Long Live the HP42S !

Valentín Albillo (#1075, PPC #4747)

If ever there was a praiseworthy HP calculator, the **HP42S** is allegedly the one. An ill-fated model, initially intended as as replacement for the very successful **HP-41** line, it was sadly maimed at a late stage in its development by redefining it to be an **HP-15C** sucessor instead, thus removing all I/O and expandability (save for IR printing), leaving it with *no mass storage capabilities* whatsoever and no way to interact with external hardware or use add-in software. Even worse, it was burdened with exactly mimicking the **HP-41** internal coding for user programs and data, a feature intended to allow it to directly run HP-41 programs loaded from mass storage, but which now just noticeably *slowed down* program execution. Adding insult to injury, though the operating system could merrily handle up to 32 Kb of RAM, it was fitted with just 7 Kb, and no official way of adding more RAM was provided, neither was an expanded model (**HP42SX**) ever released.

What were we left with ? Well, ***a fantastic calculator*** which ideally expanded classic RPN capabilities and simple-yet-effective programming model, adding such groundbreaking features as perfectly integrated *complex numbers* and *matrices*, where most any function would work with both (taking the sine of a matrix would replace each element with its sine, for instance), powerful matrix editing capabilities, *named variables* which could store any data object along with classic numbered registers, rudimentary but useful *graphic capabilities*, expanded alpha functionality, *2-line* dot matrix display, built-in and user-defined *menus*, fast program execution, low battery consumption, slim package, the works !.

Even given its painful I/O shortcomings, it still was (and ***is***) a dream calculator, easy to carry with you at all times, and with awesome computing power. It could be profitably used at all levels, from student to hardened professional, and you could write professional-looking programs for it with utmost ease due to its ergonomic, user-friendly design.

To demonstrate the fact and to provide a non-trivial example of most of the advanced **HP42S**' capabilities and the (sometimes unexpected !) ways they can be put to use when dealing with a particular problem, I'm providing here a sizable program I wrote last summers explicitly for this article.

Introducing EQUEENS, "8 Queens puzzle in style"

EQUEENS was written last August while out vacationing, thanks to my best friend (and former PPC member #4995) **Fernando del Rey** letting me have his beloved **HP42S** for the duration. Unlikely as it seems, I'd never used an **HP42S** before (despite having an unused, mint one in my collection) and found the experience both *enlightening* and *enjoyable*. The **HP42S'** capabilities really caught my eye, and though regrettably I paid it next to no attention when it was released, I now consider it one of the very best models ever produced by HP and, as a "pure programmable calculator" (alpha and meager graphics capabilities aside), probably the very best there is, bar none.

EQUEENS provides a complete, "*professional-looking*" approach to the problem of solving the classical **8-Queens Puzzle**. At the time I wrote it, I was unaware that this topic had featured recently in **Datafile**, namely the extremely interesting articles "Eight Queens ... in Half" (V22 N6 Page 15) by **Jordi Hidalgo**, and "Eight Queens Revisited" (V23 N4 Page 26) by **Bill Butler**.

As their articles were centered around the **48/49** models and I hadn't read them at the time, my program for the **HP42S** uses a different approach to find and store all 92 solutions and filter them out to extract just the 12 primary solutions. Further, the solutions are then displayed selectively both in alphanumeric and graphic forms.

The search itself is actually fairly easy to perform and can be done in just a few lines of code. It's based on this simple, 9-liner BASIC program I wrote 24+ years ago for the **SHARP PC-1211**, the infamous contemporary of the **HP-41C**:

```
1 A(Y)=A(Y)+1: IF A(Y)>X LET Y=Y-1: GOTO 1
2 GOTO 5
3 "A" CLEAR: INPUT "N=";X: Y=1: WAIT
4 A(Y)=1
5 IF Y=1 GOTO 8
6 FOR Z=1 TO Y-1: IF (A(Z)=A(Y))+(Y-Z=ABS(A(Z)-A(Y))) LET Z=Y: NEXT Z: GOTO 1
7 NEXT Z
8 Y=Y+1: IF Y<=X GOTO 4
9 USING: FOR W=1 TO X: PRINT "Queen at ";W;A(W): NEXT W: Y=Y-2: GOTO 1
```

these 9 lines of BASIC (which will work on most any SHARP pocket computer, from the original **PC-1211** onwards, simply **RUN "A"** or **DEF "A"**) will actually find *all* solutions to the ***N-Queens*** problem (not just 8-Queens), and then end with an (easily avoided) error message when there are no more solutions left.

