00FF0000&
        Height          =   8892
        Left            =   120
        Locked          =   -1  'True
        MultiLine       =   -1  'True
        ScrollBars      =   3  'Both
        TabIndex        =   2
        TabStop         =   0   'False
        Top             =   960
        Width           =   14310
     End
     Begin VB.Label lblDescr
        BackStyle       =   0  'Transparent
        Caption         =   "Input"
        Height          =   195
        Index           =   2
        Left            =   120
        TabIndex        =   3
        Top             =   0
        Width           =   1335
     End
  End
  Attribute VB_Name = "frmDebugMain"
  Attribute VB_GlobalNameSpace = False
  Attribute VB_Creatable = False
  Attribute VB_PredeclaredId = True
  Attribute VB_Exposed = False
  '==================================================
  '= frmDebugMain:
  '= debug form
  '==================================================
  Option Explicit


  Private Sub cboFakeInput_KeyPress(KeyAscii As Integer)
    If KeyAscii = 13 Then cmdFakeInput_Click
  End Sub

  Private Sub cmdFakeInput_Click()
    FakeInput = cboFakeInput.Text & vbLf
    FakeInputState = True
    cboFakeInput.SelStart = 0
    cboFakeInput.SelLength = Len(cboFakeInput.Text)
    cboFakeInput.SetFocus

    UCIMode = False
    pbIsOfficeMode = True 'TEst
  End Sub

```

```
Private Sub cmdWb_Click()
  bPostMode = True
  ParseCommand "setboard r1b2rk1/pp1n2pp/2p1p3/2Pp4/1q1Pp3/4P1PN/PP2QPBP/2R2RK1 w - - 0 15" & vbLf
  ParseCommand "sd 10" & vbLf
  ParseCommand "go" & vbLf

End Sub

Private Sub cmdNewgame_Click()
 UCIMode = True
 cboFakeInput.Text = "ucinewgame"
 cmdFakeInput_Click
End Sub

Private Sub cmdRunUCI_Click()
UCIPositionSetup "position fen r1bqk2r/pp1nbppp/2p1p3/3n4/4N3/3P1NP1/PPP1QPBP/R1B1K2R w KQkq - 0 1 moves e1g1 e8g8 c2c4 d5f6 e4c3 e6e5 f3e5 d7e5 e2e5 f8e8 d3d4 e7b4 e5f4 b4c3 b2c3 c8e6 f1e1 e6c4 e1e8 d8e8 c1d2 e8d7 a2a4 f6d5 f4e4 a8e8 e4c2 d5f6 c2b2 g7g6 a4a5 f6g4 g2f3 g4f6 b2b4 d7f5 f3g2 f5d3 d2e1 d3e2 h2h3 a7a6 b4b1 e8e7 b1d1 e2e6 d1d2 g8g7 d2f4 c4d5 f2f3 e6c8 g3g4 d5b3 f4d6 f6d5 d6a3 b3c4 a3c5 e7e2 g2f1 c8e8 f1e2 e8e2 c5d6 e2f3 g4g5 f3f1 g1h2 h7h6 g5h6 g7h7 d6e5 f7f6 e5e4 d5f4 e4e7 h7h6 e7f8 h6h7 f8e7 h7g8 e7d8 g8f7 d8d7 f7f8 d7d8 f8g7 d8e7 c4f7 e7f6 g7g8 f6d8 g8g7 d8f6 g7g8 f6d8"
FixedDepth = 15: MovesToTC = 0: TimeLeft = 20: TimeIncrement = 10: bPostMode = True
'--- start computing --------------
StartEngine

End Sub

Private Sub cmdT2_Click()
  cboFakeInput.Text = "bench 21"
  TestStart = 8
  TestEnd = 8
End Sub

Private Sub cmdTest1_Click()
  cboFakeInput.Text = "bench 23"
  TestStart = 1
  TestEnd = 1
End Sub


Private Sub cmdThink_Click()
 cboFakeInput.Text = "go"
 cmdFakeInput_Click
End Sub

Private Sub cmdTx_Click()
  cboFakeInput.Text = "bench 21"
  TestStart = 2
  TestEnd = 2
End Sub

Private Sub Form_Load()
  'txtIO = "* STDIN HANDLE: " & hStdIn & vbTab & "STDOUT HANDLE: " & hStdOut & " *" & vbCrLf
  txtIO = ""
  cboFakeInput = "bench 14"
  cboFakeInput.AddItem "analyze"
  cboFakeInput.AddItem "eval"   ' input in Immediate window and Tracexxx.txt
  cboFakeInput.AddItem "bench 6"
  cboFakeInput.AddItem "writeepd"
  cboFakeInput.AddItem "display"
  cboFakeInput.AddItem "list"
  cboFakeInput.AddItem "new"
  cboFakeInput.AddItem "setboard 1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - 0 1"
  cboFakeInput.AddItem "setboard r1b2rk1/pp1nq1p1/2p1p2p/3p1p2/2PPn3/2NBPN2/PPQ2PPP/2R2RK1 b - -"
```

```vb
    cboFakeInput.AddItem "setboard 2br2k1/ppp2p1p/4p1p1/4P2q/2P1Bn2/2Q5/PP3P1P/4R1RK b
    - -"
    cboFakeInput.AddItem "setboard 8/8/R3k3/1R6/8/8/8/2K5 b - -"
    cboFakeInput.AddItem "setboard 2k4r/1pr1n3/p1p1q2p/5pp1/3P1P2/P1P1P3/1R2Q1PP/1RB3K1
    w KQkq -"
    cboFakeInput.AddItem "setboard 6k1/1b1nqpbp/pp4p1/5P2/1PN5/4Q3/P5PP/1B2B1K1 b - -"
    cboFakeInput.AddItem "perft 3"
    cboFakeInput.AddItem "xboard" & vbLf & "new" & vbLf & "random" & vbLf & "level 40 5
    0" & vbLf & "post"
    cboFakeInput.AddItem "xboard" & vbLf & "new" & vbLf & "random" & vbLf & "sd 4" &
    vbLf & "post"
    cboFakeInput.AddItem "time 30000" & vbLf & "otim 30000" & vbLf & "e2e4"
    cboFakeInput.AddItem "force" & vbLf & "quit"
    cboFakeInput.AddItem "setboard rnbqkbnr/ppp2ppp/4p3/3pP3/3P4/8/PPP2PPP/RNBQKBNR b
    KQkq -"
    cboFakeInput.AddItem "setboard 8/p1b1k1p1/Pp4p1/1Pp2pPp/2P2P1P/3B1K2/8/8 w - -"
    cboFakeInput.AddItem "setboard 8/2R5/1r3kp1/2p4p/2P2P2/p3K1P1/P6P/8 w - -"
    cboFakeInput.AddItem "setboard 7k/p7/6K1/5Q2/8/8/8/8 w - -"
    cboFakeInput.AddItem "debug1"
    DebugMode = True
    cmdTest1_Click
    UCIMode = True
  End Sub

  Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
    ExitProgram
  End Sub

  Private Sub Form_Resize()
    On Local Error Resume Next

  ' With txtIO
  '   .Move .Left, .Top, Me.ScaleWidth - (.Left * 2), Me.ScaleHeight - 800
  ' End With
  '
  ' cboFakeInput.Width = txtIO.Width - cmdFakeInput.Width - 100
  ' cmdFakeInput.Left = cboFakeInput.Left + cboFakeInput.Width + 100
  ' On Local Error GoTo 0
  End Sub

  Attribute VB_Name = "basEPD"
  '==================================================
  '= basEPD:
  '= EPD file format handling
  '==================================================
  Option Explicit
  ' Table for board indexes
  Private EPDTable(63) As Long

  '------------------------------------------------------------------------
  ' ReadEPD()
  ' "ucinewgame" command earlier> calls INITGAME
  '------------------------------------------------------------------------
  Public Function ReadEPD(ByVal sEpdString As String) As Boolean
    Dim NumSquares  As Long, i As Long
    Dim sChar       As String
    Dim arCmdList() As String, p As Long

    Fifty = 0: GameMovesCnt = 0
    BookMovePossible = False
    HintMove = EmptyMove
    arCmdList = Split(sEpdString)
    If UBound(arCmdList) < 3 Then
      ReadEPD = False
      Exit Function
    End If

    For i = 0 To 63  ' Clear board
      Board(EPDTable(i)) = NO_PIECE
    Next

    For i = 0 To MAX_BOARD: Moved(0) = 1: Next 'set unknown moved status

    ' Part 1:  Set pieces on board
    For i = 1 To Len(arCmdList(0))
      sChar = Mid$(arCmdList(0), i, 1)

      Select Case sChar
        Case "P"
          Board(EPDTable(NumSquares)) = WPAWN
          NumSquares = NumSquares + 1
        Case "p"
          Board(EPDTable(NumSquares)) = BPAWN
          NumSquares = NumSquares + 1
        Case "N"
          Board(EPDTable(NumSquares)) = WKNIGHT
          NumSquares = NumSquares + 1
        Case "n"
          Board(EPDTable(NumSquares)) = BKNIGHT
          NumSquares = NumSquares + 1
        Case "K"
          WKingLoc = EPDTable(NumSquares)
          Board(WKingLoc) = WKING
          NumSquares = NumSquares + 1
        Case "k"
          BKingLoc = EPDTable(NumSquares)
          Board(BKingLoc) = BKING
          NumSquares = NumSquares + 1
        Case "R"
          Board(EPDTable(NumSquares)) = WROOK
          NumSquares = NumSquares + 1
        Case "r"
          Board(EPDTable(NumSquares)) = BROOK
          NumSquares = NumSquares + 1
        Case "Q"
          Board(EPDTable(NumSquares)) = WQUEEN
          NumSquares = NumSquares + 1
        Case "q"
          Board(EPDTable(NumSquares)) = BQUEEN
          NumSquares = NumSquares + 1
        Case "B"
          Board(EPDTable(NumSquares)) = WBISHOP
          NumSquares = NumSquares + 1
        Case "b"
          Board(EPDTable(NumSquares)) = BBISHOP
          NumSquares = NumSquares + 1
        Case "/"
        Case Else
          NumSquares = NumSquares + Val(sChar)
      End Select

    Next

    ' part 2: color to move
    sChar = arCmdList(1)
    If LCase(sChar) = "w" Then
      bWhiteToMove = True
    ElseIf LCase(sChar) = "b" Then
      bWhiteToMove = False
    Else
      Exit Function
    End If
    bCompIsWhite = Not bWhiteToMove

    'Part 3: castling
    Moved(WKING_START) = 1: Moved(SQ_A8) = 1: Moved(SQ_A1) = 1
```

```
    Moved(BKING_START) = 1: Moved(SQ_H8) = 1: Moved(SQ_A8) = 1

    For i = 1 To Len(arCmdList(2))
      sChar = Mid$(arCmdList(2), i, 1)

      Select Case sChar
        Case "K"
          Moved(WKING_START) = 0
          Moved(SQ_H1) = 0
        Case "Q"
          Moved(WKING_START) = 0
          Moved(SQ_A1) = 0
        Case "k"
          Moved(BKING_START) = 0
          Moved(SQ_H8) = 0
        Case "q"
          Moved(BKING_START) = 0
          Moved(SQ_A8) = 0
        Case "-"
          Exit For
      End Select

    Next

    'Part4 : EnPassant
    sChar = arCmdList(3)
    If sChar <> "-" Then
      p = FileRev(Left$(sChar, 1)) + RankRev(Right$(sChar, 1))
      If bWhiteToMove Then
        If Right$(sChar, 1) = "6" Then
          Board(p) = BEP_PIECE: EpPosArr(1) = p
        End If
      Else
        If Right$(sChar, 1) = "3" Then
          Board(p) = WEP_PIECE: EpPosArr(1) = p
        End If
      End If
    End If

    'Part5 : Fifty move half move count
    If UBound(arCmdList) >= 4 Then
      sChar = arCmdList(4)
      If sChar <> "" Then
        If Val("0" & sChar) > 0 Then Fifty = Val(sChar)
      End If
    End If

    'Part5 : full move count: 1 before first move, 2 after first black move
    GameMovesCnt = 0
    If UBound(arCmdList) >= 5 Then
      sChar = arCmdList(5)
      If sChar <> "" Then
        If Val("0" & sChar) > 0 Then
          GameMovesCnt = GetMax(0, (Val(sChar) - 1) * 2)
          If Not bWhiteToMove Then GameMovesCnt = GameMovesCnt + 1
        End If
      End If
    End If

    InitPieceSquares
    HashBoard GamePosHash(GameMovesCnt), EmptyMove   ' for 3x repetition draw
    ReadEPD = True
  End Function

  '---------------------------------------------------------------------
  'WriteEPD() -
  '---------------------------------------------------------------------
  Public Function WriteEPD() As String
```

```
    Dim i         As Long
    Dim iPiece    As Long, iEmptySquares As Long
    Dim sEPD      As String, sRow As String
    Dim sEPPiece As String, sCastle As String
    sEPPiece = "-"

    For i = 0 To 63
      If i Mod 8 = 0 And i > 0 Then
        sEPD = sEPD & "/" & sRow & Format$(iEmptySquares, "#")
        iEmptySquares = 0
        sRow = ""
      End If
      iPiece = Board(EPDTable(i))

      Select Case iPiece
        Case NO_PIECE
          iEmptySquares = iEmptySquares + 1
        Case WEP_PIECE, BEP_PIECE
          sEPPiece = Chr$(File(EPDTable(i)) + 96) & Rank(EPDTable(i))
          iEmptySquares = iEmptySquares + 1
        Case Else
          sRow = sRow & Format$(iEmptySquares, "#") & Piece2Alpha(iPiece)
          iEmptySquares = 0
      End Select

    Next

    sEPD = sEPD & "/" & sRow & Format$(iEmptySquares, "#")
    sEPD = Right$(sEPD, Len(sEPD) - 1)
    If bWhiteToMove Then
      sEPD = sEPD & " w"
    Else
      sEPD = sEPD & " b"
    End If
    If Moved(WKING_START) = 0 Then
      If Moved(SQ_H1) = 0 Then sCastle = "K"
      If Moved(SQ_A1) = 0 Then sCastle = sCastle & "Q"
    End If
    If Moved(BKING_START) = 0 Then
      If Moved(SQ_H8) = 0 Then sCastle = sCastle & "k"
      If Moved(SQ_A8) = 0 Then sCastle = sCastle & "q"
    End If
    If sCastle = "" Then sCastle = "-"
    sEPD = sEPD & " " & sCastle & " " & sEPPiece
    sEPD = sEPD & " " & CStr(Fifty)
    sEPD = sEPD & " " & CStr(GameMovesCnt \ 2 + 1)
    WriteEPD = sEPD
  End Function

  Public Sub InitEPDTable()
    EPDTable(0) = 91: EPDTable(1) = 92: EPDTable(2) = 93: EPDTable(3) = 94
    EPDTable(4) = 95: EPDTable(5) = 96: EPDTable(6) = 97: EPDTable(7) = 98
    EPDTable(8) = 81: EPDTable(9) = 82: EPDTable(10) = 83: EPDTable(11) = 84
    EPDTable(12) = 85: EPDTable(13) = 86: EPDTable(14) = 87: EPDTable(15) = 88
    EPDTable(16) = 71: EPDTable(17) = 72: EPDTable(18) = 73: EPDTable(19) = 74
    EPDTable(20) = 75: EPDTable(21) = 76: EPDTable(22) = 77: EPDTable(23) = 78
    EPDTable(24) = 61: EPDTable(25) = 62: EPDTable(26) = 63: EPDTable(27) = 64
    EPDTable(28) = 65: EPDTable(29) = 66: EPDTable(30) = 67: EPDTable(31) = 68
    EPDTable(32) = 51: EPDTable(33) = 52: EPDTable(34) = 53: EPDTable(35) = 54
    EPDTable(36) = 55: EPDTable(37) = 56: EPDTable(38) = 57: EPDTable(39) = 58
    EPDTable(40) = 41: EPDTable(41) = 42: EPDTable(42) = 43: EPDTable(43) = 44
    EPDTable(44) = 45: EPDTable(45) = 46: EPDTable(46) = 47: EPDTable(47) = 48
    EPDTable(48) = 31: EPDTable(49) = 32: EPDTable(50) = 33: EPDTable(51) = 34
    EPDTable(52) = 35: EPDTable(53) = 36: EPDTable(54) = 37: EPDTable(55) = 38
    EPDTable(56) = 21: EPDTable(57) = 22: EPDTable(58) = 23: EPDTable(59) = 24
    EPDTable(60) = 25: EPDTable(61) = 26: EPDTable(62) = 27: EPDTable(63) = 28
  End Sub
  Attribute VB_Name = "basEval"
```

```vb
'=================================================
'= basEval:
'= EVAL function : Evaluation of board position   =
'=================================================
Option Explicit
'--- Game phase
Const PHASE_MIDGAME                As Long = 128
Const PHASE_ENDGAME                As Long = 0
Public Const MAX_SEE_DIFF          As Long = 80   ' greater than value bishop minus value Knight
'Public Const TEMPO_BONUS          As Long = 23  ' 20 bonus for side to move
Public Const SPACE_THRESHOLD       As Long = 12222 ' compute space eval for opening phase only

'--- Endgame eval scale factors
Const SCALE_FACTOR_DRAW = 0
Const SCALE_FACTOR_ONEPAWN = 48
Const SCALE_FACTOR_NORMAL = 64
Const SCALE_FACTOR_MAX = 128
Const SCALE_FACTOR_NONE = 255

'--- Penalties for enemy's safe checks
Const QueenCheck                   As Long = 780
Const RookCheck                    As Long = 880
Const BishopCheck                  As Long = 435
Const KnightCheck                  As Long = 790
'--- evaluation parameters
Public IsolatedPenalty(1)          As TScore
Public BackwardPenalty(1)          As TScore
Public DoubledPenalty              As TScore
Public ConnectedBonus(1, 1, 2, 8)  As TScore
Public LeverBonus(8)               As TScore
Public ShelterWeakness(4, 8)       As Long
Public StormDanger(4, 4, 8)        As Long
Public ThreatenedByHangingPawn     As TScore
Public ThreatByRank                As TScore
Public WeakUnopposedPawn           As TScore
Public Hanging                     As TScore
Public Overload                    As TScore
Public SafeCheck                   As TScore
Public OtherCheck                  As TScore
Public PawnlessFlank               As TScore
Public ScorePawn                   As TScore
Public ScoreKnight                 As TScore
Public ScoreBishop                 As TScore
Public ScoreRook                   As TScore
Public ScoreQueen                  As TScore
Public PieceScore(17)              As Long
Public PieceAbsValue(17)           As Long
Public PieceTypeValue(6)           As Long
Public WMaterial                   As Long
Public WNonPawnMaterial            As Long
Public BMaterial                   As Long
Public BNonPawnMaterial            As Long
Public Material                    As Long
Public NonPawnMaterial             As Long
Public DrawContempt                As Long
Dim WAttack(MAX_BOARD)             As Integer    '- Fields around king: count attacks ' public+Erase is
2x faster than local in Eval function !
Dim BAttack(MAX_BOARD)             As Integer    '- Fields around king: count attacks
Dim WThreat                        As TScore, BThreat As TScore
Public PiecePosScaleFactor         As Long ' set in INI file
Public CompKingDefScaleFactor      As Long ' set in INI file
Public OppKingAttScaleFactor       As Long ' set in INI file
Public PawnStructScaleFactor       As Long ' set in INI file
Public PassedPawnsScaleFactor      As Long ' set in INI file
Public MobilityScaleFactor         As Long ' set in INI file
Public ThreatsScaleFactor          As Long ' set in INI file
Public WKingScaleFactor            As Long, BKingScaleFactor As Long
Public PawnsWMax(9)                As Long  '--- Pawn max rank (2-7) for file A-H
Public PawnsWMin(9)                 As Long  '--- Pawn min rank (2-7) for file A-H
Public WPawns(9)                    As Long  '--- number of pawns for file A-H
Public PawnsBMax(9)                 As Long
Public PawnsBMin(9)                 As Long
Public BPawns(9)                    As Long
Public RootMove                     As TMOVE
Public LastNodesCnt                 As Long
Public LastThreadCheckNodesCnt      As Long
Public StaticEvalArr(MAX_PV)        As Long ' Eval history
Public TestCnt(20)                  As Long '--- Counter for special debug cases
Public MidGameLimit                 As Long
Public EndgameLimit                 As Long
'-----------------------
'--- Piece square tables: value for piece type on specific board position
'-----------------------
Public PsqtWP(MAX_BOARD)            As TScore
Public PsqtBP(MAX_BOARD)            As TScore
Public PsqtWB(MAX_BOARD)            As TScore
Public PsqtBB(MAX_BOARD)            As TScore
Public PsqtWN(MAX_BOARD)            As TScore
Public PsqtBN(MAX_BOARD)            As TScore
Public PsqtWQ(MAX_BOARD)            As TScore
Public PsqtBQ(MAX_BOARD)            As TScore
Public PsqtWR(MAX_BOARD)            As TScore
Public PsqtBR(MAX_BOARD)            As TScore
Public PsqtWK(MAX_BOARD)            As TScore
Public PsqtBK(MAX_BOARD)            As TScore
Public PsqVal(1, 16, MAX_BOARD)     As Long ' piece square score for piece: (endgame,piece,square)
'-----------------------
'--- Mobility values for pieces
'-----------------------
Public MobilityN(9)                 As TScore
Public MobilityB(15)                As TScore
Public MobilityR(15)                As TScore
Public MobilityQ(29)                As TScore
'-----------------------------------------
Public ZeroScore                    As TScore
Public ThreatBySafePawn(5)          As TScore
Public OutpostBonusKnight(1)        As TScore
Public OutpostBonusBishop(1)        As TScore
Public ReachableOutpostKnight(1)    As TScore
Public ReachableOutpostBishop(1)    As TScore
Public KingAttackWeights(6)         As Long
Public QueenMinorsImbalance(12)     As Long
Public WBestPawnVal                 As Long, BBestPawnVal As Long, WBestPawn As Long,
BBestPawn As Long
Public GamePhase                    As Long
Public WKingAttackersWeight         As Long, WKingAttackersCount As Long,
BKingAttackersWeight As Long, BKingAttackersCount As Long
Public bEvalTrace                   As Boolean
Public bTimeTrace                   As Boolean
Public bHashTrace                   As Boolean
Public bWinboardTrace               As Boolean
Public bWbPvInUciFormat             As Boolean
Public bThreadTrace                 As Boolean
Dim PassedPawns(16)                 As Long ' List of passed pawns (Square)
Dim PassedPawnsCnt                  As Long
Dim WPassedPawnAttack               As Long, BPassedPawnAttack As Long
Public PushClose(8)                 As Long
Public PushAway(8)                  As Long
Public PushToEdges(MAX_BOARD)       As Long
Public WOutpostSq(MAX_BOARD)        As Boolean
Public BOutpostSq(MAX_BOARD)        As Boolean
' endgame
Public KRPPKRP_SFactor(8)           As Long

'--- Threat list
Dim ThreatCnt                       As Long
```

```
Public Type TThreatList
  HangCol         As enumColor
  HangPieceType   As Long
  AttackerPieceType    As Long
  AttackerSquare As Long
  AttackedSquare  As Long
End Type
Dim ThreatList(32)                   As TThreatList
' Pawn Eval
Dim Passed                           As Boolean, Opposed As Boolean, Backward As Boolean
Dim Neighbours                       As Boolean, Doubled As Boolean, Lever As Long,
Supported As Long, Phalanx As Long, LeverPush As Long
Public PassedPawnFileBonus(8)        As TScore
Public PassedPawnRankBonus(8)        As TScore
Public PassedDanger(8)               As Long
Private OwnAttCnt                    As Long
' Threats
Public ThreatByMinor(6)              As TScore ' Attacker is defended minor (B/N)
Public ThreatByRook(6)               As TScore
Public ThreatByAttackOnQueen         As TScore
Public KingOnOneBonus                As TScore
Public KingOnManyBonus               As TScore
' King protection
Public KingProtector(5)              As TScore
' Material imbalance (SF6)
Public QuadraticOurs(5, 5)           As Long
Public QuadraticTheirs(5, 5)         As Long
Public PawnSet(8)                    As Long
Public ImbPieceCount(COL_WHITE, 5) As Long
Private bWIsland                     As Boolean, bBIsland As Boolean
Private PieceSqList(15, 10)          As Integer ' <Piece type> <list number> Square List of pieces for
multiple runs thorugh piece list
Private PieceSqListCnt(15)           As Integer ' counter for  PieceLoc
' temp
Private bIniReadDone                 As Boolean
'Public RootSimpleEval As Long


'------------------------------------------------------------------------
'InitEval(ThreatMove)  Set piece values and piece square tables
'------------------------------------------------------------------------
Public Sub InitEval()
  Dim Score As Long, bSaveEvalTrace As Boolean
  ZeroScore.MG = 0: ZeroScore.EG = 0
  '--- Limit  high eval values ( VERY important for playing style!)
  If Not bIniReadDone Then
    bIniReadDone = True
    '--- Default used if INI file is missing
    PiecePosScaleFactor = Val(ReadINISetting("POSITION_FACTOR", "100"))
    MobilityScaleFactor = Val(ReadINISetting("MOBILITY_FACTOR", "100"))
    PawnStructScaleFactor = Val(ReadINISetting("PAWNSTRUCT_FACTOR", "100"))
    PassedPawnsScaleFactor = Val(ReadINISetting("PASSEDPAWNS_FACTOR", "120"))
    ThreatsScaleFactor = Val(ReadINISetting("THREATS_FACTOR", "100"))
    OppKingAttScaleFactor = Val(ReadINISetting("OPPKINGATT_FACTOR", "100"))
    CompKingDefScaleFactor = Val(ReadINISetting("COMPKINGDEF_FACTOR", "100"))
    '
    '--- Piece values  MG=midgame / EG=endgame58
    '--- SF6 values  ( scale to centipawns: \256 )
    '
    ScorePawn.MG = Val(ReadINISetting("PAWN_VAL_MG", "142"))
    ScorePawn.EG = Val(ReadINISetting("PAWN_VAL_EG", "207"))
    ScoreKnight.MG = Val(ReadINISetting("KNIGHT_VAL_MG", "784"))
    ScoreKnight.EG = Val(ReadINISetting("KNIGHT_VAL_EG", "868"))
    ScoreBishop.MG = Val(ReadINISetting("BISHOP_VAL_MG", "828"))
    ScoreBishop.EG = Val(ReadINISetting("BISHOP_VAL_EG", "916"))
    ScoreRook.MG = Val(ReadINISetting("ROOK_VAL_MG", "1286"))
    ScoreRook.EG = Val(ReadINISetting("ROOK_VAL_EG", "1378"))
    ScoreQueen.MG = Val(ReadINISetting("QUEEN_VAL_MG", "2528"))
```

```
      ScoreQueen.EG = Val(ReadINISetting("QUEEN_VAL_EG", "2698"))
      MidGameLimit = Val(ReadINISetting("MIDGAME_LIMIT", "15258")) ' for game phase
      EndgameLimit = Val(ReadINISetting("ENDGAME_LIMIT", "3915"))  ' for game phase
      ' Draw contempt in centipawns > scale to SF (needs ScorePawn.EG set)
      DrawContempt = Val(ReadINISetting(CONTEMPT_KEY, "1"))
      DrawContempt = Eval100ToSF(DrawContempt) ' in centipawns
    End If
    '--- Detect endgame stage ---
    bSaveEvalTrace = bEvalTrace: bEvalTrace = False ' Save trace setting, trace not needed here
    before init done
    Score = Eval() ' Set material,NonPawnMaterial for GamePhase calculation
    bEvalTrace = bSaveEvalTrace
    SetGamePhase NonPawnMaterial ' Set GamePhase, PieceValues, bEndGame
    InitPieceValue
    InitReductionArray
    InitConnectedPawns
    InitOutpostSq
  End Sub

  Public Sub InitPieceValue()
    '--- Piece values, always absolut, positive value
    PieceAbsValue(FRAME) = 0
    PieceAbsValue(WPAWN) = ScorePawn.MG: PieceAbsValue(BPAWN) = ScorePawn.MG
    PieceAbsValue(WKNIGHT) = ScoreKnight.MG: PieceAbsValue(BKNIGHT) = ScoreKnight.MG
    PieceAbsValue(WBISHOP) = ScoreBishop.MG: PieceAbsValue(BBISHOP) = ScoreBishop.MG
    PieceAbsValue(WROOK) = ScoreRook.MG: PieceAbsValue(BROOK) = ScoreRook.MG
    PieceAbsValue(WQUEEN) = ScoreQueen.MG: PieceAbsValue(BQUEEN) = ScoreQueen.MG
    PieceAbsValue(WKING) = 5000: PieceAbsValue(BKING) = 5000
    PieceAbsValue(13) = 0: PieceAbsValue(14) = 0
    PieceAbsValue(WEP_PIECE) = ScorePawn.MG: PieceAbsValue(BEP_PIECE) = ScorePawn.MG
    '--- Piece SCore: positive for White, negative for Black
    PieceScore(FRAME) = 0
    PieceScore(WPAWN) = ScorePawn.MG: PieceScore(BPAWN) = -ScorePawn.MG
    PieceScore(WKNIGHT) = ScoreKnight.MG: PieceScore(BKNIGHT) = -ScoreKnight.MG
    PieceScore(WBISHOP) = ScoreBishop.MG: PieceScore(BBISHOP) = -ScoreBishop.MG
    PieceScore(WROOK) = ScoreRook.MG: PieceScore(BROOK) = -ScoreRook.MG
    PieceScore(WQUEEN) = ScoreQueen.MG: PieceScore(BQUEEN) = -ScoreQueen.MG
    PieceScore(WKING) = 5000: PieceScore(BKING) = -PieceScore(WKING)
    PieceScore(13) = 0: PieceScore(14) = 0
    PieceScore(WEP_PIECE) = ScorePawn.MG: PieceScore(BEP_PIECE) = -ScorePawn.MG
    PieceTypeValue(PT_PAWN) = ScorePawn.MG
    PieceTypeValue(PT_KNIGHT) = ScoreKnight.MG
    PieceTypeValue(PT_BISHOP) = ScoreBishop.MG
    PieceTypeValue(PT_ROOK) = ScoreRook.MG
    PieceTypeValue(PT_QUEEN) = ScoreQueen.MG
    PieceTypeValue(PT_KING) = PieceScore(WKING)
  End Sub

  Public Function SetGamePhase(ByVal NonPawnMaterial As Long) As Long
    Debug.Assert NonPawnMaterial >= 0
    NonPawnMaterial = GetMax(EndgameLimit, GetMin(NonPawnMaterial, MidGameLimit))
    GamePhase = (((NonPawnMaterial - EndgameLimit) * PHASE_MIDGAME) / (MidGameLimit -
    EndgameLimit))
    bEndgame = (GamePhase <= PHASE_ENDGAME)
  End Function

  '---------------------------------------------------------------------------------------------------
  '--- Eval() - Evaluation of position
  '---        Returns value from view of side to move (positive if black to move and black is better)
  '---        Value scaled to stockfish pawn endgame value (258 = 1 pawn)
  '---
  '--- Steps:
  '---       Init: inits attacks arrays, pawn arrays, material values for pieces
  '---       Check material draw or special endgame positions
  '---       STEPS:
  '---       1. Loop over all pieces to fill pawn structure array and pawn threats
  '---       2. Loop over all pieces types: evaluate each piece except kings.
  '---          do a move generation to calculate mobility, attackers, defenders. fill attack array with piece bitcode
```

```
'---       3. Pass for pawn push (locate here because full attack info needed)
'---       4. Calculate king safety ( shelter, pawn storm, check attacks ), king distance to best pawn
'---       5. Calculate threats
'---       6. Calculate trapped bishops, passed pawns, center control, pawn islands
'---       7. Calculate total material values and endgame scale factors
'---       8. Calculate weights and total eval
'---          Add all evalution terms weighted by variables set in INI file:
'---          Material + Position(general) + PawnStructure + PassedPawns + Mobility +
'---          KingSafetyComputer + KingSafetyOpponent + Threats
'---       9. Invert score for black to move
'---      10. Add tempo value for side to move
'----------------------------------------------------------------------------------------------
Public Function Eval() As Long
  Dim a                        As Long, i As Long, Square As Long, Target As Long,
  Offset As Long, MobCnt As Long, r As Long, rr As Long, AttackBit As Long, k As Long,
   ForkCnt As Long, SC As TScore
  Dim WPos                     As TScore, BPos As TScore, WPassed As TScore, BPassed As
   TScore, WMobility As TScore, BMobility As TScore
  Dim WPawnStruct              As TScore, BPawnStruct As TScore, Piece As Long,
  WPawnCnt As Long, BPawnCnt As Long
  Dim WKSafety                 As TScore, BKSafety As TScore, bDoWKSafety As Boolean,
  bDoBKSafety As Boolean
  Dim WKingAdjacentZoneAttCnt As Long, BKingAdjacentZoneAttCnt As Long, WKingAttPieces
   As Long, BKingAttPieces As Long
  Dim KingDanger               As Long, Undefended As Long, RankNum As Long, RelRank As
   Long, QueenWeak As Boolean
  Dim FileNum                  As Long, MinWKingPawnDistance As Long,
  MinBKingPawnDistance As Long ', KingSidePawns As Long , QueenSidePawns As Long
  Dim DefByPawn                As Long, AttByPawn As Long, bAllDefended As Boolean,
  BlockSqDefended As Boolean, WPinnedCnt As Long, BPinnedCnt As Long, WKDefender As
  Long, BKDefender As Long
  Dim RankPath                 As Long, sq As Long ', WSemiOpenFiles As Long, BSemiOpenFiles
  As Long
  Dim BlockSq                  As Long, MBonus As Long, EBonus As Long, UnsafeCnt As
  Long, PieceAttackBit As Long
  Dim OwnCol                   As Long, OppCol As Long, MoveUp As Long, OwnKingLoc As
  Long, OppKingLoc As Long, BlockSqUnsafe As Boolean
  Dim WBishopsOnBlackSq        As Long, WBishopsOnWhiteSq As Long, BBishopsOnBlackSq As
   Long, BBishopsOnWhiteSq As Long, WCenterPawnsBlocked As Long, BCenterPawnsBlocked
  As Long
  Dim WPawnCntOnWhiteSq        As Long, BPawnCntOnWhiteSq As Long, WWeakUnopposedCnt As
   Long, BWeakUnopposedCnt As Long
  Dim WKingFile                As Long, BKingFile As Long, WFrontMostPassedPawnRank As
  Long, BFrontMostPassedPawnRank As Long, ScaleFactor As Long
  Dim WChecksCounted           As Long, BChecksCounted As Long, WUnsafeChecks As Long,
  BUnsafeChecks As Long, KingLevers As Long
  'Dim bLazy As Boolean, SimpleEval As Long

  '
  '------ Init Eval
  '
  If bEvalTrace Then WriteTrace "------- Start Eval ------"
  EvalCnt = EvalCnt + 1
  Eval = 0
  WPawnCnt = PieceCnt(WPAWN): BPawnCnt = PieceCnt(BPAWN)
  WKingFile = File(WKingLoc): BKingFile = File(BKingLoc)
  WNonPawnMaterial = PieceCnt(WQUEEN) * ScoreQueen.MG + PieceCnt(WROOK) * ScoreRook.MG
   + PieceCnt(WBISHOP) * ScoreBishop.MG + PieceCnt(WKNIGHT) * ScoreKnight.MG
  WMaterial = WNonPawnMaterial + WPawnCnt * ScorePawn.MG
  BNonPawnMaterial = PieceCnt(BQUEEN) * ScoreQueen.MG + PieceCnt(BROOK) * ScoreRook.MG
   + PieceCnt(BBISHOP) * ScoreBishop.MG + PieceCnt(BKNIGHT) * ScoreKnight.MG
  BMaterial = BNonPawnMaterial + BPawnCnt * ScorePawn.MG
  NonPawnMaterial = WNonPawnMaterial + BNonPawnMaterial
  Material = WMaterial - BMaterial
  SetGamePhase NonPawnMaterial

   ' Lazy eval?
'  SimpleEval = ScorePawn.EG * (PieceCnt(WPAWN) - PieceCnt(BPAWN)) + (WNonPawnMaterial -
BNonPawnMaterial)
'
' bLazy = (Abs(SimpleEval) >= ScoreRook.EG + ScoreBishop.EG + Abs(FinalScore) + Abs(RootSimpleEval))
' If bLazy Then
'   SimpleEval = SimpleEval + (Nodes And 7) - 3
'   If Not bWhiteToMove Then SimpleEval = -SimpleEval
'  Eval = SimpleEval
'  TestCnt(1) = TestCnt(1) + 1
'  Exit Function
' End If


  'Debug.Assert PieceSqListCnt(WPAWN) = PieceCnt(WPAWN)
  'Debug.Assert PieceSqListCnt(BPAWN) = PieceCnt(BPAWN)
  '
  '--- Endgame function available?
  '
  Select Case WPawnCnt + BPawnCnt
    Case 0 ' no pawns
      ' KQKR
      If (WMaterial = ScoreQueen.MG And BMaterial = ScoreRook.MG) Or (BMaterial = ScoreQueen.MG And WMaterial = ScoreRook.MG) Then
        Eval = Eval_KQKR(): GoTo lblEndEval
      End If
      '--- Insufficent material draw?
      If IsMaterialDraw() Then
        Eval = 0: Exit Function '- Endgame draw: not sufficent material for mate
      End If
    Case 1 ' one pawn
      If (WMaterial = ScoreRook.MG And BMaterial = ScorePawn.MG) Or (BMaterial = ScoreRook.MG And WMaterial = ScorePawn.MG) Then
        Eval = Eval_KRKP(): GoTo lblEndEval ' KRKP
      ElseIf (WMaterial = ScoreQueen.MG And BMaterial = ScorePawn.MG) Or (BMaterial = ScoreQueen.MG And WMaterial = ScorePawn.MG) Then
        Eval = Eval_KQKP(): GoTo lblEndEval ' KQKP
      End If
  End Select

  '----- Init Eval ---------------------
  WBestPawnVal = VALUE_NONE: WBestPawn = 0
  BBestPawnVal = VALUE_NONE: BBestPawn = 0
  WPassedPawnAttack = 0: BPassedPawnAttack = 0
  ThreatCnt = 0: WThreat = ZeroScore: BThreat = ZeroScore

  '--- Fill Pawn Arrays: number of pawns in file
  Erase WPawns: Erase BPawns: Erase PawnsWMax: Erase PawnsBMax
  For a = 0 To 9
    PawnsWMin(a) = 9: PawnsBMin(a) = 9
  Next

  WPawns(0) = -1: BPawns(0) = -1
  WPawns(9) = -1: BPawns(9) = -1
  PassedPawnsCnt = 0
  Erase WAttack(): Erase BAttack() 'Init attack arrays  (fast)
  Erase PieceSqListCnt()
  MinWKingPawnDistance = 9: MinBKingPawnDistance = 9

  '--- Step 1. loop over pieces: count pieces for material totals and game phase calculation. add piece square table score.
  '----                    calc pawn min/max rank positions per file; pawn attacks(for mobility used later)

  For a = 1 To NumPieces
    Square = Pieces(a): If Square = 0 Or Board(Square) >= NO_PIECE Then GoTo lblNextPieceCnt
    r = Board(Square):  PieceSqListCnt(r) = PieceSqListCnt(r) + 1: PieceSqList(r, PieceSqListCnt(r)) = Square ' fill piece list

    Select Case r
```

```
      Case WPAWN
        WAttack(Square + SQ_UP_LEFT) = WAttack(Square + SQ_UP_LEFT) Or PLAttackBit: WAttack(Square + SQ_UP_RIGHT) = WAttack(Square + SQ_UP_RIGHT) Or PRAttackBit ' Set pawn attack here for use in pieces eval
        FileNum = File(Square): RankNum = Rank(Square): WPawns(FileNum) = WPawns(FileNum) + 1
        If RankNum < PawnsWMin(FileNum) Then PawnsWMin(FileNum) = RankNum
        If RankNum > PawnsWMax(FileNum) Then PawnsWMax(FileNum) = RankNum
        If MaxDistance(WKingLoc, Square) < MinWKingPawnDistance Then MinWKingPawnDistance = MaxDistance(WKingLoc, Square)
        If ColorSq(Square) = COL_WHITE Then WPawnCntOnWhiteSq = WPawnCntOnWhiteSq + 1  ' for Bishop eval
        ' If FileNum < FILE_E Then QueenSidePawns = QueenSidePawns + 1 Else KingSidePawns = KingSidePawns + 1
      Case BPAWN
        BAttack(Square + SQ_DOWN_LEFT) = BAttack(Square + SQ_DOWN_LEFT) Or PLAttackBit: BAttack(Square + SQ_DOWN_RIGHT) = BAttack(Square + SQ_DOWN_RIGHT) Or PRAttackBit
        FileNum = File(Square): RankNum = Rank(Square): BPawns(FileNum) = BPawns(FileNum) + 1
        If RankNum < PawnsBMin(FileNum) Then PawnsBMin(FileNum) = RankNum
        If RankNum > PawnsBMax(FileNum) Then PawnsBMax(FileNum) = RankNum
        If MaxDistance(BKingLoc, Square) < MinBKingPawnDistance Then MinBKingPawnDistance = MaxDistance(BKingLoc, Square)
        If ColorSq(Square) = COL_WHITE Then BPawnCntOnWhiteSq = BPawnCntOnWhiteSq + 1 ' for Bishop eval
        ' If FileNum < FILE_E Then QueenSidePawns = QueenSidePawns + 1 Else KingSidePawns = KingSidePawns + 1
    End Select

lblNextPieceCnt:
  Next

  '--- KPK endgame: Eval if promoted pawn cannot be captured
  If NonPawnMaterial = 0 And (WPawnCnt + BPawnCnt = 1) Then
    If WPawnCnt = 1 Then
      sq = PieceSqList(WPAWN, 1)
      If File(sq) = FILE_A Or File(sq) = FILE_H Then
        If File(BKingLoc) = File(sq) And Rank(BKingLoc) > Rank(sq) Then Eval = 0: GoTo lblEndEval
      End If

      If bWhiteToMove Then
        If Rank(sq) = 7 Then
          If sq + SQ_UP <> WKingLoc Then ' own king not at promote square
            If MaxDistance(BKingLoc, sq + SQ_UP) > 1 Or MaxDistance(WKingLoc, sq + SQ_UP) = 1 Then
              Eval = VALUE_KNOWN_WIN: GoTo lblEndEval
            End If
          End If
        End If
        '--- Draw if opp king 2 rows in front of pawn (not at rank 8) and own king behind
        If Rank(BKingLoc) <> 8 Then
          If BKingLoc >= sq + SQ_UP + SQ_UP_LEFT And BKingLoc <= sq + SQ_UP + SQ_UP_RIGHT Then
            If WKingLoc >= sq + SQ_DOWN_LEFT And WKingLoc <= sq + SQ_DOWN_RIGHT Then Eval = 0:  GoTo lblEndEval
          End If
        End If
        '
      End If
    Else
      sq = PieceSqList(BPAWN, 1)
      If File(sq) = FILE_A Or File(sq) = FILE_H Then
        If File(WKingLoc) = File(sq) And Rank(WKingLoc) < Rank(sq) Then Eval = 0: GoTo lblEndEval
      End If

      If Not bWhiteToMove Then
        If Rank(sq) = 2 Then
          If sq + SQ_DOWN <> BKingLoc Then ' own king not at promote square
            If MaxDistance(WKingLoc, sq + SQ_DOWN) > 1 Or MaxDistance(BKingLoc, sq + SQ_DOWN) = 1 Then
              Eval = -VALUE_KNOWN_WIN: GoTo lblEndEval
            End If
          End If
        End If
        '--- Draw if opp king in front of pawn (not at rank 1) and own king behind
        If Rank(WKingLoc) <> 1 Then
          If WKingLoc >= sq + SQ_DOWN + SQ_DOWN_LEFT And WKingLoc <= sq + SQ_DOWN + SQ_DOWN_RIGHT Then
            If BKingLoc >= sq + SQ_UP_LEFT And BKingLoc <= sq + SQ_UP_RIGHT Then Eval = 0: GoTo lblEndEval
          End If
        End If
      End If
    End If
  End If
  '
  '--- King safety needed?
  '
  bDoWKSafety = CBool(BNonPawnMaterial >= ScoreQueen.MG)
  bDoBKSafety = CBool(WNonPawnMaterial >= ScoreQueen.MG)
  WKingAttackersCount = 0: WKingAttackersWeight = 0: BKingAttackersCount = 0: BKingAttackersWeight = 0
  '--- King Position
  WKSafety = ZeroScore: BKSafety = ZeroScore
  If WNonPawnMaterial > 0 And BMaterial = 0 Then
    WPos.EG = WPos.EG + (7 - MaxDistance(BKingLoc, WKingLoc)) * 12 ' follow opp king to edge for mate (KRK, KQK)
    BPos.EG = BPos.EG + PsqtBK(BKingLoc).EG
  ElseIf BNonPawnMaterial > 0 And WMaterial = 0 Then
    BPos.EG = BPos.EG + (7 - MaxDistance(WKingLoc, BKingLoc)) * 12
    WPos.EG = WPos.EG + PsqtWK(WKingLoc).EG
  Else
    AddScore WPos, PsqtWK(WKingLoc)
    AddScore BPos, PsqtBK(BKingLoc)
  End If

  '------------------------------------------------------------------
  '--- Step 2: EVAL Loop over pieces ----------------------------------------
  '------------------------------------------------------------------
  '
  '------------------------------------------------------------------
  '---- WHITE PAWNs -----------------------------------
  '------------------------------------------------------------------
  For a = 1 To PieceSqListCnt(WPAWN)
    Square = PieceSqList(WPAWN, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = RankNum: SC.MG = 0: SC.EG = 0
    WPos.MG = WPos.MG + PsqtWP(Square).MG: WPos.EG = WPos.EG + PsqtWP(Square).EG
    DefByPawn = AttackBitCnt(WAttack(Square) And PAttackBit) ' counts 1 or 2 pawns
    AttByPawn = AttackBitCnt(BAttack(Square) And PAttackBit) ' counts 1 or 2 pawns

    If bEndgame And RankNum > 4 Then If MaxDistance(Square, BKingLoc) = 1 Then SC.EG = SC.EG + 10 ' advanced pawn supported by king
    'If BPawns(FileNum) = 0 Then WSemiOpenFiles = WSemiOpenFiles + 12 \ WPawns(FileNum) ' only count once per file, so 12 \ WPawns(FileNum) works for 1,2,3,4 pawns
    Opposed = (BPawns(FileNum) > 0) And RankNum < PawnsBMax(FileNum)
    Lever = AttByPawn
    Supported = DefByPawn
    LeverPush = AttackBitCnt(BAttack(Square + SQ_UP) And PAttackBit)
    Doubled = (Board(Square + SQ_DOWN) = WPAWN) ' not SQ_UP!
    Neighbours = (WPawns(FileNum + 1) > 0 Or WPawns(FileNum - 1) > 0)
    Phalanx = AttackBitCnt(WAttack(Square + SQ_UP) And PAttackBit)
    '
    If Not Neighbours Or Lever Or RelRank >= 5 Then
```

```
      Backward = False
    Else
      r = GetMin(PawnsWMin(FileNum - 1), PawnsWMin(FileNum + 1))
      If r <= RankNum Then
        Backward = False
      Else
        Backward = True
        If r = RankNum + 1 Then ' can safely advance to not backward rank?
          If LeverPush = 0 Then If Board(Square + SQ_UP) <> BPAWN Then Backward = False
        End If
      End If
    End If

    ' Blocked pawn on center files? Needed for bishop eval
    If FileNum >= FILE_C Then If FileNum <= FILE_F Then If Board(Square + SQ_UP) < NO_PIECE Then WCenterPawnsBlocked = WCenterPawnsBlocked + 1

    '
    '----- Passed pawn?
    '
    Passed = False
    If Doubled Then GoTo lblEndWPassed

    ' Stopper two or more ranks in front?
    For k = -1 To 1
      If PawnsBMax(FileNum + k) > RankNum + 1 Then GoTo lblEndWPassed
    Next k
    If Board(Square + SQ_UP) = BPAWN Then
      ' phalanx neighbour can capture block opp pawn and became a passer
      If Phalanx > LeverPush Then If Supported >= Lever And RankNum >= 5 And bWhiteToMove Then Passed = True: GoTo lblEndWPassed
    Else
      If AttByPawn = 0 Then
        Passed = True:  GoTo lblEndWPassed
      ElseIf Phalanx >= LeverPush Then
        ' debug.print printpos, LocCoord(square)
        If Supported >= Lever Then Passed = True: GoTo lblEndWPassed
      End If
    End If
    '
    If Not Passed And Supported > 0 And RankNum >= 5 Then ' sacrify supporter pawn to create passer?
      If PawnsBMax(FileNum) = RankNum + 1 Then ' blocker pawn
        If PawnsBMax(FileNum - 1) < RankNum Then ' no other stopper left side
          If CBool(WAttack(Square) And PRAttackBit) Then ' left side supporter pawn (attacks to right)
            If Board(Square + SQ_LEFT) >= NO_PIECE Then  ' can move forward to attack stopper
              If Not CBool(BAttack(Square + SQ_LEFT) And PRAttackBit) Then ' no second left to right attacker from file-2
                Passed = True:  GoTo lblEndWPassed
              End If
            End If
          End If
        End If
        If Not Passed Then
          If PawnsBMax(FileNum + 1) < RankNum Then
            If CBool(WAttack(Square) And PLAttackBit) Then ' right side supporter pawn (attacks from left)
              If Board(Square + SQ_RIGHT) >= NO_PIECE Then  ' can move forward to attack stopper
                If Not CBool(BAttack(Square + SQ_RIGHT) And PLAttackBit) Then ' no second right to left attacker
                  Passed = True:  GoTo lblEndWPassed
                End If
              End If
            End If
          End If
        End If
      End If
    End If
lblEndWPassed:

    '--- pawn score
    If Lever Then AddScore SC, LeverBonus(RelRank)
    If Supported Or Phalanx Then ' Connected
      AddScore SC, ConnectedBonus(Abs(Opposed), Abs(Phalanx <> 0), DefByPawn, RelRank)
    ElseIf Not Neighbours Then
      MinusScore SC, IsolatedPenalty(Abs(Opposed))
      If Not Opposed Then WWeakUnopposedCnt = WWeakUnopposedCnt + 1
    ElseIf Backward Then
      MinusScore SC, BackwardPenalty(Abs(Opposed))
    End If
    If Doubled And Supported = 0 Then MinusScore SC, DoubledPenalty
    '--------------------
    If bEndgame Then
      If FileNum = 1 Or FileNum = 8 Then AddScore SC, PsqtWP(Square)
      If WPawnCnt = 1 Then SC.EG = SC.EG + 2 * RelRank * RelRank
      If SC.EG + PsqtWP(Square).EG > WBestPawnVal Then
        WBestPawnVal = SC.EG + PsqtWP(Square).EG: WBestPawn = Square
      ElseIf SC.EG = WBestPawnVal Then
        If WBestPawn = 0 Or MaxDistance(Square, WKingLoc) < MaxDistance(WBestPawn,
        WKingLoc) Then
          WBestPawnVal = SC.EG: WBestPawn = Square
        End If
      End If
    End If
    ' Passed : eval later when full attack is available
    If Passed Then
      PassedPawnsCnt = PassedPawnsCnt + 1: PassedPawns(PassedPawnsCnt) = Square
      If RankNum > 4 Then If Abs(FileNum - BKingFile) <= 2 Then WPassedPawnAttack =
      WPassedPawnAttack + 1
    End If
    '
    AddScore WPawnStruct, SC
    If bEvalTrace Then WriteTrace "WPawn: " & LocCoord(Square) & ">" & SC.MG & ", " &
    SC.EG
  Next a

  '------------------------------------------------------------
  '---- BLACK PAWNs -----------------------------------
  '------------------------------------------------------------
  For a = 1 To PieceSqListCnt(BPAWN)
    Square = PieceSqList(BPAWN, a): FileNum = File(Square): RankNum = Rank(Square):
    RelRank = (9 - RankNum): SC.MG = 0: SC.EG = 0
    'Debug.Assert Board(Square) = BPAWN
    BPos.MG = BPos.MG + PsqtBP(Square).MG: BPos.EG = BPos.EG + PsqtBP(Square).EG
    DefByPawn = AttackBitCnt(BAttack(Square) And PAttackBit) ' counts 1 or 2 pawns
    AttByPawn = AttackBitCnt(WAttack(Square) And PAttackBit) ' counts 1 or 2 pawns

    If bEndgame And RelRank > 4 Then If MaxDistance(Square, WKingLoc) = 1 Then SC.EG =
     SC.EG + 10  ' advanced pawn supported by king
    'If WPawns(FileNum) = 0 Then BSemiOpenFiles = BSemiOpenFiles + 12 \ BPawns(FileNum)
    Opposed = RankNum > PawnsWMin(FileNum)  ' PawnsWMin=9 if no pawn
    Lever = AttByPawn
    Supported = DefByPawn
    LeverPush = AttackBitCnt(WAttack(Square + SQ_DOWN) And PAttackBit)
    Doubled = Abs(Board(Square + SQ_UP) = BPAWN)
    Neighbours = (BPawns(FileNum + 1) > 0 Or BPawns(FileNum - 1) > 0)
    Phalanx = AttackBitCnt(BAttack(Square + SQ_DOWN) And PAttackBit)

    If Not Neighbours Or Lever Or RelRank >= 5 Then
      Backward = False
    Else
      r = GetMax(PawnsBMax(FileNum - 1), PawnsBMax(FileNum + 1))
      If r >= RankNum Then
        Backward = False
      Else
        Backward = True
        If r = RankNum - 1 Then ' can safely advance to not backward rank?
          If LeverPush = 0 Then If Board(Square + SQ_DOWN) <> WPAWN Then Backward = False
        End If
      End If
    End If

    ' Blocked pawn on center files? Needed for bishop eval
    If FileNum >= FILE_C Then If FileNum <= FILE_F Then If Board(Square + SQ_DOWN) < NO_PIECE Then BCenterPawnsBlocked = BCenterPawnsBlocked + 1

    '
    '-----  Passed pawn?
    '
    Passed = False
    If Doubled Then GoTo lblEndBPassed

    ' Stopper two or more ranks in front
    For k = -1 To 1
      If PawnsWMin(FileNum + k) < RankNum - 1 Then GoTo lblEndBPassed
    Next k
    If Board(Square - SQ_UP) = WPAWN Then
       If Phalanx > LeverPush Then If Supported >= Lever And RankNum <= 4 And Not bWhiteToMove Then Passed = True: GoTo lblEndBPassed
    Else
      If AttByPawn = 0 Then
        Passed = True: GoTo lblEndBPassed
      ElseIf Phalanx >= LeverPush Then
        If Supported >= Lever Then Passed = True: GoTo lblEndBPassed
      End If
    End If

    If Not Passed And Supported And RankNum <= 4 Then ' sacrify supporter pawn to create passer?
      If PawnsWMin(FileNum) = RankNum - 1 Then
        If PawnsWMin(FileNum - 1) > RankNum Then ' no other stopper left side (PawnsWMin=9 if no pawn)
          If CBool(BAttack(Square) And PRAttackBit) Then ' left side supporter pawn
            If Board(Square + SQ_LEFT) >= NO_PIECE Then  ' can move forward to attack stopper
              If Not CBool(WAttack(Square + SQ_LEFT) And PRAttackBit) Then ' no second left to right attacker from file-2
                Passed = True:  GoTo lblEndBPassed
              End If
            End If
          End If
        End If
        If Not Passed Then
          If PawnsWMin(FileNum + 1) > RankNum Then
            If CBool(BAttack(Square) And PLAttackBit) Then ' right side supporter pawn
              If Board(Square + SQ_RIGHT) >= NO_PIECE Then  ' can move forward to attack stopper
                If Not CBool(WAttack(Square + SQ_RIGHT) And PLAttackBit) Then ' no second right to left attacker
                  Passed = True: GoTo lblEndBPassed
                End If
              End If
            End If
          End If
        End If
      End If
    End If
  lblEndBPassed:

    '--- pawn score
    If Lever Then AddScore SC, LeverBonus(RelRank)
    If Supported Or Phalanx Then ' Connected
```

```
        AddScore SC, ConnectedBonus(Abs(Opposed), Abs(Phalanx <> 0), DefByPawn, RelRank)
      ElseIf Not Neighbours Then
        MinusScore SC, IsolatedPenalty(Abs(Opposed))
        If Not Opposed Then BWeakUnopposedCnt = BWeakUnopposedCnt + 1
      ElseIf Backward Then
        MinusScore SC, BackwardPenalty(Abs(Opposed))
      End If
      If Doubled And Supported = 0 Then MinusScore SC, DoubledPenalty
      '------------------------
      If bEndgame Then
        If FileNum = 1 Or FileNum = 8 Then AddScore SC, PsqtBP(Square)
        If BPawnCnt = 1 Then SC.EG = SC.EG + 2 * RelRank * RelRank
        If SC.EG + PsqtBP(Square).EG > BBestPawnVal Then
          BBestPawnVal = SC.EG + PsqtBP(Square).EG: BBestPawn = Square
        ElseIf SC.EG = BBestPawnVal Then
          If BBestPawn = 0 Or MaxDistance(Square, BKingLoc) < MaxDistance(BBestPawn, BKingLoc) Then
            BBestPawnVal = SC.EG: BBestPawn = Square
          End If
        End If
      End If
      ' Passed : eval later when full attack is available
      If Passed And Not Doubled Then
        PassedPawnsCnt = PassedPawnsCnt + 1: PassedPawns(PassedPawnsCnt) = Square
        If RelRank > 4 Then If Abs(FileNum - WKingFile) <= 2 Then BPassedPawnAttack = BPassedPawnAttack + 1
      End If
      '
      AddScore BPawnStruct, SC
      If bEvalTrace Then WriteTrace "BPawn: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG
    Next a

    '------------------------------------------------------------------
    '---- WHITE KNIGHTs ----------------------------------
    '------------------------------------------------------------------ '
    For a = 1 To PieceSqListCnt(WKNIGHT)
      Square = PieceSqList(WKNIGHT, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = RankNum: SC.MG = 0: SC.EG = 0
      WPos.MG = WPos.MG + PsqtWN(Square).MG: WPos.EG = WPos.EG + PsqtWN(Square).EG: r = 0
      ' Outpost bonus
      If WOutpostSq(Square) Then
        If Not CBool(BAttack(Square) And PAttackBit) Then ' not attacked by pawn
          ' Defended by pawn?
          AddScore SC, OutpostBonusKnight(Abs(CBool(WAttack(Square) And PAttackBit))): r = 3 ' ignore ReachableOutpost
          If bEvalTrace Then WriteTrace "WKight: " & LocCoord(Square) & "> Outpost:" & OutpostBonusKnight(Abs(CBool(WAttack(Square) And PAttackBit))).MG
        End If
      End If
      '--- Mobility
      If Moved(Square) = 0 Then If RankNum = 1 Then SC.MG = SC.MG - 45 ' develop knight
      ForkCnt = 0: MobCnt = 0
      If a = 1 Then PieceAttackBit = N1AttackBit Else PieceAttackBit = N2AttackBit

      For i = 0 To 7
        Offset = KnightOffsets(i): Target = Square + Offset
        If Board(Target) <> FRAME Then
          WAttack(Target) = WAttack(Target) Or PieceAttackBit

          Select Case Board(Target)
            Case NO_PIECE:
              If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1: SC.MG = SC.MG + 3
            Case WPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 2
              If RankNum > 3 Then If Board(Target + SQ_UP) >= NO_PIECE Then If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
            Case BPAWN: SC.MG = SC.MG + 7: SC.EG = SC.EG + 7: If Rank(Target) >= 6 Then SC.MG = SC.MG + 4
              If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1: AddThreat COL_BLACK, PT_PAWN, PT_KNIGHT, Square, Target
            Case BKNIGHT, BBISHOP: If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
              AddThreat COL_BLACK, PieceType(Board(Target)), PT_KNIGHT, Square, Target '-- no Score for WKnight : total is zero
            Case BROOK, BQUEEN: If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
              AddThreat COL_BLACK, PieceType(Board(Target)), PT_KNIGHT, Square, Target: ForkCnt = ForkCnt + 1
            Case WKING, WQUEEN: ' ignore
            Case BKING: MobCnt = MobCnt + 1: ForkCnt = ForkCnt + 1
            Case WEP_PIECE, BEP_PIECE:
              If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1: SC.MG = SC.MG + 3
            Case Else: If (Not CBool(BAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
          End Select

          If r < 2 Then ' choose best square only
            If WOutpostSq(Target) Then ' Empty or opp piece: square can be occupied.
              ' not attacked by opp pawn? Else if not blocked by own piece
              If Not CBool(BAttack(Target) And PAttackBit) Then
                r = 2: rr = 1 + Abs(CBool(WAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
              Else
                If r = 0 Then If PieceColor(Board(Target)) <> COL_WHITE Then r = 1: rr = 1 + Abs(CBool(WAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
              End If
            End If
          End If
        End If
      Next

      If ForkCnt > 1 Then AddScoreVal SC, 7 * ForkCnt * ForkCnt, 5 * ForkCnt * ForkCnt: If bWhiteToMove Then AddScoreVal SC, 35, 35
      AddScore WMobility, MobilityN(MobCnt)
      ' Minor behind pawn bonus
      If RelRank < 5 Then
        If PieceType(Board(Square + SQ_UP)) = PT_PAWN Then SC.MG = SC.MG + 16: If bEvalTrace Then WriteTrace "WKnight: " & LocCoord(Square) & "> Behind pawn 16"
      End If
      If r > 0 And r < 3 Then AddScoreWithFactor SC, ReachableOutpostKnight(r - 1), rr
      If CBool(BAttack(Square) And PAttackBit) Then AddPawnThreat BThreat, COL_WHITE, PieceType(Board(Square)), Square
      AddScoreWithFactor SC, KingProtector(PT_KNIGHT), MaxDistance(Square, WKingLoc) ' defends king?
      AddScore WPos, SC
      If bEvalTrace Then WriteTrace "WKnight: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & WPos.MG & ", " & WPos.EG
    Next a

    '------------------------------------------------------------------
    '---- BLACK KNIGHTs ----------------------------------
    '------------------------------------------------------------------ '
    For a = 1 To PieceSqListCnt(BKNIGHT)
      Square = PieceSqList(BKNIGHT, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = (9 - RankNum): SC.MG = 0: SC.EG = 0
      BPos.MG = BPos.MG + PsqtBN(Square).MG: BPos.EG = BPos.EG + PsqtBN(Square).EG: r = 0
      ' Outpost bonus
      If BOutpostSq(Square) Then
        If Not CBool(WAttack(Square) And PAttackBit) Then ' not attacked by pawn
          ' Defended by pawn?
          AddScore SC, OutpostBonusKnight(Abs(CBool(BAttack(Square) And PAttackBit))): r
```

```
         = 3 ' ignore ReachableOutpost
        If bEvalTrace Then WriteTrace "BKight: " & LocCoord(Square) & "> Outpost:" & OutpostBonusKnight(Abs(CBool(BAttack(Square) And PAttackBit))).MG
      End If
    End If
    If Moved(Square) = 0 Then If RankNum = 8 Then SC.MG = SC.MG - 45
    '--- Mobility
    ForkCnt = 0: MobCnt = 0
    If a = 1 Then PieceAttackBit = N1AttackBit Else PieceAttackBit = N2AttackBit

    For i = 0 To 7
      Offset = KnightOffsets(i)
      Target = Square + Offset
      If Board(Target) <> FRAME Then
        BAttack(Target) = BAttack(Target) Or PieceAttackBit
        Select Case Board(Target)
          Case NO_PIECE:
            If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1: SC.MG = SC.MG + 3
          Case BPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 2
            If RankNum < 6 Then If Board(Target + SQ_DOWN) >= NO_PIECE Then If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          Case WPAWN: SC.MG = SC.MG + 7: SC.EG = SC.EG + 7: If Rank(Target) <= 3 Then SC.MG = SC.MG + 4
            If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
            AddThreat COL_WHITE, PT_PAWN, PT_KNIGHT, Square, Target
          Case WKNIGHT, WBISHOP: If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
            AddThreat COL_WHITE, PieceType(Board(Target)), PT_KNIGHT, Square, Target
            '-- no Score for WKnight : total is zero
          Case WROOK, WQUEEN: If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
            AddThreat COL_WHITE, PieceType(Board(Target)), PT_KNIGHT, Square, Target: ForkCnt = ForkCnt + 1
          Case BKING, BQUEEN: ' Ignore
          Case WKING: If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
            If (Not CBool(WAttack(Target) And PAttackBit)) Then ForkCnt = ForkCnt + 1
          Case WEP_PIECE, BEP_PIECE:
            If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1: SC.MG = SC.MG + 3
          Case Else: If (Not CBool(WAttack(Target) And PAttackBit)) Then MobCnt = MobCnt + 1
        End Select

        If r < 2 Then
          If BOutpostSq(Target) Then ' Empty or opp piece: square can be occupied
            ' not attacked by opp pawn? Else if not blocked by own piece
            If Not CBool(WAttack(Target) And PAttackBit) Then
              r = 2: rr = 1 + Abs(CBool(BAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
            Else
              If r = 0 Then If PieceColor(Board(Target)) <> COL_BLACK Then r = 1: rr = 1 + Abs(CBool(BAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
            End If
          End If
        End If
      End If
    Next

    If ForkCnt > 1 Then AddScoreVal SC, 7 * ForkCnt * ForkCnt, 5 * ForkCnt * ForkCnt: If Not bWhiteToMove Then AddScoreVal SC, 35, 35
    AddScore BMobility, MobilityN(MobCnt)
    ' Minor behind pawn bonus
    If RelRank < 5 Then
      If PieceType(Board(Square + SQ_DOWN)) = PT_PAWN Then SC.MG = SC.MG + 16: If bEvalTrace Then WriteTrace "BKnight: " & LocCoord(Square) & "> Behind pawn 16"
    End If
    If r > 0 And r < 3 Then AddScoreWithFactor SC, ReachableOutpostKnight(r - 1), rr
    If CBool(WAttack(Square) And PAttackBit) Then AddPawnThreat WThreat, COL_BLACK, PieceType(Board(Square)), Square
    AddScoreWithFactor SC, KingProtector(PT_KNIGHT), MaxDistance(Square, BKingLoc)  ' defends king?
    AddScore BPos, SC
    If bEvalTrace Then WriteTrace "BKnight: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & BPos.MG & ", " & BPos.EG
  Next a

  '-------------------------------------------------------------------
  '---- WHITE BISHOPs ------------------------------------
  '-------------------------------------------------------------------  '
  For a = 1 To PieceSqListCnt(WBISHOP)
    Square = PieceSqList(WBISHOP, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = RankNum: SC.MG = 0: SC.EG = 0
    If ColorSq(Square) = COL_WHITE Then WBishopsOnWhiteSq = WBishopsOnWhiteSq + 1 Else WBishopsOnBlackSq = WBishopsOnBlackSq + 1
    WPos.MG = WPos.MG + PsqtWB(Square).MG: WPos.EG = WPos.EG + PsqtWB(Square).EG: r = 0
    ' Outpost bonus
    If WOutpostSq(Square) Then
      If Not CBool(BAttack(Square) And PAttackBit) Then ' not attacked by pawn
        ' Defended by pawn?
        AddScore SC, OutpostBonusBishop(Abs(CBool(WAttack(Square) And PAttackBit))): r = 3 ' ignore ReachableOutpost
        If bEvalTrace Then WriteTrace "WBishop: " & LocCoord(Square) & "> Outpost:" & OutpostBonusBishop(Abs(CBool(WAttack(Square) And PAttackBit))).MG
      End If
    End If
    '--- Mobility
    MobCnt = 0
    If a = 1 Then PieceAttackBit = B1AttackBit Else PieceAttackBit = B2AttackBit

    For i = 4 To 7
      Offset = DirectionOffset(i): Target = Square + Offset: AttackBit = PieceAttackBit

      Do While Board(Target) <> FRAME
        WAttack(Target) = WAttack(Target) Or AttackBit

        Select Case Board(Target)
          Case NO_PIECE:
            If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1: If Offset > 0 Then SC.MG = SC.MG + 2
          Case WPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 3:
            If RankNum > 3 Then If Board(Target + SQ_UP) >= NO_PIECE Then If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
            If Offset > 0 Then WAttack(Target + Offset) = WAttack(Target + Offset) Or BXrayAttackBit
            Exit Do
          Case BPAWN: If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
            If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PT_PAWN, PT_BISHOP, Square, Target: SC.MG = SC.MG + 7: SC.EG = SC.EG + 7
            Exit Do
          Case BKNIGHT, BBISHOP, BROOK, BQUEEN: If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
            If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PieceType(Board(Target)), PT_BISHOP, Square, Target ' Reattack: no SC because x-x=0
            Exit Do
          Case WKING: Exit Do ' ignore
          Case BKING: MobCnt = MobCnt + 1
            Exit Do
          Case WQUEEN: AttackBit = BXrayAttackBit  '--- Continue xray
          Case WEP_PIECE, BEP_PIECE:
            If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1: If
```

```
                Offset > 0 Then SC.MG = SC.MG + 2
              Case Else: If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt
               + 1
                Exit Do ' own bishop or knight
            End Select

            If r < 2 Then
              If WOutpostSq(Target) Then ' Empty or opp piece: square can be occupied
                ' not attacked by opp pawn? Else if not blocked by own piece
                If Not CBool(BAttack(Target) And PAttackBit) Then
                  r = 2: rr = 1 + Abs(CBool(WAttack(Target) And PAttackBit)) ' supported by
                  own pawn? Factor 2
                Else
                  If r = 0 Then If PieceColor(Board(Target)) <> COL_WHITE Then r = 1: rr =
                   1 + Abs(CBool(WAttack(Target) And PAttackBit)) ' supported by own pawn?
                  Factor 2
                End If
              End If
            End If
            Target = Target + Offset
          Loop

        Next

        AddScore WMobility, MobilityB(MobCnt)
        If bEvalTrace Then WriteTrace "WBishop: " & LocCoord(Square) & ">" & SC.MG & ", "
        & SC.EG & " / " & WPos.MG & ", " & WPos.EG
        ' Minor behind pawn bonus
        If RelRank < 5 Then
          If PieceType(Board(Square + SQ_UP)) = PT_PAWN Then SC.MG = SC.MG + 16: If
          bEvalTrace Then WriteTrace "WBishop: " & LocCoord(Square) & "> Behind pawn 16"
        End If
        If r > 0 And r < 3 Then AddScoreWithFactor SC, ReachableOutpostBishop(r - 1), rr
        If CBool(BAttack(Square) And PAttackBit) Then AddPawnThreat BThreat, COL_WHITE,
        PieceType(Board(Square)), Square
        AddScoreWithFactor SC, KingProtector(PT_BISHOP), MaxDistance(Square, WKingLoc) '
        defends king?
        AddScore WPos, SC
      Next a

