

Demo Items and Introduction to the PGC Format (PGN Compressed)

For suitable test data, here's a [download](#) in **PGC format**. The file contains **1872 games** in the Rauser main line of the **Sicilian dragon, Yugoslav attack [B76/22]**.

This is a [graphical size comparison](#) based on the games in the download (SDYRausr.pgc).

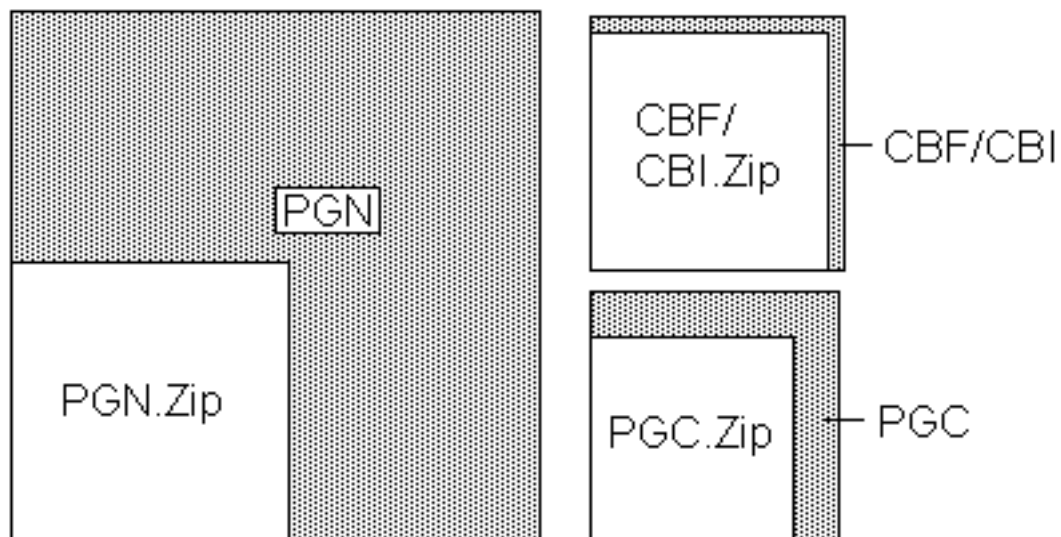
pgc2pgc3 converts v2 PGC to the new Version 3. [Download](#)

PGC compatibility can be implemented by any application which can call a DLL. The DLL (ppkd1v03.dll) does all the conversion of PGN to PGC and back again. Plus it makes the games searchable by header contents or position. The "query builder" in PGNEdit demonstrates the search syntax.

In the hope that PGC can be made a cross-platform format, a [description](#) of the PGC format has been outlined. On-line [documentation](#) for using the DLL has been updated for v3.

In addition, all source for the DLL is being provided in pgcdll.zip ([the download](#) / [the directory](#)). There's a lot there but for someone only interested in using the DLL, a short, 200-line program (exampleA) demonstrates all the basic calls.

Compression ratios shown graphically



Sizes for SDYRausr.pgn

		<i>zipped</i>		
pgn	1,126,602	318,932		
cbf/cbi	267,659	232,510		
pgc	254,222	172,168		
			cbf	260,163
			cbi	7,496

			cbf/cbi	267,659

Description of the PGC (PGN Compressed format)

See [Part 2](#) of this document for PGN tags beginning with White/Black, and those after.

A PGC record can be represented as follows:

- leader #255
- tag marker #254
- notation marker #253
- data

The game presented has the 7 basic tags, however there are only 6 tag markers. That is because Event does not have a marker. It is preceded by...

- Marked/Deleted/Version

...a byte which serves three purposes. The current Version number is 3.

There can be many more tags. All of them would precede the notation marker. Note that some tag markers do not have data (the double). Counting the tags, it can be seen that one such tag is Round. In the PGN game, Round contained only a question mark.

Since PGC is a text file, every record is also followed by #13#10. The leader means, however, that PGC could also occupy a block format.

The method of decoding, starting from the left

Version/Mark

Run the character through a translator based on this set, the **T246Set** set = [[#0..#255] - RestrictSet], so named because it holds 246 items.

The RestrictSet = [#0,TAB,CAR,LNF,#26,#251..#255]

Version/Mark could have been one of the characters in the restricted set so, to make sure it didn't step on one of them, it was translated into the "safe" T246Set.

Building "TranslatorFrom246Set"

Building the translator is a matter of stepping through the characters #1 to #255. Character #0 has filled the array [1..255]...

```
N2=0;
for N=1 to 255 do
  if character(N) in T246Set then
    {
      increment(N2);
      TranslatorFrom246Set[N]=character(N2);
    }
  else
    TranslatorFrom246Set[N]=#0;
```

After the translator is built, the beginning will look like this...

item:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value:	1	2	3	4	5	6	7	8	0	0	9	10	0	11	12	13

Note that since #9, #10, #13 (the tab and control characters) were in the restricted set, they are skipped over.

Now, finally, looking at a Version/Mark character (in the second byte) of a PGC database, we will usually find...

#15

Run #15 (a "safe" character) through the translator and we get #12. If #12 is shifted two bits to the right, the result is,

0000 1100 > 0000 0011 ("3," therefore the version of the pgc record is Version 3)

The two bits shifted out are, respectively, 1-marked, 2-deleted, 3-marked *and* deleted. Since the two bits were both zeros, the game was neither marked nor flagged for deletion..

Event Data

As mentioned before, Event has no marker. It doesn't need one since it starts at a fixed position--the third byte of the game record. Actually, it may not be there at all since if the PGN had [Event "?"] there would be no need to record the data--just put a #254 marker for Site in place of its data.

The Site marker, always present, sets the end boundary for Event. Event data thus ranges in length from zero on up.

Selecting pgc records at random, I came up with quite a few that had Site markers (#254). Finally I came to one with #228 in the third record position. By convention, in dealing with null terminated strings and using the variable **PC**, as I have in the documentation, we are dealing with PC[2]. PC starts counting from the 0-th byte so PC[2] is the same as saying the "third character" from the start of a record.

Convention

PC is (almost) always considered the source field and PC2 the destination field during both packing and unpacking operations. Both are null terminated strings allocated with 50,000 bytes of memory.

Looking forward to the next marker, the length of Event can be determined. It turns out to be 7 characters long:

```
hex          e4 20 4e 4f 52 13  c4
decimal 228 32 78 79 82 19 196
```

Unpacking Event begins with several tests. If the first character were a tab (#9) it would have meant that the field could not be packed since it contained some character out of normal ASCII. Unpacking such a field would then be a simple runout--skipping over the tab and sending the string straight to Event destination. (In the dll code, this would be called an Oh9Runout).

The next tests are for input strings of lengths 1 and 2 (table lookups) since our example is 7 characters long, they'll be covered later.

Now, some progress, the first character (#228) is tested to see if it belongs to the T146Set. The **T146Set** = [T246Set-[ASCIISet]-[#1..#4]] where the ASCIISet = ['..#127]. (The T246Set is covered above.)

The sense of removing characters #1 through #4 from the T146Set is so that they can be used to specify one of four different tables (the actual value used is reduced by one). If no marker is associated with a table item, it defaults to table 0.

#228 is in the T146Set. The test returns True. The next character (#32) is *not* so we know it stands alone. The 146 set is the most restrictive set but for that reason, the most efficient. No table information is necessary as it can be assumed to belong to the default table 0. Table 0 in the Site/Event scheme of things are those items in **cWA** which have a **cTbl** specifier of 0 (**cWA** is in **ppkdtab4.pas** which in turn is in the developers kit in

pgcdll.zip).

(cWA stands for "word array")

#228 is passed through another translator, **TOWOLoc** ("table 0 word locator"), and gives an index value of #379. The 379th item in cWA holds this information:

```
(cWrd:'Open';cTbl:0;cOrd:228),
```

cOrd and cTbl were used in packing. All we needed was the index to get the word "Open."