EQUEENS uses an improved version of this algorithm (optimized and particularized for the 8-Queens case), but the bulk of the program is dedicated to implement the "professional" features, such as the menus, filtering, graphic display, and last but not least, user-friendliness and thorough error trapping.

The following pages feature the **Program Listing** with **Notes**, a comprehensive description of its inner workings, highlighting the *advanced techniques* and *tricks* used (many of which are general enough to be used in your own programs), as well as a sample run with step-by-step graphical instructions, and an **Appendix** listing all solutions to the puzzle as found by the program.

Program listing

Here is the program listing[1]. See the **Notes**, below, for details on how to enter some of the lines.

'EQUEENS' *(1,325 bytes)*

```
  1 LBL "EQUEENS"
  2 "8 Queens v1.0"
  3 ├" Ready"
  4 AVIEW
  5 ALL
  6 RECT
  7 XEQ "*INIT"
  8 CLMENU
  9 "DISP■"
 10 KEY 1 XEQ "*DSP"
 11 "SRCH"
 12 KEY 2 XEQ "*SRCH"
 13 "FILT"
 14 KEY 3 SEQ "*FILT"
 15 "→ALL"
 16 KEY 4 XEQ "*ALL"
 17 "→PRI"
 18 KEY 5 XEQ "*PRI"
 19 "DONE"
 20 KEY 6 GTO "*DONE"
 21 LBL 00
 22 MENU
 23 STOP
 24 GTO 00
 25 LBL "*INIT"
 26 99999999
 27 STO "J"
 28  2
 29  1
 30 NEWMAT
 31 ENTER
 32 COMPLEX
 33 STO "BOAR"
 34 STO "COOR"
 35  8
 36  1
 37 DIM "COOR"
 38 INDEX "BOAR"
 39 WRAP
 40  3
 41  1
 42 COMPLEX
 43  →
 44  3
 45  9
 46 COMPLEX
 47  →
 48 SF 01
 49 CLST
 50 RTN
 51 LBL "*DSP"
 52 "Disp while"
 53 ├" search"
 54 AVIEW
 55 FC?C 01
 56 SF 01
 57 FS? 01
 58 ├" On"
 59 FC? 01
 60 ├" Off"
 61 AVIEW
 62 "DISP"
 63 FS? 01
 64 ├"■"
 65 KEY 1 XEQ "*DSP"
 66 RTN
 67 LBL "*SRCH"
 68  0
 69 STO "S"
 70 XEQ 99
 71 SIZE 10
 72  1
 73 STO 09
 74  1
 75 DIM "SOLS"
 76 GROW
 77 INDEX "SOLS"
 78 LBL 24
 79  1
 80 STO IND 09
 81 LBL 25
 82 RCL 09
 83  1
 84 X=Y?
 85 GTO 28
 86  -
 87 1E3
 88  ÷
 89  1
 90  +
 91 STO 00
 92 LBL 20
 93 RCL IND 00
 94 RCL IND 09
 95 X=Y?
 96 GTO 21
 97  -
 98 ABS
 99 RCL 09
100 RCL 00
101 BASE-
102 X#Y?
103 GTO 27
104 LBL 21
105  1
106 RCL 09
107 X=0?
108 GTO 32
109 X#Y?
110 GTO 69
111 RCL IND 09
112  4
113 X=Y?
114 GTO 32
115 LBL 69
116 ISG IND 09
117 ABS
118 RCL IND 09
119  8
120 X>=Y?
121 GTO 25
122 DSE 09
123 ABS
124 GTO 21
125 LBL 27
126 ISG 00
127 GTO 20
128 LBL 28
129 ISG 09
130 ABS
131  8
132 RCL 09
133 X<=Y?
134 GTO 24
135 ISG "S"
136 ABS
137 TONE 5
138 1.008
139  0
140 LBL 31
141 10
142  x
143 RCL+ IND ST Y
144 ISG ST Y
145 GTO 31
146 ENTER
147  →
148 X<>Y
149 RCL "S"
150 FS? 01
```