      '------------------------------------------------------------------
      '---- BLACK BISHOPs ------------------------------------
      '------------------------------------------------------------------ '
      For a = 1 To PieceSqListCnt(BBISHOP)
        Square = PieceSqList(BBISHOP, a): FileNum = File(Square): RankNum = Rank(Square):
        RelRank = (9 - RankNum): SC.MG = 0: SC.EG = 0
        If ColorSq(Square) = COL_WHITE Then BBishopsOnWhiteSq = BBishopsOnWhiteSq + 1 Else
         BBishopsOnBlackSq = BBishopsOnBlackSq + 1
        BPos.MG = BPos.MG + PsqtBB(Square).MG: BPos.EG = BPos.EG + PsqtBB(Square).EG: r =
        0
        ' Outpost bonus
        If BOutpostSq(Square) Then
          If Not CBool(WAttack(Square) And PAttackBit) Then ' not attacked by pawn
            ' Defended by pawn?
            AddScore SC, OutpostBonusBishop(Abs(CBool(BAttack(Square) And PAttackBit))): r
             = 3 ' ignore ReachableOutpost
            If bEvalTrace Then WriteTrace "BBishop: " & LocCoord(Square) & "> Outpost:" &
            OutpostBonusBishop(Abs(CBool(BAttack(Square) And PAttackBit))).MG
          End If
        End If
        '--- Mobility
        MobCnt = 0
        If a = 1 Then PieceAttackBit = B1AttackBit Else PieceAttackBit = B2AttackBit

        For i = 4 To 7
          Offset = DirectionOffset(i): Target = Square + Offset:  AttackBit =
          PieceAttackBit

```

```
    Do While Board(Target) <> FRAME
      BAttack(Target) = BAttack(Target) Or AttackBit

      Select Case Board(Target)
        Case NO_PIECE:
          If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1:  If Offset < 0 Then SC.MG = SC.MG + 2
        Case BPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 3
          If RankNum < 6 Then If Board(Target + SQ_DOWN) >= NO_PIECE Then If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          If Offset < 0 Then BAttack(Target + Offset) = BAttack(Target + Offset) Or BXrayAttackBit
          Exit Do
        Case WPAWN: If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PT_PAWN, PT_BISHOP, Square, Target: SC.MG = SC.MG + 7: SC.EG = SC.EG + 7
          Exit Do
        Case WKNIGHT, WBISHOP, WROOK, WQUEEN: If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PieceType(Board(Target)), PT_BISHOP, Square, Target
          Exit Do
        Case BKING: Exit Do ' Ignore
        Case WKING: MobCnt = MobCnt + 1
          Exit Do
        Case BQUEEN: AttackBit = BXrayAttackBit '--- Continue xray
        Case WEP_PIECE, BEP_PIECE:
          If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1:  If Offset < 0 Then SC.MG = SC.MG + 2
        Case Else: If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          Exit Do ' own bishop or knight
      End Select

      If r < 2 Then
        If BOutpostSq(Target) Then ' Empty or opp piece: square can be occupied
          ' not attacked by opp pawn? Else if not blocked by own piece
          If Not CBool(WAttack(Target) And PAttackBit) Then
            r = 2: rr = 1 + Abs(CBool(BAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
          Else
            If r = 0 Then If PieceColor(Board(Target)) <> COL_BLACK Then r = 1: rr = 1 + Abs(CBool(BAttack(Target) And PAttackBit)) ' supported by own pawn? Factor 2
          End If
        End If
      End If
      Target = Target + Offset
    Loop

  Next

  AddScore BMobility, MobilityB(MobCnt)
  If bEvalTrace Then WriteTrace "BBishop: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & BPos.MG & ", " & BPos.EG
  ' Minor behind pawn bonus
  If RelRank < 5 Then
    If PieceType(Board(Square + SQ_DOWN)) = PT_PAWN Then SC.MG = SC.MG + 16: If bEvalTrace Then WriteTrace "BBishop: " & LocCoord(Square) & "> Behind pawn 16"
  End If
  If r > 0 And r < 3 Then AddScoreWithFactor SC, ReachableOutpostBishop(r - 1), rr
  If CBool(WAttack(Square) And PAttackBit) Then AddPawnThreat WThreat, COL_BLACK, PieceType(Board(Square)), Square
  AddScoreWithFactor SC, KingProtector(PT_BISHOP), MaxDistance(Square, BKingLoc) ' defends king?
  AddScore BPos, SC
Next a
```

```
'--------------------------------------------------------------------
'---- WHITE ROOKs ------------------------------------
'--------------------------------------------------------------------  '
For a = 1 To PieceSqListCnt(WROOK)
  Square = PieceSqList(WROOK, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = RankNum: SC.MG = 0: SC.EG = 0
  WPos.MG = WPos.MG + PsqtWR(Square).MG: WPos.EG = WPos.EG + PsqtWR(Square).EG
  If WPawns(FileNum) = 0 Then
    If BPawns(FileNum) = 0 Then
      SC.MG = SC.MG + 45: SC.EG = SC.EG + 20
    Else
      SC.MG = SC.MG + 20: SC.EG = SC.EG + 7
    End If
  End If
  '--- Mobility
  MobCnt = 0
  If a = 1 Then PieceAttackBit = R1AttackBit Else PieceAttackBit = R2AttackBit

  For i = 0 To 3
    Offset = DirectionOffset(i): Target = Square + Offset: AttackBit = PieceAttackBit

    Do While Board(Target) <> FRAME
      WAttack(Target) = WAttack(Target) Or AttackBit

      Select Case Board(Target)
        Case NO_PIECE:
          If Not CBool(BAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1: If Abs(Offset) = 10 Then SC.MG = SC.MG + 7
        Case WPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 5: If RankNum > 3 Then If Board(Target + SQ_UP) >= NO_PIECE Then If Not CBool(BAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1
          Exit Do
        Case BPAWN:
          SC.MG = SC.MG + 7: SC.EG = SC.EG + 10 '--- no reattack possible
          If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PT_PAWN, PT_ROOK, Square, Target
          If RankNum >= 5 Then SC.MG = SC.MG + 8: SC.EG = SC.EG + 25  ' aligned pawns
          Exit Do
        Case BKNIGHT, BBISHOP:
          If Not CBool(BAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
          If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PieceType(Board(Target)), PT_ROOK, Square, Target  '--- no reattack possible
          Exit Do
        Case BROOK:  If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PT_ROOK, PT_ROOK, Square, Target
          MobCnt = MobCnt + 1
          Exit Do ' equal exchange, ok for mobility
        Case WKING: Exit Do ' ignore
        Case BKING: MobCnt = MobCnt + 1
          Exit Do
        Case BQUEEN: MobCnt = MobCnt + 1:  If AttackBit = PieceAttackBit Then AddThreat COL_BLACK, PT_QUEEN, PT_ROOK, Square, Target
          Exit Do
        Case WROOK, WQUEEN:
          If Offset = 10 Then
            If WPawns(FileNum) = 0 Then SC.MG = SC.MG + 12: If BPawns(FileNum) = 0 Then SC.MG = SC.MG + 15
          End If
          If Board(Target) = WROOK Then If Not CBool(BAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1
          If a = 1 Then AttackBit = R1XrayAttackBit Else AttackBit = R2XrayAttackBit '--- double lines , continue xray
        Case WEP_PIECE, BEP_PIECE:
          If Not CBool(BAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1: If Abs(Offset) = 10 Then SC.MG = SC.MG + 7
```

```vb
            Case Else: If Not CBool(BAttack(Target) And PBNAttackBit) Then MobCnt =
            MobCnt + 1
              Exit Do ' own bishop or knight
          End Select

          Target = Target + Offset
        Loop

      Next

      AddScore WMobility, MobilityR(MobCnt)
      ' Trapped rook by king : worse when cannot castle
      If Not bEndgame Then
        If MobCnt <= 3 Then
          If WPawns(FileNum) > 0 Then
            If RankNum = Rank(WKingLoc) Or Rank(WKingLoc) = 1 Then
              r = 0
              If WKingFile < FILE_E Then
                If FileNum < WKingFile Then r = -1
              Else
                If FileNum > WKingFile Then r = 1
              End If
              If r <> 0 Then

                For k = WKingFile + r To FileNum - r Step r ' own blocking pawns on files between
                king an rook
                  If WPawns(k) = 0 Then
                    r = 0: Exit For
                  ElseIf PawnsWMin(k) > RankNum + 2 Then
                    r = 0: Exit For
                  End If
                Next

                If r <> 0 Then SC.MG = SC.MG - (92 - MobCnt * 22) * (1 + Abs(Rank(
                WKingLoc) = 1 And (Moved(WKING_START) > 0 Or (Moved(Square) > 0 And
                RankNum = 1))))
              End If
            End If
          End If
        End If
      Else
        If WPawns(FileNum) > 0 And BPawns(FileNum) = 0 And PawnsWMin(FileNum) >= 5 Then
          SC.MG = SC.MG + (PawnsWMin(FileNum)): SC.EG = SC.EG + 5 * PawnsWMin(FileNum)
        End If
      End If
      If CBool(BAttack(Square) And PAttackBit) Then AddPawnThreat BThreat, COL_WHITE,
      PieceType(Board(Square)), Square
      AddScoreWithFactor SC, KingProtector(PT_ROOK), MaxDistance(Square, WKingLoc) '
      defends king?
      AddScore WPos, SC
      If bEvalTrace Then WriteTrace "WRook: " & LocCoord(Square) & ">" & SC.MG & ", " &
      SC.EG & " / " & WPos.MG & ", " & WPos.EG
    Next a

    '-------------------------------------------------------------------
    '---- BLACK ROOKs -----------------------------------
    '------------------------------------------------------------------- '
    For a = 1 To PieceSqListCnt(BROOK)
      Square = PieceSqList(BROOK, a): FileNum = File(Square): RankNum = Rank(Square):
      RelRank = (9 - RankNum): SC.MG = 0: SC.EG = 0
      BPos.MG = BPos.MG + PsqtBR(Square).MG: BPos.EG = BPos.EG + PsqtBR(Square).EG
      If BPawns(FileNum) = 0 Then
        If WPawns(FileNum) = 0 Then
          SC.MG = SC.MG + 45: SC.EG = SC.EG + 20
        Else
          SC.MG = SC.MG + 20: SC.EG = SC.EG + 7
        End If
      End If
```

```
'--- Mobility
MobCnt = 0
If a = 1 Then PieceAttackBit = R1AttackBit Else PieceAttackBit = R2AttackBit

For i = 0 To 3
  Offset = DirectionOffset(i): Target = Square + Offset: AttackBit = PieceAttackBit

  Do While Board(Target) <> FRAME
    BAttack(Target) = BAttack(Target) Or AttackBit

    Select Case Board(Target)
      Case NO_PIECE:
        If Not CBool(WAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1: If Abs(Offset) = 10 Then SC.MG = SC.MG + 7
      Case BPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 5
        If RankNum < 6 Then If Board(Target + SQ_DOWN) >= NO_PIECE Then If Not CBool(WAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1
        Exit Do
      Case WPAWN:
        SC.MG = SC.MG + 7: SC.EG = SC.EG + 10  '--- no reattack possible
        If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
        If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PT_PAWN, PT_ROOK, Square, Target
        If RankNum <= 4 Then SC.MG = SC.MG + 8: SC.EG = SC.EG + 25  ' aligned pawns
        Exit Do
      Case WKNIGHT, WBISHOP:
        If Not CBool(WAttack(Target) And PAttackBit) Then MobCnt = MobCnt + 1
        If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PieceType(Board(Target)), PT_ROOK, Square, Target
        Exit Do   '--- no reattack possible
      Case WROOK:  If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PT_ROOK, PT_ROOK, Square, Target
        MobCnt = MobCnt + 1
        Exit Do  ' equal exchange ok for mobility
      Case BKING: Exit Do ' Ignore
      Case WKING: MobCnt = MobCnt + 1
        Exit Do
      Case WQUEEN: MobCnt = MobCnt + 1:  If AttackBit = PieceAttackBit Then AddThreat COL_WHITE, PT_QUEEN, PT_ROOK, Square, Target
        Exit Do
      Case BROOK, BQUEEN:
        If Offset = -10 Then
          If BPawns(FileNum) = 0 Then SC.MG = SC.MG + 12: If WPawns(FileNum) = 0 Then SC.MG = SC.MG + 15
        End If
        If Board(Target) = BROOK Then If Not CBool(WAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1
        If a = 1 Then AttackBit = R1XrayAttackBit Else AttackBit = R2XrayAttackBit  '--- double lines , continue xray
      Case WEP_PIECE, BEP_PIECE:
        If Not CBool(WAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1: If Abs(Offset) = 10 Then SC.MG = SC.MG + 7
      Case Else: If Not CBool(WAttack(Target) And PBNAttackBit) Then MobCnt = MobCnt + 1
        Exit Do ' own bishop or knight
    End Select

    Target = Target + Offset
  Loop

Next

AddScore BMobility, MobilityR(MobCnt)
' Trapped rook by king : worse when cannot castle
If Not bEndgame Then
  If MobCnt <= 3 Then
    If BPawns(FileNum) > 0 Then
```

```
            If RankNum = Rank(BKingLoc) Or Rank(BKingLoc) = 1 Then
              r = 0
              If BKingFile < FILE_E Then
                If FileNum < BKingFile Then r = -1
              Else
                If FileNum > BKingFile Then r = 1
              End If
              If r <> 0 Then

                For k = BKingFile + r To FileNum - r Step r ' own blocking pawns on files between king an rook
                  If BPawns(k) = 0 Then
                    r = 0: Exit For
                  ElseIf PawnsBMax(k) < RankNum - 2 Then
                    r = 0: Exit For
                  End If
                Next

                If r <> 0 Then SC.MG = SC.MG - (92 - MobCnt * 22) * (1 + Abs(Rank(BKingLoc) = 8 And (Moved(BKING_START) > 0 Or (Moved(Square) > 0 And RankNum = 8))))
              End If
            End If
          End If
        End If
      Else
        If BPawns(FileNum) > 0 And WPawns(FileNum) = 0 And PawnsBMax(FileNum) <= 4 Then
          SC.MG = SC.MG + (9 - PawnsBMin(FileNum)): SC.EG = SC.EG + 5 * (9 - PawnsBMin(FileNum))
        End If
      End If
      If CBool(WAttack(Square) And PAttackBit) Then AddPawnThreat WThreat, COL_BLACK, PieceType(Board(Square)), Square
      AddScoreWithFactor SC, KingProtector(PT_ROOK), MaxDistance(Square, BKingLoc) ' defends king?
      AddScore BPos, SC
      If bEvalTrace Then WriteTrace "BRook: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & BPos.MG & ", " & BPos.EG
    Next a

    '---------------------------------------------------------------------
    '---- WHITE QUEENs ( last - full attack info needed for mobility ) -
    '---------------------------------------------------------------------
    For a = 1 To PieceSqListCnt(WQUEEN)
      Square = PieceSqList(WQUEEN, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = RankNum: SC.MG = 0: SC.EG = 0: QueenWeak = False
      WPos.MG = WPos.MG + PsqtWQ(Square).MG: WPos.EG = WPos.EG + PsqtWQ(Square).EG
      '--- Mobility
      MobCnt = 0

      For i = 0 To 7
        Offset = DirectionOffset(i): Target = Square + Offset: AttackBit = QAttackBit

        Do While Board(Target) <> FRAME
          WAttack(Target) = WAttack(Target) Or AttackBit

          Select Case Board(Target)
            Case NO_PIECE: If Not CBool(BAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
            Case WPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 2
              If RankNum > 3 Then If Board(Target + SQ_UP) >= NO_PIECE Then If Not CBool(BAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
              If Offset = SQ_UP_LEFT Or Offset = SQ_UP_RIGHT Then WAttack(Target + Offset) = WAttack(Target + Offset) Or QXrayAttackBit
              If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              Exit Do    'Defends pawn
            Case BPAWN:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1: If AttackBit = QAttackBit Then AddThreat COL_BLACK, PT_PAWN, PT_QUEEN, Square, Target
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              SC.MG = SC.MG + 7: SC.EG = SC.EG + 7
              Exit Do   'Attack pawn
            Case BKNIGHT:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1: If AttackBit = QAttackBit Then AddThreat COL_BLACK, PT_KNIGHT, PT_QUEEN, Square, Target
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              If AttackBit = QAttackBit Then AddThreat COL_BLACK, PieceType(Board(Target)), PT_QUEEN, Square, Target
              Exit Do
            Case BBISHOP:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1: If AttackBit = QAttackBit Then AddThreat COL_BLACK, PT_BISHOP, PT_QUEEN, Square, Target
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              If AttackBit = QAttackBit Then AddThreat COL_BLACK, PieceType(Board(Target)), PT_QUEEN, Square, Target
              If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              Exit Do
            Case BROOK:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1: If AttackBit = QAttackBit Then AddThreat COL_BLACK, PT_ROOK, PT_QUEEN, Square, Target
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              If AttackBit = QAttackBit Then AddThreat COL_BLACK, PieceType(Board(Target)), PT_QUEEN, Square, Target
              If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              Exit Do
            Case WKING: Exit Do ' ignore
            Case BKING: MobCnt = MobCnt + 1
              Exit Do
            Case BQUEEN: If AttackBit = QAttackBit Then AddThreat COL_BLACK, PT_QUEEN, PT_QUEEN, Square, Target: MobCnt = MobCnt + 1
              Exit Do
            Case WBISHOP:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1: SC.MG = SC.MG + 4: SC.EG = SC.EG + 2
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              If i > 3 Then AttackBit = QXrayAttackBit Else Exit Do
            Case WKNIGHT:
              If Not CBool(BAttack(Target) And PNBRAttackBit) Then
                MobCnt = MobCnt + 1
              Else
                If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
              End If
              Exit Do
```

```
          Case WROOK:
            If Not CBool(BAttack(Target) And PNBRAttackBit) Then
              If Offset = 10 Then
                If WPawns(FileNum) = 0 Then
                  SC.MG = SC.MG + 10: SC.EG = SC.EG + 5
                ElseIf BPawns(FileNum) = 0 Then
                  SC.MG = SC.MG + 15: SC.EG = SC.EG + 5
                End If
              End If
              MobCnt = MobCnt + 1 '--- double lines
            Else
              If CBool(BAttack(Target) And RBAttackBit) Then CheckWQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
            End If
            If i < 4 Then AttackBit = QXrayAttackBit Else Exit Do
          Case WEP_PIECE, BEP_PIECE: If Not CBool(BAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
          Case Else:
            Exit Do
        End Select

        Target = Target + Offset
      Loop

    Next

    AddScore WMobility, MobilityQ(MobCnt)
    If CBool(BAttack(Square) And PAttackBit) Then AddPawnThreat BThreat, COL_WHITE, PieceType(Board(Square)), Square
    AddScoreWithFactor SC, KingProtector(PT_QUEEN), MaxDistance(Square, WKingLoc) ' defends king?
    If QueenWeak Then SC.MG = SC.MG - 50: SC.EG = SC.EG - 10
    AddScore WPos, SC
    If bEvalTrace Then WriteTrace "WQueen: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & WPos.MG & ", " & WPos.EG
  Next a

  '------------------------------------------------------------------------
  '---- BLACK QUEENs ( last - full attack info needed for mobility ) --
  '------------------------------------------------------------------------
  For a = 1 To PieceSqListCnt(BQUEEN)
    Square = PieceSqList(BQUEEN, a): FileNum = File(Square): RankNum = Rank(Square): RelRank = (9 - RankNum): SC.MG = 0: SC.EG = 0: QueenWeak = False
    BPos.MG = BPos.MG + PsqtBQ(Square).MG: BPos.EG = BPos.EG + PsqtBQ(Square).EG
    '--- Mobility
    MobCnt = 0

    For i = 0 To 7
      Offset = DirectionOffset(i): Target = Square + Offset: AttackBit = QAttackBit

      Do While Board(Target) <> FRAME
        BAttack(Target) = BAttack(Target) Or AttackBit

        Select Case Board(Target)
          Case NO_PIECE: If Not CBool(WAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
          Case BPAWN: SC.MG = SC.MG + 2: SC.EG = SC.EG + 2
            If RankNum < 6 Then If Board(Target + SQ_DOWN) >= NO_PIECE Then If Not CBool(WAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
            If Offset = SQ_DOWN_LEFT Or Offset = SQ_DOWN_RIGHT Then BAttack(Target + Offset) = BAttack(Target + Offset) Or QXrayAttackBit
            If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak ' pin oder discovered attack?
            Exit Do    'Defends pawn
          Case WPAWN:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1: If AttackBit = QAttackBit Then AddThreat COL_WHITE, PT_PAWN, PT_QUEEN, Square, Target
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            SC.MG = SC.MG + 7: SC.EG = SC.EG + 7
            Exit Do   'Attack pawn
          Case WKNIGHT:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            If AttackBit = QAttackBit Then AddThreat COL_WHITE, PieceType(Board(Target)), PT_QUEEN, Square, Target
            Exit Do
          Case WBISHOP:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            If AttackBit = QAttackBit Then AddThreat COL_WHITE, PieceType(Board(Target)), PT_QUEEN, Square, Target
            Exit Do
          Case WROOK:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            If AttackBit = QAttackBit Then AddThreat COL_WHITE, PieceType(Board(Target)), PT_QUEEN, Square, Target
            Exit Do
          Case BKING: Exit Do 'Ignore
          Case WKING: MobCnt = MobCnt + 1
            Exit Do
          Case WQUEEN:  If AttackBit = QAttackBit Then AddThreat COL_WHITE, PT_QUEEN, PT_QUEEN, Square, Target: MobCnt = MobCnt + 1
            Exit Do
          Case BBISHOP:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1: SC.MG = SC.MG + 4: SC.EG = SC.EG + 2
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            If i > 3 Then AttackBit = QXrayAttackBit Else Exit Do
          Case BKNIGHT:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              MobCnt = MobCnt + 1
            Else
              If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
            End If
            Exit Do
          Case BROOK:
            If Not CBool(WAttack(Target) And PNBRAttackBit) Then
              If Offset = -10 Then
                If BPawns(FileNum) = 0 Then
                  SC.MG = SC.MG + 10: SC.EG = SC.EG + 5
                ElseIf WPawns(FileNum) = 0 Then
                  SC.MG = SC.MG + 15: SC.EG = SC.EG + 5
                End If
              End If
              MobCnt = MobCnt + 1
```

```
              Else
                If CBool(WAttack(Target) And RBAttackBit) Then CheckBQueenWeek Target, Offset, i, QueenWeak 'pin oder discovered attack?
              End If
              If i < 4 Then AttackBit = QXrayAttackBit Else Exit Do
            Case WEP_PIECE, BEP_PIECE: If Not CBool(WAttack(Target) And PNBRAttackBit) Then MobCnt = MobCnt + 1
            Case Else:
              Exit Do
          End Select
          Target = Target + Offset
        Loop
      Next

      AddScore BMobility, MobilityQ(MobCnt)
      If CBool(WAttack(Square) And PAttackBit) Then AddPawnThreat WThreat, COL_BLACK, PieceType(Board(Square)), Square
      AddScoreWithFactor SC, KingProtector(PT_QUEEN), MaxDistance(Square, BKingLoc) ' defends king?
      If QueenWeak Then SC.MG = SC.MG - 50: SC.EG = SC.EG - 10
      AddScore BPos, SC
      If bEvalTrace Then WriteTrace "BQueen: " & LocCoord(Square) & ">" & SC.MG & ", " & SC.EG & " / " & BPos.MG & ", " & BPos.EG
    Next a

    '------------------------------------------------------------------
    '---- Step 3.: Pass for pawn push ( full attack info needed for mobility )
    '------------------------------------------------------------------
    SC = ZeroScore

    For a = 1 To PieceSqListCnt(WPAWN)
      Square = PieceSqList(WPAWN, a): RelRank = Rank(Square)

      ' bonus if safe pawn push attacks an enemy piece
      For rr = 1 To 1 + Abs(RelRank = 2)
        Target = Square + SQ_UP * rr
        If Board(Target) >= NO_PIECE Then ' empty or ep-dummy piece
          SC.MG = SC.MG + 8: SC.EG = SC.EG + 8 ' pawn mobility
          ' Safe pawn push: push field not attacked by opp pawn AND defend by own piece or not attacked by opp
          If BAttack(Target) = 0 Or WAttack(Target) > 0 Then
            If Not (rr = 2 And CBool(BAttack(Square + SQ_UP) And PAttackBit)) Then ' check EnPassant capture

              For i = 9 To 11 Step 2
                r = Board(Target + i)
                If PieceColor(r) = COL_BLACK And r <> BPAWN Then
                  If Not CBool(WAttack(Target + i) And PAttackBit) Then ' already attacked by own pawn?
                    SC.MG = SC.MG + 38: SC.EG = SC.EG + 22 ' pawn threats non pawn enemy
                  End If
                End If
              Next i

            End If
          End If
        Else
          Exit For
        End If
      Next
    Next a

    If SC.MG > 0 Then AddScore WPos, SC
    SC = ZeroScore

    For a = 1 To PieceSqListCnt(BPAWN)
      Square = PieceSqList(BPAWN, a): RelRank = (9 - Rank(Square))

      ' bonus if safe pawn push attacks an enemy piece
      For rr = 1 To 1 + Abs(RelRank = 2)
        Target = Square + SQ_DOWN * rr
        If Board(Target) >= NO_PIECE Then
          SC.MG = SC.MG + 8: SC.EG = SC.EG + 8 ' pawn mobility
          ' Safe pawn push: push field not attacked by opp pawn AND defend by own piece and not attacked by opp
          If WAttack(Target) = 0 Or BAttack(Target) > 0 Then
            If Not (rr = 2 And CBool(WAttack(Square + SQ_DOWN) And PAttackBit)) Then ' check EnPassant capture

              For i = 9 To 11 Step 2
                r = Board(Target - i)
                If PieceColor(r) = COL_WHITE And r <> WPAWN Then
                  If Not CBool(BAttack(Target - i) And PAttackBit) Then ' already attacked by own pawn?
                    SC.MG = SC.MG + 38: SC.EG = SC.EG + 22 ' pawn threats non pawn enemy
                  End If
                End If
              Next i

            End If
          End If
        Else
          Exit For
        End If
      Next rr
    Next a

    If SC.MG > 0 Then AddScore BPos, SC
    '--- End pass for pawn push <<<<


    '----------------------------------------------
    '--- Step 4:  King Safety   -------------------
    '----------------------------------------------
    If bEndgame Then
      WKSafety = ZeroScore: BKSafety = ZeroScore
    Else
      Dim Bonus             As Long
      Dim KingOnlyDefended As Long, bSafe As Boolean, Tropism As Long
      '----------------------------------------------
      '--- White King Safety Eval -------------------
      '----------------------------------------------
      RankNum = Rank(WKingLoc): FileNum = WKingFile: Bonus = 0
      If (PieceCnt(BQUEEN) * 2 + PieceCnt(BROOK)) > 1 Then
        KingDanger = 0
        If WPawnCnt = 0 Then MinWKingPawnDistance = 0 Else MinWKingPawnDistance = MinWKingPawnDistance - 1
        If RankNum > 4 Then
          WKSafety.EG = WKSafety.EG - 16 * MinWKingPawnDistance
        Else
          Bonus = WKingShelterStorm(WKingLoc)
          If WhiteCastled = NO_CASTLE Then
            If WKingLoc = SQ_E1 Then
              If WPawns(7) > 0 And PawnsWMin(7) < 4 Then
                If WCanCastleOO() Then
                  Bonus = GetMax(Bonus, WKingShelterStorm(SQ_G1))
                End If
              End If
              If (WPawns(3) > 0 And PawnsWMin(3) < 4) Or (WPawns(2) > 0 And PawnsWMin(2) < 4) Then
                If WCanCastleOOO() Then
                  Bonus = GetMax(Bonus, WKingShelterStorm(SQ_C1))
                End If
              End If
            End If
          End If
          AddScoreVal WKSafety, Bonus, -16 * MinWKingPawnDistance
        End If
        If bDoWKSafety Then

          ' King tropism: firstly, find squares that opponent attacks in our king flank
          ' Secondly, add the squares which are attacked twice in that flank
            GetKingFlankFiles WKingLoc, r, rr: Tropism = 0
            For k = SQ_A1 - 1 To SQ_A1 - 1 + 40 Step 10 ' start square - 1 of rank 1-5 (camp)
              For Square = k + r To k + rr       ' files king flank
                If BAttack(Square) <> 0 Then
                  Tropism = Tropism + 1: If AttackBitCnt(BAttack(Square)) > 1 Then
                  Tropism = Tropism + 1  ' Attacked twice?
                End If
              Next
            Next

            ' Pawnless king flank penalty
            k = 0
            For i = r To rr
              If WPawns(i) + BPawns(i) > 0 Then k = 1: Exit For
            Next
            If k = 0 Then MinusScore WKSafety, PawnlessFlank


            '--- Check threats at king ring
            Undefended = 0: KingOnlyDefended = 0: WKingAttPieces = 0: KingLevers = 0
            '  add the 2 or 3 squares in front of king ring: king G1 => F3+G3+H3
            If RankNum = 1 Then
              For Target = WKingLoc + 19 To WKingLoc + 21
                If Board(Target) <> FRAME Then
                  If BAttack(Target) <> 0 Then
                    If WAttack(Target) = 0 Or WAttack(Target) = QAttackBit Then
                    Undefended = Undefended + 1
                    ' exclude double pawn defended squares
                    If AttackBitCnt(WAttack(Target) And PAttackBit) < 2 Then
                    WKingAttPieces = WKingAttPieces Or BAttack(Target)
                    If Board(Target) = WPAWN Then
                      If CBool(BAttack(Target) And PAttackBit) Then KingLevers =
                      KingLevers + 1
                    End If
                  End If
                End If
              Next
            End If

            For i = 0 To 7 ' for all directions from king square
              Offset = DirectionOffset(i): Target = WKingLoc + Offset
              If Board(Target) <> FRAME Then
                If BAttack(Target) <> 0 Then
                  ' King attacks are added later in attack array, so distance=1 and WAttack=0 is equal to king
                  attack only
                  If WAttack(Target) = 0 Then KingOnlyDefended = KingOnlyDefended + 1
                  WKingAdjacentZoneAttCnt = WKingAdjacentZoneAttCnt + AttackBitCnt(
                  BAttack(Target) And Not PAttackBit)
                  ' exclude double pawn defended squares
                  If AttackBitCnt(WAttack(Target) And PAttackBit) < 2 Then
                  WKingAttPieces = WKingAttPieces Or BAttack(Target)
                  If Board(Target) = WPAWN Then
                    If CBool(BAttack(Target) And PAttackBit) Then KingLevers =
                    KingLevers + 1
                  End If
                End If
                WKDefender = WKDefender Or WAttack(Target)
                rr = 1 ' rr=Distance to King

                Do  ' loop for a direction
                  r = BAttack(Target)
                  If CBool(r And QRBAttackBit) Then
                    bSafe = False ' Safe attack square?
                    If PieceColor(Board(Target)) <> BCOL Then
```

```
  If WAttack(Target) = 0 Then
    If rr = 1 Then
      If AttackBitCnt(BAttack(Target)) > 1 Then bSafe = True
    Else
      bSafe = True
    End If
  End If
End If
' Queen safe checks
If bSafe Then
  If CBool(r And QAttackBit) Then
    If Not CBool(WChecksCounted And QAttackBit) Then
      KingDanger = KingDanger + QueenCheck
      WChecksCounted = (WChecksCounted Or QAttackBit)
    End If
  End If
End If
If CBool(r And RBOrXrayAttackBit) Then
  If Not bSafe And rr > 1 Then ' not defended by king
    ' For minors and rooks, also consider the square as safe if attacked twice,
    ' and only defended by our queen.
    If CBool(WAttack(Target) = QAttackBit) Then
      If AttackBitCnt(BAttack(Target)) > 1 Then
        If Not (AttackBitCnt(WAttack(Target)) > 1 Or PieceColor(Board(Target)) = BCOL) Then
          bSafe = True
        End If
      End If
    End If
  End If
  '(i=0-3: orthogonal offset, 4-7:diagonal)
  ' Rook checks
  If i < 4 Then
    If CBool(r And ROrXrayAttackBit) Then ' R1Attackbit or R2Attackbit set, if 2 rooks only one is counted per square
      If bSafe Then
        ' look for both rooks, different to SF
        If CBool(r And R1OrXrayAttackBit) Then
         If Not CBool(WChecksCounted And R1AttackBit) Then ' count only once per square!
           If CBool(r And R1XrayAttackBit) Then
             If SqBetween(Target, PieceSqList(BROOK, 1), WKingLoc) Then ' xray attack only if in direct line to opp king
               KingDanger = KingDanger + RookCheck \ 3: WChecksCounted = (WChecksCounted Or R1AttackBit)
             Else
               KingDanger = KingDanger + 20 ' may be an attack plan
             End If
           Else
             KingDanger = KingDanger + RookCheck: WChecksCounted = (WChecksCounted Or R1AttackBit)
           End If
         End If
        End If
        If CBool(r And R2OrXrayAttackBit) Then
         If Not CBool(WChecksCounted And R2AttackBit) Then ' count only once per square!
           If CBool(r And R2XrayAttackBit) Then
             If SqBetween(Target, PieceSqList(BROOK, 2), WKingLoc) Then ' xray attack only if in direct line to opp king
               KingDanger = KingDanger + RookCheck \ 3: WChecksCounted = (WChecksCounted Or R2AttackBit)
             Else
               KingDanger = KingDanger + 20 ' may be an attack plan
             End If
           Else
             KingDanger = KingDanger + RookCheck: WChecksCounted = (WChecksCounted Or R2AttackBit)
             End If
           End If
          End If
        Else
          WUnsafeChecks = WUnsafeChecks + 1
        End If
      End If
    Else 'i >= 4
      ' Bishop checks
      If CBool(r And BXrayAttackBit) Then ' B1Attackbit or B2Attackbit set, if 2 rooks only one is counted
        If Not CBool(WChecksCounted And B1AttackBit) Then ' count only once! only one bishop has same color as king
          If bSafe Then
             If CBool(r And BXrayAttackBit) Then
               If SqBetween(Target, PieceSqList(BBISHOP, 1), WKingLoc) Or _
                  SqBetween(Target, PieceSqList(BBISHOP, 2), WKingLoc) Then   ' xray attack only if in direct line to opp king
                  ' do not count xray if through a blocked pawn
                  If Board(Target + Offset) <> BPAWN Or Board(Target + Offset + SQ_DOWN) >= NO_PIECE Then KingDanger = KingDanger + BishopCheck \ 3: WChecksCounted = (WChecksCounted Or B1AttackBit)
               Else
                 KingDanger = KingDanger + 10 ' may be an attack plan
               End If
             Else
               KingDanger = KingDanger + BishopCheck: WChecksCounted = (WChecksCounted Or B1AttackBit)
             End If
          Else
            WUnsafeChecks = WUnsafeChecks + 1
          End If
        End If
      End If
    End If
  End If
End If ' r and QRBAttackbit
If Board(Target) < NO_PIECE Then ' Piece found
  '
  '--- Check for pinned pieces
  '
  If (Board(Target) And 1) = WCOL Then ' own piece, look for pinned
    If i < 4 Then ' orthogonal
      If CBool(BAttack(Target) And QRAttackBit) Then  ' rook or queen, direction not clear
        For k = 1 To 7
          sq = Target + Offset * k: Piece = Board(sq)
          If Piece = FRAME Then Exit For
          If Piece < NO_PIECE Then
            If Piece = BQUEEN Or Piece = BROOK Then
              If (PieceType(Piece) <> PieceType(Board(Target))) Then
                If Piece = BROOK And Board(Target) = WQUEEN Then
                  If bWhiteToMove Then
                    AddScoreVal BThreat, 30, 50
                    If BAttack(sq) <> 0 And WAttack(sq) = QAttackBit Then
                      AddScoreVal BThreat, 75, 100 ' attacker defended? less because may be blocker move?
                      If MaxDistance(Target, sq) = 1 Then AddScoreVal BThreat, 400, 500 ' no blocker option
                    End If
                  Else
                    AddScoreVal BThreat, 1200, 1400
                  End If
                End If
                WPinnedCnt = WPinnedCnt + 1
```

```
                    ' if pinned pawn then add bonus for attacker
                    If Board(Target) = WPAWN Then AddScoreVal BPos, ThreatByRank.MG \ 2 * Rank(Target), ThreatByRank.EG \ 2 * Rank(Target)
                  End If
                End If
                Exit For
              Else
                If Not (CBool(BAttack(sq) And QRAttackBit)) Then Exit For
              End If
            Next k
          End If
        Else ' i>4 diagonal
          If CBool(BAttack(Target) And QBAttackBit) Then  ' bishop or queen, direction not clear

            For k = 1 To 7
              sq = Target + Offset * k: Piece = Board(sq)
              If Piece = FRAME Then Exit For
              If Piece < NO_PIECE Then
                If Piece = BQUEEN Or Piece = BBISHOP Then
                  If (PieceType(Piece) <> PieceType(Board(Target))) Then
                    WPinnedCnt = WPinnedCnt + 1
                    If Piece = BBISHOP And Board(Target) = WQUEEN Then
                      If bWhiteToMove Then
                        AddScoreVal BThreat, 50, 70
                        If BAttack(sq) <> 0 And WAttack(sq) = QAttackBit Then
                          AddScoreVal BThreat, 100, 130  ' attacker defended? less because may be blocker move?
                          If MaxDistance(Target, sq) = 1 Then AddScoreVal BThreat, 400, 500 ' no blocker option
                        End If
                      Else
                        AddScoreVal BThreat, 1300, 1500
                      End If
                    End If
                    ' if pinned pawn then add bonus for attacker (if pawn cannot capture attacker = distance>1)
                    If Board(Target) = WPAWN Then If MaxDistance(Target, sq) > 1 Or Offset < 0 Then AddScoreVal BPos, ThreatByRank.MG \ 2 * Rank(Target), ThreatByRank.EG \ 2 * Rank(Target)
                  End If
                End If
                Exit For
              Else
                If Not (CBool(BAttack(sq) And QBAttackBit)) Then Exit For
              End If
            Next k

          End If
        End If
      End If
      ' --- Piece found - exit direction loop
      If Board(Target) <> WQUEEN Then Exit Do ' threat Q+K
    End If
    Target = Target + Offset: rr = rr + 1
  Loop While Board(Target) <> FRAME

End If ' <<< Board(Target) <> FRAME

' Knight Check
If PieceCnt(BKNIGHT) > 0 Then
  Target = WKingLoc + KnightOffsets(i)
  If Board(Target) <> FRAME Then
    If CBool(BAttack(Target) And NAttackBit) Then
      bSafe = False ' Safe attack square?
```

```
        If PieceColor(Board(Target)) <> BCOL Then If WAttack(Target) = 0 Then bSafe = True
        If Not bSafe Then
          If CBool(WAttack(Target) = QAttackBit) Then
            If AttackBitCnt(BAttack(Target)) > 1 Then
              If Not (AttackBitCnt(WAttack(Target)) > 1 Or PieceColor(Board(Target)) = BCOL) Then bSafe = True
            End If
          End If
        End If
        If Not CBool(WChecksCounted And N1AttackBit) Then ' count only once per square!
          If bSafe Then
            KingDanger = KingDanger + KnightCheck: WChecksCounted = (WChecksCounted Or N1AttackBit) ' only one knight check expected, two are very rare
          Else
            WUnsafeChecks = WUnsafeChecks + 1
          End If
        End If
        ' Knight check fork?
        If WAttack(Target) = 0 Or (WAttack(Target) = QAttackBit And (BAttack(Target) <> NAttackBit)) Then ' no attack

         If PieceCnt(WQUEEN) + PieceCnt(WROOK) > 0 Then
          For k = 0 To 7

            Select Case Board(Target + KnightOffsets(k))
              Case WQUEEN: AddScoreVal BThreat, 25, 35
              Case WROOK: AddScoreVal BThreat, 15, 20
            End Select

          Next
         End If
        End If
      End If '<<< CBool(BAttack(Target) And NAttackBit)
    End If '<<<  Board(Target) <> FRAME
  End If '<<< PieceCnt(BKNIGHT) > 0
Next i '<<< direction

If WKingAttPieces <> 0 Then AddWKingAttackers WKingAttPieces

If WKingAttackersCount > 1 - PieceCnt(BQUEEN) Then

  ' total KingDanger
  KingDanger = KingDanger + WKingAttackersCount * WKingAttackersWeight _
               + 65 * WKingAdjacentZoneAttCnt + Abs(KingLevers > 0) * 64 _
               + 190 * (KingOnlyDefended + Undefended) _
               - 100 * Abs(CBool(WKDefender And NAttackBit)) _
               - 40 * Abs(CBool(WKDefender And BAttackBit)) _
               + 152 * (WPinnedCnt + WUnsafeChecks) _
               - 885 * Abs(PieceCnt(BQUEEN) = 0) _
               - 6 * Bonus \ 8 _
               + 5 * Tropism * Tropism \ 16 _
               + (BMobility.MG - WMobility.MG) - 10

  ' Penalty for king on open or semi-open file
  If NonPawnMaterial > 9000 And WPawns(FileNum) = 0 And WKingLoc <> WKING_START Then
    If BPawns(FileNum) = 0 Then KingDanger = KingDanger + 18 Else KingDanger = KingDanger + 9
  End If
  r = KingDanger + BPassedPawnAttack * 8 ' passed pawn attacking king?
  If r > 100 Then
    WKSafety.MG = WKSafety.MG - (r * r) \ 4096
    WKSafety.EG = WKSafety.EG - r \ 16
  End If
End If
```

```

        End If


        ' Bonus for a dangerous pawn in the center near the opponent king, for instance pawn e5 against king g8.
        If FileNum >= 4 Then If Board(SQ_E4) = BPAWN Then WKSafety.MG = WKSafety.MG - 5
        If FileNum <= 5 Then If Board(SQ_D4) = BPAWN Then WKSafety.MG = WKSafety.MG - 5 ' both possible if king centered


        ' King tropism bonus, to anticipate slow motion attacks on our king
        WKSafety.MG = WKSafety.MG - 7 * Tropism ' closeEnemies

    End If


    '---------------------------------------------
    '--- Black King Safety Eval ------------------
    '---------------------------------------------
    RankNum = Rank(BKingLoc): RelRank = (9 - RankNum): FileNum = BKingFile: Bonus = 0: KingLevers = 0
    If (PieceCnt(WQUEEN) * 2 + PieceCnt(WROOK)) > 1 Then
      KingDanger = 0
      If BPawnCnt = 0 Then MinBKingPawnDistance = 0 Else MinBKingPawnDistance = MinBKingPawnDistance - 1
      If RelRank > 4 Then
        BKSafety.EG = BKSafety.EG - 16 * MinBKingPawnDistance
      Else
        Bonus = BKingShelterStorm(BKingLoc)
        If BlackCastled = NO_CASTLE Then
          If BKingLoc = SQ_E8 Then
            If BPawns(7) > 0 And PawnsBMax(7) > 5 Then
              If BCanCastleOO() Then
                Bonus = GetMax(Bonus, BKingShelterStorm(SQ_G8))
              End If
            End If
            If (BPawns(3) > 0 And PawnsBMax(3) > 5) Or (BPawns(2) > 0 And PawnsBMax(2) > 5) Then
              If BCanCastleOOO() Then
                Bonus = GetMax(Bonus, BKingShelterStorm(SQ_C8))
              End If
            End If
          End If
        End If
        AddScoreVal BKSafety, Bonus, -16 * MinBKingPawnDistance
      End If
      If bDoBKSafety Then

          ' King tropism: firstly, find squares that opponent attacks in our king flank
          ' Secondly, add the squares which are attacked twice in that flank
          GetKingFlankFiles BKingLoc, r, rr: Tropism = 0
          For k = SQ_A1 - 1 + 30 To SQ_A1 - 1 + 70 Step 10 ' start square - 1 of rank 5-8
            For Square = k + r To k + rr      ' files king flank
              If WAttack(Square) <> 0 Then
                Tropism = Tropism + 1: If AttackBitCnt(WAttack(Square)) > 1 Then Tropism = Tropism + 1  ' Attacked twice?
              End If
            Next
          Next

          ' Pawnless king flank penalty
          k = 0
          For i = r To rr
            If WPawns(i) + BPawns(i) > 0 Then k = 1: Exit For
          Next
          If k = 0 Then MinusScore BKSafety, PawnlessFlank

```

```
'--- Check threats at king ring
Undefended = 0: KingOnlyDefended = 0: BKingAttPieces = 0
' add the 2 or 3 squares in front of king ring: king G8 => F6+G6+H6
If RankNum = 8 Then

  For Target = BKingLoc - 21 To BKingLoc - 19
    If Board(Target) <> FRAME Then
      If WAttack(Target) <> 0 Then
        If BAttack(Target) = 0 Or BAttack(Target) = QAttackBit Then
        Undefended = Undefended + 1
        ' exclude double pawn defended squares
        If AttackBitCnt(BAttack(Target) And PAttackBit) < 2 Then
        BKingAttPieces = BKingAttPieces Or WAttack(Target)
        If Board(Target) = BPAWN Then
          If CBool(WAttack(Target) And PAttackBit) Then KingLevers =
          KingLevers + 1
        End If
      End If
    End If
  Next

End If

For i = 0 To 7
  Offset = DirectionOffset(i): Target = BKingLoc + Offset
  If Board(Target) <> FRAME Then
    If WAttack(Target) <> 0 Then
      ' King attacks are added later in attack array, so distance=1 and BAttack=0 is equal to king
      attack only
      If BAttack(Target) = 0 Then KingOnlyDefended = KingOnlyDefended + 1
      BKingAdjacentZoneAttCnt = BKingAdjacentZoneAttCnt + AttackBitCnt(
      WAttack(Target) And Not PAttackBit)
      ' exclude double pawn defended squares
      If AttackBitCnt(BAttack(Target) And PAttackBit) < 2 Then
      BKingAttPieces = BKingAttPieces Or WAttack(Target)
      If Board(Target) = BPAWN Then
        If CBool(WAttack(Target) And PAttackBit) Then KingLevers =
        KingLevers + 1
      End If
    End If
    BKDefender = BKDefender Or BAttack(Target)
    rr = 1 ' rr=Distance to King

    Do ' loop for a direction
      r = WAttack(Target)
      If CBool(r And QRBAttackBit) Then
        bSafe = False ' Safe attack square?
        If PieceColor(Board(Target)) <> WCOL Then
          If BAttack(Target) = 0 Then
            If rr = 1 Then
              If AttackBitCnt(WAttack(Target)) > 1 Then bSafe = True
            Else
              bSafe = True
            End If
          End If
        End If
        ' Queen safe checks
        If bSafe Then
          If CBool(r And QAttackBit) Then
            If Not CBool(BChecksCounted And QAttackBit) Then
              KingDanger = KingDanger + QueenCheck
              BChecksCounted = (BChecksCounted Or QAttackBit)
            End If
          End If
        End If
        If CBool(r And RBOrXrayAttackBit) Then
          If Not bSafe And rr > 1 Then ' not defended by king
            ' For minors and rooks, also consider the square as safe if attacked twice,
            ' and only defended by our queen.
            If CBool(BAttack(Target) = QAttackBit) Then
              If AttackBitCnt(WAttack(Target)) > 1 Then
                If Not (AttackBitCnt(BAttack(Target)) > 1 Or PieceColor(Board(Target)) = WCOL) Then
                  bSafe = True
                End If
              End If
            End If
          End If
          '(i=0-3: orthogonal offset, 4-7:diagonal)
          ' Rook checks
          If i < 4 Then
            If CBool(r And ROrXrayAttackBit) Then ' R1Attackbit or R2Attackbit set, if 2 rooks only one is counted per square
              If bSafe Then
                ' look for both rooks, different to SF
                If CBool(r And R1OrXrayAttackBit) Then
                 If Not CBool(BChecksCounted And R1AttackBit) Then ' count only once per square!
                   If CBool(r And R1XrayAttackBit) Then
                     If SqBetween(Target, PieceSqList(WROOK, 1), BKingLoc) Then ' xray attack only if in direct line to opp king
                       KingDanger = KingDanger + RookCheck \ 3: BChecksCounted = (BChecksCounted Or R1AttackBit)
                     Else
                       KingDanger = KingDanger + 20 ' may be an attack plan
                     End If
                   Else
                     KingDanger = KingDanger + RookCheck: BChecksCounted = (BChecksCounted Or R1AttackBit)
                   End If
                 End If
                End If
                '
                If CBool(r And R2OrXrayAttackBit) Then
                 If Not CBool(BChecksCounted And R2AttackBit) Then ' count only once per square!
                   If CBool(r And R2XrayAttackBit) Then
                     If SqBetween(Target, PieceSqList(WROOK, 2), BKingLoc) Then ' xray attack only if in direct line to opp king
                       KingDanger = KingDanger + RookCheck \ 3: BChecksCounted = (BChecksCounted Or R2AttackBit)
                     Else
                       KingDanger = KingDanger + 20 ' may be an attack plan
                     End If
                   Else
                     KingDanger = KingDanger + RookCheck: BChecksCounted = (BChecksCounted Or R2AttackBit)
                   End If
                 End If
                End If
              Else
                BUnsafeChecks = BUnsafeChecks + 1
              End If
            End If
          Else ' i >= 4
            ' Bishop checks
            If CBool(r And BXrayAttackBit) Then ' B1Attackbit or B2Attackbit set, if 2 rooks only one is counted
              If Not CBool(BChecksCounted And B1AttackBit) Then ' count only once! only one bishop has same color as king
                If bSafe Then
                   If CBool(r And BXrayAttackBit) Then
                     If SqBetween(Target, PieceSqList(WBISHOP, 1), BKingLoc) Or _
                        SqBetween(Target, PieceSqList(WBISHOP, 2), BKingLoc) Then  ' xray attack only if in direct line to opp king
```

```
                ' do not count xray if through a blocked pawn
                If Board(Target + Offset) <> WPAWN Or Board(Target + Offset + SQ_UP) >= NO_PIECE Then KingDanger = KingDanger + BishopCheck \ 3: BChecksCounted = (BChecksCounted Or B1AttackBit)
              Else
                KingDanger = KingDanger + 10 ' may be an attack plan
              End If
            Else
              KingDanger = KingDanger + BishopCheck: BChecksCounted = (BChecksCounted Or B1AttackBit)
            End If
          Else
            BUnsafeChecks = BUnsafeChecks + 1
          End If
        End If
      End If
    End If
  End If
End If ' r and QRBAttackbit
If Board(Target) < NO_PIECE Then ' Piece found
  '
  '--- Check for pinned pieces
  '
  If (Board(Target) And 1) = BCOL Then ' own piece
    If i < 4 Then ' orthogonal
      If CBool(WAttack(Target) And QRAttackBit) Then  ' rook or queen, direction not clear
        For k = 1 To 7
          sq = Target + Offset * k: Piece = Board(sq)
          If Piece = FRAME Then Exit For
          If Piece < NO_PIECE Then
            If Piece = WQUEEN Or Piece = WROOK Then
              If (PieceType(Piece) <> PieceType(Board(Target))) Then
                If Piece = WROOK And Board(Target) = BQUEEN Then
                  If Not bWhiteToMove Then
                    AddScoreVal WThreat, 30, 50
                    If WAttack(sq) <> 0 And BAttack(sq) = QAttackBit Then
                      AddScoreVal WThreat, 75, 100 ' attacker defended? less because may be blocker move?
                      If MaxDistance(Target, sq) = 1 Then AddScoreVal WThreat, 400, 500 ' no blocker option
                    End If
                  Else
                    AddScoreVal WThreat, 1200, 1400
                  End If
                End If
               BPinnedCnt = BPinnedCnt + 1
                ' if pinned pawn then add bonus for attacker
                If Board(Target) = BPAWN Then AddScoreVal WPos, ThreatByRank.MG \ 2 * (9 - Rank(Target)), ThreatByRank.EG \ 2 * (9 - Rank(Target))
              End If
            End If
            Exit For
          Else
            If Not CBool(WAttack(sq) And QRAttackBit) Then Exit For
          End If
        Next k
      End If
    Else ' i>4 diagonal
      If CBool(WAttack(Target) And QBAttackBit) Then  ' bishop or queen, direction not clear

        For k = 1 To 7
          sq = Target + Offset * k: Piece = Board(sq)
          If Piece = FRAME Then Exit For
```

```
              If Piece < NO_PIECE Then
                If Piece = WQUEEN Or Piece = WBISHOP Then
                  If (PieceType(Piece) <> PieceType(Board(Target))) Then
                    BPinnedCnt = BPinnedCnt + 1
                    If Piece = WBISHOP And Board(Target) = BQUEEN Then
                      If Not bWhiteToMove Then
                        AddScoreVal WThreat, 50, 70
                        If WAttack(sq) <> 0 And BAttack(sq) = QAttackBit Then
                          AddScoreVal WThreat, 100, 130 ' attacker defended? less because may be blocker move?
                          If MaxDistance(Target, sq) = 1 Then AddScoreVal WThreat, 400, 500 ' no blocker option
                        End If
                      Else
                        AddScoreVal WThreat, 1300, 1500
                      End If
                    End If
                    ' if pinned pawn then add bonus for attacker (if pawn cannot capture attacker = distance>1)
                    If Board(Target) = BPAWN Then If MaxDistance(Target, sq) > 1 Or Offset > 0 Then AddScoreVal WPos, ThreatByRank.MG \ 2 * (9 - Rank(Target)), ThreatByRank.EG \ 2 * (9 - Rank(Target))
                  End If
                End If
                Exit For
              Else
                If Not CBool(WAttack(sq) And QBAttackBit) Then Exit For
              End If
            Next k

          End If
        End If
      End If
      ' --- Piece found - exit direction loop
      If Board(Target) <> BQUEEN Then Exit Do
    End If
    Target = Target + Offset: rr = rr + 1
  Loop While Board(Target) <> FRAME

End If ' <<< Board(Target) <> FRAME
' Knight Check
If PieceCnt(WKNIGHT) > 0 Then
  Target = BKingLoc + KnightOffsets(i)
  If Board(Target) <> FRAME Then
    If CBool(WAttack(Target) And NAttackBit) Then
      bSafe = False ' Safe attack square?
      If PieceColor(Board(Target)) <> WCOL Then If BAttack(Target) = 0 Then bSafe = True
      If Not bSafe Then
        If CBool(BAttack(Target) = QAttackBit) Then
          If AttackBitCnt(WAttack(Target)) > 1 Then
            If Not (AttackBitCnt(BAttack(Target)) > 1 Or PieceColor(Board(Target)) = WCOL) Then
              bSafe = True
            End If
          End If
        End If
      End If
      If Not CBool(BChecksCounted And N1AttackBit) Then ' count only once per square!
        If bSafe Then
          KingDanger = KingDanger + KnightCheck: BChecksCounted = (BChecksCounted Or N1AttackBit) ' only one knight check expected, two are very rare
        Else
          BUnsafeChecks = BUnsafeChecks + 1
```

```
                  End If
                End If
                ' Knight check fork?
                If BAttack(Target) = 0 Or (BAttack(Target) = QAttackBit And (WAttack(Target) <> NAttackBit)) Then ' field not defended or by queen only but other attacker

                 If PieceCnt(BQUEEN) + PieceCnt(BROOK) > 0 Then
                  For k = 0 To 7

                    Select Case Board(Target + KnightOffsets(k))
                      Case BQUEEN: AddScoreVal WThreat, 25, 35
                      Case BROOK: AddScoreVal WThreat, 15, 20
                    End Select

                  Next
                 End If

                End If
              End If  '<<< CBool(WAttack(Target) And NAttackBit)
            End If '<<< Board(Target) <> FRAME
          End If '<<< PieceCnt(WKNIGHT) > 0
        Next i '<<< direction

        If BKingAttPieces <> 0 Then AddBKingAttackers BKingAttPieces

        If BKingAttackersCount > 1 - PieceCnt(WQUEEN) Then


          ' total KingDanger
          KingDanger = KingDanger + BKingAttackersCount * BKingAttackersWeight _
                       + 65 * BKingAdjacentZoneAttCnt + Abs(KingLevers > 0) * 64 _
                       + 190 * (KingOnlyDefended + Undefended) _
                       - 100 * Abs(CBool(BKDefender And NAttackBit)) _
                       - 40 * Abs(CBool(BKDefender And BAttackBit)) _
                       + 152 * (BPinnedCnt + BUnsafeChecks) _
                       - 885 * Abs(PieceCnt(WQUEEN) = 0) _
                       - 6 * Bonus \ 8 _
                       + 5 * Tropism * Tropism \ 16 _
                       + (WMobility.MG - BMobility.MG) - 10

          ' Penalty for king on open or semi-open file
          If NonPawnMaterial > 9000 And BPawns(FileNum) = 0 And BKingLoc <> BKING_START Then
            If WPawns(FileNum) = 0 Then KingDanger = KingDanger + 18 Else KingDanger = KingDanger + 9
          End If
          r = KingDanger + WPassedPawnAttack * 8 ' passed pawn attacking king?
          If r > 100 Then
            BKSafety.MG = BKSafety.MG - (r * r) \ 4096
            BKSafety.EG = BKSafety.EG - r \ 16
          End If
        End If

      End If

      ' Bonus for a dangerous pawn in the center near the opponent king, for instance pawn e5 against king g8.
      If FileNum >= 4 Then If Board(SQ_E5) = WPAWN Then BKSafety.MG = BKSafety.MG - 5
      If FileNum <= 5 Then If Board(SQ_D5) = WPAWN Then BKSafety.MG = BKSafety.MG - 5 ' both possible if king centered


      ' King tropism bonus, to anticipate slow motion attacks on our king
      BKSafety.MG = BKSafety.MG - 7 * Tropism ' closeEnemies

    End If
End If ' Endgame
```

```


'--- Endgame King distance to best pawn. Not in PawnHash because "Fifty" may be different
If bEndgame Or (WPawnCnt + BPawnCnt <= 8) Then
  If WBestPawn > 0 Then
    i = MaxDistance(WBestPawn, WKingLoc)
    AddScoreVal WPos, 0, (7 - i) * (7 - i) * 6
    If Rank(WBestPawn) >= 5 Then AddScoreVal WPos, 0, ((Rank(WBestPawn - 4) * Rank(
    WBestPawn - 4)) * (Fifty + 1)) \ 3 * 2 '--- bonus for move pawn in endgame
  ElseIf BBestPawn > 0 Then
    i = MaxDistance(BBestPawn, WKingLoc)      '--- Close to Opp Pawn
    AddScoreVal WPos, 0, (7 - i) * (7 - i)
  End If
  If BBestPawn > 0 Then
    i = MaxDistance(BBestPawn, BKingLoc)
    If i > 2 Then AddScoreVal BPos, 0, (7 - i) * (7 - i) * 6
    If RankB(BBestPawn) >= 5 Then AddScoreVal BPos, 0, ((RankB(BBestPawn - 4) *
    RankB(BBestPawn - 4)) * (Fifty + 1)) \ 3 * 2 '--- bonus for move pawn in endgame
  ElseIf WBestPawn > 0 Then
    i = MaxDistance(WBestPawn, BKingLoc)      '--- Close to Opp Pawn
    AddScoreVal BPos, 0, (7 - i) * (7 - i)
  End If
Else
  '--- Midgame
End If
' add kings to attack array
r = 0: rr = 0

For i = 0 To 7
  Offset = DirectionOffset(i)
  Target = WKingLoc + Offset
  If Board(Target) <> FRAME Then
    WAttack(Target) = WAttack(Target) Or KAttackBit
    If PieceColor(Board(Target)) = COL_BLACK Then
      If AttackBitCnt(WAttack(Target)) > AttackBitCnt(BAttack(Target)) Then r = r +
      1  ' King attacks unprotected piece
    End If
  End If
  Target = BKingLoc + Offset
  If Board(Target) <> FRAME Then
    BAttack(Target) = BAttack(Target) Or KAttackBit
    If PieceColor(Board(Target)) = COL_WHITE Then
      If AttackBitCnt(BAttack(Target)) > AttackBitCnt(WAttack(Target)) Then rr = rr
      + 1  ' King attacks unprotected piece
    End If
  End If
Next

If r > 0 Then
  If r = 1 Then AddScore WThreat, KingOnOneBonus Else AddScore WThreat,
  KingOnManyBonus
End If
If rr > 0 Then
  If rr = 1 Then AddScore BThreat, KingOnOneBonus Else AddScore BThreat,
  KingOnManyBonus
End If

'------------------------------------------------
'--- Step 6: Eval threats ------------------------
'------------------------------------------------
CalcThreats  ' in WThreat and BThreat

If WWeakUnopposedCnt > 0 Then
  If PieceCnt(BQUEEN) + PieceCnt(BROOK) > 0 Then AddScoreWithFactor BThreat,
  WeakUnopposedPawn, WWeakUnopposedCnt
End If
If BWeakUnopposedCnt > 0 Then
  If PieceCnt(WQUEEN) + PieceCnt(WROOK) > 0 Then AddScoreWithFactor WThreat,
  WeakUnopposedPawn, BWeakUnopposedCnt
End If

' Trapped bishops at a7/h7, a2/h2
If PieceCnt(WBISHOP) > 0 Then
  ' white bishop not defended trapped at A7 by black pawn B6 (or if pawn can move to B6)
  If Board(SQ_A7) = WBISHOP Then
    If BAttack(SQ_B6) > 0 And WAttack(SQ_A7) = 0 Then
      If Board(SQ_B6) = BPAWN Or (Not bWhiteToMove And Board(SQ_B6) >= NO_PIECE And
      Board(SQ_B7) = BPAWN) Then
        AddScoreVal BThreat, ScoreBishop.MG \ 3, ScoreBishop.MG \ 4
      End If
    End If
  End If
  If Board(SQ_H7) = WBISHOP Then
    If BAttack(SQ_G6) > 0 And WAttack(SQ_H7) = 0 Then
      If Board(SQ_G6) = BPAWN Or (Not bWhiteToMove And Board(SQ_G6) >= NO_PIECE And
      Board(SQ_G7) = BPAWN) Then
        AddScoreVal BThreat, ScoreBishop.MG \ 3, ScoreBishop.MG \ 4
      End If
    End If
  End If
End If
If PieceCnt(BBISHOP) > 0 Then
  If Board(SQ_A2) = BBISHOP Then
    If WAttack(SQ_B3) > 0 And BAttack(SQ_A2) = 0 Then
      If Board(SQ_B3) = WPAWN Or (bWhiteToMove And Board(SQ_B3) >= NO_PIECE And
      Board(SQ_B2) = WPAWN) Then
        AddScoreVal WThreat, ScoreBishop.MG \ 3, ScoreBishop.MG \ 4
      End If
    End If
  End If
  If Board(SQ_H2) = BBISHOP Then
    If WAttack(SQ_G3) > 0 And BAttack(SQ_H2) = 0 Then
      If Board(SQ_G3) = WPAWN Or (bWhiteToMove And Board(SQ_G3) >= NO_PIECE And
      Board(SQ_G2) = WPAWN) Then
        AddScoreVal WThreat, ScoreBishop.MG \ 3, ScoreBishop.MG \ 4
      End If
    End If
  End If
End If
'
'--- Passed pawns (white and black). done here because full attack info is needed
'
WFrontMostPassedPawnRank = 0: BFrontMostPassedPawnRank = 0

For a = 1 To PassedPawnsCnt
  Dim AttackedFromBehind As Long, DefendedFromBehind As Long
  Square = PassedPawns(a): FileNum = File(Square): RankNum = Rank(Square)
  MBonus = 0: EBonus = 0: UnsafeCnt = 0
  If PieceColor(Board(Square)) = COL_WHITE Then
    ' White piece
    OwnCol = COL_WHITE: OppCol = COL_BLACK: MoveUp = SQ_UP
    RelRank = RankNum: OwnKingLoc = WKingLoc: OppKingLoc = BKingLoc
    ' Attack Opp King?
    If RelRank > WFrontMostPassedPawnRank Then WFrontMostPassedPawnRank = RankNum
    If PieceCnt(WBISHOP) > 0 Then     ' Bishop with same color as promote square? ( not SF logic )
      sq = SQ_A1 + FileNum - 1 + 7 * MoveUp
      If ColorSq(sq) = COL_WHITE Then
        r = Sgn(Sgn(WBishopsOnWhiteSq) - Sgn(BBishopsOnWhiteSq)) ' 0 if both sides have
        same bishop color, else +1 or -1
        If r <> 0 Then MBonus = MBonus + r * (10 + (RelRank - 1) * 3): EBonus =
        EBonus + r * (20 + (RelRank - 1) * (RelRank - 1))
      Else
        r = Sgn(Sgn(WBishopsOnBlackSq) - Sgn(BBishopsOnBlackSq)) ' 0 if both sides have
        same bishop color, else +1 or -1
        If r <> 0 Then MBonus = MBonus + r * (10 + (RelRank - 1) * 3): EBonus =
        EBonus + r * (20 + (RelRank - 1) * (RelRank - 1))
      End If
    End If
  Else
    OwnCol = COL_BLACK: OppCol = COL_WHITE: MoveUp = SQ_DOWN
    ' Black piece
    RelRank = (9 - RankNum):  OwnKingLoc = BKingLoc: OppKingLoc = WKingLoc
    If RelRank > BFrontMostPassedPawnRank Then BFrontMostPassedPawnRank = RelRank
    If PieceCnt(BBISHOP) > 0 Then  ' Bishop with same color as promote square?
      sq = SQ_A1 + FileNum - 1
      If ColorSq(sq) = COL_WHITE Then
        r = Sgn(Sgn(BBishopsOnWhiteSq) - Sgn(WBishopsOnWhiteSq)) ' 0 if both sides have
        same bishop color, else +1 or -1
        If r <> 0 Then MBonus = MBonus + r * (10 + (RelRank - 1) * 3): EBonus =
        EBonus + r * (20 + (RelRank - 1) * (RelRank - 1))
      Else
        r = Sgn(Sgn(BBishopsOnBlackSq) - Sgn(WBishopsOnBlackSq)) ' 0 if both sides have
        same bishop color, else +1 or -1
        If r <> 0 Then MBonus = MBonus + r * (10 + (RelRank - 1) * 3): EBonus =
        EBonus + r * (20 + (RelRank - 1) * (RelRank - 1))
      End If
    End If
  End If
  '
  '--- Path to promote square blocked? => penalty
  '
  r = RelRank
  rr = PassedDanger(RelRank)
  MBonus = MBonus + PassedPawnRankBonus(r).MG: EBonus = EBonus + PassedPawnRankBonus
  (r).EG
  ' Bonus based on rank ' SF9
  If rr <> 0 Then
    BlockSq = Square + MoveUp
    If Board(BlockSq) <> FRAME Then
      '  Adjust bonus based on the king's proximity
      AttackedFromBehind = 0: DefendedFromBehind = 0
      EBonus = EBonus + (GetMin(5, MaxDistance(BlockSq, OppKingLoc)) * 5 - GetMin(5,
       MaxDistance(BlockSq, OwnKingLoc)) * 2) * rr
      'If blockSq is not the queening square then consider also a second push
      If RelRank <> 7 Then EBonus = EBonus - MaxDistance(BlockSq + MoveUp,
      OwnKingLoc) * rr
      'If the pawn is free to advance, then increase the bonus
      If Board(BlockSq) >= NO_PIECE Then
        k = 0: bAllDefended = True: BlockSqDefended = True: BlockSqUnsafe = False
        ' Rook or Queen attacking/defending from behind?
        If CBool(BAttack(Square) And QRAttackBit) Or CBool(WAttack(Square) And
        QRAttackBit) Then
          sq = Square
          For RankPath = RelRank - 1 To 1 Step -1
            sq = sq - MoveUp ' move down to rank 1
            Select Case Board(sq)
              Case NO_PIECE:
              Case BROOK, BQUEEN:
                If OwnCol = COL_WHITE Then
                  BlockSqUnsafe = True: AttackedFromBehind = 1
                Else
                  DefendedFromBehind = 1:
                End If
                Exit For
              Case WROOK, WQUEEN:
                If OwnCol = COL_BLACK Then
                  BlockSqUnsafe = True: AttackedFromBehind = 1
                Else
                  DefendedFromBehind = 1
                End If
                Exit For
              Case Else:
                Exit For
            End Select
          Next
        End If

        sq = Square
        For RankPath = RelRank + 1 To 8
          sq = sq + MoveUp
          OwnAttCnt = AttackBitCnt(AttackByCol(OwnCol, sq)) + DefendedFromBehind
          If OwnAttCnt = 0 And sq <> OwnKingLoc Then
            bAllDefended = False: If sq = BlockSq Then BlockSqDefended = False
          End If
          If PieceColor(Board(sq)) = OppCol Then
            If sq = BlockSq Then BlockSqUnsafe = True Else UnsafeCnt = UnsafeCnt + 1
          ElseIf AttackBitCnt(AttackByCol(OppCol, sq)) + AttackedFromBehind > 0 Then
            If CBool(AttackByCol(OwnCol, sq) And PAttackBit) And Not CBool(AttackByCol(OppCol, sq) And PAttackBit) Then
              ' Own pawn support but no enemy pawn attack: square is safe ( NOT SF LOGIC )
            Else
              If sq = BlockSq Then BlockSqUnsafe = True Else UnsafeCnt = UnsafeCnt + 1
            End If
          End If
        Next RankPath

        If BlockSqUnsafe Then UnsafeCnt = UnsafeCnt + 1
        If UnsafeCnt = 0 Then
          k = 20
        ElseIf Not BlockSqUnsafe Then
          k = 9 '- UnsafeCnt
        Else
          k = 0
        End If
        If bAllDefended Then
          k = k + 6 '- UnsafeCnt \ 2
        ElseIf BlockSqDefended Then
          k = k + 4 '- UnsafeCnt \ 2
        End If
        ' If protected by more than one rook or queen, assign extra bonus
        If k > 0 Then
          If OwnCol = COL_WHITE Then
            If AttackBitCnt((WAttack(Square) And QOrXrayROrXrayAttackBit)) > 1 Then k = k + 2
          Else
            If AttackBitCnt((BAttack(Square) And QOrXrayROrXrayAttackBit)) > 1 Then k = k + 2
          End If
        End If
        '-- add bonus
        If k <> 0 Then MBonus = MBonus + k * rr: EBonus = EBonus + k * rr
      Else
        If PieceColor(Board(BlockSq)) = OwnCol Then MBonus = MBonus + rr + (r - 1) * 2: EBonus = EBonus + rr + (r - 1) * 2 'r-1 because rank r is 0 based in C
      End If
    End If
  End If 'rr>0
  '
  If UnsafeCnt > 0 Then MBonus = MBonus - UnsafeCnt * 8: EBonus = EBonus - UnsafeCnt ' hindered passed pawn
  '
  If OwnCol = COL_WHITE Then
    If WPawnCnt > BPawnCnt Then EBonus = EBonus + EBonus \ 4
    MBonus = MBonus + PassedPawnFileBonus(FileNum).MG: EBonus = EBonus + PassedPawnFileBonus(FileNum).EG
    If BNonPawnMaterial = 0 Then EBonus = EBonus + 20
    If Board(Square + SQ_UP) = BPAWN Then MBonus = MBonus \ 2: EBonus = EBonus \ 2 ' supporter sacrify needed
    If bWhiteToMove Then MBonus = (MBonus * 105) \ 100:   EBonus = (EBonus * 105) \ 100
    AddScoreVal WPassed, MBonus, EBonus
```