The translator, **TOWOLoc**, an array [5..250] of integers, is built as follows:

Clear TOWOLoc to zeros first then,

```
for N=1 to WordMax do
  case cWA[N].cTbl of
    {
    0: TOWOLoc[cWA[N].cOrd]=N;
    1: T1WOLoc[cWA[N].cOrd]=N;
    2: T2WOLoc ...
    3: T3 ...
    };
```

(WordMax = 584, the number of items in cWA)

Each "table" within cWA has 146 items (4 times 146 = 584).

Translators are necessary to keep from stepping on character values that are reserved, either permanently or for the immediate purpose. For instance, if encountering an alphanumeric character in a packed record, the assumption can be made that it is unpacked and can be moved directly to the output side. In the most recent case, #228, since it fell in the T146Set, another assumption--that it would translate to a table index--can be made.

Looking at successive characters in the packed field, it can be seen that the next series (32 78 79 82) are all in the ASCIISet. They can be run out directly to the output side. So far, then, we have:

'Open NOR'

Next we come to #19 and it is back in the T146Set. However this time it is followed by a character (#196) which is *also* in the T146Set. This presents a more complicated case as formatting is involved and it can't be assumed that #19 belongs to table 0.

The second byte (#196) must be processed first. It holds the table number, a format, and possible notice of a following trailer.

Here's the process for decoding the second byte ('\$' signifies hexadecimal):

- place #196 in a work byte
- form a trailer Boolean by (work **and** \$01)
- shift work 1 bit right
- compute Format = work **and** \$07;
- shift work right 3 bits
- compute Tbl = work **and** \$03;

The trailer Boolean is false (it would have appended a space to the end) but there is a Format (2). The table *does* turn out to be table 0 after all.

#19 run through the table 0 locator, T0WOLoc, translates to 541. The 541st cWA entry is:

```
(cWrd:'ch';cTbl:0;cOrd:19),
```

Format 2 specifies that whatever is obtained from the table is to be preceded by a hyphen. The combined result is thus,

```
-ch
```

Not much considering the trouble we went through but it does save a byte. Processing of Event is now completed since the string is used up. The final result is,

```
Open NOR-ch
```

It took 7 bytes to represent 11. That might be about average for an abbreviated Event.

Other rules and translations for Event

A packed Event field with a length of one will always be a T146Set character. It will always match to a cWA table 0 item.

Only a 2-length item can have two table lookup bytes together. In other cases of two T146Set bytes together, the second will hold formatting information (and table number).

Fields with lengths of two or more have a number of combinations. The first character could be #9 (covered earlier), or also a series of ASCIISet characters (runout). That leaves the first being a T146Set character--or a new introduction--a table byte *followed* by a T146Set character. These are situations that could not benefit from formatting (recall that in an earlier example, the table was in the second byte of a pair encoded).

The table number is reduced by one, giving a value of 0-3 which is then used to decode the second byte. The translator used (T0WOLoc - T3WOLoc) should correspond to the table number.

List of decoded Formats

```
0 - none
1 - ' '
2 - '- '
3 - ', '
4 - '. '
5 - '( '
6 - ') '
7 - '), '

```

Finally there can be the "modifier" byte (#252) encountered. This signals that all characters following, so long as they stay in the T146PASET, can be processed in series. **T146PASET** stands for "T146Set + ASCIISet" and that is its definition. Such runs are packed suffixes, ASCIISet characters, or a combination of both. The test made is whether a character is ASCII. If so, it goes straight to the destination. Others will translate to indexes in the suffix table, **cSA** ("suffix array"). The translator is **T0SLoc** ("table 0 suffix locator"). There is only table 0, or more correctly, the table does not have the cTbl field.

T0SLoc is constructed similarly to the other examples:

```
for N=1 to SfxMax do //SfxMax is 146
  T0SLoc[cSA[N].cOrd]=N;
```

Site Data

Site data is processed identically to Event. The only difference is that it does not start at a fixed position in a pgc record. Rather, search for the first #254 character and the Site data will follow--on up until the next #254 marker (for Date).

Date Data

Date will follow the second #254 marker. Except for the tab runout rule, all dates are length encoded. In the first case, a date with zero-length is '????.??.??'.

Length equal 1 - a date that had only the year specified. The character is run through the **TranslatorFrom246Set** as covered in the Version/Mark example. (In program code the translator is referred to as **TxfFm246**.) Add 1877 to the ordinal value of the translated character to give the year.

As an example, #100 run through the translator is #96. Adding 1877 to the ordinal gives 1973. Format as '1973.??.??'.

Length equal 2 - move the two bytes directly to a 16-bit signed integer field (a memory transfer, *not* a numeric assignment). Compute the date components as follows:

```

                                #134#121 -> 31110
Day=Year and $001f;           6
Year=Year shr 5;              972
Year=Year and $07ff;          972
Mon=Year and $000f;           12
Year=Year shr 4;              60
Year=Year and $0fff;          60
Year=Year+1937;               1997 ( '1997.12.06' )

```

Since two character dates are not translated, not all combinations are valid. The packing algorithm will have seen to using other rules when it discovers that a reserved character would be stepped on.

Length equal 3 - The first two bytes are also moved to a 16-bit signed integer field as in the previous example but in this case give a direct conversion to Year. The third byte holds the Mon. Format as, e.g., '1999.02.??'.

Length equal 4 - Four bytes will hold the actual character value of Year. Format as, e.g., '1792.??.??'.

Length equal 8 - These are just slightly compressed dates that need only the insertion of two periods. Example: '19970731' -> '1997.07.31'. Chances are it is a date that contained a reserved character when it was trial-tested using the rule for Length equal 2.

Except for the the tab rule there are no other lengths to consider.

Round Data

Round will follow the third #254 marker. In the case of zero-length (Round Marker followed immediately by the White marker) the output value is a single question mark (?).

If data is present, the first test is on the first character for #9 (tab runout rule) or any ASCIISet character (straight runout). All other contents are coded--having a length of either 1 or 2.

Length equal 1 - The character is considered by its ordinal:

```

                                #16 = $10
Fmt=(byte and $30) shr 4;      $1
Val=byte and $0f;             $0
Val=Val+1;                     $1

```

Format will never decode to 0. Of the three formats (1-3) the presentation of the Round value is:

- (1) - 01 (possible values "01" - "16")
- (2) - 0.1 ("0.1" - "1.6")
- (3) - 1.0 ("1.0" - "16.0" where the "0" is an appended literal and will always be "0")

Length equal 2 - These have either a flag (#3,#4) in the first character position or are both **T150Set** characters. The set, not previously covered, is the T246Set less the ASCIISet. Here is an example of two packed characters, both in the T150Set, being decoded:

data - #11,#1

#11 run through the translator give #9 (which, had it not been translated, would have been the unwanted tab character)

#1 translates to itself

result: 9.1 (the two value are concatenated around a period)

First character flag #3

second character - #22 (example)

#22 run through the translator gives #19

literal ".0" appended, giving 22.0

First character flag #4 (these are somewhat rare)

second character - #203

#203 run through the translator give #103

result: 103

There is no advantage in coding Rounds like "1" or "17" so they are straight ASCII runouts.

White/Black Data

(This description continues in [Part 2](#))

This is Part 2 of the PGC description ([Part 1](#))

White/Black Data

Data for the player with the white pieces will follow the fourth #254 marker and that of black, the fifth marker. There is no concept of an empty field for players as data will always be present. This was likely an artifact of design since player compression routines were the first ones written. In fact, it turns out there is a penalty for omitting a player's name as the "?" will be a tab runout (two characters). Such games are very rare.

