HP-71B Modest Mater

Valentín Albillo (#1075, PPC #4747)

The HP-71B has always been a marvelous, unique machine with an assortment of state-of-the-art features which was nonetheless unjustly marred by a "less than enthusiastic" marketing support as well as unbearably high prices when released. True, other brand names such as **SHARP** or **CASIO** did flood the market with a variety of BASIC-language handhelds, very capable and all. But nowhere would you find so powerful a BASIC dialect, supporting such advanced capabilities as recursive subprograms with independent local environments and parameter passing by value or reference. Or multi-line, recursive user-defined functions. Or full file system handling with various file types, including serial and random access. Or extensibility by means of new BASIC keywords and binary subprograms directly created in assembler, having the exact same capabilities as built-in keywords and BASIC subprograms (compare that to the typical PEEKs, POKEs and USR calls), with the whole official IDS documentation publicly available. Or being able to completely redefine the whole Operating System by means of special, hardwired ROMs (such as the FORTH/Assembler ROM). Or expandability by means of RAM/ROM modules (even custom ones) up to 512 Kb, in any combination. Or having at your disposal such powerful add-ons as the Math ROM, the HP-IL ROM, the FORTH/Assembler ROM and the Curve Fitting ROM, to name a few. And last but not least, its awesome capabilites as system controller of an extensive range of peripherals including cutting-edge automated measurement devices of all kinds, or even other computers on the loop. In other words, the HP-71B was light-years ahead of any other similar offering at its prime time, no contest really.

But ... nothing is perfect and a number of questionable decisions resulted in some less than optimal characteristics making their way to the final product. The most noticeable is the rather pathetically small display, limited to a single 22-character line. This is pretty inconvenient for any serious program or data entry, and the fact that a line can be 96 characters internally and will scroll when necessary does little to lessen the problem. Furthermore, **SHARP** had a number of 4-line x 24 char models at the time and it made a world of a difference. Anyway you could always fit an HP-IL Video Adapter (and even an external keyboard) and though losing some convenience and portability, it was still a practical arrangement for developing large programs and dealing with sizable amounts of data.

However, the other major shortcoming of the HP-71B was not that easy to overcome (till now, see **Appendix A**!) and it has to do with speed. Or lack thereof. The 71B used a newly designed CPU, codenamed 'Saturn', that would run at a mere 0.65 Mhz, dealing with data in 4-bit blocks called 'nybbles', while standard CPUs of the time would run at 2-4 Mhz, and deal with 8-bit bytes. True, the Saturn CPU was optimized for BCD math and this alone gave it an edge in speed and accuracy when dealing with complex, intensive math computations. But for every other use and purpose, it was hopelessly slow. This was aggravated somewhat by

the built-in BASIC interpreter, which for all its power and advanced features was nevertheless excruciatingly *slow* when doing such extremely common operations as branching and looping. So, you had a machine that could hold his own ground against other handhelds or micros when doing heavy math, yet it would be left far behind when executing simple loops (specially if nested) or branch-heavy routines, as much as 10 times slower than arguably lesser machines. This could be alleviated somewhat by using FORTH, a threaded language inherently much faster, or even Assembler for ultimate speed. Or better still, a mix of all three. But as for the slow CPU, there was little else you could do.

Does this mean that you can't use HP-71B's BASIC for non math-related tasks ? Not so, its powerful capabilities go a long way towards making possible an efficient implementation of algorithms that would take a lot of convoluted code and long running times in other systems. And past that, a little patience will do.

To prove it, here's a BASIC program dealing with a near worst-case problem, i.e.: *not math-related* at all, and full of *nested loops*, continuous branching, and convoluted logical decisions, exactly the kind of constructions that give 71B's BASIC the bends. Yet, with the help of its advanced features, most notably *recursive subprograms*, local environments, and *user-defined functions* (plus a little patience) we'll succeed. Well, sort of ...

Introducing MATER71, a Modest Mater program

MATER71 is a simple program I've created to solve classic **Mate-In-N** chess problems; you enter a chess position corresponding to some Mate-in-n problem, and MATER71 merrily proceeds to find the solution in the specified number of moves or less. Despite the complicated subject matter and the program's relative simplicity, it boasts an impressive list of features, among them:

- Recursively searches for a mate in N moves, generating all legal moves for one ply for both sides, then *calling itself* till a mate is found or the maximum depth is reached. N is limited only by available memory and time.
- It can search for a mate in a *specific* number of moves, or in *any number* of moves (beginning with Mate-in-1) up to a predefined maximum.
- It implements some simple yet effective *heuristics* to help speed the search.
- Will consider pawn *promotions* (to Queen) and *underpromotions* (to Rook, Bishop, or Knight) for both sides. It understands stalemates as well.
- Can *store* the given position in a file, for later retrieval, and thus it can accept a position entered either from the keyboard or *read* from a named *data file*.
- Optionally, it can *display* the entered (or read) position before proceeding to compute the mate. This can be very useful to check if the position has been entered correctly and avoid malfunctions and/or wasted time if it wasn't.