[1] You can download this listing in standard TEXT format from the HPCC web site at http://www.hpcc.org. The resulting file can then be used with an emulator or a real 42S.

```
XEQ 55
XEQ 99
 2
STO- 09
GTO 21
LBL 32
"Ok "
RCL "S"
RCL+ "S"
AIP
├" solutions"
├" found"
AVIEW
BEEP
RCL "S"
 1
DIM "SOLS"
WRAP
RTN
LBL 99
"Searching: "
ARCL "S"
├" found.."
AVIEW
RTN
LBL "*DONE"
"Bye!"
AVIEW
WRAP
EXITALL
SIZE 10
CLMENU
CLV "SOLS"
CLV "SOLP"
CLV "VARS"
CLV "BOAR"
CLV "COOR"
CLV "P"
CLV "S"
RTN
LBL "*ALL"
"All solutions"
AVIEW
SF 00
PSE
SF 25
INDEX "SOLS"
FS?C 25
GTO 88
XEQ "*SRCH"
GTO "*ALL"
LBL 88
RCL "S"
GTO 90
LBL "*PRI"
"Primary solutio"
├"ns"
AVIEW
CF 00
PSE
SF 25
INDEX "SOLP"
FS?C 25
GTO 89
XEQ "*FILT"
GTO "*PRI"
LBL 89
RCL "P"
LBL 90
STO 00
 0
STO 01
WRAP
LBL 22
RCLEL
ISG 01
ABS
RCL 01
XEQ 55
FC? 00
GTO 61
RCL "J"
RCLEL
 -
ISG 01
ABS
RCL 01
XEQ 55
LBL 61
 J+
FC?77
GTO 22
"Ok "
ARCL 01
├" solutions"
├" shown"
AVIEW
RTN
LBL "*FILT"
"Filtering.."
AVIEW
WRAP
SF 25
RCL "S"
FS?C 25
GTO 83
XEQ "*SRCH"
GTO "*FILT"
LBL 83
STO "P"
 2
STO "I"
RCL "SOLS"
STO "REGS"
INDEX "REGS"
INSR
 1
 6
DIM "VARS"
LBL 70
RCL "I"
IP
RCL "P"
X<Y?
GTO 75
1E3
 ÷
 +
STO "I"
LBL 71
RCL IND "I"
"Checking Sol: "
AIP
AVIEW
XEQ 77
INDEX "VARS"
LBL 72
RCL "I"
 1
BASE-
STO ST L
LBL 73
RCLEL
RCL IND ST L
X=Y?
GTO 74
DSE ST L
GTO 73
 J+
FC? 77
GTO 72
ISG "I"
GTO 71
LBL 75
"Ok "
RCL "P"
AIP
├" primary"
├" solut."
AVIEW
INDEX "REGS"
DELR
RCL "REGS"
STO "SOLP"
SIZE 10
RTN
LBL 74
INDEX "REGS"
 1
STO- "P"
RCL+ "I"
LASTX
STOIJ
DELR
GTO 70
LBL 77
INDEX "VARS"
STO 00
XEQ 45
RCL 00
XEQ 40
RCL "J"
RCL ST Z
STOEL
 J+
STO 00
 -
 →
RCL 00
LBL 45
XEQ 43
RCL "J"
```

```
343 RCL ST T
344 STOEL
345  J+
346  -
347  →
348 RTN
349 LBL 40
350 CLA
351 AIP
352  1
353  0
354 LBL 41
355 ATOX
356 X=0?
357 RTN
358 49
359  -
360 10^x
361 RCLx ST Z
362  +
363 ISG ST Y
364 ABS
365 GTO 41
366 LBL 43
367 CLA
368 AIP
369  0
370 LBL 44
371  -1
372 AROT
373 ATOX
374 X=0?
375 RTN
376 48
377  -
378 10
379 RCLx ST T
380  +
381 GTO 44
382 LBL 55
383 "     Solution #"
384 AIP
385 R↓
386 ├"↵     "
387 AIP
388 AVIEW
389 "U÷÷U÷÷U÷÷U÷÷"
390 ├"U÷÷U÷÷U÷÷U÷÷"
391 RCL "BOAR"
392 AGRAPH
393 R↓
394 CLA
395 AIP
396 RCLIJ
397 COMPLEX
398 INDEX "COOR"
399 1.01502
400 LBL 50
401 ATOX
402  48
403  -
404  3
405  x
406 RCL ST Y
407 COMPLEX
408  →
409 R↓
410 ISG ST X
411 GTO 50
412 "∫∫"
413 RCL "COOR"
414 AGRAPH
415 STOP
416 R↑
417 COMPLEX
418 FS? 00
419 INDEX "SOLS"
420 FC? 00
421 INDEX "SOLP"
422 STOIJ
423 END
```