Player names introduce two new tables and locator sets, Table **cNA** has index locators **T0NLoc** through **T3NLoc**. The table (not to be confused with tables cNA1 and cNA2) is the "names" table. The "locators" are name index locators. **cNA** will be found in **ppkdtab2.pas** in the developer's kit, **pgcdll.zip**. The locators are built similarly to those previously presented. Here is the current process:

```
for N=1 to NameMax do // 872 items
  with cNA[N] do
  {
    case cTbl of
      0: T0NLoc[cOrd]=N;
      1: T1NLoc[cOrd]=N;
      2: T2NLoc[cOrd]=N;
      3: T3NLoc[cOrd]=N;
    }
  };
```

The exact same thing, but using table **cnf** (**ppkdtab3.pas**), "Name First," and forming **T0NFLoc** through **T3NFLoc** is also done prior to any name field unpacking. **cnf** also has 872 items. Incidentally, **cNA** has both last and first names while **cnf** has only first names.

The "sub-table" field, **cTbl**, will contain values 0-3. There will be 146 table items with value 0, and 246 items represented by each of the values 1 through 3. ($146 + 3 * 246 = 872$).

Tests must be followed in order. The tab runout rule, mentioned briefly above, is always first.

Beginning an example, in order, data lengths of one and two represent length-encoded items.

Length equal 1 - in this example, the character is #11. Length-one items will always be **cTbl** 0, using locator **T0NLoc** to table **cNA** only. The character #11 run through the locator gives an index of 368. Item 368 in cNA is "**Keres Paul**".

Here is the program statement that formed the index: `NDX=T0NLoc[ord(D1[0])]`; The string pointer D1 is looking at the first character past a #254 marker.

It will be noted that names in cNA are not formatted with commas. The source of our example was PGN that did not use a comma either. Unformatted names are the most efficient although unlikely to be preferred by many. Still, all formatting can be handled. It just requires more than a one-byte item.

Length equal 2 - these items have several possibilities. The first character can be the modifier flag #252, meaning that the second character has only "alpha" packing (*not a table process and covered farther on*). Otherwise the first character translates to a table index and the second contains length and **cTbl** information. Here's an example:

```
#202,#79 - data (cTbl must be determined first, the 2nd character)
#79 treated as its ordinal is $4f (hexidecimal)
```

```
Len=$4f and $1f; (Len will be 15)
```

```
Tbl=$4f shr 6; (cTbl 1)
```

```
NDX=T1NLoc[ord(#202)]; (using T1NLoc because Tbl was 1)
```

NDX will be 510. Length-two data uses cNA only. The 510th item is "Miezis Normunds". *Len* is 15, the full length of the name. The decoded player name is thus **Miezis Normunds**. Again, except for a possible length reduction, say in the case of first initial only, there was no formatting involved. Since the name was used in full, it might seem that "Miezis Normunds" could have been encoded in a single byte, but recall that one-byte data always uses cTbl 0.

Lengths 3 and above

Data, not intercepted by tests so far, now has a test for an exception. If the first character is #4, then it involves no table at all but rather has been packed as a "suffix." (Suffix decoding is described further down.)

Coming to this point, all name fields will have a format byte and have a length of three or more. The three beginning characters have this significance:

- (1) - a format, or "Map" as it is called in the code
- (2) - a table ordinal to be converted to an index
- (3) - a combination length and table identifier

The third byte it decoded the same as in a previous example,

```
Len:=aByte and $1f;
Tbl:=aByte shr 6;
```

We'll consider this example,

```
#32 #170 #131 #112 #20 #87 (data length of 6 characters)
$20 170 $83 'p' 20 'W' data as interpreted
```

The second byte, 170, is assigned to variable *tOrd*.

It can be seen then that *Len* will equal 3 and *Tbl*, 2. Using cTbl 2 items in **cNA** the index is derived, $NDX=T2NLoc[tOrd]$, or $NDX=307$. Item 307 in cNA is "Hillarp Persson Tiger". Only the first three characters of the name will be used, giving "Hil".

The next character, the fourth, will be one of two possible types. Were it not a **T146PASET** character, it would be a flag with a value of #1 or #2. (Flags will be treated farther on. They can also stop suffix runouts.) T146PASET characters begin suffixes (and were treated towards the bottom of **Event**). Applying the rule for suffixes, the data string is used up. We obtain,

```
'p'      20      'W'
'p' + 'mann' + 'W'
```

(20 was an index to the suffix table, **cSA**.)

Before we had "Hil". Now we append "pmannW". The accumulated result is, "HilpmannW".

A short while ago we saved a format, or more specifically, a "Map," taken as the value of the first data byte. It was #32 or \$20. The Map is used to format the name developed,

```
map and $78 gives $20 (unchanged)
```

\$20 is the label of a "case" statement. Two procedures are called, ScanSeparate and CommaOver. Procedure "ScanSeparate" is passed the accumulated name and, starting from the second character ("ilpmannW"), looks

for the first capital letter and inserts a space ahead of it. CommaOver looks for the first blank and puts a comma in its place.

The result is "**Hilpmann,W**". It took six characters to generate ten. Since three coded characters were used to generate "Hil," it might seem that a simpler ASCII/suffix rule would have been equivalent, but recall that the "free" format came along and included the comma.

Flag bytes

#1 - interrupts a suffix runout to signal a first name table lookup (full match)

#2 - interrupts a suffix runout do signal a partial first name lookup

#3 - not used

#4 - simple flag - when found in the first data byte, means the entire name has been suffix encoded

#252 - simple flag - when found in the first data byte, means the entire name has been 6-bit "Alpha" encoded

Summary

Typical structures of packed names are,

[last name table lookup] [suffix] [#2] [first name table lookup] [suffix]

[last name table lookup] [suffix] [#1] [first name table lookup]

[last name table lookup] [#2] [first name table lookup] [suffix]

[last name table lookup] [#1] [first name table lookup]

[last name table lookup] [suffix]

[last name table lookup]

Table lookups require three bytes if partial or two bytes if full (the last name table is more correctly a last/first name table).

(to be continued)

```
RestrictSet = [#0,TAB,CAR,LNF,#26,fb251..ff255];  
XXXSet     = RestrictSet-[TAB];  
ASCIISet   = [''..'#127];  
T246Set    = [#0..#255]-RestrictSet;  
T150Set    = T246Set-ASCIISet;  
T146Set    = T150Set-[#1..#4];  
T146PASET  = T146Set+ASCIISet; {T146 Plus ASCII}  
{AllCapSet = ['A'..'Z']; }
```

Creating and Accessing a PGC database by calls made to [ppkdlv03.dll](#) from the C Language

Note: Version 3 revisions are shown in this color.

There are several dozen functions in [ppkdlv03.dll](#) but, by using only five of them, one can access a PGC database quite easily. PGNEdit, for example, uses only five basic calls which are outlined herein.

Use of the DLL requires these components:

- Your calling program
- [ppkdlv03.dll](#)
- [ppkdlv03.dat](#)
- PGC database [...**or**...]
- PGN games

The DLL & DAT file can be found in [pgcdll.zip](#) ([pgc developers kit](#)) at this site.

In one instance, the DLL deals with a local (short) filename only (no path) so it expects to find [ppkdlv03.dat](#) in its own directory. Full paths can be included and passed for PGN files and PGC databases

The DLL can be located in a remote directory. If so, change the *current directory* to its location before it is loaded. It can be changed back to another directory after use.

The "packing" and "unpacking" routines discussed here work on a game-by-game basis. As an example, the calling program would read a PGN game into memory, call [GamePackPGN](#), then write the output to a PGC file. The reverse would be reading one record from a PGC file, converting with [GameUnpkPGN](#), then writing the output to PGN.

Two searching functions are more automatic and will open, process, and close the PGC database without help. However, the first of these, [FileQuerySel](#), only returns an array of game numbers selected. It is up to the calling program to then open the PGC file, read records, and match these to the game numbers. Disposition is then whatever the program decides. Normally, they would be converted to PGN and written to a file.

[FilePosSearch](#) works more easily. The calling program need never access the database directly. The games selected will be written to an output file, in PGN.

Finally--an important step--the general process for [loading the DLL](#) and [freeing](#) its memory when through.