- While searching, it displays each *move under consideration*, identified with depth and move number; upon finding a mate, it will display the mating move, together with its depth and timing. Likewise if no mate is found.
- Comprehensive *error checking* of all user inputs (position, filenames, etc)

As this is a sample, *proof-of-concept* program, not intended for real production use, there are some limitations as well, mainly to avoid making it unordinately lengthy and slow, namely:

- It will find mates only for White, not Black (this is hardly a problem, simply reverse colours when entering the position).
- All legal moves are generated, except for *castling* and *en-passant* captures. If the solution depends on them, you might get incorrect results, i.e.: a mate might not be found when there's one, or it might be found when there's none.
- There's no error checking of the entered *position* to see if it is legal (i.e.: no Black King). For illegal positions the program's behaviour is *undefined*.
- Times are *still* lengthy: 12-120" for M1, 2'-20' for M2, 30'-300' for M3, etc.

Here is the program listing[1]. It makes use of no external LEX keywords, *except* for matrix assignment operations, taken from the Math ROM (see **Notes**, below) :

'MATER71' *(4716 Bytes)*

```
100 ! *** MATER Version 1.0 - (c) Valentin Albillo, 2004 ***
105 !
110 ! Initialization
115 !
120 DESTROY ALL @ OPTION BASE 0 @ RESTORE @ DELAY 2,0 @ STD @ INTEGER B(119)
125 DIM F(6,8),I,J,P,Q,C,G,H,T,M,N,K,D,A$,P$[6],F$[96] @ MAT B=(7)
130 FOR I=21 TO 91 STEP 10 @ FOR J=0 TO 7 @ B(I+J)=0 @ NEXT J @ NEXT I
135 DISP "MATER 1.0 - (c) V. Albillo" @ READ F,P$ @ Q=1 @ ON ERROR GOTO 155
140 !
145 ! Read problem from file
150 !
155 INPUT "Read from ";F$ @ IF F$="" THEN 180 ELSE ASSIGN #1 TO F$
160 READ #1;B,M,N,Q @ ASSIGN #1 TO * @ OFF ERROR @ GOSUB 320 @ GOTO 200
165 !
170 ! Enter problem from keyboard
175 !
180 DISP "Ent: +-/"&P$&"/A-H/1-8"
185 ON ERROR GOTO 185 @ INPUT "Piece=";A$@ IF A$="" THEN GOSUB 320 @ GOTO 200
190 A$=UPRC$(A$) @ IF NOT FNL(A$) THEN BEEP @ DISP "Illegal entry" @ GOTO 180
195 GOSUB 290 @ GOTO 185
200 ON ERROR GOTO 200 @ INPUT "Mate in [-=up to]: ",STR$(Q);Q @ Q=INT(Q)
205 IF NOT Q THEN 270
210 !
```

