Dealer V.2 User Documentation

Table of Contents

Preamble

This release of Dealer, a program originally written by Hans van Staveren and then maintained and extended by Henk Uijterwaal and others (see the Authors section), in the 1990's has been extensively modified by JGM in 2022. JGM has decided to call this release Version 2.0 because the modifications have not tried to maintain backwards compatibility with the original release.
When the program was first written, circa 1989, there were two things that were true then that are not true in 2022: firstly Linux did not yet exist, even Windows 95 was not available and secondly hardware of the day was severely limited.
As of 2022, Linux has pretty much taken over the Unix/Posix space, especially for systems that would be interested in a bridge program. Even Windows, via Windows Subsystem for Linux (WSL) version 2, can run almost all Linux command line programs, and even many Xwindow based programs natively with no porting effort required.
Whereas in the '90s 32 bit systems for personal use were quite expensive, in 2022, 64 bit systems are common and over 1000 times faster than what was available then. JGM has thus taken full advantage of these changes and developed this new version on Linux only, using whatever GNU tools were most suitable, without worrying about Posix, or Windows portability.

This document describes mainly the use of Version 2.0. It does not attempt to duplicate the information available in the original documentation which is also included in the release package. The user should read the introduction and disclaimers in that document to get a sense of Dealer's purpose and limitations.
JGM has written a companion document, titled, "Dealer Maintenance" which documents what he has learned about Dealer internals during the course of his modifications.

Description

Dealer is a program to generate bridge deals that meet certain criteria, as specified by the user.
It can be used to generate hands for bidding practice, simulation, post-mortem discussion and such. Per the original documentation it should not be used to generate hands for tournament play. There are better options, also free software, for that purpose.

V2.0 Release Notes

Major Changes

The Double Dummy Solving function is now 5 times faster than the previous version. Instead of using GIB it uses Bo Haglund's Double Dummy Solver routines.

There is the ability to get the Par result on a deal, also using BH's DDS routines.

There is the ability to evaluate a pair of hands together, using the Optimal Point Count developed by Patrick Darricades.

Dealer can now export the hands in a format suitable for predealing in a subsequent Dealer run. The predeal holdings can be specified on the command line allowing this feature to be shell scripted.

The user can now use a script to run Dealer, and have it study or simulate a variety of conditions without modifying the Input File for each run.

Dealer can now export its results to a disk file in CSV format, for analysis by other programs. Dealer can put the same, or different, CSV format results to the screen, while at the same time writing CSV results to a file.

Some input statements are now handled directly in the Lexer without involving the parser.

Extra Functionality

Dealer can now accept some numbers with decimal fractions. This is useful in the Optimal Point Count evaluation, the CCCC evaluation, the Suit Quality evaluation, and the Modern Losing Trick Count evaluation functions. It also allows the user to redefine the usual HCP counts using fractions. For example Aces can be given a value of 4.5, Tens a value of 0.5 and so forth. The user can then input conditions such as `hcp(west) >= 14.0 && hcp(west) < 18.5`

The evalcontract feature has been fixed so that it works properly and it can also evaluate Doubled and Redoubled contracts (which it could not do before.)

A new Losing Trick Count function has been added.

Variable names can now include the Underscore character.

The user can now specify a Title to be printed on the various reports, and in the PBN file. This helps document how the output was generated and for what purpose.

Minor Bug Fixes

The code for the altcount feature has been cleaned up so that the number used in the altcount command matches the numbers used in the pt0 to pt9 commands.

Several of the reports has been 'prettied' up with board numbers and seat names, and if available a title.

Removed Functionality

All the code conditionally included under 'FRANCOIS' aka Exhaust Mode has been removed.

All the code related to Windows portability has been removed.

The code that searched for a library of GIB deals has been removed. This library does not seem to be available anymore, and the new DDS functionality probably supplants it.

Hans van Staveren's clever hack that avoided modulo division, and thus sped up random number generation by a factor of two, has been removed. See the maintenance documentation if you want to know why.

Conventions Used in this Document

In most user manuals there are certain conventions that are used to make the context more clear. Here are the ones used in this document:

- Dealer **keywords** are in bold.
- In Dealer case is meaningful. **hcp** is a Dealer reserved word. HCP is not a reserved word. **spades** is a reserved word. Spades is not a reserved word.
- Many keywords in Dealer have both the singular and plural form reserved. See the glossary later in this document.
- Examples showing literal input to, and output from, Dealer is in `Courier New` font. When there is a backslash '\' at the end of a line, it means that the line is to be continued with the text that appears next. It is generally the case that there was not room on the page to show the text all on one line as it would appear on the screen.
- Generic terms that have specific meaning in a Dealer context are in *italics*. For example *compass, side, suit,* etc.
- Input that can be omitted in certain cases is shown between square brackets. [...]
- When parentheses **()** are shown in this documentation they are required by the Dealer syntax. The previous version of the documentation used parentheses for two purposes which I always found slightly confusing.
- Where the user is expected to make a choice between one of several terms the list is shown between braces {}.

For example **hcp (** *compass* [,*suit*] **)** can be read as **hcp** is a Dealer keyword that must be entered exactly as shown. The left parenthesis is required. *compass* is a Dealer term referring to the various seat names, {north, south, east, west}. The comma and term *suit* shown between the square brackets is optional and may be omitted. The closing parenthesis is required.
Hence this input: `hcp(south, clubs)` is legal, as is `hcp(west)`. But this input `hcp South` is not legal. The parentheses are missing and the word 'South' with a capital S is not a valid *compass* name.

Definitions of Terms

This section defines some of the terms that have special meaning to Dealer.

- *compass:* one of north, south, east, west
- *side:* one of NS, EW (note the capitals)
- *suit:* one of club, clubs, diamond, diamonds, heart, hearts, spade, spades (singular & plural)
- *strain:* one of notrump, notrumps, or one of the suits
- *holding*: one or more cards in a suit, arranged in descending order of rank preceded by a suit letter. Example: `HAJT432` or `CAKQJT98`

- *deal:* 52 cards in random order (shuffled and dealt)
- *shape*: a 4 character string made up of the digits 0-9 and the letter 'x'. e.g. 4432, 8032, x5x3
- *shapelist:* several shapes joined with plus or minus signs and optionally preceded by the keyword **any.** For example to specify a balanced hand that does not include a 5 card Major: `any 4432 + any 4333 + any 5332 - 5xxx - x5xx`
- *contract*: A string consisting of: the lower case letter 'z', then a digit in the range 1-7, then a capital letter giving the strain from the list {C,D,H,S,N} then 0, 1, or 2 lower case 'x's to indicate whether the contract is un-doubled, doubled, or re-doubled. Ex: `z6Sxx` or `z1N`
- *dotnum:* a number with 0, 1 or 2 digits before a decimal point which can be followed by 0, 1, or 2 digits. Examples: `99.99` or `0.01` or `.5` or `15.` The decimal point is always required; the digits before or after the decimal point may or may not be present.
- *number*: an integer made up of the digits 0-9. No commas, or decimal points.
- *expression*: an expression is made up of other expressions, grouped by parentheses and joined by arithmetic or logical operators. The simplest expression is a single term such a *number* or a bridge function call such as **`hcp(north)`**
- *quoted string* Free form text, between double quotes. e.g. `"Average of North's HCP"`
- *csvlist* Terms that can be printed by the **csvrpt** and/or the **printrpt** action(s). The terms in this list are: any valid Dealer *expression*, a *compass*, a *side*, *quoted strings*, and the keywords **deal**, **trix(deal)** and **trix(***side***)**

The Input File

The input file is where the user specifies what conditions must be satisfied to qualify the deal as 'interesting' and which actions to take when an interesting deal is found. Dealer will keep generating deals at random, and analyzing each one until one of two things occurs: either it produces the number of interesting deals the user called for, or it reaches the limit on how many deals it is required to generate before giving up.
The input file can be thought of as consisting of the following parts: environment set-up, variable definition, condition clause, list of actions to perform, and end of run output actions.
Dealer also understands *expressions* That is, in most places where a single *term* is allowed such as a *number*, or a term such as **hearts(south)** you can also put two or more terms connected by either an arithmetic, or a logical, operator. Most expressions that would be valid in C are also valid in Dealer. Examples: `hcp(south) + hcp(north)` or `hearts(south) + spades(south)` or `hcp(south,hearts)>=hcp(south,spades)and hcp(south)+hcp(north)>26`

Environment Set-up & Scripting

generate *number* The maximum number of random deals to try. Terminate the program when this number is reached, even if there were fewer than **produce** 'interesting' deals.
produce *number* The maximum number of 'interesting' deals to produce. Terminate the program after this many have been found, and their actions completed.
dealer *compass* Set the dealer for the PBN printed output. This value has no impact on the deals produced; it is strictly for documentation purposes. The default is **north**.
vulnerable {NS, EW, both, all, none } Set the vulnerability for the PBN printed output. If **par**

calculations are asked for, this is the value that will be used. **none** is the default.
opener *compass* Set the opener for **opc** evaluations. The default is **west**.
predeal *compass holding* [,*holding* ....] Predeal these cards to this player. As few as one, and as many as 13 cards may be predealt to any number of players.
Example: `predeal south SAQ542, HKJ87, D32, CAK`
An alternate form of **predeal** specification is *suit**(**compass**)**==number* This will ensure that the player gets exactly *number* in the suit, but the card(s) could be any rank. *This feature does not work in my copy of Dealer from Debian. In fact studying the code I cannot find any aspect of the shuffle routine (where the other predeal condition is taken into account) that uses this feature. Not implemented in V2*

pointcount {*list of numbers*} Set the HCP value of the cards in a suit starting with the Ace and down to the Deuce. The word **pointcount** by itself will set all values to zero. Not every card needs to be entered. The purpose of this keyword is to allow the player to redefine the usual 4-3-2-1 point count. For example: `pointcount 6 4 3 2 1` will give Aces a value of 6, Kings 4, Queens 3, Jacks 2 and Tens 1. The remaining cards will be given a value of zero. As of this version (V2.0) the user can also enter numbers with decimal points, so that Aces can have for example a value of 4.5, and Tens 0.5
`e.g. pointcount 4.5 3. 2.0 1. 0.5`
Note the use of the decimal point in all the numbers given.
Using **pointcount** in this way comes with some caveats. The most important being that if you define HCP with a *dotnum* you must use a *dotnum* everywhere that a HCP value is being compared against. It would not work to define the HCP as in this example and then to say `hcp(west)>=13` You would have to say `hcp(west)>=13.0`
See the instructions on using *dotnums*, later in this document.
altcount *number number-list* This keyword is similar to **pointcount** in that it lets the user redefine what is being counted by the **pt0** thru **pt9** keywords. The first number is the number of the alternate point count e.g. `altcount 4` will set the value of all cards counted by **pt4** to zero. The default of **pt4** is to count the number of Aces in a hand so that the default altcount 4 array is set as if the user had entered `altcount 4 1 0 0 0 0 0 0 0 0 0 0 0 0`
If you want to use **pt4** to count something different, then you can redefine this array with an **altcount** 4 command. Similarly for the other alternate counts, 0 - 9. **altcount** can also take *dotnums* as values.
Example: `altcount 9 13 9 5 2 1`
This will make **pt9** count in 'Danny Kleinman' points where an Ace has a value of 13, a King a value of 9, a Queen a value of 5, a Jack a value of 2 and a Ten a value of 1.
title *quoted string* This title will appear on some of the printouts, and also at the end of the run when the statistics are printed out. This allows the user to keep track of what the printouts were intended for.
Ex: `title "Weak NT Bidding Practice Session 1"`