I've now added program code for [loading the DLL from Delphi Pascal](#). It may help C programmers too.

If you run into trouble with any of the documentation--or implementation--please contact me at ponstad@visi.com rather than the address on my home page as I check mail there much less frequently. If my C syntax has broken down in places, you can probably guess my intention and your interpretation will likely be the better choice.

Good Luck,

-Paul Onstad

[ps: the new Tag array](#) *revised*

GamePackPGN (GAMEPACKPGN_C)

The **GamePackPGN** function (one you describe as a template) converts a PGN game to a compressed PGC line.

DWORD GamePackPGN(

```

LPHANDLE myhdl // maintained by DLL
LPSTR pc // PGN game string (input)
LPSTR pc2 // PGC game string (output)
);

```

Parameters

myhdl

It should be initialized to 0 (once) after the DLL is loaded. Thereafter it is maintained by the DLL. Subsequent DLL calls should not reinitialize unless the DLL is unloaded and then reloaded.

pc

A null terminated string representing a PGN game. All internal formatting (carriage returns and line feeds) should be present but both ends should be trimmed of any whitespace characters.

pc2

A null terminated string (returned) with the game packed in PGC format, suitable to be written as one line in a PGC text file. Also the error trap code if the function fails.

Return Values

If the function succeeds, the return value is 0x00000001. If it fails, the return is 0 (Pascal LongBool). An error message (trap code) will be passed back in *pc2* in the case of failure.

([TrapSum.txt](#) in [pgcdll.zip](#) has a list of common error trap codes.)

Remarks

Use **GamePackPGN** to create a PGC file on a record by record basis. When creating PGC, specify the file type as **text**.

GameUnpkPGN (GAMEUNPKPGN_D)

The **GameUnpkPGN** function (one you describe as a template) converts a compressed game string to PGN.

DWORD GameUnpkPGN(

```

LPHANDLE myhdl // maintained by DLL
DWORD flag // game style
LPSTR pc // packed game string (input)
LPSTR pc2 // PGN game string (output)
);
```

Parameters

myhdl
It should be initialized to 0 (once) after the DLL is loaded. Thereafter it is maintained by the DLL. Subsequent DLL calls should not reinitialize unless the DLL is unloaded and then reloaded.

flag **revised**
One of two FMT flags: **FMT_Closed** (decimal 0) or **FMT_Verbose** (decimal 16). The flag determines whether move numbers in output chess notation have period,space or not.

pc
A null terminated string representing one line read from a PGC file.

pc2
A null terminated string (returned) with the unpacked PGN game (fully formatted with carriage returns and line codes).

Return Values

If the function succeeds, the return value is 0x00000001. If it fails, the return is 0 (Pascal LongBool). An error message (trap code) will be passed back in *pc2* in the case of failure.

([TrapSum.txt](#) in [pgcdll.zip](#) has a list of common error trap codes.)

Remarks

Use **GameUnpkPGN** when your program reads a PGC file directly. PGC is a text file and can be read like any text file. A read loop would typically read the next line of text, place it in *pc*, call **GameUnpkPGN**, and then retrieve the PGN game from *pc2*.

FileQuerySel (FILEQUERYSL_F)

The **FileQuerySel** function (one you describe as a template) returns an array of game numbers which match a formal query (a search statement matching to data contents of PGN headers).

DWORD FileQuerySel(

```

LPHANDLE myhdl // maintained by DLL
LPTSTR infn // filename of the PGC database
LPSTR query // formal query statement
LPSTR as64 // error return string
DWORD sgnum // "start at" game number, usually one (1)
DWORD range // use default (0) for full range
LPLIARY liary // pointer to array of dword *revised*

```

);

Parameters

myhdl

It should be initialized to 0 (once) after the DLL is loaded. Thereafter it is maintained by the DLL. Subsequent DLL calls should not reinitialize unless the DLL is unloaded and then reloaded.

infn

A null terminated string containing the path and filename of a PGC file.

query

A null terminated string containing a formal query statement (see several examples in Remarks below).

revised [requirement specifying starting asterisk (*) dropped]

range

A range of 0 specifies the entire extent of the base, including *sgnum*. A specified range will limit the search provided $sgnum + range - 1$ is less than the total games in the base.

liary **revised**

a pointer to an dword array of any size. By convention, an array of 16001 elements has special significance since it can be specified by setting array member [0] to 0. This results in a maximum of 16,000 games selected. All other sizes should set member [0] = [total array elements - 1]. In addition, to avoid accidental memory corruption, maximum games must be in even increments of 1000 if greater than 16,000. The sense of 16,000 is that it provides a reasonable number of games were a user to make a selection that matched to every base record (the games are being written to PGN). Even increments of 1000 guard against failure to initialize member [0]. The 16,000 array type is also specified in ppkType3.pas allowing convenient memory allocation with "New(liary)" instead of "GetMem(liary,size?)".

On return, element [0] will contain the number of games selected. In turn, game numbers will have been filled in, sequentially, in element [1] on up.

Return Values

If the function succeeds, the return value is 0x00000001. If it fails, the return is 0 (Pascal LongBool). An error message (trap code) will be passed back in *as64* in the case of failure.

([Trapsum.txt](#) in [pgcdll.zip](#) has a list of common error trap codes.)

Remarks

A description of the query language can be found in the [Latest Updates](#) page of this web site. The "relaxed syntax" cannot be used with the DLL; it must be the full, formal syntax:

(whi con :kasp,anand) and (bla con :kasp,anand)

Note: the contents of *query* are no longer changed during processing and the requirement for a leading asterisk has been dropped.

Other examples:

(eco begs :A28,A29)
(bla cons and:J,Polgar)

FileQuerySel automatically opens, reads, and closes the PGC database. However, it only returns game numbers so the calling program will want to open the PGC file after the call is complete to process the selection--usually by writing the games to PGN. To do this, read the database sequentially (a text file), matching to the game numbers in *liary*. On each match, call **GameUnpkPGN** to convert the packed game into PGN. An alternative is to use [FILEUNPKLIA_G](#), a file to file call (see [ppkd1v03.pas](#)).

Latest Updates

Directory View	PGNEdit	NORMAL32
(Ya)hooScore	Slinger	PGNSite
ECOClass updated	the Source tag	high ASCII
PgnEdit Query Language	Fuzzies	Version Info
		Other Programs

All chess programs are Win95 and up. To download a utility, note the name then choose "Directory View." Once in the directory, click the desired program name to start downloading. (The other table entries link to information in *this* document.)

If a program you're looking for is not at my site, there's a good chance Michael Sharpe will have it at pgn.freeservers. If not, reference the "Other Programs" link in the table above.

The following descriptions have a certain ring of File_ID.diz-speak since that's where I took them. However I've often added my own comments below.

-Paul

PGNEdit v2.1 Chess editor, viewer, and database. Select, edit, play through, and archive games that you collect. Games are viewed in large word-processor window for easy access--copy & paste. Save in compact database format to allow quick search and retrieval (each base can hold hundreds of thousands of games), or save as PGN. Export games to, and launch, your favorite chess programs. Multiple chessboards to compare two or more games.

Version 2.1 consolidate recent changes to the PGC (PGN Compressed format). It has a much easier-to-use database functions.

With directory setting for database components, particularly the location of the DLL, PGNEdit can now share the same DLL and database with Slinger and ECOClass, making it convenient to move all databases to their own, private directory.

Features

- PGC database size unlimited
- fast, versatile database searches
- easy addition of games
- no practical limit to PGN file size
- word processor view of games, easy access and edit
- *Find List* summarizes game contents in seconds

- game cleaning *macros*
- mark games by any selection and save/split to separate file(s)
- click any game then *Export* to have it opened, ready for play by your favorite chess program
- many styles and sizes of chessboards available
- open two or more boards to compare games

[\(top\)](#)

PgnEdit Query Language

(Note: new versions of PGNEdit and Slinger have made the generation of query statements automatic. This section is only necessary as a background for understanding the construction of the statements.)