[1] You can download this listing in TEXT format and LIF format from the HPCC web site at http://www.hpcc.org. The resulting file can then be used with an emulator or a real 71B.

```
215 ! Save problem to file
220 !
225 ON ERROR GOTO 225 @ INPUT "Save in ",F$;F$ @ IF F$="" THEN 250
230 ASSIGN #1 TO F$ @ PRINT #1;B,M,N,Q @ ASSIGN #1 TO *
235 !
240 ! Solve problem and display result & statistics
245 !
250 OFF ERROR @ FOR I=MAX(1,Q) TO ABS(Q) @ P=1 @ J=0 @ DELAY 0 @ T=TIME
255 CALL FINDMATE(P,J,G,H,B,I,F,M,N,K,98,98,21,21) @ T=INT(TIME-T)
260 BEEP @ IF J THEN DELAY INF,1 @ DISP "Mate in";I;"w/ ";FNP$ @ GOTO 270
265 DELAY 2,0 @ DISP "No mate in";I;"[";STR$(T);'"]' @ NEXT I
270 DISP "Bye" @ DELAY 0,0 @ END
275 !
280 ! Enter piece info into board
285 !
290 P=POS(P$,A$[2,2]) @ IF A$[1,1]="-" THEN P=-P
295 C=FNN(A$) @ B(C)=P @ IF P=6 THEN M=C ELSE IF P=-6 THEN N=C
300 RETURN
305 !
310 ! Optionally display board position
315 !
320 ON ERROR GOTO 320 @ INPUT "Display board ? ","Y";A$
325 IF UPRC$(A$[1,1])#"Y" THEN RETURN ELSE DELAY 0,0
330 A$="      A  B  C  D  E  F  G  H" @ DISP USING "/,K,/,4X,25'-'";A$
335 FOR I=91 TO 21 STEP -10 @ DISP (I-11)/10;"[ "; @ FOR J=0 TO 7
340 P=B(I+J)*2+13 @ DISP "-k-q-r-b-n-p .+P+N+B+R+Q+K"[P,P+1]&" "; @ NEXT J
345 DISP "]";(I-11)/10 @ NEXT I @ DISP USING "4X,25'-',/,K,/";A$ @ RETURN
350 !
355 ! Ancillary user-defined functions
360 !
365 DEF FNM$(N)=CHR$(MOD(N,10)+96)&STR$(N DIV 10-1)
370 DEF FNN(A$)=10*VAL(A$[4])+NUM(A$[3])-54
375 DEF FNP$=P$[B(G),B(G)*(B(G)#1)]&FNM$(G)&FNT$&' ['&STR$(T)&'"]'
380 DEF FNR$="  =N=B=R=Q"[2*K-1,2*K]
385 DEF FNT$="x-"[1+NOT B(H),1+NOT B(H)]&FNM$(H)&FNR$&"++"[1,(Q=1)*2]
390 !
395 ! Check legality of input piece data
400 !
405 DEF FNL(A$) @ FNL=0 @ IF LEN(A$)#4 THEN END
410 IF NOT POS("+-",A$[1,1]) OR NOT POS(P$,A$[2,2]) THEN END
415 IF NOT POS("ABCDEFGH",A$[3,3]) OR NOT POS("12345678",A$[4,4]) THEN END
420 FNL=1 @ END DEF
425 !
430 ! Data for the moves array and piece representation
435 !
440 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,12,21,19,8,-12,-21
445 DATA -19,-8,4,11,9,-11,-9,0,0,0,0,4,1,10,-1,-10,0,0,0,0,8,1,11
450 DATA 10,9,-1,-11,-10,-9,8,1,11,10,9,-1,-11,-10,-9,"PNBRQK"
455 !
460 ! Subprogram to search for mates recursively up to a given depth
465 !
470 SUB FINDMATE(P,J,G,H,B(),Q,F(,),M,N,T5,C1,D1,A1,B1) @ INTEGER B2(119)
475 DIM C,T,S,I,L,D,U,A,E,K,W,X,Y,Z,R,R0,J8,W1,W2,H1,F1,N5 @ INTEGER B3(119)
480 IF P#1 THEN H1=A1 @ F1=B1 @ GOTO 505 ELSE N5=0
485 FOR C=21 TO C1 @ T=B(C) @ IF T>=1 AND T<=6 THEN H1=C @ GOTO 495
490 NEXT C
495 FOR A=21 TO D1 @ E=B(A) @ IF E<0 THEN F1=A @ GOTO 505
500 NEXT A
505 MAT B2=B @ J=0 @ W1=0 @ T5=1 @ FOR C=C1 TO H1 STEP -1
510 T=B(C) @ IF T<1 OR T>6 THEN 560
515 IF NOT W1 THEN W1=C
520 IF T=1 THEN 540
525 S=T#6 AND T#2 @ FOR I=1 TO F(T,0) @ L=F(T,I) @ D=C