Variables

Dealer allows the user to define variables which can be used as shorthand for complicated expressions. It is important to realize that that is the only thing variables are used for. They are not used to store intermediate results. When the user defines a variable, the variable name is added to a linked list of variables; an expression tree is built that, when traversed, will calculate and return the value of the variable, and a pointer to that tree is associated with the variable name. But the actual variable does not

receive a value at this time. When the variable is referred to in a condition clause or an action statement, the variable list is searched for the matching variable name, and the expression tree is evaluated and the value of the variable is calculated each time it is referred to.
Example: `HCP_NS = hcp(north) + hcp(south)`
`condition HCP_NS >= 23 and HCP_NS <= 29`
A variable named "HCP_NS" is added to the variable list. An expression tree that calculates the sum of the north HCP and the south HCP is created. But no value is stored in the variable. When the condition statement wants to test the number 23 against the variable named HCP_NS, the variable list is searched until the name HCP_NS is found, and the pointer to the expression tree is retrieved. The expression tree that calculates the sum is evaluated and the comparison is done. When the condition statement wants to compare the number 29 against the variable HCP_NS the exact same process is repeated again, starting with the search of the variable list.
This takes slightly more time than just having the condition statement evaluate:
`condition (hcp(north)+hcp(south)>=23)&&(hcp(north)+hcp(south)<=29)`
The first is more convenient for the user, but takes more time because the variable name must first be looked up (twice) before the summation can be done (also twice).
A variable name can be of any length, consisting of the digits 0-9, upper case letters, lower case letters, and, as of Version 2.0, the underscore character. There are two restrictions; first it must start with a letter (upper or lower case), second the variable name cannot conflict with a Dealer reserved word. For example it is legal to have a variables named hcpn, hcpe, hcpw representing the HCP for North, East, and West but not a variable named hcps because both **hcp** and **hcps** are reserved words in Dealer. But hcpS would be a legal variable name because case matters.

Condition Statement

Evaluating the condition statement is the most complicated thing that Dealer does. The user has great freedom to build quite complex condition statements by using the various bridge hand evaluation metrics and joining them with either arithmetic or logical operators, and grouping them with parentheses.
The arithmetic and logical operators are listed after the glossary of keywords later in this document.
The syntax of the condition statement is **`condition`** *`expression`* Parentheses are not required around *expression*, but of course the user will often need to use parentheses to ensure that the various clauses are grouped correctly. This is particularly true if the **condition** expression contains an **or** logical operator.
Dealer only acts on one condition statement. It is not a fatal error to have more than one condition statement, but only the last condition statement specified has any effect.
Example:
```
condition shape(north, any 4432 + any 4333)and hcp(north)>=15
and shape(south, 5xxx + x5xx) and hcp(south)>=8
```

Efficiency Consideration: The condition statement is evaluated in roughly the same sequence as the way it is entered in the input file. If it is not necessary to evaluate all clauses, then Dealer does not do so. For instance if the condition statement uses many clauses joined by **and** then as soon as one of those clauses is false, the whole expression must be false and Dealer does not continue to evaluate clauses that are not necessary. The same is true if the clauses are joined by **or** and one of them becomes true. Since the **tricks**, **dds**, **trix, par**, and **opc** functions are much slower than the other ones in Dealer,

it is much more efficient to put them at the end of the condition statement than at the beginning. They will then not be called unless absolutely necessary. See the maintenance documentation for more details.

Commonly Used Clauses

Most often the condition statement will consist of clauses using common bridge functions, or variables that are shorthand for expressions made up of common bridge functions. (In all of the following discussion assume that both the singular and plural form of the reserved word can be used. For example **hcp** and **hcps** mean the same thing as do **spade** and **spades**).

hcp(*compass***)** or **hcp**(*compass, suit***)**. This will return a value based on the contents of the pointcount array. Unless the user has changed this with a **pointcount** statement, this array will give a value of zero to each card from Deuce to Ten, and the common 4-3-2-1 values to Ace, King, Queen, and Jack.
Ex1: `hcp(west)` Ex2: `hcps(east,club)`

spades(*compass***)**, **hearts**(*compass***)**, **diamonds(***compass*), **clubs(***compass***)**. These functions (and their singular synonyms) will return the number of cards hand *compass* holds in the suit. A way of obtaining the length of the suit in the hand.

controls(*compass***), control(***compass, suit***)** Returns the number of controls in a hand or a suit. An Ace is counted as two controls, a King as one control. Shortness is not counted as a control.

loser (*compass***)**, e.g `.losers(north)`
The number of losers in the hand, which is the sum of the number of losers in each suit. The program does not apply any corrections to the loser-count such as "1 loser less if a player has more aces than queens".
losers (*compass, suit***)**, e.g. `loser(north, spades)`
The number of losers in a suit. The number of losers in a suit is:

- 3 cards or more: 3 - 1 for each of A, K or Q held. (Axx, Kxxx, and Qxxx evaluate to two losers)
- Void: 0 losers.
- Singleton A: 0 losers, any other singleton 1.
- Doubleton AK: 0 losers, Ax or Kx 1, any other doubleton 2.

pt0(*compass***), pt0(***compass,suit***),** this will return a count based on the values stored in the altcount 0 array. If the user has not redefined this count, the array is set giving a Ten the value of 1 and all other cards the value of zero. Thus the default for this count is to return the total number of tens in the hand, or the number of tens in a suit. Even if the user has set this count to something different, using the reserved word **ten** or **tens** will still return the same value as using the keyword **pt0.**
ptN(*compass***), ptN(***compass,suit***),** where **ptN** is one of **pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, pt9.**
These are used in the same way as **pt0**, but refer to the values in the altcount arrays 1 thru 9.
The user can count something different by using the **altcount** keyword to redefine the arrays 1 thru 9. See the description of the altcount keyword in the environment and setup section.

tens, jacks, queens, kings, aces, top2, top3, top4, top5, c13

Alternate, readable, synonyms for **pt0** to **pt9** in order -- the names correspond to what these alternate-counts do count if not overridden with the **altcount** command: number of tens/jacks/queens/kings/aces; numbers of honours in the top 2 (AK), 3 (AKQ), 4 (AKQJ), 5 (AKQJT); "c13" points, with A=6, K=4, Q=2, J=1 (a version of the "Four Aces" or "Burnstine" count using only integers, and with points in each suit that sum to 13, whence the name). Example: `top5(east,spades)` number of honours that East holds in the Spade suit (unless alternate count number 8 has been overridden with **altcount** 8).

score(*vulnerability, contract, tricks* **)**
Returns the positive or negative score that declarer will make if the given contract, at the given vulnerability condition, is played and the given number of tricks are made. The syntax for "*contract*" is of the form "z3N" for 3 no-trumps, "z7C" for 7 clubs, etc; the leading "z" is needed. This version 2.0 also includes the new ability to specify doubled contracts, z6Hx, and re-doubled contracts, z5Sxx. The result will be positive if the contract makes, negative if it goes down. *tricks* can be the result of a double dummy (**dds** or **tricks**) call.
Ex:
```
S4HxV = score( vul, z4Hx, dds(south,hearts) )
 action printes("S4HxV: ", S4HxV," , ",dds(south,hearts), " :: " ),printoneline
```
Gives this output (In actuality the score, tricks, and deal are all on one line)
```
S4HxV: 990 , 11 :: n K85.QJ9874.43.Q3 e J92..AK987.T8654 s AQ4.AK652.2.AJ92 w
T763.T3.QJT65.K7
S4HxV: 790 , 10 :: n JT3.AKJ642.T72.K e 874.87.Q93.A9863 s K96.Q5.AK65.QJ54 w
AQ52.T93.J84.T72
S4HxV: -200 , 9 :: n KT.A97432.AJT4.2 e .KJT5.KQ75.QJT83 s AQJ5.Q6.962.A974 w
9876432.8.83.K65
S3HxV: -500 , 8 ::
      n AQJ6.AQT9.K873.J e K84.8642.AQJ6.Q8 s 753.KJ73.92.K532 w T92.5.T54.AT9764
```

imps(*scoredifference* **)**
Translates a score-difference into IMPs (International Match Points). The difference, of course, can be positive or negative, and the result of "imps" will then have that same sign.

hascard (*compass, card* **)** `e.g. hascard(east, TC)` returns 1 or 0 depending on whether east holds the 10 (T) of clubs. Since **hascard** returns the value 1 if true and 0 if false, it is possible to **average**, or **frequency** count, the number of times **hascard** returned true over the sample size of produced deals.

shape(*compass* , *shapelist* **)** The user is able to specify up to a total of 32 different shapes in the input file. Hans implemented **shape** in a mind-bendingly clever way, so that no matter how complicated the *shapelist*, it takes the same amount of time to evaluate whether the hand meets the shape criteria or not. The user should prefer to use the **shape** function whenever possible, rather than checking the individual suit lengths with the **club, diamond, heart, and spad**e commands.
Example: `shape(north, xx6x + xx7x - any 4xxx - any 5xxx)`
The above might be used to specify that North has a weak two in diamonds. See the Complete Example on page 17 for more ways of using **shape**.

rnd(*number* **)** Generate a random number between zero and *number*. This function has nothing to do with bridge hands. It is used primarily to verify the functioning of the RNG. If you generate say one

million random numbers between 0 and 100 and average them, you should get something close to 50 in about a 1.5 secs.