Most databases present indexes of the various fields (PGN tag contents) for searching. This is fine but does present some hurdles. What if the data you are interested in is not the left-most part of the field? The query statement script gets by these limitations by looking for keyword contents in any portion of a field and in any field. It is thus able, for instance, to find a particular player even if some games had the player's name as "Last, First" and others, "First Last."

An example; this query would find all games where Judit Polgar had the white pieces--but would exclude her sisters:

```
(whi cons and:J,Polgar)
```

Nor would it matter if the White header contained -- J. Polgar / Polgar, J / Polgar, Judit.

The "and" in the query is saying that a White header must contain both "J" and "Polgar." The "cons" comparitor stands for "contains." The "s" on the end further qualifies the search as case sensitive (it could have been "con" -- case insensitive).

Query Examples

```
(white beg :Kasparov)
(whi beg :Kasp) and (bla beg :Karp)
(site beg not:cr)
(site con and:Wijk,Hoogovens)
(event equ : "It Open")
(eco beg :A28,A29)
```

A query statement searches for games by header contents. Each segment of a query has these components:

tag	comparison operator	condition:data
-----	-----	-----
white	beg	:Kramnik
black	begs	or:Polgar
etc.	con	and:J,Polgar
	cons	not:C55,C56
	equ	

equs
 pre
 pres

All components except data must be lower case. Tags can be shortened to the first three letters. Comparison operators are abbreviations for "begins" "contains" "equals" or "present".. If the operator ends in "s" the comparison is case sensitive.

Conditions can be "or" "and" or "not". If condition is omitted, it is the same as "or". To say that tag White must contain "and:J,Polgar" is to say the contents of the White header must contain both "J" _and_ "Polgar". The order is not important.

Each query segment must be enclosed in parenthesis. Multiple segments can be associated with "and" and "or". Segments can also be nested.

The data list: multiple items in a list are separated with commas. When a list item contains a blank, colon, or a comma, it should be enclosed in quotes. Examples:

(whi con : " ", ", ")
 (black begs : "Kramnik,V")

The "present" operator has no data list:

(whiteelo pre :)
 (eco pre not:)

Nesting: nesting can be eight levels deep. Since the syntax of the query requires a first level of parenthesis, the effective level is seven. Segments can only be associated with "and" and "or" (not "not").

((seg) and (seg)) or ((seg) and (seg))
 (((seg) or (seg)) and ((seg) or (seg))) or (seg)

To shorten long queries, numbers can be used instead of tag names (see ppktypes.pas in Extras.zip).

Incidentally, ppkdlv03.dll, used by PGNEdit, features "short-circuit" Boolean evaluation. It is thus advantageous to place the most restrictive "and" conditions--and the most inclusive "or" conditions farthest to the left.

Relaxed query syntax (PGNEdit only)

Simple queries, easily the majority, can now be entered with a "natural language" command. Such a statement does not require the enclosing parenthesis, attention to case, nor the semi-colon.

Where, before, one might have entered:

(bla con :Kramnik)

...this new equivalent would do the same thing...

Black contains Kramnik

Simple lists can make up the third "word" ...

ECO begins A45,A46

Follow the *three words, two spaces rule*: Keep the last "word" joined (even though it's actually two items there should be no spaces after commas). No case sensitivity is involved--not in the query itself nor the data matched.

[\(top\)](#)

The Source tag

The PGN Source tag is just an extension of other PGN tags--such as Event and Site. It is used to denote the source of games one has acquired. For example, if I have downloaded the TWIC series 337 game file and saved it in my .pgc database with PGNEdit, I might want to retrieve them later. It is NORMAL32 which creates the tags (Options|Set Source). "Normal" will ask that I enter a code and a description. The description is saved to an accumulating log file (along with the code) but the code itself becomes the content of the Source tag. For example:

```
[Source "TWIC337"]
```

Eight characters or less with uppercase alphabetic characters and digits 0-9 are recommended for source codes since the .pgc format knows how to compress them better.

Although I may, I really don't have to remember the codes since they are in Normal's log file. There too will be the long description and the date the games were processed by Normal. Should I wish then, sometime later, to retrieve these games, only, from my database, I would do so with a PGNEdit Query:

```
source contains TWIC337
```

[\(top\)](#)

High ASCII

High ASCII characters are those requiring eight-bit representation (versus seven) but more generally thought of as the higher range of printable characters, and which are encoded by decimal values from 128 to 255. In contrast, the English alphabet has values from 65 - 90 (CAPITALS) and 97 - 122 (lowercase).

At one time this made a big difference since, if just a single high ASCII character appeared in a document, it had to be uuencoded before transmission--making the document much larger.

PGN does not support high ASCII but that has not stopped such characters from appearing in game scores, e.g., recent correspondence games from IECC will have player names such as "Michèle."

The situation is not new since even when all chess databases ran in DOS there were similar examples. One problem of course is that during the transition from DOS to Windows everything changed. Example:

```
[Site "Z,,hringen-Erlangen"]
```

If you were to open a file with a DOS editor and find the line above it would appear:

[Site "Zähringen-Erlangen"]

So what does this have to do with U4Chess? Well, it continues to support the PGN standard but has had to acknowledge the reality of games that exist. Anyone who ran "Normal" was likely irked by its warning of "Header Conversion Skipped" whenever encountering a file with many high ASCII headers. You could translate the headers with PGNEdit but that took time. In any case, there have been two changes: NORMAL32 now has a checkbox which, if checked, allows games to be processed without the warning; PGNEdit now automatically translates the old DOS characters to their nearest English equivalent.

(At least enough of them to get through the 2150 game SMORRA.pgn file without leaving any. There will be other such characters popping up in other older files. Pass along any examples and I'll add them to the translator.)

In NORMAL32 the new checkbox can be found in the "attic" via Options|Configuration, then a click on the "More" button.

In PGNEdit the automatic translation, file-wide, is invoked by a new macro (it's the last one in the list).
[\(top\)](#)

NORMAL32 v1.6.1 PGN Normalizer. Produce quality PGN from most any chess notation. Proof scores. Used by archivists including TWIC & ICCF. Combine files and kill dups--including fuzzy dups. Prepare games for your database. Shareware with no nags, restrictions, or time-outs from U4Chess.

Note: NORMAL32.zip may not be at this site except immediately after new updates. If it doesn't appear in the "Directory View," [download](#) at pgn.freesevers.

Version 1.5 advanced to 1.6 mainly for the testing/debugging it received when faced with a lot of Yahoo scores (coordinate notation). Ambiguous moves--two rooks or knights on the same file and capable of reaching the same square--had to be fixed up. Some prior work had however gone into long games--specifically annotated ones that exceeded 5K. Much larger games are now permitted (expecting one at 65K any day now :)

This might be a good place to point out the "Normal" interprets games differently from most PGN processors. In fact, it doesn't need PGN at all since, in its least restrictive interpretation, it sees scores as nothing more than free-form text preceding move 1....and notation from there on until a legal "result" is encountered. This makes it possible to proof most all styles of chess headers and notation (coordinate, long, abbreviated....all but descriptive). If it encounters the pre-PGN standard of two-line headers (the players separated by "-" and a following tournament/date line), and if told to do so, it will convert the headers to PGN. Normal *always* reads and plays out each move of a game--making sure the score follows the rules of chess. It will warn of errors--and where possible (when set by option) will automatically correct minor errors. A very common one in many downloads (processed by Ca2pgn.exe) are checks (+) on the King that are not really checks. Normal will correct these if its "ignore checks" option is turned on.

[\(top\)](#)

ECOClass v1.3.2 ECO ("Encyclopedia of Chess Openings") Classification of chess games and copy/paste PGN reader. Assign ECO codes and, optionally, opening descriptions to games in PGN format. Paste most any notation to have it classified. Application has two chess boards. Classifies large

files in seconds. Assignments based on text file containing over 3400 opening lines (ECOMast) and is fully customizable. Many options, including base retrieval of games at the point in which they leave "book."