```
      If 1000 + EBonus > WBestPawnVal Then WBestPawn = Square: WBestPawnVal = 1000 + EBonus ' new best pawn
      If bEvalTrace Then WriteTrace "WPassed: " & LocCoord(Square) & ">" & MBonus & ", " & EBonus
    ElseIf OwnCol = COL_BLACK Then
      If BPawnCnt > WPawnCnt Then EBonus = EBonus + EBonus \ 4
      MBonus = MBonus + PassedPawnFileBonus(FileNum).MG: EBonus = EBonus + PassedPawnFileBonus(FileNum).EG
      If WNonPawnMaterial = 0 Then EBonus = EBonus + 20
      If Board(Square + SQ_DOWN) = WPAWN Then MBonus = MBonus \ 2: EBonus = EBonus \ 2 ' supporter sacrify needed
      If Not bWhiteToMove Then MBonus = (MBonus * 105) \ 100:   EBonus = (EBonus * 105) \ 100
      AddScoreVal BPassed, MBonus, EBonus
      If 1000 + EBonus > BBestPawnVal Then BBestPawn = Square: BBestPawnVal = 1000 + EBonus ' new best pawn
      If bEvalTrace Then WriteTrace "BPassed: " & LocCoord(Square) & ">" & MBonus & ", " & EBonus
    End If
  Next a
  '
  '---<<< end  Passed pawn
  '
  '--- If both sides have only pawns, score for potential unstoppable pawns
  If WNonPawnMaterial + BNonPawnMaterial = 0 Then
    If WFrontMostPassedPawnRank > 0 Then AddScoreVal WPassed, 0, WFrontMostPassedPawnRank * 20
    If BFrontMostPassedPawnRank > 0 Then AddScoreVal BPassed, 0, BFrontMostPassedPawnRank * 20 ' RelRank is used, so >0 is correct
  End If
  '
  '---  Penalty for pawns on same color square of bishop
  '
  If PieceCnt(WBISHOP) > 0 Then
    r = WPawnCntOnWhiteSq * WBishopsOnWhiteSq + (WPawnCnt - WPawnCntOnWhiteSq) * WBishopsOnBlackSq
    If r <> 0 Then
      r = r * (1 + WCenterPawnsBlocked)
      AddScoreVal WPos, -3 * r, -5 * r
    End If
    ' Bonus for bishop on a long diagonal if it can "see" both center squares and no pawns
    If WBishopsOnWhiteSq > 0 And Not bEndgame Then
      If CBool(WAttack(SQ_E4) And BAttackBit) Then
        If PieceType(Board(SQ_E4)) <> PT_PAWN Then
          If CBool(WAttack(SQ_D5) And BAttackBit) Then If PieceType(Board(SQ_D5)) <> PT_PAWN Then WPos.MG = WPos.MG + 22
        End If
      End If
    End If
    If WBishopsOnBlackSq > 0 Then
      If CBool(WAttack(SQ_D4) And BAttackBit) Then
        If PieceType(Board(SQ_D4)) <> PT_PAWN Then
          If CBool(WAttack(SQ_E5) And BAttackBit) Then If PieceType(Board(SQ_E5)) <> PT_PAWN Then WPos.MG = WPos.MG + 22
        End If
      End If
    End If
  End If
  If PieceCnt(BBISHOP) > 0 Then
    r = BPawnCntOnWhiteSq * BBishopsOnWhiteSq + (BPawnCnt - BPawnCntOnWhiteSq) * BBishopsOnBlackSq
    If r <> 0 Then
      r = r * (1 + BCenterPawnsBlocked)
      AddScoreVal BPos, -3 * r, -5 * r
    End If
    ' Bonus for bishop on a long diagonal if it can "see" both center squares and no pawns
    If BBishopsOnWhiteSq > 0 And Not bEndgame Then
      If CBool(BAttack(SQ_D5) And BAttackBit) Then
```

```
        If PieceType(Board(SQ_D5)) <> PT_PAWN Then
          If CBool(BAttack(SQ_E4) And BAttackBit) Then If PieceType(Board(SQ_E4)) <> PT_PAWN Then BPos.MG = BPos.MG + 22
        End If
      End If
    End If
    If BBishopsOnBlackSq > 0 Then
      If CBool(BAttack(SQ_E5) And BAttackBit) Then
        If PieceType(Board(SQ_E5)) <> PT_PAWN Then
          If CBool(BAttack(SQ_D4) And BAttackBit) Then If PieceType(Board(SQ_D4)) <> PT_PAWN Then BPos.MG = BPos.MG + 22
        End If
      End If
    End If
  End If
  '
  '--->>> Pawn Islands (groups of pawns) ---
  '
  r = 0: bWIsland = False   ' r : white islands
  rr = 0: bBIsland = False ' rr: black islands

  For FileNum = 1 To 9
    If WPawns(FileNum) <= 0 Then   ' File WPawns(9) = -1
      bWIsland = True
    ElseIf bWIsland Then
      r = r + 1: bWIsland = False ' empty file and pawn onleft side > island
    End If
    If BPawns(FileNum) <= 0 Then   ' File BPawns(9) = -1
      bBIsland = True
    ElseIf bBIsland Then
      rr = rr + 1: bBIsland = False ' empty file and pawn onleft side > island
    End If
  Next

  If r > 0 Then AddScoreVal WPawnStruct, -15 * r, -25 * r ' Penalty for each island
  If rr > 0 Then AddScoreVal BPawnStruct, -15 * rr, -25 * rr
  '---<<< end Pawn Islands ---
  '
  '----------------------------------------------------------------------------
  '--- Step 7: Calculate total material values and endgame scale factors    ---
  '----------------------------------------------------------------------------
  '
  ' Piece values were set in SetGamePhase
  Dim AllTotal As TScore, MatEval As Long
  AllTotal.MG = Material  ' Based on MG, no need to recalc ' (PieceCnt(WQUEEN) - PieceCnt(BQUEEN)) * ScoreQueen.MG + (PieceCnt(WROOK) - PieceCnt(BROOK)) * ScoreRook.MG + (PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) * ScoreBishop.MG + (PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT)) * ScoreKnight.MG + (WPawnCnt - BPawnCnt) * ScorePawn.MG
  AllTotal.EG = (PieceCnt(WQUEEN) - PieceCnt(BQUEEN)) * ScoreQueen.EG + (PieceCnt(WROOK) - PieceCnt(BROOK)) * ScoreRook.EG + (PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) * ScoreBishop.EG + (PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT)) * ScoreKnight.EG + (WPawnCnt - BPawnCnt) * ScorePawn.EG
  MatEval = ScaleScore(AllTotal)
  If bEvalTrace Then
    Debug.Print "Material: " & EvalSFTo100(AllTotal.MG) & "," & EvalSFTo100(AllTotal.EG)
  End If

  '
  '--- Calculate SPACE in opening phase for safe squares in center (files 3-6, ranks 2-4)
  '
  If NonPawnMaterial > SPACE_THRESHOLD Then
    r = 0: rr = 0
    For k = 3 To 6   ' files 3-6
      '--- White space
      Offset = PawnsWMin(k): Target = k + 20
      For RankNum = 2 To 4 ' WHITE
        Target = Target + 10
        If Board(Target) <> WPAWN Then
          If Not CBool(BAttack(Target) And PAttackBit) Then
            r = r + 1: If RankNum < Offset Then If RankNum >= Offset - 3 Then r = r + 1 ' extra bonus if at most three squares behind some friendly pawn
          End If
        End If
      Next

      '--- Black space
      Offset = PawnsBMin(k): Target = k + 50
      For RankNum = 5 To 7 '
        Target = Target + 10
        If Board(Target) <> BPAWN Then
          If Not CBool(WAttack(Target) And PAttackBit) Then
            rr = rr + 1: If RankNum <= Offset + 3 And RankNum > Offset Then rr = rr + 1 ' extra bonus if at most three squares behind some friendly pawn
          End If
        End If
      Next
    Next

    If r + rr <> 0 Then
      ' weight for space
      k = 0
      For i = 1 To 8 ' count open files
        If WPawns(i) = 0 Then If BPawns(i) = 0 Then k = k + 1
      Next
      If r > 0 Then
        a = WNonPawnPieces + 1 + WPawnCnt - 2 * k
        WPos.MG = WPos.MG + r * a * a \ 16
      End If
      If rr > 0 Then
        a = BNonPawnPieces + 1 + BPawnCnt - 2 * k
        BPos.MG = BPos.MG + rr * a * a \ 16
      End If
    End If
  End If '<<< Space


  '-----------------------------------------------
  '--- Step 8: Calculate weights and total eval -
  '-----------------------------------------------
  '
  '--- evaluate_initiative() /Complexity computes the initiative correction value for the
  '--- position, i.e., second order bonus/malus based on the known attacking/defending status of the players.
  '--- Semiopenfiles \12 because tricky counting to avoid count duplicate pawns per file
  '------ REMOVED: slightly better result without this logic. More complex better because no EGTB?
'  k = 12 * (Abs(WKingFile - BKingFile) - Abs(Rank(WKingLoc) - Rank(BKingLoc))) _
'     + 8 * (Abs(WSemiOpenFiles + BSemiOpenFiles) \ 12 + PassedPawnsCnt) _
'     + 12 * (WPawnCnt + BPawnCnt) _
'     + 16 * Abs(KingSidePawns > 0 And QueenSidePawns > 0) _
'     + 48 * Abs(NonPawnMaterial = 0) _
'     - 136
'
'  rr = MatEval + (WPos.EG - BPos.EG) + (WPassed.EG - BPassed.EG) ' strong side?
'  If rr > 0 Then
'    WPos.EG = WPos.EG + GetMax(k, -Abs(rr))
'  ElseIf rr < 0 Then
'    BPos.EG = BPos.EG + GetMax(k, -Abs(rr))
'  End If

  '
  '--- Material Imbalance / Score trades
  '
  Dim TradeEval         As Long
  If MatEval = 0 Then
    TradeEval = 0
  Else
```

```
    TradeEval = Imbalance() 'SF6
    AllTotal.MG = AllTotal.MG + TradeEval: AllTotal.EG = AllTotal.EG + TradeEval
  End If
  AllTotal.MG = AllTotal.MG + ((WPos.MG - BPos.MG) * PiecePosScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + ((WPos.EG - BPos.EG) * PiecePosScaleFactor) \ 100&
  AllTotal.MG = AllTotal.MG + ((WPawnStruct.MG - BPawnStruct.MG) * PawnStructScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + ((WPawnStruct.EG - BPawnStruct.EG) * PawnStructScaleFactor) \ 100&
  AllTotal.MG = AllTotal.MG + ((WPassed.MG - BPassed.MG) * PassedPawnsScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + ((WPassed.EG - BPassed.EG) * PassedPawnsScaleFactor) \ 100&
  AllTotal.MG = AllTotal.MG + ((WMobility.MG - BMobility.MG) * MobilityScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + ((WMobility.EG - BMobility.EG) * MobilityScaleFactor) \ 100&
  '
  ' different weights for defending computer king / attacking opp king
  If bCompIsWhite Then
    WKingScaleFactor = CompKingDefScaleFactor: BKingScaleFactor = OppKingAttScaleFactor
  Else
    BKingScaleFactor = CompKingDefScaleFactor: WKingScaleFactor = OppKingAttScaleFactor
  End If
  If bWhiteToMove Then
    WKingScaleFactor = WKingScaleFactor + 5
  Else
    BKingScaleFactor = BKingScaleFactor + 5
  End If

  AllTotal.MG = AllTotal.MG + (WKSafety.MG * WKingScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + (WKSafety.EG * WKingScaleFactor) \ 100&
  AllTotal.MG = AllTotal.MG - (BKSafety.MG * BKingScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG - (BKSafety.EG * BKingScaleFactor) \ 100&
  AllTotal.MG = AllTotal.MG + ((WThreat.MG - BThreat.MG) * ThreatsScaleFactor) \ 100&
  AllTotal.EG = AllTotal.EG + ((WThreat.EG - BThreat.EG) * ThreatsScaleFactor) \ 100&
  '------------------------------------------------------------------------------------------

  '
  '--- Scale Factor ---
  '
  ScaleFactor = SCALE_FACTOR_NORMAL ' Normal ScaleFactor, scales EG value only
  If GamePhase < PHASE_MIDGAME Then
    ' KRPPKRP endgame
    'if the defending king is actively placed and not passed pawn for strong side, the position is drawish
    If WNonPawnMaterial = ScoreRook.MG And BNonPawnMaterial = ScoreRook.MG Then
      If WPawnCnt = 2 And BPawnCnt = 1 Then ' white is strong side
        If WFrontMostPassedPawnRank = 0 Then ' no passed pawn for strong side
          Square = PieceSqList(WPAWN, 1) ' 1. pawn
          If Rank(BKingLoc) > Rank(Square) Then ' Opp king in front
            If Abs(File(Square) - File(BKingLoc)) <= 1 Then ' and nearby file
              r = Rank(Square): Square = PieceSqList(WPAWN, 2) ' 2. pawn
              If Rank(BKingLoc) > Rank(Square) Then ' Opp king in front
                If Abs(File(Square) - File(BKingLoc)) <= 1 Then ' and nearby file
                  ScaleFactor = KRPPKRP_SFactor(GetMax(r, Rank(Square))): GoTo lblEndScaleFactor
                End If
              End If
            End If
          End If
        End If
      ElseIf BPawnCnt = 2 And WPawnCnt = 1 Then ' black is strong side
        If BFrontMostPassedPawnRank = 0 Then
          Square = PieceSqList(BPAWN, 1) ' 1. pawn
          If RelativeRank(COL_BLACK, WKingLoc) > RelativeRank(COL_BLACK, Square) Then ' Opp king in front
```

```
            If Abs(File(Square) - File(WKingLoc)) <= 1 Then ' and nearby file
              r = RelativeRank(COL_BLACK, Square)
              Square = PieceSqList(BPAWN, 2) ' 2. pawn
              If RelativeRank(COL_BLACK, WKingLoc) > RelativeRank(COL_BLACK, Square) Then ' Opp king in front
                If Abs(File(Square) - File(WKingLoc)) <= 1 Then ' and nearby file
                  ScaleFactor = KRPPKRP_SFactor(GetMax(r, RelativeRank(COL_BLACK, Square))): GoTo lblEndScaleFactor
                End If
              End If
            End If
          End If
        End If
      End If
    End If

    '- zero or just one pawn makes it difficult to win
    If AllTotal.EG > 0 Then ' white stronger
      If NonPawnMat() = 0 Then If WPawnCnt > BPawnCnt Then ScaleFactor = 96 ' A small advantage is typically decisive in a pure pawn endings
      Select Case WPawnCnt
      Case 0:
        If WNonPawnMaterial - BNonPawnMaterial <= ScoreBishop.MG Then
          If WNonPawnMaterial < ScoreRook.MG Then
            ScaleFactor = SCALE_FACTOR_DRAW
          Else
            If BNonPawnMaterial <= ScoreBishop.MG Then ScaleFactor = 4 Else ScaleFactor = 44
          End If
        End If
      Case 1: If WNonPawnMaterial - BNonPawnMaterial <= ScoreBishop.MG Then ScaleFactor = SCALE_FACTOR_ONEPAWN
      End Select
    ElseIf AllTotal.EG < 0 Then
      If NonPawnMat = 0 Then If BPawnCnt > WPawnCnt Then ScaleFactor = 96 ' A small advantage is typically decisive in a pure pawn endings
      Select Case BPawnCnt
      Case 0:
        If BNonPawnMaterial - WNonPawnMaterial <= ScoreBishop.MG Then
          If BNonPawnMaterial < ScoreRook.MG Then
            ScaleFactor = SCALE_FACTOR_DRAW
          Else
            If WNonPawnMaterial <= ScoreBishop.MG Then ScaleFactor = 4 Else ScaleFactor = 44
          End If
        End If
      Case 1: If BNonPawnMaterial - WNonPawnMaterial <= ScoreBishop.MG Then ScaleFactor = SCALE_FACTOR_ONEPAWN
      End Select
    End If

    '
    '- Endgame with opposite-colored bishops and no other pieces (ignoring pawns)
    '- is almost a draw, in case of KBP vs KB, it is even more a draw.
    If PieceCnt(WBISHOP) = 1 And PieceCnt(BBISHOP) = 1 And WBishopsOnWhiteSq = BBishopsOnBlackSq Then ' opposite-colored bishops
      If (WNonPawnMaterial = ScoreBishop.MG Or WNonPawnMaterial = ScoreBishop.MG + ScoreQueen.MG) And BNonPawnMaterial = WNonPawnMaterial Then
        If WPawnCnt + BPawnCnt > 1 Then ScaleFactor = 31 Else ScaleFactor = 9
      Else
        ' Endgame with opposite-colored bishops, but also other pieces. Still
        ' a bit drawish, but not as drawish as with only the two bishops.
        If PieceCnt(WQUEEN) + PieceCnt(BQUEEN) = 0 Then ScaleFactor = 46
      End If
    Else
      If WNonPawnMaterial + BNonPawnMaterial = 0 Then
        ' KPsK: K and two or more pawns vs K. There is just a single rule here: If all pawns
        ' are on the same rook file and are blocked by the defending king, it's a draw.
```

```
          If WPawnCnt >= 2 And BPawnCnt = 0 Then
            If File(PieceSqList(WPAWN, 1)) = 1 Or File(PieceSqList(WPAWN, 1)) = 8 Then
              r = 0

              For a = 1 To PieceSqListCnt(WPAWN)
                If File(PieceSqList(WPAWN, a)) <> File(PieceSqList(WPAWN, 1)) Then r = 1
                : Exit For
                If Abs(File(PieceSqList(WPAWN, a)) - File(BKingLoc)) > 1 Then r = 1:
                Exit For
                If Rank(PieceSqList(WPAWN, a)) >= Rank(BKingLoc) Then r = 1: Exit For
              Next

              If r = 0 Then ScaleFactor = 0 ' Draw
            End If
          ElseIf BPawnCnt >= 2 And WPawnCnt = 0 Then
            If File(PieceSqList(BPAWN, 1)) = 1 Or File(PieceSqList(BPAWN, 1)) = 8 Then
              r = 0

              For a = 1 To PieceSqListCnt(BPAWN)
                If File(PieceSqList(BPAWN, a)) <> File(PieceSqList(BPAWN, 1)) Then r = 1
                : Exit For
                If Abs(File(PieceSqList(BPAWN, a)) - File(WKingLoc)) > 1 Then r = 1:
                Exit For
                If Rank(PieceSqList(BPAWN, a)) <= Rank(WKingLoc) Then r = 1: Exit For
              Next

              If r = 0 Then ScaleFactor = 0 ' Draw
            End If
          End If
        End If
        If ScaleFactor = SCALE_FACTOR_NORMAL Or ScaleFactor = SCALE_FACTOR_ONEPAWN Then
          ' Endings where weaker side can stop one of the enemy's pawn are drawish.
          If AllTotal.EG > 0 Then ' White is strong side
                ScaleFactor = GetMin(40 + 7 * WPawnCnt, ScaleFactor)
          ElseIf AllTotal.EG < 0 Then ' Black is strong side
                ScaleFactor = GetMin(40 + 7 * BPawnCnt, ScaleFactor)
          End If
        End If
      End If
    End If

  lblEndScaleFactor:


    '
    '--- Added all to eval score (SF based scaling:  Eval*100/SFPawnEndGameValue= 100 centipawns =1 pawn)
    '--- Example: Eval=240 => 1.00 pawn
    Eval = AllTotal.MG * GamePhase + AllTotal.EG * CLng(PHASE_MIDGAME - GamePhase) *
    ScaleFactor \ SCALE_FACTOR_NORMAL
    Eval = Eval \ PHASE_MIDGAME
    ' Initiative: Keep more pawns when attacking
    Bonus = (50 * (14 - (WPawnCnt + BPawnCnt))) \ 14
    If Eval > 0 Then
      Eval = GetMax(Eval - Bonus, Eval \ 2)
    ElseIf Eval < 0 Then
      Eval = GetMin(Eval + Bonus, Eval \ 2)
    End If
  lblEndEval:
    If bEvalTrace Then
      bEvalTrace = False
      WriteTrace "---- EVAL TRACE : " & Now()
      WriteTrace PrintPos
      WriteTrace "Material: " & EvalSFTo100(MatEval)
      WriteTrace "Trades  : " & EvalSFTo100(TradeEval)
      WriteTrace "Position: " & ShowScoreDiff100(WPos, BPos) & "  => W" & ShowScore(WPos
      ) & ", B" & ShowScore(BPos)
      WriteTrace "PawnStru: " & ShowScoreDiff100(WPawnStruct, BPawnStruct) & " => W" &
      ShowScore(WPawnStruct) & ", B" & ShowScore(BPawnStruct)
      WriteTrace "PassedPw: " & ShowScoreDiff100(WPassed, BPassed) & " => W" & ShowScore(WPassed) & ", B" & ShowScore(BPassed)
      WriteTrace "Mobility: " & ShowScoreDiff100(WMobility, BMobility) & " => W(" & ShowScore(WMobility) & ", B" & ShowScore(BMobility)
      WriteTrace "KSafety : " & ShowScoreDiff100(WKSafety, BKSafety) & " => W(" & ShowScore(WKSafety) & ", B" & ShowScore(BKSafety)
      WriteTrace "Threats : " & ShowScoreDiff100(WThreat, BThreat) & " => W(" & ShowScore(WThreat) & ", B" & ShowScore(BThreat)
      WriteTrace "Eval    : " & Eval & "  (" & EvalSFTo100(Eval) & "cp)"
      WriteTrace "-----------------"
      bTimeExit = True
    End If

    '-----------------------------------------------
    '--- Step 9: Invert score for black to move   ---
    '-----------------------------------------------
    If Not bWhiteToMove Then Eval = -Eval '--- Invert for black

    '------------------------------------------------
    '--- Step 10: Add tempo value for side to move ---
    '------------------------------------------------
    'Eval = Eval + TEMPO_BONUS ' Tempo for side to move
    Eval = Eval + (16 + NonPawnMaterial \ ScoreKnight.MG \ 2) ' use dynamic tempo, more during opening

    ' draw value?
    If Eval = DrawContempt Then
      Eval = Eval + 1 ' if not a real draw then make a difference
    End If

  '  If Eval = DrawContempt And bWhiteToMove Then
  '    Eval = Eval + 1 ' if not a real draw then make a difference
  '  ElseIf Eval = -DrawContempt And Not bWhiteToMove Then
  '    Eval = Eval - 1 ' if not a real draw for black then make a difference
  '  Else  '---  other logics: tested, results bad
  '   ' Eval = (Eval \ 16) * 16 ' add granularity
  '
  '     ' dump eval when shuffling
  '    If Fifty > 14 Then  ' slows down engine !?!
  '      If Abs(Eval) > 5 Then
  '        'Eval = (Eval * (230& - Fifty)) \ 200& ' dump eval when shuffling
  '        Eval = (Eval * (200& - Fifty)) \ 214& ' SF
  '      End If
  '    End If
  '  End If

    '
  End Function

  '--------------------------------
  '-------- END OF EVAL ------------
  '--------------------------------

  Private Function Eval__EndOfEval_DUMMY()
    ' for faster navigation in source
  End Function

  Private Function IsMaterialDraw() As Boolean
    '( Protector logic )
    IsMaterialDraw = False
    If PieceCnt(WPAWN) + PieceCnt(BPAWN) = 0 Then ' no pawns
      '---  no heavies Q/R */
      If PieceCnt(WROOK) = 0 And PieceCnt(BROOK) = 0 And PieceCnt(WQUEEN) = 0 And PieceCnt(BQUEEN) = 0 Then
        If PieceCnt(BBISHOP) = 0 And PieceCnt(WBISHOP) = 0 Then
          '---  only knights */
          '---  it pretty safe to say this is a draw */
          If PieceCnt(WKNIGHT) < 3 And PieceCnt(BKNIGHT) < 3 Then IsMaterialDraw = True:
          Exit Function
        ElseIf PieceCnt(WKNIGHT) = 0 And PieceCnt(BKNIGHT) = 0 Then
          '--- only bishops */
          '--- not a draw if one side two other side zero
          '--- else its always a draw                */
          If Abs(PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) < 2 Then IsMaterialDraw = True:
          Exit Function
        ElseIf (PieceCnt(WKNIGHT) < 3 And PieceCnt(WBISHOP) = 0) Or (PieceCnt(WBISHOP) =
         1 And PieceCnt(WKNIGHT) = 0) Then
          '--- we cant win, but can black? */
          If (PieceCnt(BKNIGHT) < 3 And PieceCnt(BBISHOP) = 0) Or (PieceCnt(BBISHOP) = 1
           And PieceCnt(BKNIGHT) = 0) Then IsMaterialDraw = True: Exit Function '--- guess
          not */
        End If
      ElseIf PieceCnt(WQUEEN) = 0 And PieceCnt(BQUEEN) = 0 Then
        If PieceCnt(WROOK) = 1 And PieceCnt(BROOK) = 1 Then
          '--- rooks equal */
          '--- one minor difference max: a draw too usually */
          If (PieceCnt(WKNIGHT) + PieceCnt(WBISHOP)) < 2 And (PieceCnt(BKNIGHT) +
          PieceCnt(BBISHOP)) < 2 Then IsMaterialDraw = True: Exit Function
        ElseIf (PieceCnt(WROOK) = 1 And PieceCnt(BROOK) = 0) Then
          '--- one rook */
          '--- draw if no minors to support AND minors to defend  */
          If (PieceCnt(WKNIGHT) + PieceCnt(WBISHOP) = 0) And ((PieceCnt(BKNIGHT) +
          PieceCnt(BBISHOP) = 1) Or (PieceCnt(BKNIGHT) + PieceCnt(BBISHOP) = 2)) Then
          IsMaterialDraw = True: Exit Function
        ElseIf PieceCnt(BROOK) = 1 And PieceCnt(WROOK) = 0 Then
          '--- one rook */
          '--- draw if no minors to support AND minors to defend  */
          If (PieceCnt(BKNIGHT) + PieceCnt(BBISHOP) = 0) And ((PieceCnt(WKNIGHT) +
          PieceCnt(WBISHOP) = 1) Or (PieceCnt(WKNIGHT) + PieceCnt(WBISHOP) = 2)) Then
          IsMaterialDraw = True: Exit Function
        End If
      End If
    End If
  End Function

  Public Function AdvancedPawnPush(ByVal Piece As Long, ByVal Target As Long) As Boolean
    AdvancedPawnPush = False
    If Piece = WPAWN Then

      Select Case Rank(Target)
        Case 7, 8: AdvancedPawnPush = True
        Case 6:
          '--- if no enemy in front and no enemy pawns left or right
          If (Board(Target + SQ_UP) >= NO_PIECE Or (Board(Target + SQ_UP) And 1) = WCOL)
           Then If Board(Target + SQ_UP_LEFT) <> BPAWN And Board(Target + SQ_UP_RIGHT)
          <> BPAWN Then AdvancedPawnPush = True
      End Select

    ElseIf Piece = BPAWN Then

      Select Case Rank(Target)
        Case 1, 2: AdvancedPawnPush = True
        Case 3:
          If (Board(Target + SQ_DOWN) >= NO_PIECE Or (Board(Target + SQ_DOWN) And 1) =
          BCOL) Then If Board(Target + SQ_DOWN_LEFT) <> WPAWN And Board(Target +
          SQ_DOWN_RIGHT) <> WPAWN Then AdvancedPawnPush = True
      End Select

    End If
  End Function

  Public Function AdvancedPassedPawnPush(ByVal Piece As Long, ByVal Target As Long) As
  Boolean
    AdvancedPassedPawnPush = False
    If Piece = WPAWN Then

      Select Case Rank(Target)
        Case 7, 8: AdvancedPassedPawnPush = True
        Case 6:
          '--- if no enemy pawn in front and no enemy pawns left or right
          If Board(Target + SQ_UP) = BPAWN Then Exit Function
          If Board(Target + SQ_UP_LEFT) = BPAWN Then Exit Function
          If Board(Target + SQ_UP_RIGHT) = BPAWN Then Exit Function
          AdvancedPassedPawnPush = True
      End Select

    ElseIf Piece = BPAWN Then

      Select Case Rank(Target)
        Case 1, 2: AdvancedPassedPawnPush = True
        Case 3:
          '--- if no enemy pawn in front and no enemy pawns left or right
          If Board(Target + SQ_DOWN) = WPAWN Then Exit Function
          If Board(Target + SQ_DOWN_LEFT) = WPAWN Then Exit Function
          If Board(Target + SQ_DOWN_RIGHT) = WPAWN Then Exit Function
          AdvancedPassedPawnPush = True
      End Select

    End If
  End Function

  Public Function PieceSquareVal(ByVal Piece As Long, ByVal Square As Long) As Long
    '--- Piece value for a square
    PieceSquareVal = 0
    If bEndgame Then

      Select Case Piece
        Case NO_PIECE
        Case WPAWN
          PieceSquareVal = PsqtWP(Square).EG
        Case BPAWN
          PieceSquareVal = PsqtBP(Square).EG
        Case WKNIGHT
          PieceSquareVal = PsqtWN(Square).EG
        Case BKNIGHT
          PieceSquareVal = PsqtBN(Square).EG
        Case WBISHOP
          PieceSquareVal = PsqtWB(Square).EG
        Case BBISHOP
          PieceSquareVal = PsqtBB(Square).EG
        Case WROOK
          PieceSquareVal = PsqtWR(Square).EG
        Case BROOK
          PieceSquareVal = PsqtBR(Square).EG
        Case WQUEEN
          PieceSquareVal = PsqtWQ(Square).EG
        Case BQUEEN
          PieceSquareVal = PsqtBQ(Square).EG
        Case WKING
          PieceSquareVal = PsqtWK(Square).EG
        Case BKING
          PieceSquareVal = PsqtBK(Square).EG
      End Select

    Else

      Select Case Piece
        Case NO_PIECE
        Case WPAWN
          PieceSquareVal = PsqtWP(Square).MG
        Case BPAWN
          PieceSquareVal = PsqtBP(Square).MG
        Case WKNIGHT
          PieceSquareVal = PsqtWN(Square).MG
        Case BKNIGHT
          PieceSquareVal = PsqtBN(Square).MG
        Case WBISHOP
          PieceSquareVal = PsqtWB(Square).MG
        Case BBISHOP
          PieceSquareVal = PsqtBB(Square).MG
        Case WROOK
          PieceSquareVal = PsqtWR(Square).MG
        Case BROOK
          PieceSquareVal = PsqtBR(Square).MG
        Case WQUEEN
          PieceSquareVal = PsqtWQ(Square).MG
        Case BQUEEN
          PieceSquareVal = PsqtBQ(Square).MG
        Case WKING
          PieceSquareVal = PsqtWK(Square).MG
        Case BKING
          PieceSquareVal = PsqtBK(Square).MG
      End Select

    End If
  End Function

  Public Sub FillPieceSquareVal()
    Dim Piece As Long, Target As Long

    For Piece = 1 To 16
      For Target = SQ_A1 To SQ_H8
        bEndgame = False
        PsqVal(0, Piece, Target) = PieceSquareVal(Piece, Target)
        bEndgame = True
        PsqVal(1, Piece, Target) = PieceSquareVal(Piece, Target)
      Next
    Next

  End Sub

  Private Function AttackByCol(Col As Long, Square As Long) As Long
    If Col = COL_WHITE Then AttackByCol = WAttack(Square) Else AttackByCol = BAttack(
    Square)
  End Function

  Public Sub AddPawnThreat(Score As TScore, _
                           ByVal HangCol As enumColor, _
                           ByVal PieceType As enumPieceType, _
                           ByVal Square As Long)
    'SF6:  const Score ThreatBySafePawn[PIECE_TYPE_NB] = {
    '       S(0, 0), S(0, 0), S(107, 138), S(84, 122), S(114, 203), S(121, 217)
    '     const Score ThreatenedByHangingPawn = S(71, 61);
    '--- attack by black pawn?
    If HangCol = COL_WHITE Then
      If Board(Square + SQ_UP_LEFT) = BPAWN Then
        If Board(Square + SQ_UP_LEFT + SQ_UP_LEFT) = BPAWN Or Board(Square + SQ_UP_LEFT
        + SQ_UP_RIGHT) = BPAWN Then
          AddScore Score, ThreatBySafePawn(PieceType)
        Else
          AddScore Score, ThreatenedByHangingPawn
        End If
      ElseIf Board(Square + SQ_UP_RIGHT) = BPAWN Then
        If Board(Square + SQ_UP_RIGHT + SQ_UP_LEFT) = BPAWN Or Board(Square +
        SQ_UP_RIGHT + SQ_UP_RIGHT) = BPAWN Then
          AddScore Score, ThreatBySafePawn(PieceType)
        Else
          AddScore Score, ThreatenedByHangingPawn
        End If
      End If
    Else ' attack by white pawn?
      If Board(Square + SQ_DOWN_LEFT) = WPAWN Then
        If Board(Square + SQ_DOWN_LEFT + SQ_DOWN_LEFT) = WPAWN Or Board(Square + SQ_DOWN_LEFT + SQ_DOWN_RIGHT) = WPAWN Then
          AddScore Score, ThreatBySafePawn(PieceType)
        Else
          AddScore Score, ThreatenedByHangingPawn
        End If
      ElseIf Board(Square + SQ_DOWN_RIGHT) = WPAWN Then
        If Board(Square + SQ_DOWN_RIGHT + SQ_DOWN_LEFT) = WPAWN Or Board(Square + SQ_DOWN_RIGHT + SQ_DOWN_RIGHT) = WPAWN Then
          AddScore Score, ThreatBySafePawn(PieceType)
        Else
          AddScore Score, ThreatenedByHangingPawn
        End If
      End If
    End If
  End Sub

  Public Sub AddThreat(ByVal HangCol As enumColor, _
                       ByVal HangPieceType As enumPieceType, _
                       ByVal AttackerPieceType As enumPieceType, _
                       ByVal AttackerSquare As Long, _
                       ByVal AttackedSquare As Long)
    ' Add threat to threat list. calculate score later when full attack array data is available
    ThreatCnt = ThreatCnt + 1

    With ThreatList(ThreatCnt)
      .HangCol = HangCol
      .HangPieceType = HangPieceType
      .AttackerPieceType = AttackerPieceType
      .AttackerSquare = AttackerSquare
      .AttackedSquare = AttackedSquare
    End With

  End Sub

  Public Sub CalcThreats()
    If ThreatCnt = 0 Then Exit Sub
    Dim i As Long, Defended As Boolean, StronglyProtected As Boolean, Weak As Boolean
    Dim UsAttackCnt As Long, ThemAttackCnt As Long, RelRank As Long, PawnProtected As Boolean, Score As TScore

    For i = 1 To ThreatCnt

      With ThreatList(i)
        '
        ' Add a bonus according to the kind of attacking pieces
        '
        Score = ZeroScore
        If .HangCol = COL_WHITE Then ' view from attacker side = us, attacked = them
          UsAttackCnt = AttackBitCnt(BAttack(.AttackedSquare)): ThemAttackCnt = AttackBitCnt(WAttack(.AttackedSquare))
          PawnProtected = CBool(WAttack(.AttackedSquare) And PAttackBit): RelRank = Rank(.AttackedSquare)
        Else ' Black
          UsAttackCnt = AttackBitCnt(WAttack(.AttackedSquare)): ThemAttackCnt = AttackBitCnt(BAttack(.AttackedSquare))
          PawnProtected = CBool(BAttack(.AttackedSquare) And PAttackBit): RelRank = 9 - Rank(.AttackedSquare)
        End If
        '
        ' StronglyProtected: by pawn or by more defenders then attackers
        StronglyProtected = PawnProtected Or (ThemAttackCnt > UsAttackCnt)
        ' Non-pawn enemies strongly defended
        Defended = .HangPieceType <> PT_PAWN And StronglyProtected
        ' Enemies not strongly defended and under our attack
        Weak = Not StronglyProtected
        If Defended Or Weak Then
          If .AttackerPieceType = PT_BISHOP Or .AttackerPieceType = PT_KNIGHT Then
```

```
          AddScore Score, ThreatByMinor(.HangPieceType)
          If .HangPieceType <> PT_PAWN Then
            AddScoreVal Score, ThreatByRank.MG * RelRank, ThreatByRank.EG * RelRank
          End If
        End If
        If Weak Then If ThemAttackCnt = 0 Then AddScore Score, Hanging ' hanging
        If .HangPieceType <> PT_PAWN Then ' Overload: attacked and defended only once
          If ThemAttackCnt = 1 Then AddScore Score, Overload
        End If
      End If
      If (.HangPieceType = PT_QUEEN Or Weak) And .AttackerPieceType = PT_ROOK Then
        AddScore Score, ThreatByRook(.HangPieceType)
        If .HangPieceType <> PT_PAWN Then
          AddScoreVal Score, ThreatByRank.MG * RelRank, ThreatByRank.EG * RelRank
        End If
      End If
      If Score.MG <> 0 Or Score.EG <> 0 Then If .HangCol = COL_WHITE Then AddScore BThreat, Score Else AddScore WThreat, Score
    End With

lblNext:
  Next

End Sub

Public Sub AddWKingAttackers(ByVal AttackBit As Long)
 If AttackBit And PLAttackBit Then AddWKingAttack PT_PAWN
 If AttackBit And PRAttackBit Then AddWKingAttack PT_PAWN
 If AttackBit And N1AttackBit Then AddWKingAttack PT_KNIGHT
 If AttackBit And N2AttackBit Then AddWKingAttack PT_KNIGHT
 If AttackBit And B1AttackBit Then AddWKingAttack PT_BISHOP
 If AttackBit And B2AttackBit Then AddWKingAttack PT_BISHOP
 If AttackBit And BXrayAttackBit Then If Not (AttackBit And (B1AttackBit Or B2AttackBit)) Then WKingAttackersCount = WKingAttackersCount + 1
 If AttackBit And (R1AttackBit Or R1XrayAttackBit) Then AddWKingAttack PT_ROOK
 If AttackBit And (R2AttackBit Or R2XrayAttackBit) Then AddWKingAttack PT_ROOK
 If AttackBit And (QAttackBit Or QXrayAttackBit) Then AddWKingAttack PT_QUEEN
End Sub

Public Sub AddBKingAttackers(ByVal AttackBit As Long)
 If AttackBit And PLAttackBit Then AddBKingAttack PT_PAWN
 If AttackBit And PRAttackBit Then AddBKingAttack PT_PAWN
 If AttackBit And N1AttackBit Then AddBKingAttack PT_KNIGHT
 If AttackBit And N2AttackBit Then AddBKingAttack PT_KNIGHT
 If AttackBit And B1AttackBit Then AddBKingAttack PT_BISHOP
 If AttackBit And B2AttackBit Then AddBKingAttack PT_BISHOP
 If AttackBit And BXrayAttackBit Then If Not (AttackBit And (B1AttackBit Or B2AttackBit)) Then BKingAttackersCount = BKingAttackersCount + 1
 If AttackBit And (R1AttackBit Or R1XrayAttackBit) Then AddBKingAttack PT_ROOK
 If AttackBit And (R2AttackBit Or R2XrayAttackBit) Then AddBKingAttack PT_ROOK
 If AttackBit And (QAttackBit Or QXrayAttackBit) Then AddBKingAttack PT_QUEEN
End Sub

Public Sub AddWKingAttack(PT As enumPieceType)
  WKingAttackersCount = WKingAttackersCount + 1
  WKingAttackersWeight = WKingAttackersWeight + KingAttackWeights(PT)
End Sub

Public Sub AddBKingAttack(PT As enumPieceType)
  BKingAttackersCount = BKingAttackersCount + 1
  BKingAttackersWeight = BKingAttackersWeight + KingAttackWeights(PT)
End Sub

Public Function InitConnectedPawns()
  ' SF6
  Dim Seed(8) As Long, Opposed As Long, Phalanx As Long, Support As Long, r As Long, v As Long, x As Long
  ReadLngArr Seed(), 0, 0, 13, 24, 18, 76, 100, 175, 330
```

```
    For Opposed = 0 To 1
      For Phalanx = 0 To 1
        For Support = 0 To 2
          For r = 2 To 7
            If Phalanx > 0 Then x = (Seed(r + 1) - Seed(r)) / 2 Else x = 0
            v = 17 * Support
            v = v + Seed(r)
            If Phalanx > 0 Then v = v + (Seed(r + 1) - Seed(r)) \ 2
            If Opposed > 0 Then v = v / 2 '>> operator for opposed in VB: /2
            ConnectedBonus(Opposed, Phalanx, Support, r).MG = v
            ConnectedBonus(Opposed, Phalanx, Support, r).EG = v * ((r - 1) - 2) \ 4 'rank
            r ist zero based in C, so (r-1)
          Next
        Next
      Next
    Next

  End Function

  Public Sub InitImbalance()  'SF6
    '// pair pawn knight bishop rook queen  OUR PIECES
    ReadIntArr2 QuadraticOurs(), 0, 1667 'Bishop pair
    ReadIntArr2 QuadraticOurs(), PT_PAWN, 40, 0  'Pawn
    ReadIntArr2 QuadraticOurs(), PT_KNIGHT, 32, 255, -3          'Knight
    ReadIntArr2 QuadraticOurs(), PT_BISHOP, 0, 104, 4, 0         'Bishop
    ReadIntArr2 QuadraticOurs(), PT_ROOK, -26, -2, 47, 105, -149           'Rook
    ReadIntArr2 QuadraticOurs(), PT_QUEEN, -185, 24, 122, 137, -134, 0     'Queen
    '// pair pawn knight bishop rook queen  THEIR PIECES
    ReadIntArr2 QuadraticTheirs(), 0, 0 'Bishop pair
    ReadIntArr2 QuadraticTheirs(), PT_PAWN, 36, 0       'Pawn
    ReadIntArr2 QuadraticTheirs(), PT_KNIGHT, 9, 63, 0              'Knight
    ReadIntArr2 QuadraticTheirs(), PT_BISHOP, 59, 65, 42, 0         'Bishop
    ReadIntArr2 QuadraticTheirs(), PT_ROOK, 46, 39, 24, -24, 0             'Rook
    ReadIntArr2 QuadraticTheirs(), PT_QUEEN, 101, 100, -37, 141, 268, 0    'Queen
    '// PawnSet[pawn count] contains a bonus/malus indexed by number of pawns
    ReadIntArr PawnSet(), 24, -32, 107, -51, 117, -9, -126, -21, 31
  End Sub

  Public Function Imbalance() As Long 'SF
    Dim v As Long, Key As Long
    Key = CalcMaterialKey()
    Imbalance = ProbeMaterialHash(Key)
    If Imbalance <> VALUE_NONE Then Exit Function
    ImbPieceCount(COL_WHITE, 0) = Abs(PieceCnt(WBISHOP) > 1)  'index 0 used for bishop pair
    ImbPieceCount(COL_BLACK, 0) = Abs(PieceCnt(BBISHOP) > 1)  'index 0 used for bishop pair
    ImbPieceCount(COL_WHITE, PT_PAWN) = PieceCnt(WPAWN)
    ImbPieceCount(COL_BLACK, PT_PAWN) = PieceCnt(BPAWN)
    ImbPieceCount(COL_WHITE, PT_KNIGHT) = PieceCnt(WKNIGHT)
    ImbPieceCount(COL_BLACK, PT_KNIGHT) = PieceCnt(BKNIGHT)
    ImbPieceCount(COL_WHITE, PT_BISHOP) = PieceCnt(WBISHOP)
    ImbPieceCount(COL_BLACK, PT_BISHOP) = PieceCnt(BBISHOP)
    ImbPieceCount(COL_WHITE, PT_ROOK) = PieceCnt(WROOK)
    ImbPieceCount(COL_BLACK, PT_ROOK) = PieceCnt(BROOK)
    ImbPieceCount(COL_WHITE, PT_QUEEN) = PieceCnt(WQUEEN)
    ImbPieceCount(COL_BLACK, PT_QUEEN) = PieceCnt(BQUEEN)
    v = (ColImbalance(COL_WHITE) - ColImbalance(COL_BLACK)) \ 16
    'If Imbalance <> VALUE_NONE And Imbalance <> v Then MsgBox "Diff"
    Imbalance = v
    SaveMaterialHash Key, Imbalance
  End Function

  Public Function ColImbalance(ByVal Col As enumColor) As Long
    Dim Bonus As Long, pt1 As Long, pt2 As Long, Us As Long, Them As Long, v As Long
    If Col = COL_WHITE Then
      Us = COL_WHITE: Them = COL_BLACK: Bonus = PawnSet(PieceCnt(WPAWN))
      If PieceCnt(WQUEEN) = 1 Then If PieceCnt(BQUEEN) = 0 Then Bonus = Bonus +
      QueenMinorsImbalance(PieceCnt(BKNIGHT) + PieceCnt(BBISHOP))
    Else
      Us = COL_BLACK: Them = COL_WHITE: Bonus = PawnSet(PieceCnt(BPAWN))
      If PieceCnt(BQUEEN) = 1 Then If PieceCnt(WQUEEN) = 0 Then Bonus = Bonus +
      QueenMinorsImbalance(PieceCnt(WKNIGHT) + PieceCnt(WBISHOP))
    End If

    For pt1 = 0 To PT_QUEEN
      If ImbPieceCount(Us, pt1) > 0 Then
        v = 0

        For pt2 = 0 To pt1
          v = v + QuadraticOurs(pt1, pt2) * ImbPieceCount(Us, pt2) + QuadraticTheirs(pt1
          , pt2) * ImbPieceCount(Them, pt2)
        Next pt2

        Bonus = Bonus + ImbPieceCount(Us, pt1) * v
      End If
    Next pt1

    ColImbalance = Bonus
  End Function

  Public Sub AddScore(ScoreTotal As TScore, ScoreAdd As TScore)
    ScoreTotal.MG = ScoreTotal.MG + ScoreAdd.MG: ScoreTotal.EG = ScoreTotal.EG +
    ScoreAdd.EG
  End Sub

  Public Sub AddScoreWithFactor(ScoreTotal As TScore, ScoreAdd As TScore, Factor As Long
  )
    ScoreTotal.MG = ScoreTotal.MG + ScoreAdd.MG * Factor: ScoreTotal.EG = ScoreTotal.EG
    + ScoreAdd.EG * Factor
  End Sub

  'Public Sub AddScore100(ScoreTotal As TScore, ScoreAdd As TScore)
  ' ' Score 100 centipawns based: scale to SF pawn value
  '  ScoreTotal.MG = ScoreTotal.MG + (ScoreAdd.MG * ScorePawn.EG) \ 100&: ScoreTotal.EG = ScoreTotal.EG +
  (ScoreAdd.EG * ScorePawn.EG) \ 100&
  'End Sub

  Public Sub AddScoreVal(ScoreTotal As TScore, ByVal MGScore As Long, ByVal EGSCore As
  Long)
    ScoreTotal.MG = ScoreTotal.MG + MGScore: ScoreTotal.EG = ScoreTotal.EG + EGSCore
  End Sub

  Public Sub SetScoreVal(ScoreSet As TScore, ByVal MGScore As Long, ByVal EGSCore As
  Long)
    ScoreSet.MG = MGScore: ScoreSet.EG = EGSCore
  End Sub

  Public Function EvalSFTo100(ByVal Eval As Long) As Long
    If Abs(Eval) < MATE_IN_MAX_PLY Then EvalSFTo100 = (Eval * 100&) / CLng(ScorePawn.EG)
     Else EvalSFTo100 = Eval
  End Function

  Public Function Eval100ToSF(ByVal Eval As Long) As Long
    Eval100ToSF = (Eval * CLng(ScorePawn.EG)) / 100&
  End Function

  Public Sub MinusScore(ScoreTotal As TScore, ScoreMinus As TScore)
    ScoreTotal.MG = ScoreTotal.MG - ScoreMinus.MG: ScoreTotal.EG = ScoreTotal.EG -
    ScoreMinus.EG
  End Sub

  Public Function ScaleScore(Score As TScore) As Long
    ' Calculate score for game phase
    ScaleScore = Score.MG * GamePhase + Score.EG * CLng(PHASE_MIDGAME - GamePhase) ' *
    SF6 / SCALE_FACTOR_NORMAL
    ScaleScore = ScaleScore \ PHASE_MIDGAME
```

```
End Function

'Public Function ScaleScore100(Score As TScore, ByVal ScaleVal As Long) As TScore
'  ScaleScore100.MG = (Score.MG * ScaleVal) \ 100&: ScaleScore100.EG = (Score.EG * ScaleVal) \ 100&
'End Function

Public Function ShowScore(Score As TScore) As String
  ' show MG, EG Score as text
  ShowScore = "(" & CStr(Score.MG) & "," & CStr(Score.EG) & ")=" & ScaleScore(Score)
End Function

Public Function ShowScoreDiff100(Score1 As TScore, Score2 As TScore) As String
  ' show MG, EG Score as text
  Dim Diff As TScore
  Diff.MG = Score1.MG - Score2.MG: Diff.EG = Score1.EG - Score2.EG
  ShowScoreDiff100 = ShowScore(Diff)
End Function

Public Function PieceSQ(ByVal Side As enumColor, _
                        ByVal SearchPieceType As enumPieceType) As Long
  Dim a As Long, p As Long

  For a = 1 To NumPieces
    p = Board(Pieces(a)): If PieceType(p) = SearchPieceType And PieceColor(p) = Side Then PieceSQ = Pieces(a): Exit Function
  Next

End Function

Public Function Eval_KRKP() As Long
  Dim WKSq          As Long, BKSq As Long, RookSq As Long, PawnSq As Long, StrongSide As enumColor, WeakSide As enumColor
  Dim StrongKingLoc As Long, WeakKingLoc As Long, QueeningSq As Long, Result As Long, SideToMove As enumColor
  If WMaterial > BMaterial Then
    StrongSide = COL_WHITE: WeakSide = COL_BLACK: StrongKingLoc = WKingLoc: WeakKingLoc = BKingLoc
  Else
    StrongSide = COL_BLACK: WeakSide = COL_WHITE: StrongKingLoc = BKingLoc: WeakKingLoc = WKingLoc
  End If
  If bWhiteToMove Then SideToMove = COL_WHITE Else SideToMove = COL_BLACK
  WKSq = RelativeSq(StrongSide, StrongKingLoc)
  BKSq = RelativeSq(StrongSide, WeakKingLoc)
  RookSq = RelativeSq(StrongSide, PieceSQ(StrongSide, PT_ROOK))
  PawnSq = RelativeSq(WeakSide, PieceSQ(WeakSide, PT_PAWN))
  QueeningSq = SQ_A1 + File(PawnSq) - 1 + 7 * SQ_UP
  '-- If the stronger side's king is in front of the pawn, it's a win
  If WKSq < PawnSq And File(WKSq) = File(PawnSq) Then
    Result = ScoreRook.EG - MaxDistance(WKSq, PawnSq)
    '-- If the weaker side's king is too far from the pawn and the rook, it's a win.
  ElseIf MaxDistance(BKSq, PawnSq) >= (3 + Abs(SideToMove = WeakSide)) And MaxDistance(BKSq, RookSq) >= 3 Then
    Result = ScoreRook.EG - MaxDistance(WKSq, PawnSq)
    '-- If the pawn is far advanced and supported by the defending king, the position is drawish
  ElseIf Rank(BKSq) <= 3 And MaxDistance(BKSq, PawnSq) = 1 And Rank(WKSq) >= 4 And MaxDistance(WKSq, PawnSq) > (2 + Abs(SideToMove = StrongSide)) Then
    Result = 80 - 8 * MaxDistance(WKSq, PawnSq)
  Else
    Result = 200 - 8 * (MaxDistance(WKSq, PawnSq + SQ_DOWN) - MaxDistance(BKSq, PawnSq + SQ_DOWN) - MaxDistance(PawnSq, QueeningSq))
  End If
  If StrongSide = SideToMove Then Eval_KRKP = Result Else Eval_KRKP = -Result
  If Not bWhiteToMove Then Eval_KRKP = -Eval_KRKP
End Function

Public Function Eval_KQKP() As Long
  ' KQ vs KP. In general, this is a win for the stronger side, but there are a
  ' few important exceptions. A pawn on 7th rank and on the A,C,F or H files
  ' with a king positioned next to it can be a draw, so in that case, we only
  ' use the distance between the kings.
  Dim WinnerKSq As Long, LoserKSq As Long, PawnSq As Long, StrongSide As enumColor, WeakSide As enumColor
  Dim Result    As Long, SideToMove As enumColor
  If WMaterial > BMaterial Then
    StrongSide = COL_WHITE: WeakSide = COL_BLACK: WinnerKSq = WKingLoc: LoserKSq = BKingLoc
  Else
    StrongSide = COL_BLACK: WeakSide = COL_WHITE: WinnerKSq = BKingLoc: LoserKSq = WKingLoc
  End If
  PawnSq = PieceSQ(WeakSide, PT_PAWN)
  If bWhiteToMove Then SideToMove = COL_WHITE Else SideToMove = COL_BLACK
  Result = PushClose(MaxDistance(WinnerKSq, LoserKSq))
  If RelativeRank(WeakSide, PawnSq) <> 7 Or MaxDistance(LoserKSq, PawnSq) <> 1 Then
    Result = Result + ScoreQueen.EG - ScorePawn.EG
  Else

    Select Case File(PawnSq) ' For File A,C,F,H
      Case 2, 4, 5, 7: Result = Result + ScoreQueen.EG - ScorePawn.EG
    End Select

  End If
  If StrongSide = SideToMove Then Eval_KQKP = Result Else Eval_KQKP = -Result
  If Not bWhiteToMove Then Eval_KQKP = -Eval_KQKP
End Function

Public Function Eval_KQKR() As Long
  Dim WinnerKSq As Long, LoserKSq As Long, StrongSide As enumColor, WeakSide As enumColor
  Dim Result    As Long, SideToMove As enumColor
  If WMaterial > BMaterial Then
    StrongSide = COL_WHITE: WeakSide = COL_BLACK: WinnerKSq = WKingLoc: LoserKSq = BKingLoc
  Else
    StrongSide = COL_BLACK: WeakSide = COL_WHITE: WinnerKSq = BKingLoc: LoserKSq = WKingLoc
  End If
  If bWhiteToMove Then SideToMove = COL_WHITE Else SideToMove = COL_BLACK
  Result = ScoreQueen.EG - ScoreRook.EG + PushToEdges(LoserKSq) + PushClose(MaxDistance(WinnerKSq, LoserKSq))
  If StrongSide = SideToMove Then Eval_KQKR = Result Else Eval_KQKR = -Result
  If Not bWhiteToMove Then Eval_KQKR = -Eval_KQKR
End Function

Private Function WKingShelterStorm(ShelterKingLoc As Long) As Long
  Dim Center As Long, k As Long, r As Long, RelFile As Long, Safety As Long, RankUs As Long, RankThem As Long, RankNum As Long
  Safety = 258
  ' Opp pawn rank A/H protects king
  If File(WKingLoc) = 1 Or File(WKingLoc) = 8 Then
    If Rank(WKingLoc) <= 2 Then If Board(WKingLoc + SQ_UP) = BPAWN Then Safety = 350
  End If

  '--- Pawn shelter
  Center = GetMax(2, GetMin(7, File(ShelterKingLoc))): RankNum = Rank(ShelterKingLoc) ' FIle A=>B, File H=>G

  For k = Center - 1 To Center + 1
    ' Pawn shelter/storm
    RankUs = 1
    If WPawns(k) > 0 Then If PawnsWMin(k) >= RankNum Then RankUs = PawnsWMin(k)
    RankThem = 1
    If BPawns(k) > 0 Then If PawnsBMin(k) >= RankNum Then RankThem = PawnsBMin(k)
    If RankThem = RankNum + 1 And k = File(ShelterKingLoc) Then
      r = 1 ' BlockedByKing
    ElseIf RankUs = 1 Then
      r = 2 ' NoFriendlyPawn
    ElseIf RankThem = RankUs + 1 Then
      r = 3 ' BlockedByPawn
    Else
      r = 4 ' Unblocked
    End If
    RelFile = GetMin(k, 9 - k)
    Safety = Safety - (ShelterWeakness(RelFile, RankUs) + StormDanger(r, RelFile,
    RankThem))
  Next

  If Center >= 6 Then
    If Board(SQ_H3) = BPAWN Then
      If Board(SQ_H2) = WPAWN Then If Board(SQ_G3) = WPAWN Then If Board(SQ_F2) =
      WPAWN Then Safety = Safety + 300
    End If
    If Board(SQ_F3) = BPAWN Then
      If Board(SQ_H2) = WPAWN Then If Board(SQ_G3) = WPAWN Then If Board(SQ_F2) =
      WPAWN Then Safety = Safety + 300
    End If
  ElseIf Center <= 3 Then
    If Board(SQ_A3) = BPAWN Then
      If Board(SQ_A2) = WPAWN Then If Board(SQ_B3) = WPAWN Then If Board(SQ_C2) =
      WPAWN Then Safety = Safety + 300
    End If
    If Board(SQ_C3) = BPAWN Then
      If Board(SQ_A2) = WPAWN Then If Board(SQ_B3) = WPAWN Then If Board(SQ_C2) =
      WPAWN Then Safety = Safety + 300
    End If
  End If

  WKingShelterStorm = Safety
End Function

Private Function BKingShelterStorm(ByVal ShelterKingLoc As Long) As Long
  Dim Center As Long, k As Long, r As Long, RelFile As Long, Safety As Long, RankUs As
   Long, RankThem As Long, RankNum As Long
  Safety = 258
  ' Opp pawn rank A/H protects king
  If File(BKingLoc) = 1 Or File(BKingLoc) = 8 Then
    If Rank(BKingLoc) >= 7 Then If Board(BKingLoc + SQ_DOWN) = WPAWN Then Safety = 350
  End If  '--- Pawn shelter
  Center = GetMax(2, GetMin(7, File(ShelterKingLoc))): RankNum = 9 - Rank(
  ShelterKingLoc)  ' FIle A=>B, File H=>G

  For k = Center - 1 To Center + 1
    ' Pawn shelter/storm
    RankUs = 1
    If BPawns(k) > 0 Then If 9 - PawnsBMax(k) >= RankNum Then RankUs = (9 - PawnsBMax(
    k))
    RankThem = 1
    If WPawns(k) > 0 Then If 9 - PawnsWMax(k) >= RankNum Then RankThem = (9 -
    PawnsWMax(k))
    If RankThem = RankNum + 1 And k = File(ShelterKingLoc) Then
      r = 1 ' BlockedByKing
    ElseIf RankUs = 1 Then
      r = 2 ' NoFriendlyPawn
    ElseIf RankThem = RankUs + 1 Then
      r = 3 ' BlockedByPawn
    Else
      r = 4 ' Unblocked
    End If
    RelFile = GetMin(k, 9 - k)
    Safety = Safety - (ShelterWeakness(RelFile, RankUs) + StormDanger(r, RelFile,
    RankThem))
  Next
  If Center >= 6 Then
```

```
      If Board(SQ_H6) = WPAWN Then
        If Board(SQ_H7) = BPAWN Then If Board(SQ_G6) = BPAWN Then If Board(SQ_F7) =
        BPAWN Then Safety = Safety + 250
      End If
      If Board(SQ_F6) = WPAWN Then
        If Board(SQ_H7) = BPAWN Then If Board(SQ_G6) = BPAWN Then If Board(SQ_F7) =
        BPAWN Then Safety = Safety + 150
      End If
    ElseIf Center <= 3 Then
      If Board(SQ_A6) = WPAWN Then
        If Board(SQ_A7) = BPAWN Then If Board(SQ_B6) = BPAWN Then If Board(SQ_C7) =
        BPAWN Then Safety = Safety + 250
      End If
      If Board(SQ_C6) = WPAWN Then
        If Board(SQ_A7) = BPAWN Then If Board(SQ_B6) = BPAWN Then If Board(SQ_C7) =
        BPAWN Then Safety = Safety + 150
      End If
    End If

    BKingShelterStorm = Safety
  End Function

  Private Sub GetKingFlankFiles(ByVal KingLoc As Long, FileFrom As Long, FileTo As Long)

    Select Case File(KingLoc)
      Case 1 To 3: FileFrom = FILE_A: FileTo = FILE_D  ' File A-C> A-D
      Case 4 To 5: FileFrom = FILE_C: FileTo = FILE_F  ' File D-E> C-F
      Case 6 To 8: FileFrom = FILE_E: FileTo = FILE_H  ' File F-H> E-H
    End Select

  End Sub

  'Public Function PinnedPieceDir(ByVal PinnedLoc As Long, ByVal MoveTarget As Long, PieceCol As enumColor) As
  Long
  ''  check if a piece is pinned to king and returns direction offset from piece to king, if not pinned =  0
  ' PinnedPieceDir = 0
  ' If PieceCol = COL_WHITE Then
  '  PinnedPieceDir = WPinnedPieceDir(PinnedLoc)
  '  If PinnedPieceDir <> 0 Then
  '   If SameXRay(MoveTarget, WKingLoc) Then PinnedPieceDir = 0 ' move in pinned direction Ok
  '  End If
  ' ElseIf PieceCol = COL_BLACK Then
  '  PinnedPieceDir = BPinnedPieceDir(PinnedLoc)
  '  If PinnedPieceDir <> 0 Then
  '   If SameXRay(MoveTarget, BKingLoc) Then PinnedPieceDir = 0 ' move in pinned direction Ok
  '  End If
  ' End If
  ''If PinnedPieceDir <> 0 Then Stop
  'End Function

  Public Function WPinnedPieceDir(ByVal PinnedLoc As Long) As Long
    '-- check if a piece is pinned to king and returns direction offset from piece to king, if not pinned =  0
    Dim k As Long, sq As Long, Offset As Long, Piece As Long
    WPinnedPieceDir = 0
    If PinnedLoc = WKingLoc Then Exit Function
    Offset = DirOffset(PinnedLoc, WKingLoc)   ' Find direction to king
    If Offset = 0 Then Exit Function

    ' no other piece between piece and own king?
    sq = PinnedLoc
    For k = 1 To 7
      sq = sq + Offset: If sq = WKingLoc Then Exit For ' pinned possible
      Piece = Board(sq)  ': If Piece = FRAME Then Exit For ' should not happen
      If Piece < NO_PIECE Then Exit Function ' other piece found > not pinned
    Next k

    ' check other direction for attacker
    sq = PinnedLoc
    For k = 1 To 7
      sq = sq - Offset
      Piece = Board(sq): If Piece = FRAME Then Exit For
      If Piece < NO_PIECE Then
        Select Case Piece
        Case BQUEEN:
          WPinnedPieceDir = Offset: Exit Function ' pinned by queen
        Case BROOK:
          If Abs(Offset) = 10 Or Abs(Offset) = 1 Then WPinnedPieceDir = Offset: Exit
          Function ' pinned by rook
        Case BBISHOP:
          If Abs(Offset) = 9 Or Abs(Offset) = 11 Then WPinnedPieceDir = Offset: Exit
          Function ' pinned by bishop
        End Select
        Exit Function ' other piece found
      End If
    Next k
    ' not pinned here
  End Function

  Public Function BPinnedPieceDir(ByVal PinnedLoc As Long) As Long
    '-- check if a piece is pinned to king and returns direction offset from piece to king, if not pinned =  0
    Dim k As Long, sq As Long, Offset As Long, Piece As Long
    BPinnedPieceDir = 0
    If PinnedLoc = BKingLoc Then Exit Function
    Offset = DirOffset(PinnedLoc, BKingLoc)   ' Find direction to king
    If Offset = 0 Then Exit Function

    ' no other piece between piece and own king?
    sq = PinnedLoc
    For k = 1 To 7
      sq = sq + Offset: If sq = BKingLoc Then Exit For ' pinned possible
      Piece = Board(sq) ': If Piece = FRAME Then Exit For ' should not happen
      If Piece < NO_PIECE Then Exit Function ' other piece found > not pinned
    Next k

    ' check other direction for attacker
    sq = PinnedLoc
    For k = 1 To 7
      sq = sq - Offset
      Piece = Board(sq): If Piece = FRAME Then Exit For
      If Piece < NO_PIECE Then
        Select Case Piece
        Case WQUEEN:
          BPinnedPieceDir = Offset: Exit Function ' pinned by queen
        Case WROOK:
          If Abs(Offset) = 10 Or Abs(Offset) = 1 Then BPinnedPieceDir = Offset: Exit
          Function ' pinned by rook
        Case WBISHOP:
          If Abs(Offset) = 9 Or Abs(Offset) = 11 Then BPinnedPieceDir = Offset: Exit
          Function ' pinned by bishop
        End Select
        Exit Function ' other piece found
      End If
    Next k
    ' not pinned here
  End Function

  'Public Function PinnedPieceW(ByVal PinnedLoc As Long, ByVal Direction As Long) As Boolean
  ' ' white pieces it threatend by pinned pieces and slider attack?
  '  Dim k As Long, sq As Long, Offset As Long, AttackBit As Long, Piece As Long
  '  PinnedPieceW = False
  '  If Direction < 4 Then ' Queen or rook orthogonal
  '    If Not CBool(BAttack(PinnedLoc) And QRAttackBit) Then Exit Function
  '    AttackBit = QRAttackBit
  '  Else ' Queen or bishop diagonal
  '    If Not CBool(BAttack(PinnedLoc) And QBAttackBit) Then Exit Function
  '    AttackBit = QBAttackBit
```

```
' End If
' Offset = DirectionOffset(Direction)
'
' For k = 1 To 8
'   sq = PinnedLoc + Offset * k: Piece = Board(sq)
'   If Piece = FRAME Then Exit For
'   If Piece < NO_PIECE Then
'     If Piece = BQUEEN Then PinnedPieceW = True: Exit Function
'     If Piece = BROOK Then If Direction < 4 Then PinnedPieceW = True: Exit Function
'     If Piece = BBISHOP Then If Direction >= 4 Then PinnedPieceW = True: Exit Function
'     Exit For
'   Else
'     If Not (CBool(BAttack(sq) And AttackBit)) Then Exit For
'   End If
' Next k
'
'End Function
'
'Public Function PinnedPieceB(ByVal PinnedLoc As Long, ByVal Direction As Long) As Boolean
' ' black pieces it threatend by pinned pieces and slider attack?
' Dim k As Long, sq As Long, Offset As Long, AttackBit As Long, Piece As Long
' PinnedPieceB = False
' If Direction < 4 Then ' Queen or rook orthogonal
'   If Not CBool(WAttack(PinnedLoc) And QRAttackBit) Then Exit Function
'   AttackBit = QRAttackBit
' Else ' Queen or bishop diagonal
'   If Not CBool(WAttack(PinnedLoc) And QBAttackBit) Then Exit Function
'   AttackBit = QBAttackBit
' End If
' Offset = DirectionOffset(Direction)
'
' For k = 1 To 8
'   sq = PinnedLoc + Offset * k: Piece = Board(sq)
'   If Piece = FRAME Then Exit For
'   If Piece < NO_PIECE Then
'     If Piece = WQUEEN Then PinnedPieceB = True: Exit Function
'     If Piece = WROOK Then If Direction < 4 Then PinnedPieceB = True: Exit Function
'     If Piece = WBISHOP Then If Direction >= 4 Then PinnedPieceB = True: Exit Function
'     Exit For
'   Else
'     If Not (CBool(WAttack(sq) And AttackBit)) Then Exit For
'   End If
' Next k
'
'End Function

Public Sub InitOutpostSq()
  Dim sq As Long

  For sq = SQ_A1 To SQ_H8
    If Rank(sq) >= 4 And Rank(sq) <= 6 Then WOutpostSq(sq) = True
    If Rank(sq) >= 3 And Rank(sq) <= 5 Then BOutpostSq(sq) = True
  Next sq

End Sub

'Public Function NonPawnMatForSide(ByVal UseColOfSideToMove As Boolean) As Long
' If UseColOfSideToMove Then
'   If bWhiteToMove Then
'     NonPawnMatForSide = PieceCnt(WQUEEN) * ScoreQueen.MG + PieceCnt(WROOK) * ScoreRook.MG +
PieceCnt(WBISHOP) * ScoreBishop.MG + PieceCnt(WKNIGHT) * ScoreKnight.MG
'   Else
'     NonPawnMatForSide = PieceCnt(BQUEEN) * ScoreQueen.MG + PieceCnt(BROOK) * ScoreRook.MG +
PieceCnt(BBISHOP) * ScoreBishop.MG + PieceCnt(BKNIGHT) * ScoreKnight.MG
'   End If
' Else
'   If Not bWhiteToMove Then
'     NonPawnMatForSide = PieceCnt(WQUEEN) * ScoreQueen.MG + PieceCnt(WROOK) * ScoreRook.MG +
PieceCnt(WBISHOP) * ScoreBishop.MG + PieceCnt(WKNIGHT) * ScoreKnight.MG
'   Else
'     NonPawnMatForSide = PieceCnt(BQUEEN) * ScoreQueen.MG + PieceCnt(BROOK) * ScoreRook.MG + PieceCnt(BBISHOP) * ScoreBishop.MG + PieceCnt(BKNIGHT) * ScoreKnight.MG
'   End If
' End If
'End Function

Public Function NonPawnMat() As Long
  NonPawnMat = (PieceCnt(WQUEEN) + PieceCnt(BQUEEN)) * ScoreQueen.MG + (PieceCnt(WROOK) + PieceCnt(BROOK)) * ScoreRook.MG + (PieceCnt(WBISHOP) + PieceCnt(BBISHOP)) * ScoreBishop.MG + (PieceCnt(WKNIGHT) + PieceCnt(BKNIGHT)) * ScoreKnight.MG
End Function

'Public Function MaterialTotal() As Long
' ' from view of white
' MaterialTotal = (PieceCnt(WQUEEN) - PieceCnt(BQUEEN)) * ScoreQueen.MG + (PieceCnt(WROOK) - PieceCnt(BROOK)) * ScoreRook.MG + (PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) * ScoreBishop.MG + (PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT)) * ScoreKnight.MG + (PieceCnt(WPAWN) - PieceCnt(BPAWN)) * ScorePawn.MG
'End Function

'Public Function PositionalDiff(ByVal Score As Long) As Long
' ' from view of white: absolute difference material piece values / Total sccore = Positional score
' If Score = VALUE_NONE Then PositionalDiff = 0: Exit Function
' Dim MatScore As Long
' MatScore = (PieceCnt(WQUEEN) - PieceCnt(BQUEEN)) * ScoreQueen.MG + (PieceCnt(WROOK) - PieceCnt(BROOK)) * ScoreRook.MG + (PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) * ScoreBishop.MG + (PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT)) * ScoreKnight.MG + (PieceCnt(WPAWN) - PieceCnt(BPAWN)) * ScorePawn.MG
' PositionalDiff = Abs(Score - MatScore)
'End Function

Public Sub CheckWQueenWeek(ByVal sq As Long, ByVal Offset As Long, ByVal Direction As Long, ByRef Result As Boolean)
  ' Queen pinned or discovered threat possible
  If Result Then Exit Sub ' count only once
  Dim r As Long
  Result = False: r = sq + Offset ' next sq in same direction
  Select Case Board(r)
  Case BROOK: If Direction < 4 Then Result = True
  Case BBISHOP: If Direction > 3 Then Result = True
  Case NO_PIECE:
    If Direction < 4 Then ' Rook
      ' 2nd part: compare both attackbits, may be from different rooks: R1Attackbit or R2Attackbit
      If CBool(BAttack(sq) And RAttackBit) Then If (BAttack(r) And RAttackBit) = (BAttack(sq) And RAttackBit) Then Result = True
    Else ' Bishop?
      If CBool(BAttack(sq) And BAttackBit) Then If (BAttack(r) And BAttackBit) = (BAttack(sq) And BAttackBit) Then Result = True
    End If
  End Select
End Sub