530 D=D+L @ U=B(D) @ IF U>0 THEN 535 ELSE GOSUB 685 @ IF NOT U AND S THEN 530
```

```
535 NEXT I @ GOTO 560
540 D=C+9 @ IF B(D)<0 THEN GOSUB 675
545 D=C+11 @ IF B(D)<0 THEN GOSUB 675
550 D=C+10 @ IF NOT B(D) THEN GOSUB 675 ELSE 560
555 IF C<31 OR C>38 THEN 560 ELSE D=C+20 @ IF NOT B(D) THEN GOSUB 675
560 NEXT C @ END
565 !
570 ! Test if the king is under check
575 !
580 DEF FNJ(C,H,K,E,X,V) @ FNJ=1 @ FOR K=1 TO 4 @ E=F(3,K) @ X=H
585 X=X+E @ V=B(X)*C @ IF V=3 OR V=5 THEN END ELSE IF NOT V THEN 585
590 NEXT K @ FOR K=1 TO 4 @ E=F(4,K) @ X=H
595 X=X+E @ V=B(X)*C @ IF V=4 OR V=5 THEN END ELSE IF NOT V THEN 595
600 NEXT K @ IF B(H-9*C)=C THEN END ELSE IF B(H-11*C)=C THEN END ELSE X=2*C
605 IF B(H-12)=X OR B(H-21)=X OR B(H-19)=X OR B(H-8)=X THEN END
610 IF B(H+12)=X OR B(H+21)=X OR B(H+19)=X OR B(H+8)=X THEN END
615 IF ABS(M-N)>11 THEN 630 ELSE X=6*C
620 IF B(H-1)=X OR B(H-11)=X OR B(H-10)=X OR B(H-9)=X THEN END
625 IF B(H+1)=X OR B(H+11)=X OR B(H+10)=X OR B(H+9)=X THEN END
630 FNJ=0 @ END DEF
635 !
640 ! Local user-defined functions
645 !
650 DEF FNP$=FNS$&FNT$&"=N=B=R=Q"[2*T5-3,2*T5-2]&"+"[1,Q=1]
655 DEF FNS$="PNBRQK"[B2(C),B2(C)*(B2(C)#1)]&FNM$(C)
660 DEF FNT$="x-"[1+NOT B2(D),1+NOT B2(D)]&FNM$(D)
665 DEF FNM$(N)=CHR$(MOD(N,10)+96)&STR$(N DIV 10-1)
670 !
675 IF D<91 OR D>98 THEN 685
680 FOR T5=5 TO 2 STEP -1 @ T=T5 @ GOSUB 685 @ NEXT T5 @ T5=1 @ RETURN
685 B(C)=0 @ B(D)=T @ IF T=6 THEN R=M @ M=D
690 IF P=1 THEN G=C @ H=D
695 IF P#Q THEN 700 ELSE IF NOT FNJ(1,N,0,0,0,0) THEN 800
700 IF FNJ(-1,M,0,0,0,0) THEN 800 ELSE N5=N5+1
705 IF P=1 THEN DISP "Try ";STR$(Q);".";STR$(N5);": ";FNP$
710 W1=MAX(W1,D) @ H1=MIN(H1,D) @ MAT B3=B @ J=1 @ J8=1 @ IF P#Q THEN 730
715 A=N @ E=-6 @ W2=A @ FOR W=1 TO 8 @ Y=A+F(6,W) @ Z=B(Y)
720 IF Z#7 AND Z>=0 THEN GOSUB 820
725 NEXT W
730 W2=0 @ FOR A=D1 TO F1 STEP -1 @ E=B(A) @ IF E>-1 THEN 795
735 IF P=Q AND E=-6 THEN 795
740 IF NOT W2 THEN W2=A
745 IF E=-1 THEN 770
750 K=E#-6 AND E#-2 @ FOR W=1 TO F(-E,0) @ X=F(-E,W) @ Y=A
755 Y=Y+X @ Z=B(Y) @ IF Z=7 OR Z<0 THEN 765
760 GOSUB 820 @ IF NOT Z AND K THEN 755
765 NEXT W @ GOTO 795
770 Y=A-9 @ Z=B(Y) @ IF Z>0 AND Z#7 THEN GOSUB 810
775 Y=A-11 @ Z=B(Y) @ IF Z>0 AND Z#7 THEN GOSUB 810
780 Y=A-10 @ IF NOT B(Y) THEN GOSUB 810 ELSE 795
785 IF A<81 OR A>88 THEN 795
790 Y=A-20 @ IF NOT B(Y) THEN GOSUB 810
795 NEXT A @ IF J8 THEN J=FNJ(1,N,0,0,0,0)
800 MAT B=B2 @ IF T=6 THEN M=R
805 IF NOT J THEN RETURN ELSE END
810 IF Y<21 OR Y>28 THEN 820
815 FOR E5=-5 TO -2 @ E=E5 @ GOSUB 820 @ NEXT E5 @ RETURN
820 B(A)=0 @ B(Y)=E @ IF E=-6 THEN R0=N @ N=Y
825 IF FNJ(1,N,0,0,0,0) THEN 850
830 W2=MAX(W2,Y) @ F1=MIN(F1,Y) @ J=0 @ J8=0 @ IF P=Q THEN 840
835 CALL FINDMATE(P+1,J,0,0,B,Q,F,M,N,O,W1,W2,H1,F1) @ IF J THEN 850
840 POP @ MAT B=B2 @ IF T=6 THEN M=R
845 IF E=-6 THEN N=R0 @ RETURN ELSE RETURN
850 MAT B=B3 @ GOTO 845
```

