Bo Haglund, Soren Hein, Bob Richardson

Rev X, 2014-11-16

Latest DLL issue with this description is available at http://www.bahnhof.se/wb758135/ (http://www.bahnhof.se/wb758135/)

Description of the DLL functions supported in Double Dummy Problem Solver 2.8

Callable functions

The callable functions are all preceded with `extern "C" __declspec(dllimport) int __stdcall`. The prototypes are available in `dll.h`, in the include directory.

Return codes are given at the end.

Not all functions are present in all versions of the DLL. For historical reasons, the function names are not entirely consistent with respect to the input format. Functions accepting binary deals will end on Bin, and those accepting PBN deals will end on PBN in the future. At some point existing function names may be changed as well, so use the new names!

The Basic Functions

The basic functions `SolveBoard` and `SolveBoardPBN` each solve a single hand and are thread-safe, making it possible to use them for solving several hands in parallel. The other callable functions use the SolveBoard functions either directly or indirectly.

The Multi-Thread Double Dummy Solver Functions

The double dummy trick values for all 5 * 4 = 20 possible combinations of a hand's trump strain and declarer hand alternatives are solved by a single call to one of the functions `CalcDDtable` and `CalcDDtablePBN`. Threads are allocated per strain. in order to save computations.

To obtain better utilization of available threads, the double dummy (DD) tables can be grouped using one of the functions `CalcAllTables` and `CalcAllTablesPBN`.

Solving hands can be done much more quickly using one of the multi-thread alternatives for calling SolveBoard. Then a number of hands are grouped for a single call to one of the functions `SolveAllBoards`, `SolveAllChunksBin` and `SolveAllChunksPBN`. The hands are then solved in parallel using the available threads.

The number of threads is automatically configured by DDS on Windows, taking into account the number of processor cores and available memory. The number of threads can be influenced using by calling `SetMaxThreads`. This function should probably always be called on Linux/Mac, with a zero argument for auto-configuration.

Calling `FreeMemory` causes DDS to give up its dynamically allocated memory.

The PAR Calculation Functions

The PAR calculation functions find the optimal contract(s) assuming open cards and optimal bidding from both sides. In very rare cases it matters which side or hand that starts the bidding, i.e. which side or hand that is first to bid its optimal contract.

Two alternatives are given:

1. The PAR scores / contracts are calculated separately for each side. In almost all cases the results will be identical for both sides, but in rare cases the result is dependent on which side that “starts the bidding”, i.e. that first finds the bid that is most beneficial for the own side. One example is when both sides can make 1 NT.
2. The dealer hand is assumed to “start the bidding”.

The presentation of the par score and contracts are given in alternative formats.

The functions `Par`, `SidesPar` and `DealerPar` do the par calculation; their call must be preceded by a function call calculating the double dummy table values.

The functions `SidesParBin` and `DealerParBin` provide binary output of the par results, making it easy to tailor-make the output text format. Two such functions, `ConvertToSidesTextFormat` and `ConvertToDealerTextFormat`, are included as examples.

It is possible as an option to perform par calculation in `CalcAllTables` and `CalcAllTablesPBN`.

The par calculation is executed using a single thread. But the calculation is very fast and its duration is negligible compared to the double dummy calculation duration.

Double Dummy Value Analyser Functions

The functions `AnalysePlayBin`, `AnalysePlayPBN`, `AnalyseAllPlaysBin` and `AnalyseAllPlaysPBN` take the played cards in a game or games and calculate and present their double dummy values.

Function	Arguments	Format	Comment

`SolveBoard`	`struct deal dl`
	`int target`
	`int solutions`
	`int mode`
	`struct futureTricks *futp`
	`int threadIndex`
		Binary	The most basic function, solves a single hand from the beginning or from later play

`SolveBoardPBN`	`struct dealPBN (dealPBN) dlPBN`
	`int target`
	`int solutions`
	`int mode`
	`struct futureTricks *futp`
	`int threadIndex`
		PBN	As SolveBoard, but with PBN deal format.

`CalcDDtable`	`struct ddTableDeal tableDeal` `struct ddTableResults *tablep`	Binary	Solves an initial hand for all possible declarers and denominations (up to 20 combinations.)
CalcDDtablePBN	`struct ddTableDealPBN tableDealPBN` `struct ddTableResults *tablep`	PBN	As CalcDDtable, but with PBN deal format.

`CalcAllTables`	`struct ddTableDeals *dealsp`
	`int mode`
	`int trumpFilter[5]`
	`struct ddtablesRes *resp`
	`struct allParResults *pres`
		Binary	Solves a number of hands in parallel. Multi-threaded.