As mentioned, ECOClass has turned out to be a fairly versatile chess reader in addition to being a classifier. Chess analysis from web sites can be copied, comments and all, and quickly formatted to play. Odd notation doesn't usually faze it.

Version 1.3 added an option for the classification history of a line:

```
D12 QGD Slav: 4.e3 Bf5, ply depth 8
d4 d5 c4 c6 Nf3 Nf6 e3 Bf5
```

```
1.d4           {A40 Queen's pawn game}
1...d5        {D00 Queen's pawn game}
2.c4          {D06 Queen's gambit}
2...c6        {D10 QGD Slav defence}
3.Nf3         {D11 QGD Slav: 3.Nf3}
3...Nf6       {D11/11 QGD Slav: 4.e3} {4.e3}
4.e3          {D11/11 QGD Slav: 4.e3}
4...Bf5       {D12 QGD Slav: 4.e3 Bf5}           {5.;Nc3}
```

You can [download ECOClass here](#) (or by entering "Directory View" at top)

[\(top\)](#)

(Ya)**hooScore v1.0** Converter to PGN. Converts native Yahoo game scores to PGN via simple copy/paste while archiving them to a single file. Process with NORMAL32.exe once games are in PGN format. Normal will convert the notation as well. Freeware.

hooScore came about (had trouble with the name) after visiting Yahoo's on-line chess gaming site. There, the standard is not PGN but the format just below on the left. hooScore converts these headers to PGN (right):

before

```
;Title: Yahoo! Chess Game
;White: cliff98_uk
;Black: usn_ship
;Date: Sun Feb 25 03:33:19 PST 2001
```

after

```
[Event "?"]
[Site "Yahoo! Chess Game"]
[Date "2001.02.25"]
[Round "?"]
[White "cliff98_uk"]
[Black "usn_ship"]
[Result "0-1"]
```

The games are accumulated to a file so [Normal32](#) will be able to process whatever has been converted on a single run. Normal converts the coordinate notation (1. e2-e4 c7-c5 2. g1-f3) to PGN standard (1.e4 c5 2.Nf3).

Yahoo scores don't contain a Result but if the game ended in mate, hooScore will automatically assign

the proper code--*provided* the Result "radio button" is left as "unknown." For other games, if the result is known prior to translation, select the appropriate radio button.

Often, when many games are played, the result will not be recalled until after the game is converted and viewed on a games viewer (pgnedit--hint, hint :). In such cases, leave the result as unknown and edit it in later (don't forget, in two places).

Note: hooScore has no "About" and thus no version display but the internal version can be found by right-clicking the exe, selecting "Properties," and then the "Version" tab. Versions 1.0.0.9 and higher should be free of the bug that caused a crash on a large, second paste (hooScore now clears its window before any paste). Also, the first version of the program would not recognize the start of a Yahoo score if it was the very first line. This has also been fixed.

[\(top\)](#)

PGNSite v2.0 Recent addition. Quick little utility to customize the contents of your game headers. Get them the way you want them to look. Handles files of up to 100,000 games. Freeware.

PGNSite determines what changes it should make to the contents of Event, Site, and Round by reading a text table. Two such tables come with the download; one is compete and the other is a skeleton example which can more easily be customized.

Everyone has probably noticed that scores get clobbered over time. One of the most irksome header problems (for me at least) are those scores that begin "it op" (variations of) which is followed by the city or tournament name. In a chess database this makes reassembling various events extremely difficult. It also leads to more indexes and thus, size inefficiency. In any case, this is just one condition that PGNSite can correct if told to do so, e.g., move the "it op" to the end of the line. It also makes hundreds of other corrections of course. It standardizes names, drops meaningless data, left-zeroes Round, and establishes common abbreviations. It makes no changes that are not under user control.

Note: for standardizing player names, the equivalent program is HighRank (still in 16 bit).

[\(top\)](#)

Slinger v1.4.2 Freeware. This program has replaced the old MBase32 (implementations of MultiBase). It maintains chess databases in PGC (PGN Compressed) format such as those created by PGNEdit. While PGNEdit is able to save and append to PGC, it has no features to sort bases, transfer data, or eliminate duplicate games. Enter Slinger, which serves all these functions. Slinger is also able to access base games by game number and ranges of game numbers.

Version 1.4.2 has added "Selection date summary" to the Action List. This item takes the current selection and summarizes game dates and player information.

Since Slinger sees little difference between PGN and PGC--and is very fast in exchanging games between the two formats--it can also be thought of as a maintenance tool for large quantities of PGN....as might be created by other databases. Slinger has some functions that other bases are unlikely to have but if they are able to create PGN, the special features of Slinger can be used as well. Base sampling, easy sorting, and the elimination of fuzzy duplicates are examples of these special features (see the "Fuzzy Duplicate" section right below)..

Incidentally, PGC has an unlimited static index so games can be reorganized without the constant fear of corruption (which is still a problem in both the leading chess databases).

[\(top\)](#)

Fuzzy Duplicates in Chess Databases

"Fuzzies" is the term I find most appropriate to identify chess scores whereby a single game is given multiple interpretations. There are many reasons why this happens but no matter how it occurs, once a fuzzy becomes a fuzzy, they're hard to track down and get rid of.

Do you have a fuzzy problem? :) You won't know unless you look so here's one way to find out. Take a healthy sample from your database representing all games of a particular ECO (or ECO range) and write to PGN. Convert this file to a PGC database with either Slinger or PGNEdit. A selection of games which transpose (English opening) will turn up more fuzzies than fixed openings (King's gambit).

Now, with Slinger's "Action" menu, select and run "Sort fuzzy macro C." This creates a checkpoint PGN file (cpAside.pgn) which contains the fuzzy candidates. Meanwhile, it marks these candidate games as "deleted" in the PGC file. NORMAL32 is used for the actual presentation and deletion of the fuzzy games in cpAside.pgn:

- start Normal32
- Options|PGNSort|Macro C
- click the large "Sort" button
- now click "PGNSort"

If you end up overwhelmed by the number of fuzzies in your database, the only recourse might be to use the "Random rule" which is fully automatic--at the first PGNSort prompt, select Options (o) then choose the random rule. Of the rules, only "random" is fully automatic. It's unbiased but it makes no effort to decide which game of two (or more) is the "better" one. The temptation will be to go with a better rule (the Combination rule) but if you spend 10 minutes with it, resolving 50 fuzzies and still have 50,000 to go, you'll never finish. (To start the automatic run, Enter "a".)

Some fuzzies are legitimate games--just very similar. Here's an unusual example:

Correspondence games will often be found following the moves of an earlier game--often those of GMs--but here we have a situation of two games merging together and moving apart four separate times--finally taking individual paths at move 25. Was it just a coincidence, or was the 1996 game used as a model? If it was, it appears to be an abandoned game. White was not winning with 32.Qd6.

That sort of thing can happen with tree-searching...following $5+2=1-$ down to $1+0=0-$ and then discovering the last game has a bogus result. White did win in any case so who's to say it was an error?

BTW, while PGNSort sees the two games as fuzzies it will also note that the players are different (except with the random rule) so there is no automatic selection of one over the other.

[Event "corr ClassM.010"]

[Site "IECG"]

[Date "1996.?.?.?"]