Notes

- Simple *matrix assignments* are made in lines 125,505,710,800,840 and 850. These require the Math ROM, and greatly help *speeding* the process. If you don't have a Math ROM, you might consider using an emulator (see **Appendix A**), which does include its ROM image. Else, simply substitute all matrix assignments for calls to some subprogram or subroutine to do the assignment.
- Lines 120-135 initialize the program, declaring and initializing all variables, among them matrices B (the board, initialized to a blank position) and F (the move array, initialized using the DATA statements at 440-450.
- Lines 155-205 enter the position to solve either from a data file (155-160) or from the keyboard (180-205), in which case inputs are checked for proper format by means of FNL, a utility multiline user-defined function (405-420).
- If correct, the board is updated with each entry (290-300) and when done, you're given the option of displaying (320-345) the resulting position (very useful for checking its correctness, specially if you're using an HP-IL video display or an emulator). You're then asked for the number of moves to mate (200-205) and the whole problem is then optionally saved to a file (225-230).
- Now, lines 250-270 proceed to search for the mate in (or up to) the specified number of moves by calling the FINDMATE subprogram with appropriately initialized parameters (line 255). Upon returning from the call, the result is properly displayed and program execution gently ends.
- Lines 365-385 include an assortment of utility single-line, user-defined functions to assist in conversions *internal-format* <-> *user-notation*, as well as displaying the moves, including proper handling of underpromotions, captures, pawn moves, and checks/checkmates. This is done quickly and very efficiently thanks to a powerful combination of subtring extractions and logical operations (see in particular lines 375-385, which are well worth some careful study).
- Lines 470-850 comprise the fundamental, *recursive subprogram* FINDMATE which actually does the real work. Local variables are declared in lines 470-475, then lines 485-500 help refine the actual position on the board where White/Black pieces can be located, to help speed the move generator later on.
- Enter the Pseudo-Legal[2] Move Generator for White (505-560), which will generate all pseudo-legal moves for White and will find out if Black has a legal answer for them or else gets checkmated (or stalemated). First we set up a loop to scan the board (505), then make sure we've found a White piece (510). If it happens to be a pawn (520), we proceed to generate and test each possible pseudo-legal move (540-555), else lines 525-535 do likewise for every non-pawn piece, whether far-reaching (Queen, Rook, Bishop) or not (Knight, King).

[2] A pseudo-legal move is one that would be legal if leaving your own King under check were permitted. For efficiency, we only test for full-legality after the move proves to be useful.

In all cases, a call is made to the appropriate subroutine entry point (line 675 or 685) to check full legality and Black responses, if any.

- Lines 580-630 implement the *extremely important*, multi-line UDF that tests whether the specified King (Black's or White's) is under check. This UDF gets called *all the time*, both to check if a generated White move gives check to Black (an essential, time-saving heuristic for the final, deepest ply), if Black's generated answer is legal or else (still) places it in check, and also if a pseudo-legal White move is actually fully legal or not. As it befits its importance and frequence of use, this UDF tries to be as efficient as possible, testing each possible line of attack radiating from the King's position (line 600 for Pawns, 605-610 for Knights, 615-625 for the enemy King) and returning as soon as a check is stablished to exist, if any.

  Also, this UDF illustrates a *very important technique* in HP-71B's BASIC programming, which overcomes a serious limitation of User-Defined Functions, namely that, unlike subprograms, they *can't declare local variables* for its own use. All variables declared in the calling program or subprogram are visible inside the UDF, and all variables declared in the UDF are visible from the calling program/subprogram. This is a real nuisance, because should you need some local variables inside your UDF (for loops, for example) you must make extra-sure that there are no *conflicts* with some external variables, and vice versa. So, you're bound to check each used variable (declared or not) each time you intend to use some inside your UDF, which is error-prone and a real chore.

  But there's a way to use local variables inside an UDF. My technique makes use of the fact that function parameters are truly local, mere place-holders with arbitrary names for any values or actual variable references the calling program or subprogram cares to pass. The parameter variables can't be seen from outside the UDF, so that an X parameter won't conflict at all with an X variable defined outside, as the external variable X can't be seen from inside the UDF, and the internal X parameter can't be seen externally, so any changes to it won't affect the external variable. Changes to the X parameter will have an effect only on the actual variable passed in the function call (say A), and *if and only if* it has been passed by *reference*, not if it's passed by value. So, to declare and use truly local variables in our UDF, we simple declare the variables we need as *parameters* in our function's definition, and simply have the calling program pass 0's or ""'s in their place. This way we can use these local variables, without concern that they might have the same name as any external ones, and without worrying that changing them would change any external variables, as they're "passed" by value (0 or "").

  So for instance, in my **Test Check** UDF (FNJ), in addition to 2 actual parameters (the side to test, **C**, and the King's location, **H**) I want to use 4 local variables inside the UDF (**K**, **E**, **X**, and **V)**, so the necessary definition becomes

```
DEF FNJ(C,H,K,E,X,V)
```

and when calling the function, 0's are provided as place-holders for the *fake parameters* (so using them won't affect external variables K,E,X,V), like this:

```
825 IF FNJ(1,N,0,0,0,0) THEN 850
```

- Lines 675-850 implement the Pseudo-Legal Move Generator for Black (*unrolled for speed*, as the logic is significantly different from White's). First, White Pawn's moves are checked for *promotions* and *underpromotions* (675-680). The details are very similar to White's generator, but we implement here a number of simple yet important time-saving heuristics. For example, line 695 ensures that at maximum depth, *only White moves that result in a check* are considered, as otherwise they can't possibly give checkmate. Then, line 700 tests White moves (which are only known to be pseudo-legal) for *full legality* before committing valuable time and effort to generate Black's responses.
- Line 795 tests the case where Black has *no legal move*, in which case a further test is performed to ascertain whether Black's King is under check or not. If it is, *checkmated* is scored, else Black is just *stalemated* and the search continues.
- Line 825 tests whether Black's pseudo-legal answer results in its King being under check (in which case it is *not* fully legal and another answer is immediately generated), or else it's fully legal, in which case the subprogram calls *itself* recursively (line 835) to search for a mate in the new, updated position (1 White and 1 Black move already made), only in one move less.