CalcAllTablesPBN	struct ddTableDealsPBN (ddTableDealsPBN) *dealsp int mode int trumpFilter[5] struct ddTablesRes *resp struct allParResults *pres	PBN	As CalcAllTables, but with PBN deal format.
SolveAllBoards	struct boardsPBN *bop struct solvedBoards (solvedBoards) *solvedp	PBN	Consider using this instead of the next 3 “Chunk” functions”!
SolveAllChunksBin	struct boards *bop struct solvedBoards *solvedp int chunkSize	Binary	Solves a number of hands in parallel. Multi-threaded.
SolveAllChunks	struct boardsPBN *bop struct solvedBoards *solvedp int chunkSize	PBN	Alias for SolveAllChunksPBN; don't use!
SolveAllChunksPBN	struct boardsPBN" *bop struct solvedBoards *solvedp int chunkSize	PBN	Solves a number of hands in parallel. Multi-threaded.
Par	struct ddTableResults *tablep struct parResults *presp int vulnerable	No format	Solves for the par contracts given a DD result table.
DealerPar	struct ddTableResults *tablep struct parResultsDealer *presp int dealer int vulnerable	No format	Similar to Par(), but requires and uses knowledge of the dealer.
DealerParBin	struct ddTableResults *tablep struct parResultsMaster *presp int dealer int vulnerable	Binary	Similar to DealerPar, but with binary output.
ConvertToDealerTextFormat	struct parResultsMaster *pres char *resp	Text	Example of text output from DealerParBin.
SidesPar	struct ddTableResults *tablep struct parResultsDealer *presp int vulnerable	No format	Par results are given for sides with the DealerPar output format.
SidesParBin	struct ddTableResults *tablep struct parResultsMaster sidesRes[2] int vulnerable	Binary	Similar to SidesPar, but with binary output.
ConvertToSidesTextFormat	struct parResultsMaster *pres char *resp	Text	Example of text output from SidesParBin.

CalcPar	struct ddTableDeal tableDeal int vulnerable struct ddTabeleResults *tablep struct parResults *presp	Binary	Solves for both the DD result table and the par contracts. Is deprecated, use a CalcDDtable function plus Par() instead!
CalcParPBN	struct ddTableDealPBN tableDealPBN struct ddTableResults *tablep int vulnerable struct parResults *presp	PBN	As CalcPar, but with PBN input format. Is deprecated, use a CalcDDtable function plus Par() instead!
AnalysePlayBin	struct deal (deal) dl struct playTraceBin play struct solvedPlay *solvedp int thrId	Binary	Returns the par result after each card in a particular play sequence.
AnalysePlayPBN	struct dealPBN dlPBN struct playTracePBN playPBN struct solvedPlay *solvedp int thrId	PBN	As AnalysePlayBin, but with PBN deal format.
AnalyseAllPlaysBin	struct boards *bop struct playTracesBin *plp struct solvedPlays *solvedp int chunkSize	Binary	Solves a number of hands with play sequences in parallel. Multi-threaded.
AnalyseAllPlaysPBN	struct boardsPBN *bopPBN struct playTracesPBN *plpPBN struct solvedPlays *solvedp int chunkSize	PBN	As AnalyseAllPlaysBin, but with PBN deal format.
SetMaxThreads	int userThreads	PBN	Used at initial start and can also be called with a request for allocating memory for a specified number of threads. Is apparentlyÃ‚Â¸mandatory on Linux and Mac (optional on Windows)
FreeMemory	void		Frees DDS allocated dynamical memory.
ErrorMessage	int code char line[80]	Turns a return code into an error message string.	

Data structures

Common encodings are as follows

Encoding	Element	Value
Suit	Spades	0
	Hearts	1
	Diamonds	2
	Clubs	3
	NT	4
Hand	North	0
	East	1
	South	2
	West	3
Side	N-S	0
	E-W	1
Card	Bit 2	Rank of deuce
	...	
	Bit 13	Rank of king
	Bit 14	Rank of ace
Holding	A value of 16388 = 16384 + 4 is the encoding for the holding “A2” (ace and deuce). The two lowest bits are always zero.	
PBN	Example: W:T5.K4.652.A98542 K6.QJT976.QT7.Q6 432.A.AKJ93.JT73 AQJ987.8532.84.K	

struct	**Field**	**Comment**
`deal`	`int trump`	Suit encoding
	`int first`	The hand leading to the trick. Hand encoding.
	`int currentTrickSuit[3]`	Up to 3 cards may already have been played to the trick. Suit encoding.
	`int currentTrickRank[3]`	Up to 3 cards may already have been played to the trick. Value range 2-14. Set to 0 if no card has been played.
	`unsigned int remainCards[4][4]`	1st index is Hand, 2nd index is Suit. remainCards is Holding encoding.