[Round "?"]
 [White "Duriez,Jean Luc"]
 [Black "Albarran,Gustavo"]
 [Result "1-0"]
 [ECO "D31"]
 [WhiteElo "2274"]
 [BlackElo "2095"]

1.d4 d5 2.c4 e6 3.Nc3 c6 4.Nf3 dxc4 5.a4 Bb4 6.Bd2 a5 7.e3 b5 8.axb5 Bxc3
 9.Bxc3 cxb5 10.b3 Bb7 11.bxc4 b4 12.Bb2 Nf6 13.Bd3 Nbd7 14.Qc2 Qc7 15.e4 e5
 16.O-O O-O 17.Rfe1 h6 18.c5 exd4 19.Bxd4 Rfe8 20.e5 Nd5 21.e6 Rxe6 22.Rxe6
 fxe6 23.Re1 Nf4 24.Be4 Rc8 25.Ne5 Bxe4 26.Qxe4 Nd5 27.Ng4 Nf8 28.Ne5 a4
 29.Ng6 a3 30.Nxf8 Rxf8 31.Qxe6+ Qf7 32.Qd6 1-0

[Event "corr TGM 12728"]
 [Site "IECG"]
 [Date "1999.03.08"]
 [Round "01"]
 [White "Morihaman,Nicolau"]
 [Black "Lee,Shuman"]
 [Result "1-0"]
 [ECO "D31"]

1.d4 d5 2.c4 e6 3.Nf3 c6 4.Nc3 dxc4 5.a4 Bb4 6.e3 b5 7.Bd2 Bb7 8.axb5 Bxc3
 9.Bxc3 cxb5 10.b3 a5 11.bxc4 b4 12.Bb2 Nf6 13.Bd3 Nbd7 14.O-O O-O 15.Qc2 Qc7
 16.e4 e5 17.Rfe1 Rfe8 18.c5 exd4 19.Bxd4 h6 20.e5 Nd5 21.e6 Rxe6 22.Rxe6
 fxe6 23.Re1 Nf4 24.Be4 Rc8 25.g3 Nh3+ 26.Kf1 e5 27.Nxe5 Nxe5 28.Bh7+ Kh8
 29.Bxe5 Qf7 30.Bf5 Ng5 31.Bxc8 Bxc8 32.f4 Ba6+ 33.Kf2 Qh5 34.Ke3 Qf3+ 35.Kd2
 Ne6 36.c6 b3 37.Qg6 Qd5+ 38.Ke3 Qc5+ 39.Kf3 Qxc6+ 40.Kg4 Qc4 41.Kh4 1-0

[\(top\)](#)

Version Information

Since a new version number can sometimes trigger rebuilding an INI file, U4Chess does not change major version numbers often. Rather, the programs rely on "build" numbers. To get full version information, right click the EXE, select Properties and then the Version tab.

Incidentally, although INI files are a bit old fashioned, I prefer them to using the Registry. The U4 philosophy is to leave a clean machine should anyone care to remove one of the applications. What goes into the Registry never comes out and those of us on older machines appreciate what that can mean over time.

[\(top\)](#)

Other Programs

I have only 2 Mb allotted to me on this site so if new programs come up, some of the old ones will have to come off. So long as Michael Sharpe can take them, everything's okay so be sure to check his site (his

link at the [top](#) of this page).

Here's a complete program list:

This Site

- PGNEdit ECOClass Slinger (and recent updates)

[PGN.freeservers](#)

- NORMAL32 - prepare PGN games for entry in most any database
- (Ya)hooScore - with Normal, converts Yahoo games scores to PGN
- HighRank - standardize player names; add ratings
- CChunk - splits large PGN files--or combines
- CollEdit - do almost anything with lists (like rating lists) and small PGN files
- PGNScan - find games by rules of notation (strings, partial strings, move X precedes Y, etc.)
- PGNSite - customizes contents of PGN Site, Event & Round
- SEClean - when Site equals Event, eliminate one or the other
- ChessU4 - what started it all, PGN chess viewer
- ChessU3 - says, "I started it all," PGN tree searcher
- ClassC - analyze openings by color; not for everyone (abstract)
- PGNSort - sort PGN files by headers, notation--or randomly

-Paul Onstad [\(top\)](#)

[\(home\)](#)

FilePosSearch (FILEPOSSRCH_H)

The **FilePosSearch** function (one you describe as a template) returns after writing a file of PGN games that matched the position search argument. It will also, on option, return an array of the game numbers selected.

DWORD FilePosSearch(

```

LPHANDLE myhdl    // maintained by DLL
DWORD flag        // flag, set to 0x00000029 (41)
LPTSTR infn       // filename of the PGC database
LPTSTR outfn      // filename of temporary work file
LPSTR query       // formal chess position description
LPSTR pc          // work string (50000 bytes)
LPSTR pc2         // work string (50000 bytes) and error return string
LPLIARY liary1    // pointer to array of dword, or null pointer
LPLIARY liary2    // pointer to array of dword, or null pointer
LPDWORD numsel   // number of games selected (same as liary2[0])

```

);

Parameters

myhdl

It should be initialized to 0 (once) after the DLL is loaded. Thereafter it is maintained by the DLL. Subsequent DLL calls should not reinitialize unless the DLL is unloaded and then reloaded.

flag *revised*

[**FMT_Closed** or **FMT_Open**] + [**CHK_ErrorOn** or **CHK_ErrorOff**] + [**FIO_...** flag]

Open/closed formats apply to chess notation--with or without space after period.

CHK_ErrorOff speeds position searching by ignoring checks and mates.

FIO flags handle disposition of the output file, whether it should be overwritten or not. See `ppkType3.pas`.

A common setting is 41 (decimal)--a combination of "create or overwrite," "check error off," and "closed format."

infn

A null terminated string containing the path and filename of a PGC file.

outfn

A null terminated string containing a filename which will receive the PGN games (if any) selected. Note: this file is always written--even if *liary1* is combined with *liary2* for other purposes.

query

a null terminated string containing a formal position-search statement. See examples in Remarks, below.

pc

A work area provided the DLL, pre-allocated to 50,000 bytes. It need not be initialized with any particular contents.

pc2

The same as *pc* except that, if the function fails, it will return with the error trap code.

liary1 *revised*

Input selection. See [FileQuerySel](#) for a description of this element. Advanced use could form a selection by a header search, pass it to this routine, and this routine will select, by position, only from games matching to the first search..

For present purposes, this parameter can be passed as null.

liary2

Output results selection. See [FileQuerySel](#) example.

For present purposes, this parameter can be passed as null.

numsel

The number of games selected. If *liary2* were used, its element [0] would contain the same number.

Return Values

If the function succeeds, the return value is 0x00000001. If it fails, the return is 0 (Pascal LongBool). An error message (trap code) will be passed back in *pc2* in the case of failure.

(The file [Trapsum.txt pgcdll.zip](#) has a list of all error trap codes.)

Remarks

A description of position search statements can be found in [MBase32.zip](#). The simplest form is to select by a string of chess notation:

```
L:1.e4 e5 2.Nf3 Nc6 3.Bb5
```

The string can be followed by a result (ignored). **FilePosSearch** calculates the ply depth by the number of moves presented but it can be overridden by a specified ply or with a range:

```
L,5:1.e4 e5 2.Nf3 Nc6 3.Bb5
```

```
L,3,8:1.e4 e5 2.Nf3 Nc6 3.Bb5
```

This is only necessary for some deeper lines that can reach the same position in a different number of moves. For searches using FEN or EPD the range (or ply) must be specified. It can be estimated or it can include every move of every game:

```
P,1,1000:r1bqkbnr/pppp1ppp/2n5/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R
```

Note that "P" precedes a FEN or EPD line. Since only the position portion is considered, there is no difference between the two. An abbreviated "FEN" is also permitted (no "slash" characters with ranks possibly run together).

FilePosSearch automatically opens, reads, and closes the PGC database--in the process creating the output PGN file.

Loading the DLL

```
dllhandle = LoadLibrary('ppkdlv03.dll'); *revised*
```

```
if (dllhandle != 0)
```

```
{
```

```
/* GetProcAddress for each of the following:
```

```
'GAMEUNPKPGN_D'
```

```
'GAMEPACKPGN_C'
```

```
'FILEQUERYSL_F'
```

```
'FILEPOSSRCH_H'
```

```
'DESTROY'
```

```
Allocate memory for null terminated strings
```

```
pc, pc2      (50000 bytes each, work strings)
```

```
infn, outfn  (file name size including path)
```

```
query       (512 bytes)
```

```
as64        (64 bytes, a rtn error msg)
```

```
Initialize the data handle
```

```
myhdl = 0; *important* this is the data  
handle passed to the dll and setting it  
to 0 is the signal for its startup  
initialization. The value it passes back  
must be retained for subsequent calls but  
is not otherwise referenced by the calling  
program. */
```

```
};
```

Destroy (DESTROY)

The **Destroy** function (one you describe as a template) frees the data used by the DLL.