Notes

- Lines 9 and 64 include the *small block* character found in the **MISC** submenu of the **ALPHA** menu, lines 102 and 109 are the “X not equal Y” logical test, and line 383 begins with *exactly five* spaces.
- Line 386 begins with a “Line Feed” character “L/F” shown in the listing with the symbol **“↵“**. You must enter the correct Line Feed character instead, which can be found at the end of the second row of the **PUNC** submenu of the **ALPHA** menu. Once entered, place *exactly five* spaces after it.
- Line 412 includes two “Integral” characters, which can be found in the **MATH** submenu of the **ALPHA** menu.

Programming details & techniques

This program is intended as a comprehensive demo of many HP42S’ advanced capabilities and the professional style of programming it encourages, featuring user-friendly menus and prompts, labeled and graphics output, error trapping and logic to detect and perform necessary actions when omitted by the user.

This being so, program’s length is a secondary concern and thus as many lines as necessary are used to achieve the goal, while still optimizing each and every routine. A detailed explanation of the application’s inner workings follows, discussing relevant techniques as appropriate:

- All user-callable internal routines featured in the menu have suitably meaningful names beginning with a **"*"** so you'll be able to recognize in the Catalog that they're internal to this application. Other internal routines use numeric labels instead. This makes the listing more readable without wasting memory, slowing down the process, or cluttering up the catalog.

- **"EQUEENS"** (*Eight Queens*) is the application's main entry point. First of all, lines 1-24 show a welcome message to the user identifying the application's version, then **"*INIT"** is called to initialize, the menu is built, and a loop is entered where the menu is shown after each menu option is completed.

- The **"*INIT"** (*Initialization*) routine (lines 25-50) initializes all *global* variables used by the application, and a pair of complex matrices, namely **BOAR** (Board), which is used to draw the 8x8 board *very* quickly, and **COOR** (Coordinates), used to draw each solution's queens over the board. It also stores a constant used to rapidly generate a symmetric solution and sets a flag used to specify whether or not the application will display each solution as it is found.

- The **"*DSP"** (*Display as found*) routine (51-66) is called from the menu option DISP■ to toggle **On/Off** the immediate display of each solution as it's found during the search. A message is shown specifying the current setting, and the menu option is recreated with a *small block* appended if the status is On.

- The **"*SRCH"** (*Search*) routine (67-169) is called from the menu option SRCH to search for all solutions to the puzzle, which are stored (and optionally displayed) upon finding. The number of solutions is kept in variable **S**, which is initially cleared (68-69), 10 numbered registers are allocated (71) and a matrix **SOLS** is created to hold the solutions with the smallest possible initial size, but specifying it'll *automatically grow* as each element is filled in (72-77).

  The search itself (78-155) is the heart of the application, a simple affair of exhaustively trying in turn all possible legal places for each queen, *backtracking* when a newly placed one is found to be under attack from some other. Registers **00** and **09** are used as *indexes* and each of the registers **01-08**, their numeric addresses acting as *rows*, store the *column* position of the queen in that row. When the last queen is successfully placed, the number of solutions is incremented and a tight loop is entered (135-145) to coalesce said 8 registers' contents into a single 8-digit number (17582463, say) , the position of each digit being the row number, and the digit itself being the column number for each queen. The solution is then stored in matrix **SOLS** (77, 147) and if the user opted to display each solution when found, a call is made to immediately display it (148-152). The search for further solutions is then resumed (153-155).

  When the search is over, execution branches to label **32**, where the total number of solutions is assembled and displayed, minor cleaning is performed and execution returns to the menu (156-169). Several techniques worth mentioning:

  - One of the 42S' Enhanced RPN greatest assets is the possibility of using both *registers* addressed by number as in previous RPN models, as well as *named variables*. Both types have their strengths, but best is to combine them, taking advantage of their intrinsic characteristics when needed. Here we're using the numeric registers as a *fast*, memory-saving scratch area, using them for multiple indexing and looping, dinamically allocated as needed, while the named variables (**SOLS**, **BOAR**, **COOR**, etc.) are used for more permanent, *global* data.