struct	**Field**	**Comment**
`dealPBN`	`int trump`	Suit encoding
	`int first`	The hand leading to the trick. Hand encoding.
	`int currentTrickSuit[3]`	Up to 3 cards may already have been played to the trick. Suit encoding. Set to 0 if no card has been played.
	`int currentTrickRank[3]`	Up to 3 cards may already have been played to the trick. Value range 2-14. Set to 0 if no card has been played.
	`char remainCards[80]`	Remaining cards. PBN encoding.

struct	**Field**	**Comment**
`ddTableDeal`	`unsigned int cards[4][4]`	Encodes a deal. First index is hand. Hand encoding. Second index is suit. Suit encoding.

struct	**Field**	**Comment**
`ddTableDealPBN`	`char cards[80]`	Encodes a deal. PBN encoding.

struct	**Field**	**Comment**

ddTableDeals	int noOfTables	Number of DD table deals in structure, at most MAXNOOFTABLES
	struct ddTableDeal deals[X]	X = MAXNOOFTABLES * DDS_STRAINS
struct	**Field**	**Comment**
ddTableDealsPBN	int noOfTables	Number of DD table deals in structure
	struct ddTableDealPBN deals[X]	X = MAXNOOFTABLES * DDS_STRAINS
struct	**Field**	**Comment**
boards	int noOfBoards	Number of boards
	struct deal[MAXNOOFBOARDS]	
	int target[MAXNOOFBOARDS]	See SolveBoard
	int solutions[MAXNOOFBOARDS]	See SolveBoard
	int mode[MAXNOOFBOARDS]	See SolveBoard
struct	**Field**	**Comment**
boardsPBN	int noOfBoards	Number of boards
	struct dealPBN[MAXNOOFBOARDS]	
	int target[MAXNOOFBOARDS]	See SolveBoard
	int solutions[MAXNOOFBOARDS]	See SolveBoard
	int mode[MAXNOOFBOARDS]	See SolveBoard
struct	**Field**	**Comment**
futureTricks	int nodes	Number of nodes searched by the DD solver.
	int cards	Number of cards for which a result is returned. May be all the cards, but equivalent ranks are omitted, so for a holding of KQ76 only the cards K and 7 would be returned, and the “equals” field below would be 2048 (Q) for the king and 54 (6) for the 7.
	int suit[13]	Suit of the each returned card. Suit encoding.
	int rank[13]	Rank of the returned card. Value range 2-14.
	int equals[13]	Lower-ranked equals. Holding encoding.
	int score[13]	-1: target not reached. Otherwise: Target of maximum number of tricks.
struct	**Field**	**Comment**
solvedBoards	int noOfBoards	
	struct futureTricks solvedBoard [MAXNOOFBOARDS]	
Struct	**Field**	**Comment**
ddTableResults	int resTable[5][4]	Encodes the solution of a deal for combinations of denomination and declarer. First index is denomination. Suit encoding. Second index is declarer. Hand encoding. Each entry is a number of tricks.
struct	**Field**	**Comment**
ddTablesRes	int noOfBoards	Number of DD table deals in structure, at most MAXNOOFTABLES
	struct ddTableResults	

	results[X]	X = MAXNOOFTABLES * DDS_STRAINS
Struct	**Field**	**Comment**
parResults	char parScore[2][16]	First index is NS/EW. Side encoding.
	char parContractsString[2][128]	First index is NS/EW. Side encoding.
Struct	**Field**	**Comment**
allParResults	struct parResults[MAXNOOFTABLES]	There are up to 20 declarer/strain combinations per DD table
Struct	**Field**	**Comment**
parResultsDealer	int number	
	int score	
	char contracts[10][10]	
Struct	**Field**	**Comment**
parResultsMaster	int score	
	int number	
	struct contractType contracts[10]	
Struct	**Field**	**Comment**
contractType	int underTricks	
	int overTricks	
	int level	
	int denom	
	int seats	
Struct	**Field**	**Comment**
parTextResults	char parText[2][128]	
	int equal	
Struct	**Field**	**Comment**
playTraceBin	int number	Number of cards in the play trace, starting from the beginning of the hand.
	int suit[52]	Suit encoding.
	int rank[52]	Encoding 2 .. 14 (not Card encoding).
Struct	**Field**	**Comment**
playTracePBN	int number	Number of cards in the play trace, starting from the beginning of the hand.
	int cards[106]	String of cards with no space in between, also not between tricks. Each card consists of a suit (C/D/H/S) and then a rank (2 .. A). The string must be null-terminated.
Struct	**Field**	**Comment**
playTracesBin	int noOfBoards	
	struct playTraceBin plays[MAXNOOFBOARDS / 10]	
Struct	**Field**	**Comment**
playTracesPBN	int noOfBoards	
	struct playTracePBN plays[MAXNOOFBOARDS / 10]	
Struct	**Field**	**Comment**
solvedPlay	int number	
	int tricks[53]	Starting position and up to 52 cards
Struct	**Field**	**Comment**

```
solvedPlays      int noOfBoards
                 struct solvedPlay
                solved[MAXNOOFBOARDS]
```