Ex: Input file: `produce 1000000  action average "RNG Test" rnd( 100 )`
Result:
```
RNG Test : Mean=49.59, Std Dev=28.86, Var=832.75, Sample Size=1000000
Time needed    1.294 sec
```

Note that because the RNG generates a uniform distribution the standard deviation should always be approximately 28.86 (if between 0 and 100. 0.2886 if between 0 and 1).

Clauses Using Dotnums

All of the values used by Dealer internally are integers. This presents a bit of a problem when trying to use metrics where fractions are possible. The solution used in the past has been to have the user multiply all the numbers that would be fractions by 100. Here is how the **cccc** and **quality** keywords were described in the previous version of Dealer:<quote>
cccc (*compass***)**
quality (*compass, suit*)
Both **quality** and **cccc** use the algorithms described in _The Bridge World_, October 1982, with the single exception that the values are multiplied by 100 (so that we can use integers for them). Thus, a minimum opening bid is about 1200, rather than 12.00 as expressed in the text. </quote>
Ex1: `cccc(north) >=1325 & cccc(south) <= 650` refers to cccc values of 13.25 and 6.5

This description is still valid; the user can continue to work in this way. As an option the user can now also enter numbers with decimal points in them such as 12.00, 6.5, or even 11.37 if that would make sense. When the input file is parsed all such numbers are immediately converted to integers by multiplying them by 100 so that internally, the two expressions:
`cccc(north) >=1325 && cccc(south) <= 650` and
`cccc(north)>=13.25 && cccc(south) <= 6.50` have exactly the same effect. The user is spared the inconvenience of having to multiply numbers by 100. However if the user uses any of these values in print statements, or average statements, it is as if he had multiplied by 100 since Dealer has no notion internally of any number that is not an integer.

ltc(*compass*) **ltc(** *compass, suit* **)** Returns a 'modern losing trick count' which allows for half losers in a suit. Whereas the **losers** keyword evaluates AQ- as one loser, and Qxx as two losers, the **ltc** function evaluates them as half a loser, and two and a half losers, respectively. The user can use either *dotnums* or losers multiplied by 100 in the input file. Example: `ltc(west) < 5.5` or `ltc(west) < 550` both mean the same thing.

opc(*side***)** This will return the 'Optimal Point Count' as defined by Patrick Darricades in his books on the subject. To use this function requires the user to have the perl programming language installed, and to have a copy of JGM's Optimal Point Count perl program (which is included with Dealer 2.0). This point count also counts in half points and thus uses *dotnum*s in the same way as the **ltc, cccc**, and **quality** functions. One of the unique aspects of OPC is that it evaluates a pair of hands together,

assigning extra points for fit and honors in partner's long suit(s), extra support points for shortness with 3 or 4 card trump support, and deducting points for honors facing shortness in partner's hand. It does not just evaluate two hands in isolation and then add the two values together to get a total. See PD's books for a more complete description.
Because this function runs an external program, (much the same way as the **tricks** function runs the external GIB program) it is much slower than simply counting hcp, or even cccc points.

Double Dummy Solvers (DDSx2, Par, and GIB Tricks)

dds(*compass, strain* **)** Calls Bo Haglund's Double Dummy Solver library to calculate the number of tricks that can be taken by the given hand, in the given strain (suit or notrump). Dealer can call DDS in two different ways; if the number of tricks is required for fewer than 5 different *compass-strain* combinations, it is more efficient to use DDS's single result solution (mode 1); if more than 5 different *compass-strain* combinations are to be analyzed, then it is more efficient to use DDS's mode in which it calculates all 20 possible *compass-strain* results for the given hand in one go (mode 2). The single result method is the default, as that is the most likely scenario. The user can force the 20 result method via the -M command line switch. (See command line parameters later in this document. See also the maintenance documentation for a discussion of the times taken for the various methods of double dummy solutions.)

par(*side* **)** Calls Bo Haglund's Double Dummy Solver to calculate the par result on the deal. If the user asks for a par result, the DDS mode is automatically set to mode 2 since the par calculations require DDS to calculate all 20 results before returning the par result. DDS when calculating the par result, returns not only the par score (e.g. -300), but also the contract (e.g. 5Cx) that led to that score. But Dealer does not (yet) return the contract information to the user. The user can specify the vulnerability when asking for the par result via the **vulnerable** keyword. The default is None Vulnerable.

Example: This Input File extract

```
condition shape(south, x5xx + x6xx +x7xx) && shape(north, x3xx + x4xx ) &&
hcp(south)+hcp(north) >=24
action printrpt(NS, par(NS), hcp(north)+hcp(south) ),
average "Par NS = " par(NS)
```

will generate output like this:

```
 n AQ93.K953.K42.KT s K75.JT862.A87.Q7 ,450,25
 n K.JT53.KT752.AK7 s .KQ942.AJ8643.83 ,300,24
 n AT842.QJ9.A92.K3 s 976.KT743.KJ5.A5 ,430,25
 n KQJ76.AK8.8.AKT5 s T82.J9654.76.QJ6 ,300,24
 n 9.T93.K953.AQJT3 s A2.AKQ8765.2.K52 ,990,26
Par NS = : Mean=  528.0000, Std Dev=  408.2259, Var=166648.4211
```

tricks(*compass, strain* **)**
Runs GIB's double-dummy engine to compute the number of tricks that, at double-dummy par, will be taken by the given declarer in the given strain (suit or notrumps). GIB must be installed separately for this function to work. The executable must be named `gibcli` (lower case) and be in the directory `/usr/games`. (This is a change from the version currently distributed by Debian.) In addition GIB requires that the file `eval.dat` be located in the user's current working directory, not just in the directory where GIB, or Dealer, is installed. If you get an error from GIB when using this function

check to see if you have the `eval.dat` file that comes with GIB copied to your current working directory. As of Dealer version 2.0 there should be no need to use this function, since DDS mode 1 does the same thing, is 5 times faster and does not require extra files or an extra program. A user might have several Dealer input files that he does not want to change, or he might have trouble getting DDS to run, but those are the only possible reasons to continue to use this function.

Arithmetic and Logical Operators

In the condition statement (and also in the variable definitions) the various terms making up the complete condition are joined together by either arithmetic operators (+ and - being the most common ones used, but *, / and % are also available) or by logical operators. The logical operators come from the following list:
{**and, &&, or, ||, not, !, >, >=, <, <=, ==, !=, ?:** }. The use of the arithmetic operators should present no surprises, but some of the logical operators need explaining if you are not familiar with C programming.
&& is a synonym for '**and**'. Note that you need both ampersands, just entering one will result in an error. || is a synonym for '**or**'. Again you need to enter two vertical bar symbols. **!** is a synonym for '**not**'. '>' means 'greater than' , '>=' means 'greater than or equal' and similarly for less than (<) and less than or equal (<=). To test for equality you need to enter **TWO** equal signs(==). It is a common error for people not familiar with programming to enter for example hcp(north, spades) = 7 to ensure that north has a good spade suit. This will result in an error; what you need to say is:
`hcp(north, spades) == 7` This will test to see if north has at least AQJ or AK in the spade suit.
The opposite of == is != i.e. "not equal'.
That leaves the mysterious **?:** Dealer uses this combination as a kind of if statement.
It can be useful if you want to chose the larger (or smaller) of two quantities. For example the score keyword needs the number of tricks taken, and the contract and vulnerability
i.e. `score(vul, z3Sx, MajTrx).` You want the variable MajTrx to be determined from the deal via a call to the DDS routines as in `MajTrx = dds(north,hearts)` But suppose you are trying to compare playing in a 4-4 spade fit vs a 5-3 heart fit. You might want to call **score** with either the tricks in spades or the tricks in hearts, whichever is greater. You can't do that with a >= operator because that one just gives a true or false result.
So `MajTrx = dds(north, hearts) > dds(north,spades)` will set MajTrx to 1 if hearts gets more tricks than spades, and 0 if it's the other way round. What you want is:

```
MajTrx=(dds(north,hearts)>dds(north,spades))? dds(north,hearts):dds(north, spades)
```

This will test hearts vs spades, (the clause to the left of the **?**) and set MajTrx to the heart tricks if the condition is true, and to the spade tricks if the condition is false.
So the syntax for **?:** is `(logic test) ? action-if-true : action-if-false` This **?:** operator is quite powerful; in C you can daisy chain conditional operators together one after the other. If you want to know more read up on C's 'ternary operator', sometimes called the 'trinary operator' or 'conditional evaluation' operator.

The Action Statement

General Description

Only the last action statement entered in the input file has any effect, even though several action statements can be entered. If there is no action statement entered, the default action is **printall.**
The action statement can consist of a list of actions, separated by commas.

When the condition statement as entered by the user evaluates to true for a given deal, Dealer will then perform the action(s) listed in the action statement. Action(s) always involve producing some form of output; the output can take place immediately at the time the deal is produced, or the action can create some data that will not be output until the end of the run.

The output from the actions **export** and **csvrpt** go to a file that the user specifies via one of the command line switches (or to stdout). The output of all other actions goes to stdout, generally the user's screen. It usually does not make sense for there to be more than one action generating screen output at the time each deal is produced because then the several output streams are all mixed together. For this reason the most common occurrence is to have only one print action in the action list. {But **export** and **csvrpt** actions can be mixed with each other and with print actions since they can each be output to their own file.}
Ex: `action printoneline`
An exception is using the **printes** action as a debugging tool, while keeping the 'real' action also active. See the example below.
However it is common to have one action that produces immediate output and then (possibly several) other action(s) such as **average** and **frequency** that produce output at the end of the run. Actions that produce output at the end of the run do not interfere with each other so you can have as many of them as you want.
Here is an example of an action statement (using a variable defined earlier) that illustrates how this is done:

```
action  printes("vul 6D=  ",myscore1u, \n),
        printoneline,
        printes(".............", \n),
        average "My 6D score" myscore1u,
        frequency "My Tricks" (myTricks, 6, 13)
```

For each 'interesting' deal the above action list will print: a debugging value, then the whole deal in single line format, then a row of dots. If the user has asked for 100 'interesting' deals then this pattern will be repeated each time, generating 300 lines of output.
Then at the end of the run Dealer will print one line with the average score and then a frequency table consisting of a heading line and 10 lines of of numbers.