Public Sub CheckBQueenWeek(ByVal sq As Long, ByVal Offset As Long, ByVal Direction As Long, ByRef Result As Boolean)
  ' Queen pinned or discovered threat possible
  If Result Then Exit Sub ' count only once
  Dim r As Long
  Result = False: r = sq + Offset ' next sq in same direction
  Select Case Board(r)
  Case WROOK: If Direction < 4 Then Result = True
  Case WBISHOP: If Direction > 3 Then Result = True
  Case NO_PIECE:
    If Direction < 4 Then ' Rook
      ' 2nd part: compare both attackbits, may be from different rooks: R1Attackbit or R2Attackbit
      If CBool(WAttack(sq) And RAttackBit) Then If (WAttack(r) And RAttackBit) = (WAttack(sq) And RAttackBit) Then Result = True
    Else ' Bishop?
      If CBool(WAttack(sq) And BAttackBit) Then If (WAttack(r) And BAttackBit) = (
      WAttack(sq) And BAttackBit) Then Result = True
    End If
  End Select
End Sub

'Public Function DistToOppKing(Piece As Integer, Target As Integer) As Long
'  If PieceColor(Piece) = COL_WHITE Then
'    DistToOppKing = MaxDistance(BKingLoc, Target)
'  Else
'    DistToOppKing = MaxDistance(WKingLoc, Target)
'  End If
'End Function

'Public Function CalcSimpleEval() As Long
'  Dim WNonPawnMaterial As Long, BNonPawnMaterial As Long
'  WNonPawnMaterial = PieceCnt(WQUEEN) * ScoreQueen.MG + PieceCnt(WROOK) * ScoreRook.MG +
PieceCnt(WBISHOP) * ScoreBishop.MG + PieceCnt(WKNIGHT) * ScoreKnight.MG
'  BNonPawnMaterial = PieceCnt(BQUEEN) * ScoreQueen.MG + PieceCnt(BROOK) * ScoreRook.MG +
PieceCnt(BBISHOP) * ScoreBishop.MG + PieceCnt(BKNIGHT) * ScoreKnight.MG
'
'  CalcSimpleEval = ScorePawn.EG * (PieceCnt(WPAWN) - PieceCnt(BPAWN)) + (WNonPawnMaterial -
BNonPawnMaterial)
'  If Not bWhiteToMove Then CalcSimpleEval = -CalcSimpleEval
'End Function
VERSION 5.00
Begin {C62A69F0-16DC-11CE-9E98-00AA00574A4F} frmChessX
   Caption         =   "ChessBrainVBA 3.03"
   ClientHeight    =   10500
   ClientLeft      =   45
   ClientTop       =   375
   ClientWidth     =   15915
   OleObjectBlob   =   "frmChessX.frx":0000
   ShowModal       =   0    'False
   StartUpPosition =   3   'Windows-Standard
End
Attribute VB_Name = "frmChessX"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
'==================================================================
'= VBAChessBrainX, a chess playing winboard engine by Roger Zuehlsdorf (Copyright 2015)
'= and is based on LarsenVb by Luca Dormio(http://xoomer.virgilio.it/ludormio/download.htm)
'=
'= VBAChessBrainX is free software: you can redistribute it and/or modify
'= it under the terms of the GNU General Public License as published by
'= the Free Software Foundation, either version 3 of the License, or
'= (at your option) any later version.
'=
'= VBAChessBrainX is distributed in the hope that it will be useful,
'= but WITHOUT ANY WARRANTY; without even the implied warranty of
'= MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
'= GNU General Public License for more details.
'=
'= You should have received a copy of the GNU General Public License
'= along with this program.  If not, see <http://www.gnu.org/licenses/>.
'==================================================================

Option Explicit


' GUI controls
Dim oField(1 To 64) As Control
Dim oFieldEvents(1 To 64) As clsBoardField
Dim oLabelsX(1 To 8) As Control
Dim oLabelsX2(1 To 8) As Control
Dim oLabelsY(1 To 8) As Control
Dim oLabelsY2(1 To 8) As Control
Dim oPiecePics(1 To 12) As Control
Dim oPieceCnt(1 To 6) As Control

Dim i As Long






Private Sub chkFlipBoard_Change()
 If chkFlipBoard.Value = True Then
   FlipBoard False
 Else
   FlipBoard True
 End If
End Sub

Private Sub chkShowThinking_Change()
  txtIO.Visible = chkShowThinking
End Sub


Private Sub chkTableBases_Click()
 If chkTableBases.Value = True Then
  TableBasesRootEnabled = True
  WriteINISetting "TB_ROOT_ENABLED", "1"
  TableBasesSearchEnabled = True
  WriteINISetting "TB_SEARCH_ENABLED", "1"
  optSecondsPerMove.Value = 1
  cboSecondsPerMove.Value = "30"  ' Min 20 sec fxFCr EGTB Init needed
 Else
  TableBasesRootEnabled = False
  WriteINISetting "TB_ROOT_ENABLED", "0"
  TableBasesSearchEnabled = False
  WriteINISetting "TB_SEARCH_ENABLED", "0"
 End If
End Sub

Private Sub cmdClearBoard_Click()
 Dim i As Integer
 For i = SQ_A1 To SQ_H8
   If Board(i) <> FRAME Then Board(i) = NO_PIECE
 Next
 ShowBoard
End Sub

Private Sub cmdClearCommand_Click()
  cboFakeInput = ""
End Sub


Private Sub SelectPiece(PieceType As Integer)
  Dim i As Integer
  For i = 1 To 12: Me.Controls("Piece" & CStr(i)).SpecialEffect = 0: Next
  SetupPiece = PieceType
  Me.Controls("Piece" & CStr(PieceType)).SpecialEffect = 3
End Sub

Private Sub cmdEndSetup_Click()
  Dim i As Integer, WKCnt As Integer, BKCnt As Integer, bPosLegal As Boolean

  ' Is position legal?
  bPosLegal = True: WKCnt = 0: BKCnt = 0
  For i = SQ_A1 To SQ_H8
    Select Case Board(i)
```

```
      Case WKING: WKCnt = WKCnt + 1: If WKCnt > 1 Then bPosLegal = False: MsgBox
      Translate("Illegal positition: only one White King allowed!")
      Case BKING: BKCnt = BKCnt + 1: If BKCnt > 1 Then bPosLegal = False: MsgBox
      Translate("Illegal positition: only one Black King allowed!")
      Case WPAWN, BPAWN: If Rank(i) = 1 Or Rank(i) = 8 Then bPosLegal = False:: MsgBox
      Translate("Illegal positition: Pawn rank must between 2 and 7!")
      End Select
    Next
    If WKCnt = 0 Then bPosLegal = False: MsgBox Translate("Illegal positition: White
    King needed!")
    If BKCnt = 0 Then bPosLegal = False: MsgBox Translate("Illegal positition: Black
    King needed!")
    If Not bPosLegal Then Exit Sub

    SetupBoardMode = False
    cmdClearBoard.Visible = False
    cmdEndSetup.Visible = False
    chkWOO.Visible = False
    chkWOOO.Visible = False
    chkBOO.Visible = False
    chkBOOO.Visible = False
    lblSelectPiece.Visible = False
    cmdSetup.Visible = True

    ' Init data
    Erase arFiftyMove()
    Fifty = 0
    Erase Moved()

    OpeningHistory = " "
    BookPly = BOOK_MAX_PLY + 1 ' no book

    ' Castling
    WhiteCastled = NO_CASTLE
    BlackCastled = NO_CASTLE
    If Not chkWOO.Value Then Moved(SQ_H1) = 1 ' Rook moved flag
    If Not chkWOOO.Value Then Moved(SQ_A1) = 1 ' Rook moved flag
    If Not chkBOO.Value Then Moved(SQ_H8) = 1 ' Rook moved flag
    If Not chkBOOO.Value Then Moved(SQ_A8) = 1 ' Rook moved flag

    InitPieceSquares
    GameMovesCnt = 0
    HintMove = EmptyMove
    GamePosHash(GameMovesCnt) = HashBoard(EmptyMove) ' for 3x repetition draw
    ShowMoveList
    ShowBoard
    psLastFieldClick = "": psFieldFrom = "": psFieldTarget = "": plFieldFrom = 0:
    plFieldTarget = 0
  End Sub

  Private Sub cmdHint_Click()
    If HintMove.From > 0 Then
      If Board(HintMove.From) <> NO_PIECE Then
        SendCommand ">" & Translate("Hint") & ": " & MoveText(HintMove)
        ResetGUIFieldColors
        ShowMove HintMove.From, HintMove.Target
        DoEvents
        Sleep 2000
        ResetGUIFieldColors
      End If
    End If
  End Sub

  Private Sub cmdSetup_Click()
    If cmdStop.Visible Then Exit Sub ' Thinking
    SetupBoardMode = True
    cmdClearBoard.Visible = True
    cmdEndSetup.Visible = True
    lblSelectPiece.Visible = True
    chkWOO.Visible = True: chkWOO = False
    chkWOOO.Visible = True: chkWOOO = False
    chkBOO.Visible = True: chkBOO = False
    chkBOOO.Visible = True: chkBOOO = False
    cmdSetup.Visible = False
    txtIO = Translate("Select piece and click at square")
  End Sub

  Private Sub cmdSwitchSideToMove_Click()
    If cmdStop.Visible = True Then Exit Sub
    bWhiteToMove = Not bWhiteToMove
    ShowColToMove
  End Sub



  Private Sub cmdTestPos1_Click()
   ' Read from INI or use default
   optSecondsPerMove.Value = 1
   If cboSecondsPerMove.Value < 8 Then cboSecondsPerMove.Value = "8"
   cboFakeInput.Text = "setboard " & ReadINISetting("TEST_POSITION1",
   "1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - bm Re8+; id WAC.250;Mate in 8;")
   cmdFakeInput_Click
  End Sub

  Private Sub cmdTestPos2_Click()
   optSecondsPerMove.Value = 1
   If cboSecondsPerMove.Value < 10 Then cboSecondsPerMove.Value = "10"
   cboFakeInput.Text = "setboard " & ReadINISetting("TEST_POSITION2",
   "2k4B/bpp1qp2/p1b5/7p/1PN1n1p1/2Pr4/P5PP/R3QR1K b - - bm Ng3+; id WAC.273;")
   cmdFakeInput_Click
  End Sub

  Private Sub cmdTestPos3_Click()
   optSecondsPerMove.Value = 1
   If cboSecondsPerMove.Value < 10 Then cboSecondsPerMove.Value = "10"
   cboFakeInput.Text = "setboard " & ReadINISetting("TEST_POSITION3",
   "r3q2r/2p1k1p1/p5p1/1p2Nb2/1P2nB2/P7/2PNQbPP/R2R3K b - - bm Rxh2+; id WAC.266;")
  ' cboFakeInput.Text = "setboard 8/5P2/8/4K3/2k5/8/8/8 w - -"  ' Promote test
   cmdFakeInput_Click
  End Sub

  Private Sub cmdTestPos4_Click()
   cboFakeInput.Text = "setboard " & ReadINISetting("TEST_POSITION4",
   "8/6k1/6p1/8/7r/3P1KP1/8/8 w - - 0 1 ; Tablebase test;")
   optSecondsPerMove.Value = 1
   cboSecondsPerMove.Value = "30"  ' Min 20 sec fxFCr EGTB Init needed
   cmdFakeInput_Click
  End Sub

  Private Sub cmdWriteFEN_Click()
   Dim s As String, r As String
   s = WriteEPD()
   r = InputBox(Translate("please copy"), Translate("EPD position string"), s)
  End Sub

  Private Sub cmdZoomMinus_Click()
    If Me.Zoom > 30 Then
      Me.Zoom = Me.Zoom - 5
      Me.Width = Me.Width * 0.95
      Me.Height = Me.Height * 0.95
    End If
  End Sub

  Private Sub cmdZoomPlus_Click()
    Me.Zoom = Me.Zoom + 5
    Me.Width = Me.Width * 1.05
    Me.Height = Me.Height * 1.05
  End Sub







  Private Sub imgLangDE_Click()
    ' Translate to german
    WriteINISetting "LANGUAGE", "DE"
    InitTranslate
    TranslateForm
    ShowBoard
  End Sub

  Private Sub imgLangEN_Click()
    ' Translate to english
    WriteINISetting "LANGUAGE", "EN"
    InitTranslate
    TranslateForm
    ShowBoard
    MsgBox "Please restart for english"
  End Sub

  Private Sub UserForm_Initialize()
    ' GUI Start: Init
    ' Application.Workbooks.Parent.Visible = False ' Don't show EXCEL
    SetVBAPathes
    ReadColors
    CreateBoard
    LoadPiecesPics
    InitTimes
    InitTestSets

    InitEngine
    InitGame
    TranslateForm
    ShowBoard
    chkTableBases.Value = TableBasesRootEnabled
    Me.Show
  End Sub

  Public Sub cmdThink_Click()
    '
    '--- Start thinking for computer move
    '
    Static bThinking As Boolean
    If bThinking Or SetupBoardMode Then Exit Sub
    bThinking = True
    txtIO = ""

    SetTimeControl

    bPostMode = True
    bForceMode = False
    Result = NO_MATE

    If bWhiteToMove And optBlack = False Then optBlack = True

    If bWhiteToMove And optBlack = True Then
     optWhite = True
     SendToEngine "white"
    ElseIf Not bWhiteToMove And optWhite = True Then
     optBlack = True
     SendToEngine "black"
    End If
```

```
    If optWhite Then bCompIsWhite = True Else bCompIsWhite = False

    DoEvents
    cmdThink.Caption = Translate("Thinking") & "..."
    cmdThink.Enabled = False
    cmdStop.Visible = True
    DoEvents

    SendToEngine "go"

    If optWhite Then bCompIsWhite = True Else bCompIsWhite = False

    '--- Start chess engine ----------------------
    StartEngine
    '--- End thinking

    '--- Human to move
    cmdThink.Caption = Translate("Think") & " !"
    cmdThink.Enabled = True
    cmdStop.Visible = False

    bThinking = False
    ShowBoard
    ShowLastMoveAtBoard
    ShowMoveList
    Me.Show
  End Sub



  Private Sub cmdFakeInput_Click()
     '--- parse command input
      FakeInput = cboFakeInput.Text & vbLf
      FakeInputState = True
      cboFakeInput.SelStart = 0
      cboFakeInput.SelLength = Len(cboFakeInput.Text)
      cboFakeInput.SetFocus
      SetupBoardMode = False

      If InStr(FakeInput, "setboard") > 0 Then
        InitGame
        txtMoveList = ""
        Erase arGameMoves()
        GameMovesCnt = 0
        Result = NO_MATE
      End If

      ParseCommand FakeInput
      ShowBoard

      If bWhiteToMove Then
        optWhite.Value = True
      Else
        optBlack.Value = True
      End If
      ShowColToMove
      psLastFieldClick = "": plFieldFrom = 0: plFieldTarget = 0
  End Sub

  Public Sub ShowBoard()
    Dim x As Long, y As Long, Pos As Long, piece As Long

    For x = 1 To 8
      For y = 1 To 8
        Pos = x + (y - 1) * 8
        piece = Board(SQ_A1 + x - 1 + (y - 1) * 10)
        If piece = NO_PIECE Then
          Set oField(Pos).Picture = Nothing
        ElseIf piece >= 1 And piece <= 12 Then
          Set oField(Pos).Picture = oPiecePics(piece).Picture
        End If
      Next
    Next
    ResetGUIFieldColors

    ' Show piece counts for white; call Eval to get counts
    InitEval
    x = Eval()
    oPieceCnt(PieceDisplayOrder(WPAWN) + 1).Caption = CStr(PieceCnt(WPAWN) - PieceCnt(BPAWN))
    oPieceCnt(PieceDisplayOrder(WKNIGHT) + 1).Caption = CStr(PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT))
    oPieceCnt(PieceDisplayOrder(WBISHOP) + 1).Caption = CStr(PieceCnt(WBISHOP) - PieceCnt(BBISHOP))
    oPieceCnt(PieceDisplayOrder(WROOK) + 1).Caption = CStr(PieceCnt(WROOK) - PieceCnt(BROOK))
    oPieceCnt(PieceDisplayOrder(WQUEEN) + 1).Caption = CStr(PieceCnt(WQUEEN) - PieceCnt(BQUEEN))

    ' instead of king count show total sum
    oPieceCnt(PieceDisplayOrder(WKING) + 1).Caption = CStr(PieceCnt(WPAWN) - PieceCnt(BPAWN) + (PieceCnt(WKNIGHT) - PieceCnt(BKNIGHT)) * 3 + _
                                                      (PieceCnt(WBISHOP) - PieceCnt(BBISHOP)) * 3 + (PieceCnt(WROOK) - PieceCnt(BROOK)) * 5 + (PieceCnt(WQUEEN) - PieceCnt(BQUEEN)) * 9)

    Me.Repaint
    ShowColToMove
  End Sub

  Private Sub CreateBoard()
   '--- Create Square Images and Labels
   Dim lFieldWidth As Long, lFrameWidth As Long
   Dim x As Long, y As Long, i As Long, bBackColorIsWhite As Boolean

   bBackColorIsWhite = False
   lFieldWidth = Me.fraBoard.Width \ 9 ' 8 + 1xFrame
   lFrameWidth = lFieldWidth / 2

   For y = 1 To 8
    '--- Label board with A - H
    Set oLabelsX(y) = Me.fraBoard.Controls.Add("Forms.Label.1", "LabelX")
    With oLabelsX(y)
      .Width = lFieldWidth: .Height = lFrameWidth: .FontSize = 12: .TextAlign = 2: .Font.Bold = True
      .Left = lFrameWidth + (y - 1) * lFieldWidth: .Top = 8 * lFieldWidth + lFrameWidth
      .BackStyle = 0: .ForeColor = WhiteSqCol: .Caption = Chr$(Asc("A") - 1 + y): .BackColor = WhiteSqCol
    End With

    Set oLabelsX2(y) = Me.fraBoard.Controls.Add("Forms.Label.1", "LabelX2")
    With oLabelsX2(y)
      .Width = lFieldWidth: .Height = lFrameWidth: .FontSize = 12: .TextAlign = 2: .Font.Bold = True
      .Left = lFrameWidth + (y - 1) * lFieldWidth: .Top = 2 '1 * lFieldWidth
      .BackStyle = 0: .ForeColor = WhiteSqCol: .Caption = Chr$(Asc("A") - 1 + y): .BackColor = WhiteSqCol
    End With


    '--- Label board with 1 - 8
    Set oLabelsY(y) = Me.fraBoard.Controls.Add("Forms.Label.1", "LabelY")
    With oLabelsY(y)
      .Width = lFrameWidth: .Height = lFieldWidth: .FontSize = 12: .TextAlign = 2: .Font.Bold = True
```

```vba
      .Left = 0: .Top = (8 - y) * lFieldWidth + lFrameWidth + lFieldWidth \ 3
      .BackStyle = 0: .ForeColor = WhiteSqCol: .Caption = CStr(y): .BackColor =
      WhiteSqCol
    End With

    Set oLabelsY2(y) = Me.fraBoard.Controls.Add("Forms.Label.1", "LabelY2")
    With oLabelsY2(y)
      .Width = lFrameWidth: .Height = lFieldWidth: .FontSize = 12: .TextAlign = 2: .
      Font.Bold = True
      .Left = lFrameWidth + (9 - 1) * lFieldWidth: .Top = (8 - y) * lFieldWidth +
      lFrameWidth + lFieldWidth \ 3
      .BackStyle = 0: .ForeColor = WhiteSqCol: .Caption = CStr(y): .BackColor =
      WhiteSqCol
    End With


    '--- set square images
    For x = 1 To 8
      i = x + (y - 1) * 8
      Set oField(i) = Me.fraBoard.Controls.Add("Forms.Image.1", "Square" & i)

      Set oFieldEvents(i) = New clsBoardField: oFieldEvents(i).SetBoardField oField(i) '
      To catch click events
      oFieldEvents(i).Name = "Square" & i

      With oField(i)
        .Width = lFieldWidth: .Height = lFieldWidth: .PictureSizeMode =
        fmPictureSizeModeZoom
        .Left = lFrameWidth + (x - 1) * lFieldWidth:  .Top = lFrameWidth + (8 - y) *
        lFieldWidth
        .Tag = 20 + x + (y - 1) * 10 '--- Engine field number
        If bBackColorIsWhite Then .BackColor = WhiteSqCol Else .BackColor = BlackSqCol
        bBackColorIsWhite = Not bBackColorIsWhite
      End With
    Next x
    bBackColorIsWhite = Not bBackColorIsWhite
   Next y
 End Sub

 Private Sub FlipBoard(bWhiteAtBottom As Boolean)
  '--- Create Square Images and Labels
  Dim lFieldWidth As Long, lFrameWidth As Long
  Dim x As Long, y As Long, i As Long

  lFieldWidth = Me.fraBoard.Width \ 9 '8 + 1xFrame
  lFrameWidth = lFieldWidth / 2

  For y = 1 To 8
    '--- Label board with A - H
    With oLabelsX(y)
      If bWhiteAtBottom Then
       .Left = lFrameWidth + (y - 1) * lFieldWidth
      Else
       .Left = 8 * lFieldWidth - (lFrameWidth + (y - 1) * lFieldWidth)
      End If
    End With

    With oLabelsX2(y)
      If bWhiteAtBottom Then
       .Left = lFrameWidth + (y - 1) * lFieldWidth
      Else
       .Left = 8 * lFieldWidth - (lFrameWidth + (y - 1) * lFieldWidth)
      End If
    End With

    '--- Label board with 1 - 8
    With oLabelsY(y)
      If bWhiteAtBottom Then
```

```
          .Top = (8 - y) * lFieldWidth + lFrameWidth + lFieldWidth \ 3
        Else
          .Top = (y - 1) * lFieldWidth + lFrameWidth + lFieldWidth \ 3
        End If
      End With

      With oLabelsY2(y)
        If bWhiteAtBottom Then
          .Top = (8 - y) * lFieldWidth + lFrameWidth + lFieldWidth \ 3
        Else
          .Top = (y - 1) * lFieldWidth + lFrameWidth + lFieldWidth \ 3
        End If
      End With

      '--- set square images
      For x = 1 To 8
        i = x + (y - 1) * 8
        With oField(i)
         If bWhiteAtBottom Then
           .Left = lFrameWidth + (x - 1) * lFieldWidth:  .Top = lFrameWidth + (8 - y) *
           lFieldWidth
         Else
           .Left = 8 * lFieldWidth - (lFrameWidth + (x - 1) * lFieldWidth): .Top = 8 *
           lFieldWidth - (lFrameWidth + (8 - y) * lFieldWidth)
         End If
        End With
      Next x
    Next y
  End Sub


  Private Sub LoadPiecesPics()
  Dim PicExtension As String
  Dim sFile As String
  Dim i As Long, lFieldWidth As Long

  PicExtension = "cur"

  sFile = Dir(psDocumentPath & "\WhitePawn.*") '--- Get image extension
  If Trim(sFile) <> "" Then PicExtension = Right(sFile, 3) ' "cur"

  lFieldWidth = Me.fraPieces.Width \ 6

  ' Init piece count fields
  For i = 1 To 6
    Set oPieceCnt(i) = Me.fraPieceCnt.Controls.Add("Forms.Label.1", "PieceCnt")
    With oPieceCnt(i)
      .Width = lFieldWidth: .Height = lFieldWidth \ 2: .FontSize = 10: .TextAlign = 2: .
      Font.Bold = True
      .Left = (i - 1) * (lFieldWidth - 2): .Top = 0
      .BackStyle = 0: .ForeColor = &H80000012: .Caption = "   "
    End With
  Next i

  '--- Init piece pictures
  For i = 1 To 12
     Set oPiecePics(i) = Me.Controls("Piece" & CStr(i))   ' Preloaded images

   ' Load piece images dynamical
   If False Then
    Set oPiecePics(i) = Me.fraPieces.Controls.Add("Forms.Image.1", "Pieces")

    Select Case i
    Case 1
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhitePawn." &
      PicExtension)
    Case 2
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackPawn." &
      PicExtension)
    Case 3
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhiteKnight." &
      PicExtension)
    Case 4
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackKnight." &
      PicExtension)
    Case 5
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhiteKing." &
      PicExtension)
    Case 6
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackKing." &
      PicExtension)
    Case 7
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhiteRook." &
      PicExtension)
    Case 8
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackRook." &
      PicExtension)
    Case 9
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhiteQueen." &
      PicExtension)
    Case 10
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackQueen." &
      PicExtension)
    Case 11
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\WhiteBishop." &
      PicExtension)
    Case 12
      Set oPiecePics(i).Picture = LoadPicture(psDocumentPath & "\BlackBishop." &
      PicExtension)
    End Select

    With oPiecePics(i)
      If i Mod 2 = 0 Then
        .Top = lFieldWidth: .Left = PieceDisplayOrder(i) * lFieldWidth
      Else
        .Top = 0: .Left = PieceDisplayOrder(i) * lFieldWidth
      End If
      .Width = lFieldWidth: .Height = lFieldWidth
    End With
   End If

  Next
  End Sub

  Private Function PieceDisplayOrder(piece As Long) As Integer
    Select Case piece
    Case WPAWN, BPAWN: PieceDisplayOrder = 0
    Case WKNIGHT, BKNIGHT: PieceDisplayOrder = 1
    Case WBISHOP, BBISHOP: PieceDisplayOrder = 2
    Case WROOK, BROOK: PieceDisplayOrder = 3
    Case WQUEEN, BQUEEN: PieceDisplayOrder = 4
    Case WKING, BKING: PieceDisplayOrder = 5
    Case Else: PieceDisplayOrder = 0
    End Select
  End Function

  Private Sub cmdForward_Click()
    'TODO
  End Sub

  Private Sub cmdLoadFEN_Click()
   Dim sFEN As String
   sFEN = InputBox(Translate("Enter FEN position:"), Translate("FEN position"))
   If Trim(sFEN) <> "" Then
     cboFakeInput = "setboard " & sFEN
     cmdFakeInput_Click
```

```
  End If
End Sub

Private Sub cmdNewGame_Click()
   If cmdStop.Visible = True Then Exit Sub ' Thinking
   SendToEngine "new"
   txtIO = ""
   txtMoveList = ""
   Result = NO_MATE
   ShowBoard
End Sub

Private Sub cmdSave_Click()
 Dim sFile As String
 If psGameFile = "" Then psGameFile = "Game1.pgn"
 sFile = InputBox(Translate("Enter file name to save:"), "", psGameFile)
 sFile = psDocumentPath & "\" & sFile

 ' Write Game File
 WriteGame sFile

End Sub


Private Sub cmdLoad_Click()
 Dim sFile As String
 If psGameFile = "" Then psGameFile = "Game1.pgn"
 sFile = InputBox(Translate("Enter file name to load:"), "", psGameFile)
 sFile = psDocumentPath & "\" & sFile

 If Dir(sFile) = "" Then MsgBox Translate("File not found!"): Exit Sub
 ' Write Game File
 cmdNewGame_Click

 ReadGame sFile
 ShowBoard
End Sub

Private Sub cmdStop_Click()
   If SetupBoardMode Then Exit Sub
   bTimeExit = True
   SendCommand "---" & Translate("Stopped") & "!---"
End Sub


Private Sub SetTimeControl()
 Dim lMin1 As Integer, lSec1 As Integer, lSec2 As Integer, lDepth As Long, sLevel As
 String

 'SendToEngine "sd 2" :Exit Sub  ' Test with fixed depth
 If optSecondsPerMove.Value = True Then
   lSec1 = CLng("0" & cboSecondsPerMove.Value): If lSec1 < 1 Then lSec1 = 2 '- max
   Seconds per Move
   SendToEngine "st " & CStr(lSec1)
 ElseIf optMinutesPerGame.Value = True Then
   lMin1 = CLng("0" & cboMinutesPerGame.Value): If lMin1 < 1 Then lMin1 = 2
   SendToEngine "level 0 " & CStr(lMin1) & " 0" '- max Minutes per Game
 ElseIf optFixedDepth.Value = True Then
   lDepth = CLng("0" & cboFixedDepth.Value): If lDepth < 1 Then lDepth = 5
   SendToEngine "sd 0 " & CStr(lDepth) ' Fixed depth
 ElseIf optBlitz.Value = True Then
   lMin1 = CLng("0" & cboBlitzMin1.Value): If lMin1 < 0 Then lMin1 = 0 '- Minutes per Game
   sLevel = CStr(lMin1)
   lSec1 = CLng("0" & cboBlitzSec1.Value): If lSec1 < 0 Then lSec1 = 0 '- Seconds per
   Game
   If lSec1 > 0 Then sLevel = sLevel & ":" & CStr(lSec1)
   lSec2 = CLng("0" & cboBlitzSec2.Value): If lSec2 < 0 Then lSec2 = 0 '- Increment per
   move
```

```
    sLevel = sLevel & " " & CStr(lSec2)
    SendToEngine "level  " & sLevel
  End If
End Sub

Private Sub SendToEngine(isCommand As String)
  ParseCommand isCommand & vbCrLf
End Sub

Private Sub TranslateForm()
  Dim ctrl As Control, sText As String, sTextEN As String

  If LangCnt = 0 Then Exit Sub

  For Each ctrl In Me.Controls
    Select Case TypeName(ctrl)
    Case "CommandButton", "Label", "OptionButton", "CheckBox", "Frame"
      sTextEN = ctrl.Caption
      sText = Translate(sTextEN)
      If sText <> sTextEN Then ctrl.Caption = sText
    End Select
  Next ctrl
End Sub

Private Sub cmdUndo_Click()
  SendToEngine "undo"
  ShowBoard
  HintMove = EmptyMove
  ShowLastMoveAtBoard
  ShowMoveList
End Sub


Private Sub fraBoard_Click()
  ' board/square clicks are handled in class clsBoardField: ImageEvents_Click
End Sub


Private Sub InitTestSets()
'txtIO = "* STDIN HANDLE: " & hStdIn & vbTab & "STDOUT HANDLE: " & hStdOut & " *" & vbCrLf
txtIO = ""
cboFakeInput = "setboard 1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - 0 1 ;e3e8 Mate in 8"
'Aggiungiamo alcuni comandi di debug
cboFakeInput.AddItem "setboard 1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - 0 1 ;e3e8 Mate in 8"
cboFakeInput.AddItem "setboard r4rk1/pbq2pp1/1ppbpn1p/8/2PP4/1P1Q1N2/PBB2PPP/R3R1K1 w - - 0 1; WAC249 c4c4,d4d5 "
cboFakeInput.AddItem "eval"  ' Show evaluation of position in thinking window and writes in Trace file
cboFakeInput.AddItem "bench 3"
'cboFakeInput.AddItem "bench 5"
'cboFakeInput.AddItem "debug1 "
'cboFakeInput.AddItem "setboard r1b2rk1/pp1nq1p1/2p1p2p/3p1p2/2PPn3/2NBPN2/PPQ2PPP/2R2RK1 b - -"
'cboFakeInput.AddItem "setboard 2br2k1/ppp2p1p/4p1p1/4P2q/2P1Bn2/2Q5/PP3P1P/4R1RK b - -"
'cboFakeInput.AddItem "setboard 8/8/R3k3/1R6/8/8/8/2K5 b - -"
'cboFakeInput.AddItem "setboard 2k4r/1pr1n3/p1p1q2p/5pp1/3P1P2/P1P1P3/1R2Q1PP/1RB3K1 w KQkq -"
'cboFakeInput.AddItem "setboard 6k1/1b1nqpbp/pp4p1/5P2/1PN5/4Q3/P5PP/1B2B1K1 b - -"
'cboFakeInput.AddItem "display"
'cboFakeInput.AddItem "xboard" & vbLf & "new" & vbLf & "random" & vbLf & "level 40 5 0" & vbLf & "post"
'cboFakeInput.AddItem "xboard" & vbLf & "new" & vbLf & "random" & vbLf & "sd 4" & vbLf & "post"
'cboFakeInput.AddItem "time 30000" & vbLf & "otim 30000" & vbLf & "e2e4"
'cboFakeInput.AddItem "force" & vbLf & "quit"


'cboFakeInput.AddItem "setboard rnbqkbnr/ppp2ppp/4p3/3pP3/3P4/8/PPP2PPP/RNBQKBNR b KQkq -"
'cboFakeInput.AddItem "setboard 8/p1b1k1p1/Pp4p1/1Pp2pPp/2P2P1P/3B1K2/8/8 w - -"
'cboFakeInput.AddItem "setboard 8/2R5/1r3kp1/2p4p/2P2P2/p3K1P1/P6P/8 w - -"
'cboFakeInput.AddItem "setboard 7k/p7/6K1/5Q2/8/8/8/8 w - -"
```

```vb

'cboFakeInput.AddItem "writeepd"
'cboFakeInput.AddItem "display"
'cboFakeInput.AddItem "debug1"
End Sub

Public Sub InitTimes()
 Dim i As Integer
 With cboSecondsPerMove
    .AddItem "1": .AddItem "2": .AddItem "3": .AddItem "5": .AddItem "8": .AddItem
    "10": .AddItem "15": .AddItem "20": .AddItem "30": .AddItem "60"
 End With

 With cboMinutesPerGame
    .AddItem "1": .AddItem "2": .AddItem "3": .AddItem "5": .AddItem "8": .AddItem
    "10": .AddItem "15": .AddItem "20": .AddItem "30": .AddItem "60"
 End With

 With cboFixedDepth
   For i = 1 To 15
    .AddItem CStr(i)
   Next
 End With

 With cboBlitzMin1
    .AddItem "0": .AddItem "1": .AddItem "2": .AddItem "3": .AddItem "5": .AddItem "8"
    : .AddItem "10": .AddItem "15": .AddItem "20": .AddItem "30": .AddItem "60"
 End With

 With cboBlitzSec1
    .AddItem "0": .AddItem "15": .AddItem "30": .AddItem "30": .AddItem "45"
 End With

 With cboBlitzSec2
    .AddItem "0": .AddItem "1": .AddItem "2": .AddItem "3": .AddItem "5": .AddItem "8"
    : .AddItem "10": .AddItem "15": .AddItem "20": .AddItem "30": .AddItem "60"
 End With

End Sub

Public Sub ReadColors()
  WhiteSqCol = Val(ReadINISetting("WHITE_SQ_COLOR", "&HC0FFFF"))
  BlackSqCol = Val(ReadINISetting("BLACK_SQ_COLOR", "&H80FF&"))
  BoardFrameCol = Val(ReadINISetting("BOARD_FRAME_COLOR", "&H000040C0&"))
  fraBoard.BackColor = BoardFrameCol
End Sub


Public Sub ShowMoveList()
Dim i As Integer

txtMoveList = ""
If GameMovesCnt = 0 Then Exit Sub
If arGameMoves(1).piece Mod 2 = 0 Then txtMoveList = "       "
For i = 1 To GameMovesCnt
  If Len(txtMoveList) > 32000 Then txtMoveList = ""

  If arGameMoves(i).piece Mod 2 = 1 Then
    If arGameMoves(i).From > 0 Or arGameMoves(i + 1).From > 0 Then
      txtMoveList = txtMoveList & Left(MoveText(arGameMoves(i)) & Space(6), 6)
    End If
  Else
    If arGameMoves(i).From > 0 Then txtMoveList = txtMoveList & " - " & MoveText(
    arGameMoves(i)) & vbCrLf
  End If
Next i
  txtMoveList.SetFocus: txtMoveList.SelStart = Len(txtMoveList): txtMoveList.SelLength
   = 0
  DoEvents
End Sub

Private Sub Piece1_Click()
  SelectPiece 1
End Sub
Private Sub Piece2_Click()
  SelectPiece 2
End Sub
Private Sub Piece3_Click()
  SelectPiece 3
End Sub
Private Sub Piece4_Click()
  SelectPiece 4
End Sub
Private Sub Piece5_Click()
  SelectPiece 5
End Sub
Private Sub Piece6_Click()
  SelectPiece 6
End Sub
Private Sub Piece7_Click()
  SelectPiece 7
End Sub
Private Sub Piece8_Click()
  SelectPiece 8
End Sub
Private Sub Piece9_Click()
  SelectPiece 9
End Sub
Private Sub Piece10_Click()
  SelectPiece 10
End Sub
Private Sub Piece11_Click()
  SelectPiece 11
End Sub
Private Sub Piece12_Click()
  SelectPiece 12
End Sub
Attribute VB_Name = "basHash"
'==================================================
'= basHash:
'= Hash functions for transposition table
'==================================================
Option Explicit

Public Const MAX_THREADS       As Long = 64
Public Const MAX_HASHSIZE_MB   As Long = 1400   ' limit by 32 bit around 1500Mb / also long datatype
overflow for bytes if more / limit in VB6-Development 300MB
'The style of the hash table rows
Public Const TT_NO_BOUND       As Byte = 0
Public Const TT_UPPER_BOUND    As Byte = 1
Public Const TT_LOWER_BOUND    As Byte = 2
Public Const TT_EXACT          As Byte = 3 ' = TT_UPPER_BOUND or TT_LOWER_BOUND !
Public Const HASH_CLUSTER      As Long = 4
Public Const TT_TB_BASE_DEPTH  As Long = 222
Public Const MATERIAL_HASHSIZE As Long = 8192

Public Const HASH_SIZE_FACTOR  As Long = 34000   ' entries per MB hash

Public Type THashKey
  ' 2x 32 bit
  HashKey1 As Long
  Hashkey2 As Long
End Type

Public ZobristHash1      As Long ' for calculation of hash key
Public ZobristHash2      As Long
```

```vb

Public HashWhiteToMove  As Long ' hashkey to add for  white to move
Public HashWhiteToMove2 As Long

Public HashWCanCastle   As Long
Public HashWCanCastle2  As Long

Public HashBCanCastle   As Long
Public HashBCanCastle2  As Long

Public HashExcluded     As Long
Public InHashCnt        As Long
Public HashAccessCnt    As Long
Public HashUsage        As Long
Private bHashUsed       As Boolean
Public bHashVerify      As Boolean
Public HashGeneration   As Long
Public EmptyHash        As THashKey

Private Type HashTableEntry
  Position1 As Long ' 2x32 bit position hash key
  Position2 As Long
  Depth As Integer ' not Byte, negative values possible for QSearch
  Generation As Byte
  IsChecking As Boolean
  MoveFrom As Byte
  MoveTarget As Byte
  MovePromoted As Byte
  EvalType As Byte
  Eval As Long
  StaticEval As Long
  PvHit As Boolean
  ThreadNum As Byte ' used for thread hit cnt => for testing only
End Type

Private moHashMap                               As clsHashMap
Public HashSizeMB                               As Long
Public HashSizeMax                               As Long
Public HashSize                                 As Long ' in bytes
Public bHashSizeIgnoreGUI                       As Boolean ' HASHSIZE_IGNORE_GUI
Dim ZobristTable(SQ_A1 To SQ_H8, 0 To 16)       As Long ' key for each piece type and board
position
Dim ZobristTable2(SQ_A1 To SQ_H8, 0 To 16)      As Long
'Dim FiftyZobristTable(0 To 100)            As Long ' fifty move draw: make different hash when fifty increases> not
better
'Dim FiftyZobristTable2(0 To 100)            As Long
Dim MatZobristTable(0 To 10, 0 To 12)           As Long
'The main array to hold the hash table
Private HashTable()                             As HashTableEntry
Private HashCluster(0 To HASH_CLUSTER - 1)      As HashTableEntry
' Pointer to multi-Thread map data
Public NoOfThreads                              As Long
Public ThreadNum                                As Long   ' 0 = Main Thread
Public MainThreadStatus                         As Long, LastThreadStatus  As Long ' 1 =
start, 0 = stop, -1 = Exit
Public ThreadCommand                            As String

Public HashMapEnd                               As Long
Public HashMapHashSizePtr                       As Long
Public HashMapThreadStatusPtr(MAX_THREADS - 1)  As Long
Public HashMapBestPVPtr(MAX_THREADS - 1)        As Long ' Best pv for 10 moves
Public HashMapBoardPtr                          As Long
Public HashMapMovedPtr                          As Long
Public HashMapWhiteToMovePtr                    As Long
Public HashMapGameMovesCntPtr                   As Long
Public HashMapGameMovesPtr                      As Long
Public HashMapGamePosHashPtr                    As Long
Public HashMapSearchPtr                         As Long

Public HashRecLen                               As Long
Public HashClusterLen                           As Long
Private BestPV(10)                              As TMOVE
Public SingleThreadStatus(MAX_THREADS - 1)      As Long ' 1 = start, 0 = stop, -1 = Stopped
Private HashMapFile As String
Public bTraceHashCollision                      As Boolean

Public HashFoundFromOtherThread As Long
Private Type TMaterialHashEntry
  Hashkey As Long
  Score As Long
End Type

Public MaterialHash(MATERIAL_HASHSIZE) As TMaterialHashEntry

Public Sub InitHash()
  'Initialize the hash-table
  ' Use maximum hash size form INI file and memory command
  Dim NewHashSize As Long
  bHashTrace = CBool(ReadINISetting("HASHTRACE", "0") <> "0")
  HashSizeMB = GetMin(MAX_HASHSIZE_MB, Val(ReadINISetting("HASHSIZE", "64"))) ' 2 GB for 32 bit ( max 1.5 GB?)
  If CBool(ReadINISetting("HASHSIZE_IGNORE_GUI", "0") = "0") Then
    HashSizeMB = GetMax(HashSizeMB, MemoryMB) ' memory command value from GUI
  End If
  HashSizeMB = GetMin(MAX_HASHSIZE_MB, HashSizeMB) ' in 1 core: vb array MB, in IDE max around 350MB, EXE 1.5 GB
  If InIDE Then HashSizeMB = GetMin(128, HashSizeMB) ' Limited in IDE, depends on local memory usage

'HashSizeMB = 1400
'NoOfThreads = 2
'ThreadNum = 0 ' TEST

lblHashSize:
  If bHashTrace Then WriteTrace "Init hash size start " & HashSizeMB & "MB " & Now()
  If ThreadNum <= 0 Then  ' for helper threads if hash size was changed
   If Not pbMSExcelRunning Then
     WriteINISetting "HASH_USED", CStr(HashSizeMB)
   End If
  Else
     HashSizeMB = Val(ReadINISetting("HASH_USED", "64")) ' read from main thread
  End If
  HashSize = HashSizeMB * HASH_SIZE_FACTOR    ' in Bytes, seems to fit...? hash len = 31
  HashUsage = 0
  bHashUsed = False
  #If VBA_MODE = 0 Then ' Find unique file name if more than one version is CB are running
    HashMapFile = ReadINISetting("HASH_MAP_FILE", "CBVBHash" & Trim(App.Major) & Trim(
    App.Minor) & Trim(App.Revision) & "_" & GetAppTimeString() & ".DAT")  ' Change in INI to run 2x CB engine
  #End If

  bHashVerify = CBool(ReadINISetting("HASH_VERIFY", "0") <> "0") ' verify hash read/write to avoid collisions for many cores
  If NoOfThreads < 2 Then bHashVerify = False
  bTraceHashCollision = bHashVerify And CBool(ReadINISetting("HASH_COLL_TRACE", "0")
  <> "0") ' trace hash read/write collisions for > 1 core
  HashRecLen = LenB(HashCluster(0)): HashClusterLen = HashRecLen * HASH_CLUSTER

  If bHashTrace Then WriteTrace "InitHash: HashSize:" & HashSize & ", Threads:" &
  NoOfThreads
  If NoOfThreads <= 1 Then
    If bHashTrace Then WriteTrace "InitHash: Redim HashTable(0) done " & Now()
    If HashSize > HashSizeMax Then
      ReDim HashTable(HashSize + HASH_CLUSTER) ' may be OutOfMemory Error here
      If bHashTrace Then WriteTrace "InitHash: Redim done HashTable Size= " & HashSize
```

```
          & " entries " & Now()
      HashSizeMax = HashSize
    Else
      If bHashTrace Then WriteTrace "InitHash: Keep HashTable Size= " & HashSize & " entries " & Now()
      ' REDIM HashTable > creates random error:  Out of memory  / needs unfragmented memory fo rrequested size
      Dim j As Long
      For j = 1 To HashSize: HashTable(j).Position1 = 0: Next
   End If
    'MsgBox "Hashtable " & NoOfThreads & "/ " & ThreadNum
  ElseIf NoOfThreads > 1 Then
    ' Structure for game data
    ' ThreadStatus as long ' 1 = start, 0 = stop, -1 = Exit
    ReDim HashTable(0)  ' internal hash not needed
    If bHashTrace Then WriteTrace "InitHash: Init hash map " & HashSize & " Bytes " & Now()

    ' HashMapEnd value starts a 0, every part of memory added will increase the value to address the next one
    HashMapEnd = 0
    If bHashTrace Then WriteTrace "HashMap: " & NoOfThreads & "/ " & ThreadNum & ", HashMapEnd:" & HashMapEnd & " MB:" & HashSizeMB
    Dim i As Long

    For i = 0 To MAX_THREADS - 1
      HashMapThreadStatusPtr(i) = HashMapEnd: HashMapEnd = HashMapEnd + LenB(MainThreadStatus)
      'If bHashTrace Then WriteTrace "InitHash:HashMapThreadStatusPtr:" & i & ":" & HashMapThreadStatusPtr(i)
    Next

    For i = 0 To MAX_THREADS - 1
      HashMapBestPVPtr(i) = HashMapEnd: HashMapEnd = HashMapEnd + LenB(PV(0, 0)) * 10
    Next

    HashMapBoardPtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(Board(0)) * MAX_BOARD
    HashMapMovedPtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(Moved(0)) * MAX_BOARD
    HashMapWhiteToMovePtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(bWhiteToMove)
    HashMapGameMovesCntPtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(GameMovesCnt)
    HashMapGameMovesPtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(arGameMoves(0)) * MAX_GAME_MOVES
    HashMapGamePosHashPtr = HashMapEnd: HashMapEnd = HashMapEnd + LenB(GamePosHash(0)) * MAX_GAME_MOVES

    ' the real hash for search is allocated now:
    HashMapSearchPtr = HashMapEnd
    HashMapEnd = HashMapEnd + HashRecLen * (HashSize + HASH_CLUSTER)
    ' allocate hash map file for multiple threads
    If ThreadNum >= 0 Then
      If bHashTrace Then WriteTrace "InitHash:OpenHashMap: HashMapEnd " & HashMapEnd
      NewHashSize = HashMapEnd
      OpenHashMap NewHashSize
      If NewHashSize <> HashMapEnd Then
        HashSizeMB = NewHashSize \ 1024# \ 1024# ' use reduced hash size
        WriteTrace "InitHash: New HashSize: " & HashSizeMB & " / " & Now()
        GoTo lblHashSize
      End If
    End If
  End If
  If bHashTrace Then WriteTrace "Init hash size done " & HashSize & " entries " & Now()
End Sub

Public Sub HashBoard(HashKeyOut As THashKey, ExcludedMove As TMOVE)
  Dim i As Long, sq As Long
  ZobristHash1 = 0: ZobristHash2 = 0

  For i = 1 To NumPieces
    sq = Pieces(i): If sq <> 0 Then ZobristHash1 = ZobristHash1 Xor ZobristTable(sq,
    Board(sq)): ZobristHash2 = ZobristHash2 Xor ZobristTable2(sq, Board(sq))
  Next

  If EpPosArr(Ply) > 0 Then HashSetPiece EpPosArr(Ply), Board(EpPosArr(Ply))
  If bWhiteToMove Then
    ZobristHash1 = ZobristHash1 Xor HashWhiteToMove: ZobristHash2 = ZobristHash2 Xor
    HashWhiteToMove2
  End If
  If Moved(WKING_START) = 0 Then ' white can castle?
    If Moved(SQ_H1) = 0 Then ZobristHash1 = ZobristHash1 Xor HashWCanCastle
    If Moved(SQ_A1) = 0 Then ZobristHash2 = ZobristHash2 Xor HashWCanCastle2
  End If
  If Moved(BKING_START) = 0 Then ' black can castle?
    If Moved(SQ_H8) = 0 Then ZobristHash1 = ZobristHash1 Xor HashBCanCastle
    If Moved(SQ_A8) = 0 Then ZobristHash2 = ZobristHash2 Xor HashBCanCastle2
  End If
  If ExcludedMove.From > 0 Then ' use from/target sq to be different to normal position
    ZobristHash1 = ZobristHash1 Xor ZobristTable(ExcludedMove.From, ExcludedMove.Piece
    ): ZobristHash2 = ZobristHash2 Xor ZobristTable2(ExcludedMove.Target,
    ExcludedMove.Piece)
  End If

  HashKeyOut.HashKey1 = ZobristHash1: HashKeyOut.Hashkey2 = ZobristHash2
End Sub

Public Function HashGetKey() As THashKey
  HashGetKey.HashKey1 = ZobristHash1
  HashGetKey.Hashkey2 = ZobristHash2
End Function

Public Sub NextHashGeneration()
  HashGeneration = GetMin(255, GameMovesCnt \ 2 + 1)
End Sub

Public Sub HashSetKey(ByRef Hashkey As THashKey)
  ZobristHash1 = Hashkey.HashKey1
  ZobristHash2 = Hashkey.Hashkey2
End Sub


Public Function HashTableSave(Hashkey As THashKey, _
                              Depth As Long, _
                              HashMove As TMOVE, _
                              EvalType As Long, _
                              EvalScore As Long, _
                              StaticEval As Long, _
                              PvHit As Boolean)
  'Dim FiftyHash As THashKey
  'If Fifty >= 4 Then ' fifty move draw: make different hash when fifty increases every 8 moves > prolbem with 3x
  draw detection using hash
  '  FiftyHash.HashKey1 = Hashkey.HashKey1 Xor FiftyZobristTable(Fifty \ 8): FiftyHash.Hashkey2 =
  Hashkey.Hashkey2 Xor FiftyZobristTable2(Fifty \ 8)
  'Else
  '  FiftyHash.HashKey1 = Hashkey.HashKey1: FiftyHash.Hashkey2 = Hashkey.Hashkey2
  'End If

  If ThreadNum < 0 Then ' single core using internal VB array
    InsertIntoHashTable Hashkey, Depth, HashMove, EvalType, EvalScore, StaticEval,
    PvHit
  Else ' multiple cores using global hash map
    InsertIntoHashMap Hashkey, Depth, HashMove, EvalType, EvalScore, StaticEval, PvHit
  End If
End Function



Public Function InsertIntoHashTable(Hashkey As THashKey, _
                                    ByVal Depth As Long, _
```

```
                                    HashMove As TMOVE, _
                                    ByVal EvalType As Long, _
                                    ByVal EvalScore As Long, _
                                    ByVal StaticEval As Long, _
                                    ByVal PvHit As Boolean)
  '--- Insert hash entry into hash array if only one thread (faster than access to global mapped memory)
  Dim ClusterIndex As Long, NewHashMove As TMOVE, i As Long, ReplaceIndex As Long, MaxReplaceValue As Long, ReplaceValue As Long, bPosFound As Boolean
  Debug.Assert HashMove.From = 0 Or (HashMove.Piece <> NO_PIECE And Board(HashMove.From) <> NO_PIECE)
  If bTimeExit Then Exit Function ' score not exact
  SetMove NewHashMove, HashMove  ' Don't overwrite move of caller function
  bHashUsed = True: bPosFound = False
  MaxReplaceValue = 9999
  '--- Compute hash key
  ZobristHash1 = Hashkey.HashKey1: ZobristHash2 = Hashkey.Hashkey2
  ClusterIndex = HashKeyCompute() * HASH_CLUSTER
  ReplaceIndex = ClusterIndex
  If HashAccessCnt < 2100000000 Then HashAccessCnt = HashAccessCnt + 1

  For i = 0 To HASH_CLUSTER - 1
    With HashTable(ClusterIndex + i)
      If .Position1 = 0 Then ReplaceIndex = ClusterIndex + i: Exit For ' use empty entry
      If HashGeneration = .Generation Then If HashUsage < 2100000000 Then HashUsage = HashUsage + 1
      ' Don't overwrite more valuable entry
      If (.Position1 = ZobristHash1 And .Position2 = ZobristHash2) Then
        ' Position found: Preserve hash move if no new move
        If .MoveFrom > 0 And NewHashMove.From = 0 Then ' old hash move exists
          NewHashMove.From = .MoveFrom: NewHashMove.Target = .MoveTarget: NewHashMove.Promoted = .MovePromoted: NewHashMove.IsChecking = .IsChecking
        End If
        ReplaceIndex = ClusterIndex + i: bPosFound = True
        Exit For
      Else
        ' Other position found. Overwrite?
        ReplaceValue = .Depth - 8 * (HashGeneration - .Generation)
        If ReplaceValue < MaxReplaceValue Then
          MaxReplaceValue = ReplaceValue: ReplaceIndex = ClusterIndex + i
        End If
      End If
    End With
  Next


  With HashTable(ReplaceIndex)
    '--- Save hash data, preserve hash move if no new move
    If Not bPosFound Or EvalType = TT_EXACT Or Depth > .Depth - 4 Then
      .Position1 = ZobristHash1: .Position2 = ZobristHash2
      .MoveFrom = NewHashMove.From: .MoveTarget = NewHashMove.Target: .MovePromoted = NewHashMove.Promoted
      .EvalType = EvalType: .Eval = ScoreToHash(EvalScore)
      .StaticEval = StaticEval: .Depth = Depth
      .Generation = HashGeneration
      .IsChecking = NewHashMove.IsChecking
      .PvHit = PvHit
      Debug.Assert .MoveFrom = 0 Or Board(.MoveFrom) <> NO_PIECE
    End If
  End With

End Function

Public Function HashTableRead(Hashkey As THashKey, _
                              ByRef HashDepth As Long, _
                              HashMove As TMOVE, _
                              ByRef EvalType As Long, _
                              ByRef EvalScore As Long, _
                              ByRef StaticEval As Long, _
```

```
                                ByRef PvHit As Boolean, ByRef HashThreadNum As Long) As
                                Boolean
 ' Dim FiftyHash As THashKey
 ' If Fifty >= 4 Then ' fifty move draw: make different hash when fifty increases every 8 moves > prolbem with 3x draw detection using hash
 '   FiftyHash.HashKey1 = Hashkey.HashKey1 Xor FiftyZobristTable(Fifty \ 8): FiftyHash.Hashkey2 = Hashkey.Hashkey2 Xor FiftyZobristTable2(Fifty \ 8)
 ' Else
 '   FiftyHash.HashKey1 = Hashkey.HashKey1: FiftyHash.Hashkey2 = Hashkey.Hashkey2
 ' End If

  If ThreadNum < 0 Then ' single core using internal VB array
    HashTableRead = IsInHashTable(Hashkey, HashDepth, HashMove, EvalType, EvalScore,
    StaticEval, PvHit)
    HashThreadNum = -1
  Else ' multiple cores using global hash map
    HashTableRead = IsInHashMap(Hashkey, HashDepth, HashMove, EvalType, EvalScore,
    StaticEval, PvHit, HashThreadNum)
  End If
End Function

Public Function IsInHashTable(Hashkey As THashKey, _
                              ByRef HashDepth As Long, _
                              HashMove As TMOVE, _
                              ByRef EvalType As Long, _
                              ByRef EvalScore As Long, _
                              ByRef StaticEval As Long, _
                              ByRef PvHit As Boolean) As Boolean
  '--- Search for hash entry into hash array if one thread
  Dim IndexKey As Long, i As Long

  IsInHashTable = False: ClearMove HashMove: EvalType = TT_NO_BOUND: EvalScore =
  VALUE_NONE: StaticEval = VALUE_NONE: HashDepth = -MAX_GAME_MOVES
  ZobristHash1 = Hashkey.HashKey1: ZobristHash2 = Hashkey.Hashkey2
  IndexKey = HashKeyCompute() * HASH_CLUSTER

  For i = 0 To HASH_CLUSTER - 1
    If HashTable(IndexKey + i).Position1 = 0 Then If ZobristHash1 <> 0 Then Exit
    Function '--- empty entry, not found
      With HashTable(IndexKey + i)
        If ZobristHash1 = .Position1 And ZobristHash2 = .Position2 Then
          If .Depth > HashDepth Then
            ' entry found
            IsInHashTable = True: PvHit = False
            If InHashCnt < 2000000 Then InHashCnt = InHashCnt + 1
            '--- Read hash data
            If .MoveFrom > 0 Then
              HashMove.From = .MoveFrom: HashMove.Target = .MoveTarget:
              HashMove.IsChecking = .IsChecking
              If Board(.MoveTarget) <= NO_PIECE Then HashMove.Captured = Board(.
              MoveTarget)
              HashMove.Piece = Board(.MoveFrom): HashMove.CapturedNumber = Squares(.
              MoveTarget)
              HashMove.Promoted = .MovePromoted: If HashMove.Promoted <> 0 Then
              HashMove.Piece = HashMove.Promoted
              Debug.Assert HashMove.Piece <> NO_PIECE
              HashMove.IsLegal = True

              'If Not MovePossible(HashMove) Then Stop
              Select Case HashMove.Piece
                Case WPAWN
                  If .MoveTarget - .MoveFrom = 20 Then
                    HashMove.EnPassant = ENPASSANT_WMOVE
                  ElseIf Board(.MoveTarget) = BEP_PIECE Then
                    HashMove.EnPassant = ENPASSANT_CAPTURE
                    HashMove.Captured = BEP_PIECE
                  End If
                Case BPAWN
```

```
                  If .MoveFrom - .MoveTarget = 20 Then
                    HashMove.EnPassant = ENPASSANT_BMOVE
                  ElseIf Board(.MoveTarget) = WEP_PIECE Then
                    HashMove.EnPassant = ENPASSANT_CAPTURE
                    HashMove.Captured = WEP_PIECE
                  End If
                Case WKING
                  If .MoveFrom = SQ_E1 Then
                    If .MoveTarget = SQ_G1 Then
                      HashMove.Castle = WHITEOO
                    ElseIf .MoveTarget = SQ_C1 Then
                      HashMove.Castle = WHITEOOO
                    End If
                  End If
                Case BKING
                  If .MoveFrom = SQ_E8 Then
                    If .MoveTarget = SQ_G8 Then
                      HashMove.Castle = BLACKOO
                    ElseIf .MoveTarget = SQ_C8 Then
                      HashMove.Castle = BLACKOOO
                    End If
                  End If
              End Select
            End If
            EvalType = .EvalType: EvalScore = HashToScore(.Eval): StaticEval = .StaticEval
            HashDepth = .Depth
            PvHit = .PvHit
            .Generation = HashGeneration ' Update generation> still valid in this game
            Exit For
          End If
        End If
      End With
  Next

End Function

Public Function LimitDouble(ByVal d As Double) As Long
  ' Prevent overflow by looping off anything beyond 31 bits
  Const MaxNumber As Double = 2 ^ 31
  LimitDouble = CLng(d - (Fix(d / MaxNumber) * MaxNumber))
End Function

Public Sub InitZobrist()
  ' init values for hash calculation. 2x32 bit for 64 bit key
  Static bDone As Boolean
  Dim p         As Long, s As Long
  If bDone Then Exit Sub
  bDone = True
  ZobristHash1 = 0: ZobristHash2 = 0
  Randomize 1001 ' init random generator with fix value

  ' create hash value for each piece type and each board position
  For s = SQ_A1 To SQ_H8
    For p = 0 To 16
      ZobristTable(s, p) = CalcUniqueKey(): ZobristTable2(s, p) = CalcUniqueKey()
    Next
  Next

  HashWhiteToMove = CalcUniqueKey(): HashWhiteToMove2 = CalcUniqueKey()
  HashWCanCastle = CalcUniqueKey(): HashWCanCastle2 = CalcUniqueKey()
  HashBCanCastle = CalcUniqueKey(): HashBCanCastle2 = CalcUniqueKey()

' ' for rule: draw after fifty quiet moves , make a different hash key when fifty counter increases
' For s = 1 To 100
'   FiftyZobristTable(s) = CalcUniqueKey(): FiftyZobristTable2(s) = CalcUniqueKey()
' Next

```

```
    ' keys for material values total
    For s = 0 To 10 ' Material hash: Piece cnt
      For p = 0 To 12 ' Piece
        MatZobristTable(s, p) = CalcUniqueKey()
      Next
    Next

  End Sub

  Public Function CalcMaterialKey() As Long
    CalcMaterialKey = MatZobristTable(PieceCnt(WQUEEN), WQUEEN) Xor MatZobristTable(
    PieceCnt(BQUEEN), BQUEEN) Xor MatZobristTable(PieceCnt(WROOK), WROOK) Xor
    MatZobristTable(PieceCnt(BROOK), BROOK) Xor MatZobristTable(PieceCnt(WBISHOP),
    WBISHOP) Xor MatZobristTable(PieceCnt(BBISHOP), BBISHOP) Xor MatZobristTable(
    PieceCnt(WKNIGHT), WKNIGHT) Xor MatZobristTable(PieceCnt(BKNIGHT), BKNIGHT) Xor
    MatZobristTable(PieceCnt(WPAWN), WPAWN) Xor MatZobristTable(PieceCnt(BPAWN), BPAWN)
  End Function

  Private Function CalcUniqueKey() As Long
    Static KeyList((SQ_H8 + 1) * 17 * 2 + 8) As Long
    Static ListCnt                           As Long
    Dim l                                    As Long, i As Long
  NextTry:
    l = 65536 * (Int(Rnd * 65536) - 32768) Or Int(Rnd * 65536)

    For i = 1 To ListCnt
      If KeyList(i) = l Then GoTo NextTry
    Next

    ListCnt = ListCnt + 1: KeyList(ListCnt) = l
    CalcUniqueKey = l
  End Function

  Public Sub HashSetPiece(ByVal Position As Long, ByVal Piece As Long)
    If Piece = FRAME Or Piece = NO_PIECE Then Exit Sub
    ZobristHash1 = ZobristHash1 Xor ZobristTable(Position, Piece)
    ZobristHash2 = ZobristHash2 Xor ZobristTable2(Position, Piece)
  End Sub

  Public Sub HashDelPiece(ByVal Position As Long, ByVal Piece As Long)
    If Piece = FRAME Or Piece = NO_PIECE Then Exit Sub
    ZobristHash1 = ZobristHash1 Xor ZobristTable(Position, Piece)
    ZobristHash2 = ZobristHash2 Xor ZobristTable2(Position, Piece)
  End Sub

  Public Sub HashMovePiece(ByVal From As Long, Target As Long, ByVal Piece As Long)
    ZobristHash1 = ZobristHash1 Xor ZobristTable(From, Piece) Xor ZobristTable(Target,
    Piece)
    ZobristHash2 = ZobristHash2 Xor ZobristTable(From, Piece) Xor ZobristTable2(Target,
    Piece)
  End Sub

  Public Function HashKeyCompute() As Long
    HashKeyCompute = ZobristHash1 Xor ZobristHash2
    If HashKeyCompute = -2147483648# Then HashKeyCompute = HashKeyCompute + 1
    HashKeyCompute = Abs(HashKeyCompute) Mod (HashSize \ HASH_CLUSTER)
  End Function

  Public Function HashKeyComputeMap() As Long
    HashKeyComputeMap = ZobristHash1 Xor ZobristHash2
    If HashKeyComputeMap = -2147483648# Then HashKeyComputeMap = HashKeyComputeMap + 1
    HashKeyComputeMap = Abs(HashKeyComputeMap) Mod (HashSize \ HASH_CLUSTER)
  End Function

  Public Sub SetHashToMove()
    If bWhiteToMove Then
      ZobristHash1 = ZobristHash1 Xor HashWhiteToMove: ZobristHash2 = ZobristHash2 Xor
      HashWhiteToMove2
```

```vb
    End If
  End Sub

  Public Sub HashSetCastle()
    If WhiteCastled = NO_CASTLE Then ZobristHash1 = ZobristHash1 Xor HashWCanCastle:
    ZobristHash2 = ZobristHash2 Xor HashWCanCastle2
    If BlackCastled = NO_CASTLE Then ZobristHash1 = ZobristHash1 Xor HashBCanCastle:
    ZobristHash2 = ZobristHash2 Xor HashBCanCastle2
  End Sub

  Public Function ScoreToHash(ByVal Score As Long) As Long
    If Score = VALUE_NONE Then
      ScoreToHash = Score
    ElseIf Score >= MATE_IN_MAX_PLY Then
      ScoreToHash = Score + Ply
    ElseIf Score <= -MATE_IN_MAX_PLY Then
      ScoreToHash = Score - Ply
    Else
      ScoreToHash = Score
    End If
  End Function

  Public Function HashToScore(ByVal Score As Long) As Long
    If Score = VALUE_NONE Then
      HashToScore = Score
    ElseIf Score >= MATE_IN_MAX_PLY Then
      HashToScore = Score - Ply
    ElseIf Score <= -MATE_IN_MAX_PLY Then
      HashToScore = Score + Ply
    Else
      HashToScore = Score
    End If
  End Function

  Public Function HashUsagePerc() As String
    If HashSize = 0 Then
      HashUsagePerc = ""
    Else
      If HashUsage > HashSize Then HashUsage = HashSize
      HashUsagePerc = Format$(CDbl(HashUsage) * 100& / HashSize, "0.0")
    End If
  End Function

  Public Function HashUsageUCI() As Long
    Dim x As Single
    If HashSize = 0 Or HashUsage <= 0 Then
      HashUsageUCI = 0
    Else
      x = HashUsage: x = x * CSng(1000) / CSng(1 + HashAccessCnt)
      HashUsageUCI = GetMin(1000, CLng(x))
    End If
  End Function

  Public Function OpenHashMap(ByRef TotalSize As Long) As Long
    '--- init global mapped memory if more then one thread, used by all threads!
    Static OldHashSize As Long
    If OldHashSize = 0 Then
      Set moHashMap = New clsHashMap
    End If
    If OldHashSize = 0 Or OldHashSize <> TotalSize Then
      If ThreadNum = 0 Then
        If OldHashSize = 0 Then
          Set moHashMap = New clsHashMap
          If bThreadTrace Then WriteTrace "OpenHashMap: New clsHashMap: " & TotalSize
        Else
          If bThreadTrace Then WriteTrace "OpenHashMap: CloseMap"
          moHashMap.CloseMap
        End If
        moHashMap.CreateMap HashMapFile, TotalSize ' TotalSize may be reduced if not enough memory !!!
        If bThreadTrace Then WriteTrace "OpenHashMap: CreateMap: Size " & TotalSize
      ElseIf ThreadNum > 0 Then
        moHashMap.OpenMap HashMapFile, TotalSize
        If bThreadTrace Then WriteTrace "OpenHashMap: OpenMap: Size " & TotalSize
      End If
      OldHashSize = TotalSize
    Else
      If ThreadNum = 0 Then moHashMap.ClearMap TotalSize
    End If
  End Function

  Public Function CloseHashMap() As Long
    moHashMap.CloseMap
  End Function

  Public Function InsertIntoHashMap(Hashkey As THashKey, _
                                    ByVal Depth As Long, _
                                    HashMove As TMOVE, _
                                    ByVal EvalType As Long, _
                                    ByVal Eval As Long, _
                                    ByVal StaticEval As Long, _
                                    ByVal PvHit As Boolean)
    '--- Insert hash entry into global mapped memory if more then one thread, used by all threads!
    Dim ClusterIndex As Long, NewHashMove As TMOVE, i As Long, ReplaceIndex As Long, MaxReplaceValue As Long, ReplaceValue As Long, bPosFound As Boolean
    Debug.Assert HashMove.From = 0 Or HashMove.Piece <> NO_PIECE
    Debug.Assert NoOfThreads > 1
    If bTimeExit Then Exit Function ' score not exact

    'If ThreadNum > 0 Then Exit Function '###################TESTc2

    SetMove NewHashMove, HashMove  ' Don't overwrite
    bHashUsed = True: bPosFound = False
    MaxReplaceValue = 9999
    '--- Compute hash key
    ZobristHash1 = Hashkey.HashKey1: ZobristHash2 = Hashkey.Hashkey2
    ClusterIndex = HashKeyComputeMap() * HASH_CLUSTER
    ReplaceIndex = 0
    moHashMap.ReadMapHashCluster ClusterIndex, VarPtr(HashCluster(0)), HashClusterLen ' read this cluster only
    If HashAccessCnt < 2100000000 Then HashAccessCnt = HashAccessCnt + 1

    For i = 0 To HASH_CLUSTER - 1
      With HashCluster(i) ' search in retrieved cluster
        If .Position1 = 0 Then ReplaceIndex = ClusterIndex + i: Exit For ' empty entry found
        If HashGeneration = .Generation Then If HashUsage < 2100000000 Then HashUsage = HashUsage + 1
        ' Don't overwrite more valuable entry
        If (.Position1 = ZobristHash1 And .Position2 = ZobristHash2) Then
          ' Position found: Preserve hash move if no new move
          If NewHashMove.From = 0 And .MoveFrom > 0 Then
            NewHashMove.From = .MoveFrom: NewHashMove.Target = .MoveTarget: NewHashMove.Promoted = .MovePromoted: NewHashMove.IsChecking = .IsChecking
          End If
          ReplaceIndex = ClusterIndex + i: bPosFound = True
          Exit For
        Else
          ' Other position found. Find least valuable entry
          ReplaceValue = .Depth - 8 * (HashGeneration - .Generation)
          If ReplaceValue < MaxReplaceValue Then
            MaxReplaceValue = ReplaceValue: ReplaceIndex = ClusterIndex + i
          End If
        End If
      End With
    Next

```

```
    With HashCluster(ReplaceIndex - ClusterIndex)
      '--- Save hash data, preserve hash move if no new move
       If Not bPosFound Or EvalType = TT_EXACT Or Depth > .Depth - 4 Then
        .Position1 = ZobristHash1: .Position2 = ZobristHash2
        .MoveFrom = NewHashMove.From: .MoveTarget = NewHashMove.Target: .MovePromoted =
        NewHashMove.Promoted
        .EvalType = EvalType: .Eval = ScoreToHash(Eval)
        .StaticEval = StaticEval: .Depth = Depth
        .Generation = HashGeneration
        .IsChecking = NewHashMove.IsChecking
        .PvHit = PvHit
        If ThreadNum >= 0 Then .ThreadNum = ThreadNum
        '--- Write Hash Map: replace index in Cluster only
        moHashMap.WriteMapHashEntry ReplaceIndex, VarPtr(HashCluster(ReplaceIndex -
        ClusterIndex))
        Debug.Assert .MoveFrom = 0 Or Board(.MoveFrom) <> NO_PIECE
      End If
    End With

  End Function

  Public Function IsInHashMap(Hashkey As THashKey, _
                              ByRef HashDepth As Long, _
                              HashMove As TMOVE, _
                              ByRef EvalType As Long, _
                              ByRef Eval As Long, _
                              ByRef StaticEval As Long, _
                              ByRef PvHit As Boolean, ByRef HashThreadNum As Long) As
                              Boolean
    '--- search for hash entry in global mapped memory if more then one thread
    Dim IndexKey As Long, i As Long
    Debug.Assert NoOfThreads > 1
    IsInHashMap = False: ClearMove HashMove: EvalType = TT_NO_BOUND: Eval = VALUE_NONE:
    StaticEval = VALUE_NONE: HashDepth = -MAX_GAME_MOVES
    ZobristHash1 = Hashkey.HashKey1: ZobristHash2 = Hashkey.Hashkey2
    IndexKey = HashKeyComputeMap() * HASH_CLUSTER
    moHashMap.ReadMapHashCluster IndexKey, VarPtr(HashCluster(0)), HashClusterLen

    For i = 0 To HASH_CLUSTER - 1

      With HashCluster(i)
        If .Position1 = 0 Then If ZobristHash1 <> 0 Then Exit Function '--- empty entry, not
        found
          If ZobristHash1 = .Position1 And ZobristHash2 = .Position2 Then
            If .Depth > HashDepth Then
              ' entry found
              IsInHashMap = True: PvHit = False
              If InHashCnt < 2000000 Then InHashCnt = InHashCnt + 1
              '--- Read hash data
              If .MoveFrom > 0 Then
                HashMove.From = .MoveFrom: HashMove.Target = .MoveTarget:
                HashMove.IsChecking = .IsChecking
                If Board(.MoveTarget) <= NO_PIECE Then HashMove.Captured = Board(.
                MoveTarget)
                HashMove.Piece = Board(.MoveFrom): HashMove.CapturedNumber = Squares(.
                MoveTarget)
                HashMove.Promoted = .MovePromoted: If HashMove.Promoted <> 0 Then
                HashMove.Piece = HashMove.Promoted
                Debug.Assert HashMove.Piece <> NO_PIECE
                HashMove.IsLegal = True

                'If Not MovePossible(HashMove) Then Stop
                Select Case HashMove.Piece
                  Case WPAWN
                    If .MoveTarget - .MoveFrom = 20 Then
                      HashMove.EnPassant = ENPASSANT_WMOVE
                    ElseIf Board(.MoveTarget) = BEP_PIECE Then
                      HashMove.EnPassant = ENPASSANT_CAPTURE
```

```vb
                    HashMove.Captured = BEP_PIECE
                  End If
                Case BPAWN
                  If .MoveFrom - .MoveTarget = 20 Then
                    HashMove.EnPassant = ENPASSANT_BMOVE
                  ElseIf Board(.MoveTarget) = WEP_PIECE Then
                    HashMove.EnPassant = ENPASSANT_CAPTURE
                    HashMove.Captured = WEP_PIECE
                  End If
                Case WKING
                  If .MoveFrom = SQ_E1 Then
                    If .MoveTarget = SQ_G1 Then
                      HashMove.Castle = WHITEOO
                    ElseIf .MoveTarget = SQ_C1 Then
                      HashMove.Castle = WHITEOOO
                    End If
                  End If
                Case BKING
                  If .MoveFrom = SQ_E8 Then
                    If .MoveTarget = SQ_G8 Then
                      HashMove.Castle = BLACKOO
                    ElseIf .MoveTarget = SQ_C8 Then
                      HashMove.Castle = BLACKOOO
                    End If
                  End If
              End Select
            End If
            ' Read values for entry
            EvalType = .EvalType: Eval = HashToScore(.Eval): StaticEval = .StaticEval
            HashDepth = .Depth
            PvHit = .PvHit
            HashThreadNum = .ThreadNum
            If .Generation <> HashGeneration Then
              .Generation = HashGeneration ' Update generation, each game move is a new generation
              '--- Write Hash Map: replace index in Cluster only
              moHashMap.WriteMapHashEntry IndexKey + i, VarPtr(HashCluster(i))
            End If
            'If ThreadNum >= 0 Then If .ThreadNum <> GetMax(0, ThreadNum) Then HashFoundFromOtherThread = HashFoundFromOtherThread + 1
            Exit For
          End If
        End If
    End With
  Next i

End Function

Public Function InitThreads()
  Static bInitDone As Boolean
  Dim i            As Long
  DoEvents
  #If VBA_MODE = 0 Then
    If Not bInitDone And NoOfThreads > 1 Then
      If CreateAppLockFile() Then ' Already started?
        If bThreadTrace Then WriteTrace "InitThreads: NoOfThreads=" & NoOfThreads
        MainThreadStatus = 0: WriteMainThreadStatus 0 ' idle
  ' Dim tStart As Single, tEnd As Single
  ' tStart = Timer
  ' Dim sCmd As String
        For i = 2 To NoOfThreads
          StartProcess App.Path & "\ChessBrainVB.exe thread" & Trim$(CStr(i - 1)) ' Much faster
        Next
          '---Shell App.Path & "\ChessBrainVB.exe thread" & Trim$(CStr(i - 1)), vbMinimizedNoFocus ' SHELL is MUCH slower ( 1 sec per call?!?)