Functions

SolveBoard	**SolveBoardPBN**
`struct deal dl`	`struct dealPBN dl`
`int target`	`int target`
`int solutions`	`int solutions`
`int mode`	`int mode`
`struct futureTricks *futp`	`struct futureTricks *futp`
`int threadIndex`	`int threadIndex`

SolveBoardPBN is just like SolveBoard, except for the input format. Historically it was one of the first functions, and it exposes the thread index directly to the user. Later functions generally don't do that, and they also hide the implementation details such as transposition tables, see below.

SolveBoard solves a single deal " `dl` " and returns the result in " `*futp` " which must be declared before calling SolveBoard.

If you have multiple hands to solve, it is always better to group them together into a single function call than to use SolveBoard.

SolveBoard is thread-safe, so several threads can call SolveBoard in parallel. Thus the user of DDS can create threads and call SolveBoard in parallel over them. The maximum number of threads is fixed in the DLL at compile time and is currently 16. So " `threadIndex` " must be between 0 and 15 inclusive; see also the function SetMaxThreads. Together with the PlayAnalyse functions, this is the only function that exposes the thread number to the user.

There is a " `transposition table` " memory associated with each thread. Each node in the table is effectively a position after certain cards have been played and other certain cards remain. The table is not deleted automatically after each call to SolveBoard, so it can be reused from call to call. However, it only really makes sense to reuse the table when the hand is very similar in the two calls. The function will still run if this is not the case, but it won't be as efficient. The reuse of the transposition table can be controlled by the " `mode` " parameter, but normally this is not needed and should not be done.

The three parameters " `target` ", " `solutions` " and " `mode` " together control the function. Generally speaking, the target is the number of tricks to be won (at least) by the side to play; solutions controls how many solutions should be returned; and mode controls the search behavior. See next page for definitions.

For equivalent cards, only the highest is returned, and lower equivalent cards are encoded in the `futureTricks` structure (see " `equals` ").

target	**solutions**	**comment**
-1	1	Find the maximum number of tricks for the side to play. Return only one of the optimum cards and its score.
-1	2	Find the maximum number of tricks for the side to play. Return all optimum cards and their scores.
0	1	Return only one of the cards legal to play, with score set to 0.
0	2	Return all cards that legal to play, with score set to 0.
1 .. 13	1	If score is -1: Target cannot be reached. If score is 0: In fact no tricks at all can be won. If score is > 0: score will always equal target, even if more tricks can be won. One of the cards achieving the target is returned.
1 .. 13	2	Return all cards meeting (at least) the target. If the target cannot be achieved, only one card is returned with the score set

		as above.
any	3	Return all cards that can be legally played, with their scores in descending order.

mode	Reuse TT?	comment
0	Automatic if same trump suit and the same or nearly the same cards distribution, deal.first can be different.	Do not search to find the core if the hand to play has only one card, including its equivalents, to play. Score is set to Ã¢â‚¬â€œ2 for this card, indicating that there are no alternative cards. If there are multiple choices for cards to play, search is done to find the score. This mode is very fast but you don't always search to find the score.
1		Always search to find the score. Even when the hand to play has only one card, with possible equivalents, to play.
2	Always	

Note: mode no longer always has this effect internally in DDS. We think mode is no longer useful, and we may use it for something else in the future. If you think you need it, let us know!

“Reuse” means “reuse the transposition table from the previous run with the same thread number”. For mode = 2 it is the responsibility of the programmer using the DLL to ensure that reusing the table is safe in the actual situation. Example: Deal is the same, except for `deal.first` . The Trump suit is the same.