Note that it is not necessary to put each action on a separate line.
A further note is that if you want to save the stdout output to a file, you re-direct stdout using standard Linux command line syntax.

Actions That Produce Output at End of Run

These actions are: **average**, **frequency**, **evalcontract**, and **print**.
average [*quoted-string*] *expression* The keyword **average** is followed by an optional *quoted string*, then by an *expression* whose average is to be calculated. Note that the square brackets shown here do not appear in the input file, they merely indicate that in this instance the *quoted-string* is optional. The expression may be enclosed in parentheses but this is not necessary if the expression does not require it. The expression may be a single variable or any mix of variable(s) and Dealer functions, much the same as any expression that would appear in a condition statement.

Example: (Note the comma at the end of the first line. Commas are necessary to link several actions into a list.)

```
action average "North GIB Tricks"  Ntr4,
       average "South GIB Tricks"  Str3
```

Output:

```
North GIB Tricks: Mean=9.20, Std Dev=2.59, Var=6.70, Sample Size=5
South GIB Tricks: Mean=8.34, Std Dev=1.30, Var=1.70, Sample Size=5
```

The extra information regarding the standard deviation and sample size is new in Version 2.0 Previous versions of Dealer did not do this.

frequency [*quoted-string*] **(** *expression* **,** *lowerbound***,** *upperbound* **)** The keyword **frequency** is followed by an optional *quoted string*, then a left parenthesis, which is required in this case, then the expression to be evaluated, a comma, a number representing the lowerbound of the table, a comma, then a number representing the upper bound of the table, then the closing right parenthesis.
During the run, for each 'interesting' deal, Dealer will count the number of times that each value of the expression occurs and generate a table accordingly. If the difference between upperbound and lowerbound is 'n' there will be n+1 slots in the table, plus two more to hold the counts that occur outside of the specified range. This is probably best explained with an example. The following shows the output when Dealer was asked to produce 1000 hands at random with no additional condition attached:
Here is the action statement:

```
action frequency "HCP North Between 15 and 17" (hcp(north), 15, 17)
```

And this is the end of run output:

```
Frequency HCP North Between 15 and 17 :
Low            849
   15           47
   16           33
   17           21
High            50
```

Dealer also has a '2 Dimensional' version of a frequency count; instead of just counting the number of times one thing occurs, it can count the number of times two things occur and then plot one versus the other.
The syntax is:
frequency [*quoted-string*] **(***expression1* **,** *lowbound1***,** *upbound1, expression2, lowbound2, upbound2***)**

For example we might compare the number of controls in a hand to the number of high card points in the hand to see how they relate:

```
action frequency "Controls(across) Vs HCP(down)"
          ( hcp(north), 10, 23, controls(north),1,9)
```

Since there are never fewer than zero, or more than 12, controls we limit the top row accordingly. The example limits the range of hcp under discussion to save space on the page.

Here is the output of the above action statement for 5000 hands:

```
Frequency Controls(across) Vs HCP(down):
```

	Low	1	2	3	4	5	6	7	8	9	High	Sum
Low	402	599	823	367	80	0	0	0	0	0	0	2271
10	0	25	107	210	118	0	0	0	0	0	0	460
11	0	3	58	166	143	48	0	0	0	0	0	418
12	0	2	26	128	170	74	11	0	0	0	0	411
13	0	1	8	72	154	116	11	0	0	0	0	362
14	0	0	2	34	106	115	39	0	0	0	0	296
15	0	0	1	13	55	110	48	15	0	0	0	242
16	0	0	0	2	26	61	73	19	2	0	0	183
17	0	0	0	1	13	27	46	34	1	0	0	122
18	0	0	0	0	3	19	34	26	7	0	0	89
19	0	0	0	0	2	7	22	24	7	2	0	64
20	0	0	0	0	0	1	10	16	17	0	0	44
21	0	0	0	0	0	0	2	9	7	3	0	21
22	0	0	0	0	0	0	1	4	3	2	0	10
23	0	0	0	0	0	0	0	0	0	3	0	3
High	0	0	0	0	0	0	0	0	0	4	0	4
Sum	402	630	1025	993	870	578	297	147	44	14	0	5000

evalcontract (*side, contract, vuln* **)** The **evalcontract** action will cause Dealer to use DDS to find the number of tricks that can be taken in the given strain by the given side, for each 'interesting' deal.
There can be only one **evalcontract** action per side; there is no error message if there is more than one, but there is only one set of counters per side to store the **evalcontract** counts in. So two **evalcontract** statements for the same side will cause the counters to be twice as large as they should be and you will get garbage results.
The **evalcontract** code calls DDS with *compass*=**west** if the side is **EW** and with *compass*=**south** if the side is **NS**. This implies that if the user wants to know how many tricks were taken on average, as well as the average score, he should use the same compass direction in his own **dds** call to take advantage of caching, otherwise the run will take twice as long.
Dealer will then keep track of the number of times each trick count was achieved. At the end of the run it will generate a score for each of these trick counts (from 0 to 13) and figure out the average.
Below is an example in which the each side has 17 - 23 HCP, NS are balanced and want to play 2NT, and EW have a heart fit and take the push to 3H.

```
produce    1000
title " Eval NS in 2NTx vul and EW in 3Hxx nv"
condition ( hcp(north)+hcp(south)>=17 && hcp(north)+hcp(south)<=23 && hearts(east)+
hearts(west)>=9 && (hcp(east)+hcp(west))>=17 &&(hcp(east)+hcp(west))<=23)
action
         evalcontract ( NS , z2Nx , vul ),
         evalcontract ( EW , z3Hxx , nv  )
```

The Result:

```
Contract 2Nx V by NS Average Result = -519.83, Success pct= 15.30, Fail pct= 84.70
Contract 3Hxx  by EW Average Result =  539.92, Success pct= 70.20, Fail pct= 29.80

 Eval NS in 2NTx vul and EW in 3Hxx nv
Generated 18625 hands
Produced  1000 hands
Initial random seed 113
Time needed   27.874 sec
```

It is interesting to also look at the time taken as reported by the end of run statistics.
`Time needed   27.874 sec`
The bulk of the time was taken by the 2000 calls to the DDS library.

For comparison to produce and analyze 10,000 deals to generate the 2D frequency plot took only 0.038 seconds. This illustrates the relative slowness of the DDS family of functions. And note also that Version 2.0 uses DDS which is 5x faster than GIB as used in the previous version. When you are used to 1 sec or less response times, 27 secs feels like an eternity. :) !

print(*compass [,compass, compass...]* **)** This is the action that is used to print the different compass directions on different pages. From one to four compass directions may be specified, but the usual case is to specify two. Thus one partner can have all the 'interesting' hands for one direction, while the other partner gets the hands for the direction opposite. This is ideal for bidding practice. The title, (if there is one), player name, and board number are also included on this report. Here is an example of the action statement

```
action print (south, north )
```

and the output(with title):

```
Bidding Practice National Teams Session 1

North hands:

   1.                2.                3.                4.
J T 6 3           J 6 4 2           7                 K J 2
K 2               A T               J T 3             7 4 2
Q T 7 6 5         J 7 4 2           J T 9 8 3         7
8 2               J 9 6             A J 9 8           J 9 7 6 4
2

   5.                6.                7.                8.
K T 9 8           A J 8             7 4               J 9
K 6 5 2           Q 8 7 3           J 9 3             A T 9 2
J 7 6 5 2         7 6 2             Q 9 5 2           Q 7 2
-                 T 5 2             T 8 6 5           T 6 4 3
```

<there is a form feed that is output here. This should cause a new page on most printers>

```
Bidding Practice National Teams Session 1

South hands:

   1.                2.                3.                4.
9 8 2             A Q               A J T 9 8 6 3 2   A 9 5
A T 3             9 7 4 3 2         8 4               A Q T 6
J 4 2             A Q 5 3           Q 5               J 8 3 2
K Q 5 3           K 2               2                 K 3


5.                6.                7.                8.
J 4 2             -                 T 6               7 6 3 2
A T 9 8 7         A K 6 5 2         A T 7 6 5 2       K 8 4
A                 Q T 9 8 5         J 8 6 3           K T 5 4 3
A K J 6           A Q 6             7                 K
```

Notice that even though south appeared before north in the action statement the north hands were printed first. It is always the case that the hands (if asked for) are printed in player number order: North, East, South, West.