Even better, the 42S allows the user to handle *all* numeric registers *en masse* as a *single* named variable, the **REGS** matrix. This is put to good use when filtering the solutions to extract just the primary ones (259-325). All solutions are placed in the **REGS** variable so that we can handle them easily, and have the **REGS** matrix *shrinking in size* (312, 324) as derived, non-primary solutions are identified and discarded.

- Also, we need not check all 8 columns for the position of the 1st queen, as symmetry considerations mean that a queen placed at columns 5-8 gives a mirror-symmetric solution to any already found for columns 1-4. So, our search needs only check columns 1-4, thus *halving* both the search time and the storage requirements for the solutions. Each solution found actually stands for two, and this is reflected (158-159) when reporting the number of solutions found and further on when filtering the solutions to find out the primary, non-equivalent solutions

- Finally, note how the rarely seen **BASE-** instruction is used (101) to save time and program steps while computing INT(R09)-INT(R00). Both index registers hold fractional parts at the time but we're only interested in the result of subtracting their integer parts, and **BASE-** *saves 3 lines and runs faster* over the usual construct **RCL 09**, **INT**, **RCL 00**, **INT**, **-** . The same technique is used at line 290.

• The **"*DONE"** (*Done with program*) routine (176-190) is called from the user menu option DONE, and simply ends the application. It includes clearing the menu and the named variables so that they don't take up space and clutter the variables catalog (182-189), resizing the numeric registers (181), and terminating execution with a farewell message (177-178). This kind of clean-up is essential for any program and greatly contributes to the "professional" look.

Notice how the message is shown *immediately* upon entering the routine, so that the user feels a *quick* response time when selecting the option. While reading it, the routine performs its task and finishes, with no *perceived* running time !

• The **"*ALL"** (*Display all solutions*) routine (191-248, partly shared with the **"*PRI"** routine) displays ***all*** solutions previously found and stored by **"*SRCH"** (*Search*), and is called from the user menu option →ALL. First of all it sets a flag (194) later used in the shared routine (219-248) to distinguish between **"*ALL"** and **"*PRI",** then tries to detect and cater for the fact that perhaps the user *forgot* to perform the search before pressing →ALL, efectively attempting to display solutions that haven't been found and stored yet !.

This is accomplished (196-204) by raising Flag 25 and trying to index the **SOLS** matrix, where all solutions are stored. If they haven't been stored yet, the **SOLS** matrix *does not exist*, so indexing it *fails* and Flag 25 gets cleared. This is detected and a call to **"*SRCH"** (*Search*) is *automatically* performed on behalf of the user, then execution goes to **"*ALL"** to try again.

If the indexing *is* successful, execution goes to the shared routine (see **"*PRI"**), with the number of solutions to display in the X register. This detection mechanism effectively allows the user to execute options out of order: if any necessary data are missing, the application will automatically detect the fact and get them first, without bothering the user with error messages/prompts and without enforcing a fixed order of operation. This enhances user-friendliness.

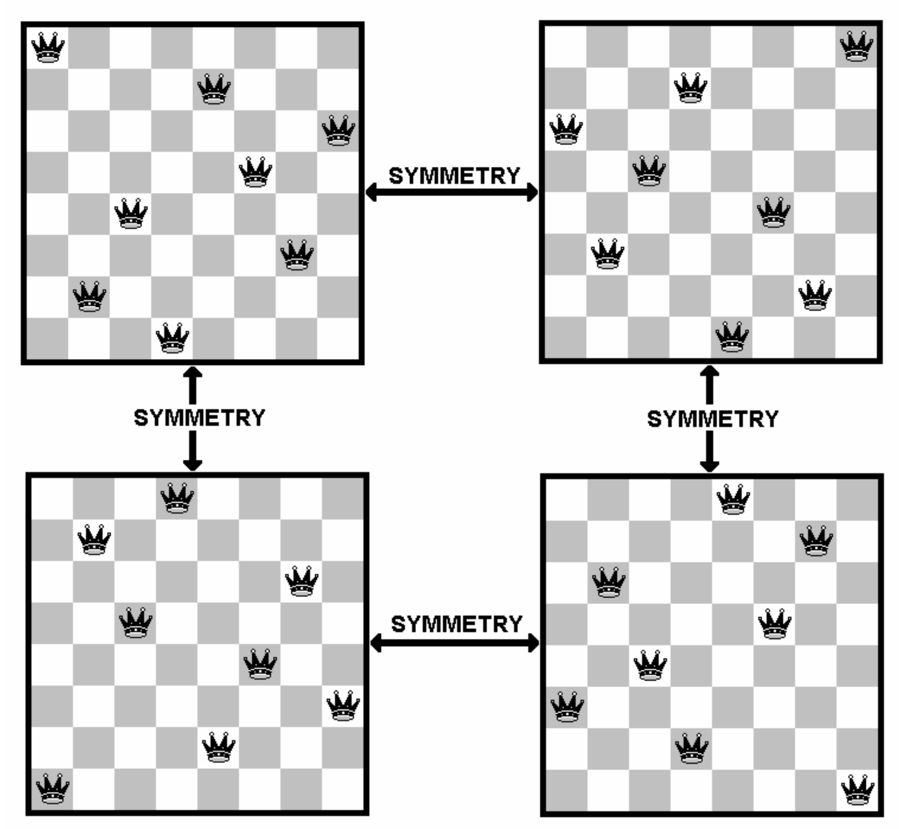
- The **"*PRI"** (*Display primary solutions*) routine (205-248, partly shared with the **"*ALL"** routine) displays only the ***primary*** solutions previously found and stored by **"*FILT"** (*Filter solutions*), and is called from the user menu option →PRI. It clears a flag (209) later used in the shared routine, then checks if the user actually *forgot* filtering the solutions (FILT) before pressing →PRI, thus trying to display primary solutions not filtered out and stored yet.