   '  tEnd = Timer()
   '  WriteTrace "Threads started:" & ", Time:" & Format$(tEnd - tStart, "0.00000")
```

```
        Sleep 500
      End If
    End If
  #End If
  bInitDone = True
End Function

Public Function CreateAppLockFile() As Boolean
  ' for main thread: create a locked file that gets unlocked when main thread end/crashed
  ' this file is checked by the helper threads: if file is unlocked also exit helper threads
  Static lLOCK_FILEHANDLE As Long
  Sleep 200 ' wait for end of previous exe run
  #If VBA_MODE = 0 Then
    Debug.Assert NoOfThreads > 1
    lLOCK_FILEHANDLE = FreeFile()
    On Error GoTo lblLockErr
    Open App.Path & "\CB_THREAD0.TXT" For Append Access Write Lock Write As #lLOCK_FILEHANDLE
    Print #lLOCK_FILEHANDLE, "Temporary lock file. Main thread started:" & Now()
    CreateAppLockFile = True
  #End If
lblExit:
  Exit Function
lblLockErr:
  CreateAppLockFile = False
  WriteTrace "Already started? Cannot open Application lock file: CB_THREAD0.TXT " & Now()
  Resume lblExit
End Function

Public Function CheckAppLockFile() As Boolean
  ' this file is checked is used by the helper threads: returns true if file is unlocked > also exit helper threads
  Dim lLOCK_FILEHANDLE2 As Long
  On Error GoTo lblErr
  CheckAppLockFile = False
  #If VBA_MODE = 0 Then
    lLOCK_FILEHANDLE2 = FreeFile()
    Open App.Path & "\CB_THREAD0.TXT" For Append Access Write Lock Write As #lLOCK_FILEHANDLE2
    CheckAppLockFile = False ' File unlocked-> main thread was terminated-> exit helper threads too
    Close #lLOCK_FILEHANDLE2
  #End If
  Exit Function
lblErr:
  CheckAppLockFile = True
End Function

Public Function WriteLog(isLine As String) As Boolean
  ' write debug log
  Dim lLOCK_FILEHANDLE3 As Long
  #If VBA_MODE = 0 Then
    lLOCK_FILEHANDLE3 = FreeFile()
    Open psEnginePath & "\DEBUG_LOG.TXT" For Append As #lLOCK_FILEHANDLE3
    Print #lLOCK_FILEHANDLE3, isLine
    Close #lLOCK_FILEHANDLE3
  #End If
End Function



Public Sub CheckThreadTermination(ByVal bCheckAlways As Boolean)
  Debug.Assert NoOfThreads > 1
  If ThreadNum >= 1 Then
    If bCheckAlways Or (Nodes > LastThreadCheckNodesCnt + (GUICheckIntervalNodes * 50)) Then
      LastThreadCheckNodesCnt = Nodes
      If Not CheckAppLockFile() Then
```

```vb
          '>>> END of program here because main thread was terminated
          CloseHashMap
          If bThreadTrace Then WriteTrace "!!! Main Thread terminated: Stop helper thread! " & Now()
          End '<<<<
        End If
      End If
    End If
End Sub

Public Function WriteMainThreadStatus(ByVal ilNewThreadStatus As Long) As Long
  Debug.Assert NoOfThreads > 1
  SingleThreadStatus(0) = ilNewThreadStatus
  moHashMap.WriteMapPos HashMapThreadStatusPtr(0), VarPtr(ilNewThreadStatus), CLng(LenB(ilNewThreadStatus))
  If bThreadTrace Then WriteTrace "WriteMainThreadStatus: " & HashMapThreadStatusPtr(0)
End Function

Public Function ReadMainThreadStatus() As Long
  Static LastRead      As Long
  Dim MainThreadStatus As Long
  Debug.Assert NoOfThreads > 1
  moHashMap.ReadMapPos HashMapThreadStatusPtr(0), VarPtr(MainThreadStatus), CLng(LenB(MainThreadStatus))
  SingleThreadStatus(0) = MainThreadStatus
  ReadMainThreadStatus = MainThreadStatus
  If bThreadTrace Then If LastRead <> ReadMainThreadStatus Then WriteTrace "ReadMainThreadStatus:Threadnum=" & ThreadNum & ", Ptr:" & HashMapThreadStatusPtr(0) & ", MainStatus:" & ReadMainThreadStatus & " / " & Now()
  LastRead = ReadMainThreadStatus
End Function

Public Function WriteHelperThreadStatus(ByVal ilThreadNum As Long, _
                                        ByVal ilNewThreadStatus As Long) As Long
  ' Write run status for current thread
  Debug.Assert NoOfThreads > 1 And ilThreadNum > 0
  SingleThreadStatus(ilThreadNum) = ilNewThreadStatus
  moHashMap.WriteMapPos HashMapThreadStatusPtr(ilThreadNum), VarPtr(ilNewThreadStatus), CLng(LenB(ilNewThreadStatus))
End Function

Public Function ReadHelperThreadStatus(ByVal ilThreadNum As Long) As Long
  ' Write run status for current thread
  Dim HelperThreadStatus As Long
  Debug.Assert NoOfThreads > 1 And ilThreadNum > 0
  moHashMap.ReadMapPos HashMapThreadStatusPtr(ilThreadNum), VarPtr(HelperThreadStatus), CLng(LenB(HelperThreadStatus))
  SingleThreadStatus(ilThreadNum) = HelperThreadStatus
  ReadHelperThreadStatus = HelperThreadStatus
End Function

Public Function WriteMapGameData() As Long
  ' Write game moves to map for other threads
  Debug.Assert NoOfThreads > 1
  moHashMap.WriteMapPos HashMapBoardPtr, VarPtr(Board(0)), CLng(LenB(Board(0)) * MAX_BOARD)
  moHashMap.WriteMapPos HashMapMovedPtr, VarPtr(Moved(0)), CLng(LenB(Moved(0)) * MAX_BOARD)
  moHashMap.WriteMapPos HashMapWhiteToMovePtr, VarPtr(bWhiteToMove), CLng(LenB(bWhiteToMove))
  moHashMap.WriteMapPos HashMapGameMovesCntPtr, VarPtr(GameMovesCnt), CLng(LenB(GameMovesCnt))
  arGameMoves(MAX_GAME_MOVES - 1).Target = Fifty ' tricky fix to avoid new map size
  moHashMap.WriteMapPos HashMapGameMovesPtr, VarPtr(arGameMoves(0)), CLng(LenB(arGameMoves(0)) * MAX_GAME_MOVES)
  moHashMap.WriteMapPos HashMapGamePosHashPtr, VarPtr(GamePosHash(0)), CLng(LenB(GamePosHash(0)) * MAX_GAME_MOVES)
```

```
End Function

Public Function ReadMapGameData() As Long
  ' Read game moves to map for other threads
  Dim bToMove As Boolean
  Debug.Assert NoOfThreads > 1
  moHashMap.ReadMapPos HashMapBoardPtr, VarPtr(Board(0)), CLng(LenB(Board(0)) * MAX_BOARD)
  InitEpArr
  moHashMap.ReadMapPos HashMapMovedPtr, VarPtr(Moved(0)), CLng(LenB(Moved(0)) * MAX_BOARD)
  moHashMap.ReadMapPos HashMapWhiteToMovePtr, VarPtr(bToMove), CLng(LenB(bToMove))
  bWhiteToMove = bToMove: bCompIsWhite = bWhiteToMove
  moHashMap.ReadMapPos HashMapGameMovesCntPtr, VarPtr(GameMovesCnt), CLng(LenB(GameMovesCnt))
  moHashMap.ReadMapPos HashMapGameMovesPtr, VarPtr(arGameMoves(0)), CLng(LenB(arGameMoves(0)) * MAX_GAME_MOVES)
  Fifty = arGameMoves(MAX_GAME_MOVES - 1).Target ' tricky fix to avoid new map size
  moHashMap.ReadMapPos HashMapGamePosHashPtr, VarPtr(GamePosHash(0)), CLng(LenB(GamePosHash(0)) * MAX_GAME_MOVES)
  InitPieceSquares
End Function

Public Function ClearMapBestPVforThread() As Long
  Dim th As Long
  Erase BestPV()

  For th = 0 To MAX_THREADS - 1
    moHashMap.WriteMapPos HashMapBestPVPtr(th), VarPtr(BestPV(0)), CLng(LenB(BestPV(0)) * 10)
  Next

End Function

Public Function WriteMapBestPVforThread(ByVal CompletedDepth As Long, _
                                        ByVal BestScore As Long, _
                                        BestMove As TMOVE) As Long
  ' Write PV from helper thread for main thread
  Dim i As Long
  Debug.Assert NoOfThreads > 1
  Debug.Assert HashMapBestPVPtr(ThreadNum) + CLng(LenB(PV(0, 0)) * 10) < HashMapBoardPtr
  ' Use PV0 to store some values... not nice...
  Erase BestPV
  If CompletedDepth > 0 Then

    For i = 0 To GetMin(9, PVLength(1)): BestPV(i) = PV(1, i): Next
    If BestPV(1).From = 0 Then
      ' use BestMove instead
      BestPV(1) = BestMove: BestPV(0).From = 1
    End If
  End If
  BestPV(0).Target = CompletedDepth: BestPV(0).SeeValue = BestScore: BestPV(0).From = GetMin(9, PVLength(1)): BestPV(0).OrderValue = Nodes
  If bThreadTrace Then WriteTrace "WriteMapBestPVforThread: D:" & CompletedDepth & ", PV:" & MoveText(BestPV(1)) & " / " & Now()
  moHashMap.WriteMapPos HashMapBestPVPtr(ThreadNum), VarPtr(BestPV(0)), CLng(LenB(BestPV(0)) * 10)
End Function

Public Function ReadMapBestPVforThread(ByVal SelThread As Long, _
                                       ByRef CompletedDepth As Long, _
                                       ByRef BestScore As Long, _
                                       ByRef BestPVLength As Long, _
                                       ByRef HelperNodes As Long, _
                                       BestPV() As TMOVE) As Boolean
  ' Write PV from helper thread for main thread
  Debug.Assert NoOfThreads > 1
```

```
    Debug.Assert HashMapBestPVPtr(SelThread) + CLng(LenB(BestPV(0)) * 10) <
    HashMapBoardPtr
    ReadMapBestPVforThread = False
    Erase BestPV
    ' Use PV0 to get some values... not nice...
    moHashMap.ReadMapPos HashMapBestPVPtr(SelThread), VarPtr(BestPV(0)), CLng(LenB(
    BestPV(0)) * 10)
    CompletedDepth = BestPV(0).Target: BestScore = BestPV(0).SeeValue: BestPVLength =
    BestPV(0).From: HelperNodes = BestPV(0).OrderValue
    If BestPV(1).From = 0 Or BestPV(1).Target = 0 Then
      If bThreadTrace Then WriteTrace "!!!???ReadMapBestPVforThread:PV Empty Thread:" &
      SelThread & ", Completed Depth:" & CompletedDepth
    End If
    If bThreadTrace Then WriteTrace "ReadMapBestPVforThread: PV:" & MoveText(BestPV(1))
    & " / " & Now()
    ReadMapBestPVforThread = (BestPVLength > 0)
  End Function

  Public Function SetThreads(ByVal iMaxThreads As Long)
    ' set thread numbers: 1-4
    NoOfThreads = GetMax(1, Val("0" & Trim$(ReadINISetting("THREADS", "1"))))
    NoOfThreads = GetMax(NoOfThreads, iMaxThreads)
    NoOfThreads = GetMin(NoOfThreads, MAX_THREADS)
  'NoOfThreads = 2 '#######testc2
    If NoOfThreads <= 1 Then
      ThreadNum = -1 ' Single core mode
    Else
      ThreadNum = 0 ' main thread
    End If
    'WriteTrace "SetThreads= " & NoOfThreads & " / " & Now()
  End Function

  Public Function MaterialHashCompute(ByVal Key As Long) As Long
    If Key = -2147483648# Then Key = Key + 1
    MaterialHashCompute = Abs(Key) Mod MATERIAL_HASHSIZE
  End Function

  Public Function SaveMaterialHash(ByVal Key As Long, ByVal Score As Long)
    Dim Index As Long
    Index = MaterialHashCompute(Key)

    With MaterialHash(Index)
      .Hashkey = Key
      .Score = Score
    End With

  End Function

  Public Function ProbeMaterialHash(ByVal Key As Long) As Long
    Dim Index As Long
    Index = MaterialHashCompute(Key)

    With MaterialHash(Index)
      If .Hashkey = Key Then
        ProbeMaterialHash = .Score
      Else
        ProbeMaterialHash = VALUE_NONE
      End If
    End With

  End Function

  Public Function InIDE() As Boolean
     ' running IDE ( VB development environment) ? if compiled EXE returns false
      Static i As Byte
      i = i + 1
      If i = 1 Then Debug.Assert Not InIDE()
      InIDE = (i = 0)
      i = 0
  End Function

  Public Function GetAppTimeString() As String
    ' returns exe filedatetime with digits only
    Dim p As Long, s As String
    GetAppTimeString = ""
    s = Now()
    #If VBA_MODE = 0 Then
      If Dir(App.EXEName & ".exe") <> "" Then
        s = FileDateTime(App.EXEName & ".exe")
      End If
    #End If
    For p = 1 To Len(s)
      If IsNumeric(Mid$(s, p, 1)) Then GetAppTimeString = GetAppTimeString & Mid$(s, p,
      1)
    Next
  End Function





  VERSION 1.0 CLASS
  BEGIN
    MultiUse = -1  'True
    Persistable = 0  'NotPersistable
    DataBindingBehavior = 0   'vbNone
    DataSourceBehavior  = 0   'vbNone
    MTSTransactionMode  = 0   'NotAnMTSObject
  END
  Attribute VB_Name = "clsHashMap"
  Attribute VB_GlobalNameSpace = False
  Attribute VB_Creatable = True
  Attribute VB_PredeclaredId = False
  Attribute VB_Exposed = False
  Option Explicit
  '
  '==========
  ' HashMap =
  '==========
  '
  '- class for sharing a read/write memory-mapped file
  '- backed by the Windows paging file rather than a specific
  '- disk file between processes running under the same User account
  '- on the same system.
  '
  '- The Name values can optionally be prefixed "Global\" or "Local\"
  '- (see the documentation) and the rest can consist of any
  '- characters except the "\" character.
  '
  '- After we obtain a handle to the object, we'll create a single
  '- "view" containing the entire object as one BLOB.
  '
  '- When all handles to the mapped object have been closed, it disappears.
  '
  Private Const API_NULL              As Long = 0
  Private Const API_FALSE             As Long = 0
  Private Const INVALID_HANDLE_VALUE As Long = -1
  Private Const PAGE_READWRITE        As Long = 4
  Private Const SECTION_MAP_WRITE = &H2
  Private Const FILE_MAP_WRITE = SECTION_MAP_WRITE
  Private Const ERROR_ALREADY_EXISTS As Long = 183

  Private Type SECURITY_ATTRIBUTES
    nLength As Long
    lpSecurityDescriptor As Long
    bInheritHandle As Long
```

```
End Type

Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Private Declare Sub CopyMemory _
                Lib "kernel32" _
                Alias "RtlMoveMemory" (ByVal Destination As Long, _
                                       ByVal Source As Long, _
                                       ByVal Length As Long)
Private Declare Function RtlCompareMemory _
                Lib "ntdll" (ByRef Source1 As Any, _
                             ByRef Source2 As Any, _
                             ByVal Length As Long) As Long

Private Declare Function CreateFileMapping _
                Lib "kernel32" _
                Alias "CreateFileMappingA" (ByVal hFile As Long, _
                                            ByVal lpFileMappigAttributes As Long, _
                                            ByVal flProtect As Long, _
                                            ByVal dwMaximumSizeHigh As Long, _
                                            ByVal dwMaximumSizeLow As Long, _
                                            ByVal lpName As String) As Long
Private Declare Function MapViewOfFile _
                Lib "kernel32" (ByVal hFileMappingObject As Long, _
                                ByVal dwDesiredAccess As Long, _
                                ByVal dwFileOffsetHigh As Long, _
                                ByVal dwFileOffsetLow As Long, _
                                ByVal dwNumberOfBytesToMap As Long) As Long
Private Declare Function OpenFileMapping _
                Lib "kernel32" _
                Alias "OpenFileMappingA" (ByVal dwDesiredAccess As Long, _
                                          ByVal bInheritHandle As Long, _
                                          ByVal lpName As String) As Long
Private Declare Function UnmapViewOfFile _
                Lib "kernel32" (ByVal lpBaseAddress As Long) As Long
Private Declare Sub ZeroMemory2 _
                Lib "kernel32.dll" _
                Alias "RtlZeroMemory" (Destination As Any, _
                                       ByVal Length As Long)
Private hObj           As Long
Private lpMap          As Long
Private mSize          As Long
Private Ptr            As Long
Private CheckData(128) As Byte, CheckDataPtr As Long
Private VerifyArr(500) As Byte


Public Sub CloseMap()
  UnmapViewOfFile lpMap
  CloseHandle hObj
  hObj = 0
End Sub

Public Function CreateMap(ByVal Name As String, ByRef Size As Long) As Boolean
  'Returns True if the memory mapped file already exists.  If so CAUTION,
  'the size will be its previously-created size.
  ' Size may be reduced if not enough memory - and returned to caller !!!
  Dim i As Long, ErrCode As Long
  If Size < 1 Then Err.Raise 5, TypeName(Me), "Size must be at least 1 byte"
  For i = 1 To 10
    DoEvents
    Err.Clear
    hObj = CreateFileMapping(INVALID_HANDLE_VALUE, API_NULL, PAGE_READWRITE, 0, Size, 
    Name)
    ErrCode = Err.LastDllError
    If hObj = API_NULL Then
      Err.Raise &H80049300, TypeName(Me), "CreateFileMapping system error " & CStr(
      Err.LastDllError)
    End If
    CreateMap = (Err.LastDllError = ERROR_ALREADY_EXISTS)

    DoEvents
    lpMap = MapViewOfFile(hObj, FILE_MAP_WRITE, 0, 0, 0)
    If lpMap = API_NULL Then
      Me.CloseMap
      If i = 10 Then
        Err.Raise &H80049302, TypeName(Me), "MapViewOfFile system error " & CStr(Err.LastDllError)
        Exit Function
      ElseIf i >= 3 Then ' try reducing size
        Size = (Size / 3) * 2
      End If

      WriteTrace "***error> CreateMap: " & i & " / " & Name & " / Size= " & Size & " / Err:" & CStr(Err.LastDllError) & " / " & Now()
     ' Err.Raise &H80049302, TypeName(Me), "MapViewOfFile system error " & CStr(Err.LastDllError)
      Sleep 100
    Else
      Exit For
    End If
  Next
  ZeroMemory2 ByVal lpMap, Size
  If bThreadTrace Then WriteTrace "---Creat map:ZERO_MAP / " & Now()
  mSize = Size
End Function

Public Sub ClearMap(ByRef Size As Long)
  ZeroMemory2 ByVal lpMap, Size
  If bThreadTrace Then WriteTrace "--- Clear_MAP / " & Now()
End Sub


Public Sub OpenMap(ByVal Name As String, ByVal Size As Long)
  Dim i As Long
  If Size < 1 Then Err.Raise 5, TypeName(Me), "Size must be at least 1 byte"
  DoEvents
  hObj = OpenFileMapping(FILE_MAP_WRITE, API_FALSE, Name)
  If hObj = API_NULL Then
    Err.Raise &H80049304, TypeName(Me), "OpenFileMapping system error " & CStr(Err.LastDllError)
  End If
  For i = 1 To 5
    DoEvents
    lpMap = MapViewOfFile(hObj, FILE_MAP_WRITE, 0, 0, Size)
    If lpMap = API_NULL Then
      Err.Raise &H80049306, TypeName(Me), "MapViewOfFile system error " & CStr(Err.LastDllError)
    Else
      Exit For
    End If
  Next
  mSize = Size
  CheckDataPtr = VarPtr(CheckData(0))
End Sub

Public Sub ReadMapHashCluster(ByVal Index As Long, _
                              ByVal lpData As Long, _
                              ByVal Size As Long)
  'Pass a pointer lpData and a length in bytes Size.
  Dim i As Long
  If Index * HashRecLen + Size > mSize Then Err.Raise 5, TypeName(Me), "Size must not exceed mapped size"
  If hObj = API_NULL Then Err.Raise &H80049308, TypeName(Me), "ReadMap: Map not open"

  Ptr = lpMap + HashMapSearchPtr + Index * HashRecLen
  'If bHashTrace Then WriteTrace "ReadMapHashCluster: " & Index & "/" & Index & "Ptr:" & Ptr & "/ Nodes:" & Nodes & " / " & Now()
```

```
    For i = 1 To 3 ' about 1 hash collision for 1.000.000.000 endgame nodes measured
      CopyMemory ByVal lpData, ByVal Ptr, ByVal Size
      If bHashVerify Then
        ' If bHashTrace Then WriteTrace "ReadMapHashCluster:Verify :" & VarPtr(VerifyArr(0)) & "Ptr:" & Ptr & Now()
        CopyMemory ByVal VarPtr(VerifyArr(0)), ByVal Ptr, ByVal Size
        ' If bHashTrace Then WriteTrace "ReadMapHashCluster: Compare " & VarPtr(VerifyArr(0)) & "Ptr:" & Ptr &
        Now()
        If RtlCompareMemory(ByVal Ptr, ByVal VarPtr(VerifyArr(0)), ByVal Size) <> Size
        Then
          ' Difference found => try again
          If bTraceHashCollision Then WriteTrace "HashMapDifference: Read " & Index &
          "/" & i & "/ Nodes:" & Nodes & " / " & Now()
        Else
          Exit For
        End If
      Else
        Exit For
      End If
    Next
    'If bHashTrace Then WriteTrace "ReadMapHashCluster:End "
  End Sub

  Public Sub WriteMapHashEntry(ByVal ReplaceIndex As Long, ByVal lpData As Long)
    'Pass a pointer lpData and a length in bytes Size.
    Dim i As Long
    If (ReplaceIndex + 1) * HashRecLen > mSize Then Err.Raise 5, TypeName(Me), "Size
    must not exceed mapped size"
    If hObj = API_NULL Then Err.Raise &H8004930A, TypeName(Me), "WriteMap: Map not open"
    Ptr = lpMap + HashMapSearchPtr + ReplaceIndex * HashRecLen

    For i = 1 To 3  ' about 1 hash collision for 1.000.000.000 endgame nodes measured
      CopyMemory ByVal Ptr, ByVal lpData, ByVal HashRecLen
      '--- Reread the written entry to verify that there was no parallel write from other thread that mixed up the data
      '--- Try max 3 times
      If RtlCompareMemory(ByVal Ptr, ByVal lpData, ByVal HashRecLen) <> HashRecLen Then
        ' Difference found => try again
        If bTraceHashCollision Then WriteTrace "HashMapDifference: Write " &
        ReplaceIndex & "/" & i & "/ Nodes:" & Nodes & " / " & Now()
      Else
        Exit For
      End If
    Next

  End Sub

  Public Sub WriteMapPos(ByVal StartPos As Long, ByVal lpData As Long, ByVal Size As
  Long)
    'Pass a pointer lpData and a length in bytes Size.
    If StartPos + Size > mSize Then Err.Raise 5, TypeName(Me), "Size must not exceed
    mapped size"
    If hObj = API_NULL Then Err.Raise &H8004930A, TypeName(Me), "WriteMap: Map not open"
    Ptr = lpMap + StartPos
    CopyMemory ByVal Ptr, ByVal lpData, ByVal Size
  End Sub

  Public Sub ReadMapPos(ByVal StartPos As Long, ByVal lpData As Long, ByVal Size As Long
  )
    'Pass a pointer lpData and a length in bytes Size.
    If StartPos + Size > mSize Then Err.Raise 5, TypeName(Me), "Size must not exceed
    mapped size"
    If hObj = API_NULL Then Err.Raise &H80049308, TypeName(Me), "ReadMap: Map not open"
    Ptr = lpMap + StartPos
    CopyMemory ByVal lpData, ByVal Ptr, ByVal Size
  End Sub

  Private Sub Class_Terminate()
    If hObj <> 0 Then CloseMap
  End Sub
```

```
Attribute VB_Name = "basIO"
'==================================================
'= IOBas:
'= Winboard / UCI communication / output of think results
'==================================================
Option Explicit
'--- Win32 API functions
Declare Function GetStdHandle Lib "kernel32" (ByVal nStdHandle As Long) As Long
Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Declare Function PeekNamedPipe _
        Lib "kernel32" (ByVal hNamedPipe As Long, _
                        lpBuffer As Any, _
                        ByVal nBufferSize As Long, _
                        lpBytesRead As Long, _
                        lpTotalBytesAvail As Long, _
                        lpBytesLeftThisMessage As Long) As Long
Declare Function ReadFile _
        Lib "kernel32" (ByVal hFile As Long, _
                        lpBuffer As Any, _
                        ByVal nNumberOfBytesToRead As Long, _
                        lpNumberOfBytesRead As Long, _
                        lpOverlapped As Any) As Long
Declare Function WriteFile _
        Lib "kernel32" (ByVal hFile As Long, _
                        ByVal lpBuffer As String, _
                        ByVal nNumberOfBytesToWrite As Long, _
                        lpNumberOfBytesWritten As Long, _
                        lpOverlapped As Any) As Long
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)
Declare Function GetPrivateProfileString _
        Lib "kernel32" _
        Alias "GetPrivateProfileStringA" (ByVal lpApplicationName As String, _
                                          ByVal lpKeyName As Any, _
                                          ByVal lpDefault As String, _
                                          ByVal lpReturnedString As String, _
                                          ByVal nSize As Long, _
                                          ByVal lpFileName As String) As Long
Declare Function WritePrivateProfileString _
        Lib "kernel32" _
        Alias "WritePrivateProfileStringA" (ByVal lpApplicationName As String, _
                                            ByVal lpKeyName As Any, _
                                            ByVal lpString As Any, _
                                            ByVal lpFileName As String) As Long
Public Declare Sub ZeroMemory2 _
               Lib "kernel32.dll" _
               Alias "RtlZeroMemory" (Destination As Any, _
                                      ByVal Length As Long)
Public hStdIn  As Long    ' Handle Standard Input
Public hStdOut As Long    ' Handle Standard Output
Public Const STD_INPUT_HANDLE = -10&
Public Const STD_OUTPUT_HANDLE = -11&
Public psEnginePath            As String     ' path of engine directory (init different VB6 / Office)
Public psDocumentPath          As String     ' path of office document
Public pbIsOfficeMode          As Boolean
Public plLastPostNodes         As Long ' to avoid duplicate outputs
Public EGTBasesEnabled         As Boolean
Public EGTBasesMaxPieces       As Long  ' 3,4,5,6 piece set
Public EGTBasesMaxPly          As Long ' max ply using EGTB in search
Public EGTBasesPath            As String   ' SYZYGY EGTB files path
Private oProxy                 As Object ' for online tablebases
Public bEGTbBaseTrace          As Boolean
Public EGTBasesHitsCnt         As Long ' count for GUI output
Public EGTBRootProbeDone       As Boolean
Public EGTBRootResultScore     As Long
Public EGTBBestMoveStr         As String, EGTBBestMoveListStr As String
Public EGTBMoveListCnt(MAX_PV) As Long, EGTBMoveList(MAX_PV, 199) As String
Public UCISyzygyPath           As String
Public UCISyzygyMaxPieceSet    As Long
Public UCISyzygyMaxPly          As Long
'-------------------------------
' Log file
'-------------------------------
Public bLogPV                   As Boolean   ' log PV in post mode
Public bLogMode                 As Boolean
Public LogFile                  As Long
Public LastFullPV               As String
Public LanguageENArr(200)      As String
Public LanguageArr(200)        As String
Public LangCnt                  As Long
Public psLanguage               As String

'------------------------------------------------------------------------
Public Sub OpenCommHandles()
  ' Open IO channels to Winboard
  hStdIn = GetStdHandle(STD_INPUT_HANDLE)
  hStdOut = GetStdHandle(STD_OUTPUT_HANDLE)
End Sub

Public Sub CloseCommChannels()
  ' Close IO channels to Winboard
  CloseHandle hStdIn
  CloseHandle hStdOut
  If EGTBasesEnabled And Not DebugMode Then
    ' wait to avoid windows error when programs exits in AREAN after tablesbase access  in Win7 ( ok for Win10)
    Dim i As Long

    For i = 1 To 15
      Sleep 500
      DoEvents
    Next

  End If
End Sub

'------------------------------------------------------------------------
'PollCommand() - check standard input
'
' returns TRUE if data found
'------------------------------------------------------------------------
Function PollCommand() As Boolean
  If ThreadNum <= 0 Then
    #If DEBUG_MODE <> 0 Then
      ' from Debug form
      PollCommand = FakeInputState
    #Else
      ' winboard input
      Dim sBuff       As String
      Dim lBytesRead  As Long
      Dim lTotalBytes As Long
      Dim lAvailBytes As Long
      Dim rc          As Long

      sBuff = String(4096, Chr$(0))
      rc = PeekNamedPipe(hStdIn, ByVal sBuff, 4096, lBytesRead, lTotalBytes,
      lAvailBytes)
      PollCommand = CBool(rc And lBytesRead > 0)
    #End If
  Else
    '--- Multi-thread mode: helper threads get commands from main thread

    MainThreadStatus = ReadMainThreadStatus()

    'If bThreadTrace Then WriteTrace "PollCommand: ThreadStatusCheck:" & MainThreadStatus & " " &
    LastThreadStatus & " / " & Now()
    Select Case MainThreadStatus
      Case 1
```

```
          If LastThreadStatus <> MainThreadStatus Then
            ThreadCommand = "go" & vbLf: PollCommand = True
            If bThreadTrace Then WriteTrace "PollCommand: MainThreadStatus = 1" & " / "
            & Now()
        End If
        Case 0
          If LastThreadStatus <> MainThreadStatus Then
            ThreadCommand = "exit" & vbLf: PollCommand = True: bTimeExit = True
            If bThreadTrace Then WriteTrace "PollCommand: MainThreadStatus = 0" & " / "
            & Now()
          Else
            Sleep 25
          End If
      End Select

      LastThreadStatus = MainThreadStatus
    End If
End Function

'--------------------------------------------------------------------------
'ReadCommand()
'--------------------------------------------------------------------------
Function ReadCommand() As String
    If ThreadNum > 0 Then
      If bThreadTrace Then WriteTrace "ReadCommand: ThreadCommand = " & ThreadCommand &
      " / " & Now()
      ReadCommand = ThreadCommand
      ThreadCommand = ""
      Exit Function
    End If
    #If DEBUG_MODE <> 0 Then
      ReadCommand = FakeInput ' from Debug form
      FakeInputState = False
      FakeInput = ""
    #Else
      Dim sBuff      As String
      Dim lBytesRead As Long
      Dim rc         As Long
      sBuff = String$(4096, Chr$(0))
      rc = ReadFile(hStdIn, ByVal sBuff, 4096, lBytesRead, ByVal 0&)
      ReadCommand = Left$(sBuff, lBytesRead)
    #End If
End Function

'--------------------------------------------------------------------------
'SendCommand()
'
'--------------------------------------------------------------------------
Function SendCommand(ByVal sCommand As String) As String
    Dim p As Long, s As String, sOut As String
    #If VBA_MODE = 1 Then

      ' OFFICE VBA
      With frmChessX
        If .txtIO.Visible Then
          If Len(.txtIO) > 64000 Then .txtIO = ""
          If .txtIO <> "" Then .txtIO = .txtIO & vbCrLf
          If Len(sCommand) > 120 Then
            s = sCommand: sOut = ""
            Do While Len(s) > 120
              p = InStrRev(Left$(s, 120), " ")
              sOut = sOut & Left$(s, p)
              s = Trim$(Mid$(s, p + 1))
              If s <> "" Then sOut = sOut & vbCrLf & Space(14)
            Loop
            sOut = sOut & s
            .txtIO = .txtIO & sOut
          Else
```

```
              .txtIO = .txtIO & sCommand
            End If
            .txtIO.SetFocus
            .txtIO.SelStart = Len(.txtIO)
            .txtIO.SelLength = 0
            DoEvents
          End If
        End With

      #End If
      #If DEBUG_MODE <> 0 Then

        ' VB DEBUG FORM
        With frmDebugMain
          If Len(.txtIO) > 32000 Then .txtIO = Left$(.txtIO, 8000)
          .txtIO = .txtIO & vbCrLf & sCommand
          .txtIO.SelStart = Len(.txtIO)
          .txtIO.SelLength = 0
          .Refresh
        End With

      #End If
      #If DEBUG_MODE = 0 And VBA_MODE = 0 Then
        ' WINBOARD STDOUT channel
        Dim lBytesWritten As Long
        Dim lBytes        As Long
        Dim rc            As Long
        sCommand = vbLf & sCommand & vbLf
        lBytes = Len(sCommand)
        rc = WriteFile(hStdOut, ByVal sCommand, lBytes, lBytesWritten, ByVal 0&)
      #End If
      SendCommand = sCommand
    End Function

    Public Sub WriteGame(sFile As String)
      '--- Write file for game un UCI format
      '
      ' Format:
      '[Event "F/S Return Match"]
      '[Site "Belgrade, Serbia Yugoslavia|JUG"]
      '[Date "1992.11.04"]
      '[Round "29"]
      '[White "Fischer, Robert J."]
      '[Black "Spassky, Boris V."]
      '[Result "1/2-1/2"]
      ' 1. e2e4 e7e5 2. c2c4 f8e7 3. d2d4 e5d4 4. b1c3 d4c3
      Dim i As Long, h As Long, s As String, MoveCnt As Long, Cnt As Long
      Cnt = GameMovesCnt
      If Cnt = 0 Then Exit Sub
      s = "": MoveCnt = 0

      For i = 1 To Cnt Step 2
        MoveCnt = MoveCnt + 1
        s = s & CStr(MoveCnt) & ". " & CompToCoord(arGameMoves(i))
        If i + 1 <= Cnt Then s = s & " " & CompToCoord(arGameMoves(i + 1)) & " "
      Next i

      If s <> "" Then
        h = FreeFile()
        Open sFile For Append Lock Write As #h
        Print #h, "[Date " & Chr$(34) & Format(Now(), "YYYY.MM.DD HH:NN") & Chr$(34) & "]"
        Print #h, "[White " & Chr$(34) & "?" & Chr$(34) & "]"
        Print #h, "[Black " & Chr$(34) & "?" & Chr$(34) & "]"
        Print #h, "[Result " & Chr$(34) & "?" & Chr$(34) & "]"
        Print #h, s
        Close #h
      End If
    End Sub
```

```
Public Sub ReadGame(sFile As String)
  ' Read PGN File
  Dim h             As Long, s As String, m As Long, sInp As String, m1 As String, m2
  As String
  Dim asMoveList() As String
  InitGame
  BookMovePossible = False
  bForceMode = True
  h = 10 'FreeFile()
  Open sFile For Input As #h

  Do Until EOF(h)
    Line Input #h, sInp
    sInp = Trim(sInp) & "  "
    If Left(sInp, 1) <> "[" Then '--- Ignore Header Tags
      asMoveList = Split(sInp, ".") ' split at move number dot

      For m = 0 To UBound(asMoveList)
        s = asMoveList(m)
        s = Replace(s, "-", "")
        s = Replace(s, "x", "")
        s = Replace(s, "+", "")
        s = Left(s, 10)
        If Left(s, 1) = " " Then ' behind move number
          s = Trim(s)
          'Debug.Print s
          m1 = Trim(Left(s, 4))
          If Len(m1) = 4 Then
            'Debug.Print m1, asMoveList(m)
            ParseCommand m1 & vbLf
          End If
          If Len(s) > 8 Then
            m2 = Trim$(Mid(s, 6, 4))
            If Len(m2) >= 4 Then
              'Debug.Print m2, asMoveList(m)
              ParseCommand m2 & vbLf
            End If
          End If
        End If
      Next

    End If
  Loop

  Close #h
End Sub

Public Sub SendThinkInfo(Elapsed As Single, ActDepth As Long, CurrentScore As Long,
Alpha As Long, Beta As Long)
  Static FinalMoveForHint As TMOVE
  Static sLastInfo As String
  Dim sPost             As String, j As Long, sPostPV As String
  'pbIsOfficeMode = False 'Test
  If pbIsOfficeMode Then
    '--- MS OFFICE
    sPost = "  " & Translate("Depth") & ":" & ActDepth & "/" & MaxPly & "  " & Translate
    ("Score") & ":" & FormatScore(EvalSFTo100(CurrentScore)) & "  " & Translate("Nodes"
    ) & ":" & Format("0.000", CalcNodes()) & "  " & Translate("Sec") & ":" & Format(
    Elapsed, "0.00")
    If plLastPostNodes <> CalcNodes() Then
      SendCommand sPost
      plLastPostNodes = CalcNodes()
      sPostPV = "       >" & Translate("Line") & ": "

      For j = 1 To PVLength(1) - 1
        sPostPV = sPostPV & "  " & GUIMoveText(PV(1, j))
        ' Save Hint move
```

```
          If j = 1 And Not MovesEqual(FinalMoveForHint, PV(1, 1)) Then HintMove =
          EmptyMove ' for case that 1. ply as hash move only
          If j = 2 Then
            If PV(1, j).From > 0 Then HintMove = PV(1, j): FinalMoveForHint = PV(1, 1)
          End If
        Next

        If sPost <> sLastInfo Then
          SendCommand sPostPV
          sLastInfo = sPost
          ShowMoveInfo MoveText(FinalMove), ActDepth, MaxPly, EvalSFTo100(CurrentScore),
           Elapsed ' VBA mode only
        End If
      End If
    Else
      '--- VB6
      If UCIMode Then
        ' format: info depth 1 seldepth 1 multipv 1 score cp 417 nodes 51 nps 25500 tbhits 0 time 2 pv e8g8
        sPost = "info depth " & ActDepth & " seldepth " & MaxPly & " multipv 1 score " &
         UciGUIScore(CurrentScore, Alpha, Beta)
        If Nodes > 1000 Then sPost = sPost & " hashfull " & HashUsageUCI()
        sPost = sPost & " nodes " & CalcNodes() & " nps " & CalcNPS(Elapsed) & " tbhits
        " & EGTBasesHitsCnt & " time " & Int(Elapsed * 1000#) & " pv"
      Else
        sPost = ActDepth & " " & EvalSFTo100(CurrentScore) & " " & (Int(Elapsed) * 100)
        & " " & CalcNodes()
      End If

      sPostPV = ""
      For j = 1 To GetMax(1, PVLength(1) - 1)
        If PV(1, j).From <> 0 Then sPostPV = sPostPV & " " & GUIMoveText(PV(1, j))
      Next

      Dim bLastFullPVUsed As Boolean
      bLastFullPVUsed = False
      If Len(Trim(sPostPV)) > 12 Then ' more than 2 moves
        If Len(Trim(sPostPV)) < Len(Trim(LastFullPV)) Then
          If Left(Trim(LastFullPV), Len(Trim(sPostPV))) = Trim(sPostPV) Then
           sPostPV = LastFullPV
           bLastFullPVUsed = True
          End If
        End If
        If Not bLastFullPVUsed Then
          LastFullPV = sPostPV
          LastFullPVLen = PVLength(1)
          For j = 1 To PVLength(1): SetMove LastFullPVArr(j), PV(1, j): Next
        End If
      Else
        If Left(Trim(sPostPV), 5) = Left(Trim(LastFullPV), 5) Then
          If Len(Trim(sPostPV)) < Len(Trim(LastFullPV)) Then
            sPostPV = LastFullPV
          End If
        End If
      End If
      sPost = sPost & sPostPV
      If Not UCIMode And Not bWbPvInUciFormat Then sPost = sPost & "(" & MaxPly & "/" &
      HashUsagePerc & ")"
      If Not GotExitCommand() Then
        If sPost <> sLastInfo Then
         SendCommand sPost
         sLastInfo = sPost
        End If
      End If
    End If
End Sub

'Public Sub SendRootInfo(Elapsed As Single, ActDepth As Long, CurrentScore As Long, Alpha As Long, Beta As
Long)
```

```vb
' Dim sPost As String, j As Long, sPV As String
' 'CurrentScore = ScaleScoreByEGTB(CurrentScore)
' If pbIsOfficeMode Then
'   '--- MS OFFICE
'   sPost = " " & Translate("Depth") & ":" & ActDepth & "/" & MaxPly & " " & Translate("Score") & ":" & FormatScore(EvalSFTo100(CurrentScore)) & " " & Translate("Nodes") & ":" & Format("0.000", CalcNodes()) & " " & Translate("Sec") & ":" & Format(Elapsed, "0.00")
'   If plLastPostNodes <> Nodes Or Nodes = 0 Then
'     SendCommand sPost
'     plLastPostNodes = Nodes
'     sPost = "      >Line: "
'
'     For j = 1 To PVLength(1) - 1
'       sPost = sPost & " " & MoveText(PV(1, j))
'     Next
'
'     SendCommand sPost
'     ShowMoveInfo MoveText(FinalMove), ActDepth, MaxPly, EvalSFTo100(CurrentScore), Elapsed
'   End If
' Else
'   ' VB6
'   If UCIMode Then
'     ' format: info depth 1 seldepth 1 multipv 1 score cp 417 nodes 51 nps 25500 tbhits 0 time 2 pv e8g8
'     sPost = "info depth " & ActDepth & " seldepth " & MaxPly & " multipv 1 score " & UciGUIScore(CurrentScore, Alpha, Beta) & " nodes " & CalcNodes() & " nps " & CalcNPS(Elapsed) & " tbhits " & EGTBasesHitsCnt & " time " & Int(Elapsed * 1000#) & " pv"
'   Else
'     sPost = ActDepth & " " & EvalSFTo100(CurrentScore) & " " & (Int(Elapsed) * 100) & " " & CalcNodes()
'   End If
'   sPV = ""
'
'   For j = 1 To PVLength(1) - 1
'     If PV(1, j).From <> 0 Then sPV = sPV & " " & GUIMoveText(PV(1, j))
'   Next
'
'   If Len(Trim(sPV)) > 8 Then
'     LastFullPV = sPV
'   Else
'     If Trim(Left(sPV, 5)) = Trim(Left(LastFullPV, 5)) Then
'       sPV = LastFullPV
'     End If
'   End If
'   sPost = sPost & sPV
'   If Not GotExitCommand() Then
'     SendCommand sPost
'   End If
' End If
' If bWinboardTrace Then If bLogPV Then LogWrite Space(6) & sPost
'End Sub

Public Function GotExitCommand() As Boolean
  Dim sInput As String
  GotExitCommand = False
  If PollCommand Then
    sInput = ReadCommand
    If Left$(sInput, 1) = "." Then
      SendAnalyzeInfo
    Else
      If sInput <> "" Then
        ParseCommand sInput
        GotExitCommand = bExitReceived
      End If
    End If
  End If
End Function

Public Function FormatScore(ByVal lScore As Long) As String
  If lScore < -MATE_IN_MAX_PLY And lScore >= -MATE0 Then
```

```vb
    FormatScore = "-M" & CStr((Abs(MATE0) - Abs(lScore)) \ 2)
  ElseIf lScore > MATE_IN_MAX_PLY And lScore <= MATE0 Then
    FormatScore = "+M" & (MATE0 - lScore) \ 2
  ElseIf lScore = VALUE_NONE Then
    FormatScore = "?"
  Else
    FormatScore = Format$(lScore / 100#, "+0.00;-0.00")
  End If
End Function

Public Sub SendAnalyzeInfo()
  Dim sPost As String, Elapsed As Single
  Elapsed = TimeElapsed
  sPost = "stat01: " & Int(Elapsed) & " " & CalcNodes() & " " & RootDepth & " " & "1 1"
  If Not GotExitCommand() Then
    SendCommand sPost
  End If
End Sub

Public Sub WriteTrace(s As String)
  Dim h As Long
  On Error Resume Next
  'Debug.Print s
  If s <> "" Then
    h = FreeFile()
    If ThreadNum <= 0 Then
      Open psEnginePath & "\Trace_" & Format(Date, "YYMMDD") & ".txt" For Append Lock Write As #h
    Else
      Open psEnginePath & "\Trace_" & Format(Date, "YYMMDD") & "_T" & Trim(CStr(GetMax(0, ThreadNum))) & ".txt" For Append Lock Write As #h
    End If
    Print #h, s
    Close #h
  End If
  If pbIsOfficeMode Then SendCommand s
End Sub

'---------------------------------------------------------------------------
'ReadINISetting: Read values fromm INI file
'---------------------------------------------------------------------------
Function ReadINISetting(ByVal sSetting As String, ByVal sDefault As String) As String
  Dim sBuffer    As String
  Dim lBufferLen As Long
  sBuffer = Space(260)
  lBufferLen = GetPrivateProfileString("Engine", sSetting, sDefault, sBuffer, 260, psEnginePath & "\" & INI_FILE)
  If lBufferLen > 0 Then
    ReadINISetting = Left$(sBuffer, lBufferLen)
  Else
    'LogWrite "Error retrieving setting: " & sSetting, True, True
  End If
End Function

'---------------------------------------------------------------------------
' WriteINISetting: write values to INI file
'---------------------------------------------------------------------------
Function WriteINISetting(ByVal sSetting As String, ByVal sValue As String) As Boolean
  Dim lBufferLen As Long
  lBufferLen = WritePrivateProfileString("Engine", sSetting, sValue, psEnginePath & "\" & INI_FILE)
  If lBufferLen > 0 Then
    WriteINISetting = True
  Else
    LogWrite "Error writing setting: " & sSetting & "=" & sValue, True
    WriteINISetting = False
  End If
End Function

'------------------------------------------------------------------------
'LogWrite: Write log file
'bTime adds the time
'------------------------------------------------------------------------
Public Sub LogWrite(sLogString As String, Optional ByVal BTime As Boolean)
  Dim sStr As String
  LogFile = FreeFile
  sStr = sLogString
  If BTime Then sStr = Now & " - " & sStr
  Open psEnginePath & "\" & LCase(psAppName) & ".log" For Append Lock Write As
  #LogFile
  Print #LogFile, sStr
  'Debug.Print sStr
  Close #LogFile
End Sub

Public Sub ShowMoveInfo(ByVal sMove As String, _
                        ByVal lDepth As Long, _
                        ByVal lMaxPly As Long, _
                        ByVal lScore As Long, _
                        ByVal lTime As Single)
  #If VBA_MODE Then

    With frmChessX
      If InStr(sMove, "x") = 0 Then
        .lblMove = Translate("Move") & ": " & UCase(Left$(sMove, 2)) & "-" & UCase$(
        Mid$(sMove, 3))
      Else
        .lblMove = Translate("Move") & ": " & UCase(Left$(sMove, 2)) & "x" & UCase$(
        Mid$(sMove, 4))
      End If
      .lblDepth = Translate("Depth") & ": " & CStr(lDepth) & "/" & CStr(lMaxPly) & ":"
       & CStr(RootMoveCnt)
      .lblScore = Translate("Score") & " : " & FormatScore(lScore)
      .lblTime = Translate("Time") & ": " & Format(lTime, "0.00") & "s"
      DoEvents
    End With

  #End If
End Sub

Public Function FieldNumToCoord(ByVal ilFieldNum As Long) As String
  FieldNumToCoord = Chr$(Asc("a") + ((ilFieldNum - 1) Mod 8)) & Chr$(Asc("1") + ((
  ilFieldNum - 1) \ 8))
End Function

'
'--- Translate functions ---
'
Public Sub ReadLangFile(ByVal isLanguage As String)
  '--- sample: isLanguage = "DE"
  Dim sLine   As String
  Dim i       As Long
  Dim sFile   As String
  Dim f       As Long
  Dim c       As String
  Dim sTextEN As String
  Dim sText   As String
  sFile = psEnginePath & "\ChessBrainVB_Language_" & isLanguage & ".txt"
  LangCnt = 0
  If Dir(sFile) <> "" Then
    f = FreeFile()
    Open sFile For Input As #f

    Do While Not EOF(f)
      Line Input #f, sLine
```

```vb
        sLine = Trim$(sLine) 'Input
        If Not sLine = "" Then
          'Debug.Print sLine
          c = Left$(LTrim$(sLine), 1)
          If c <> ";" Then
            If StringSplit(sLine, sTextEN, sText) Then
              LangCnt = LangCnt + 1
              LanguageENArr(LangCnt) = sTextEN
              LanguageArr(LangCnt) = sText
            End If
          End If
        End If
      Loop

      Close #f
    End If ' File Exists
  End Sub

  Public Sub InitTranslate()

    If pbMSExcelRunning Then
      psLanguage = "EN"
      #If VBA_MODE = 1 Then
      InitTranslateExcel
      #End If
    Else ' other VBA Office
      psLanguage = "EN"
      ReadLangFile "DE"
    End If
  End Sub

  Public Function Translate(ByVal isTextEN As String) As String
    Dim i As Long
    If pbIsOfficeMode Then

      For i = 1 To LangCnt
        If LanguageENArr(i) = isTextEN Then Translate = LanguageArr(i): Exit Function
      Next

    End If
    Translate = isTextEN
  End Function

  Public Function StringSplit(sInput As String, _
                              ByRef sTextEN As String, _
                              ByRef sText As String) As Boolean
    'Split String from Format "english#languageX#"
    Dim v As Variant
    v = Split(sInput, "#", -1, vbBinaryCompare)
    If Not UBound(v) = 2 Then
      StringSplit = False
      Exit Function
    End If
    sTextEN = v(0): sText = v(1): StringSplit = True
  End Function

  Public Function InitTableBases() As Boolean
    On Error GoTo lblErr
    EGTBasesEnabled = CBool(Trim(ReadINISetting("EGTB_ENABLED", "0")) = "1") Or
    TableBasesRootEnabled
    If Not EGTBasesEnabled Then InitTableBases = False: Exit Function
    'pbIsOfficeMode = True 'TEST
    If pbIsOfficeMode Then ' for VBA-GUI only
      ' Online endgame tablebases
      ' see: https://github.com/lichess-org/lila-tablebase
      EGTBasesMaxPieces = 7
      EGTBasesMaxPly = 1
      InitTableBases = True
    Else
      ' winboard / UCI mode: using SYZYGY endgame tablebases
      EGTBasesPath = Trim(ReadINISetting("TB_SYZYGY_PATH", psEnginePath))
      If UCIMode And Trim$(UCISyzygyPath) <> "" Then
        EGTBasesPath = UCISyzygyPath
      End If
      EGTBasesMaxPieces = Val("0" & ReadINISetting("TB_SYZYGY_MAX_PIECES", "0"))
      If UCIMode And UCISyzygyMaxPieceSet > 0 Then
        EGTBasesMaxPieces = UCISyzygyMaxPieceSet
      End If
      ' probe for first x plies only
      EGTBasesMaxPly = Val("0" & ReadINISetting("TB_SYZYGY_MAX_PLY", "1"))   ' ply 1=root
      InitTableBases = (EGTBasesMaxPieces > 2 And EGTBasesPath <> "")
      If UCIMode And UCISyzygyMaxPly > 0 Then
        EGTBasesMaxPly = UCISyzygyMaxPly
      End If
      If Trim$(EGTBasesPath) = "" Then EGTBasesEnabled = False: Exit Function
      '
      EGTBasesHitsCnt = 0
      If InitTableBases Then
        Dim ResultScore As Long, BestMove As String, MoveListStr As String, MoveCnt As
        Long
        InitTableBases = ProbeEGTB("8/8/8/3k4/5P2/5K2/8/8 b - - 0 1", ResultScore, True,
         BestMove, MoveListStr)
        If UCIMode Then
          If InitTableBases Then
            SendCommand "info string tablebases found"
          Else
            SendCommand "info string tablebases not found at:" & EGTBasesPath
          End If
        End If
      End If
      If bEGTbBaseTrace Then WriteTrace "InitTableBases: Path:" & EGTBasesPath & "
      PieceSet:" & EGTBasesMaxPieces & " > " & InitTableBases
    End If
    If bEGTbBaseTrace Then WriteTrace "Init endgame tablebase OK! "
  lblExit:
    Exit Function
  lblErr:
    If bEGTbBaseTrace Then WriteTrace "Init endgame tablebase:ERROR! "
    InitTableBases = False
    EGTBasesEnabled = False
    Resume lblExit
  End Function

  Public Function IsTimeForEGTbBaseProbe() As Boolean
    If Not pbIsOfficeMode Then
      IsTimeForEGTbBaseProbe = False
      If FixedDepth <> NO_FIXED_DEPTH Then IsTimeForEGTbBaseProbe = True: Exit Function
      ' If Ply < GetMax(3, RootDepth \ 3) Then
      If CBool(TimeLeft > 1.5) Then
        IsTimeForEGTbBaseProbe = True
      End If
      ' End If
    Else
      '  max 20 sec for initial online TB call needed, expect refresh after 30 min pause
      IsTimeForEGTbBaseProbe = CBool(TimeLeft > 20 Or FixedDepth <> NO_FIXED_DEPTH)
    End If
    If bEGTbBaseTrace And Not IsTimeForEGTbBaseProbe Then WriteTrace "No time for
    endgame tablebase access: " & TimeLeft
  End Function

  Public Function IsEGTbBasePosition() As Boolean
    Dim ActPieceCnt As Long
    ActPieceCnt = 2 + WNonPawnPieces + PieceCnt(WPAWN) + BNonPawnPieces + PieceCnt(BPAWN
    )
    IsEGTbBasePosition = CBool(ActPieceCnt <= EGTBasesMaxPieces)
  End Function
```

```vb

Public Sub TestTableBase()
  Dim sFEN As String, GameResultScore As Long, BestMove As String, BestMovesList As
  String
  Dim i    As Long
pbIsOfficeMode = True
TableBasesRootEnabled = True
InitTableBases

  For i = 1 To 1
    If i Mod 2 = BCOL Then
      sFEN = "6k1/6p1/8/8/8/8/4P2P/6K1 b - -"
    Else
      sFEN = "7k/4P3/6K1/8/8/8/8/8 w - -"
      'sFEN = "R7/P4k2/8/8/8/8/r7/6K1 w - -"
    End If
    sFEN = "8/6k1/6p1/8/7r/3P1KP1/8/8 w - - 0 1"
    sFEN = "2Q1k3/8/4K3/8/8/3P4/8/8 b - - 0 11"
    sFEN = "8/k7/2K5/8/3P4/1Q6/8/8 b - - 0 11"

    If ProbeTablebases(sFEN, GameResultScore, True, BestMove, BestMovesList) Then
      Debug.Print sFEN & " / Score: " & GameResultScore & "  > " & BestMove & " / " &
      Left(BestMovesList, 80)
      DoEvents
    Else
      Debug.Print "Error"
    End If
  Next

End Sub

Public Function ProbeTablebases(ByVal sFEN As String, _
                                ByRef GameResultScore As Long, _
                                ByVal bShowBestMoves As Boolean, _
                                ByRef BestMove As String, _
                                ByRef BestMovesList As String) As Boolean
  If pbIsOfficeMode Then
    ProbeTablebases = ProbeOnlineEGTB(sFEN, GameResultScore, BestMove, BestMovesList)
  Else
    ProbeTablebases = ProbeEGTB(sFEN, GameResultScore, bShowBestMoves, BestMove,
    BestMovesList)
  End If
End Function

'Public Function ProbeOnlineEGTB(ByVal sFEN As String, _
'                                ByRef GameResultScore As Long, _
'                                ByVal bShowBestMoves As Boolean, _
'                                ByRef BestMove As String, _
'                                ByRef BestMovesList As String) As Boolean
' ' Online Web Access needed !
' ' Documentation: http://www.lokasoft.nl/tbapi.aspx
' ' Comsvcs.dll needed
' ' function returns false if no result
'  Static bInitDone As Boolean
'  Static bInitOk   As Boolean
'  Dim sResult      As String, sCommand As String
'  GameResultScore = VALUE_NONE: BestMove = "": BestMovesList = "": ProbeOnlineEGTB = False
'  If Not bInitDone Then
'    bInitOk = InitTableBases()
'    bInitDone = True
'  End If
'  If Not bInitOk Then ProbeOnlineEGTB = False: Exit Function
'  On Error GoTo lblErr
' ' The score is given as distance to mat, or 0 when the position is a draw.
' ' An error response is returned when position is invalid or not in database. '
' ' e.g.  M5 = color to move gives mate in 5 , -M3 = color to move gets mated in 5 moves.
'  sCommand = "curl http://tablebase.lichess.ovh/standard/mainline?fen=4k3/8/8/8/8/4K3/8/8_w_-_"
'  sResult = GetCommandOutput(sCommand)
```

```vb
'
'  If Trim$(sResult) = "" Then Exit Function  ' TODO Trim$(oProxy.ProbePosition(sFEN))
'  If sResult = "0" Then
'    GameResultScore = 0
'  ElseIf Left$(sResult, 1) = "M" Then
'    GameResultScore = MATE0 - 2 * Val("0" & Mid$(sResult, 2))
'  ElseIf Left$(sResult, 2) = "-M" Then
'    GameResultScore = -MATE0 + 2 * Val("0" & Mid$(sResult, 3))
'  End If
'  ' Shows list of best move
'  If GameResultScore <> VALUE_NONE Then
'    ProbeOnlineEGTB = True
'    If bShowBestMoves Then
'      BestMovesList = "" ' TODO
'    End If
'  End If
'  If bEGTbBaseTrace Then WriteTrace "endgame tablebase move: " & BestMove & " / Score: " & GameResultScore & " " & Now() & vbCrLf & PrintPos()
'lblExit:
'  Exit Function
'lblErr:
'  bInitDone = False
'  ProbeOnlineEGTB = False
'  Resume lblExit
'End Function

Public Function ExtractFirstTbMove(ByVal sMoveList As String) As String
  Dim sMove As String, p As Long, c As String

  For p = 1 To Len(sMoveList)
    c = Mid$(sMoveList, p, 1)
    If (c >= "a" And c <= "h") Or (c >= "0" And c <= "9") Then
      If Len(sMove) <= 4 Then sMove = sMove & c
    ElseIf InStr("QRNB", c) > 0 Then
      ' Promote piece
      If Len(sMove) = 4 Then sMove = sMove & c
    ElseIf c = " " Or c = Chr$(10) Then
      Exit For
    End If
  Next

  If Len(sMove) = 4 Or Len(sMove) = 5 Then
    ExtractFirstTbMove = sMove
  Else
    ExtractFirstTbMove = ""
  End If
End Function

Public Function ProbeEGTB(ByVal sFEN As String, _
                          ByRef GameResultScore As Long, _
                          ByVal bShowBestMoves As Boolean, _
                          ByRef BestMove As String, _
                          ByRef BestMovesListStr As String) As Boolean
  '
  '--- Use Fathom.exe to access Syzygy Endgame Tabelebases
  '--- Output string is parsed for result and bestmove
  '
  Dim sCommand As String, sRet As String, p As Long, p2 As Long, i As Long, sResult As String, sSearch As String, sOut As String, MoveList() As String, TmpMove As TMOVE, MoveCnt As Long, DTZ As Long
  GameResultScore = VALUE_NONE: BestMove = "": BestMovesListStr = "": ProbeEGTB = False:  EGTBMoveListCnt(Ply) = 0: DTZ = 0
  On Error GoTo lblErr
  '
  '--- Call Fathom.exe and return output
  '
  sCommand = psEnginePath & "\Fathom.exe --path=" & Chr$(34) & EGTBasesPath & Chr$(34) & " " & Chr$(34) & sFEN & Chr$(34)
  sOut = GetCommandOutput(sCommand)
  If Trim$(sOut) = "" Then Exit Function
  sOut = Replace(sOut, Chr$(34), "") 'Remove "
  ' search for DTZ (distance to zero for fifty counter): [DTZ 11]
  sRet = Trim$(sOut)
  p = InStr(sRet, "[DTZ")
  If p > 0 Then
    sRet = Mid$(sRet, p + Len("[DTZ") + 1)
    p = InStr(sRet, "]"): If p = 0 Then Exit Function
    sRet = Trim$(Left$(sRet, GetMax(p - 1, 0)))
    DTZ = Val("0" & Trim$(sRet))
  End If
  sRet = Trim$(sOut)
  'Debug.Print sOut
  ' search for result: [WDL "Win"]
  p = InStr(sRet, "[WDL "): If p = 0 Then Exit Function
  sRet = Mid$(sRet, p + 5)
  p = InStr(sRet, "]"): If p = 0 Then Exit Function
  sResult = Left$(sRet, p - 1)

  Select Case sResult
    Case "Win"
      sSearch = "[WinningMoves"
      GameResultScore = ScorePawn.EG * 20# - 3 * (Ply + DTZ): ProbeEGTB = True
    Case "Draw", "CursedWin", "BlessedLoss" 'CursedWin/BlessedLoss: 50 move draw avoids loss/win
      sSearch = "[DrawingMoves"
      GameResultScore = 0: ProbeEGTB = True
    Case "Loss"
      sSearch = "[LosingMoves"
      GameResultScore = -(ScorePawn.EG * 20# - 3 * (Ply + DTZ)): ProbeEGTB = True
    Case Else
      sSearch = "????"
      Exit Function
  End Select

  EGTBasesHitsCnt = EGTBasesHitsCnt + 1
  ' search for moves: [WinningMoves "Rexd1, Re6, Rdxd1, Rc3"]
  p = InStr(sRet, sSearch): If p = 0 Then Exit Function
  sRet = Mid$(sRet, p + Len(sSearch) + 1)
  p = InStr(sRet, "]"): If p = 0 Then Exit Function
  sRet = Trim$(Left$(sRet, GetMax(p - 1, 0)))
  Dim s As String, CaptureVal As Long, BestCaptureVal As Long, tmp As String
  If sRet <> "" Then
    ' Convert best move to internal move (Rexd1  => e1d1), generate moves and find matching move
    MoveList = Split(sRet, " ")
    CaptureVal = -99999

    For i = 0 To UBound(MoveList())
      s = Trim$(MoveList(i))
      If s <> "" And InStr(s, ".") = 0 Then ' ignore move cnt '1. '
        If InStr(s, "-") = 0 Then ' ignore result '1-0'
          EGTBMoveListCnt(Ply) = EGTBMoveListCnt(Ply) + 1
          EGTBMoveList(Ply, EGTBMoveListCnt(Ply)) = CompToCoord(GetMoveFromSAN(s))
          If EGTBMoveListCnt(Ply) = 1 Then
            BestMove = EGTBMoveList(Ply, 1)
            'Debug.Print MoveText(BestMove)
          End If
          tmp = EGTBMoveList(Ply, EGTBMoveListCnt(Ply))
          TmpMove = TextToMove(tmp)
          If InStr(s, "x") > 0 Or Len(tmp) = 5 Then ' prefer captures/promotions
            If Len(tmp) = 5 Then
              CaptureVal = PieceAbsValue(TmpMove.Promoted) - PieceAbsValue(
              TmpMove.Piece) ' promotion
            Else
              CaptureVal = GetSEE(TmpMove)   ' try best capture
            End If
          Else
            CaptureVal = (PsqVal(1, TmpMove.Piece, TmpMove.Target) - PsqVal(1,
            TmpMove.Piece, TmpMove.From))
          End If
          If CaptureVal > BestCaptureVal Then
            BestCaptureVal = CaptureVal
            BestMove = EGTBMoveList(Ply, EGTBMoveListCnt(Ply))
          End If
          'Debug.Print MoveCnt & ">:" & s
        End If
      End If
    Next

    ' If sResult = "Loss" Then ' do not return move filter
    '  EGTBMoveListCnt = 0
    ' End If
  End If
  ' Find first move of best line " 1. d8=Q Kg4 2. Ke6 Kf4
  If bShowBestMoves Then
    BestMovesListStr = Mid$(sOut, InStrRev(sOut, "]") + 5)    ' find last ] from  [LosingMoves..]
  End If
  sRet = Trim$(Replace(BestMovesListStr, "...", ".")) & " "  ' black to move : "1..."
  MoveCnt = 0
  MoveList = Split(sRet, " ")

  For i = 0 To UBound(MoveList())
    s = Trim$(MoveList(i))
    If s <> "" And InStr(s, ".") = 0 Then ' ignore move cnt '1. '
      If InStr(s, "-") = 0 Then ' ignore result '1-0'
        MoveCnt = MoveCnt + 1
        ' If MoveCnt = 1 Then
        '   BestMove = CompToCoord(GetMoveFromSAN(s))
        'Debug.Print MoveText(BestMove)
        ' End If
        'Debug.Print MoveCnt & ">:" & s
      End If
    End If
  Next

  'If MoveCnt > 0 Then
  ' Select Case sResult
  ' Case "Win"
  '   If BestCaptureVal > 150 Then MoveCnt = MoveCnt \ 2
  '   GameResultScore = ScorePawn.EG * 20# - 3 * MoveCnt
  ' Case "Loss"
  '   If BestCaptureVal > 150 Then MoveCnt = MoveCnt + 200 ' prefer good captures
  '   GameResultScore = -(ScorePawn.EG * 20# - 6 * MoveCnt)
  ' Case Else
  '   ' keep 0
  ' End Select
  'End If
lblExit:
  Exit Function
lblErr:
  ProbeEGTB = False
  Resume lblExit
End Function

Public Function CalcNodes() As Long
  Dim TotalNodes As Double
  If NoOfThreads > 1 Then TotalNodes = CDbl(NoOfThreads) * CDbl(Nodes) Else TotalNodes
   = Nodes
  If TotalNodes > 2147483647# Then CalcNodes = 9999999 Else CalcNodes = TotalNodes
End Function

Public Function CalcNPS(ByVal ElapsedTime As Single) As Long
  Dim TotalNodes As Double
  If NoOfThreads > 1 Then TotalNodes = CDbl(NoOfThreads) * CDbl(Nodes) Else TotalNodes
   = Nodes
  CalcNPS = CDbl(TotalNodes) / GetMaxSingle(0.01, ElapsedTime)
```

```
End Function

Public Function ScaleScoreByEGTB(Score As Long) As Long
  'If Ply > 1 Then Stop
  If EGTBRootResultScore = VALUE_NONE Or Abs(Score) > MATE_IN_MAX_PLY Or Ply > 1 Then
    ScaleScoreByEGTB = Score
  ElseIf EGTBRootResultScore > 0 Then
    ScaleScoreByEGTB = ScorePawn.EG * 20 + Score
  ElseIf EGTBRootResultScore < 0 Then
    ScaleScoreByEGTB = -ScorePawn.EG * 20 + Abs(Score)
  ElseIf EGTBRootResultScore = 0 Then
    ScaleScoreByEGTB = Score \ 10
  End If
End Function