Actions Outputting for Each Interesting Deal

These actions are: **printpbn, printall, printcompact, printoneline, printside, printew, printns, printes, export, csvrpt,** and **printrpt**

printpbn Print the deal in PBN format, with some extra fields defined for documentation. If a title has been entered it is also output to the PBN report, along with the dealer and the vulnerability. The primary use of this action is to export the deal in a format understood by other software.

printall Output each deal, 4 hands across in the order North, East, South, West. This is the default action if nothing else is specified. In Version 2.0 **printall** also puts out the title, and the seat names on the report as well as the board number.

```
                  [Print All Misfit Example]
  1. North          East              South             West
    K Q T 9 3       J 5               4 2               A 8 7 6
    5               A K J 8 7 2       9 6 4 3           Q T
    A K T 9 5 2     J 8 4 3           7                 Q 6
    A               3                 K Q J T 8 4       9 7 6 5 2

  2. North          East              South             West
    K Q 5 3 2       J 8               T                 A 9 7 6 4
    A 5             K T               Q J 8 3           9 7 6 4 2
    A 3             T 9 6 5 2         K Q J 7 4         8
    K 5 3 2         Q J T 4           9 8 6             A 7
```

printcompact [**(***optional expression***)**] This action prints the deal in a compact format, 4 lines per deal. It has been used in the past primarily to output the deal in a format GIB will accept for Double Dummy analysis. The optional expression can be used to show something like the number of tricks.
Example: `printcompact (dds(north, spades))` Gives the output:

```
n AT3.JT2.QT6.K942
e 95.Q54.J8742.QJ8
s Q764.K63.AK53.A7
w KJ82.A987.9.T653
9
```

The title is NOT output as part of this report as it is often used as the input to other programs.
printoneline[**(***optional expression***)**] The output from this action is very similar to the printcompact output but it is all on one line. Example: `action printoneline (dds(north,spades))`
Result: (The number of tricks North can take in a spade contract is at the end of the line.)

```
n KQ6.K763.AK85.AT e AJ72.J2.Q763.982 s T954.AT84.94.KQ4 w 83.Q95.JT2.J7653 10
n KT85.J87.AK8.Q53 e J3.QT.Q432.A8764 s AQ6.AK543.965.K2 w 9742.962.JT7.JT9 11
n KJT64.AJ3.J8.AQT e 9875.986.T4.7643 s Q3.K42.AK72.KJ92 w A2.QT75.Q9653.85 12
```

This format is quite efficient, and it is convenient to read also. Each hand takes exactly the same number of characters to show regardless of distribution, so it looks good on the screen. The title is NOT output as part of this report as it is often used as the input to other programs.

printside(*side***), printew, printns** These three actions are all related. They print only two of the 4 hands side by side. **printside(** *EW* **)** and **printew** do the same thing, as do **printside(** *NS* **)** and **printns**. The output looks like this:

```
    [Example of Reports Title  ]
   1. South              North
     J 9 8 2             Q 6
     K                   Q T 8
     Q T                 A K 9 3 2
     A Q J T 9 5         8 7 6

   2. South              North
     5 3                 A K Q 9
     K J 9 2             A 8
     Q 8 6 3 2           A T 7
     A 9                 K 8 4 3
```

The previous version of Dealer had only the action **printew.**

printes(*expression-list* **)** This action is most often used to print various expressions in order to debug the input file. As shown at the beginning of this section you can have more than one **printes** action intermixed with other actions. The *expression-list* can contain *quoted strings* to describe what is being printed, *variables*, and any other valid *expression.*
Example : `action printes( "North Tricks in Spades ", nSpTrix,`
`", NS HCP= ", hcp(north)+ hcp(south), \n )`
Will produce output like the following, one line per deal that matches the condition statement.

```
North Tricks in Spades 8,  NS HCP= 25
North Tricks in Spades 9,  NS HCP= 25
North Tricks in Spades 10,  NS HCP= 24
```

printes does not automatically format anything for you. If you want spaces or commas between the expressions you must put them in yourself. **printes** also does not put out newlines; if you don't put them out then all of the output will just be one long line. Notice that the newline expression, **\n,** is NOT between quote marks.

export(*side* **)** or **export(** *compass* **)** The intent of this action is to output the side, or the compass, in a format that can be re-input to Dealer as a predeal argument. This allows the user to generate hands that meet a certain criteria, and then input those same hands into Dealer again to analyze how many tricks they would take against specified, or random, opposition. See the discussion of command line switches and scripting later in this document. If no -X switch is put on the command line the output goes to stdout the same as any other print related action. However if the user specifies a file name via the -X command line switch, the export output will be put to that file. This would allow the subsequent run of Dealer to be done via a shell script. **export** will output either a side or a compass, but not a complete deal. However you can have more than one export action if necessary.

Example: **export(** EW **)**

```
-E ST865,HT73,DAJ97,C94  -W SAJ4,HKJ9,D853,CAQT7
-E HJT7542,DKJT7,CKT3  -W SKQ9,HA8,DAQ43,CQ985
```

Example: **export(** south **)**

```
-S SK632,HK654,DA97,C87
```

Notice that the format of the output is compatible with the format of the **predeal** *holding* specifications.

csvrpt(*expression-list* **)** This action is described in its own section later in this document (p20).

printrpt(*csvlist*) This action is exactly the same as **csvrpt** but it goes to stdout (the user's screen unless redirected) only. Allows the user to print arbitrary expressions and selections of hands to the screen, while still having a separate file for data output. Think of **printrpt** as a combination of **printoneline** and **printes** in one action.

Example: `action printrpt(NS, "S:",trix(south), EW,"W:", trix(west)   )`

Output Result:*(each deal is on one line. \ means line continued)*

```
 n KQ963.QJ4.J6.Q54 s 52.52.AKQ5.KJT62 ,'S:',9,8,6,8,7, \
e A4.A9873.T72.A83 w JT87.KT6.9843.97 ,'W:',4,5,7,5,4
 n J5.J9753.T.AJT52 s AT76.A8.KQ72.873 ,'S:',9,6,8,7,8, \
e K92.K6.A8653.964 w Q843.QT42.J94.KQ ,'W:',4,7,5,6,5
```

CSV Reports

csvrpt([*expr*|*string*|*side*|*compass*|**deal**]**)** and **printrpt(**[*expr*|*string*|*side*|*compass*|**deal**]**)**

*The following description applies to both the **csvrpt** and the **printrpt** actions. The sole exception being that the **csvrpt** action can output to either the screen or a specified file, whereas the **printrpt** action will only output to the screen.*

The original version of Dealer was primarily focused on printing out bridge hands for a human to read. But Dealer has also proven very valuable in simulation studies; in such studies it is often a downstream computer program that will want to process the Dealer output. In the previous version of Dealer this could be done with a combination of **printoneline** and **printes** actions, but it was somewhat tedious and awkward. The PBN report could also be used for this purpose, but to use it needs special PBN input software; you cannot just import the report to a spreadsheet or database.

This version of Dealer has the ability to print information to an arbitrary filename (or stdout) in CSV (Comma Separated Value) format. There are many programs, such as Libre Office Calc, Excel, and various SQL databases, that can import CSV formatted data directly.

The file to write the CSV output to, is specified on the command line with the -C option, for example `-C /tmp/MyDealerRun.csv` This file is opened in append mode by default. This allows the user to first write to this file various heading lines, such as a Title, a Run Date, and various column headings before running the Dealer program. It also allows the user to concatenate several Dealer runs into one file.

If the file does not exist then it is created. The user can also force the file to be opened in write mode where any existing file is overwritten and not appended to, by preceding the file name with `w:` For example `-C w:/tmp/MyDealerRun.csv`

The type of action that creates a CSV output line is: **csvrpt** (or **printrpt** for screen only output). The **csvrpt** action will automatically put commas between the fields, quotes around the strings, and a newline at the end of each output record. The kinds of output that can be handled by this action are: a) a text string between double quotes, b) any valid Dealer expression, c) from 1 to 4 hands from the current deal, d) the number of tricks that a hand, or side, can take, or e) the number of tricks possible in all 20 declarer-strain combinations for the deal.

Types (a) and (b) simply duplicate the functionality of the **printes** action albeit somewhat more conveniently. Type (c) allows the various hands to be conveniently made part of the output record. Types (d) and (e) allow the user to get the number of tricks a hand (or all hands) can take in all 5 strains with a single action clause. Whereas the **dds** keyword will return the number of tricks possible for one *compass* direction playing in one *strain*, **trix(***compass***)** will return the number of tricks possible in all 5 strains for that compass direction, and **trix (deal)** will return all 20 possible results. ***(deal** is a keyword; compass is one of {**north,east,south,west**})*
The order of the strains is from left to right, Clubs, Diamonds, Hearts, Spades, No Trump. The hands are given in the order North, East, South, West. These results will be a list of numbers comma separated so that they can be easily imported into another program.

Example 1: Print a label, an expression, and the North South Hands

```
action csvrpt ("Full Test", hcp(north)+hcp(south), NS   )
```

Result:

```
 'Full Test',26,n AKJ3.KJ943.T76.J s T9762.A.Q.AK9842
 'Full Test',16,n Q65.T.J98754.J63 s 972.AKQJ986.6.QT
```

Example 2: Some labels and expressions mixed including a call to the double dummy solver and showing the actual hands for one side only:

```
     action csvrpt("South HCP", hcp(south),
                   "North HCP", hcp(north),
                   "N+S HCP", hcp(south)+hcp(north),
                   "Tricks S NT= ", dds(south,notrump),
                   "N S Hands:", NS )
```

will result in output such as the following: (each record is all on 1 line, shown here on 2 lines)

```
 'South HCP',13,'North HCP',13,'N+S HCP',26,'Tricks S NT= ',8,   \
        'N S Hands:', n AKJ3.KJ943.T76.J s T9762.A.Q.AK9842
 'South HCP',12,'North HCP',4,'N+S HCP',16,'Tricks S NT= ',6,    \
        'N S Hands:',n Q65.T.J98754.J63 s 972.AKQJ986.6.QT
```

Example 3: Using the **trix** clause with a compass in the **csvrpt** action. Showing the four hands, but as two sets of two so each side is together. The Input File to Dealer action statement is:

```
action csvrpt (NS,"S:",trix(south), EW, "W:", trix(west) )
```

And the CSV output for two deals looks like this (each deal is printed on 1 line, shown here on 2 lines)

```
 n AKJ3.KJ943.T76.J s T9762.A.Q.AK9842 ,'S:',11,6,8,12,8,   \
 e 854.87.KJ843.T75 w Q.QT652.A952.Q63 ,'W:',1,7,5,0,4
 n Q65.T.J98754.J63 s 972.AKQJ986.6.QT ,'S:',2,5,8,4,6,     \
 e JT43.4.AK32.A754 w AK8.7532.QT.K982 ,'W:',11,8,5,9,6
```

Example 4: Using the **trix** clause calling for all 20 trick counts, and showing all four hands:
Here is the action clause:

```
action csvrpt ( deal, trix(deal) )
```

And this is the output (Again each deal is printed all on one line)

```
 n AKJ3.KJ943.T76.J e 854.87.KJ843.T75 s T9762.A.Q.AK9842         \
w Q.QT652.A952.Q63 ,11,6,8,12,8,1,7,5,0,4,11,6,8,12,8,1,7,5,0,4
 n Q65.T.J98754.J63 e JT43.4.AK32.A754 s 972.AKQJ986.6.QT         \
w AK8.7532.QT.K982 ,2,5,8,4,6,11,8,5,9,6,2,5,8,4,6,10,8,5,9,6
```

You may have noticed that in all of the above there are no commas in the hand strings when either a *side* such as **NS** or a **deal** is wanted. The whole hand string is treated as one value for the CSV file. If you want each hand string to be a separate value you have to call for them separately. That is **csvrpt(NS)** will print `n AKJ3.KJ943.T76.J s T9762.A.Q.AK9842` whereas **csvrpt(north,south)** will print: `n AKJ3.KJ943.T76.J,s T9762.A.Q.AK9842`
Notice the comma before the 's'.