  As before, this is accomplished (211-216) by raising Flag 25 and trying to index the **SOLP** matrix, where primary solutions are stored. If the primary solutions haven't been filtered yet, the **SOLP** matrix doesn't exist and Flag 25 is cleared, which triggers a call to **"*FILT"** (*Filter*) to automatically do the proper thing without bothering the user, then execution goes back to **"*PRI"** for a retry.

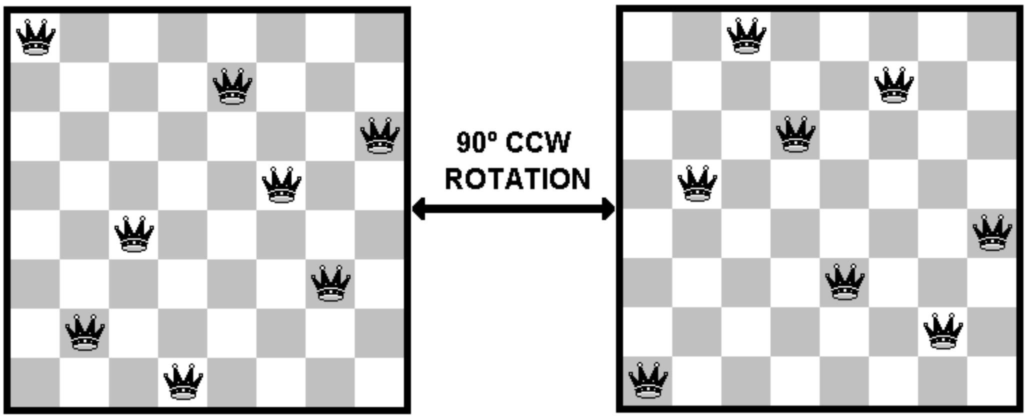
  Once successful, the shared routine is executed with the number of primary solutions to display in **X**. This routine (219-248) recalls in turn all elements of the indexed matrix (**SOLS** or **SOLP**), which are the stored solutions (respectively, ***all*** or ***primary***) and passes them and their index number, to subroutine **55** (225-229), which creates and shows the actual *graphics* display.

  For All Solutions, after displaying each of them we then generate its mirror-symmetrical counterpart and call subroutine **55** again (232-238), then the next element is selected and the loop termination condition (Flag 77) is checked (239-242). Finally the number of solutions displayed is shown (243-248).

- The **"*FILT"** (*Filter solutions*) routine (249-325) is called from the user menu option FILT, to filter all solutions found and select just the ***primary*** ones, which get stored in matrix **SOLP** for later display. It does this by recognizing and discarding all equivalent solutions, either generated by a *symmetry* or a *rotation* of a primary solution. For instance, given any solution, you can derive 3 additional ones by left-right and top-bottom mirror symmetries, like this:



Apart from *symmetry*, you can also derive further solutions by *rotating* a given solution, like this example where a 90º counter-clockwise rotation is applied:



By using reflections and rotations you can derive *up to 7 additional equivalent solutions* starting from a given one, except if it's symmetrical to boot, where you'll get less than 7 distinct derived solutions. This is the case here, because our puzzle has **12 primary solutions**, which normally would result in 12*8 = 96 solutions in all, except for the fact that one of the primary solutions is symmetrical, thus limiting the maximum number of solutions to just **92**.