Usage instructions

> **Note: If there's some error when entering data (wrong format or files, etc), the program will repeat the input procedure as necessary till everything's correct.**

a) To start the program:

```
RUN [ENTER]     -> MATER 1.0 - (c) V. Albillo
                -> Read from
```

b) To read a saved position from a data file, key in its name and **[ENTER]** else press just **[ENTER]** to see a reminder of correct entry format and be asked for the colour, type, and location for each piece on the board:

```
 [ENTER]        -> Ent: +-/PNBRQK/A-H/1-8
                -> Piece=
```

c) To specify each piece, you must enter a 4-character string, where the 1st character specifies its colour (+ for White, - for Black), the 2nd character specifies its type (**P**awn, k**N**ight, **B**ishop, **R**ook, **Q**ueen, or **K**ing), and the 3rd and 4th characters (**A-H** and **1-8**, respectively) specify its location in *standard algebraic chess notation*. For instance, a White King in its initial position would be entered as **+KE1**. Press **[ENTER]** to enter it on the board, the program will ask for the next piece. Press just **[ENTER]** when there are no more pieces left (see **Examples**).

d) Next, you'll be asked if you want to display the board position:

```
[ENTER]        -> Display board ? Y
```

Accept the default **Y** by pressing **[ENTER]** or else press **N**, then **[ENTER]**. Now you'll be prompted for the (maximum) number of moves to give checkmate in:

```
[ENTER]        -> Mate in [-=up to]: 1
```

e) To search for a mate in a *given* number of moves (say **3**), enter that number. The program will search for a mate in that *precise* number of moves. This is faster, as no time is used searching for mates in less moves, but if one does exist if might not be found, or if found it might not be recognized as a shorter mate (i.e., it might be reported as an M3 though it's actually an M2, say). On the other hand, if you enter a negative number (say **–3**), the program will search for mates *up to* that number of moves beginning with M1, then M2, etc. This is slower, as time is used for the shallower searches, but is *guaranteed* to find and report correctly any mate in the *least* possible number of moves, be it the maximum specified or less.

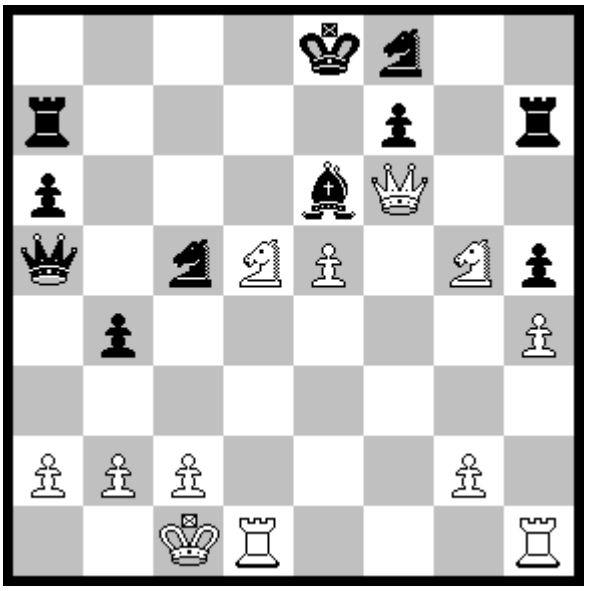
f) Now you'll be asked if you want to save the problem in a file. Key in the name of the file and **[ENTER]** or simply **[ENTER]** if you'd rather not. The program will now begin its search, displaying the White moves it tries in *standard full algebraic chess notation* (labeled with depth and number move) as well as the intermediate and final results (labeled with elapsed time), like this:

Try 3.1: d6xc7	*M-in-3 search, 1st move, pawn captures*
[...]	
Try 3.5: Kf5-f6	*M-in-3 search, 5th move, King moves*
Mate in 3 w/ Kf5-f6 [90"]	*Mate in 3 with King from f5 to f6, in 90"*

The program will stop for you to see the result, press **[ENTER]** to finish.