A Complete Example

The following complete example is taken verbatim from the original user documentation. It shows the use of the most common types of conditions and actions.
The intent of this example (as described therein) is to produce some hands that match this situation: You south, hold SAQ542, HKJ87, D32, CAK and the auction starts 1C on your left, 2D by partner, pass on your left. With the predeal command, you can assign these 13 cards to this player. Then you create

expressions describing the 1C opener and the 2D overcall. Finally, you generate a number of hands fitting the conditions and (hopefully) find the solution for your problem.

Here is the final input file to Dealer

```
generate   10000
produce    25
vulnerable ew
dealer     west
predeal    south SAQ542, HKJ87, D32, CAK
west1n = shape(west, any 4333 + any 4432 + any 5332 - 5xxx - x5xx) &&
         hcp(west)>14 && hcp(west)<18
west1h = hearts(west)>= 5
west1s = spades(west)>= 5
west1d = diamonds(west)>clubs(west) || ((diamonds(west)==clubs(west))==4)
west1c = (not west1n) && hcp(west)>10 && clubs(west)>=3
         && (not west1h) && (not west1s) && (not west1d)
north2d = (hcp(north)>5 && hcp(north)<12) &&
          shape(north, xx6x + xx7x - any 4xxx - any 5xxx)
condition  west1c && north2d
action     printall
```

End of Run Statistics

The user can control whether he wants the end of run statistics to be printed by means of the -v command line switch. The default is to print them. Setting -v on the command line turns them off. The end of run statistics look like this:

```
Print All Example
Generated 599 hands
Produced  3 hands
Initial random seed 189709657221267
Time needed    0.015 sec
```

The title is "Print All Example". No title is printed if none was entered.

Command Line Parameters

The command line parameters follow the usual Unix/Linux idiom of a dash then a single letter (no space between the dash and the letter). The single letter is known as the command line option or switch. Both terms are used. Some options take a value, some options do not. The -v option referred to earlier is one that does not take a value. All of the options that Dealer uses are a single character. Some are digits, some are upper or lower case letters. Upper and lower case letters refer to different options. For example the option -v will turn off the end of run statistics, while the option -V will print the version information. The options can be in any order, they can come before or after the input filename.

Example: `dealerv2 -p 10  Descr.misfit -s2`

This command says to produce 10 deals, that the input file is named Descr.misfit and that the starting seed for the RNG is 2.

It is common in the Linux/Unix world for the option -h to print a help message and the option -V to print version information.
Here is the output from `dealerv2 -V`

```
Version info....
Revision: 2.0.0
Date: 2022/02/16
$Author: Hans, Henk, JGM $
```

Here is the help message you get with the -h option. The list of valid options is on the line after the Usage text.

```
--- HELP COMING ---
./dealerv2 Usage: -[options] [input_filename | stdin] [>output_file]
List of Run Time Options (all are optional):
[hmquvVg:p:s:x:C:D:M:O:P:R:T:N:E:S:W:X:0:1:2:3:4:5:6:7:8:9:]
h=Help u=UC_toggle v={Verbose, toggle EOJ stats} m={progress Meter}
q={PBN Quiet mode} V={show Version info and exit}

These next switches all require values either integers or strings
g={override Inputfile Generate} p={override Inputfile Produce}
{s=starting Seed for RNG} {x=eXchangeMode:2|3}
C={Filename for CSV Report. (Precede with w: to truncate, else opened for append)}
N:E:S:W ={Compass predeal holding}  O={Opener(NSEW) for OPC eval Default=[W|S]}
M={dds_Mode: 1=single solution, 2=20x solutions} R={Resources/Threads(1..9)}
P={vulnerability for Par computation: 0=NoneVul, 1=NS, 2=EW, 3=Both}
T={Title in quotes}X={Filename to open for eXporting predeal holdings}
D={Debug verbosity level 0-9; (minimal effect in production version)}
-0 to -9={one word or string in double quotes to set $0 thru $9            \
     script parms in Input File}
--- HELP DONE -
```

The first part of the help message lists those options that do not take a value, the next part lists the options that do require a value. The ones that take a value have a colon after the option letter.
If you set the Debug (-D) option to 1 (or more) you will get the various options and their settings printed out at the start of the run in alphabetical order. (There are also some fields shown that are not specifically options but are related to options, for example the length of the title string, or the MaxRamMB field. Like so:

```
dealerv2 -D1 -T"Example Title" -s11 -XExport.dat -M2 -P1 -R6 -OW -x2 -p5 \
         -g100 -0 pt0 -9 hcp
```

The output showing the Options Settings:

```
Showing Options with Verbosity = 1
      g:Maxgenerate=[100]
      m:ProgressMeter=[1]
      p:Maxproduce=[5]
      q:Quiet=[0]
      s:Seed=[11]
      u:UpperCase=[1]
      v:Verbose=[1 ; 1]
      x:eXchange aka Swapping=[2]
```

```
      D:Debug Verbosity=[1] set to 1
      M:DDS Mode=[2] set to 2
      O:Opener=[W, 3]
      P:Par Vuln=[1]
      R:MaxThreads=[6]
      R:MaxRamMB=[960]
      T:Title=[Example Title],len=13
      N:PreDeal=[]
      S:PreDeal=[]
      E:PreDeal=[]
      W:PreDeal=[]
      X:Fname=[Export.dat]
Showing Script Vars with Verbosity = 1
      [$0]=pt0
      [$9]=hcp
```

You are also shown any script variables that have been set. If none are set then none are shown.

The following describes the effect of each of these option letters in detail. One thing to note is that where a value can be specified in the input file and as an option (for example **generate** or **produce**) the value set on the command line takes precedence. This saves the user the trouble of opening a text editor and changing the input file when only a minor change such as the number to produce is required.

-g number Generate. This option allows the user to override the value in the input file or the default value. The option letter 'g', must be followed by a numeric value.
-m Progress Meter. This option turns on the progress meter. If the run time is long-ish because the condition is difficult to meet, or there are many calls to a DDS function the progress meter can print out the percentage completion during the run. It compares the number of hands produced so far to the number the user has asked for.
-p number Produce. This option allows the user to override the value in the input file or the default value. The option letter 'p' must be followed by a numeric value.
-q Suppress PBN output (useful for testing, then switch it back on when generating the "final" sample).
-s number Starting seed for the RNG. By specifying a starting seed you can get repeatable results by specifying the same starting seed the next time. A value of zero, or no -s option uses the Linux Kernel entropy pool to generate a starting seed.
-u Toggle upper case mode.
-v Toggle verbose mode. This will turn off the end of run statistics.
-x 2|3 Specify e**x**change aka swapping mode. In the previous version of Dealer this was implemented with 3 separate switches, -0 , -2, -3. Version 2.0 implements swapping/exchange mode with one switch which takes one of two values. **-x2** keeps the N/S hands the same while exchanging the E/W hands. **-x3** keeps the North hand the same while permuting the other 3 hands in every way possible.
-C filename CSV output filename for the **csvrpt** command. If no -C filename is entered the command will output to stdout (which may be redirected). See the **csvrpt** description for append or write mode.
-D number Set the verbosity level for the debugging print statements. The usual range is from 0 to 9. A value of zero will suppress all debugging output. The higher the number the more verbose the debugging output will be. With one exception, unless Dealer was compiled with -DJGMDBG as an

option to gcc the -D switch will have no effect, since most of the debugging print statements are conditional on that symbol being defined. The one exception is that if -D1 (or higher) is specified then the list of option settings given earlier is shown.
-M 1|2 Set the DDS mode. 1= Single solution per call. 2=all 20 strain-declarer solutions per call.
-O N|E|S|W set the Opener to be used in **opc** evaluations. Default is West.
-P 0|1|2|3 Set the vulnerability to use for the Par calculations. 0=none, 1=NS, 2=EW, 3=both.
-R 1 - 9 Set the Resources for DDS mode 2 solutions. Sets the number of threads, and the number of threads also sets the amount of RAM required. Adding threads does not speed up DDS Mode 1.
-T **quoted-string** Sets the Title to be printed on the reports. The user should limit the title to no more than 100 characters. If no title is entered on the command line or in the input file then no title is printed.
-X filename Sets the output filename for the **export** command. If no -X filename is entered the command will output to stdout (which may be redirected).
-N, -S, -E, -W *holding-list* These switches allow the user to specify what is to be predealt to any or all of the compass directions. As shown in the example of the export output, a *holding* is a suit letter (capitals) followed by the cards in that suit that are to be predealt, in descending order. A *holding-list* is several holdings joined by commas. Any number of cards in any number of suits can be pre-dealt.
-0 to -9 Gives a value to the script parameter(s) $1 thru $9. The text (or number) that follows this parameter(s) will be substituted for the symbol(s) $1 thru $9 wherever they are found in the input file. See the instructions on using script variables later in the document.