Once again, **"*FILT"** raises Flag 25 and tries to determine if there are solutions to filter. If not, it calls **"*SRCH"**, then tries again (253-258). Else, all solutions stored in **SOLS** are copied to **REGS**, which brings us two benefits:

- as **REGS** is the system variable which holds all *numbered* registers, we can access elements (solutions) individually using standard *indirect addressing*, while we simultaneously use *matrix element addressing* with another matrix variable. Thus, we can compare (293-295) a solution extracted from **VARS** (**RCLEL)** to another solution extracted from **REGS** (**RCL IND ST L)**. This would be very difficult to achieve using just matrix element addressing, as we would need to constantly *change pointers back and forth* with **INDEX - STOIJ - RCLIJ**, for instance, while here the matrix pointers *coexist* and synergically *cooperate* with as many indirect addressing pointers as necessary. This is seen at 280-303, where **RCLEL** and **J+** are *intermixed* with **RCL IND "I"** , **RCL IND ST L** and **ISG "I"**, **DSE ST L** to detect and remove duplicated variants.
- as **REGS** is a *matrix variable*, we can *remove* elements from it, the remaining ones automatically shifting positions to fill up the void. This is seen in 317-325, where a duplicated solution is removed from **REGS**.

"*FILT" traverses the solutions' list (270-303) and fills up a matrix called **VARS** with *all* symmetric and rotated *variants* for each solution in turn (280-286). These are then compared against other solutions on the list (292-296), and if there's a match, the solution being tested is *removed* (317-325). Once done with the removals, all remaining (primary) solutions are copied from **REGS** to **SOLP** and the user is told how many primary solutions were left (304-316).

- **Subroutine 77** (326-381) is an important utility routine called from ***FILT"** (*Filter*). Given a solution in **X**, it fills up matrix **VARS** with 6 variants generated from it by symmetries and rotations (the left-right symmetric variant isn't generated or tested, as it's an *implicit* solution displayed by **"*ALL"** but never stored in **SOLS**, actually)**.** It calls lower-level internal routines (labels 40,41,43,44,45) to generate and store each variant into **VARS**. Use is made of **AIP** (351, 368) and **ATOX** (355, 373) to decompose the solution (previously placed in the Alpha register) and reform it into a variant in X. Using the Alpha register and the stack *simultaneously*, we keep the decomposing and recreating processes going on at the same time without *interfering*.
- Finally, **subroutine 55** (382-423) is the last and very important utility routine called from the common part of "*ALL" and "*PRI" to display each solution both in alphanumerical and *full graphical form*, like this:

The board is represented by an 8x8 dot grid and each Queen is represented by a small 2x2 block. Additionally, both the solution´s number (passed in **X**) and column representation (passed in **Y**) are displayed as well.

This routine uses some advanced techniques to overcome several programming challenges, like *optimizing for speed*. Normally, drawing the 8x8 dot grid and placing the queens in their proper positions would require *nested loops* which take some noticeable time to run and worse, the user sees the display while slowly forming, instead of appearing fully formed at once. This is solved by using an *advanced capability* of **AGRAPH** which, shockingly, is **totally absent** from the User's Manual ! So much for the famed thoroughness of HP manuals of the past ... **HP42S** users are supposed to *buy yet another manual* if they want the smallest glimpse of this very important feature that should have been *documented* in the User's Manual to begin with.

The advanced feature is: if there's a *complex matrix* in **X** when executing **AGRAPH,** the contents of the Alpha register, interpreted as a *bit pattern*, will be placed in the display beginning at the *locations* specified by the matrix elements, each representing the coordinates of a pixel. See this feature in action in lines 389-392, where the dot grid's bit representation is placed in Alpha and the complex matrix **BOAR** (previously created and filled up by "*INIT") is placed in **X**, then **AGRAPH** draws the 8x8 dot grid *almost instantaneously*.