DWORD Destroy(

```
LPHANDLE myhdl // data handle passed to DLL  
);
```

Parameters

myhdl
Prior to unloading the DLL, its data should be freed by a call to **Destroy**. Do so only if the handle was successfully allocated by calls to other DLL functions (check to see that it has a non-zero value).

Return Values

If the function succeeds, the return value is 0x00000001. If it fails, the return is 0 (Pascal LongBool).

Remarks

When freeing the DLL's data, preallocated pointers can be freed as well (*pc*, *pc2*, *query*, *as64*, *liary*, and filename strings).

Pascal Example

Note: The LoadDLL function is listed in full, however for simplification, only the **GameUnpkPGN** example has been fully expanded. Templates for all referenced functions in the DLL *are* included.

uses

Windows, SysUtils ...

const

ppkDLL = 'ppkdlv03.dll';
 PCharStrLen = 50000;
 PCharShortLen = 512;

type

PCBufAry = **array** [0..PCharStrLen] **of** **char**;
 PCBufPtr = ^PCBufAry;

//templates

TGameUnpkPGN = **function**(**var** MemHdl:THandle;Flag:integer;PC,PC2:PChar):LongBool;
TGamePackPGN = **function**(**var** MemHdl:THandle;PC,PC2:PChar):LongBool;
TFileQuerySl = **function**(**var** MemHdl:THandle;FNIn,Query,AS64:PChar;SGame,RGame:integer;
 LIAry:LIArrayPtr):LongBool;
TFilePosSrch = **function**(**var** MemHdl:THandle;Flag:integer;FNIn,FNOut,Query,PC,PC2:PChar;
 LINil,LINil;**var** NumSel:longint):LongBool;
TDestroy = **function**(**var** MemHdl:THandle):LongBool;

var

Handle : THandle; //DLL handle
 MyHdl : THandle; //private handle
 PC, PC2 : PChar;
GameUnpkPGN : **TGameUnpkPGN**;
 fIn, fOut : text;

function TForm5.LoadDLL:Boolean;

begin

result:=false;
 Handle:=LoadLibrary(PChar(ppkDLL));
if Handle <> 0 **then**
begin
 @ **GameUnpkPGN**:=GetProcAddress(Handle,'GAMEUNPKPGN_D');
 @GamePackPGN:=GetProcAddress(Handle,'GAMEPACKPGN_C');
 @FileQuerySl:=GetProcAddress(Handle,'FILEQUERYSL_F');
 @FilePosSrch:=GetProcAddress(Handle,'FILEPOSSRCH_H');
 @Destroy :=GetProcAddress(Handle,'DESTROY');
if (@ **GameUnpkPGN**<>nil) **and**
 (@GamePackPGN<>nil) **and**
 (@FileQuerySl<>nil) **and**
 (@FilePosSrch<>nil) **and** (@Destroy<>nil) **then**

begin

```
GetMem(PC,PCharStrLen);
GetMem(PC2,PCharStrLen);
GetMem(FNIn,PCShortLen);
GetMem(FNOut,PCShortLen);
GetMem(Query,PCShortLen);
GetMem(AS64,PCShortLen);
MyHdl:=0;
result:=True;
```

end

else

begin

```
FreeLibrary(Handle);
Handle:=0;
```

end;

end;

end;

Next is the read loop--reading one record (a line of text) from a PGC file and writing one (formatted) PGN game

Flag:=0;

PCBuf:=PCBufPtr(PC);

repeat

```
Readln(fIn,PCBuf^);
```

```
if not GameUnpkPGN(myHdl,Flag,PC,PC2) then begin
```

```
s:=StrPas(PC2);
```

```
raise exception.create(s); end;
```

```
WriteLn(fOut,#13#10,PC2);
```

until EOF(fIn);

[Main DLL/PGC page](#)

...back to Pascal again. Here are the tags supported by [ppkd1v03](#):

this page completely revised

```
type
  TagType = (Event,Site,Date,Round_,White,Black,Result_,
            ECO,WhiteElo,BlackElo,Source,Opening,
            WhiteCountry,BlackCountry,SetUp,FEN,
            Annotator,Comment, {18}
            PlyCount,EventDate,
            Anon1,Anon2,Anon3,Anon4,Anon5, {25}
            TimeControl,Mode,Termination,
            Variation,SubVariation,
            WhiteTitle,BlackTitle); {32}
```

```
TagAry3 = array [Event..High(TagType)] of PChar;
```

```
const
```

```
AnonSet = [Anon1..Anon5];
```

```
cPGNTag3 : TagAry3 = (
```

```
  {1} ('Event'),
```

```
  {2} ('Site'),
```

```
  {3} ('Date'),
```

```
  {4} ('Round'),
```

```
  {5} ('White'),
```

```
  {6} ('Black'),
```

```
  {7} ('Result'),
```

```
  {8} ('ECO'),      {1}
```

```
  {9} ('WhiteElo'), {2}
```

```
{10} ('BlackElo'), {3}
```

```
{11} ('Source'),   {4}
```

```
{----Archive implementation above. Call 'ArchiveTags' after dll loaded for reduced set
```

```
{12} ('Opening'),  {5}
```

```
{13} ('WhiteCountry'), {6}
```

```
{14} ('BlackCountry'), {7}
```

```
{15} ('SetUp'),    {8}
```

```
{16} ('FEN'),      {9}
```

```
{17} ('Annotator'), {10}
```

```
{18} ('Comment'),  {11}
```

```
{----Default implementation above. Call 'ExtendTags' after dll loaded for extended set
```

```
{19} ('PlyCount'),
```

```
{20} ('EventDate'),
```

```
{21} ('Anon1'),
```

```
{22} ('Anon2'),
```

```
{23} ('Anon3'),
```

```
{24} ('Anon4'),
```

```
{25} ('Anon5'),
```

```
{----Extended implementation above. Call 'FullTags' after dll loaded for full set
{26} ('TimeControl'),
{27} ('Mode'),
{28} ('Termination'),
{29} ('Variation'),
{30} ('SubVariation'),
{31} ('WhiteTitle'),
{32} ('BlackTitle'));
```

There are four possible configurations--each compatible with every other. Without specification, the DLL defaults to the first 18 tags ("Comment" being the last). Reduced implementations will not "see" any of the tags above them (were a pgn file to come from some outside source) nor will they be able to create them. A reduced set going from pgn > pgn > pgn would lose the extended tag contents.

The advantage of having a reduced set is speed and smaller size. If an extended set is implemented but none of the extended tags are in use, the size will be the same, however, processing will be slower since all packing operations will scan incoming PGN for *all* tags, building a packed record of *all* placeholders--then stripping the unused placeholders off the end until coming to the first placeholder that is followed by actual contents.

All tags become single byte flags. Counting them determines which actual tag is being represented.

To implement an Archive or an Extended set, an application must always make a call to one of the following, immediately after the DLL is loaded:

```
ARCHIVEDTAGS_VV
EXTENDED_TAGS_UU
FULLTAGSIMPL_TT (see ppkdlv03.pas in pgcdll.zip for the format of these calls)
```

If "Anon" tags are being translated (extended or full only), the next call would be to:

```
SETTAGTRANSS_WW
```

The call requires a translate string parameter. Sample:

```
'Anon1:MyTag Anon2:MyTag2'
```

(Note the placement of colons and spaces.) Up to five translations can be made with the string (Anon1-Anon5). When translations are used, the translated tag (eg, "MyTag") can be present in any input PGN and will appear in output PGN for those games where it held data. Header query statements can also search using the translated name.

[top](#)