Examples

1. White to play and mate in 2 or less

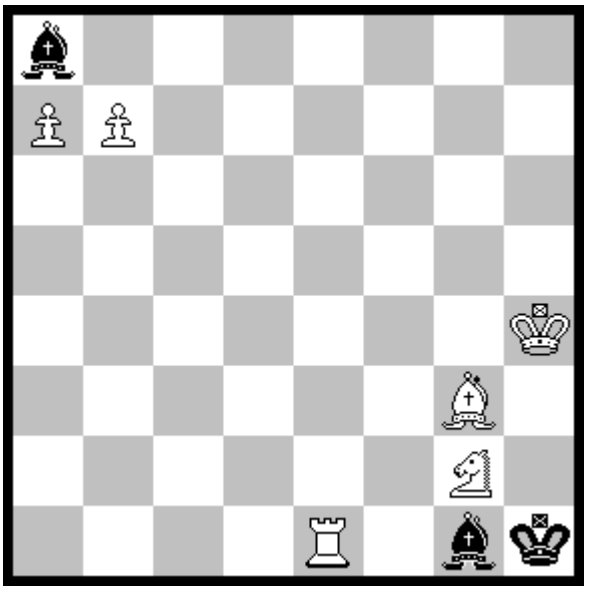


```
>RUN        [ENTER]
MATER 1.0 - (c) V. Albillo
Read from   [ENTER]
Ent: +-/PNBRQK/A-H/1-8
Piece=-KE8  [ENTER]  Piece=+ND5 [ENTER]
Piece=-NF8  [ENTER]  Piece=+PE5 [ENTER]
Piece=-RA7  [ENTER]  Piece=+PH4 [ENTER]
Piece=-PF7  [ENTER]  Piece=+PA2 [ENTER]
Piece=-RH7  [ENTER]  Piece=+PB2 [ENTER]
Piece=-PA6  [ENTER]  Piece=+PC2 [ENTER]
Piece=-BE6  [ENTER]  Piece=+PG2 [ENTER]
Piece=-QA5  [ENTER]  Piece=+KC1 [ENTER]
Piece=-NC5  [ENTER]  Piece=+RD1 [ENTER]
Piece=-PH5  [ENTER]  Piece=+RH1 [ENTER]
Piece=-PB4  [ENTER]  Piece=+NG5 [ENTER]
Piece=+QF6  [ENTER]  Piece=     [ENTER]
Display board ? Y  [ENTER]

     A  B  C  D  E  F  G  H
   ------------------------
 8 [  .  .  .  . -k -n  .  . ] 8   Mate in [-=up to]:-2 [ENTER]
 7 [ -r  .  .  .  . -p  . -r ] 7   Save in  [ENTER]     Try 2.1: Qf6-g6
 6 [ -p  .  .  . -b +Q  .  . ] 6   Try 1.1: Qf6xf7+     Try 2.2: Qf6-h6
 5 [ -q  . -n +N +P  . +N -p ] 5   Try 1.2: Qf6-e7+     Try 2.3: Qf6-g7
 4 [  . -p  .  .  .  .  . +P ] 4   Try 1.3: Qf6-d8+     Try 2.4: Qf6-h8
 3 [  .  .  .  .  .  .  .  . ] 3   Try 1.4: Qf6xe6+     Try 2.5: Qf6xf7
 2 [ +P +P +P  .  .  . +P  . ] 2   Try 1.5: Nd5-c7+     Try 2.6: Qf6-e7
 1 [  .  . +K +R  .  .  . +R ] 1   No mate in 1[40"]    Mate in 2 w/Qf6-e7
   ------------------------                             [480"]   [ENTER]
     A  B  C  D  E  F  G  H                             Bye
                                                        >
```

Notes: We specified **–2** as the number of moves to mate, so the program searches for Mate-in-1, then Mate-in-2. If we had specified **2** (instead of **–2**), a possible mate in 1 (if it existed) might not have been found or reported as such. Notice that when searching for Mate in 1, *only* the 5 White moves which give check are tried (and labeled with a "+", for "check"). But when searching for Mate in 2, *all* legal White moves are tried till a checkmating move (**Qf6-e7+**) is found. Notice that the move *isn't* labeled as a check (ending "+"), though it actually is. Also, both the option to read the problem from a data file ("**`Read from`**") and the option to save the problem to a data file ("**`Save in`**") are declined by just pressing **`[ENTER]`**, though the option to *display the board* in the (virtual or real) HP-IL video display ("**`Display board ? Y`**") is duly accepted by default by pressing **`[ENTER].`**