Public Function UciGUIScore(ByVal UciScore As Long, ByVal Alpha As Long, ByVal Beta As Long) As String
     If UciScore <= -MATE_IN_MAX_PLY Then
        UciGUIScore = "mate -" & CStr((MATE0 - Abs(UciScore)) \ 2)
      ElseIf UciScore >= MATE_IN_MAX_PLY Then
        UciGUIScore = "mate " & CStr((MATE0 - UciScore) \ 2)
      Else
        UciGUIScore = "cp " & EvalSFTo100(UciScore)
        If UciScore <= Alpha Then
          UciGUIScore = UciGUIScore & " upperbound"
        ElseIf UciScore >= Beta Then
          UciGUIScore = UciGUIScore & " lowerbound"
        End If
      End If
End Function

Public Function TestEGTB() As String

' see: https://github.com/lichess-org/lila-tablebase

'Public Function TestEGTB(GameResult As enumEndOfGame) As String
' --- curl http://tablebase.lichess.ovh/standard/mainline?fen=4k3/6KP/8/8/6r1/8/7p/8_w_-_-
'curl http://tablebase.lichess.ovh/standard/mainline?fen=4k3/8/8/8/8/4K3/8/8_w_-_-
'{"mainline":[],"winner":null,"dtz":0,"precise_dtz":0}
Dim sInp As String, i As Long, sWinner As String, sCommand As String
Dim sTBMoves As String
Dim EGTBArr() As String
Dim GameResult As enumEndOfGame


TestEGTB = ""

sCommand = "curl http://tablebase.lichess.ovh/standard/mainline?fen=4k3/P7/8/8/8/4K3/p7/8_w_-_-"
sInp = GetCommandOutput(sCommand)


If sInp = "too many pieces" Then GameResult = NO_MATE: Exit Function

'sInp = "{""mainline"":[{""uci"":""g7h8"",""san"":""Kh8"",""dtz"":3,""precise_dtz"":3},{""uci"":""g4a4"",""san"":""Ra4"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""h8g7"",""san"":""Kg7"",""dtz"":1,""precise_dtz"":1},{""uci"":""h2h1q"",""san"":""h1=Q"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""g7f6"",""san"":""Kf6"",""dtz"":1,""precise_dtz"":1},{""uci"":""h1h7"",""san"":""Qxh7"",""dtz"":-5,""precise_dtz"":-5},{""uci"":""f6e5"",""san"":""Ke5"",""dtz"":4,""precise_dtz"":4},{""uci"":""h7g6"",""san"":""Qg6"",""dtz"":-3,""precise_dtz"":-3},{""uci"":""e5d5"",""san"":""Kd5"",""dtz"":2,""precise_dtz"":2},{""uci"":""g6d6"",""san"":""Qd6+"",""dtz"":-1,""precise_dtz"":-1},{""uci"":""d5d6"",""san"":""Kxd6"",""dtz"":21,""precise_dtz"":21},{""uci"":""a4a5"",""san"":""Ra5"",""dtz"":-20,""precise_dtz"":-20},{""uci"":""d6e6"",""san"":""Ke6"",""dtz"":19,""precise_dtz"":19},{""uci"":""a5h5"",""san"":""Rh5"",""dtz"":-18,""precise_dtz"":-18},{""uci"":""e6d6"",""san"":""Kd6"" "
'sInp = sInp & " ""winner"":""b"",""dtz"":-4,""precise_dtz"":-4}"
'sInp = """mainline"":[],""winner"":null,""dtz"":0,""precise_dtz"":0}"



EGTBArr() = Split(sInp, "uci"":""")
```

```
sTBMoves = ""
For i = 1 To UBound(EGTBArr)
  sTBMoves = sTBMoves & Left$(EGTBArr(i), 4) & " "
Next i
sTBMoves = Trim(sTBMoves)

i = InStr(sInp, "winner")
sWinner = ""
If i > 0 Then sWinner = Mid$(sInp, i + 9, 1) ' w =white, b =black, u =Null(Draw)
Select Case sWinner
Case "w"
  GameResult = WHITE_WON
Case "b"
  GameResult = BLACK_WON
Case "u"
  GameResult = DRAW_RESULT
Case Else
  GameResult = NO_MATE
End Select

' Public Enum enumEndOfGame ' Game result
'  NO_MATE = 0
'  WHITE_WON = 1
'  BLACK_WON = 2
'  DRAW_RESULT = 3
'  DRAW3REP_RESULT = 4
' End Enum

Debug.Print sTBMoves, sWinner
End Function

Public Function ProbeOnlineEGTB(ByVal sFEN As String, _
                                ByRef GameResultScore As Long, _
                                ByRef BestMove As String, _
                                ByRef BestMovesList As String) As Boolean
  ' Online Web Access needed ! Uses Windows program curl.exe (comes with Windows)
  ' Documentation: see: https://github.com/lichess-org/lila-tablebase
  ' sample call : curl.exe http://tablebase.lichess.ovh/standard/mainline?fen=4k3/6KP/8/8/6r1/8/7p/8_w_-_-
  ' function returns false if no result
  ' sampel string returned:
  ' 'sResult =
  "{""mainline"":[{""uci"":""g7h8"",""san"":""Kh8"",""dtz"":3,""precise_dtz"":3},{""uci"":""g4a4"",""san"":""Ra4"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""h8g7"",""san"":""Kg7"",""dtz"":1,""precise_dtz"":1},{""uci"":""h2h1q"",""san"":""h1=Q"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""g7f6"",""san"":""Kf6"",""dtz"":1,""precise_dtz"":1},{""uci"":""h1h7"",""san"":""Qxh7"",""dtz"":-5,""precise_dtz"":-5},{""uci"":""f6e5"",""san"":""Ke5"",""dtz"":4,""precise_dtz"":4},{""uci"":""h7g6"",""san"":""Qg6"",""dtz"":-3,""precise_dtz"":-3},{""uci"":""e5d5"",""san"":""Kd5"",""dtz"":2,""precise_dtz"":2},{""uci"":""g6d6"",""san"":""Qd6+"",""dtz"":-1,""precise_dtz"":-1},{""uci"":""d5d6"",""san"":""Kxd6"",""dtz"":21,""precise_dtz"":21},{""uci"":""a4a5"",""san"":""Ra5"",""dtz"":-20,""precise_dtz"":-20},{""uci"":""d6e6"",""san"":""Ke6"",""dtz"":19,""precise_dtz"":19},{""uci"":""a5h5"",""san"":""Rh5"",""dtz"":-18,""precise_dtz"":-18},{""uci"":""e6d6"",""san"":""Kd6"" "
  ' 'sResult = sResult & " ""winner"":""b"",""dtz"":-4,""precise_dtz"":-4}"
  '  "sInp = """mainline"":[],""winner"":null,""dtz"":0,""precise_dtz"":0}"
  ' Test FEN/EPD: 8/8/1P6/5pr1/8/4R3/7k/2K5 w - -

  Static bInitDone As Boolean
  Static bInitOk   As Boolean
  Dim sResult      As String, sCommand As String
  Dim i As Long, sWinner As String
  Dim lDTM As Long ' Distance to mate
  Dim sTBMove As String, bMate As Boolean
  Dim GameResult As enumEndOfGame


  GameResultScore = VALUE_NONE: BestMove = "": BestMovesList = "": GameResult =
  NO_MATE: ProbeOnlineEGTB = False
  If Not bInitDone Then
    bInitOk = InitTableBases()
    bInitDone = True
  End If
```

```
    If Not bInitOk Then ProbeOnlineEGTB = False: Exit Function
    On Error GoTo lblErr
    ' The score is given as distance to mat, or 0 when the position is a draw.
    ' An error response is returned when position is invalid or not in database. '
    ' e.g.  M5 = color to move gives mate in 5 , -M3 = color to move gets mated in 5 moves.

    sCommand = "curl http://tablebase.lichess.ovh/standard?fen=" & Replace(sFEN, " ", "_")
    sResult = Trim(GetCommandOutput(sCommand))

  '  sResult = """checkmate"":false,""stalemate"":false,""variant_win"":false,""variant_loss"":false,""insufficient_material"":false,""dtz"":-4,""precise_dtz"":-4,""dtm"":-10,""category"":""loss"",""moves"":[{""uci"":""g7h8"",""san"":""Kh8"",""zeroing"":false,""checkmate"":false,""stalemate"":false,""variant_win"":false,""variant_loss"":false,""insufficient_material"":false,""dtz"":3,""precise_dtz"":3,""dtm"":9,""category"":""win""}"

    'sCommand = "curl http://tablebase.lichess.ovh/standard/mainline?fen=" & Replace(sFEN, " ", "_")
    'sResult = "{""mainline"":[{""uci"":""g7h8"",""san"":""Kh8"",""dtz"":3,""precise_dtz"":3},{""uci"":""g4a4"",""san"":""Ra4"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""h8g7"",""san"":""Kg7"",""dtz"":1,""precise_dtz"":1},{""uci"":""h2h1q"",""san"":""h1=Q"",""dtz"":-2,""precise_dtz"":-2},{""uci"":""g7f6"",""san"":""Kf6"",""dtz"":1,""precise_dtz"":1},{""uci"":""h1h7"",""san"":""Qxh7"",""dtz"":-5,""precise_dtz"":-5},{""uci"":""f6e5"",""san"":""Ke5"",""dtz"":4,""precise_dtz"":4},{""uci"":""h7g6"",""san"":""Qg6"",""dtz"":-3,""precise_dtz"":-3},{""uci"":""e5d5"",""san"":""Kd5"",""dtz"":2,""precise_dtz"":2},{""uci"":""g6d6"",""san"":""Qd6+"",""dtz"":-1,""precise_dtz"":-1},{""uci"":""d5d6"",""san"":""Kxd6"",""dtz"":21,""precise_dtz"":21},{""uci"":""a4a5"",""san"":""Ra5"",""dtz"":-20,""precise_dtz"":-20},{""uci"":""d6e6"",""san"":""Ke6"",""dtz"":19,""precise_dtz"":19},{""uci"":""a5h5"",""san"":""Rh5"",""dtz"":-18,""precise_dtz"":-18},{""uci"":""e6d6"",""san"":""Kd6"" ""winner"":""b"",""dtz"":-4,""precise_dtz"":-4}"


    ' more than 7 pieces?
    If sResult = "" Or sResult = "too many pieces" Then ProbeOnlineEGTB = False: Exit Function

    '--- search for UCI moves in result string
    sResult = Replace(sResult, """", " ")

    bMate = False
    If InStr(Left$(sResult, 90), "checkmate :true") > 0 Then
      lDTM = 0
      bMate = True
    Else
      i = InStr(sResult, "uci :")
      sTBMove = Trim$(Mid$(sResult, i + 6, 5))  ' 5th character for promotion: qrbn
      sResult = Mid$(sResult, i)
      i = InStr(sResult, "dtm :")
      If Mid$(sResult, i + 5, 4) = "null" Then
        lDTM = -1 ' NO_MATE
      Else
        lDTM = Val(Trim$(Mid$(sResult, i + 5, 5)))
      End If
    End If

    If lDTM < 0 Then
      If bWhiteToMove Then
        GameResult = WHITE_WON: GameResultScore = MATE0 - lDTM
      Else
        GameResult = BLACK_WON: GameResultScore = -MATE0 + lDTM
      End If
    ElseIf lDTM > 0 Then
       If Not bWhiteToMove Then
        GameResult = WHITE_WON: GameResultScore = MATE0 - Abs(lDTM)
      Else
        GameResult = BLACK_WON: GameResultScore = -MATE0 + Abs(lDTM)
      End If
    ElseIf lDTM = 0 Then
      If bMate Then ' Mate
        If bWhiteToMove Then
          GameResult = WHITE_WON: GameResultScore = MATE0
        Else
          GameResult = BLACK_WON: GameResultScore = -MATE0
        End If
      Else 'draw
        GameResult = DRAW_RESULT: GameResultScore = 0
      End If
    Else
      GameResult = NO_MATE
      GameResultScore = VALUE_NONE
    End If

    ' Shows list of best move
    If GameResult <> NO_MATE Then
      ProbeOnlineEGTB = True
      BestMove = Trim$(Left$(sTBMove, 5))
      BestMovesList = BestMove
    End If

    If bEGTbBaseTrace Then WriteTrace "endgame tablebase move: " & BestMove & " / Score: " & GameResultScore & " " & Now() & vbCrLf & PrintPos()
  lblExit:
    Exit Function
  lblErr:
    bInitDone = False
    ProbeOnlineEGTB = False
    Resume lblExit
  End Function








  VERSION 5.00
  Begin VB.Form frmMain
     BorderStyle     =   3  'Fixed Dialog
     Caption         =   "ChessBrain VB"
     ClientHeight    =   3435
     ClientLeft      =   2580
     ClientTop       =   1920
     ClientWidth     =   5250
     Icon            =   "Main.frx":0000
     LinkTopic       =   "Form1"
     MaxButton       =   0   'False
     MinButton       =   0   'False
     ScaleHeight     =   3435
     ScaleWidth      =   5250
     StartUpPosition =   2  'CenterScreen
     Begin VB.Label lblDescr
        AutoSize        =   -1  'True
        BackStyle       =   0  'Transparent
        Caption         =   "In option General->Commandline parameters please add -xboard"
        BeginProperty Font
           Name            =   "MS Sans Serif"
           Size            =   9.75
           Charset         =   0
           Weight          =   400
           Underline       =   0   'False
           Italic          =   0   'False
           Strikethrough   =   0   'False
        EndProperty
        Height          =   945
        Index           =   2
        Left            =   1800
        TabIndex        =   6
```

```
         ToolTipText     =   "GNU General Public License"
         Top             =   1920
         UseMnemonic     =   0   'False
         Width           =   2850
         WordWrap        =   -1  'True
      End
      Begin VB.Label lblDescr 
         Alignment       =   1  'Right Justify
         AutoSize        =   -1  'True
         BackStyle       =   0  'Transparent
         Caption         =   "based on engines: LarsenVB (by Luca Dormio) and Faile (by Adrien M. Regimbald) / Stockfish"
         Height          =   390
         Index           =   4
         Left            =   795
         TabIndex        =   5
         ToolTipText     =   "GNU General Public License"
         Top             =   840
         UseMnemonic     =   0   'False
         Width           =   3525
         WordWrap        =   -1  'True
      End
      Begin VB.Label lblDescr 
         BackStyle       =   0  'Transparent
         Caption         =   "ChessBrainVB 4.00 by Roger Zuehlsdorf 2023"
         BeginProperty Font 
            Name            =   "MS Sans Serif"
            Size            =   13.5
            Charset         =   0
            Weight          =   400
            Underline       =   0   'False
            Italic          =   0   'False
            Strikethrough   =   0   'False
         EndProperty
         Height          =   855
         Index           =   0
         Left            =   960
         TabIndex        =   4
         Top             =   120
         UseMnemonic     =   0   'False
         Width           =   3405
      End
      Begin VB.Label lblDescr 
         AutoSize        =   -1  'True
         BackStyle       =   0  'Transparent
         Caption         =   "Please use a winboard chess GUI (i.e. ARENA)"
         BeginProperty Font 
            Name            =   "MS Sans Serif"
            Size            =   9.75
            Charset         =   0
            Weight          =   400
            Underline       =   0   'False
            Italic          =   0   'False
            Strikethrough   =   0   'False
         EndProperty
         Height          =   480
         Index           =   3
         Left            =   1692
         TabIndex        =   3
         ToolTipText     =   "GNU General Public License"
         Top             =   1332
         UseMnemonic     =   0   'False
         Width           =   2256
         WordWrap        =   -1  'True
      End
      Begin VB.Label lblCmd 
         AutoSize        =   -1  'True
         BackStyle       =   0  'Transparent
         Caption         =   "Quit"
         BeginProperty Font
            Name            =   "MS Sans Serif"
            Size            =   12
            Charset         =   0
            Weight          =   400
            Underline       =   0   'False
            Italic          =   0   'False
            Strikethrough   =   0   'False
         EndProperty
         Height          =   300
         Index           =   2
         Left            =   252
         MousePointer    =   99  'Custom
         TabIndex        =   2
         Top             =   2700
         Width           =   432
      End
      Begin VB.Label lblCmd
         AutoSize        =   -1  'True
         BackStyle       =   0  'Transparent
         Caption         =   "Play game  "
         BeginProperty Font
            Name            =   "MS Sans Serif"
            Size            =   12
            Charset         =   0
            Weight          =   400
            Underline       =   0   'False
            Italic          =   0   'False
            Strikethrough   =   0   'False
         EndProperty
         Height          =   300
         Index           =   0
         Left            =   216
         MousePointer    =   99  'Custom
         TabIndex        =   1
         Top             =   1332
         Width           =   1224
      End
      Begin VB.Label lblDescr
         Alignment       =   1  'Right Justify
         AutoSize        =   -1  'True
         BackStyle       =   0  'Transparent
         Caption         =   "Copyright: GNU GENERAL PUBLIC LICENSE V3"
         Height          =   330
         Index           =   1
         Left            =   1440
         TabIndex        =   0
         ToolTipText     =   "GNU General Public License"
         Top             =   3120
         UseMnemonic     =   0   'False
         Width           =   3600
      End
      Begin VB.Image imgIco
         Height          =   480
         Left            =   105
         Top             =   105
         Width           =   480
      End
      Begin VB.Image imgPointer
         Height          =   480
         Left            =   735
         Picture         =   "Main.frx":0442
         Top             =   105
         Visible         =   0   'False
         Width           =   480
      End
   End
```

```vb
Attribute VB_Name = "frmMain"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
'================================================
'= frmMain:
'= Main form ( not shown under winboard)
'================================================
Option Explicit
Private sWBPath As String    'path winboard.exe

Private Function BrowseForFolders() As String
  BrowseForFolders = InputBox("Enter path of Winboard.exe (or edit INI file):")
  ' WinAPI removed to avoid problems with missing reference
  '###WIN32 sTitle = StrConv("Select location of winboard.exe (or use ARENA GUI) :", vbFromUnicode)
  '###WIN32 BInfo.hwndOwner = Me.hWnd
  '###WIN32 BInfo.lpszTitle = StrPtr(sTitle)
  '###WIN32 BInfo.ulFlags = BIF_RETURNONLYFSDIRS
  '###WIN32 lpIdList = SHBrowseForFolder(BInfo)
  '###WIN32 If lpIdList Then
  '###WIN32    sFolderName = String$(260, 0)
  '###WIN32    SHGetPathFromIDList lpIdList, sFolderName
  '###WIN32    sFolderName = Left$(sFolderName, InStr(sFolderName, Chr(0)) - 1)
  '###WIN32    CoTaskMemFree lpIdList
  '###WIN32 End If
  '###WIN32 BrowseForFolders = sFolderName
End Function

'--------------------------------------------------------------------------
'GetCmdLine() - pass command line to ChessBrainVB
'
'--------------------------------------------------------------------------
Private Function GetCmdLine() As String
 ' GetCmdLine = " -cp -fcp ""ChessBrainVB -xboard"" -fd """ & psEnginePath & """  -scp ""ChessBrainVB -xboard""
 -sd """ & psEnginePath & """"
End Function

Private Sub SetWBPath()
  sWBPath = ReadINISetting("WINBOARD", "")
  If sWBPath = "" Then
    sWBPath = BrowseForFolders
  Else
    On Local Error Resume Next
    If Dir$(sWBPath & "\winboard.exe") = "" Then
      sWBPath = BrowseForFolders
    End If
    On Local Error GoTo 0
  End If
End Sub

Private Sub Form_Load()
  Dim i As Long
  imgIco.Picture = Me.Icon
  Set Me.Icon = Nothing

  With App
    Me.Caption = Me.Caption & "   ver. " & .Major & "." & Format(.Minor, "00") & "." &
     Format(.Revision, "0000")
    'lblDescr(1) = .LegalCopyright
  End With

  For i = 0 To lblCmd.UBound
    lblCmd(i).MouseIcon = imgPointer.Picture
  Next

  'lblDescr(0) = LoadResString(resMainTitle)
  'lblDescr(2) = LoadResString(resMainOption)
```

```
    'lblCmd(0) = LoadResString(resMainPlay)
    'lblCmd(1) = LoadResString(resMainBookEd)
    'lblCmd(2) = LoadResString(resMainQuit)
  End Sub

  Private Sub Form_MouseMove(Button As Integer, Shift As Integer, x As Single, y As
  Single)
    Dim i As Long

    For i = 0 To lblCmd.UBound
      lblCmd(i).Font.Underline = False
      lblCmd(i).Font.Bold = False
    Next

  End Sub

  Private Sub lblCmd_Click(Index As Integer)
    Dim sBookName As String

    Select Case Index
      Case 0        'Winboard
        SetWBPath
        On Local Error GoTo CmdError:
        Shell sWBPath & "\winboard.exe" & GetCmdLine, vbNormalFocus
        WriteINISetting "WINBOARD", sWBPath
        End

      Case 2        'Quit
        End
    End Select

    On Local Error GoTo 0
    Exit Sub
  CmdError:

    Select Case Err.Number
      Case 53

        Select Case Index
          Case 0
            MsgBox "Cannot find Winboard", vbCritical
          Case 1
            MsgBox "Cannot find  BookEdit", vbCritical
        End Select
    End Select

    Screen.MousePointer = vbDefault
  End Sub

  Private Sub lblCmd_MouseMove(Index As Integer, _
                               Button As Integer, _
                               Shift As Integer, _
                               x As Single, _
                               y As Single)
    Dim i                As Long
    Static LastIndex As Long
    If Index <> LastIndex Then

      For i = 0 To lblCmd.UBound
        lblCmd(i).Font.Underline = False
        lblCmd(i).Font.Bold = False
      Next

      LastIndex = Index
    End If
    lblCmd(Index).Font.Underline = True
    lblCmd(Index).Font.Bold = True
  End Sub
```

```
Attribute VB_Name = "basProcess"
'======================
'= basProcess:
'= start processes for multi core case
'======================

Option Explicit

Private Type STARTUPINFO
 cb As Long
 lpReserved As String
 lpDesktop As String
 lpTitle As String
 dwX As Long
 dwY As Long
 dwXSize As Long
 dwYSize As Long
 dwXCountChars As Long
 dwYCountChars As Long
 dwFillAttribute As Long
 dwFlags As Long
 wShowWindow As Integer
 cbReserved2 As Integer
 lpReserved2 As Long
 hStdInput As Long
 hStdOutput As Long
 hStdError As Long
End Type

Private Type PROCESS_INFORMATION
 hProcess As Long
 hThread As Long
 dwProcessID As Long
 dwThreadID As Long
End Type

Const STARTF_USESHOWWINDOW = &H1&
Const NORMAL_PRIORITY_CLASS = &H20&
Const SW_HIDE = 3

Private Declare Function CreateProcess Lib "kernel32" Alias "CreateProcessA" (ByVal lpApplicationName As String, ByVal lpCommandLine As String, lpProcessAttributes As Any, lpThreadAttributes As Any, ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long, lpEnvironment As Any, ByVal lpCurrentDriectory As String, lpStartupInfo As STARTUPINFO, lpProcessInformation As PROCESS_INFORMATION) As Long
Private Declare Function WaitForSingleObject Lib "kernel32" (ByVal hHandle As Long, ByVal dwMilliseconds As Long) As Long
Private Declare Function GetExitCodeProcess Lib "kernel32" (ByVal hProcess As Long, lpExitCode As Long) As Long
Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long

'--- Shells the passed command line and waits for the process to finish
'--- Returns the exit code of the shelled process
Function StartProcess(strCmdLine As String) As Long
  Dim udtProc As PROCESS_INFORMATION, udtStart As STARTUPINFO

  'initialize the STARTUPINFO structure
  udtStart.cb = Len(udtStart) 'size
  udtStart.dwFlags = STARTF_USESHOWWINDOW 'uses show window command
  udtStart.wShowWindow = SW_HIDE 'the hide window command

  'Launch the application
  CreateProcess vbNullString, strCmdLine, ByVal 0&, ByVal 0&, 0, NORMAL_PRIORITY_CLASS, ByVal 0&, vbNullString, udtStart, udtProc

End Function


```

```
Attribute VB_Name = "basSearch"
 Option Explicit
'=============================================================================================
'= basSearch:                                                    =
'=                                                            =
'= Search functions: Think->SearchRoot->Search->QSearch>Eval
'= Think.....: Init search and call "SearchRoot" with increasing iterative depth 1,2,3... until time is over
'= SearchRoot: create root moves at ply 1 and call "Search" starting with ply 2
'= Search....: search for best move by recursive calls to itself down to iterative depth or time is over
'=           when iterative depth reached, calls QSearch
'= QSearch...: quiescence search calculates all captures and checks (first QS-ply only) by recursive calls to itself
'=           When all captures are done, the final position evaluation is returned
'=============================================================================================

Public Result                                        As enumEndOfGame ' game result win/draw
Public RootDepth                                     As Long ' start search depth of root
Public Nodes                                         As Long ' counter for calls of SEARCH function
Public QNodes                                        As Long ' counter for calls of QSSEARCH function
Public QSDepthMax                                    As Long ' max QS search depth reached
Public EvalCnt                                       As Long ' counter for calls of EVAL function
Public RootDelta                                     As Long ' delta of alpha beta at root
Public bEndgame                                      As Boolean ' switch for endgame logic
Public PlyScore(MAX_DEPTH)                           As Long ' score for current search ply
Public MaxPly                                        As Long ' may ply reached in Search
Public PV(MAX_PV, MAX_PV)                            As TMOVE '--- principal variation(PV): best path of moves in current search tree
Public LastFullPVArr(MAX_PV)                         As TMOVE ' list of moves in search
Public LastFullPVLen                                 As Long
Public PVLength(MAX_PV)                              As Long
Private bSearchingPV                                 As Boolean '--- often used for special handling (more exact search)
Public HintMove                                      As TMOVE ' user hint move for GUI
Public MovesList(MAX_PV)                             As TMOVE '--- currently searched move path
Public CntRootMoves                                  As Long ' number of root moves: zero = draw
Public PliesFromNull                                 As Long '--- number of moves since last null move : for 3x draw detection
Public FinalMove                                     As TMOVE, FinalScore As Long '--- Final move selected
Public PieceCntRoot                                  As Long ' number of pieces on board at root
Private bOnlyMove                                    As Boolean  ' direct response if only one move
Private RootStartScore                               As Long ' Eval score at root from view of side to move
Public PrevGameMoveScore                             As Long ' Eval score at root from view of side to move
Private RootMatScore                                 As Long ' Material score at root from view of side to move
Public RootMoveCnt                                   As Long ' current root move for GUI
Public LastFinalScore                                As Long ' Final move score
Public bFailedLowAtRoot                              As Boolean ' bad root move > needs more time
Public DoubleExtensions(MAX_PV)                      As Long ' counts search extensions to avoid search explosion
Public CutOffCnt(MAX_PV)                             As Long ' cutoff
Public ttPVArr(MAX_PV)                               As Boolean

'--- Search performance: move ordering, cuts of search tree ---
```

```
Public History(COL_WHITE, MAX_BOARD, MAX_BOARD)          As Long  ' move history From square -> To square for color
Public CaptureHistory(BEP_PIECE, MAX_BOARD, BEP_PIECE)   As Long  ' capture history moving piece -> To square > captured Piece type
Public StatScore(MAX_PV + 3)                             As Long  ' statistics score per search ply
Public CounterMove(15, MAX_BOARD)                        As TMOVE ' Good move against previous move
Public ContinuationHistory(15 * MAX_BOARD, 15 * MAX_BOARD) As Integer  ' statistics for follow up moves ; Integer for less memory
Public CmhPtr(MAX_PV)                                    As Long ' Pointer to first move of ContinuationHistory

Public Type TKiller
  Killer1             As TMOVE 'killer moves: good moves for better move ordering
  Killer2             As TMOVE
  Killer3             As TMOVE
End Type

Public Killer(MAX_PV)                  As TKiller
Public Killer0                         As TKiller
Public Killer2                         As TKiller
Public Reductions(63)                   As Long ' [moveNumber]
Public BestMovePly(MAX_PV)             As TMOVE
Public EmptyMove                       As TMOVE
Public CaptPruneMargin(6)              As Long

'--- piece bit constants for attack arrays, used for evaluation
Public Const PLAttackBit As Long = 1 ' Pawn attack to left side (from white view)
Public Const PRAttackBit As Long = 2 ' Pawn attack to right side (from white view) (to count multiple pawn attacks)
Public Const N1AttackBit As Long = 4 ' for 1. knight
Public Const N2AttackBit As Long = 8 ' for 2. knight
Public Const B1AttackBit As Long = 16
Public Const B2AttackBit As Long = 32
Public Const R1AttackBit As Long = 64
Public Const R2AttackBit As Long = 128
Public Const QAttackBit As Long = 256
Public Const KAttackBit As Long = 512
Public Const BXrayAttackBit As Long = 1024 ' Xray attack through own bishop/queen, one xray enough because different square colors
Public Const R1XrayAttackBit As Long = 2048 ' Xray attack through own rook/queen
Public Const R2XrayAttackBit As Long = 4096 ' to count multiple rook attacks, not needed for bishop and queens (promotion needed)
Public Const QXrayAttackBit As Long = 8192 ' Xray attack through own bishop/rook/queen
'--- combined attack bits
Public Const PAttackBit As Long = PLAttackBit Or PRAttackBit
Public Const NAttackBit As Long = N1AttackBit Or N2AttackBit
Public Const BAttackBit As Long = B1AttackBit Or B2AttackBit
Public Const BOrXrayAttackBit As Long = B1AttackBit Or B2AttackBit Or BXrayAttackBit
Public Const RAttackBit As Long = R1AttackBit Or R2AttackBit
Public Const R1OrXrayAttackBit As Long = R1AttackBit Or R1XrayAttackBit
Public Const R2OrXrayAttackBit As Long = R2AttackBit Or R2XrayAttackBit
Public Const ROrXrayAttackBit As Long = R1AttackBit Or R2AttackBit Or R1XrayAttackBit Or R2XrayAttackBit
Public Const PBNAttackBit As Long = PAttackBit Or NAttackBit Or BAttackBit
Public Const RBAttackBit As Long = RAttackBit Or BAttackBit
Public Const RBOrXrayAttackBit As Long = ROrXrayAttackBit Or BOrXrayAttackBit
Public Const QOrXrayAttackBit As Long = QAttackBit Or QXrayAttackBit
Public Const QOrXrayROrXrayAttackBit As Long = QOrXrayAttackBit Or ROrXrayAttackBit
Public Const QBAttackBit As Long = QAttackBit Or BAttackBit
Public Const QRAttackBit As Long = QAttackBit Or RAttackBit
Public Const QRBAttackBit As Long = QAttackBit Or RAttackBit Or BAttackBit    ' slider attacks, detect pinned pieces
Public Const QRBOrXrayAttackBit As Long = QAttackBit Or QXrayAttackBit Or ROrXrayAttackBit Or BOrXrayAttackBit     ' slider attacks, detect pinned pieces
Public Const QRBNAttackBit As Long = QAttackBit Or RAttackBit Or BAttackBit Or NAttackBit
Public Const PNBRAttackBit As Long = PAttackBit Or NAttackBit Or BAttackBit Or
RAttackBit
'----
Public AttackBitCnt(QXrayAttackBit * 2)      As Long     ' Returns number of attack bits set
Public EasyMove                       As TMOVE
Public EasyMovePV(3)                  As TMOVE
Public EasyMoveStableCnt              As Long
Public bEasyMovePlayed                As Boolean
Public QSDepth                        As Long
Private TmpMove                       As TMOVE
Public bFirstRootMove                 As Boolean
'Public bEvalBench              As Boolean
Public LegalRootMovesOutOfCheck As Long
Public IsTBScore                      As Boolean
Public SkipSize(20)                   As Long 'multi core search: sizes and phases of the skip-blocks, used
for distributing search depths across the threads
Public SkipPhase(20)                  As Long
Public DepthInWork                    As Long ' multi core search: For decision if better thread

Public FinalCompletedDepth            As Long ' root depth completed
Private NullMovePly                   As Long ' search depth for null move verification

Public TableBasesRootEnabled          As Boolean
Public TableBasesSearchEnabled        As Boolean

'--- end if declarations -----------------------------------------------------------------------------------
'-------------------------------------------------------------------------
'StartEngine: starts the chess engine to return a move
'-------------------------------------------------------------------------
Public Sub StartEngine()
  Dim CompMove        As TMOVE
  Dim sCoordMove      As String
  Dim bOldEvalTrace As Boolean
  Dim i               As Long
  '--- in winboard FORCE mode return, also check side to move
  'Debug.Print bCompIsWhite, bWhiteToMove, bForceMode, Result
  If bAnalyzeMode Then bCompIsWhite = bWhiteToMove

  '--- for main loop exit here if opponent has to move
  If bCompIsWhite <> bWhiteToMove Or bForceMode Or Result <> NO_MATE Then Exit Sub

  '--- for single core is ThreadnNum=-1, fot multi core main thread is ThreadNum=0, core 2 is ThreadNum=1
  If NoOfThreads > 1 And ThreadNum = 0 Then
    InitThreads
  End If

  '--- Init Search data
  QNodes = 0
  QSDepthMax = 0
  Nodes = 0
  Ply = 1
  Result = NO_MATE
  TimeStart = Timer
  bOldEvalTrace = bEvalTrace

  ' If DebugMode And ThreadNum = 0 Then
  '   DEBUGReadGame "bug001game.txt"
  '   FixedTime = 30
  ' End If

  '--- Multi core search: init thread data
  If ThreadNum = 0 Then
    If bThreadTrace Then WriteTrace "StartEngine: WriteMainThreadStatus 1 " & " / " &
    Now()
    ClearMapBestPVforThread
    WriteMapGameData
    MainThreadStatus = 1: WriteMainThreadStatus 1 ' start helper threads
  ElseIf ThreadNum > 0 Then
```

```
    ' Read game data for helper thread
    If bThreadTrace Then WriteTrace "StartEngine ReadMapGameData" & " / " & Now()
    ReadMapGameData
    bCompIsWhite = bWhiteToMove
    If bThreadTrace Then WriteTrace "StartEngine gamemoves: " & GameMovesCnt & " / " &
     Now()
    FixedDepth = 80 ' NO_FIXED_DEPTH
    MovesToTC = 0
    TimeLeft = 180000
  End If

  '========================================================================
  '= --- Start search ---
  '========================================================================

  CompMove = Think()   '--- Calculate engine move <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

  If bAnalyzeMode Or bOldEvalTrace Then
    bAnalyzeMode = False
    bCompIsWhite = Not bCompIsWhite
    Exit Sub
  End If

  '--- Set time
  SearchTime = TimeElapsed()
  TimeLeft = (TimeLeft - SearchTime) + TimeIncrement

  '========================================================================
  '--- Check  search result
  '========================================================================
  sCoordMove = CompToCoord(CompMove)
  If sCoordMove = "" And UCIMode Then sCoordMove = "(none)"

  Select Case Result
    Case NO_MATE
      PlayMove CompMove
      GameMovesAdd CompMove
      If UCIMode Then
        SendCommand "bestmove" & " " & sCoordMove
      Else
        SendCommand Translate("move") & " " & sCoordMove
      End If
    Case BLACK_WON
      ' Mate?
      If CompMove.From <> 0 Then
        PlayMove CompMove
        GameMovesAdd CompMove
        If UCIMode Then
          SendCommand "bestmove" & " " & sCoordMove
        Else
          SendCommand Translate("move") & " " & sCoordMove
          SendCommand "0-1 {" & Translate("Black Mates") & "}"
        End If
      Else
        If UCIMode Then
          SendCommand "bestmove (none)" ' ??? try same as Stockfish
        End If
      End If
    Case WHITE_WON
      ' Mate?
      If CompMove.From <> 0 Then
        PlayMove CompMove
        GameMovesAdd CompMove
        If UCIMode Then
          SendCommand "bestmove" & " " & sCoordMove
        Else
          SendCommand Translate("move") & " " & sCoordMove
          SendCommand "1-0 {" & Translate("White Mates") & "}"
        End If
      Else
        If UCIMode Then
          SendCommand "bestmove (none)" ' ??? try same as Stockfish
        End If
      End If
    Case DRAW3REP_RESULT
      ' Draw?
      PlayMove CompMove
      GameMovesAdd CompMove
      If UCIMode Then
        SendCommand "bestmove" & " " & sCoordMove
      Else
        SendCommand Translate("move") & " " & sCoordMove
        SendCommand "1/2-1/2 {" & Translate("Draw by repetition") & "}"
      End If
    Case DRAW_RESULT:
        If UCIMode Then
          SendCommand "bestmove (none)"
        Else
          SendCommand "1/2-1/2 {" & Translate("Draw no move") & "}"
        End If
    Case Else
      '
      '--- Send move to GUI
      '
      If CompMove.From <> 0 Then
        PlayMove CompMove
        GameMovesAdd CompMove
        If UCIMode Then
          SendCommand "bestmove" & " " & sCoordMove
        Else
          SendCommand Translate("move") & " " & sCoordMove
        End If
        '--- Draw?
        If Fifty >= 100 Then
          SendCommand "1/2-1/2 {" & Translate("50 Move Rule") & "}"
        Else '--- no move
          SendCommand "1/2-1/2 {" & Translate("Draw") & "}"
        End If
      End If
  End Select

  'WriteTrace "move: " & CompMove & vbCrLf ' & "(t:" & Format(SearchTime, "###0.00") & " s:" & FinalScore ' & "
  n:" & Nodes & " qn:" & QNodes & " q%:" & ")"
End Sub


'=======================================================
' THINK: Start of Search with iterative deepening      =
'       aspiration windows used in 3 steps           =
'       called by: STARTENGINE, calls: SEARCH         =
'=======================================================
Public Function Think() As TMOVE
  Dim Elapsed              As Single
  Dim CompMove             As TMOVE, LastMove As TMOVE
  Dim IMax                 As Long, i As Long, j As Long, k As Long
  Dim BoardTmp(MAX_BOARD)  As Long
  Dim GoodMoves            As Long
  Dim RootAlpha            As Long
  Dim RootBeta             As Long
  Dim OldScore             As Long, Delta As Long
  Dim bOldEvalTrace        As Boolean
  Dim Hashkey              As THashKey
  Dim AdjustedDepth As Long, FailedHighCnt As Long

  '--- Thread management
  Dim bHelperMove          As Boolean, HelperCompletedDepth As Long, HelperBestScore As
```

```
 Long, HelperNodes As Long, HelperPvLength As Long, HelperPV(11) As TMOVE
'--------------------------------------------
ClearMove CompMove
ResetMaterial
' init counters
Nodes = 0
QNodes = 0
EvalCnt = 0
HashUsage = 0
HashAccessCnt = 0
InitEval
bEvalTrace = bEvalTrace Or CBool(ReadINISetting("EVALTRACE", "0") <> "0") ' after InitEval
bOldEvalTrace = bEvalTrace
MaxPly = 0
EGTBasesHitsCnt = 0
LastNodesCnt = 0: RootMoveCnt = 0: LastThreadCheckNodesCnt = 0
plLastPostNodes = 0: IsTBScore = False
NextHashGeneration ' set next generation for hash entries
LastFullPV = ""
Erase LastFullPVArr: LastFullPVLen = 0
'HashFoundFromOtherThread = 0
FinalCompletedDepth = 0: DepthInWork = 0
' init easy move
EasyMove = GetEasyMove() ' get easy move from previous Think call
If bTimeTrace Then WriteTrace "Think: Easymove: " & MoveText(EasyMove) & " " & Now()
ClearEasyMove
bEasyMovePlayed = False
BestMoveChanges = 0
SetMove FinalMove, EmptyMove

' Tracing
bTimeTrace = CBool(ReadINISetting("TIMETRACE", "0") <> "0")
If bTimeTrace Then
  WriteTrace " "
  WriteTrace "----- Start thinking, GAME MOVE >>>: " & GameMovesCnt \ 2 & " <<<"
ElseIf bLogPV Then
  If bWinboardTrace Then LogWrite Space(6) & "----- Start thinking, GAME MOVE >>>: "
   & GameMovesCnt \ 2 & " <<<"
End If

' reset move lists
For i = 0 To 99: PlyScore(i) = 0: MovesList(i).From = 0: MovesList(i).Target = 0:
Next i

' reset debug counter
For i = 0 To 20: TestCnt(i) = 0:  Next


bTimeExit = False '--- Used for stop search, currently searched line result is not valid!!

'============================
'=  Opening book move ?    =
'============================
If BookMovePossible Then
   CompMove = ChooseBookMove
   If CompMove.From <> 0 Then
     FinalScore = 0
     If UCIMode Then
       SendCommand "info string book move: " & CompToCoord(CompMove)
     Else
       SendCommand "0 0 0 0 (Book Move)"
     End If
     Think = CompMove
     Exit Function '<<<< EXIT with book move
   End If
   BookMovePossible = False
 End If
```

```
'--- Init search scores ---
FinalScore = -MATE0
RootStartScore = Eval()     ' Output for EvalTrace, sets EvalTrace=false
If bOldEvalTrace Then ClearMove Think: Exit Function   ' Exit if we only want an EVAl trace
'LogWrite "Start Think "

'--- Init timer ---
TimeStart = Timer
AllocateTime
'Debug.Print "OptTime=" & OptimalTime & " , MaxTime=" & MaximumTime

'--- init hash map for multi core search
If ThreadNum > 0 Then InitHash ' check new hash size

HashBoard Hashkey, EmptyMove
InHashCnt = 0
IMax = MAX_DEPTH
If bThreadTrace Then WriteTrace "Think: Threadnum=" & ThreadNum & " " & Now() & vbCrLf & " start board= " & vbCrLf & PrintPos
If ThreadNum > 0 Then WriteHelperThreadStatus ThreadNum, 1

' copy current board before start of search to restore it later
CopyIntArr Board, BoardTmp

' - not better ?
'--- Init search data--
''    Erase History()
''    Erase ContinuationHistory()
'--- Rescale history ???? not better, same results with 32, 64, 128
'  For j = SQ_A1 To SQ_H8
'    For k = SQ_A1 To SQ_H8
'      For i = COL_WHITE To COL_BLACK
'        History(i, j, k) = History(i, j, k) \ 32
'      Next
'      ContinuationHistory(i, j) = ContinuationHistory(j, k) \ 32
'    Next
'  Next
'Erase CounterMove()

'==> Keep old data in History arrays!
Erase Killer()
Erase PV()
If ThreadNum > 0 Then WriteMapBestPVforThread 0, VALUE_NONE, EmptyMove
Erase MovesList()
CntRootMoves = 0
LastChangeMove = ""
FinalScore = -VALUE_INFINITE
Result = NO_MATE

EGTBMoveListCnt(1) = 0: EGTBRootResultScore = VALUE_NONE: EGTBRootProbeDone = False

'---------------------------
'--- Iterative deepening ----
'---------------------------
For RootDepth = 1 To IMax

  '--- Distribute search depths across the threads
  If ThreadNum > 0 Then
    Dim th As Long
    th = (ThreadNum - 1) Mod 20
    If ((RootDepth + SkipPhase(th)) / GetMax(1, SkipSize(th))) Mod 2 <> 0 And RootDepth > 1 Then
      If RootDepth > 1 Then PlyScore(RootDepth) = PlyScore(RootDepth - 1)
      GoTo lblNextRootDepth
    Else
      If bThreadTrace Then WriteTrace "Think: RootDepth= " & RootDepth & " / " & Now()
    End If
  End If

  Elapsed = TimeElapsed ' get time

  bResearching = False
  If ThreadNum <= 0 Then ' main thread
    BestMoveChanges = BestMoveChanges * 0.505 ' Age out PV variability metric
    bFailedLowAtRoot = False
  End If

  If Not FixedDepthMode And FixedTime = 0 And Not bAnalyzeMode Then
    If Not CheckTime() And RootDepth > 1 Then
      If bTimeTrace Then WriteTrace "Exit SearchRoot2: Used: " & Format$(Elapsed,
      "0.00") & ", Given:" & Format$(OptimalTime, "0.00")
      Exit For
    End If
  Else
    If RootDepth > FixedDepth Then Exit For ' Fixed depth reached -> Exit
  End If
  If EGTBasesHitsCnt > 0 And RootDepth > 40 Then bTimeExit = True: Exit For
  bSearchingPV = True
  GoodMoves = 0
  PlyScore(RootDepth) = 0
  FailedHighCnt = 0

  '
  '--- Aspiration window between alpha and beta
  '
  RootAlpha = -MATE0: RootBeta = MATE0: Delta = -MATE0
  OldScore = PlyScore(RootDepth - 1)
  If RootDepth >= 4 Then
    Delta = 18 ' aspiration window size / critical value!
    If ThreadNum > 0 Then ' helper threads with different windows
      Delta = 17 + ((ThreadNum + 1) And 3)
    End If
    Debug.Assert Abs(Delta) <= 200000

    RootAlpha = GetMax(OldScore - Delta, -MATE0)
    RootBeta = GetMin(OldScore + Delta, MATE0)

    If OldScore > MATE_IN_MAX_PLY Then
      RootBeta = MATE0
    ElseIf OldScore < -MATE_IN_MAX_PLY Then
      RootAlpha = -MATE0
    End If
  End If

  bFailedLowAtRoot = False
  AdjustedDepth = RootDepth
  Debug.Assert Abs(RootAlpha) <= Abs(VALUE_NONE)

  '=========================================
  '--- Start with a small aspiration window and, in the case of a fail high/low, re-search with a bigger window
  until we don't fail high/low anymore.
  '=========================================
  Do While (True)
    '
    '--------- SEARCH ROOT ----------------
    '
    AdjustedDepth = GetMax(1, RootDepth - FailedHighCnt)
    '=======================================================================
    LastMove = SearchRoot(RootAlpha, RootBeta, AdjustedDepth, GoodMoves) '<<<<<<<<<
    SEARCH ROOT <<<<<<<<<<<<<<<<<<<<
    '=======================================================================

    #If DEBUG_MODE Then
   '  If RootDepth > 5 Then
```

```
       '    SendCommand "D:" & RootDepth & ">>> Search A:" & RootAlpha & ", B:" & RootBeta & " => SC: " & FinalScore
       '  End If
        #End If          '
        Debug.Assert Abs(FinalScore) <= Abs(VALUE_NONE)
        Debug.Assert Abs(RootAlpha) <= Abs(VALUE_NONE)
        Debug.Assert Abs(RootBeta) <= Abs(VALUE_NONE)

        '--LastMove.From = 0  no move => draw
        If bTimeExit Or IsTBScore Or LastMove.From = 0 Or (bOnlyMove And RootDepth = 1 ) Then Exit Do

        '
        '--- Research: if no move found in Alpha-Beta window
        '
        bSearchingPV = True: GoodMoves = 0

        ' GUI info
        If (RootDepth > 1 Or IsTBScore) And bPostMode And PVLength(1) >= 1 Then
          Elapsed = TimeElapsed()
          If Not bExitReceived Then SendThinkInfo Elapsed, RootDepth, FinalScore, RootAlpha, RootBeta ' Output to GUI
        End If

        If FinalScore <= RootAlpha Then '<<< search failed low
          #If DEBUG_MODE Then
            If RootDepth > 5 Then
              SendCommand "             Research " & " SC:" & FinalScore & " <= A:" & RootAlpha
            End If
          #End If
          RootBeta = (RootAlpha + RootBeta) \ 2
          RootAlpha = GetMax(FinalScore - Delta, -MATE0)

          If ThreadNum <= 0 Then FailedHighCnt = 0
          bResearching = True
          If ThreadNum <= 0 Then bFailedLowAtRoot = True

        ElseIf FinalScore >= RootBeta Then '<<<< search failed high
          #If DEBUG_MODE Then
            If RootDepth > 5 Then
              SendCommand "             Research " & " SC:" & FinalScore & "       >= B:" & RootBeta
            End If
          #End If
          If ThreadNum <= 0 Then FailedHighCnt = FailedHighCnt + 1
          RootBeta = GetMin(FinalScore + Delta, MATE0)
          bResearching = True
        Else
          Exit Do '<<< search result in alpha/beta window: finish this search depth
        End If

        ' mate search?
        If FinalScore > 2 * ScoreQueen.EG And FinalScore <> MATE0 Then
          RootBeta = MATE0
        ElseIf FinalScore < -2 * ScoreQueen.EG And FinalScore <> -MATE0 Then
          RootAlpha = -MATE0
        End If

        ' set new delta for research
        Debug.Assert Abs(Delta) <= 200000
        If Abs(Delta) < MATE_IN_MAX_PLY Then Delta = Delta + (Delta \ 4 + 5)
        Debug.Assert Abs(Delta) <= 200000

        DoEvents
      Loop

      '=========================================
      '--- Search result for current iteration ---
      '==========================================

      If (bOnlyMove And RootDepth = 1) Then FinalScore = LastFinalScore Else
      LastFinalScore = FinalScore

      If FinalScore <> VALUE_NONE And FinalScore <> -VALUE_INFINITE Then
        If Not bTimeExit Then
          If FinalMove.From > 0 Then FinalCompletedDepth = AdjustedDepth
        End If
        If ThreadNum > 0 And Trim(MoveText(PV(1, 1))) = "" Then
          If bThreadTrace Then WriteTrace "!!!???Think:PV Empty "
        Else
          If ThreadNum > 0 And PVLength(1) > 1 Then
            WriteMapBestPVforThread FinalCompletedDepth, FinalScore, FinalMove
          Else
            If bThreadTrace Then WriteTrace "Think: else PVLen<2" & PVLength(1)
          End If
        End If
        CompMove = FinalMove
        PlyScore(RootDepth) = FinalScore
        If bPostMode And PVLength(1) >= 1 Then
          Elapsed = TimeElapsed()
          If Not bExitReceived Then SendThinkInfo Elapsed, RootDepth, FinalScore,
          RootAlpha, RootBeta ' Output to GUI
        End If
      End If

      CopyIntArr BoardTmp, Board  ' copy old position to main board / just to be sure

      If bOnlyMove Or IsTBScore Then
        bOnlyMove = False: Exit For
      End If
      If RootDepth > 2 And FinalScore > MATE0 - RootDepth Then
        Exit For
      End If
      If bTimeExit Or IsTBScore Or (RootDepth = 1 And LastMove.From = 0) Then GoTo
      lblIterationsExit

      ' easy move?
      If RootDepth >= 7 - 3 * Abs(pbIsOfficeMode) And EasyMove.From > 0 And Not
      FixedDepthMode And Not FixedTime > 0 Then
        If bTimeTrace Then WriteTrace "Easy check PV (IT:" & RootDepth & "): EM:" &
        MoveText(EasyMove) & ": PV1:" & MoveText(PV(1, 1))
        If MovesEqual(PV(1, 1), EasyMove) Then
          If bTimeTrace Then WriteTrace "Easy check2 bestmove: " & Format(
          BestMoveChanges, "0.000")
          If BestMoveChanges < 0.03 Then
            Elapsed = TimeElapsed()
            If bTimeTrace Then WriteTrace "Easy check3 Elapsed: " & Format$(Elapsed,
            "0.00") & Format$(OptimalTime * 5# / 42#, "0.00")
            If Elapsed > OptimalTime * 5# / 44# Then
                bEasyMovePlayed = True
                bTimeExit = True
                If bTimeTrace Then
                  WriteTrace "Easy move played: " & MoveText(EasyMove) & " Elapsed:" &
                   Format$(Elapsed, "0.00") & ", Opt:" & Format$(OptimalTime, "0.00")
                  & ", Max:" & Format$(MaximumTime, "0.00") & ", Left:" & Format$(
                  TimeLeft, "0.00")
                End If
            End If
          End If
        End If
      End If

      If RootDepth > 15 Then ' emergency exit or mate found?
        If RootDepth > 80 Or (Abs(FinalScore) > MATE0 - 6 And Abs(FinalScore) < MATE0)
         Then bTimeExit = True
```

```
        End If

        If bTimeExit Then
          Exit For
        Else
          If PV(1, 3).From > 0 Then
            UpdateEasyMove
          Else
            If EasyMovePV(3).From > 0 Then ClearEasyMove
          End If
        End If

lblNextRootDepth:
        If ThreadNum > 0 Then If ReadMainThreadStatus() = 0 Then bTimeExit = True: Exit
        For
       ' WriteTrace "HashfromOtherThread: Rootdepth=" & RootDepth & " : " & HashFoundFromOtherThread & " /
       nodes:" & Nodes & " " & Now()

      Next ' search depth <<<<<<<<
      '===================================================================

lblIterationsExit:
      If bThreadTrace Then WriteTrace "Think: finished nodes: " & Nodes & " / " & Now()

      '--- Time management
      Elapsed = TimeElapsed()
      If EasyMoveStableCnt < 6 Or bEasyMovePlayed Then ClearEasyMove
      'LogWrite "End Think " & MoveText(CompMove) & " Result:" & Result
      If FinalScore <> VALUE_NONE Then PrevGameMoveScore = FinalScore Else
      PrevGameMoveScore = 0

      '===========================================================
      Think = CompMove '--- Return move <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      '===========================================================

      '--------------------
      ' Stop Helper Threads
      '--------------------
      If ThreadNum = 0 Then
        If bThreadTrace Then WriteTrace "Think; end think: stop threads" & ThreadNum &
        "/" & NoOfThreads & " / " & Now()
        MainThreadStatus = 0: WriteMainThreadStatus 0 ' stop threads
        If (bOnlyMove And RootDepth = 1) Then Sleep 80 ' give helper threads time to start
        '--- Wait until Helper Threads are finished
        Dim hCnt As Long, thHelp As Long, bAllStopped As Boolean, ThrStatus As Long
        Dim tStart As Single, tEnd As Single
        If bThreadTrace Then tStart = Timer
        For hCnt = 1 To 10 ' try 10 times * sleep duration
          bAllStopped = True
          Sleep 50  ' wait in ms

          For thHelp = 1 To NoOfThreads - 1
            ThrStatus = ReadHelperThreadStatus(thHelp)
            If ThrStatus <> 0 Then
              If bThreadTrace Then WriteTrace "Think: stop threads:wait for  thread no "
               & thHelp & " / " & Now()
              bAllStopped = False: Exit For
            End If
          Next

          If bAllStopped Then
            If bThreadTrace Then WriteTrace "Think: all threads stopped-> exit" & " / "
            & Now()
            Exit For
          End If
        Next
        tEnd = Timer()
        If bThreadTrace Then WriteTrace "Threads stopped:" & bAllStopped & ",
        VerifyCnt=" & hCnt & ", Time:" & Format$(tEnd - tStart, "0.00000")

        '--- All threads stopped, is there a helper thread with deeper iteration?
        If bAllStopped Then
          If bThreadTrace Then WriteTrace "Think: Main= D:" & FinalCompletedDepth &
          ",DW:" & DepthInWork & "/S:" & FinalScore & "/M:" & MoveText(PV(1, 1))

          For thHelp = 1 To NoOfThreads - 1
            bHelperMove = ReadMapBestPVforThread(thHelp, HelperCompletedDepth,
            HelperBestScore, HelperPvLength, HelperNodes, HelperPV())
            'If Nodes < 1000000000 Then Nodes = Nodes + HelperNodes ' avoid overflow
            If bHelperMove And HelperPV(1).From > 0 Then
              If bThreadTrace Then WriteTrace "Think: check helper:" & thHelp & " = D:"
              & HelperCompletedDepth & "/S:" & HelperBestScore & "/L" & HelperPvLength &
               "/M:" & MoveText(HelperPV(1))
              If (HelperCompletedDepth >= FinalCompletedDepth Or HelperCompletedDepth >=
               DepthInWork) And HelperBestScore > FinalScore And HelperPvLength > 0 Then
                If MovePossible(HelperPV(1)) Then
                  ' Use result of this helper thread
                  If bThreadTrace Then
                    If UCIMoveText(HelperPV(1)) <> UCIMoveText(Think) Then
                      If bThreadTrace Then WriteTrace "!!!Think: use better move:" &
                      MoveText(HelperPV(1))
                    End If
                  End If
                  HelperPvLength = GetMin(GetMax(1, HelperPvLength), 9)
                  Think = HelperPV(1): FinalScore = HelperBestScore: FinalCompletedDepth
                   = HelperCompletedDepth
                  Erase PV()

                  For i = 1 To HelperPvLength: PV(1, i) = HelperPV(i): Next
                  PVLength(1) = HelperPvLength
                  If bThreadTrace Then WriteTrace "Think: use " & thHelp & " , Move:" &
                  MoveText(Think) & " Score:" & FinalScore
                Else
                  If bThreadTrace Then WriteTrace "Think: ??? wrong move " & thHelp & "
                  , Move:" & MoveText(HelperPV(1)) & " Score:" & FinalScore
                End If
              End If
            End If
          Next

        Else
          If bThreadTrace Then WriteTrace "***!!!***Think: NOT ALL THREADS STOPPED!"
        End If

        '======================================================================
        ' show result info in GUI
        '======================================================================
        SendThinkInfo Elapsed, GetMax(RootDepth, FinalCompletedDepth), FinalScore,
        RootAlpha, RootBeta ' show always with new nodes count

      ElseIf ThreadNum > 0 Then ' helper thread trace info
        If bThreadTrace Then WriteTrace "StartEngine: stopped thread: " & ThreadNum
        WriteHelperThreadStatus ThreadNum, 0
      End If

      If bTimeTrace Then WriteTrace "Think: end : " & MoveText(Think) & " " & Now()

    End Function '<<<<<<<< end of THINK <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

  '======================================================================
  '= SearchRoot: Search root moves                            =
  '=         called by THINK,  calls SEARCH                   =
  '======================================================================
  Private Function SearchRoot(ByVal Alpha As Long, _
                              ByVal Beta As Long, _
                              ByVal Depth As Long, _
```

```
                                  GoodMoves As Long) As TMOVE
  Dim RootScore              As Long, CurrMove As Long
  Dim BestRootScore          As Long
  Dim BestRootMove           As TMOVE, CurrentMove As TMOVE, HashMove As TMOVE
  Dim LegalMoveCnt           As Long, bCheckBest As Boolean, QuietMoves As Long,
  CaptureMoves As Long
  Dim Elapsed                As Single, lExtension As Long
  Dim PrevMove               As TMOVE
  Dim CutNode                As Boolean, r As Long, Factor As Long, s As String
  Dim NewDepth               As Long, Depth1 As Long, bCaptureOrPromotion As Boolean
  Dim Hashkey                As THashKey, EgCnt As Long, i As Long, bLegal As Boolean
  Dim EGTBBestRootMoveRootStr As String, EGTBBestRootMoveListRootStr As String
  Dim Improving              As Long
  Dim ss                     As Long ' Search stack pointer
  Dim BestValueCnt           As Long

  Dim bHashFound As Boolean, ttHit As Boolean, HashEvalType As Long, HashScore As Long
  , HashStaticEval As Long, HashDepth As Long
  Dim ttMove As TMOVE, ttValue As Long, HashPvHit As Boolean

  '---------------------------------------------
  ss = 1 ' reset search stack
  Ply = 1   ' start with ply 1 for root

  InitPieceSquares '-- also sets WKINGLOC and BKINGLOC needed for InCheck-Function later!
  InitEpArr

  EGTBRootResultScore = VALUE_NONE
  RootStartScore = Eval()
  PieceCntRoot = 2 + PieceCnt(WPAWN) + PieceCnt(WKNIGHT) + PieceCnt(WBISHOP) +
  PieceCnt(WROOK) + PieceCnt(WQUEEN) + PieceCnt(BPAWN) + PieceCnt(BKNIGHT) + PieceCnt(
  BBISHOP) + PieceCnt(BROOK) + PieceCnt(BQUEEN) ' For TableBases
  ' PlyMatScore (1) = WMaterial - BMaterial
  RootMatScore = WMaterial - BMaterial: If Not bWhiteToMove Then RootMatScore = -
  RootMatScore
  'RootSimpleEval = CalcSimpleEval()
  StaticEvalArr(0) = RootStartScore
  StaticEvalArr(ss + 1) = VALUE_NONE

  CutNode = False: QSDepth = 0
  bOnlyMove = False
  GoodMoves = 0: RootMoveCnt = 0
  ClearMove PrevMove
  BestRootScore = -MATE0
  ClearMove BestRootMove
  PliesFromNull = GameMovesCnt
  ClearMove BestMovePly(ss): ClearMove BestMovePly(ss + 1)
  If GameMovesCnt > 0 Then PrevMove = arGameMoves(GameMovesCnt)
  PrevMove.IsChecking = InCheck()
  Improving = Abs(Not PrevMove.IsChecking)
  StatScore(0) = 0
  If PrevMove.From > 0 Then StatScore(0) = History(PieceColor(PrevMove.Piece),
  PrevMove.From, PrevMove.Target) - 4000

  ' init history values
  CmhPtr(ss) = 0
  NullMovePly = 0
  RootDelta = 0

  StatScore(ss) = 0

  With Killer(ss + 2)
    ClearMove .Killer1: ClearMove .Killer2: ClearMove .Killer3
  End With

  ' ---Test time needed for evaluation function
  ' Debug.Print "-------------"
  ' If bEvalBench Then
```

```
    '  'Benchmark evalutaion
    '  Dim start As Single, ElapsedT As Single, lCnt As Long
    '  start = Timer
    '  For lCnt = 1 To 1500000
    '    RootStartScore = Eval()
    '  Next
    '  ElapsedT = TimerDiff(start, Timer)
    '  MsgBox Format$(ElapsedT, "0.000")
    '  End
    ' End If

    LegalMoveCnt = 0
    QuietMoves = 0
    CaptureMoves = 0
    bFirstRootMove = True
    PVLength(ss) = ss
    SearchStart = Timer

    ' Root check extent
    If InCheck Then
      Depth = Depth + 1
    End If


    RootDelta = Beta - Alpha
    ttPVArr(1) = True
    CutOffCnt(ss) = 0: CutOffCnt(ss + 1) = 0: CutOffCnt(ss + 2) = 0

    '=========================================
    '---  Root moves loop                 =
    '=========================================
    If RootDepth = 1 Then   ' for first call generate root moves
      GenerateMoves 1, False, CntRootMoves
      OrderMoves 1, CntRootMoves, PrevMove, EmptyMove, EmptyMove, False,
      LegalRootMovesOutOfCheck
      SortMovesStable 1, 0, CntRootMoves - 1     ' Sort by OrderVal
      'For CurrMove = 0 To CntRootMoves - 1: Debug.Print RootDepth, CurrMove, MoveText(Moves(1, CurrMove)),
      Moves(1, CurrMove).OrderValue: Next

    Else  ' rootmoves already generated, sort by value
      SortMovesStable 1, 0, CntRootMoves - 1      ' Sort by last iteration scores
      'For CurrMove = 0 To CntRootMoves - 1: Debug.Print RootDepth, CurrMove, MoveText(Moves(1, CurrMove)),
      Moves(1, CurrMove).OrderValue: Next
      For CurrMove = 1 To CntRootMoves - 1: Moves(1, CurrMove).OrderValue = -100000000:
      Next
    End If

    ClearMove SearchRoot: IsTBScore = False
    '--- Endgame Tablebase check for root position
    If EGTBasesEnabled And Not EGTBRootProbeDone Then
      EGTBRootProbeDone = True
      If bEGTbBaseTrace Then WriteTrace "TB-Root: TPos:" & IsEGTbBasePosition() & ",
      IsTime:" & IsTimeForEGTbBaseProbe
      If IsEGTbBasePosition() And IsTimeForEGTbBaseProbe Then
        Dim sTbFEN As String
        sTbFEN = WriteEPD()
        '<<<<< Tablebase access >>>>>>>>>>>>
        If ProbeTablebases(sTbFEN, EGTBRootResultScore, True, EGTBBestRootMoveRootStr,
        EGTBBestRootMoveListRootStr) Then
          EGTBBestRootMoveRootStr = LCase$(EGTBBestRootMoveRootStr)  ' lower promoted piece
          If bEGTbBaseTrace Then WriteTrace "TB-Root: Move " & EGTBBestRootMoveRootStr &
           " " & EGTBRootResultScore & " ListCnt=" & EGTBMoveListCnt(ss)

          For CurrMove = 0 To CntRootMoves - 1
            'Debug.Print CompToCoord(Moves(1, CurrMove))
            If CompToCoord(Moves(1, CurrMove)) = EGTBBestRootMoveRootStr Then
              SearchRoot = Moves(1, CurrMove)
              Moves(1, CurrMove).OrderValue = 5 * MATE0
              OrderMoves 1, CntRootMoves, PrevMove, EmptyMove, EmptyMove, False, LegalRootMovesOutOfCheck
              FinalMove = SearchRoot: FinalScore = EGTBRootResultScore: BestRootScore = FinalScore: PV(1, 1) = SearchRoot: PVLength(1) = 2
              ' Debug.Print "RootPos: "; CompToCoord(Moves(1, CurrMove)), FinalScore
              Elapsed = TimeElapsed()
              bTimeExit = True ' no more search
              LegalMoveCnt = 1
              If pbIsOfficeMode Then
                If EGTBRootResultScore = 0 Then
                 s = "DRAW"
                ElseIf EGTBRootResultScore > 0 Then
                  If EGTBRootResultScore = 100000 Then
                    s = "White mates!"
                  Else
                    s = "White wins in " & Abs(100000 - EGTBRootResultScore - 1) \ 2 & " moves"
                  End If
                ElseIf EGTBRootResultScore < 0 Then
                  If EGTBRootResultScore = -100000 Then
                    s = "Black mates!"
                  Else
                    s = "Black wins in " & Abs(100000 + EGTBRootResultScore + 1) \ 2 & " moves"
                  End If
                End If
                SendCommand s
              End If
              GoTo lblEndRootMoves   '<<<<<<<<<<<< NO MORE SEARCH  NEEDED for tablebase move
            End If
          Next
        End If
      End If
    End If ' <<< Endgame Tablebase check

    Elapsed = TimeElapsed()
    BestValueCnt = 0

    '=============================================
    '= loop for root moves                    =
    '=============================================
    For CurrMove = 0 To CntRootMoves - 1
      CurrentMove = Moves(1, CurrMove)
      MovePickerDat(ss).CurrMoveNum = CurrMove
      '  WriteTrace "SearchRoot RootDepth=" & RootDepth & " " & CurrMove & " " & MoveText(CurrentMove) & " Cnt=" & EGTBMoveListCnt(ss)
      ' Debug.Print MoveText(CurrentMove)
      RootScore = -VALUE_INFINITE
      If EGTBMoveListCnt(1) > 0 Then
        ' Filter for endgame tablebase move: Ignore loosingmoves if draw or win from tablebases
        For EgCnt = 1 To EGTBMoveListCnt(1)
          If CompToCoord(CurrentMove) = EGTBMoveList(1, EgCnt) Then GoTo lblEGMoveOK
        Next
        GoTo lblNextRootMove
      End If
  lblEGMoveOK:
      '
      CmhPtr(ss) = CurrentMove.Piece * MAX_BOARD + CurrentMove.Target ' set pointer for history move statistics
      ' WriteTrace "SearchRoot RootDepth=" & RootDepth & " " & CurrMove & " OK "

      '-------------------
      '--- Make root move -
      '-------------------
      RemoveEpPiece
      MakeMove CurrentMove: Ply = Ply + 1: bCheckBest = False: bLegal = False

      If CheckLegal(CurrentMove) Then
```

```
        Nodes = Nodes + 1: bLegal = True: LegalMoveCnt = LegalMoveCnt + 1: RootMoveCnt =
         LegalMoveCnt
        bCaptureOrPromotion = CurrentMove.Captured <> NO_PIECE Or CurrentMove.Promoted
        <> 0
        HashBoard Hashkey, EmptyMove
        If pbIsOfficeMode And RootDepth > 3 Then ' Show move cnt
          ShowMoveInfo MoveText(FinalMove), RootDepth, MaxPly, EvalSFTo100(FinalScore),
          Elapsed
        End If
        If UCIMode Then
          If TimeElapsed() > 3 Then
            SendCommand "info depth " & RootDepth & " currmove " & UCIMoveText(
            CurrentMove) & " currmovenumber " & LegalMoveCnt
          End If
        End If
        bFirstRootMove = CBool(LegalMoveCnt = 1)
        SetMove MovesList(ss), CurrentMove
        StaticEvalArr(ss) = RootStartScore
        RootMove = CurrentMove
        '-----------------
        'WriteTrace "Root:" & RootDepth & ": " & MoveText(CurrentMove) & " Score:" & FinalScore
        r = 0
        lExtension = 0
        '
        '--- Check extension ---
        '
        If (CurrentMove.IsChecking) Then
          If SEEGreaterOrEqual(CurrentMove, 0) Then
            lExtension = 1: GoTo lblEndExtensions
          End If
        End If
        ' Castling extension
        If CurrentMove.Castle <> NO_CASTLE Then
          lExtension = 1: GoTo lblEndExtensions
        End If
        ' Passed pawn move extension
        If PieceType(CurrentMove.Captured) = PT_PAWN Then
          If AdvancedPassedPawnPush(CurrentMove.Piece, CurrentMove.Target) Then
            lExtension = 1: GoTo lblEndExtensions
          End If
        End If

lblEndExtensions:

        '--- new search depth
        NewDepth = GetMax(0, Depth + lExtension - 1)

        '
        '--- Step 16. Reduced depth search (LMR). If the move fails high it will be re-searched at full depth.
        '
        r = Reduction(Improving, Depth, LegalMoveCnt, (Beta - Alpha), RootDelta)
        r = r - 1 'is Pv

        If Not bCaptureOrPromotion Then
          '--- Decrease reduction for moves that escape a capture
          If CurrentMove.Castle = NO_CASTLE Then
            TmpMove.From = CurrentMove.Target: TmpMove.Target = CurrentMove.From:
            TmpMove.Piece = CurrentMove.Piece: TmpMove.Captured = NO_PIECE:
            TmpMove.SeeValue = VALUE_NONE
            ' Move back to old square, were we in danger there?
            If Not SEEGreaterOrEqual(TmpMove, -MAX_SEE_DIFF) Then r = r - 2  ' old square
            was dangerous
          End If
        End If

        StatScore(ss) = History(PieceColor(CurrentMove.Piece), CurrentMove.From,
        CurrentMove.Target) - 4000  ' fill here if needed in next ply
        Dim CmH As Long
        CmH = PrevMove.Piece * MAX_BOARD + PrevMove.Target
        If CmH > 0 Then StatScore(ss) = StatScore(ss) + 2& * ContinuationHistory(CmH, CmhPtr(ss)) ' 2& to avoid integer overflow

        '--- Decrease/increase reduction for moves with a good/bad history
         If StatScore(ss) > 0 Then Factor = 22000 Else Factor = 20000
         r = GetMax(0, r - StatScore(ss) \ Factor)
         If RootDepth <= 6 + Abs(ThreadNum >= 1) * 4 + Abs(ThreadNum >= 3) * 4 Then r = 0 ' find some tactics, more if multiple threads

lblNoMoreReductions:
        '-----------------------------------------------
        '--->>>>  S E A R C H <<<<----------------------
        '-----------------------------------------------

        '---  Step 17. Late moves reduction
        If Depth >= 2 And LegalMoveCnt > 1 + 1 And Not bCaptureOrPromotion Then
          Depth1 = GetMax(NewDepth - r, 1): Depth1 = GetMin(Depth1, NewDepth + 1)
          RootScore = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, Depth1, CurrentMove, EmptyMove, True, lExtension)
          If (RootScore > Alpha And Depth1 < NewDepth) Then
            If NewDepth > Depth1 Then
              RootScore = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, NewDepth, CurrentMove, EmptyMove, True, lExtension)
            End If
          End If
        ElseIf LegalMoveCnt > 1 Then
          ' Full-depth search when LMR is skipped
          RootScore = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, NewDepth, CurrentMove, EmptyMove, True, lExtension)
        End If