1st call, East leads: `SolveBoard(deal, -1, 1, 1, &fut, 0), deal.first=1`

2nd call, South leads: `SolveBoard(deal, -1, 1, 2, &fut, 0), deal.first=2`

3rd call, West leads: `SolveBoard(deal, -1, 1, 2, &fut, 0), deal.first=3`

4th call, North leads: `SolveBoard(deal, -1, 1, 2, &fut, 0), deal.first=0`

CalcDDtable	**CalcDDtablePBN**
`struct ddTableDeal tableDeal`	`struct ddTableDealPBN tableDealPBN`
`struct ddTableResults *tablep`	`struct ddTableResults *tablep`

CalcDDtablePBN is just like CalcDDtable, except for the input format. CalcDDtable solves a single deal “ tableDeal ” and returns the double-dummy values for the initial 52 cards for all the 20 combinations of denomination and declarer in “*tablep” , which must be declared before calling CalcDDtable.

CalcAllTables	**CalcAllTablesPBN**
`struct ddTableDeals *dealsp`	`struct ddTableDealsPBN *dealsp`
`int mode`	`int mode`
`int trumpFilter[5]`	`int trumpFilter[5]`
`struct ddTablesRes *resp`	`struct ddTablesRes *resp`
<struct allParResults *presp	`struct allParResults *presp`

CalcAllTablesPBN is just like CalcAllTables, except for the input format.

CallAllTables calculates the double dummy values of the denomination/declarer hand combinations in “*dealsp” for a number of DD tables in parallel. This increases the speed compared to calculating these values using a CalcDDtable call for each DD table. The results are returned in “*resp” which must be defined before CalcAllTables is called.

The “mode” parameter contains the vulnerability (Vulnerable encoding; not to be confused with the SolveBoard mode) for use in the par calculation. It is set to -1 if no par calculation is to be performed.

There are 5 possible denominations or strains (the four trump suits and no trump). The parameter “trumpFilter” describes which, if any, of the 5 possibilities that will be excluded from the calculations. They are defined in Suit encoding order, so setting trumpFilter to {FALSE, FALSE, TRUE, TRUE, TRUE} means that values will only be calculated for the trump suits spades and hearts.

The maximum number of DD tables in a CallAllTables call depends on the number of strains required, see the following table:

Number of strainsMaximum number of DD tables

5	32
4	40
3	53
2	80
1	160

SolveAllBoards	**SolveAllChunksBin**	**SolveAllChunksPBN**
`struct boards *bop`	`struct boards *bop`	`struct boardsPBN *bop`
`struct solvedBoards *solvedp`	`struct solvedBoards *solvedp`	`struct solvedBoards *solvedp`
`int chunkSize`	`int chunkSize`	

`SolveAllChunks` is an alias for SolveAllChunksPBN; don't use it.

`SolveAllBoards` used to be an alias for SolveAllChunksPBN with a chunkSize of 1; however this has been changed in v2.8, and we now recommend only to use SolveAllBoards and not the chunk functions any more; explanation follows.

The SolveAll* functions invoke SolveBoard several times in parallel in multiple threads, rather than sequentially in a single thread. This increases execution speed. Up to 200 boards are permitted per call.

It is important to understand the parallelism and the concept of a chunk.

If the chunk size is 1, then each of the threads starts out with a single board. If there are four threads, then boards 0, 1, 2 and 3 are initially solved. If thread 2 is finished first, it gets the next available board, in this case board 4. Perhaps this is a particularly easy board, so thread 2 also finishes this board before any other thread completes. Thread 2 then also gets board 5, and so on. This continues until all boards have been solved. In the end, three of the threads will be waiting for the last thread to finish, which causes a bit of inefficiency.

The transposition table in a given thread (see SolveBoard) is generally not reused between board 2, 4 and 5 in thread 2. This only happens if SolveBoard itself determines that the boards are suspiciously similar. If the chunk size is 2, then initially thread 0 gets boards 0 and 1, thread 1 gets boards 2 and 3, thread 2 gets boards 4 and 5, and thread 3 gets boards 6 and 7. When a thread is finished, it gets two new boards in one go, for instance boards 8 and 9. The transposition table in a given thread is reused within a chunk.

No matter what the chunk size is, the boards are solved in parallel. If the user knows that boards are grouped in chunks of 2 or 10, it is possible to force the DD solver to use this knowledge. However, this is rather limiting on the user, as the alignment must remain perfect throughout the batch.

SolveAllBoards now detects repetitions automatically within a batch, whether or not the hands are evenly arranged and whether or not the duplicates are next to each other. This is more flexible and transparent to the user, and the overhead is negligible. Therefore, use SolveAllBoards!