Example Simulation Exercise

An example of the use of these options and the **export** command is as follows:
First run Dealer to **produce** some number (say 100) of NS hands that meet some criteria such as both being balanced with a total of 26.0 to 26.75 Optimal Points between them. Export these hands to a file OPC26NT.exp
Then create another description file that produces 200 deals where the NS hands are predealt. Average the number of tricks that NS can take, in some strain, in this case NoTrump.
Create a shell script that reads the OPC26NT.exp file one line at a time and for each line runs Dealer with the second description file. You will thus create 100 examples of a pair of hands that probably wants to play in 3NT, and you will average the number of tricks that each of those hands can take over 200 deals. A total of 20,000 situations analyzed.
Here is the sample code:
The first Dealer Input File, Descr.26NT

```
generate   100000
produce    100
title "OPC Bal 26opc  -s123 -m -v -D0"
opener west
condition ( shape(west, any 4333+any 4432+any 5332 -5xxx -x5xx ) &&
            shape(east, any 4333+any 4432+any 5332 -5xxx -x5xx ) &&
            26.0 <= opc(EW,notrump) && opc(EW,notrump) <=26.75 )

action export ( EW ),
       printoneline
```

The screen output looks like: (100 lines printed)

```
n KJ2.Q52.K87.KJ87 e 8743.A9.642.QT64 s AT.JT874.953.932 w Q965.K63.AQJT.A5
n 92.A65.9852.JT92 e J3.Q872.AJ64.AK5 s KQT865.K93.T3.73 w A74.JT4.KQ7.Q864
```

The Export file looks like:

```
-E S8743,HA9,D642,CQT64  -W SQ965,HK63,DAQJT,CA5
-E SJ3,HQ872,DAJ64,CAK5  -W SA74,HJT4,DKQ7,CQ864
```

The second Dealer Input File looks like:

```
generate   100000
produce    200
title "OPC Bal 26opc  Verify"
opener west

action  frequency "Bal 2opc" (dds(west,notrump), 6, 12 ),
        average "3NT Success Pct " ( dds(west,notrump) >= 9 )*100,
```

The shell script that runs this file is:

```
#!/bin/bash
usage() {
 echo usage: $0 file_of_predeals Ofile_for_append DealerIn_File \
           e.g. DOP26NT.exp DOP26NT.results Check.DOP26NT
}

if   [ $# -lt 3 ] ; then usage ; exit 0 ; fi
fin=${1}
fout=${2}
fctl=${3}
echo "Opening Predeal file ${fin} Appending to Output file ${fout} \
           using control file ${fctl}"
while read predeal; do
      precnt=$(( $precnt+1 ))
      echo "Starting Record # $precnt "
      echo Analyzing EW hands: ${predeal} >>${fout}
        ./dealerv2 -m -s117 -M1 -D0 -v ${predeal} ${fctl}     >>${fout}
done <${fin}
echo "Done. Results appended to ${fout}"
```

Note how the script reads the file created earlier by the export command and stores the text in the shell variable ${predeal}. The script then passes that text into the Dealer predeal specification via the command line options -W and -E. In effect the line:

```
./dealerv2 -m -s117 -M1 -D0 -v ${predeal} ${fctl}  >>${fout}
```

becomes:

```
./dealerv2 -m -s117 -M1 -D0 -v   \
      -E S8743,HA9,D642,CQT64  -W SQ965,HK63,DAQJT,CA5 ${fctl}  >>${fout}
```

The results in the output file *(${fout})* look like:

```
Analyzing EW hands: -E S8743,HA9,D642,CQT64 -W SQ965,HK63,DAQJT,CA5
Frequency Bal 2opc:
    6         31
    7         49
    8         15
    9          4
   10          1
   11          0
   12          0
3NT Success Pct : Mean= 63.0000, Std Dev= 25.6432, Var= 657.5758, Sample Size=200
Analyzing EW hands: -E SJ3,HQ872,DAJ64,CAK5 -W SA74,HJT4,DKQ7,CQ864
Frequency Bal 2opc:
    6          0
    7          0
    8         46
    9         49
   10          5
   11          0
   12          0
3NT Success Pct : Mean= 54.00, Std Dev= 50.09, Var=  2509.09, Sample Size=200
```

There will be one set of such results for each of the EW hands that were exported to the export file. So what we have done is used Dealer to generate some hands that we wish to analyze, and then we have exported those hands in a format where we can use Dealer to do the analysis for us. We have then automated the whole process with a shell script. In practice we would want the second Dealer file to produce 1000 - 5000 deals and average the results. We would then have generated 100 'Game in NoTrump' hands, and analyzed several thousand cases for each of those hands to see how often they made 9 or more tricks. And we would not have had to spend a lot of time in at the computer manually editing the Dealer input file to change the predeal specification.

Scripting the Input File

Using Version 2.0 of Dealer the user has the ability to make the Input File contain some 'variables' that can be specified at run time. There are 10 of these, and they are written in the Input File as $0 thru $9. When Flex sees one of these in the Input File, it temporarily leaves the Input File and it starts reading from a string variable. It reads as many words as there are in the string variable, passing each word to the parser in turn. When it reaches the end of the string variable, it resumes reading the Input file where it left off. The net effect is that the two characters in the Input File (e.g. $0 etc.) are replaced by as many words as there are in the input string. Many times there will be only one word in the input string but this is not a requirement. So long as the end result is a valid Dealer statement, your creativity can be your guide.
The input strings that Flex reads from are filled by specifying their values on the command line with the options -0 thru -9.

Here is an example where we first generate some Weak NT hands (12 -14 hcp for West) and then some strong ones (West has 15 - 17 hcp). Notice that the script variables $1, $2, and $3 are just single 'words'

aka tokens. But $0 is several 'words' or tokens; however many it takes to specify the shape of the East hand.

This is the relevant part of the file, "Descr.NTscript". $1 is a compass direction(west), $2 and $3 are numbers, and $0 is a shape specification.

```
NTshape = shape($1, any 4333 + any 4432 + any 5332 - 5xxx - x5xx)
condition shape ( east, $0 ) && NTshape &&
   (hcp($1) >= $2) && (hcp($1) <= $3 )
action printew
```

We run dealer passing the values of $0 - $3 on the command line:

```
./dealerv2 Descr.NTscript -s1 -1 west -2 12 -3 14 \
                          -0 "5xxx + x5xx - any xxx0 - any xxx1"
```

A couple of hands produced from the above:

```
 1. West            East              2. West            East
    K 5             A Q J 7 3            K 7             3 2
    K 4 3           A J 8 7              A J 2           Q 7 6 5 3
    K T 8 3         A 6                  A J 9 3         Q T 5
    K T 9 5         8 2                  J 9 3 2         Q 8 6
```

Next we run dealer with different values of hcp on the command line:

```
./dealerv2 Descr.NTscript -s1 -1 west -2 "15" -3 "17"  \
                          -0 "5xxx + x5xx - any xxx0 - any xxx1"
```

And we get hands like these two:

```
   7. West          East            8. West          East
     A K 9          Q 3 2             K 9 7          Q J 6 2
     J 9            A Q 8 7 5         A K 6          T 9 8 5 2
     Q T 9          6 4               A T 3          7 6
     A J T 5 3      K 6 4             Q J T 4        8 3
```

Since we are just doing text substitution we can even have some keywords be set by variables. For example if we want to switch between **HCP** and '**C13**' points we make the condition clause:

```
condition NTshape && ( $9($1)>=$2) && ( $9($1)<=$3 )&& shape(east,$0)
```

and then run Dealer with either:

```
/dealerv2 Descr.NTscript -s1 -1 west -2 "15" -3 "17" -9 hcp  \
                          -0 "5xxx + x5xx - any xxx0 - any xxx1"
```

or

```
/dealerv2 Descr.NTscript -s1 -1 west -2 "15" -3 "17" -9 pt9 \
                          -0 "5xxx + x5xx - any xxx0 - any xxx1"
```

The first run will of course duplicate the strong NT examples above. In the second run, the West hands no longer have 15 - 17 HCP but instead 15 - 17 "C13" points where an Ace=6, King=4, Queen=2 and

Jack=1.
Here are some West hands:

1.	2.	5.	6.
K 5	T 8 5	K 8 5 2	A 3
K 4 3	K 8 6	Q T 4	Q 9 2
K T 8 3	T 9	K Q T	Q J T 4
K T 9 5	A K J T 3	K 9 2	K Q 9 6
4*4 = 16	4+6+4+1 = 15	4+2+6+4 = 16	6+2+3+6 = 17

The values taken on by the script variables are simply passed in on the command line; this allows the user to write shell scripts, and to read the script variables from a file, or to be generated via a *for* loop and so on. See the Example Simulation Exercise section prior, for how this could be done.
By harnessing the power of shell scripting, Dealer V2.0 should allow the user to run unattended simulations, quite conveniently.

Caveat: The scripting variables in the Input File cannot be between quotes; if you write "$1" for example, it loses it's special meaning and just becomes the text string $1. The only case I can think of where this might be a problem is if you want to debug your script and you put the $1 in a **printes** statement.
For example if $1="north" and the North hand has 14 HCP, then the statement:

```
printes( "HCP for compass $1 = " , hcp($1) , \n )
```

will print out: `HCP for compass $1 = 14`

And if you try it like this:

```
printes( "HCP for compass ", $1, " = " , hcp($1) , \n )
```

You will get an error because the parser will see the statement:

```
printes( "HCP for compass ", north, " = " , hcp(north) , \n )
```

and the first `'north'` is not a valid expression in this context.

Thus far we have seen that it is straightforward to use the script variables in the place of numbers or key words, or even strings where the bare string is valid, such as in a **shape** statement. However if you want to have the a string appear in a print statement such as **printes**, or as a label for the **action** or **frequency** output, you must resort to a bit of trickery.

In order to be recognized as a string by the lexer, the string must be in double quotes. So this means that after the lexer has made the substitution, there must be double quotes in the result.
So in order to have for example the date as part of your **average** label, your command line would need to be like this:

```
./dealerv2 -2 ' "Rundate=2022-Feb-12" ' Descr.STRscr
```

This will save the string between the single quotes, including the double quotes, in the script variable so that after the lexer has substituted the string for the $2 script variable, there is a valid Dealer string there, i.e. text between double quotes.

And the Input File could be:

```
action
printoneline,  average $2 controls(west)
```

which would generate output like this:

```
n KQ72.JT532.5.KQ4 e J8654.6.K43.AJ93 s T.K.AQJT96.T8752 w A93.AQ9874.872.6
n A87653.A4.T5.T93 e KQJT2.963.9643.8 s 4.J872.KQJ872.54 w 9.KQT5.A.AKQJ762
Rundate=2022-Feb-12: Mean= 3.0000, Std Dev=1.6997, Var= 2.8889, Sample Size=10
```

Running Dealer

Windows users see the section on Dealer and WSL. Then if necessary come back to this section or the next one.

Dealer itself does not need to be installed in any particlar place; you can install it in your home directory or (if you have super user priviledges) in any convenient path such as `/usr/games`.