        If (LegalMoveCnt = 1 Or RootScore > Alpha) And Not bTimeExit Then
          If NewDepth < 1 Then
            RootScore = -QSearch(ss + 1, PV_NODE, -Beta, -Alpha, MAX_DEPTH, CurrentMove, QS_CHECKS)
          Else
            RootScore = -Search(ss + 1, PV_NODE, -Beta, -Alpha, NewDepth, CurrentMove, EmptyMove, False, 0)
          End If
        End If
      End If

      '------------------
      '--- 18. Unmake move
      '------------------
      Call RemoveEpPiece: Ply = Ply - 1: UnmakeMove CurrentMove: ResetEpPiece

      '-------------------------
      ' check for best legal move
      '-------------------------
      If bTimeExit Then Exit For
      If Not bLegal Then GoTo lblNextRootMove
      '
      bCheckBest = True
      If RootDepth = 1 Then
        If EGTBMoveListCnt(1) > 0 And FinalMove.From > 0 Then bCheckBest = False 'Keep best EGTB move
      End If
      '
      If (LegalMoveCnt = 1 Or RootScore > Alpha) And bCheckBest Then
        'Debug.Print "Root:" & RootDepth, Ply, RootScore, MoveText(FinalMove)
        ' Set root move order value for next iteration <<<<<<<<<<<<<<<<
        FinalScore = RootScore: FinalMove = CurrentMove
        Moves(1, CurrMove).OrderValue = RootScore ' Root move ordering
        BestMovePly(ss) = FinalMove
        If LegalMoveCnt > 1 Then BestMoveChanges = BestMoveChanges + 1
        If Not bTimeExit Then
```

```
          GoodMoves = GoodMoves + 1
          DepthInWork = RootDepth ' For decision if better thread
        End If
        '--------------------
        '--- Save final move -
        '--------------------

        ' Store PV: best moves
        UpdatePV ss, FinalMove
        If PVLength(1) = 2 Then
          ' try to get 2nd move from hash
          HashMove = GetHashMove(Hashkey)
          If HashMove.From > 0 Then
            PV(1, 2) = HashMove: PVLength(1) = 3
          Else
            ClearMove PV(1, 2)
          End If
          If LastFullPVLen > 2 Then
            If MovesEqual(PV(1, 1), LastFullPVArr(1)) Then
              For r = 1 To LastFullPVLen:  SetMove PV(1, r), LastFullPVArr(r): Next
              PVLength(1) = LastFullPVLen
            End If
          End If
        ElseIf PVLength(1) > 2 Then
          For r = 1 To PVLength(1): SetMove LastFullPVArr(r), PV(1, r): Next
          LastFullPVLen = PVLength(1)
        End If
        If PV(1, 1).From > 0 Then ' helper thread writes result fxFCr main thread 0
          If ThreadNum > 0 Then WriteMapBestPVforThread FinalCompletedDepth, FinalScore,
           FinalMove
        End If
        LastChangeDepth = RootDepth
        LastChangeMove = MoveText(PV(1, 1))
      End If
      '
      '------- normal alpha beta check ------------------------
      '

      If RootScore > BestRootScore Then
        BestRootScore = RootScore

        If RootScore > Alpha Then
          BestRootMove = BestRootMove

          If RootScore >= Beta Then
              Exit For ' fail high
          Else
             If Depth > 2 And Depth < 12 And Beta < 14000 And RootScore > -12000 Then
             Depth = Depth - 2
             Alpha = RootScore
          End If
        ElseIf BestRootMove.From = 0 Then
          BestValueCnt = BestValueCnt + 1
          If BestValueCnt >= 3 Then Exit For
        End If
      End If
      '
      '--- Add Quiet move, used for pruning and history update
      '
      If Not MovesEqual(BestRootMove, CurrentMove) Then
        If CurrentMove.Captured = NO_PIECE And CurrentMove.Promoted = 0 And QuietMoves <
         64 Then
          QuietMoves = QuietMoves + 1: QuietsSearched(ss, QuietMoves) = CurrentMove
        ElseIf CurrentMove.Captured <> NO_PIECE And CaptureMoves < 32 Then
          CaptureMoves = CaptureMoves + 1: CapturesSearched(ss, CaptureMoves) =
          CurrentMove
        End If
    End If
      'If bTimeTrace Then WriteTrace "SearchRoot: FixedTime: " & FixedTime & " " & FixedDepthMode & ", TimeDiff:" & TimeElapsed()
      If Not FixedDepthMode And GoodMoves > 0 And Not bAnalyzeMode Then
        If FixedTime > 0 Then
          If TimeElapsed() >= FixedTime - 0.1 Then
            bTimeExit = True
          End If
        ElseIf (RootDepth > LIGHTNING_DEPTH) Then ' Time for next move?
          If Not CheckTime() Then
            SearchTime = TimeElapsed()
            If bTimeTrace Then WriteTrace "Exit SearchRoot3: Used:" & Format$(SearchTime, "0.00") & " OptimalTime:" & Format$(OptimalTime, "0.00")
            bTimeExit = True
          End If
        End If
      End If

      If (bTimeExit And LegalMoveCnt > 0) Or RootScore = MATE0 - 1 Then Exit For

      If pbIsOfficeMode Then
        If bTimeExit Then
          SearchTime = TimeElapsed()
          'Debug.Print Nodes, SearchTime
        End If
        #If VBA_MODE = 1 Then
          '-- Office sometimes lost focus for Powerpoint
          If Application.Name = "Microsoft PowerPoint" Then
            If RootDepth > 4 Then frmChessX.cmdStop.SetFocus
          End If
        #End If
        If RootDepth > 2 Then DoEvents
      Else
        If RootDepth > 6 Then DoEvents
      End If
      If bTimeExit Then Exit For
      '
  lblNextRootMove:
    Next CurrMove

    '---<<< End of root moves loop -------------

  lblEndRootMoves:
    '-------------------
    '--- End of game?  -
    '-------------------
    If LegalMoveCnt = 0 Then ' no move
      If InCheck Then ' Mate
        If bWhiteToMove Then
          Result = BLACK_WON
        Else
          Result = WHITE_WON
        End If
      Else ' draw
        Result = DRAW_RESULT: FinalScore = 0
        SetMove FinalMove, EmptyMove
      End If
      GoodMoves = -1
    Else
      If (LegalMoveCnt = 1 And RootDepth = 1) And Not bTimeExit Then bOnlyMove = True: RootScore = 0: FinalScore = 0 ' single move only?
      If RootScore = MATE0 - 2 Then ' Mate
        If bWhiteToMove Then
          Result = WHITE_WON
        Else
          Result = BLACK_WON
        End If
      Else
        If Fifty > 99 Then ' Draw 50 moves rule
```

```
        Result = DRAW_RESULT
      End If
    End If
  End If

  If FinalMove.From > 0 And Not bTimeExit Then
    UpdateStats ss, FinalMove, BestRootScore, Beta, QuietMoves, CaptureMoves,
    EmptyMove, RootDepth ' update statistics
    '--------------------------
    '--->>> Save hash for root
    '--------------------------
    HashBoard Hashkey, EmptyMove ' was changed above

    If FinalScore >= Beta Then
      HashEvalType = TT_LOWER_BOUND
    ElseIf FinalMove.From >= SQ_A1 Then
      HashEvalType = TT_EXACT
    Else
      HashEvalType = TT_UPPER_BOUND
    End If

    HashBoard Hashkey, EmptyMove ' changed before
    HashTableSave Hashkey, Depth, FinalMove, HashEvalType, FinalScore, StaticEvalArr(0
    ), True
    '  WriteTrace "SearchRoot SAVE TT:" & ThreadNum & ". " & RootDepth & " > " & MoveText(FinalMove) & " < "
    & FinalScore

    '----------------------
    '<<< Save hash for root
    '----------------------

  End If ' FinalMove.From

  '--------------------
  ' Return final move -
  '--------------------
  SearchRoot = FinalMove

  'WriteDebug "Root: " & RootDepth & " Best:" & MoveText(SearchRoot) & " Sc:" & BestRootScore & " M:" &
  GoodMoves
End Function


'========================================================================
'= Search: Search moves from ply=2 to x.                          =
'=        called by SEARCHROOT, calls SEARCH recursively , then QSEARCH.  =
'=        Returns eval score for a position with a specific search depth  =
'========================================================================
Private Function Search(ByVal ss As Long, _
                        ByVal PVNode As Boolean, _
                        ByVal Alpha As Long, _
                        ByVal Beta As Long, _
                        ByVal Depth As Long, _
                        InPrevMove As TMOVE, _
                        ExcludedMove As TMOVE, _
                        ByVal CutNode As Boolean, ByVal PrevMoveExtension As Long) As
                        Long
  '----------------------
  Dim CurrentMove       As TMOVE, Score As Long, bNoMoves As Boolean, bLegalMove As
  Boolean, LegalMovesOutOfCheck As Long
  Dim NullScore         As Long, PrevMove As TMOVE, QuietMoves As Long, CaptureMoves
  As Long, rBeta As Long, rDepth As Long
  Dim StaticEval        As Long, GoodMoves As Long, NewDepth As Long, LegalMoveCnt As
  Long, MoveCnt As Long
  Dim lExtension        As Long, lPlyExtension As Long, bTTMoveIsSingular As Boolean
  Dim bMoveCountPruning As Boolean, bKillerMove As Boolean, bTTCapture As Boolean,
  lSingularExtension As Long
  Dim r                 As Long, Improving As Long, bCaptureOrPromotion As Boolean,
  LmrDepth As Long, Depth1 As Long
  Dim BestValue          As Long, bIsNullMove As Boolean, ThreatMove As TMOVE,
  TryBestMove As TMOVE
  Dim bHashFound         As Boolean, ttHit As Boolean, HashEvalType As Long, HashScore
  As Long, HashStaticEval As Long, HashDepth As Long, HashThreadNum As Long
  Dim EvalScore          As Long, Hashkey As THashKey, HashMove As TMOVE, ttMove As
  TMOVE, ttValue As Long, HashPvHit As Boolean
  Dim BestMove           As TMOVE, sInput As String, MoveStr As String, Factor As Long,
   HistoryVal As Long
  Dim CmH                As Long, Fmh1 As Long, FMh3 As Long, HistVal As Long, CurrPtr
  As Long, Cm_Ok As Boolean
  Dim IsEGTbPos          As Boolean, bSingularExtensionNode As Boolean, ttPv As Boolean
  , bSkipQuiets As Boolean
  Dim bSingularQuietLMR As Boolean, bLikelyFailLow As Boolean, Bonus As Long,
  bAlmostFutilPruned As Boolean
  '----------------------
  Debug.Assert Not (PVNode And CutNode)
  Debug.Assert (PVNode Or (Alpha = Beta - 1))
  Debug.Assert (-VALUE_INFINITE <= Alpha And Alpha < Beta And Beta <= VALUE_INFINITE)
  Debug.Assert ss = Ply

  '----------------------------------------
  '--- Step 1. Initialize node for search -
  '----------------------------------------
  SetMove PrevMove, InPrevMove  '--- bug fix: make copy to avoid changes in parameter use
  BestValue = -VALUE_INFINITE: ClearMove BestMove:  ClearMove BestMovePly(ss):
  ClearMove BestMovePly(ss + 1)
  EvalScore = VALUE_NONE

  StaticEvalArr(ss + 1) = VALUE_NONE
  If ExcludedMove.From = 0 Then
    StaticEval = VALUE_NONE: StaticEvalArr(ss) = VALUE_NONE
  Else
    StaticEval = StaticEvalArr(ss)
  End If

  If bSearchingPV Then PVNode = True: CutNode = False ' searching main line is always principle
  variation

  If Ply > MaxPly Then MaxPly = Ply '--- Max depth reached in normal search

  ' ---- Q S E A R C H  ?-----
  If Depth <= 0 Or Ply >= MAX_DEPTH - 5 Then
    Search = QSearch(ss, PVNode, Alpha, Beta, MAX_DEPTH, PrevMove, QS_CHECKS)
    Exit Function  '<<<<<<< R E T U R N >>>>>>>>
  End If

  ClearMove ThreatMove: bTTMoveIsSingular = False
  bIsNullMove = (PrevMove.From < SQ_A1)
  EGTBMoveListCnt(ss) = 0
  '--- Debug ---
  ' dmoves  ' list search moves in debug window
  'If Ply = 2 And Left$(MoveText(PrevMove),4) = "c6d6" Then Stop ' Left needed for checking +
  'If RootDepth = 3 And Ply = 2 Then Debug.Print PrintPos, Movetext(PrevMove): Stop
  'If Nodes = 1127 Then Stop
  'If Ply > 70 Then Stop
  'If SearchMovesList = "h2c2 a1h1" Then Stop
  '  If Ply = 2 And Left$(MoveText(PrevMove), 4) = "g5d8" Then Stop ' Left needed for checking +

  bAlmostFutilPruned = False
  StatScore(ss) = 0
  CmhPtr(ss) = 0
  DoubleExtensions(ss) = DoubleExtensions(ss - 1)
  With Killer(ss + 2)
    ClearMove .Killer1: ClearMove .Killer2: ClearMove .Killer3
  End With
  CutOffCnt(ss + 2) = 0
  ttPv = PVNode: ttPVArr(ss) = ttPv
```

```

  '
  '--- Step 2. Check for aborted search and immediate draw
  '
  HashBoard Hashkey, ExcludedMove ' Save current position hash keys for insert later
  GamePosHash(GameMovesCnt + Ply - 1) = Hashkey


  ' Step 2. Check immediate draw
  If Fifty > 99 Then  ' 50 moves rule draw ?
    If CompToMove() Then Search = DrawContempt Else Search = -DrawContempt
    PVLength(ss) = 0
    Exit Function
  End If

  If Not bIsNullMove Then
    '--- 3x repeated position draw?
    If Fifty >= 3 And PliesFromNull >= 3 Then
      If Is3xDraw(Hashkey, GameMovesCnt, Ply) Then
        If CompToMove() Then Search = DrawContempt Else Search = -DrawContempt
        PVLength(ss) = 0
        Exit Function
      End If
    End If
  End If

  ' Endgame tablebase position?
  IsEGTbPos = False
  If EGTBasesEnabled And Ply <= EGTBasesMaxPly Then
    ' For first plies only because TB access is very slow for this implementation
    '  If EGTBRootResultScore = VALUE_NONE And PrevMove.Captured <> NO_PIECE Then ' not a TB position at root
    'If Ply <= EGTBasesMaxPly And PrevMove.Captured <> NO_PIECE Then ' captured because else TB access in previous ply
    If IsEGTbBasePosition() Then
      If IsTimeForEGTbBaseProbe() Then
        IsEGTbPos = True
      End If
    End If
    ' End If
  End If

  '
  '--- Step 3.:  Mate distance pruning
  '
  Alpha = GetMax(-MATE0 + Ply, Alpha)
  Beta = GetMin(MATE0 - Ply + 1, Beta)
  If Alpha >= Beta Then Search = Alpha: Exit Function

  If Alpha < DrawContempt And Fifty >= 3 And PliesFromNull >= 3 Then
 ' If Alpha < -DrawContemptForSide() And Fifty >= 3 Then
      If CyclingMoves(ss) Then
        Alpha = DrawContempt
        If Alpha >= Beta Then Search = Alpha: Exit Function
      End If
  End If

  '
  '--- Step 4. Transposition hash table lookup
  '
  NullScore = VALUE_NONE
  bHashFound = False: ttHit = False: ClearMove HashMove
  ttHit = False: ClearMove ttMove: ttValue = VALUE_NONE: bTTCapture = False

  If Depth >= 0 Then
    ttHit = HashTableRead(Hashkey, HashDepth, HashMove, HashEvalType, HashScore, HashStaticEval, HashPvHit, HashThreadNum)
    If ttHit Then
```

```
        SetMove ttMove, HashMove: ttValue = HashScore
        If HashMove.From <> 0 Then
          SetMove BestMovePly(ss), HashMove
          bTTCapture = (ttMove.Captured <> NO_PIECE Or ttMove.Promoted <> 0)
        End If
        If ExcludedMove.From = 0 Then ttPv = ttPv Or HashPvHit: ttPVArr(ss) = ttPv ' ttPv=PvNode earlier
      End If

       Dim bDoTT As Boolean

       If ThreadNum <= 0 Then     ' single core / main thread  / different to Stockfish logic HashDepth> Depth
          bDoTT = (Not PVNode Or HashDepth = TT_TB_BASE_DEPTH) And HashDepth >= Depth And ttHit And ttValue <> VALUE_NONE And ExcludedMove.From = 0
       Else ' multi core helper threads: different logic
          bDoTT = (Not PVNode Or HashDepth = TT_TB_BASE_DEPTH) And (HashDepth >= Depth - Abs(HashEvalType = TT_EXACT)) And ttHit And ttValue <> VALUE_NONE And ExcludedMove.From = 0
       End If
       If bDoTT Then
        If ttValue >= Beta Then
          bHashFound = CBool(HashEvalType And TT_LOWER_BOUND) ' bit wise compare eq: (HashEvalType = TT_LOWER_BOUND Or HashEvalType = TT_EXACT)
        Else
          bHashFound = CBool(HashEvalType And TT_UPPER_BOUND) ' bit wise compare eq: ((HashEvalType = TT_UPPER_BOUND Or HashEvalType = TT_EXACT)
        End If
        If bHashFound Then
          If IsEGTbPos And HashDepth <> TT_TB_BASE_DEPTH Then
            ' Ignore Hash and continue with TableBase query
          Else
            If ttMove.From >= SQ_A1 Then
              If ttValue >= Beta Then
                If Not bTTCapture Then
                  '--- Update statistics
                  UpdQuietStats ss, ttMove, PrevMove, StatBonus(Depth)
                End If

                ' Extra penalty for a quiet TT move in previous ply when it gets refuted
                If PrevMove.Captured = NO_PIECE Then
                  If PrevMove.From > 0 Then
                    If MovePickerDat(ss - 1).CurrMoveNum < 2 Or MovesEqual(PrevMove, Killer(ss - 1).Killer1) Then
                      UpdateContHistStats ss - 1, PrevMove.Piece, PrevMove.Target, -StatBonus(Depth + 1)
                    End If
                  End If
                End If
              ElseIf Not bTTCapture Then
                ' Penalty for a quiet ttMove that fails low
                Bonus = -StatBonus(Depth)
                UpdHistory ttMove.Piece, ttMove.From, ttMove.Target, Bonus
                UpdateContHistStats ss, ttMove.Piece, ttMove.Target, Bonus
              End If ' ttValue >= Beta
            End If ' ttMove.From >= SQ_A1

            If Fifty < 90 Then
              Search = ttValue
              BestMovePly(ss) = ttMove
              Exit Function  ' <<<< exit with TT move
            End If
          End If
        End If
      End If
    End If  '--- End Hash

    If Ply + Depth > MAX_DEPTH Then Depth = MAX_DEPTH - Ply - 2
    StaticEval = StaticEvalArr(ss)
    bNoMoves = True
    ClearMove BestMovePly(ss)

    '--- Check Time ---
    If Not FixedDepthMode Or ThreadNum > 0 Then
      '-- Fix:Nodes Mod 1000 > not working because nodes are incremented in QSearch too
      If (Nodes > LastNodesCnt + (GUICheckIntervalNodes * 2 \ (1 + Abs(bEndgame)))) And
      (RootDepth > LIGHTNING_DEPTH Or Ply = 2) Then
        #If DEBUG_MODE <> 0 Then
          DoEvents
        #End If
        ' --- Check new commands from GUI (i.e. analyze stop)
        If PollCommand Then
          If bThreadTrace Then WriteTrace "Search PollCommand: ThreadCommand =" &
          ThreadCommand & " / " & Now()
          sInput = ReadCommand
          If Left$(sInput, 1) = "." Then
            SendAnalyzeInfo
          Else
            If sInput <> "" Then
              ParseCommand sInput
            End If
          End If
        End If
        If ThreadNum > 0 Then CheckThreadTermination False   '<<< program my end here
        LastNodesCnt = Nodes
        If bTimeExit Then Search = 0: Exit Function
        If FixedTime > 0 Then
          If Not bAnalyzeMode And TimeElapsed() >= FixedTime - 0.1 Then bTimeExit = True
          : Exit Function
        ElseIf Not bAnalyzeMode Then
          If TimeElapsed() > MaximumTime Then
            If bTimeTrace Then WriteTrace "Exit Search: TimeElapsed: " & Format$(
            TimeElapsed() & ", Maximum:" & Format$(MaximumTime, "0.00")
            bTimeExit = True: Search = 0: Exit Function
          End If
        End If
      End If
    End If

    '
    '--- / Step 5. Tablebase (endgame) - not active any more because too slow with external calls
    '
    ' Tablebase access / too slow  in live tests
  '  If IsEGTbPos And HashDepth <> TT_TB_BASE_DEPTH Then   ' Postion already done and saved in hash?
  '    Dim sTbFEN As String, lEGTBResultScore As Long, sEGTBBestMoveStr As String, sEGTBBestMoveListStr As
  String
  '    sTbFEN = WriteEPD()
  '    If bEGTbBaseTrace Then WriteTrace "TB-Search: check move " & MoveText(PrevMove) & ", ply=" & Ply
  '    If ProbeTablebases(sTbFEN, lEGTBResultScore, True, sEGTBBestMoveStr, sEGTBBestMoveListStr) Then
  '      BestMove = TextToMove(sEGTBBestMoveStr)
  '      StaticEval = Eval(): lEGTBResultScore = lEGTBResultScore + StaticEval
  '      If bEGTbBaseTrace Then WriteTrace "TB-Search: Move " & sEGTBBestMoveStr & " " & lEGTBResultScore & "
  ply=" & Ply
  '      'Search = lEGTBResultScore
  '      HashTableSave HashKey, TT_TB_BASE_DEPTH, EmptyMove, TT_EXACT, lEGTBResultScore,
  lEGTBResultScore, ttPv
  '      SetMove ttMove, BestMove
  '    End If
  '  End If
  '

    '--- / Step 6. Evaluate the position statically
    If PrevMove.IsChecking Then
      StaticEval = VALUE_NONE: StaticEvalArr(ss) = VALUE_NONE: EvalScore = VALUE_NONE:
      Improving = 0
      GoTo lblSkipEarlyPruning   ' lblMovesLoop worse
    ElseIf ExcludedMove.From <> 0 Then
```

```
       StaticEval = StaticEvalArr(ss)
       EvalScore = StaticEval
    ElseIf ttHit Then
      If HashStaticEval = VALUE_NONE Then StaticEval = Eval() Else StaticEval = HashStaticEval
      EvalScore = StaticEval
      If ttValue <> VALUE_NONE Then
        If ttValue > EvalScore Then
          If CBool(HashEvalType And TT_LOWER_BOUND) Then EvalScore = ttValue
        Else
          If CBool(HashEvalType And TT_UPPER_BOUND) Then EvalScore = ttValue
        End If
      End If
    Else
      If StaticEval = VALUE_NONE Then
        StaticEval = Eval() '<<< evaluate position
      End If
      HashTableSave Hashkey, DEPTH_NONE, EmptyMove, TT_NO_BOUND, VALUE_NONE, StaticEval, ttPv  ' Save TT
      EvalScore = StaticEval
    End If
    StaticEvalArr(ss) = StaticEval

    '--- Improving ?
    Improving = 1
    If StaticEvalArr(ss - 2) <> VALUE_NONE Then
      Improving = Abs(StaticEval > StaticEvalArr(ss - 2))
    ElseIf StaticEvalArr(ss - 4) <> VALUE_NONE Then
      Improving = Abs(StaticEval > StaticEvalArr(ss - 4))
    End If

    If RootDepth <= 4 Then GoTo lblMovesLoop

    If (bWhiteToMove And CBool(WNonPawnMaterial = 0)) Or (Not bWhiteToMove And CBool(BNonPawnMaterial = 0)) Then GoTo lblMovesLoop
    '
    '--- Step 7. Razoring (skipped when in check)
    '
    If EvalScore < Alpha - 450 - 250 * Depth * Depth Then
      Score = QSearch(ss, NON_PV_NODE, Alpha - 1, Alpha, MAX_DEPTH, PrevMove, QS_CHECKS)
      If Score < Alpha Then
        CutOffCnt(ss) = CutOffCnt(ss) - 1
        Search = Score
        Exit Function
      End If
    End If
    '
    '--- Step 8. Futility pruning: child node (skipped when in check)
    '
    If Not PVNode And Depth < 9 And EvalScore > Beta And EvalScore < VALUE_KNOWN_WIN + 1 Then  ' >=beta bad? Different to SF
      If EvalScore - FutilityMargin(Depth, Improving) - StatScore(ss - 1) \ 280 >= Beta Then
        Search = EvalScore
        Exit Function
      End If
    End If

    '
    '--- Step 9. NULL MOVE ------------
    '
    If Not PVNode And PrevMove.From > 0 And PrevMoveExtension = 0 And EvalScore >= Beta And EvalScore >= StaticEval Then
     If Not bIsNullMove And StatScore(ss - 1) < 18755 And ExcludedMove.From = 0 Then
      If Fifty < 80 And Abs(Beta) < VALUE_KNOWN_WIN And Abs(StaticEval) < 2 * VALUE_KNOWN_WIN And Alpha <> DrawContempt - 1 Then
        If (StaticEval >= Beta - (35 * Depth) + 222) Then
          If (bWhiteToMove And WNonPawnPieces > 0) Or (Not bWhiteToMove And
                  BNonPawnPieces > 0) Then
                  If Ply >= NullMovePly Then
                    '--- Do NULLMOVE ---
                    Dim bOldToMove As Boolean, OldPliesFromNull As Long
                    bOldToMove = bWhiteToMove
                    OldPliesFromNull = PliesFromNull: PliesFromNull = 0
                    bWhiteToMove = Not bWhiteToMove 'MakeNullMove
                    ClearMove BestMovePly(ss + 1): CmhPtr(ss) = 0: RemoveEpPiece: ClearMove
                    MovesList(ss)
                    Ply = Ply + 1: EpPosArr(Ply) = 0: Fifty = Fifty + 1: ClearMove CurrentMove:
                    MovePickerDat(ss).CurrMoveNum = 0
                    Debug.Assert EvalScore - Beta >= 0

                    '--- Stockfish
                    r = GetMin((EvalScore - Beta) \ 168, 6) + Depth \ 3 + 4
                    If Depth - r <= 0 Then
                      NullScore = -QSearch(ss + 1, NON_PV_NODE, -Beta, -Beta + 1, MAX_DEPTH,
                      CurrentMove, QS_CHECKS)
                    Else
                      NullScore = -Search(ss + 1, NON_PV_NODE, -Beta, -Beta + 1, Depth - r,
                      CurrentMove, EmptyMove, Not CutNode, 0)
                    End If
                    Call RemoveEpPiece: Ply = Ply - 1: ResetEpPiece: Fifty = Fifty - 1: CmhPtr(
                    ss) = 0: PliesFromNull = OldPliesFromNull

                    ' UnMake NullMove
                    bWhiteToMove = bOldToMove
                    If bTimeExit Then Search = 0: Exit Function

                    If NullScore < -MATE_IN_MAX_PLY Then ' Mate threat : own extra logic
                       SetMove ThreatMove, BestMovePly(ss + 1)
                       lPlyExtension = 1: GoTo lblMovesLoop
                    End If

                    If NullScore >= Beta Then
                      If NullScore >= MATE_IN_MAX_PLY Then NullScore = Beta

                      If NullMovePly <> 0 Or (Abs(Beta) < VALUE_KNOWN_WIN And Depth < 12) Then
                        Search = NullScore
                        Exit Function
                      End If

                      '
                      ' Do verification search at high depths
                      '
                      NullMovePly = Ply + 3 * (Depth - r) \ 4 ' search depth for verification
                      If Depth - r <= 0 Then
                        Score = QSearch(ss, NON_PV_NODE, Beta - 1, Beta, MAX_DEPTH, PrevMove,
                        QS_CHECKS)
                      Else
                        Score = Search(ss, NON_PV_NODE, Beta - 1, Beta, Depth - r, PrevMove,
                        EmptyMove, False, 0)
                      End If
                      NullMovePly = 0
                      If Score >= Beta Then
                        Search = NullScore
                        Exit Function '--- Return Null Score, not Score!
                      End If

                    End If

                    '--- Capture Threat?  ( not SF logic )
                    If BestMovePly(ss + 1).From <> 0 Then
                      If (BestMovePly(ss + 1).Captured <> NO_PIECE Or NullScore < -
                      MATE_IN_MAX_PLY) Then
                        If Board(BestMovePly(ss + 1).Target) = BestMovePly(ss + 1).Captured Then
                         ' not changed by previous move
                          SetMove ThreatMove, BestMovePly(ss + 1)
```

```
              End If
            End If
          End If

        End If ' Ply >= NullMovePly
      End If
     End If
    End If
   End If
  End If
  '
  '--- Step 10. ProbCut (skipped when in check)
  '
  ' If we have a very good capture (i.e. SEE > seeValues[captured_piece_type])
  ' and a reduced search returns a value much above beta, we can (almost) safely prune the previous move.
  If Not PVNode And Depth > 4 And PrevMove.Target > 0 Then

    If Abs(Beta) < MATE_IN_MAX_PLY And Abs(StaticEval) < 2 * VALUE_KNOWN_WIN Then
      rBeta = GetMin(Beta + 186 - 54 * Improving, MATE0)

      If Not (ttHit And HashDepth >= Depth - 3 And ttValue <> VALUE_NONE And ttValue < rBeta) Then '+++2023+++

        Debug.Assert PrevMove.Target > 0
        MovePickerInit ss, ttMove, PrevMove, ThreatMove, True, False, GENERATE_ALL_MOVES

        Do While MovePicker(ss, CurrentMove, LegalMovesOutOfCheck)
          If CurrentMove.Captured <> NO_PIECE Or CurrentMove.Promoted > 0 Then
              If ExcludedMove.From <> 0 Then If MovesEqual(ExcludedMove, CurrentMove) Then GoTo lblNextProbCut
              rDepth = Depth - 4
              Debug.Assert rDepth >= 1
              '--- do the current move on the board ----------------------------
              CmhPtr(ss) = CurrentMove.Piece * MAX_BOARD + CurrentMove.Target
              Call RemoveEpPiece: MakeMove CurrentMove: Ply = Ply + 1
              bLegalMove = False
              If CheckLegal(CurrentMove) Then
                bLegalMove = True: SetMove MovesList(ss), CurrentMove
                ' Perform a preliminary qsearch to verify that the move holds
                Score = -QSearch(ss + 1, NON_PV_NODE, -rBeta, -rBeta + 1, MAX_DEPTH, CurrentMove, QS_CHECKS)
                ' If the qsearch held perform the regular search
                If Score >= rBeta Then
                  Score = -Search(ss + 1, NON_PV_NODE, -rBeta, -rBeta + 1, rDepth, CurrentMove, EmptyMove, Not CutNode, 0)
                End If
              End If
              '--- Undo move ------------
              Call RemoveEpPiece: Ply = Ply - 1: UnmakeMove CurrentMove: ResetEpPiece

              If Score >= rBeta And bLegalMove Then
                HashTableSave Hashkey, Depth - 3, CurrentMove, TT_LOWER_BOUND, Score, StaticEval, ttPv
                SetMove BestMovePly(ss), CurrentMove
                Search = Score
                Exit Function '---<<< Return
              End If
           End If
lblNextProbCut:
        Loop ' While MovePicker

     End If
    End If
  End If

lblSkipEarlyPruning:
  '
```

```
    ' Step 11. If the position is not in TT, decrease depth by 3.
    '
    ' Use qsearch if depth is equal or below zero (~9 Elo)
    If PVNode And ttMove.From = 0 Then
      Depth = Depth - (2 + 2 * Abs(ttHit And HashDepth >= Depth))
      If Depth <= 0 Then
        Search = QSearch(ss, PVNode, Alpha, Beta, MAX_DEPTH, PrevMove, QS_CHECKS)
        Exit Function  '<<<<<<< R E T U R N >>>>>>>>
      End If
    End If

    If CutNode And Depth >= 7 And ttMove.From = 0 Then
      Depth = Depth - 2 ' never zero
    End If
    '
    '--- Moves Loop ----------------
    '
  lblMovesLoop:

    ' Probcut idea
    rBeta = Beta + 391
    If PrevMove.IsChecking And Not PVNode And Depth >= 2 Then
      If bTTCapture And CBool(HashEvalType And TT_LOWER_BOUND) And ttValue >= rBeta And
      HashDepth >= Depth - 3 Then
        If Abs(ttValue) <= VALUE_KNOWN_WIN And Abs(Beta) <= VALUE_KNOWN_WIN Then
          Search = rBeta
          Exit Function  '<<<<<<< R E T U R N >>>>>>>>
        End If
      End If
    End If

    '---------

    Dim DrawMoveBonus As Long
    DrawMoveBonus = DrawValueForSide(bWhiteToMove)
    bSkipQuiets = False

    '
    '----  Singular extension search.
    '
    bTTMoveIsSingular = False
    lSingularExtension = 0
    If ttMove.From > 0 And ExcludedMove.From = 0 And HashDepth >= Depth - 3 Then
      bSingularExtensionNode = (Ply < RootDepth * 2) And (Depth >= 4 - Abs(RootDepth - 1
       > 20) + 2 * Abs(PVNode And ttPv)) _
                                    And Abs(ttValue) < VALUE_KNOWN_WIN And CBool(HashEvalType
                                     And TT_LOWER_BOUND)
    Else
      bSingularExtensionNode = False
    End If

   '--- SF logic (but moved before moves loop too avoid recursive call problems)
   If bSingularExtensionNode Then

     If MovePossible(ttMove) Then
        '--- Current move excluded
        '--- Make move        -
        Call RemoveEpPiece: MakeMove ttMove: Ply = Ply + 1
        bLegalMove = CheckLegal(ttMove)
        '--- Undo move ------------
        Call RemoveEpPiece: Ply = Ply - 1: UnmakeMove ttMove: ResetEpPiece

        If bLegalMove Then
          rBeta = GetMax(ttValue - ((3 + 2 * Abs(ttPv And Not PVNode)) * Depth) \ 2, -
          MATE0)
          'rBeta = GetMax(ttValue - ((82 + 65 * Abs(ttPv And Not PVNode)) * Depth) \ 64, -MATE0)

          Score = Search(ss, NON_PV_NODE, rBeta - 1, rBeta, (Depth - 1) \ 2, PrevMove,
        ttMove, CutNode, 0)
        DoubleExtensions(ss) = DoubleExtensions(ss - 1)

        If Score < rBeta Then
          bTTMoveIsSingular = True
          If Not bTTCapture And Not bIsNullMove Then
            CounterMove(PrevMove.Piece, PrevMove.Target) = ttMove
          End If
          lSingularExtension = 1
          bSingularQuietLMR = Not bTTCapture

          '(better for tactic but worse in game??? ) '+++SING2
         ' If Not PVNode And Score < rBeta - 25 And DoubleExtensions(ss) <= 10 And DoubleExtensions(ss) <= 1 + (RootDepth \ 12) Then  '  Avoid search explosion
          If Not PVNode And Score < rBeta - 25 And DoubleExtensions(ss) <= 10 Then ' Avoid search explosion
             lSingularExtension = 2
             If Depth < 13 Then Depth = Depth + 1
          End If

        ElseIf rBeta >= Beta Then
          Search = rBeta
          BestMovePly(ss) = ttMove
          Exit Function
        ElseIf ttValue >= Beta Then
           lSingularExtension = -2 - Abs(Not PVNode)

'        If Depth + lSingularExtension < HashDepth And Not PVNode Then
'          Search = ttValue
'          Exit Function
'        End If

        ElseIf CutNode Then
           If Depth < 17 Then lSingularExtension = -3 Else lSingularExtension = -1
        ElseIf ttValue <= Score Then
           lSingularExtension = -1
        End If
      End If ' bLegalMove
    End If ' MovePossible
 End If ' bSingularExtensionNode

  '-----------------------------------

  '--- Capture Threat?  ( not SF logic )
  If ThreatMove.From = 0 Then
    If BestMovePly(ss + 1).From <> 0 Then
      If (BestMovePly(ss + 1).Captured <> NO_PIECE) Then
        If Board(BestMovePly(ss + 1).Target) = BestMovePly(ss + 1).Captured Then 'not changed by previous move
          If Board(BestMovePly(ss + 1).From) = BestMovePly(ss + 1).Piece Then
            bWhiteToMove = Not bWhiteToMove
            If MovePossible(BestMovePly(ss + 1)) Then
              SetMove ThreatMove, BestMovePly(ss + 1)
            End If
            bWhiteToMove = Not bWhiteToMove
          End If
        End If
      End If
    End If
  End If

  '-------------------------------------------------
  '---- Step 12. Loop through moves      ------------
  '-------------------------------------------------
  PVLength(ss) = ss
  LegalMoveCnt = 0: QuietMoves = 0: CaptureMoves = 0: MoveCnt = 0
  If ttMove.From > 0 Then SetMove TryBestMove, ttMove Else ClearMove TryBestMove
  '
```

```
    ' Init MovePicker -----------------------------------
    '
    MovePickerInit ss, TryBestMove, PrevMove, ThreatMove, False, False, GENERATE_ALL_MOVES
    Score = BestValue
    ' Set move history pointer
    CmH = CmhPtr(ss - 1): Cm_Ok = (MovesList(ss - 1).From > 0)
    Fmh1 = 0:  FMh3 = 0 ' follow up moves
    If ss > 2 Then Fmh1 = CmhPtr(ss - 2): If ss > 4 Then FMh3 = CmhPtr(ss - 4)

    bMoveCountPruning = False
    bSingularQuietLMR = False
    bLikelyFailLow = (PVNode And ttMove.From <> 0 And CBool(HashEvalType And TT_UPPER_BOUND) And HashDepth >= Depth)
    '
    '--- Loop over moves --------------------------------
    '
    Do While MovePicker(ss, CurrentMove, LegalMovesOutOfCheck)
      If ExcludedMove.From > 0 Then If MovesEqual(CurrentMove, ExcludedMove) Then GoTo lblNextMove ' skip excluded move
      If PrevMove.IsChecking Then If Not CurrentMove.IsLegal Then GoTo lblNextMove '--- Legality for checks already tested in Ordermoves!
      bLegalMove = False: MoveCnt = MoveCnt + 1
      'Debug.Print "Search:" & RootDepth & ", ss:" & ss & " " & MoveText(CurrentMove)

      If EGTBMoveListCnt(ss) > 0 Then '--- move from tablebases?
        ' Filter for endgame tablebase move: Ignore loosing moves if draw or win from tablebases
        MoveStr = CompToCoord(CurrentMove)
        For r = 1 To EGTBMoveListCnt(ss)
          If MoveStr = EGTBMoveList(ss, r) Then GoTo lblEGMoveOK
        Next
        GoTo lblNextMove
      End If
  lblEGMoveOK:

      '--- set pointer to history statistics
      CurrPtr = CurrentMove.Piece * MAX_BOARD + CurrentMove.Target
      CmhPtr(ss) = CurrPtr

      '--- move count pruning / specifix login for ChessBrainVB: examine more moves if draw score
      bMoveCountPruning = Depth < 15 And MoveCnt >= FutilityMoveCnt(Improving, Depth) + Abs(Abs(BestValue) = DrawMoveBonus And BestValue > StaticEval) * 10
      bCaptureOrPromotion = (CurrentMove.Captured <> NO_PIECE Or CurrentMove.Promoted <> 0)
      bKillerMove = IsKiller1Move(ss, CurrentMove)
      lExtension = 0
      NewDepth = Depth - 1
      '
      '--- Step 14. Pruning at shallow depth ---------
      '
      r = Reduction(Improving, Depth, LegalMoveCnt, (Beta - Alpha), RootDelta) ' depth reduction depending on depth and move counter

      '--- Step 14. Pruning at shallow depth
      If BestValue > -MATE_IN_MAX_PLY Then
        ' reduce depth for next Late Move Reduction search
        LmrDepth = GetMax(NewDepth - r, 0)

        If bCaptureOrPromotion Or CurrentMove.IsChecking Or AdvancedPawnPush(CurrentMove.Piece, CurrentMove.Target) Then
          ' Capture or check
          If Not CurrentMove.IsChecking And Not PrevMove.IsChecking And Not PVNode And LmrDepth < 7 Then
            If StaticEval + 182 + 230 * LmrDepth + PieceAbsValue(CurrentMove.Captured) + CaptureHistory(CurrentMove.Piece, CurrentMove.Target, CurrentMove.Captured) \ 7 < Alpha Then
             GoTo lblNextMove
            End If
          End If
          If Not SEEGreaterOrEqual(CurrentMove, -206 * Depth) Then GoTo lblNextMove  ' piece can be captured?

        Else '--- not a capture > quiet move ----------

          If Not bKillerMove And bMoveCountPruning Then
            '
            ' Threat move logic specific to ChessBrainVB
            '
            With BestMovePly(ss + 1) ' new threat move?
              If .From > 0 And .Captured <> NO_PIECE Then
                If ThreatMove.From <> .From And ThreatMove.Target <> .Target Then
                  If Board(.Target) = .Captured Then
                    If BestMovePly(ss).From <> 0 And BestMovePly(ss).Target <> .Target And BestMovePly(ss).Target <> .From Then ' not changed by previous move
                      SetMove ThreatMove, BestMovePly(ss + 1) ' new threat move
                    End If
                  End If
                End If
              End If
            End With
            If ThreatMove.From > 0 Then ' try to avoid threat move
              ' don't skip threat escape
              If CurrentMove.From <> ThreatMove.Target Then ' threat escape?
                ' blocking threat move makes sense only with less or equal valuable piece
                If (PieceAbsValue(CurrentMove.Piece) - 80 < PieceAbsValue(ThreatMove.Piece)) Then
                  If IsBlockingMove(ThreatMove, CurrentMove) Then
                    ' blocking move - so do NOT skip this move
                    'Debug.Print PrintPos, MoveText(ThreatMove), MoveText(CurrentMove) : Stop
                  Else
                    bSkipQuiets = True
                    GoTo lblNextMove  ' skip this move, not a threat move defeat
                  End If
                End If
              End If
            Else
              bSkipQuiets = True
              GoTo lblNextMove ' not a threat move
            End If ' ThreatMove.From

          End If ' Not bKillerMove

          '--- ContinuationHistory based pruning
          HistoryVal = 0
          If CmH > 0 Then HistoryVal = HistoryVal + ContinuationHistory(CmH, CurrPtr)
          If Fmh1 > 0 Then HistoryVal = HistoryVal + ContinuationHistory(Fmh1, CurrPtr)
          If FMh3 > 0 Then HistoryVal = HistoryVal + ContinuationHistory(FMh3, CurrPtr)

          If LmrDepth < 5 And HistoryVal < -4405 * (Depth - 1) Then GoTo lblNextMove

          HistoryVal = HistoryVal + 2 * History(PieceColor(CurrentMove.Piece), CurrentMove.From, CurrentMove.Target)
          LmrDepth = LmrDepth + HistoryVal \ 7278
          LmrDepth = GetMax(LmrDepth, -2)

          Dim FutilVal As Long
          FutilVal = StaticEval + 104 + 145 * LmrDepth + HistoryVal \ 52
          If Not PrevMove.IsChecking And LmrDepth < 13 Then
            If FutilVal <= Alpha Then
              GoTo lblNextMove
            ElseIf FutilVal <= Alpha + 20 Then
              bAlmostFutilPruned = True
            End If
          End If
          LmrDepth = GetMax(LmrDepth, 0)

```

```
            '--- SEE based LMP
            If Not SEEGreaterOrEqual(CurrentMove, -24 * LmrDepth * LmrDepth - 16 *
            LmrDepth) Then GoTo lblNextMove
          End If ' bCaptureOrPromotion

        End If ' BestValue

        '
        '--- Step 13. Extensions
        '
        DoubleExtensions(ss) = DoubleExtensions(ss - 1) ' may be overwritten in searches before

        'if  We take care to not overdo to avoid search getting stuck.
        If Ply + 1 < RootDepth * 2 Then

         '- Singular move extent first , extension may be > 1 or < 0
         If lSingularExtension <> 0 And MoveCnt = 1 Then
           If MovesEqual(CurrentMove, ttMove) Then lExtension = lSingularExtension: GoTo
           lblEndExtensions
         End If

         '- Mate threat extent
         If lPlyExtension > 0 Then lExtension = 1: GoTo lblEndExtensions

         '- Single move check escape extent
         If (PrevMove.IsChecking) Then
           If LegalMovesOutOfCheck <= 1 Then lExtension = 1: GoTo lblEndExtensions
         End If

         '- Checking extension ---
         If (CurrentMove.IsChecking) Then
           If Depth > 10 And Abs(StaticEval) > 88 Then
             lExtension = 1: GoTo lblEndExtensions
           End If
         End If

         '- Queen exchange extent
         If Depth < 12 Then
           If PieceType(CurrentMove.Captured) = PT_QUEEN Then
             If PieceType(CurrentMove.Piece) = PT_QUEEN Then lExtension = 1: GoTo
             lblEndExtensions
           End If
         End If

         '- Castling extent
         If CurrentMove.Castle <> NO_CASTLE Then
           lExtension = 1: GoTo lblEndExtensions
         End If

         '- Good killer move extent
         If PVNode And bKillerMove Then
          If CmH > 0 And ttMove.From > 0 Then
           If MovesEqual(CurrentMove, ttMove) Then
             If ContinuationHistory(CmH, CurrPtr) > 5705 Then lExtension = 1: GoTo
             lblEndExtensions
           End If
          End If
         End If

         '- Passed pawn move extent
         If PieceType(CurrentMove.Captured) = PT_PAWN Then
            If AdvancedPassedPawnPush(CurrentMove.Piece, CurrentMove.Target) Then
            lExtension = 1: GoTo lblEndExtensions
         End If
       End If ' Ply < RootDepth * 2

lblEndExtensions:
        '- Add extensions to new depth for this move
        NewDepth = GetMax(0, NewDepth + lExtension)
        DoubleExtensions(ss) = DoubleExtensions(ss - 1) + Abs(lExtension >= 2)

        '-------------------------
        '--- Step 15. Make move  -
        '-------------------------
        Call RemoveEpPiece: MakeMove CurrentMove: Ply = Ply + 1
        If Not PrevMove.IsChecking And CurrentMove.Castle = NO_CASTLE Then
          CurrentMove.IsLegal = CheckLegalNotInCheck(CurrentMove)
    '     If CurrentMove.IsLegal Then ' verify correctness
    '       If Not CheckLegal(CurrentMove) Then WriteTrace PrintPos & MoveText(PrevMove) & " " & MoveText(CurrentMove): MsgBox "C1": Stop: End
    '     Else
    '      If CheckLegal(CurrentMove) Then WriteTrace PrintPos: MsgBox "C2": Stop: End
    '     End If
        ElseIf Not CurrentMove.IsLegal Then
          CurrentMove.IsLegal = CheckLegal(CurrentMove)
        End If

        '- move is legal
        If CurrentMove.IsLegal Then
          Nodes = Nodes + 1: LegalMoveCnt = LegalMoveCnt + 1

          #If DEBUG_MODE <> 0 Then
            If (Nodes \ 1000) Mod 5 = 0 Then DoEvents ' allow break in debug mode
          #End If

          bNoMoves = False: bLegalMove = True
          SetMove MovesList(ss), CurrentMove
          '
          '--- Step 16. Reduced depth search (LMR). If the move fails high it will be re-searched at full depth.
          '

          r = Reduction(Improving, Depth, LegalMoveCnt, (Beta - Alpha), RootDelta)

          If ttPv And Not bLikelyFailLow Then
            r = r - 2
          Else
            If bAlmostFutilPruned Then r = r - 1
          End If
          If MovePickerDat(ss - 1).CurrMoveNum > 7 Then r = r - 1 'Decrease reduction if opponent's move count is high
          If CutNode Then
            r = r + 2
          ElseIf CurrentMove.Castle = NO_CASTLE Then
            '--- Decrease reduction for moves that escape a capture
            TmpMove.From = CurrentMove.Target: TmpMove.Target = CurrentMove.From: TmpMove.Piece = CurrentMove.Piece: TmpMove.Captured = NO_PIECE: TmpMove.SeeValue = VALUE_NONE
            ' Move back to old square, were we in danger there?
            If Not SEEGreaterOrEqual(TmpMove, -MAX_SEE_DIFF) Then r = r - 2 ' old square was dangerous
          End If
          If bTTCapture Then r = r + 1  ' If TTMove was a capture, quiets rarely are better
          If PVNode Then If Depth > 0 Then r = r - (1 + 12 \ (3 + Depth)) ' PV node deeper
          If bSingularQuietLMR Then r = r - 1  ' quiet singular move
          If CutOffCnt(ss + 1) > 3 Then r = r + 1   ' many cutoffs for next ply
          If ttMove.From <> 0 Then If MovesEqual(CurrentMove, ttMove) Then r = r - 1 ' TT move
          If bKillerMove And CmH > 0 Then ' Good killer move
            If ContinuationHistory(CmH, CurrPtr) >= 3722 Then r = r - 1
          End If

    '     If ss > 2 And Fifty > 3 Then
    '      If CurrentMove.From = MovesList(ss - 2).Target Then
    '       If CurrentMove.Target = MovesList(ss - 2).From Then
    '         r = r + 1
    '         If ss > 4 Then If MovesEqual(CurrentMove, MovesList(ss - 4)) Then r = r + 1
    '     End If
    '    End If
    '   End If

    '   If ss > 4 And Fifty > 4 Then ' repeated move
    '     If MovesEqual(CurrentMove, MovesList(ss - 4)) Then r = r + 2 ': TestCnt(1) = TestCnt(1) + 1
    '   End If
    '   If CutOffCnt(ss + 1) > 3 Then
    '     r = r + 1  ' many cutoffs
    '   ElseIf ttMove.From <> 0 Then
    '     If MovesEqual(CurrentMove, ttMove) Then r = r - 1 ' TT move
    '   End If

          '
          HistVal = 2 * History(PieceColor(CurrentMove.Piece), CurrentMove.From, CurrentMove.Target)
          If CmH > 0 Then HistVal = HistVal + ContinuationHistory(CmH, CurrPtr)
          If Fmh1 > 0 Then HistVal = HistVal + ContinuationHistory(Fmh1, CurrPtr)
          If FMh3 > 0 Then HistVal = HistVal + ContinuationHistory(FMh3, CurrPtr)
          StatScore(ss) = HistVal - 4082

          '--- Decrease/increase reduction by comparing opponent's stat score
          If StatScore(ss) >= 0 And StatScore(ss - 1) < 0 Then
            r = r - 1
            If StatScore(ss) > StatScore(ss - 1) + 5000 Then r = r - 1
          ElseIf StatScore(ss - 1) >= 0 And StatScore(ss) < 0 Then
            r = r + 1
            If StatScore(ss) < StatScore(ss - 1) - 5000 Then r = r + 1
          End If

          '--- Decrease/increase reduction for moves with a good/bad history
          Factor = 11111 + 4700 * Abs(Depth > 5 And Depth < 22)
          r = r - StatScore(ss) \ Factor
          If r < 0 Then r = 0 ' ?! if r<0 search explosions
    lblNoMoreReductions:
          '---------  Step 17. Late moves reduction / extension
          If Depth >= 2 And LegalMoveCnt > 1 + Abs(PVNode) And _
              (Not ttPv Or Not bCaptureOrPromotion Or (CutNode And MovePickerDat(ss - 1).CurrMoveNum >= 1)) Then
            Depth1 = NewDepth - r
            If Depth1 < 1 Then Depth1 = 1 Else If Depth1 > NewDepth + 1 Then Depth1 = NewDepth + 1
            'rBeta = Abs(StaticEval >= Alpha - 81 * Depth)
            'If Depth1 < rBeta Then Depth1 = rBeta Else If Depth1 > NewDepth + 1 Then Depth1 = NewDepth + 1

            '--- Reduced SEARCH ---------
            Score = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, Depth1, CurrentMove, EmptyMove, True, lExtension)
            If (Score > Alpha And Depth1 < NewDepth) Then
              Dim bDoDeeperSearch As Boolean, bDoEvenDeeperSearch As Boolean, bDoShallowerSearch As Boolean
              bDoDeeperSearch = (Score > (Alpha + 58 + 12 * (NewDepth - Depth1)))
              bDoEvenDeeperSearch = (Score > Alpha + 588 And DoubleExtensions(ss) <= 5)
              bDoShallowerSearch = (Score < BestValue + NewDepth)

              DoubleExtensions(ss) = DoubleExtensions(ss) + Abs(bDoEvenDeeperSearch)

              NewDepth = NewDepth + Abs(bDoDeeperSearch) - Abs(bDoShallowerSearch) + Abs(bDoEvenDeeperSearch)
              If NewDepth > Depth1 Then
                Score = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, NewDepth, CurrentMove, EmptyMove, Not CutNode, lExtension)
              End If
              If Score <= Alpha Then
                Bonus = -StatBonus(Depth)    ' better than NewDepth?
              ElseIf Score >= Beta Then
                Bonus = StatBonus(Depth)
              Else
```

```
              Bonus = 0
            End If
            UpdateContHistStats ss, CurrentMove.Piece, CurrentMove.Target, Bonus
          End If ' Score

        ElseIf (Not PVNode Or LegalMoveCnt > 1) Then
            If ttMove.From = 0 And CutNode Then r = r + 2
            If NewDepth - Abs(r > 4) <= 0 Then
              Score = -QSearch(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, MAX_DEPTH,
              CurrentMove, QS_CHECKS)
            Else
              Score = -Search(ss + 1, NON_PV_NODE, -(Alpha + 1), -Alpha, NewDepth - Abs(
              r > 4), CurrentMove, EmptyMove, Not CutNode, lExtension)
           End If
        End If ' Depth >= 3 ...


        '----------------------------------------------------------------
        '--->>>> R E C U R S I V E   M A I N   S E A R C H <<<<----
        '----------------------------------------------------------------

        ' For PV nodes only, do a full PV search on the first move or after a fail
        ' high (in the latter case search only if value < beta), otherwise let the
        ' parent node fail low with value <= alpha and to try another move.
        If (PVNode And (LegalMoveCnt = 1 Or (Score > Alpha And Score < Beta))) And Not
        bTimeExit Then
          If NewDepth <= 0 Or (Ply + NewDepth >= MAX_DEPTH) Then
            Score = -QSearch(ss + 1, PV_NODE, -Beta, -Alpha, MAX_DEPTH, CurrentMove,
            QS_CHECKS)
          Else
            Score = -Search(ss + 1, PV_NODE, -Beta, -Alpha, NewDepth, CurrentMove,
            EmptyMove, False, lExtension)
          End If ' NewDepth
        End If ' PVNode

  lblSkipMove:
      End If '--- CheckLegal

      '---------------------------
      '---  Step 18. Undo move --
      '---------------------------
      Call RemoveEpPiece: Ply = Ply - 1: UnmakeMove CurrentMove: ResetEpPiece
      '
      If bTimeExit Then Search = 0: Exit Function

      '------------------------------------------
      '--- Step 19. Check for a new best move --
      '------------------------------------------
      If Score > BestValue And bLegalMove Then
        BestValue = Score

        If (Score > Alpha) Then
          GoodMoves = GoodMoves + 1
          SetMove BestMove, CurrentMove
          If PVNode Then UpdatePV ss, CurrentMove '--- Save PV ---
          If PVNode And Score < Beta Then
            If Depth > 1 And Depth < 6 And Beta < 10500 And Score > -10500 Then Depth =
            Depth - 1
            Alpha = Score
            Debug.Assert Depth > 0
          Else
            '--- Fail High  ---
            CutOffCnt(ss) = CutOffCnt(ss) + 1
            If StatScore(ss) < 0 Then StatScore(ss) = 0
            Exit Do
          End If
        End If
      End If

      If bLegalMove Then
        '--- Add Quiet move, used for pruning and history update
        If Not MovesEqual(BestMove, CurrentMove) Then
          If Not bCaptureOrPromotion And QuietMoves < 64 Then
           QuietMoves = QuietMoves + 1: SetMove QuietsSearched(ss, QuietMoves),
           CurrentMove
          ElseIf CurrentMove.Captured <> NO_PIECE And CaptureMoves < 32 Then
           If Not MovesEqual(BestMove, CurrentMove) Then CaptureMoves = CaptureMoves + 1
           : CapturesSearched(ss, CaptureMoves) = CurrentMove
          End If
        End If
      Else
        MoveCnt = MoveCnt - 1 ' not legal
      End If
  lblNextMove:
    Loop
    '--------------------------
    '--- next move in search ---
    '--------------------------

    '--------------------------------------------
    '--- Step 20. Check for mate and stalemate ---
    '--------------------------------------------
    If bNoMoves Then
      Debug.Assert LegalMovesOutOfCheck = 0 Or ExcludedMove.From > 0
      If ExcludedMove.From > 0 Then
        BestValue = Alpha
      ElseIf InCheck() Then '-- mate - do check again to be sure
        BestValue = -MATE0 + Ply ' mate in N plies
      Else ' draw
        If CompToMove() Then BestValue = DrawContempt Else BestValue = -DrawContempt
      End If
    ElseIf BestMove.From > 0 Then
      '--- New best move
      SetMove BestMovePly(ss), BestMove
      UpdateStats ss, BestMove, BestValue, Beta, QuietMoves, CaptureMoves, PrevMove,
      Depth + Abs((Not PVNode And Not CutNode) Or (BestValue > Beta + ScorePawn.MG))

      '--- Extra penalty for a quiet TT move in previous ply when it gets refuted
      If PrevMove.Captured = NO_PIECE Then
        If PrevMove.From > 0 And ss > 2 And CmH > 0 Then
          If MovePickerDat(ss - 1).CurrMoveNum = 0 Or IsKiller1Move(ss - 1, CurrentMove)
           Then
            UpdateContHistStats ss - 1, PrevMove.Piece, PrevMove.Target, -StatBonus(
            Depth + 1)
          End If
        End If
      End If
    Else
      '--- failed low - no best move
      ClearMove BestMovePly(ss)
      ' Bonus for prior countermove that caused the fail low
      If Depth >= 3 Or PVNode Then
        If PrevMove.Captured = NO_PIECE Then
          If Cm_Ok And ss > 2 Then
            r = Abs(Depth > 5) + Abs(PVNode Or CutNode) + Abs(BestValue < Alpha - 97 *
            Depth) + Abs(MovePickerDat(ss - 1).CurrMoveNum > 10)
            UpdateContHistStats ss - 1, PrevMove.Piece, PrevMove.Target, StatBonus(Depth
            ) * r
            'UpdHistory PrevMove.Piece, PrevMove.From, PrevMove.Target, StatBonus(Depth) * r * 3 \ 5
          End If
        End If
      End If
    End If

    If Fifty > 99 Then ' Draw 50 moves rule ?
      If CompToMove() Then BestValue = DrawContempt Else BestValue = -DrawContempt
    End If

    If BestValue <= Alpha Then ' add to pv?
      ttPv = ttPv Or (ttPVArr(ss - 1) And Depth > 3): ttPVArr(ss) = ttPv
    End If

    If ExcludedMove.From = 0 Then
      '-----------------------------------
      '--- Save hash values for best move ---
      '-----------------------------------
      If BestValue >= Beta Then
        HashEvalType = TT_LOWER_BOUND
      ElseIf PVNode And BestMove.From >= SQ_A1 Then
        HashEvalType = TT_EXACT
      Else
        HashEvalType = TT_UPPER_BOUND
      End If

      If BestValue = DrawMoveBonus Then Depth1 = GetMin(4, Depth) Else Depth1 = Depth
      HashTableSave Hashkey, Depth1, BestMove, HashEvalType, BestValue, StaticEval, ttPv
       'Save eval in hash table
    End If

    Search = BestValue ' return best score for search. Best move is saved in BestMovePly(ss) and PV.

  End Function
  '=====================================================================================
  '= end of SEARCH                                                  =
  '=====================================================================================


  '=====================================================================================
  '= QSearch (Quiescence Search): search for quiet position until no more capture possible,           =
  '=                             finally calls position evaluation                         =
  '=         called by SEARCH, calls QSEARCH recursively , then EVAL                          =
  '=====================================================================================
  Private Function QSearch(ByVal ss As Long, _
                           ByVal PVNode As Boolean, _
                           ByVal Alpha As Long, _
                           ByVal Beta As Long, _
                           ByVal Depth As Long, _
                           InPrevMove As TMOVE, _
                           ByVal GenerateQSChecks As Boolean) As Long
    '
    Dim PrevMove As TMOVE, Hashkey As THashKey, HashMove As TMOVE, bHashBoardDone As Boolean, ttDepth As Long, MoveCnt As Long, LegalMovesOutOfCheck As Long
    Dim bHashFound  As Boolean, ttHit As Boolean, HashEvalType As Long, HashScore As Long, HashStaticEval As Long, HashDepth As Long, HashPvHit As Boolean, ttPv As Boolean, HashThreadNum As Long
    If ss > MAX_DEPTH Then MsgBox "SS overflow:" & ss

    QSDepth = QSDepth + 1: If QSDepth > QSDepthMax Then QSDepthMax = QSDepth
    ClearMove BestMovePly(ss)

    If Not PVNode Then GenerateQSChecks = False ' QSChecks for PVNodes in first QS ply only because slow
    '
    SetMove PrevMove, InPrevMove: HashScore = VALUE_NONE
    bHashFound = False: ttHit = False: ClearMove HashMove: bHashBoardDone = False
    If Fifty > 99 Then  ' Draw ?
      If CompToMove() Then QSearch = DrawContempt Else QSearch = -DrawContempt
     QSDepth = QSDepth - 1
     Exit Function
    End If

    If Fifty >= 3 And PliesFromNull >= 3 Then
      HashBoard Hashkey, EmptyMove: bHashBoardDone = True ' Save current keys for insert later
      If Is3xDraw(Hashkey, GameMovesCnt, Ply) Then
        If CompToMove() Then QSearch = DrawContempt Else QSearch = -DrawContempt
        QSDepth = QSDepth - 1
        Exit Function ' -- Exit
      End If
    End If

    If (Depth <= 0 Or Ply >= MAX_DEPTH) Then
      QSearch = Eval(): QSDepth = QSDepth - 1
      Exit Function  '-- Exit
    End If

    '--- Mate distance pruning
  ' Alpha = GetMax(-MATE0 + Ply, Alpha)
  ' Beta = GetMin(MATE0 - Ply, Beta)
  ' If Alpha >= Beta Then QSearch = Alpha: Exit Function

    '--- Check Hash ---------------
    If Not bHashBoardDone Then HashBoard Hashkey, EmptyMove ' Save current keys for insert later
    GamePosHash(GameMovesCnt + Ply - 1) = Hashkey

    If PrevMove.IsChecking Or GenerateQSChecks Then
      ttDepth = DEPTH_QS_CHECKS   ' = 0
    Else
      ttDepth = DEPTH_QS_NO_CHECKS ' = -1
    End If
    ttHit = HashTableRead(Hashkey, HashDepth, HashMove, HashEvalType, HashScore,
    HashStaticEval, HashPvHit, HashThreadNum)
    ttPv = ttHit And HashPvHit
    If Not PVNode And ttHit Then
      If HashScore <> VALUE_NONE And HashDepth >= ttDepth Then
        If HashScore >= Beta Then
          bHashFound = (HashEvalType And TT_LOWER_BOUND)
        Else
          bHashFound = (HashEvalType And TT_UPPER_BOUND)
        End If
        If bHashFound Then
          SetMove BestMovePly(ss), HashMove
          QSearch = HashScore: QSDepth = QSDepth - 1
          Exit Function ' -- Exit
        End If
      End If
    End If

    '-------------------------------------------------------------------------------
    Dim CurrentMove As TMOVE, bNoMoves As Boolean, Score As Long, BestMove As TMOVE
    Dim bLegalMove  As Boolean, FutilBase As Long, FutilScore As Long, StaticEval As
    Long, BestValue As Long
    Dim bCapturesOnly As Boolean

    BestValue = -VALUE_INFINITE: StaticEval = VALUE_NONE
    If ttHit And HashMove.From > 0 Then SetMove BestMovePly(ss), HashMove Else ClearMove
     BestMovePly(ss)
    '-----------------------
    If PrevMove.IsChecking Then
      FutilBase = -VALUE_INFINITE
      bCapturesOnly = False ' search all moves to prove mate
    Else
      '--- SEARCH CAPTURES ONLY ----
      If ttHit Then
        If HashStaticEval = VALUE_NONE Then
          StaticEval = Eval()
        Else
          StaticEval = HashStaticEval
        End If
        BestValue = StaticEval
        If HashScore <> VALUE_NONE Then
          If HashScore > BestValue Then
            If CBool(HashEvalType And TT_LOWER_BOUND) Then BestValue = HashScore
          Else
            If CBool(HashEvalType And TT_UPPER_BOUND) Then BestValue = HashScore
          End If
        End If
      Else
        StaticEval = Eval()
        BestValue = StaticEval
      End If
      '--- Stand pat. Return immediately if static value is at least beta
      If BestValue >= Beta Then
        If Not ttHit Then
          HashTableSave Hashkey, DEPTH_NONE, EmptyMove, TT_LOWER_BOUND, BestValue, StaticEval, False
        End If
        QSearch = BestValue: QSDepth = QSDepth - 1
        Exit Function '-- exit
      End If
      If PVNode And BestValue > Alpha Then Alpha = BestValue
      FutilBase = StaticEval + 200
      bCapturesOnly = True ' Captures only
    End If ' PrevMove.IsChecking
    StaticEvalArr(ss) = StaticEval

    PVLength(ss) = ss: bNoMoves = True
    Dim QuietCheckEvasions As Long
    QuietCheckEvasions = 0

    '
    '---- QSearch moves loop ---------------
    '
    ' New: Always use hash move
    If HashMove.From > 0 Then ' Hash move is capture or check ?
      If GenerateQSChecks And HashMove.IsChecking Then
        ' keep Hash move
      ElseIf bCapturesOnly And HashMove.Captured <> NO_PIECE Then
        ' keep Hash move
      Else
        ClearMove HashMove
      End If
    End If

    Dim CmH As Long, Fmh As Long, CurrPtr As Long
    CmH = PrevMove.Piece * MAX_BOARD + PrevMove.Target
    If Ply > 2 Then Fmh = CmhPtr(Ply - 2) Else Fmh = 0

    MovePickerInit ss, HashMove, PrevMove, EmptyMove, bCapturesOnly, False, GenerateQSChecks

    Do While MovePicker(ss, CurrentMove, LegalMovesOutOfCheck)
      ' Debug.Print "QS:" & ss, MoveText(CurrentMove)
      MoveCnt = MoveCnt + 1
      If PrevMove.IsChecking Then
        If LegalMovesOutOfCheck = 0 Then
          '--- Mate
          QSearch = -MATE0 + Ply: QSDepth = QSDepth - 1
          Exit Function
        Else
          If Not CurrentMove.IsLegal Then GoTo lblNext
        End If
      ElseIf QSDepth > 6 Then ' recaptures only after 5 QS calls (starts with 1)
        If CurrentMove.Target <> PrevMove.Target Then GoTo lblNext
      End If

      Score = VALUE_NONE
```

```
        '------------------
        '--- Futil Pruning -
        '------------------
        'If BestValue > -MATE_IN_MAX_PLY And ((bWhiteToMove And CBool(WNonPawnMaterial <> 0)) Or (Not bWhiteToMove And CBool(BNonPawnMaterial <> 0))) Then
        If BestValue > -MATE_IN_MAX_PLY Then
          If Not CurrentMove.IsChecking And CurrentMove.Target <> PrevMove.Target And FutilBase > -VALUE_KNOWN_WIN And CurrentMove.Promoted = 0 Then
            If MoveCnt > 2 Then GoTo lblNext
            FutilScore = FutilBase
            If CurrentMove.Captured <> NO_PIECE Then FutilScore = FutilScore + PieceAbsValue(CurrentMove.Captured)

            If FutilScore <= Alpha Then
              If FutilScore > BestValue Then BestValue = FutilScore
              GoTo lblNext
            End If

            If FutilBase <= Alpha Then
              If Not SEEGreaterOrEqual(CurrentMove, 1) Then
                If FutilBase > BestValue Then BestValue = FutilBase
                GoTo lblNext
              End If
            End If

            If FutilBase > Alpha Then
              If Not SEEGreaterOrEqual(CurrentMove, (Alpha - FutilBase) * 4) Then
                BestValue = Alpha
                GoTo lblNext
              End If
            End If

          End If ' Not CurrentMove.IsChecking
          If QuietCheckEvasions > 1 Then Exit Do

          ' Continuation history based pruning
          If CurrentMove.Captured = NO_PIECE Then
              If CmH > 0 Then
                CurrPtr = CurrentMove.Piece * MAX_BOARD + CurrentMove.Target
                If ContinuationHistory(CmH, CurrPtr) < 0 Then
                  If Fmh > 0 Then
                    If ContinuationHistory(Fmh, CurrPtr) < 0 Then
                      GoTo lblNext
                    End If
                  End If
                End If
              End If
          End If

          ' Don't search moves with negative SEE values
          If Not SEEGreaterOrEqual(CurrentMove, -110) Then GoTo lblNext
        End If ' BestValue

        If PrevMove.IsChecking Then If CurrentMove.Captured = NO_PIECE Then QuietCheckEvasions = QuietCheckEvasions + 1

        '-----------------
        '--- Do QS move -
        '-----------------
        CmhPtr(ss) = CurrentMove.Piece * MAX_BOARD + CurrentMove.Target
        Call RemoveEpPiece: MakeMove CurrentMove: Ply = Ply + 1: bLegalMove = False

        If Not PrevMove.IsChecking And CurrentMove.Castle = NO_CASTLE Then
          CurrentMove.IsLegal = CheckLegalNotInCheck(CurrentMove)
    '     If CurrentMove.IsLegal Then  ' verify correctness
    '       If Not CheckLegal(CurrentMove) Then WriteTrace PrintPos & MoveText(PrevMove) & " " & MoveText(CurrentMove): MsgBox "C3": Stop: End
    '     Else
```

```
'      If CheckLegal(CurrentMove) Then WriteTrace PrintPos: MsgBox "C4": Stop: End
'    End If
      ElseIf Not CurrentMove.IsLegal Then
        CurrentMove.IsLegal = CheckLegal(CurrentMove)
      End If

      If CurrentMove.IsLegal Then
        Nodes = Nodes + 1: QNodes = QNodes + 1: bLegalMove = True: bNoMoves = False
        SetMove MovesList(ss), CurrentMove
        '------------------------------------
        '--- QSearch recursive  -------------
        '------------------------------------
        Score = -QSearch(ss + 1, PVNode, -Beta, -Alpha, Depth - 1, CurrentMove,
        QS_NO_CHECKS)
      End If

      '------------------
      '--- Undo QS move -
      '------------------
      Call RemoveEpPiece: Ply = Ply - 1: UnmakeMove CurrentMove: ResetEpPiece

      ' check for best move
      If (Score > BestValue) And bLegalMove Then
        BestValue = Score