Par	**DealerPar**
`struct ddTableResults *tablep`	`struct ddTableResults *tablep`
`struct parResults *presp`	`struct parResultsDealer *presp`
`int vulnerable`	`int dealer`
	`int vulnerable`

SidesPar

```
struct ddTableResults *tablep

struct parResultsDealer *sidesRes[2]

int vulnerable
```

DealerParBin	**SidesParBin**
`struct ddTableResults *tablep`	`struct ddTableResults *tablep`
`struct parResultsMaster *presp`	`struct parResultsMaster *presp`

`int vulnerable`	`int dealer` `int vulnerable`

ConvertToDealerTextForamat	**ConvertToSidesTextFormat**
`struct parResultsMaster *pres`	`struct parResultsMaster *pres`
`char *resp`	`struct parTextResults *resp`

The functions Par, DealerPar, SidesPar, DealerParBin and SidesParBin calculate the par score and par contracts of a given double-dummy solution matrix `*tablep` which would often be the solution of a call to CalcDDtable. Since the input is a table, there is no PBN and non-PBN version of these functions.

Before the functions can be called, a structure of the type “parResults” , `parResultsDealer` or `parResultsMaster` must already have been defined.

The `vulnerable` parameter is given using Vulnerable encoding.

The Par() function uses knowledge of the vulnerability, but not of the dealer. It attempts to return results for both declaring sides. These results can be different in some rare cases, for instance when both sides can make 1NT due to the opening lead.

The DealerPar() function also uses knowledge of the `dealer` using Hand encoding. The rgument is that in all practical cases, the dealer is known when the vulnerability is known. Therefore all results returned will be for the same side.

The SidesPar() function is similar to the Par() function, the only difference is that the par results are given in the same format as for DealerPar().

In Par() and SidesPar() there may be more than one par score; in DealerPar() that is not the case. Par() returns the scores as a text string, for instance “NS -460”, while DealerPar() and SidesPar() use an integer, -460.

There may be several par contracts, for instance 3NT just making and 5C just making. Each par contract is returned as a text string. The formats are a bit different betweeen the two output format alternatives.

Par() returns the par contracts separated by commas. Possible different trick levels of par score contracts are enumerated in the contract description, e.g the possible trick levels 3, 4 and 5 in no trump are given as 345N. Examples:

- “NS:NS 23S,NS 23H”. North and South as declarer make 2 or 3 spades and hearts contracts, 2 spades and 2 hearts with an overtrick. This is from the NS view, shown by “NS:” meaning that NS made the first bid. Note that this information is actually not enough, as it may be that N and S can make a given contract and that either E or W can bid this same contract (for instance 1NT) before N but not before S. So in the rare cases where the NS and EW sides are not the same, the results will take some manual inspection.
- “NS:NS 23S,N 23H”: Only North makes 3 hearts.
- “EW:NS 23S,N 23H”: This time the result is the same when EW open the bidding.

DealerPar() and SidesPar() give each par contract as a separate text string:

- “4S*-EW-1” means that E and W can both sacrifice in four spades doubled, going down one trick.
- “3N-EW” means that E and W can both make exactly 3NT.
- “4N-W+1” means that only West can make 4NT +1. In the last example, 5NT just making can also be considered a par contract, but North-South don't have a profitable sacrifice against 4NT, so the par contract is shown in this way. If North-South did indeed have a profitable sacrifice, perhaps 5C*_NS-2, then par contract would have been shown as “5N-W”. Par() would show “4N-W+1” as “W 45N”.
- SidesPar() give the par contract text strings as described above for each side.

DealerParBin and SidesParBin are similar to DealerPar and SidesPar, respectively, except that both functions give the output results in binary using the `parResultsMaster` structure. This simplifies the writing of a conversion program to get an own result output format. Examples of such programs are

ConvertToDealerTextFormat and ConvertToSidesTextFormat.

After DealerParBin or SidesParBin is called, the results in parResultsMaster are used when calling ConvertToDealerTextFormat resp. ConvertToSidesTextFormat.

Output example from ConvertToDealerTextFormat:
“Par 110: NS 2S NS 2H”

Output examples from ConvertToSidesTextFormat:
“NS Par 130: NS 2D+2 NS 2C+2” when it does not matter who starts the bidding.
”NS Par -120: W 2NT
EW Par 120: W 1NT+1” when it matters who starts the bidding.