The Dealer binary is called: `dealerv2` It does not rely on any other files to execute in most cases; the exception(s) being if you want to use the **opc** function you need to have the external DOP perl program installed in `/usr/local/bin`, and if for some obscure reason you want to use GIB as your double dummy solver, you need to have GIB properly installed as `/usr/games/gibcli` See the section on Double Dummy functions.

At this time Dealer must be installed manually. There is no automatic install script. See the section on installing Dealer in this document. In what follows we assume that Dealer is installed somewhere in your execution path or home directory.

There is not yet a man page for Dealer version 2. This document is it.

Your first Dealer command should probably be `dealerv2 -h - V` as shown. This will produce a rather extensive help message and the Version information and exit. (See the section on Command line arguments.)

To run Dealer and actually produce some hands you first create a file that specifies your conditions and actions. In the Examples provided, these files usually have a name of the form Descr.xxxxx where the xxxxx gives some hint as to what the purpose of the input file is. You then run Dealer like so:

`Prod/dealerv2 Descr.Demo` and you will get some output to the screen.

To run the binary as it exists in the github repo, will most likely require that you have installed the same version of the various libraries that are standard with GCC. This likely means the same version of Ubuntu and the same version of GCC that were present when the binary was built.
If you get a complaint about `libgomp1` not found you can try:
(This will save you having to have the developer tools , and rebuilding Dealer).

```
  sudo apt-get install libgomp1
```

The final character above is a ONE not an ell.
If you are not so lucky you will need to rebuild Dealer from source. See the next section.

Installing and Building Dealer -- Linux

More detail on installing Dealer is in the Maintainer's guide. This is a quick summary for those that just want to use the program. Note that in order to Build Dealer you will need certain developer tools installed. To clone the repository you will need `git` installed.

The easiest way to get DealerV2 is to use `git` and clone the repository. Change to the directory where you would like the copy of the repository to reside, e.g. your Home directory or /usr/games etc and issue the command:

```
git clone https://github.com/dealerv2/Dealer-Version-2-
```

Note the dash at the end of the line is required.

This command will create the directory: Dealer-Version-2-

To see the structure of this directory you can type: `tree -L 1 -F Dealer-Version-2-`

The result will be:

```
├── DDS_Demo/
├── Debug/
├── DebugExamples/
├── docs/
├── DOP4DealerV2.tar.gz
├── Examples/
├── include/
├── lib/
├── LICENSE
├── Prod/
├── README.md
└── src/
```

The primary directories of interest are the Prod/ directory and the Examples/ directory.

The binary for Dealer is Prod/dealerv2. There are many examples of Dealer input files with names beginning with "Descr." in the Examples directory.

If you want to rebuild Dealer from source you need to have some developer tools installed. These are all standard Linux packages so you should be able to install them with your package manager (`apt-get, rpm, dnf, synaptic` etc.) Of course to install these you will need sudo priviledges.

The tools you will need are:

```
bison, flex, make, gcc, g++ and git.
```

Bison should install m4 with it; if it does not you will need to install `m4` also.

`Git` if you choose to clone the repo (recommended) instead of downloading the zip file.

Once the tools are installed, to (re)-build the binary `cd` to the Prod directory and type:

```
make clean ; make allheaders ; make
```

Ignore the warnings about unused variables and the tmpname function and so forth. If you have a reasonably standard Linux setup this should be all you need to rebuild dealerv2 for your system.

For further notes on building the Debug version and verifying the functioning of Dealer see the Maintainer's documentation.

Installing and Building Dealer -- Windows

If you are a Windows user and want to use DealerV2 there is bad news and good news.
The bad news is that Dealer Version 2, unlike Dealer Version 1.x, is a Linux only program. There has been no attempt to maintain portability to other flavors of Unix, or Windows, or even MacOS. If you want a native Windows version you will have to port the code yourself. Since I know nothing about Windows development, and very little about Windows generally, I cannot help you.

The good news is that DealerV2 will run under Windows Subsystem for Linux (WSL). Several people have downloaded the Dealer V2 repository from Github, and built and run DealerV2 under WSL. Here in general are the steps that they found necessary:
1) Install WSL from the Microsoft store. You will need to decide what flavor of WSL to get. From the reports on Github both Untuntu 18.04 and 20.04 will work.
2) Once you have Ubuntu/Linux/WSL working, you will need to install the developer tools necessary to build from source.
3) When the developer tools are all installed, then you can rebuild Dealer as described in the preceding section, "Installing and Building Dealer -- Linux".

Install and Run on WSL without (Re) Building

A user has reported he successfully executed dealerv2 on his windows pc following this process

1: Open a command prompt in administrator mode
2: Execute `wsl --install`
3: Restart computer - after restart windows will install Linux
4: Enter name and password of your own choice and close that window. (*This name and password will be your User ID under Linux. By default this UID will have 'administrator' aka 'superuser' or 'sudo' priviledges.*
5: Open a command prompt and switch to the directory where you want your input files
6: Type `bash` and `enter` and you will now start Linux
7: Enter (You may need to install git. See the next section).

```
sudo git clone https://github.com/dealerv2/Dealer-Version-2-.git
```

(You will have to enter your password from step 4 for sudo to work)
8 Execute command: (This will save you having to have the developer tools , and rebuilding Dealer).

```
sudo apt-get install libgomp1
```

The final character above is a ONE not an ell.
Go to the section after the ," Running Dealer on WSL".

Rebuild Dealer V2 from Source on WSL

The steps are the same as in the previous section up to step 6.
From there install the developer tools you need: `git, make, flex, bison, gcc, g++`
with commands:

```
sudo apt update && sudo apt upgrade
sudo apt install make
```

```
sudo apt-get install gcc
sudo apt-get install -y bison
sudo apt-get install flex
sudo apt-get install g++
sudo git clone https://github.com/dealerv2/Dealer-Version-2-.git
```

Running Dealer on WSL after Install and/or Build

Switch to the downloaded repository

```
cd Dealer-Version-2-/Prod
```

type

```
./dealerv2 -h -V
```

This will give you a brief help message, the version information , and then exit.
To actually have Dealer produce some output you need to give some specifications.
There are several specification files in the Examples directory or you can enter directly from the command line.

From the command line for example:

```
./dealerv2 <ENTER>
produce 5
condition hcp(north> > 12
^D  (Control-D)
```

Control-D is the Linux End of File signal from the keyboard.

Now you should get the output from dealer
From an Examples description file, from within the Prod directory
./dealerv2 ../Examples/Descr.Demo
And you will see 4 sets of hands for East West.

Glossary of Reserved Words in Dealer

Case matters. Order of cards usually matters. Words ending in 's' can be reserved.

Here is a list of all the reserved words in alphabetical order:
ace, aces, action, all, altcount, and, any, average, both, c13, cccc, club, clubs, condition, control,
controls, csvrpt, dds, deal, dealer, diamond, diamonds, EW, export, frequency, generate, gib,
hascard, hcp, hcps, heart, hearts, imp, imps, jack, jacks, king, kings, loser, losers, ltc,
none, not, notrump, notrumps, NS, nv, opener, opc, or,
par, pointcount, predeal, print, printall, printcompact, printes, printew, printns, printoneline, printpbn,
printrpt, printside, produce, pt0, pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, pt9,
quality, queen, queens, rnd, score, shape, spades, spades,
ten, tens, title, top2, top3, top4, top5, trick, tricks, trix, vul, vulnerable

In addition these arithmetic operators and some special characters, have special meaning to Dealer:
`+ - * / % = > < >= <= != == ? : " , ( ) # /* */ //`
The scripting variables $`0,$1,$2,$3,$4,$5,$6,$7,$8,$9` can also be considered to be reserved words.

Authors

These authors are listed in the original Dealer documentation:

- Original code: Hans van Staveren (sater@xs4all.nl)
- Modifications such as the "loser" and "control" function by Henk Uijterwaal.
- GNU random generator introduced by Bruce Moore.
- Exhaust Mode by Francois Dellacherie
- pbn_to_ascii.pl post-processor by Robin Barker.
- cccc() and quality() functions by Danil Suits.
- bug fixes, caching, new keywords/counts, options -2, -3, -l, by Alex Martelli.
- Tricks code by Paul Hankin and Micke Hovmoller.
- Dos/Windows C support, and MSVC++ project files, by Paul Baxter.

Version 2.0 (2022) Author: JGM

- JGM - **Added**: Bo Haglund's DDS solver for Tricks and Par calculations
- Optimal Point Count evaluation
- Modern Losing Trick Count
- Title and Improvements to various reports
- Export function, and Scripting of predeals
- Scripting values and keywords in the Input File
- CSV Report action, and Print Report action
- Doubled and Redoubled contracts for both score and evalcontract
- Score calculation mods
- Decimal point numbers
- Bug fixes to altcount, and evalcontract
- Updated the RNG to GNU rand48
- Modified the seeding method to use the Linux kernel entropy pool.

- JGM - **Removed**: DOS/Windows code, GIB library code, Exhaust mode code, zero52[NRANDVALS] array and code.
- JGM - **Wrote:** Brief manual on operation of Dealer for code maintainers.
- Wrote a new manual for users with both old an new functionality described.

Copyright

Original Copyright Notice from Hans van Staveren

Hans van Staveren's original README file

This program is hereby put in the public domain. Do with it whatever you want, but I would like you not to redistribute it in modified form without mentioning the fact of modification. I will accept bug reports and modification requests, without any obligation of course, but fixing bugs someone else put in is beyond me.
When you report bugs please mention the version number in the source files, and preferably send context diffs if you changed anything. I might put in your fixes, and distribute a new version someday. I would prefer if you did *not* use this program for generating hands for tournaments. I have not investigated the random number generation closely enough for me to be comfortable with that thought. (Note: random number generation has been significantly improved since the original release, for which this disclaimer was written; check the source code for more details).

Hans van Staveren
Amsterdam, Holland

Copyright Version 2.0

The code developed by JGM is copyright JGM (2022) and released under the GNU Public License Version 3 or later at your option.

Copyright (C) 2022, J. G. Morse Richmond B.C. Canada, V7A2R1

The portions of this program developed by JG Morse are free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.