        If Score > Alpha Then
          SetMove BestMove, CurrentMove
          SetMove BestMovePly(ss), CurrentMove
          'If bSearchingPV And PVNode Then UpdatePV ss, CurrentMove
          If Score < Beta Then
            Alpha = Score
          Else
            'If CutOffCnt(ss + 1) > 1 Then CutOffCnt(ss) = CutOffCnt(ss) + 1
            Exit Do '--- Fail high: >= Beta
          End If
        End If
      End If
lblNext:
    Loop '--- QS moves

    '--- Mate?
    If PrevMove.IsChecking And bNoMoves Then
      If InCheck() Then
        QSearch = -MATE0 + Ply ' mate in N plies, check again to be sure
        QSDepth = QSDepth - 1
        Exit Function
      End If
    End If

    '--- Save Hash values ---
    If BestValue >= Beta Then HashEvalType = TT_LOWER_BOUND Else HashEvalType =
    TT_UPPER_BOUND
    HashTableSave Hashkey, ttDepth, BestMove, HashEvalType, BestValue, StaticEval, ttPv
    ' save eval in hash table

    QSDepth = QSDepth - 1
    SetMove BestMovePly(ss), BestMove ' return QS best move
    QSearch = BestValue ' return QS score
End Function

'=========================================================================
'= OrderMoves()                                          =
'= Assign an order value to the generated moves                    =
'=========================================================================
Private Sub OrderMoves(ByVal Ply As Long, _
                       ByVal NumMoves As Long, _
                       PrevMove As TMOVE, _
                       BestMove As TMOVE, _
```

```
                     ThreatMove As TMOVE, _
                     ByVal bCapturesOnly As Boolean, _
                     LegalMovesOutOfCheck As Long)

Dim i                As Long, From As Long, Target As Long, Promoted As Long, Captured As Long, lValue As Long, Piece As Long, EnPassant As Long
Dim bSearchingPVNew As Boolean, BestValue As Long, BestIndex As Long, WhiteMoves As Boolean, CmH As Long
Dim bLegalsOnly      As Boolean, TmpVal As Long, PieceVal As Long, CounterMoveTmp As TMOVE, KingLoc As Long, v As Long
Dim Fm1              As Long, Fm2 As Long, Fm3 As Long, Fm5 As Long, CurrPtr As Long, bIsChecking As Boolean
'---------
LegalMovesOutOfCheck = 0
If NumMoves = 0 Then Exit Sub
bSearchingPVNew = False
BestValue = -9999999: BestIndex = -1 '--- save highest score
WhiteMoves = CBool((Board(Moves(Ply, 0).From) And 1) = 1) ' to be sure to have correct side ...
' set killer moves
Killer0 = Killer(Ply)
If Ply > 2 Then
  Killer2 = Killer(Ply - 2)
Else
  ClearMove Killer2.Killer1: ClearMove Killer2.Killer2: ClearMove Killer2.Killer3
End If

bLegalsOnly = PrevMove.IsChecking And Not bCapturesOnly ' Count legal moves in normal search (not in QSearch)
If bWhiteToMove Then KingLoc = WKingLoc Else KingLoc = BKingLoc

'--- set pointer to history statistics
CmH = PrevMove.Piece * MAX_BOARD + PrevMove.Target
If Ply > 2 Then Fm1 = CmhPtr(Ply - 2) Else Fm1 = 0
If Ply > 3 Then Fm2 = CmhPtr(Ply - 3) Else Fm2 = 0
If Ply > 4 Then Fm3 = CmhPtr(Ply - 4) Else Fm3 = 0
If Ply > 6 Then Fm5 = CmhPtr(Ply - 6) Else Fm5 = 0
SetMove CounterMoveTmp, CounterMove(PrevMove.Piece, PrevMove.Target)
'----------------
'--- Moves loop -
'----------------
For i = 0 To NumMoves - 1
  With Moves(Ply, i) ' assign move fields for speed reasons
    From = .From: Target = .Target: Promoted = .Promoted: Captured = .Captured: Piece = .Piece: EnPassant = .EnPassant: bIsChecking = .IsChecking
    .IsLegal = False: .SeeValue = VALUE_NONE
  End With

  lValue = 0
  '--- Count legal moves if in check
  If bLegalsOnly Then
    If Moves(Ply, i).Castle = NO_CASTLE Then ' castling not allowed in check
      ' Avoid costly legal proof for moves with cannot be a check evasion, EnPassant bug fixed here(wrong mate score if ep Capture is only legal move)
      If From <> KingLoc And PieceType(Captured) <> PT_KNIGHT And Not SameXRay(From, KingLoc) And Not SameXRay(Target, KingLoc) And EpPosArr(Ply) = 0 Then
        ' ignore this move because it  cannot be a check evasion
      Else
        ' Do move and test for legal
        RemoveEpPiece
        MakeMove Moves(Ply, i)
        If CheckEvasionLegal() Then Moves(Ply, i).IsLegal = True: LegalMovesOutOfCheck = LegalMovesOutOfCheck + 1
        ' Undo move
        UnmakeMove Moves(Ply, i)
        ResetEpPiece
      End If
    End If
    If Moves(Ply, i).IsLegal Then
```

```
          lValue = lValue + 3 * MATE0 '- Out of check moves have top order value
        Else
          lValue = -999999 ' not a legal evasion
          GoTo lblIgnoreMove
        End If
      End If

      PieceVal = PieceAbsValue(Piece)

      '--- Is Move checking ?
      If Not bIsChecking Then bIsChecking = IsCheckingMove(Piece, From, Target, Promoted
      , EnPassant)
      If bIsChecking Then
        If Not bCapturesOnly Then
          If Captured = NO_PIECE Then lValue = lValue + 9000
        Else
          lValue = lValue + 800 ' in QSearch search captures first??
        End If
        lValue = lValue + PieceVal \ 6
        If Ply > 2 Then
          If MovesList(Ply - 2).IsChecking Then lValue = lValue + 500 ' Repeated check
        End If
        Moves(Ply, i).IsChecking = True
      End If
      '--- bonus for main line
      If bSearchingPV Then
        If From = PV(1, Ply).From And Target = PV(1, Ply).Target And Promoted = PV(1,
        Ply).Promoted Then
          bSearchingPVNew = True: lValue = lValue + 2 * MATE0 ' Highest score
          GoTo lblNextMove
        End If
      End If
      '--- bonus for threat move
      If ThreatMove.From <> 0 Then
        If Target = ThreatMove.From Then
          lValue = lValue + 600  ' Try capture, additional bonus later for captures
        End If
        If From = ThreatMove.Target Then ' Try escape capture
          If PieceVal > PieceAbsValue(Board(ThreatMove.From)) + 80 Then
            lValue = lValue + 4000 + (PieceVal - PieceAbsValue(Board(ThreatMove.From)))
            \ 2
          Else
            lValue = lValue + 2000 + PieceVal \ 4
          End If
'    Else
'      ' blocking move?
'      If (PieceVal - 80 < PieceAbsValue(ThreatMove.Piece)) Then ' blocking makes sense only with less or equal
valuable piece
'        If IsBlockingMove(ThreatMove, Moves(Ply, i)) Then lValue = lValue + 300 +
PieceAbsValue(ThreatMove.Captured) \ 4
'      End If
        End If
      End If
      '--- Capture bonus
      If Captured <> NO_PIECE Then
        '-- Captures
        If Not bEndgame Then
          If bWhiteToMove Then lValue = lValue - 100 * Rank(Target) Else lValue = lValue
           - 100 * (9 - Rank(Target))
        End If
        If Piece = WKING Or Piece = BKING Then
          TmpVal = PieceAbsValue(Captured) ' cannot be defended because legal move
        Else
          TmpVal = PieceAbsValue(Captured) - PieceVal
        End If
        v = CaptureHistory(Piece, Target, Captured) \ 150
        If TmpVal > MAX_SEE_DIFF Then
          '--- Winning capture
          lValue = lValue + TmpVal * 5 + 6000 + v
        ElseIf TmpVal > -MAX_SEE_DIFF Then
          '--- Equal capture
          lValue = lValue + PieceAbsValue(Captured) - PieceVal \ 2 + 800 + v
        Else
          '--- Loosing capture? Check with SEE later in MovePicker
          lValue = lValue + PieceAbsValue(Captured) \ 2 - PieceVal + v
        End If
        If Target = PrevMove.Target Then lValue = lValue + 250 ' Recapture
        '-- King attack?
        If WhiteMoves Then
          If Piece <> WPAWN Then If MaxDistance(Target, BKingLoc) <= 2 And Target <> BKingLoc Then lValue = lValue + (PieceVal \ 2 + 400) \ MaxDistance(Target, BKingLoc)
        Else
          If Piece <> BPAWN Then If MaxDistance(Target, WKingLoc) <= 2 And Target <> WKingLoc Then lValue = lValue + (PieceVal \ 2 + 400) \ MaxDistance(Target, WKingLoc)
        End If
      Else
        '
        '--- Not a Capture, substract 30000 to select captures first
        '
        If Not bCapturesOnly Then lValue = lValue + MOVE_ORDER_QUIETS ' negative value for MOVE_ORDER_QUIETS > set to -30000
        'bonus per killer move:
        If From = Killer0.Killer1.From Then If Target = Killer0.Killer1.Target Then lValue = lValue + 3000: GoTo lblKillerDone
        If From = Killer0.Killer2.From Then If Target = Killer0.Killer2.Target Then lValue = lValue + 2500: GoTo lblKillerDone
        If From = Killer0.Killer3.From Then If Target = Killer0.Killer3.Target Then lValue = lValue + 2200: GoTo lblKillerDone

        If Ply > 2 Then '--- killer bonus for previous move of same color
          If From = Killer2.Killer1.From Then If Target = Killer2.Killer1.Target Then lValue = lValue + 2700: GoTo lblKillerDone
          If From = Killer2.Killer2.From Then If Target = Killer2.Killer2.Target Then lValue = lValue + 200
          ' Killer3 not better
        End If
        If PrevMove.Target <> 0 Then
          If CounterMoveTmp.Target = Target Then
            lValue = lValue + 250 ' Bonus for Countermove
            If CounterMoveTmp.Piece = Piece Then lValue = lValue + 250 - PieceVal \ 20
          End If
        End If
      End If

      '--- value for piece square table  difference of move
      lValue = lValue + PieceAbsValue(Promoted) \ 2 + (PsqVal(Abs(bEndgame), Piece, Target) - PsqVal(Abs(bEndgame), Piece, From)) * 2

      '--- Attacked by pawn or pawn push?
      If WhiteMoves Then
        If Piece = WPAWN Then
          If Rank(Target) >= 6 Then If AdvancedPawnPush(Piece, Target) Then lValue = lValue + 250
        Else
          If Board(Target + 9) = BPAWN Then lValue = lValue - PieceVal \ 4 Else If Board(Target + 11) = BPAWN Then lValue = lValue - PieceVal \ 4     '--- Attacked by Pawn
          If Board(Target - 9) = WPAWN Then lValue = lValue + 50 + PieceVal \ 8 Else If Board(Target - 11) = WPAWN Then lValue = lValue + 50 + PieceVal \ 8     '--- Defended by Pawn
          TmpVal = MaxDistance(Target, BKingLoc): lValue = lValue - TmpVal * TmpVal ' closer to opp king
        End If
      Else
        If Piece = BPAWN Then
```

```
        If Rank(Target) <= 3 Then If AdvancedPawnPush(Piece, Target) Then lValue = lValue + 250
      Else
        If Board(Target - 9) = WPAWN Then lValue = lValue - PieceVal \ 4 Else If Board(Target - 11) = WPAWN Then lValue = lValue - PieceVal \ 4    '--- Attacked by Pawn
        If Board(Target + 9) = BPAWN Then lValue = lValue + 50 + PieceVal \ 8 Else If Board(Target + 11) = BPAWN Then lValue = lValue + 50 + PieceVal \ 8      '--- Defended by Pawn
        TmpVal = MaxDistance(Target, WKingLoc): lValue = lValue - TmpVal * TmpVal ' closer to opp king
      End If
    End If
lblKillerDone:
    ' Check evasions
    If PrevMove.IsChecking Then
      If Piece = WKING Or Piece = BKING Then lValue = lValue + 200   ' King check escape move?
      If Target = PrevMove.Target Then lValue = lValue + 200 ' Capture checking piece?
      ' If PrevMove.Target > 0 Then lValue = lValue + History(PieceColor(Piece), From, Target) \ 6
    Else  ' not in check
      ' ContinuationHistory
      If Captured = NO_PIECE And Promoted = 0 Then
        v = 2& * History(PieceColor(Piece), From, Target)  ' 2& data type to avoid overflow
        If PrevMove.Target > 0 Then
          CurrPtr = Piece * MAX_BOARD + Target
          '  2& = LONG data type to avoid overflow
          v = v + (2& * ContinuationHistory(CmH, CurrPtr) + ContinuationHistory(Fm1, CurrPtr) + ContinuationHistory(Fm2, CurrPtr) + ContinuationHistory(Fm3, CurrPtr) + ContinuationHistory(Fm5, CurrPtr))
          v = v \ 12   ' bonus per history heuristic: Caution: big effects! +++order
        End If
        ' If v < TestCnt(1) Then TestCnt(1) = v
        ' If v > TestCnt(2) Then TestCnt(2) = v
        lValue = lValue + v
      End If
    End If ' PrevMove.IsChecking

lblNextMove:
    '--- Hashmove
    If BestMove.From = From Then If BestMove.Target = Target Then lValue = lValue + MATE0 \ 2: GoTo lblCheckBest
    '--- Move from Internal Iterative Depening
    If BestMovePly(Ply).From = From Then If BestMovePly(Ply).Target = Target Then lValue = lValue + MATE0 \ 2
lblCheckBest:
    If lValue > BestValue Then BestValue = lValue: BestIndex = i  '- save best for first move
lblIgnoreMove:
    ' Set order value for move picker
    Moves(Ply, i).OrderValue = lValue
  Next '---- Move

  bSearchingPV = bSearchingPVNew
  'Debug:  for i=0 to nummoves-1: Debug.Print i,Moves(ply,i).ordervalue, MoveText(Moves(ply,i)):next

  If BestIndex > 0 Then
    ' Swap best move to top
    SwapMove Moves(Ply, 0), Moves(Ply, BestIndex)
  End If
End Sub

'---------------------------------------------------------------------------------
'- BestMoveAtFirst: get best move from generated move list, scored by OrderMoves.
'-              Faster than SortMoves if alpha/beta cut in the first moves
'---------------------------------------------------------------------------------
Public Sub BestMoveAtFirst(ByVal Ply As Long, _
                           ByVal StartIndex As Long, _
                           ByVal NumMoves As Long)
  Dim i As Long, MaxScore As Long, MaxPtr As Long, ActScore As Long
```

```vb
    MaxScore = -9999999
    MaxPtr = StartIndex
    For i = StartIndex To NumMoves
      ActScore = Moves(Ply, i).OrderValue: If ActScore > MaxScore Then MaxScore =
      ActScore: MaxPtr = i
    Next i
    If MaxPtr > StartIndex Then
      SwapMove Moves(Ply, StartIndex), Moves(Ply, MaxPtr)
    End If
    ' For i = StartIndex To NumMoves '--- check for correct order
    '   If Moves(Ply, StartIndex - 1).OrderValue < Moves(Ply, i - 1).OrderValue Then Stop
    ' Next
  End Sub

  ' Stable sort: order of equal values is not changed
  Private Sub SortMovesStable(ByVal Ply As Long, ByVal iStart As Long, ByVal iEnd As
  Long)
    Dim i As Long, j As Long, iMin As Long, IMax As Long
    iMin = iStart + 1: IMax = iEnd
    i = iMin: j = i + 1

    Do While i <= IMax
      If Moves(Ply, i).OrderValue > Moves(Ply, i - 1).OrderValue Then
        SwapMove Moves(Ply, i), Moves(Ply, i - 1)
        If i > iMin Then i = i - 1
      Else
        i = j: j = j + 1
      End If
    Loop

  '  For i = iStart To iEnd - 1 ' Check sort order
  '   If Moves(Ply, i).OrderValue < Moves(Ply, i + 1).OrderValue Then Stop
  '  Next
  End Sub


  '---------------------------
  '--- init move picker list -
  '---------------------------
  Public Function MovePickerInit(ByVal ActPly As Long, _
                                 BestMove As TMOVE, _
                                 PrevMove As TMOVE, _
                                 ThreatMove As TMOVE, _
                                 ByVal bCapturesOnly As Boolean, _
                                 ByVal bMovesGenerated As Boolean, _
                                 ByVal bGenerateQSChecks As Boolean)

    With MovePickerDat(ActPly)
      .CurrMoveNum = 0
      .EndMoves = 0
      SetMove .BestMove, BestMove
      .bBestMoveChecked = False
      .bBestMoveDone = False
      SetMove .PrevMove, PrevMove
      SetMove .ThreatMove, ThreatMove
      .bCapturesOnly = bCapturesOnly
      .bMovesGenerated = bMovesGenerated
      .LegalMovesOutOfCheck = -1
      If bGenerateQSChecks Then .GenerateQSChecksCnt = 1 Else .GenerateQSChecksCnt = 0
    End With

  End Function

  '---------------------------------------------
  '- Move picker
  '-  Returns next move in "Move"
  '-  or function returns false if no more moves
  '---------------------------------------------
```

```
Public Function MovePicker(ByVal ActPly As Long, _
                           Move As TMOVE, _
                           LegalMovesOutOfCheck As Long) As Boolean
  Dim SeeVal As Long, NumMovesPly As Long, BestMove As TMOVE
  MovePicker = False: LegalMovesOutOfCheck = 0

  With MovePickerDat(ActPly)
    ' First: try BestMove. If Cutoff then no move generation needed.
    If Not .bBestMoveChecked Then
      .bBestMoveChecked = True
      If .BestMove.From <> 0 Then
        SetMove BestMove, .BestMove
        If Not .PrevMove.IsChecking Then ' Check: First generate all out of check moves, LegalMovesOutOfCheck needed
          If MovePossible(BestMove) Then
            SetMove Move, BestMove: .bBestMoveDone = True: MovePicker = True: Move.OrderValue = 5 * MATE0

            If bSearchingPV Then
              If Move.From = PV(1, ActPly).From And Move.Target = PV(1, ActPly).Target And Move.Promoted = PV(1, ActPly).Promoted Then
                ' keep SearchingPV
              Else
                bSearchingPV = False
              End If
            End If
            Exit Function '--- return best move before move generation
          End If
        End If
      End If
    End If
    '
    If Not .bMovesGenerated Then
      ' Generate all moves
      GenerateMoves ActPly, .bCapturesOnly, .EndMoves
      ' Order moves
      OrderMoves ActPly, .EndMoves, .PrevMove, .BestMove, .ThreatMove, .bCapturesOnly, .LegalMovesOutOfCheck
      .bMovesGenerated = True: .GenerateQSChecksCnt = 0: .CurrMoveNum = 0
    End If
    LegalMovesOutOfCheck = .LegalMovesOutOfCheck
    .CurrMoveNum = .CurrMoveNum + 1  ' array index starts at 0 = nummoves-1
    ' ignore Hash move, already done
    If .bBestMoveDone And BestMove.From <> 0 Then
      If MovesEqual(BestMove, Moves(ActPly, .CurrMoveNum - 1)) Then
        .CurrMoveNum = .CurrMoveNum + 1
      End If
    End If
    NumMovesPly = .EndMoves
    If NumMovesPly <= 0 Or .CurrMoveNum > NumMovesPly Then ClearMove Move: Exit Function
    If .CurrMoveNum > 1 Then ' First move is already sorted to top in OrderMoves
      ' sort best move to top of remaining list
      BestMoveAtFirst ActPly, .CurrMoveNum - 1, NumMovesPly - 1
    End If
    '----
    Do
      SetMove Move, Moves(ActPly, .CurrMoveNum - 1)
      If Not Move.IsChecking And Move.Captured = NO_PIECE Then MovePicker = True: Exit Function ' Quiet move
      If Move.OrderValue < MOVE_ORDER_BAD_CAPTURES + 5000 Then MovePicker = True: Exit Function  ' Bad Capture
      If .CurrMoveNum >= NumMovesPly Then MovePicker = True: Exit Function  ' Last move
      If Move.OrderValue > 1000 Then MovePicker = True: Exit Function ' Good Capture or killer
      '--- examine capture: good or bad?
      If PieceAbsValue(Move.Captured) - PieceAbsValue(Move.Piece) < -MAX_SEE_DIFF Then
        '-- Bad capture?
        SeeVal = GetSEE(Move): Move.SeeValue = SeeVal ' Slow! Delay the costly SEE until this move is needed - may be not needed if cutoffs earlier
        Moves(ActPly, .CurrMoveNum - 1).SeeValue = SeeVal  ' Save for later use
        If SeeVal >= -MAX_SEE_DIFF Then
          MovePicker = True: Exit Function
        Else
          Move.OrderValue = MOVE_ORDER_BAD_CAPTURES + SeeVal * 5 ' negative See!  - Set to fit condition above < -15000
          '- to avoid new list sort: append this bad move to the end of the move list (add new record), skip current list entry
          'Moves(ActPly, .CurrMoveNum  - 1).From = 0 ' Delete move in list,not needed ??
          NumMovesPly = NumMovesPly + 1: MovePickerDat(ActPly).EndMoves = NumMovesPly: Moves(ActPly, NumMovesPly - 1) = Move
        End If
      Else
        MovePicker = True: Exit Function  ' good captures
      End If
      .CurrMoveNum = .CurrMoveNum + 1 ' skip bad capture
    Loop

  End With

End Function

Public Function CompToMove() As Boolean
  If bCompIsWhite Then CompToMove = bWhiteToMove Else CompToMove = Not bWhiteToMove
End Function

Private Function FixedDepthMode() As Boolean
  '--- if no time limit use depth limit
  FixedDepthMode = CBool(FixedDepth <> NO_FIXED_DEPTH)
End Function

Public Function IsAnyLegalMove(ByVal NumMoves As Long) As Boolean
  ' Count legal moves
  Dim i As Long
  IsAnyLegalMove = False

  For i = 0 To NumMoves - 1
    RemoveEpPiece
    MakeMove Moves(Ply, i)
    If CheckLegal(Moves(Ply, i)) Then IsAnyLegalMove = True
    UnmakeMove Moves(Ply, i)
    ResetEpPiece
    If IsAnyLegalMove = True Then Exit Function
  Next i

End Function

'---------------------------------------------
'--- Check 3xRepetion Draw in current moves
'---------------------------------------------
Public Function Is3xDraw(Hashkey As THashKey, _
                         ByVal GameMoves As Long, _
                         ByVal SearchPly As Long) As Boolean
  Dim i As Long, Repeats As Long, EndPos As Long, StartPos As Long, PlyDiff As Long, Key1 As Long
  Is3xDraw = False

  If CompToMove Then
    PlyDiff = Fifty: If PliesFromNull < Fifty Then PlyDiff = PliesFromNull
  Else
    PlyDiff = Fifty - 1: If PliesFromNull - 1 < Fifty - 1 Then PlyDiff = PliesFromNull - 1
  End If
  If PlyDiff < 4 Then Exit Function
  If SearchPly > 1 Then SearchPly = SearchPly - 1
  StartPos = GameMoves + SearchPly - 1: If StartPos < 0 Then StartPos = 0
```

```
    EndPos = GameMoves + SearchPly - PlyDiff: If EndPos < 0 Then EndPos = 0
    If StartPos - EndPos < 2 Then Exit Function

    Repeats = 0: Key1 = Hashkey.HashKey1
    If Key1 = 0 Then Exit Function
    For i = StartPos - 1 To EndPos Step -2
      If Key1 = GamePosHash(i).HashKey1 Then
        If Hashkey.Hashkey2 = GamePosHash(i).Hashkey2 Then
          '2 repeats=3 equal positions.  1 repeated position in search=>Draw; or 1 in game plus 1 in search(except
          root) = 2 => draw
          Repeats = Repeats + 1
          If Repeats + Abs(i > GameMoves) >= 2 Then
            Is3xDraw = True: Exit Function
          End If
        End If
      End If
    Next i
  End Function

  Public Function CyclingMoves(ByVal ActPly As Long) As Boolean
    '--- repeated move ?  i.e.  "Ra1-a4  <opp move> Ra4-a1
    CyclingMoves = False

    If ActPly > 3 Then
      If Fifty >= 3 And PliesFromNull >= 3 Then
        If MovesList(ActPly - 3).From = MovesList(ActPly - 1).Target Then
          If MovesList(ActPly - 3).Target = MovesList(ActPly - 1).From Then
            If MovesList(ActPly - 2).Castle = NO_CASTLE And MovesList(ActPly - 1).Castle
             = NO_CASTLE Then
              If Not SqBetween(MovesList(ActPly - 1).Target, MovesList(ActPly - 2).From,
               MovesList(ActPly - 2).Target) Then
                CyclingMoves = True
              End If
            End If
          End If
        End If
      End If
    ElseIf ActPly = 2 Then
      If GameMovesCnt > 1 Then
        If arGameMoves(GameMovesCnt - 1).From = MovesList(ActPly - 1).Target Then
          If arGameMoves(GameMovesCnt - 1).Target = MovesList(ActPly - 1).From Then
            If arGameMoves(GameMovesCnt).Castle = NO_CASTLE And MovesList(ActPly - 1).
            Castle = NO_CASTLE Then
              If Not SqBetween(MovesList(ActPly - 1).Target, arGameMoves(GameMovesCnt).
              From, arGameMoves(GameMovesCnt - 1).Target) Then
                CyclingMoves = True
              End If
            End If
          End If
        End If
      End If
    End If
  End Function

  'Private Function IsKillerMove(ByVal ActPly As Long, Move As TMOVE) As Boolean
  '  If Move.From = 0 Then IsKillerMove = False: Exit Function
  '  IsKillerMove = True
  '  With Killer(ActPly)
  '    If Move.From = .Killer1.From Then If Move.Target = .Killer1.Target Then Exit Function
  '    If Move.From = .Killer2.From Then If Move.Target = .Killer2.Target Then Exit Function
  '    If Move.From = .Killer3.From Then If Move.Target = .Killer3.Target Then Exit Function
  '  End With
  '
  '  IsKillerMove = False
  'End Function
  '
  Private Function IsKiller1Move(ByVal ActPly As Long, Move As TMOVE) As Boolean 'first
  killer first?
    If Move.From = 0 Then IsKiller1Move = False: Exit Function
    IsKiller1Move = False
    With Killer(ActPly).Killer1
      If Move.From = .From Then If Move.Target = .Target Then If Move.Piece = .Piece
      Then IsKiller1Move = True
    End With
  End Function

  Public Function FutilityMoveCnt(ilImproving As Long, ilDepth As Long) As Long
    If ilImproving <> 0 Then FutilityMoveCnt = (3 + ilDepth * ilDepth) Else
    FutilityMoveCnt = (3 + ilDepth * ilDepth) \ 2
  End Function


  Public Function FutilityMargin(ByVal iDepth As Long, ByVal Improving As Long) As Long
    FutilityMargin = (154& * (iDepth - Improving))
  End Function


  Public Sub InitReductionArray()
    ' Init reductions array
    Dim mc As Long
    Debug.Assert NoOfThreads > 0

    For mc = 1 To 63
      Reductions(mc) = CLng(19.47 + Log(CDbl(NoOfThreads)) \ 2) * Log(CDbl(mc))
      'Debug.Print mc, Reductions(mc)
    Next mc

  End Sub

  '-------------------------
  '- Returns depth reduction
  '-------------------------
  Public Function Reduction(ByVal Improving As Long, _
                            ByVal Depth As Long, _
                            ByVal MoveNumber As Long, ByVal Delta As Long, ByVal
                            RootDelta As Long) As Long
    Dim r As Long
    If MoveNumber > 63 Then MoveNumber = 63
    r = Reductions(Depth) * Reductions(MoveNumber)
    Reduction = (r + 1372 - ((Delta * 1037) \ RootDelta)) \ 1024
    Debug.Assert Reduction >= 0
    If Improving = 0 Then If r > 936 Then Reduction = Reduction + 1
  End Function

  '--------------------
  '- Updates statistics
  '--------------------
  Private Function UpdateStats(ByVal ActPly As Long, _
                               BestMove As TMOVE, _
                               ByVal BestScore As Long, _
                               ByVal Beta As Long, _
                               ByVal QuietMovesSearched As Long, _
                               ByVal CaptureMovesSearched As Long, _
                               PrevMove As TMOVE, _
                               ByVal Depth As Long)
    '
    '--- Update Killer moves and History-Score
    '
    Dim j As Long, Bonus1 As Long
    Debug.Assert BestMove.Piece > FRAME And BestMove.Piece < NO_PIECE

    Bonus1 = StatBonus(Depth + 1)

    ' if NOT a capture
    If BestMove.From >= SQ_A1 And BestMove.Captured = NO_PIECE Then
      Dim Bonus2 As Long
```

```
     'If BestScore > Beta + 145 Then
     'If BestScore > Beta + 150 And BestScore > 0 Then ###beta###
     '--- not clear why * 150 instead of + 150 works much better here
     If BestScore > Beta * 150 Then Bonus2 = Bonus1 Else Bonus2 = StatBonus(Depth)

    ' Increase stats for the best move in case it was a quiet move
    UpdQuietStats ActPly, BestMove, PrevMove, Bonus2

    '--- Decrease History for previous tried quiet moves that did not cut off
    For j = 1 To QuietMovesSearched
       With QuietsSearched(ActPly, j)
         If .From = BestMove.From And .Target = BestMove.Target And .Piece = BestMove.Piece Then
           ' ignore
         Else
           UpdHistory .Piece, .From, .Target, -Bonus2
           If PrevMove.Target > 0 Then UpdateContHistStats ActPly, .Piece, .Target, -Bonus2
         End If
       End With
     Next j

  Else ' a Capture
    ' Increase stats for the best move in case it was a capture move
    UpdCaptureHistory BestMove.Piece, BestMove.Target, BestMove.Captured, Bonus1
  End If

  ' << Extra penalty for a quiet TT move in previous ply when it gets refuted > in Search Code

  '  Decrease stats for all non-best capture moves
  For j = 1 To CaptureMovesSearched
    With CapturesSearched(ActPly, j)
       If .From = BestMove.From And .Target = BestMove.Target And .Piece = BestMove.Piece Then
          ' ignore
       Else
         UpdCaptureHistory .Piece, .Target, .Captured, -Bonus1
       End If
    End With
  Next

End Function

Public Sub UpdQuietStats(ByVal ActPly As Long, _
                             CurrentMove As TMOVE, _
                             PrevMove As TMOVE, _
                             ByVal Bonus As Long)
  '--- update killer moves
  With Killer(ActPly)
    If CurrentMove.Target <> PrevMove.From Then ' not if opp moved attacked piece away > not a killer for other moves
      SetMove .Killer3, .Killer2: SetMove .Killer2, .Killer1: SetMove .Killer1, CurrentMove
    End If
  End With

  UpdHistory CurrentMove.Piece, CurrentMove.From, CurrentMove.Target, Bonus
  UpdateContHistStats ActPly, CurrentMove.Piece, CurrentMove.Target, Bonus

  If PrevMove.From >= SQ_A1 And PrevMove.Captured = NO_PIECE Then
    '--- CounterMove:
    SetMove CounterMove(PrevMove.Piece, PrevMove.Target), CurrentMove
  End If

End Sub

Public Sub UpdHistory(ByVal Piece As Long, _
                      ByVal From As Long, _
```

```
                            ByVal Target As Long, _
                            ByVal ScoreVal As Long)
  ' range +/- 10692
  Debug.Assert Piece > FRAME And Piece < NO_PIECE
  History(PieceColor(Piece), From, Target) = History(PieceColor(Piece), From, Target)
  + ScoreVal - (History(PieceColor(Piece), From, Target) * Abs(ScoreVal) \ 7183)
  'Debug.Assert Abs(History(PieceColor(Piece), From, Target)) <= 7183
End Sub

Public Sub UpdCaptureHistory(ByVal Piece As Long, _
                            ByVal Target As Long, _
                            ByVal CapturedPiece As Long, _
                            ByVal ScoreVal As Long)
  Debug.Assert Piece > FRAME And Piece < NO_PIECE
  CaptureHistory(Piece, Target, CapturedPiece) = CaptureHistory(Piece, Target,
  CapturedPiece) + ScoreVal - (CaptureHistory(Piece, Target, CapturedPiece) * Abs(
  ScoreVal) \ 10692)
  'Debug.Assert Abs(CaptureHistory(Piece, Target, CapturedPiece)) <= 10692
End Sub

Public Sub UpdateContHistStats(ByVal ActPly As Long, _
                               ByVal Piece As Long, _
                               ByVal Square As Long, _
                               ByVal Bonus As Long)
  Debug.Assert Piece > FRAME And Piece < NO_PIECE
  If ActPly > 1 Then
    If MovesList(ActPly - 1).From > 0 Then
      ContHistVal MovesList(ActPly - 1).Piece, MovesList(ActPly - 1).Target, Piece,
      Square, Bonus
    End If
    If ActPly > 2 Then
      If MovesList(ActPly - 2).From > 0 Then
        ContHistVal MovesList(ActPly - 2).Piece, MovesList(ActPly - 2).Target, Piece,
        Square, Bonus
      End If
'     If ActPly > 3 Then
'       If MovesList(ActPly - 3).From > 0 Then
'         ContHistVal MovesList(ActPly - 3).Piece, MovesList(ActPly - 3).Target, Piece, Square, Bonus \ 4
'       End If
        If ActPly > 4 And Not MovesList(ActPly - 1).IsChecking Then ' no more when in check
          If MovesList(ActPly - 4).From > 0 Then
            ContHistVal MovesList(ActPly - 4).Piece, MovesList(ActPly - 4).Target,
            Piece, Square, Bonus
          End If
          If ActPly > 6 Then
            If MovesList(ActPly - 6).From > 0 Then
              ContHistVal MovesList(ActPly - 6).Piece, MovesList(ActPly - 6).Target,
              Piece, Square, Bonus
            End If
          End If '6
        End If '4
    ' End If ' 3
    End If '2
  End If '1
End Sub

Public Sub ContHistVal(ByVal PrevPiece As Long, _
                       ByVal PrevSquare As Long, _
                       ByVal Piece As Long, _
                       ByVal Square As Long, _
                       ByVal ScoreVal As Long)
  ' Range +/-29952
  Debug.Assert Piece > FRAME And Piece < NO_PIECE
  Dim PrevPtr As Long, CurrPtr As Long
  PrevPtr = PrevPiece * MAX_BOARD + PrevSquare: CurrPtr = Piece * MAX_BOARD + Square
  ContinuationHistory(PrevPtr, CurrPtr) = ContinuationHistory(PrevPtr, CurrPtr) +
  ScoreVal - (ContinuationHistory(PrevPtr, CurrPtr) * Abs(ScoreVal) \ 29952)
  'Debug.Assert Abs(ContinuationHistory(PrevPtr, CurrPtr)) <= 29952
```

```
End Sub

'-------------------------------
'- update moves for current line
'-------------------------------
Public Sub UpdatePV(ByVal ActPly As Long, Move As TMOVE)
  Dim j As Long
  SetMove PV(ActPly, ActPly), Move
  If PVLength(ActPly + 1) > 0 Then

    For j = ActPly + 1 To PVLength(ActPly + 1) - 1
      SetMove PV(ActPly, j), PV(ActPly + 1, j)
    Next

    PVLength(ActPly) = PVLength(ActPly + 1)
  End If
End Sub

Public Function MovePossible(Move As TMOVE) As Boolean
  ' for test of HashMove before move generation if this move is possible. This may avoid move generation
  Dim Offset As Long, sq As Long, Diff As Long, AbsDiff As Long, OldPiece As Long
  MovePossible = False
  OldPiece = Move.Piece: If Move.Promoted > 0 Then OldPiece = Board(Move.From)
  If Move.From < SQ_A1 Or Move.From > SQ_H8 Or OldPiece < 1 Or Move.From = Move.Target Or OldPiece = NO_PIECE Then Exit Function
  If Board(Move.Target) = FRAME Then Exit Function
  If Board(Move.From) <> OldPiece Then Exit Function
  If Move.Captured < NO_PIECE Then If Board(Move.Target) <> Move.Captured Then Exit Function
  If bWhiteToMove Then
    If (OldPiece And 1) <> 1 Then Exit Function
  Else
    If (OldPiece And 1) <> 0 Then Exit Function
  End If
  If Board(Move.Target) <> NO_PIECE Then
    If (Board(Move.Target) And 1) = (OldPiece And 1) Then Exit Function  ' same color
  End If
  Diff = Move.Target - Move.From: AbsDiff = Abs(Diff)
  If PieceType(OldPiece) = PT_PAWN Then
    If (AbsDiff = 9 Or AbsDiff = 11) And Board(Move.Target) = NO_PIECE Then Exit Function
    If AbsDiff = 10 And Board(Move.Target) <> NO_PIECE Then Exit Function
    If AbsDiff = 20 Then
      If Board(Move.From + 10 * Sgn(Diff)) <> NO_PIECE Then Exit Function
      If Board(Move.Target) <> NO_PIECE Then Exit Function
    End If
    MovePossible = True
    Exit Function
  ElseIf OldPiece = WKNIGHT Or OldPiece = BKNIGHT Then

    ' Knight
    Select Case AbsDiff
      Case 8, 12, 19, 21
        MovePossible = True ' OK
    End Select

    Exit Function
  ElseIf OldPiece = WKING Then
    ' WKing: Castling
    If AbsDiff = 2 Then
      If Move.From <> WKING_START Or Moved(WKING_START) > 0 Then Exit Function
      If Diff = 2 Then
        If Board(Move.From + 1) <> NO_PIECE Or Board(Move.From + 2) <> NO_PIECE Or Board(Move.From + 3) <> WROOK Then Exit Function
      ElseIf Diff = -2 Then
        If Board(Move.From - 1) <> NO_PIECE Or Board(Move.From - 2) <> NO_PIECE Or Board(Move.From - 3) <> NO_PIECE Or Board(Move.From - 4) <> WROOK Then Exit Function
```

```vb
        End If
      End If
      MovePossible = True
      Exit Function
    ElseIf OldPiece = BKING Then
      ' BKing: Castling
      If AbsDiff = 2 Then
        If Move.From <> BKING_START Or Moved(BKING_START) > 0 Then Exit Function
        If Diff = 2 Then
          If Board(Move.From + 1) <> NO_PIECE Or Board(Move.From + 2) <> NO_PIECE Or
          Board(Move.From + 3) <> BROOK Then Exit Function
        ElseIf Diff = -2 Then
          If Board(Move.From - 1) <> NO_PIECE Or Board(Move.From - 2) <> NO_PIECE Or
          Board(Move.From - 3) <> NO_PIECE Or Board(Move.From - 4) <> BROOK Then Exit
          Function
        End If
      End If
      MovePossible = True
      Exit Function
    End If
    '--- Sliding piece blocked?
    If MaxDistance(Move.From, Move.Target) > 1 Then
      If AbsDiff Mod 9 = 0 Then
        Offset = Sgn(Diff) * 9
      ElseIf AbsDiff Mod 11 = 0 Then
        Offset = Sgn(Diff) * 11
      ElseIf AbsDiff Mod 10 = 0 Then
        Offset = Sgn(Diff) * 10
      Else
        Offset = Sgn(Diff) * 1
      End If

      Select Case OldPiece
        Case WROOK, BROOK:
          If Abs(Offset) <> 1 And Abs(Offset) <> 10 Then Exit Function
        Case WBISHOP, BBISHOP:
          If Abs(Offset) <> 9 And Abs(Offset) <> 11 Then Exit Function
        Case WQUEEN, BQUEEN:
          If Abs(Offset) <> 1 And Abs(Offset) <> 10 And Abs(Offset) <> 9 And Abs(Offset)
           <> 11 Then Exit Function
      End Select

      For sq = Move.From + Offset To Move.Target - Offset Step Offset
        If Board(sq) < NO_PIECE Then Exit Function
      Next

    End If
    MovePossible = True
  End Function

  Public Function PawnOnRank7() As Boolean
    ' check if side to move has a pawn on relative rank 7
    Dim i As Long
    If bWhiteToMove Then
      For i = SQ_A7 To SQ_H7
        If Board(i) = WPAWN Then PawnOnRank7 = True: Exit Function
      Next
    Else
      For i = SQ_A2 To SQ_H2
        If Board(i) = BPAWN Then PawnOnRank7 = True: Exit Function
      Next
    End If
    PawnOnRank7 = False
  End Function

  Public Sub ClearEasyMove()
    EasyMovePV(1) = EmptyMove: EasyMovePV(2) = EmptyMove: EasyMovePV(3) = EmptyMove
    EasyMoveStableCnt = 0
```

```vb
End Sub

Public Sub UpdateEasyMove()
  Dim i As Long, bDoUpdate As Boolean
  If MovesEqual(PV(1, 3), EasyMovePV(3)) Then
    EasyMoveStableCnt = EasyMoveStableCnt + 1
  Else
    EasyMoveStableCnt = 0
  End If
  bDoUpdate = False

  For i = 1 To 3
    If PV(1, i).From > 0 Then If Not MovesEqual(EasyMovePV(i), PV(1, i)) Then
    bDoUpdate = True
  Next

  If bDoUpdate Then
    For i = 1 To 3: EasyMovePV(i) = PV(1, i): Next
    'If bTimeTrace Then WriteTrace "UpdateEasyMove: " & MoveText(PV(1, 1)) & " " & MoveText(PV(1, 2)) & " " &
    MoveText(PV(1, 3))
  End If
End Sub

Public Function GetEasyMove() As TMOVE
  ' Return Easy move if previous moves are as expected
  SetMove GetEasyMove, EmptyMove
  If GameMovesCnt >= 2 And EasyMovePV(3).From > 0 Then
    If bTimeTrace Then WriteTrace "GetEasyMove: EM3" & MoveText(EasyMovePV(3)) & " (
    EM1:" & MoveText(EasyMovePV(1)) & " = GM1:" & MoveText(arGameMoves(GameMovesCnt -
    1)) & "  / EM2:" & MoveText(EasyMovePV(1)) & " = GM2:" & MoveText(arGameMoves(
    GameMovesCnt))
    If MovesEqual(EasyMovePV(1), arGameMoves(GameMovesCnt - 1)) And MovesEqual(
    EasyMovePV(2), arGameMoves(GameMovesCnt)) Then
      SetMove GetEasyMove, EasyMovePV(3)
    End If
  End If
End Function

Public Sub InitAttackBitCnt()
  ' fill array with attacking pieces count for attack bits set
  Dim i As Long, Cnt As Long

  For i = 1 To QXrayAttackBit * 2
    Cnt = 0
    If i And PLAttackBit Then Cnt = Cnt + 1
    If i And PRAttackBit Then Cnt = Cnt + 1
    If i And N1AttackBit Then Cnt = Cnt + 1
    If i And N2AttackBit Then Cnt = Cnt + 1
    If i And B1AttackBit Then Cnt = Cnt + 1
    If i And B2AttackBit Then Cnt = Cnt + 1
    If i And (R1AttackBit Or R1XrayAttackBit) Then Cnt = Cnt + 1
    If i And (R2AttackBit Or R2XrayAttackBit) Then Cnt = Cnt + 1
    If i And QAttackBit Then Cnt = Cnt + 1
    If i And KAttackBit Then Cnt = Cnt + 1
    If i And BXrayAttackBit Then Cnt = Cnt + 1   ' for multiple bishops
    If i And QXrayAttackBit Then Cnt = Cnt + 1   ' for multiple queens
    AttackBitCnt(i) = Cnt
  Next

End Sub

Public Function StatBonus(ByVal Depth As Long) As Long
   ' StatBonus = Depth * Depth + 2 * Depth - 2
  StatBonus = 340 * Depth - 470: If StatBonus > 1710 Then StatBonus = 1710
End Function


Public Function GetHashMove(Hashkey As THashKey) As TMOVE
```

```
  ' get best move for hint at root
  Dim ttHit As Boolean, HashEvalType As Long, HashScore As Long, HashStaticEval As
  Long, HashDepth As Long, HashMove As TMOVE, HashPvHit As Boolean, HashThreadNum As
  Long
  ClearMove GetHashMove
  ttHit = HashTableRead(Hashkey, HashDepth, HashMove, HashEvalType, HashScore,
  HashStaticEval, HashPvHit, HashThreadNum)
  If ttHit Then
    If HashMove.From <> 0 Then SetMove GetHashMove, HashMove
  End If
End Function

Public Function MoveInMoveList(ByVal ActPly As Long, _
                               ByVal StartIndex As Long, _
                               ByVal EndIndex As Long, _
                               CheckMove As TMOVE) As Boolean
  ' Check if the move is in the generate move list, and copies missing attribute ( IsChecking,...)
  Dim i As Long, tmp As TMOVE
  MoveInMoveList = False
  If CheckMove.From = 0 Then Exit Function

  For i = StartIndex To EndIndex
    'Debug.Print MoveText(Moves(ActPly, i))
    tmp = Moves(ActPly, i)
    If CheckMove.From <> tmp.From Then GoTo lblNext
    If CheckMove.Target <> tmp.Target Then GoTo lblNext
    If CheckMove.Promoted <> tmp.Promoted Then GoTo lblNext
    If CheckMove.Captured <> tmp.Captured Then GoTo lblNext
    ' Found
    SetMove CheckMove, tmp   ' return all attributes of the move
    MoveInMoveList = True
    Exit Function
lblNext:
  Next

End Function

Public Function DrawValueForSide(bSideToMoveIsWhite As Boolean) As Long
  If bCompIsWhite Then
    If bSideToMoveIsWhite Then DrawValueForSide = DrawContempt Else DrawValueForSide =
     -DrawContempt
  Else
    If Not bSideToMoveIsWhite Then DrawValueForSide = DrawContempt Else
    DrawValueForSide = -DrawContempt
  End If
End Function










Attribute VB_Name = "basTime"
Option Explicit
'======================
'= basTime:        =
'= Time management    =
'======================
Public bTimeExit                 As Boolean
Public TimeStart                 As Single
Public SearchStart               As Single
Public SearchTime                As Single
Public ExtraTimeForMove          As Single
Public TimeLeft                  As Single
Public OpponentTime              As Single
Public TimeIncrement              As Long
Public LevelMovesToTC             As Long
Public MovesToTC                  As Long
Public SecondsPerGame             As Long
Public FixedDepth                 As Long   '=NO_FIXED_DEPTH if time limit is used
Public FixedTime                  As Single
Public LastChangeDepth            As Long, LastChangeMove As String
Public bResearching               As Boolean '--- out of aspiration windows: more time
Public BestMoveChanges            As Single ' More time if best move changes often
Public MaximumTime                As Single
Public OptimalTime                As Single
Public MoveOverhead               As Single
Public MoreTimeForFirstMove       As Boolean   ' fill Hash table


'----------------
'- AllocateTime()
'----------------

Public Sub AllocateTime()
 Dim GameMovesDone As Long

  If bTimeTrace Then
    WriteTrace " ---------------------------------------------"
    WriteTrace ">> Start AllocateTime  MTOC:" & MovesToTC & ", MoveCnt=" & CStr(GameMovesCnt) & ", Left:" & Format$(TimeLeft, "0.00")
  End If

  GameMovesDone = (GameMovesCnt + 1) \ 2 ' Full move = 2* Half move

  If Not UCIMode And LevelMovesToTC > 0 Then
    MovesToTC = LevelMovesToTC - (GameMovesDone Mod LevelMovesToTC)
    If bTimeTrace Then
       WriteTrace "CalcTime WB: LevelMovesToTC=" & LevelMovesToTC & ", MovesToTC=" & MovesToTC & ", MovesDone:" & GameMovesDone
    End If
  End If
  '
  OptimalTime = CalcTime(MovesToTC, TimeIncrement, TimeLeft, MoveOverhead + 0.05 * (NoOfThreads - 1), True)
  MaximumTime = CalcTime(MovesToTC, TimeIncrement, TimeLeft, MoveOverhead + 0.05 * (NoOfThreads - 1), False)
  '
  MaximumTime = GetMinSingle(MaximumTime, TimeLeft / 2#)
  OptimalTime = GetMinSingle(MaximumTime, OptimalTime)

  If OptimalTime < 0.2 Then
    OptimalTime = GetMinSingle(0.2 + 0.05 * NoOfThreads, 1#): MaximumTime = OptimalTime
  End If
  MoreTimeForFirstMove = False
  If bTimeTrace Then
    WriteTrace ">>>> Time allocated Opt: " & Format$(OptimalTime, "0.00") & " / Max:" & Format$(MaximumTime, "0.00") & " MTOC:" & MovesToTC & " MoveCnt=" & CStr(GameMovesCnt) & ", Left:" & Format$(TimeLeft, "0.00")
  End If
End Sub

'
'--- Calculate time for move
'
Public Function CalcTime(ByVal MovesToTC As Long, _
                         ByVal TimeIncr As Single, _
                         ByVal MyTime As Single, _
                         ByVal MoveOverhead As Single, _
                         ByVal TimeTypeIsOptimum As Boolean) As Single
 Dim Ratio As Single, Inc As Single, k As Single, SafetyMargin As Single
 Dim GameMovesDone As Long
 GameMovesDone = (GameMovesCnt + 1) \ 2 ' Full move = 2* Half move

 If MyTime <= 0 Then CalcTime = 0: Exit Function

 Inc = TimeIncr * GetMaxSingle(60#, 125# - 0.1 * CSng((GameMovesDone - 23) * (GameMovesDone - 23)))
 SafetyMargin = 1.5

 If MovesToTC > 0 Then
   If TimeTypeIsOptimum Then
     Ratio = 1#
   Else
     If MovesToTC <= 10 Then Ratio = 2.5 Else Ratio = 4.5
   End If
   Ratio = Ratio / CSng(GetMin(45, GetMax(1, MovesToTC)))

   If GameMovesDone <= 40 Then
     Ratio = Ratio * (1.3 - 0.001 * CSng((GameMovesDone - 23) * (GameMovesDone - 23)))
   Else
     Ratio = Ratio * 1.45
   End If
   Ratio = Ratio * (1# + Inc / (MyTime * 8.2))
   If MovesToTC <= 3 Then SafetyMargin = 3#
 Else
   k = 1# + 21# * CSng(GameMovesDone) / CSng(500 + GameMovesDone)
   If TimeTypeIsOptimum Then Ratio = 0.021 Else Ratio = 0.075
   Ratio = Ratio * (k + Inc / MyTime)
 End If
 If MoreTimeForFirstMove Then Ratio = Ratio * 1.5
 '
 CalcTime = GetMinSingle(1#, Ratio) * GetMaxSingle(0.01, MyTime - MoveOverhead - SafetyMargin - TimeIncr / 10#)
End Function


Public Function TimerDiff(ByVal StartTime As Single, ByVal EndTime As Single) As Single
  If StartTime - 0.1 > EndTime Then ' Timer resets to 0 ad midnight > EndTime > Startime
    EndTime = EndTime + CSng(60& * 60& * 24&)
  End If
  TimerDiff = EndTime - StartTime
  If TimerDiff < 0 Then TimerDiff = 0.1
End Function

Public Function TimeElapsed() As Single
 TimeElapsed = TimerDiff(TimeStart, Timer())
End Function
'
'--- Check for time exceeded
'
Public Function CheckTime() As Boolean
  Dim Elapsed As Single, Improve As Single, Optimum2 As Single, NewScore As Long, PrevScore As Long
  CheckTime = True

  Elapsed = TimeElapsed()
  If FinalScore = VALUE_NONE Then NewScore = 0 Else NewScore = FinalScore
  If PrevGameMoveScore = VALUE_NONE Then PrevScore = FinalScore - 80 Else PrevScore = PrevGameMoveScore

  Improve = GetMaxSingle(229#, GetMinSingle(715#, 357# + 119# * Abs(bFailedLowAtRoot) - 5# * CSng(NewScore - PrevScore)))
  ' if score jumps up, extra time to make sure it holds
  If Abs(NewScore) < ScorePawn.MG * 5 Then
    If (NewScore - PrevScore) > ScorePawn.MG * 2 \ 3 Then
      Improve = Improve * 2
      If (NewScore - PrevScore) > ScorePawn.MG * 2 Then Improve = Improve * 2
    End If
  End If
  Optimum2 = (OptimalTime * (1# + BestMoveChanges) * Improve) / 640#

  If Elapsed >= GetMinSingle(MaximumTime, Optimum2) Then
    CheckTime = False
    If bTimeTrace Then
        WriteTrace "CheckTime D" & RootDepth & ": Elapsed:" & Format$(Elapsed, "0.00")
         & ", Opt2:" & Format$(Optimum2, "0.00") & ", Opt:" & Format$(OptimalTime,
        "0.00") & ", Max:" & Format$(MaximumTime, "0.00")
    End If
  End If

End Function
Attribute VB_Name = "UtilVBAbas"
'=======================================================================
'= UtilVBAbas:
'= functions for VBA GUI ( VBA= Visual Basic for Application in MS-Office)
'=======================================================================
Option Explicit
Public Const TEST_MODE As Boolean = True
Public ThisApp As Object ' Office object: Excel, Word,...
Public psGameFile As String
Public LastInfoNodes As Long

Public psLastFieldClick As String
Public psLastFieldMouseDown As String
Public psLastFieldMouseUp As String

Public SetupBoardMode As Boolean  ' manual board setup using GUI
Public SetupPiece As Long

' GUI colors
Public WhiteSqCol As Long
Public BlackSqCol As Long
Public BoardFrameCol As Long

Public plFieldFrom As Long, plFieldTarget As Long
Public psFieldFrom As String, psFieldTarget As String
Dim plFieldFromColor As Long, plFieldTargetColor As Long
Dim psMove As String

Sub run_ChessBrainX()
  Main
End Sub

Public Sub SetVBAPathes()
  pbIsOfficeMode = True
  Set ThisApp = Application
  Select Case ThisApp.Name
      Case "Microsoft Excel"
         psDocumentPath = ThisApp.ActiveWorkbook.Path

      Case "Microsoft Word"
        psDocumentPath = ThisApp.ActiveDocument.Path

     ' Case "Microsoft Powerpoint"
     '  psDocumentPath = ActivePresentation.Path

      Case Else
        psDocumentPath = ThisApp.ActiveWorkbook.Path
  End Select
  psAppName = "ChessBrainX"
  psEnginePath = psDocumentPath
End Sub

Public Sub DoFieldClicked()
  ' square click handling: 1. click: select FROM square, 2. click: select TARGET square => do move
```

```vb
    Dim bIsLegal As Boolean, NumLegalMoves As Long, FieldPos As Long, FieldTarget As
    Long
    Dim sPromotePiece As String, lResult As Long

    '--- Setup board mode:  if square not empty: 1 click: white piece, 2. click: black piece, 3. click: clear field
    If SetupBoardMode Then
      If Trim(psLastFieldClick) <> "" Then
        If SetupPiece > 0 Then
          psFieldFrom = psLastFieldClick
          plFieldFrom = Val("0" & Mid(psLastFieldClick, Len("Square") + 1))
          FieldPos = FieldNumToBoardPos(plFieldFrom)

          If Board(FieldPos) = NO_PIECE Or (PieceType(Board(FieldPos)) <> PieceType(
          SetupPiece)) Then
            Board(FieldPos) = SetupPiece
          ElseIf PieceColor(Board(FieldPos)) = COL_WHITE Then
            If PieceColor(SetupPiece) = COL_WHITE Then
               Board(FieldPos) = SetupPiece + 1 ' Black piece, same type
            Else
               Board(FieldPos) = NO_PIECE
            End If
          ElseIf PieceColor(Board(FieldPos)) = COL_BLACK Then
            If PieceColor(SetupPiece) = COL_BLACK Then
               Board(FieldPos) = SetupPiece - 1 ' white piece, same type
            Else
               Board(FieldPos) = NO_PIECE
            End If
          Else
            ' Clear
             Board(FieldPos) = NO_PIECE
          End If
          frmChessX.ShowBoard
          DoEvents
        End If
      End If
      Exit Sub
    End If

    ' Move input
    If Trim(psLastFieldClick) <> "" Then
      If plFieldFrom = 0 Then

        '--- First click: Field from
        psFieldFrom = psLastFieldClick
        plFieldFrom = Val("0" & Mid(psLastFieldClick, Len("Square") + 1))
        FieldPos = FieldNumToBoardPos(plFieldFrom)
        If Board(FieldPos) < NO_PIECE Then
          '-- check color to move
          If bWhiteToMove And Board(FieldPos) Mod 2 <> 1 Or _
            Not bWhiteToMove And Board(FieldPos) Mod 2 <> 0 Then
            '--- wrong color
            SendCommand "Wrong color! "
            plFieldFrom = 0
            ResetGUIFieldColors
          Else
            frmChessX.Controls(psLastFieldClick).BackColor = &HFF8080
            ShowLegalMovesForPiece FieldNumToCoord(plFieldFrom)
          End If
        Else
          ' ignore empty field
          plFieldFrom = 0
          ResetGUIFieldColors
        End If

      Else

        '--- Second click: Field target
        If psLastFieldClick = psFieldFrom Then
```

```
         ResetGUIFieldColors
         DoEvents
         plFieldFrom = 0
      Else
        psFieldTarget = psLastFieldClick
        plFieldTarget = Val("0" & Mid(psLastFieldClick, Len("Square") + 1))
        frmChessX.Controls(psLastFieldClick).BackColor = &HC0FFC0
        DoEvents
        Sleep 250
        '--- Check player move
        bIsLegal = CheckGUIMoveIsLegal(FieldNumToCoord(plFieldFrom), FieldNumToCoord(plFieldTarget), NumLegalMoves)
        If bIsLegal Then
          ' Promotion?
          sPromotePiece = "": FieldPos = FieldNumToBoardPos(plFieldFrom): FieldTarget = FieldNumToBoardPos(plFieldTarget)
          If (Board(FieldPos) = WPAWN And Rank(FieldTarget) = 8) Or (Board(FieldPos) = BPAWN And Rank(FieldTarget) = 1) Then
            lResult = MsgBox(Translate("Promote to queen?"), vbYesNo) ' or Knight
            If lResult = vbYes Then sPromotePiece = "q" Else sPromotePiece = "n"
          End If
          '--- Send move to Engine
          psMove = FieldNumToCoord(plFieldFrom) & FieldNumToCoord(plFieldTarget) & sPromotePiece & vbLf
          ParseCommand psMove
          frmChessX.ShowMoveList
          frmChessX.ShowBoard
        Else
          If NumLegalMoves = 0 Then
            If InCheck() Then
              SendCommand "Mate!"
            Else
              SendCommand "No legal move -> Draw!!!"
            End If
          Else
            SendCommand "Illegal move: " & FieldNumToCoord(plFieldFrom) & FieldNumToCoord(plFieldTarget) & " !!!"
          End If
        End If

        'Reset
        plFieldFrom = 0: plFieldTarget = 0
        ResetGUIFieldColors

        If bIsLegal And frmChessX.chkAutoThink = True Then
          DoEvents
          frmChessX.cmdThink_Click
          DoEvents
        End If
      End If
    End If
  Else
   ResetGUIFieldColors
  End If
  DoEvents
End Sub


Public Function FieldNumToBoardPos(ByVal ilFieldNum As Long) As Long
   Dim s As String
   s = FieldNumToCoord(ilFieldNum)
   FieldNumToBoardPos = FileRev(Left(s, 1)) + RankRev(Mid(s, 2, 1))
End Function


Public Function CheckGUIMoveIsLegal(MoveFromText, MoveTargetText, oLegalMoves As Long) As Boolean
  ' Input: "e2", "e4", Output:  oLegalMoves:Number of Legal Moves
```

```
    Dim a As Long, NumMoves As Long, From As Long, Target As Long
    CheckGUIMoveIsLegal = False

    Ply = 0
    oLegalMoves = GenerateLegalMoves(NumMoves)
    If oLegalMoves > 0 Then
      From = FileRev(Left(MoveFromText, 1)) + RankRev(Mid(MoveFromText, 2, 1))
      Target = FileRev(Left(MoveTargetText, 1)) + RankRev(Mid(MoveTargetText, 2, 1))

      For a = 0 To NumMoves - 1
         If Moves(0, a).From = From And Moves(0, a).Target = Target Then
            CheckGUIMoveIsLegal = Moves(0, a).IsLegal: Exit For
         End If
      Next a
    End If
  End Function

  Public Sub ShowLegalMovesForPiece(MoveFromText)
    ' Input: square as text "e2"
    Dim a As Long, NumMoves As Long, From As Long, Target As Long
    Dim NumLegalMoves As Long, ctrl As Control, bFound As Boolean

    Ply = 0: bFound = False
    NumLegalMoves = GenerateLegalMoves(NumMoves)
    From = FileRev(Left(MoveFromText, 1)) + RankRev(Mid(MoveFromText, 2, 1))
    If NumLegalMoves = 0 Then
      SendCommand "No legal moves!"
    Else
      For Each ctrl In frmChessX.Controls
        Target = Val("0" & ctrl.Tag)
        If Target > 0 Then
          For a = 0 To NumMoves - 1
           If Moves(0, a).From = From And Moves(0, a).Target = Target And Moves(0, a).
           IsLegal Then
             ctrl.BackColor = &HC0FFC0
             bFound = True
           End If
          Next a
        End If
      Next ctrl
      If Not bFound Then
        SendCommand "No legal move for this piece!"
      End If
    End If

  End Sub

  Public Sub ResetGUIFieldColors()
   Dim x As Long, y As Long, bBackColorIsWhite As Boolean, i As Long

   bBackColorIsWhite = False

   For y = 1 To 8
    For x = 1 To 8
      i = x + (y - 1) * 8
      With frmChessX.fraBoard.Controls("Square" & i)
        If bBackColorIsWhite Then
         If .BackColor <> WhiteSqCol Then .BackColor = WhiteSqCol
        Else
         If .BackColor <> BlackSqCol Then .BackColor = BlackSqCol
        End If
      End With
      bBackColorIsWhite = Not bBackColorIsWhite
    Next x
    bBackColorIsWhite = Not bBackColorIsWhite
   Next y
  End Sub
```

```


Public Function GenerateLegalMoves(olTotalMoves As Long) As Long
  ' Returns all moves in Moves(ply). Moves(x).IsLegal=true for legal moves
  Dim LegalMoves As Long, lLegalMoves As Long, i As Long, NumMoves As Long

  GenerateMoves Ply, False, NumMoves
  Ply = 0: lLegalMoves = 0

  For i = 0 To NumMoves - 1
    RemoveEpPiece
    MakeMove Moves(Ply, i)
    If CheckLegal(Moves(Ply, i)) Then
     Moves(Ply, i).IsLegal = True: lLegalMoves = lLegalMoves + 1
     Debug.Print MoveText(Moves(Ply, i))
    End If
    UnmakeMove Moves(Ply, i)
    ResetEpPiece
    'Debug.Print MovesText(Moves(0, i)), Moves(Ply, i).IsLegal
  Next
  olTotalMoves = NumMoves
  GenerateLegalMoves = lLegalMoves
End Function

Public Sub ShowColToMove()
  With frmChessX.lblColToMove
    If bWhiteToMove Then
      .BackColor = vbWhite
      .ForeColor = vbBlack
      .Caption = Translate("White to move")
    Else
      .BackColor = vbBlack
      .ForeColor = vbWhite
      .Caption = Translate("Black to move")
    End If
  End With
End Sub

Public Sub ShowLastMoveAtBoard()
 If GameMovesCnt = 0 Then Exit Sub
 ShowMove arGameMoves(GameMovesCnt).From, arGameMoves(GameMovesCnt).Target
End Sub

Public Sub ShowMove(From As Long, Target As Long)
 ' show move on board with different backcolor
 Dim Pos As Long, ctrl As Control

 If From > 0 Then
    For Each ctrl In frmChessX.Controls
      Pos = Val("0" & ctrl.Tag)
      If Pos = From Then ctrl.BackColor = &HC0FFC0
    Next ctrl
 End If

 If Target > 0 Then
    For Each ctrl In frmChessX.Controls
      Pos = Val("0" & ctrl.Tag)
      If Pos = Target Then ctrl.BackColor = &HC0FFC0
    Next ctrl
 End If
End Sub
SUB

;===========================================================================================================
; CHESSBRAINVB.INI
; =================
; chess engine ChessBrainVB V3 for winboard and UCI interfaces like Arena GUI. by
```

```
Roger Zuehlsdorf (2018)
; based on LarsenVB by Luca Dormio (http://xoomer.virgilio.it/ludormio/download.htm)
and Faile
; by Adrien M. Regimbald
; and ideas from Stockfish, Protector and other engines.
; Author: Roger Zuehlsdorf (2018)
;=====================================================================================
=======================

;=====================================================================================
=======================
;--- Settings for chess engine
;=====================================================================================
=======================
[Engine]

; if not winboard path set: for ARENA GUI use XBOARD Mode
XBOARD_MODE=1
; in winbord mode show PV line without extra character like x !
WB_PV_IN_UCI=0

; Opening book, empty entry for no book or rename file ( use ARENA main book for
better results )
OPENING_BOOK=
;OPENING_BOOK=
;OPENING_BOOK=CB_BOOK.TXT
; if no book above is found then use a small dummy book for fun (1=aktive, 0 = off,
empty: OfficeMode=1, UCI/WB=0)
USE_INTERNAL_BOOK=
;USE_INTERNAL_BOOK=1
;USE_INTERNAL_BOOK=0

; set number of threads (max 8); 0 or 1: normal single thread, 2-8: multiple processes
 of EXE running
; Winboard "cores" command overrides this setting if cores value > THREADS (UCI
setting overrides too )
THREADS=1
; Threads send from GUI ignored if set
THREADS_IGNORE_GUI=0
;THREADS_IGNORE_GUI=1

; Hash size in MB. (31 bytes per entry) (UCI/WB settings overrides if higher)
; max 1400 MB
HASHSIZE=64
; Hash used value will be overwritten by main task to communicate with helper threads
HASH_USED=64
; Hash size send from GUI ignored if set
HASHSIZE_IGNORE_GUI=0
;HASHSIZE_IGNORE_GUI=1

; Verify Hash reads to handle hash collisions, slows down engine 5%, but avoids some
bad moves
HASH_VERIFY=1

; Overhead for move communication to GUI and start/stop thinking in milliseconds (more
 threads may need more? 1000ms)
; recommended for ARENA: 800ms, for ChessGUI 1000ms, for SMP 1500ms
MOVEOVERHEAD=1000

; Contempt value: draw score in centipawns(100= 1Pawn) from engine view. Against
better engine set positive value
CONTEMPT=1

; UCI/WB-Mode:SYZYGY ; Office(EXCEL): Online probe ): Enabled endgame tablebase access
: 0= disabled, 1 = enabled.
EGTB_ENABLED=1
;EGTB_ENABLED=0

```

```
; Syzygy endgame tablebases
;  PATH = path for Syzygy files ( if path is empty then programs sets EGTB_ENABLED=0 )
;  MAX_PIECES = max piece count for table base probe (3,4,5,6)
;  MAX_PLY = tb access first first x plies only (slow access): 1=root
;  UCI GUI may overwrite this settings; winboard uses this settings here

;TB_SYZYGY_PATH="C:\Chess\TB\Syzygy\Syzygy 3-4-5"
TB_SYZYGY_PATH=""
TB_SYZYGY_MAX_PIECES=5
TB_SYZYGY_MAX_PLY=3

;--- online tablebase access used for Office VBA GUI only
; Endgame table base online Web service.
; First call needs about 15 seconds to init connection.
; Conditions fo ruse of TB: 5 pieces or less on board + minimum 20 seconds time remaining for engine
; Used for PLY=1 only because too slow for deep searches
; Lokasoft Web service is used, 5 pieces
TB_ONL_URL="http://www.lokasoft.nl/tbweb/tbapi.wsdl"


;============================================================================================================
;--- Evaluation of position (factor in percent: 100 = unchanged score, 0 = zero score, 200 = double score)
;============================================================================================================
; Position: piece position evaluation values(i.e. piece square tables)
POSITION_FACTOR=100

; Mobility: mobility of pieces
MOBILITY_FACTOR=100

; Pawn structure: pawn value depending on supported, isolated, backwards,...
PAWNSTRUCT_FACTOR=100

; Passed pawns: passed pawns value depending on rank, safe advance to promote square ,...
PASSEDPAWNS_FACTOR=120
;PASSEDPAWNS_FACTOR=100

; Threats: bonus for threats at opponent pieces depending on piece types
THREATS_FACTOR=100
;THREATS_FACTOR=100

; Opponent king attack: bonus for safe king shelter, penalty for attack options
OPPKINGATT_FACTOR=100
;OPPKINGATT_FACTOR=100

; Computer king defense: bonus for safe king shelter, penalty for attack options
COMPKINGDEF_FACTOR=100
;COMPKINGDEF_FACTOR=100

;============================================================================================================
;--- Piece values (MG: Midgame, EG: Endgame for scaling using game phases) based on Stockfish6 ---
;--- This values are default in engine if entries are missing here
;============================================================================================================
PAWN_VAL_MG=142
PAWN_VAL_EG=207

KNIGHT_VAL_MG=784
KNIGHT_VAL_EG=868

BISHOP_VAL_MG=828
BISHOP_VAL_EG=916
```

```

ROOK_VAL_MG=1286
ROOK_VAL_EG=1378

QUEEN_VAL_MG=2528
QUEEN_VAL_EG=2698

; for game phase calculation
MIDGAME_LIMIT=15258
ENDGAME_LIMIT=3915


;==================================================================================================================
;--- Debug settings
;==================================================================================================================
;enable PV log = 1 ; disable PV log =0 , same as command parameter "-log".
LogPV=0

;Trace file settings: 0 / 1
EVALTRACE=0
TIMETRACE=0
HASHTRACE=0
HASH_COLL_TRACE=0
COMMANDTRACE=0
TBBASE_TRACE=0
THREADTRACE=0

;==================================================================================================================
;---  MS OFFICE GUI settings (not used for winboard engine)
;==================================================================================================================

; Translate for language: DE => ChessBrainVB_Language_DE.txt
LANGUAGE=EN

; Color for GUI board squares
;WHITE_SQ_COLOR ="&H00C0FFFF&"
WHITE_SQ_COLOR = "&HC0FFFF"
;BLACK_SQ_COLOR = "&H0080C0FF&"
BLACK_SQ_COLOR = "&H80FF&"

BOARD_FRAME_COLOR = &H000040C0&

;--- Chess test positions (EXCEL GUI) in FEN(EPD) format (from WAC (Win At Chess)) test set)
TEST_POSITION1 = "1b5k/7P/p1p2np1/2P2p2/PP3P2/4RQ1R/q2r3P/6K1 w - - bm Re8+; id WAC.250;Mate in 8;"
TEST_POSITION2 = "2k4B/bpp1qp2/p1b5/7p/1PN1n1p1/2Pr4/P5PP/R3QR1K b - - bm Ng3+; id WAC.273;"
TEST_POSITION3 = "r3q2r/2p1k1p1/p5p1/1p2Nb2/1P2nB2/P7/2PNQbPP/R2R3K b - - bm Rxh2+; id WAC.266;"
TEST_POSITION4 ="8/6k1/6p1/8/7r/3P1KP1/8/8 w - - 0 1; Tablebase test"
;==================================================================================================================

TB_ROOT_ENABLED=0
TB_SEARCH_ENABLED=0


```