CalcPar	**CalcParPBN**
`struct ddTableDeal dl`	`struct ddTableDealPBN dlN`
`int vulnerable`	`struct ddTableResults *tp`
`struct ddTableResults *tp`	`int vulnerable`
`struct parResults *presp`	`struct parResults *presp`

CalcParPBN is just like CalcPar, except for the input format.

Each of these functions calculates both the double-dummy table solution and the par solution to a given deal.

Both functions are deprecated. Instead use one of the CalcDDtable functions followed by Par().

AnalysePlayBin	**AnalysePlayPBN**
`struct deal dl`	`struct dealPBN dlPBN`
`struct playTraceBin play`	`struct playTracePBN playPBN`
`struct solvedPlay *solvedp`	`struct solvedPlay *solvedp`
`int thrId`	`int thrId`

AnalysePlayPBN is just like AnalysePlayBin, except for the input format.

The function returns a list of double-dummy values after each specific played card in a hand. Since the function uses SolveBoard, the same comments apply concerning the thread number `thrId` and the transposition tables.

As an example, let us say the DD result in a given contract is 9 tricks for declarer. The play consists of the first trick, two cards from the second trick, and then declarer claims. The lead and declarer's play to the second trick (he wins the first trick) are sub-optimal. Then the trace would look like this, assuming each sub-optimal costs 1 trick:

9 10 10 10 10 9 9

The number of tricks are always seen from declarer's viewpoint (he is the one to the right of the opening leader). There is one more result in the trace than there are cards played, because there is a DD value before any card is played, and one DD value after each card played.

AnalyseAllPlaysBin	**AnalyseAllPlaysPBN**
`struct boards *bop`	`struct boardsPBN *bopPBN`
`struct playTracesBin *plp`	`struct playTracesPBN *plpPBN`
`struct solvedPlays *solvedp`	`struct solvedPlays *solvedp`
`int chunkSize`	`int chunkSize`

AnalyseAllPlaysPBN is just like AnalyseAllPlaysBin, except for the input format.

The AnalyseAllPlays* functions invoke SolveBoard several times in parallel in multiple threads, rather than sequentially in a single thread. This increases execution speed. Up to 20 boards are permitted per call.

Concerning chunkSize, exactly the 21 same remarks apply as with SolveAllChunksBin.

SetMaxThreads	**FreeMemory**
`int userThreads`	`void`

SetMaxThreads returns the actual number of threads.

DDS has a preferred memory size per thread, currently about 95 MB, and a maximum memory size per thread, currently about 160 MB. It will also not use more than 70% of the available memory. It will not create more threads than there are processor cores, as this will only require more memory and will not improve performance. Within these constraints, DDS auto-configures the number of threads.

DDS first detects the number of cores and the available memory. If this doesn't work for some reason, it defaults to 1 thread which is allowed to use the maximum memory size per thread.

DDS then checks whether a number of threads equal to the number of cores will fit within the available memory when each thread may use the maximum memory per thread. If there is not enough memory for this, DDS scales back its ambition. If there is enough memory for the preferred memory size, then DDS still creates a number of threads equal to the number of cores. If there is not even enough memory for this, DDS scales back the number of threads to fit within the memory.

The user can suggest to DDS a number of threads by calling SetMaxThreads. DDS will never create more threads than requested, but it may create fewer if there is not enough memory, calculated as above. Calling SetMaxThreads is optional, not mandatory. DDS will always select a suitable number of threads on its own.

It may be possible, especially on non-Windows systems, to call SetMaxThreads() actively, even though the user does not want to influence the default values. In this case, use a 0 argument.

SetMaxThreads can be called multiple times even within the same session. So it is theoretically possible to change the number of threads dynamically.

It is possible to ask DDS to give up its dynamically allocated memory by calling FreeMemory. This could be useful for instance if there is a long pause where DDS is not used within a session. DDS will free its memory when the DLL detaches from the user program, so there is no need for the user to call this function before detaching.