The queens are similarly drawn *all at once* (412-414), the two "integral" characters being the bit representation of a 2x2 solid block, and the **COOR** complex matrix holding the precise locations for all 8 queens in this particular solution. However, unlike the static matrix **BOAR** which needs be initialized only once, the **COOR** matrix has to be filled up with the complex values corresponding to the locations using a loop to dissect the solution one column at a time (398-411) and create the appropriate complex element to be stored in **COOR** (407-408). Thus, drawing the queens isn't as fast as drawing the board grid, but all the queens do appear *at once* upon execution of **AGRAPH** (414).

Anoher interesting technique used in this routine caters for the fact that we need to index matrix **COOR** for the loop that stores the queens' locations, but actually **subroutine 55** is called *inside another loop* which traverses the solutions (all/primary) and thus has previously indexed either **SOLS** or **SOLP** . But the 42S doesn't allow having *more than one* indexed matrix at a time !

The answer is to keep track of the index position in the 1^{st} matrix (**SOLS** or **SOLP**) so that it can be restored back after we're finished with the 2^{nd} matrix (**COOR**). This is cleverly done at 396-397, where the index position is recalled and converted to a *complex number* so that row/column pointers use up a *single* stack entry, restored back at 416-422. The complex value is *split* into its two components and the matrix (**SOLS** or **SOLP**) is re-indexed before restoring the index. Combining *both* pointers into a *single* complex value allows them to *float* on the stack during the proceedings, thus no need to save them elsewhere. This technique can be profitably used in many different situations.

Usage

To begin *executing* the application, simply:

`XEQ "EQUEENS"`

The program will initialize and the menu will appear:

As you can see by the small block in the DISP■ option, displaying each solution as they're found is **On** by default (*interactive mode*). We'd rather search for all solutions without pausing to display any, so that the program can run unattended and we can have a cup of tea while the search goes on, so we'll *deactivate* it:

Click DISP■

The menu refreshes and a confirming message does appear:

Now we'll start the *search* for good, click SRCH

This will search for and store all solutions to the puzzle. It's a *lengthy* process so you'd better leave the program alone and go attend other businesses. The display will refresh as each solution is found, like this:

...

When all solutions have been found, the menu will be displayed again, with an informative message telling us just how many solutions were found, **92 in all**:

Now we'll *filter* the solutions to extract the primary ones, which aren't reflections or rotations of one another. To start the filtering process, simply click FILT

The filtering process will begin. It'll take much less time than the previous search, and while it goes on, the display will refresh to inform you of the particular solution being tested at the moment, like this:

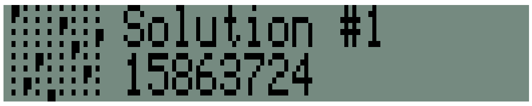
...


...

When the filtering is over, the menu will appear again and an informative message will tell you how many primary solutions were found and stored, **12 in all**:

Now that the search and filtering are over, we want to *display* the solutions found:
Click →ALL

A message confirming the operation will appear, then each solution in turn :

...


Press 

to display the *next* solution, and so on until the last one:

Press 

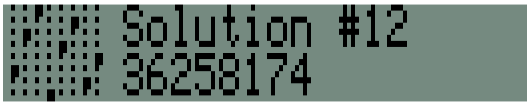
one last time to *return* to the menu:

If you want to *display* just the *primary* solutions, simply click →PRI

...


Press 

to display the *next* primary solution, and so on until:



Press 

one last time to *return* to the menu:

That's all. Now, to *end* the application and *clean up*, simply click 

Clean-up is performed, a farewell message is displayed and the application ends:

Appendix A: All solutions

All 92 solutions; the 12 primary solutions are in **bold face** and underlined:

15863724	84136275	**16837425**	83162574	17468253	82531746
17582463	82417536	**24683175**	75316824	**25713864**	74286135
25741863	74258136	**26174835**	73825164	**26831475**	73168524
27368514	72631485	**27581463**	72418536	28613574	71386425
31758246	68241753	**35281746**	64718253	35286471	64713528
35714286	64285713	**35841726**	64158273	**36258174**	63741825
36271485	63728514	36275184	63724815	36418572	63581427
36428571	63571428	36814752	63185247	36815724	63184275
36824175	63175824	37285146	62714853	37286415	62713584
38471625	61528374	41582736	58417263	41586372	58413627
42586137	57413862	42736815	57263184	42736851	57263148
42751863	57248136	42857136	57142863	42861357	57138642
46152837	53847162	46827135	53172864	46831752	53168247
47185263	52814736	47382516	52617483	47526138	52473861
47531682	52468317	48136275	51863724	48157263	51842736
48531726	51468273				