Return codes

Value	Code	Comment
1	RETURN_NO_FAULT	
-1	RETURN_UNKNOWN_FAULT	Currently happens when fopen() returns an error or when AnalyseAllPlaysBin() gets a different number of boards in its first two arguments.
-2	RETURN_ZERO_CARDS	SolveBoard(), self-explanatory.
-3	RETURN_TARGET_TOO_HIGH	SolveBoard(), target is higher than the number of tricks remaining.
-4	RETURN_DUPLICATE_CARDS	SolveBoard(), self-explanatory.
-5	RETURN_TARGET_WRONG_LO	SolveBoard(), target is less than -1.
-7	RETURN_TARGET_WRONG_HI	SolveBoard(), target is higher than 13.
-8	RETURN_SOLNS_WRONG_LO	SolveBoard(), solutions is less than 1.
-10	RETURN_SOLNS_WRONG_HI	SolveBoard(), solutions is higher than 3.
-11	RETURN_TOO_MANY_CARDS	SolveBoard(), self-explanatory.
-12	RETURN_SUIT_OR_RANK	SolveBoard(), either currentTrickSuit or currentTrickRank have wrong data.
-13	RETURN_PLAYED_CARD	SolveBoard(), card already played is also a card still remaining to play.
-14	RETURN_CARD_COUNT	SolveBoard(), wrong number of remaining cards for a hand.
-15	RETURN_THREAD_INDEX	SolveBoard(), thread number is less than 0 or higher than the maximum permitted.
-16	RETURN_MODE_WRONG_LO	SolveBoard(), mode is less than 0
-17	RETURN_MODE_WRONG_HI	SolveBoard(), mode is greater than 2
-18	RETURN_TRUMP_WRONG	SolveBoard(), trump is not one or 0, 1, 2, 3, 4
-19	RETURN_FIRST_WRONG	SolveBoard(), first is not one or 0, 1, 2

-98	RETURN_PLAY_FAULT	AnalysePlay*() family of functions. (a) Less than 0 or more than 52 cards supplied. (b) Invalid suit or rank supplied. (c) A played card is not held by the right player.
-99	RETURN_PBN_FAULT	Returned from a number of places if a PBN string is faulty.
-101	RETURN_TOO_MANY_THREADS	Currently never returned.
-102	RETURN_THREAD_CREATE	Returned from multi-threading functions.
-103	RETURN_THREAD_WAIT	Returned from multi-threading functions when something went wrong while waiting for all threads to complete.
-201	RETURN_NO_SUIT	CalcAllTables*(), returned when the denomination filter vector has no entries.
-202	RETURN_TOO_MANY_TABLES	CalcAllTables*(), returned when too many tables are requested.
-301	RETURN_CHUNK_SIZE	SolveAllChunks*(), returned when the chunk size is < 1.

Revision History

Rev A	2006-02-25	First issue
Rev B	2006-03-20	Updated issue
Rev C	2006-03-28	Updated issue. Addition of the SolveBoard parameter ”mode”
Rev D	2006-04-05	Updated issue. Usage of target=0 to list all cards that are legal to play
Rev E	2006-05-29	Updated issue. New error code Ã¢â‚¬â€œ10 for number of cards > 52
Rev F	2006-08-09	Updated issue. New mode parameter value = 2. New error code Ã¢â‚¬â€œ11 for calling SolveBoard with mode = 2 and forbidden values of other parameters
Rev F1	2006-08-14	Clarifications on conditions for returning scores for the different combinations of the values for target and solutions
Rev F2	2006-08-26	New error code Ã¢â‚¬â€œ12 for wrongly set values of deal.currentTrickSuit and deal.currentTrickRank
Rev G	2007-01-04	New DDS release 1.1, otherwise no change compared to isse F2
Rev H	2007-04-23	DDS release 1.4, changes for parameter mode=2.
Rev I	2010-04-10	DDS release 2.0, multi-thread support
Rev J	2010-05-29	DDS release 2.1, OpenMP support, reuse of previous DD transposition table results of similar deals
Rev K	2010-10-27	Correction of fault in the description: 2nd index in resTable of the structure ddTableResults is declarer hand
Rev L	2011-10-14	Added SolveBoardPBN and CalcDDtablePBN
Rev M	2012-07-06	Added SolveAllBoards. Rev N, 2012-07-16 Max number of threads is 8
Rev O	2012-10-21	Max number of threads is configured at initial start-up, but never exceeds 16
Rev P	2013-03-16	Added functions CalcPar and CalcParPBN
Rev Q	2014-01-09	Added functions CalcAllTables/CalcAllTablesPBN
Rev R	2014-01-13	Updated functions CalcAllTables/CalcAllTablesPBN
Rev S	2014-01-13	Updated functions CalcAllTables/CalcAllTablesPBN
Rev T	2014-03-01	Added function SolveAllChunks
Rev U	2014-09-15	Added functions DealerPar, SidesPar, AnalysePlayBin, AnalysePlayPBN, AnalyseAllPlaysBin, AnalyseAllPlaysPBN
Rev V	2014-10-14	Added functions SetMaxThreads, FreeMemory, DealerParBin, SidesParBin, ConvertToDealerTextFormat, ConvertToSidesTextFormat
Rev X	2014-11-16	Extended maximum number of tables when calling CalcAllTables.