2. White to play and mate in 4 or less



```
>RUN       [ENTER]
MATER 1.0 - (c) V. Albillo
Read from  [ENTER]
Ent: +-/PNBRQK/A-H/1-8
Piece=-BA8 [ENTER]
Piece=+PA7 [ENTER]
Piece=+PB7 [ENTER]
Piece=+KH4 [ENTER]
Piece=-KH1 [ENTER]
Piece=+BG3 [ENTER]
Piece=+NG2 [ENTER]
Piece=-BG1 [ENTER]
Piece=+RE1 [ENTER]
Piece=     [ENTER]
```

```
Display board ? Y  [ENTER]

     A  B  C  D  E  F  G  H
   -------------------------
 8 [ -b  .  .  .  .  .  .  . ] 8
 7 [ +P +P  .  .  .  .  .  . ] 7
 6 [  .  .  .  .  .  .  .  . ] 6
 5 [  .  .  .  .  .  .  .  . ] 5
 4 [  .  .  .  .  .  .  . +K ] 4
 3 [  .  .  .  .  .  . +B  . ] 3
 2 [  .  .  .  .  .  . +N  . ] 2
 1 [  .  .  .  . +R  . -b -k ] 1
   -------------------------
     A  B  C  D  E  F  G  H

Mate in [-=up to]: -4 [ENTER]
Save in   AMAZING1    [ENTER]
Try 1.1: Re1xg1+
No mate in 1 [60"]
Try 2.1: b7xa8=Q
Try 2.2: b7xa8=R
Try 2.3: b7xa8=B
Try 2.4: b7xa8=N
Try 2.5: b7-b8=Q
Try 2.6: b7-b8=R
Try 2.7: b7-b8=B
Try 2.8: b7-b8=N
Try 2.9: Kh4-h5
Try 2.10: Kh4-g5
Try 2.11: Kh4-g4
Try 2.12: Kh4-h3
Try 2.13: Bg3-f4
Try 2.14: Bg3-e5
Try 2.15: Bg3-d6
Try 2.16: Bg3-c7
Try 2.17: Bg3-b8
Try 2.18: Bg3-f2
Try 2.19: Bg3-h2
Try 2.20: Ng2-f4
Try 2.21: Ng2-e3
Try 2.22: Re1-f1
Try 2.23: Re1xg1
Try 2.24: Re1-e2
Try 2.25: Re1-e3
Try 2.26: Re1-e4
Try 2.27: Re1-e5
Try 2.28: Re1-e6
Try 2.29: Re1-e7
Try 2.30: Re1-e8
Try 2.31: Re1-d1
Try 2.32: Re1-c1
Try 2.33: Re1-b1
Try 2.34: Re1-a1
No mate in 2 [2207"]
Try 3.1: b7xa8=Q
Try 3.2: b7xa8=R
Try 3.3: b7xa8=B
Try 3.4: b7xa8=N
Mate in 3 w/ b7xa8=N
[6945"]  [ENTER]
Bye
>
```

Notes: We specified **–4** as the number of moves to mate (and saved the problem as a data file called AMAZING1), so the program searches for Mate-in-1, -in-2, -in-3, and –in-4. This guarantees that if a mate is possible in *less* than 4 moves (as it's actually the case here), it will be reported as such. White has **34 legal moves** in this position, but for Mate-in-1 the *only* move ever tried is the only one that gives check, and for Mate-in-3 only 4 White moves are tried before the *amazing underpromotion to a Knight* (!!) is found to actually achieve mate in 3 (not 4).

Appendix A: HP-71B emulators

At long last, it seems the time has really come for *all* to use and appreciate the 71B's power. Now there exists an *amazing* HP-71B emulator developed by Jean-François Garnier, which will run under Windows at *speeds up to 25x* that of the original 71, including support for virtual (and even real!) HP-IL peripherals such as 80-column video displays and disk drives, input/output access to PC files, and large RAM/IRAM. The standard version of this emulator is downloadable *for free* from the author's web site, together with an assortment of ROMs including the System ROMs themselves, *Math*, HP-IL, JPC, even the Forth/Assembler ROM.

You can use it to run the program presented in this article or any others (be they BASIC or binary files (LIF format)) available on the Net, and of course create and execute your own, with the convenience of using the PC keyboard and monitor. For details and downloads, try this URL:

Web Site: http://membres.lycos.fr/jeffcalc/emu71.html

There also exists an HP-71B emulator for the 48/49 Series calculators, which offers the great advantage of *true pocket portability* as well as 2x-3x faster speed than the original 71B. If interested, you'd be well advised to contact the original author for details, at this URL:

Web Site: http://hrastprogrammer.tripod.com/HP71X/index.htm

Appendix B: Possible improvements

As this is not a production program, just a test to prove a point, it makes little sense to try and improve its performance, which cannot match that of dedicated programs or hardware, but you might want to try the following, as a useful exercise:

- Instead of using INTEGER arrays for the boards, set aside some variable or data file in RAM and use PEEK and POKE to access board locations. Their contents can be represented in 4 bits (one *nybble*) so you'll save *a lot* of RAM when making deep recursive calls and/or storing positions in data files.
- Implement full-legal *castling* and *en-passant* pawn captures.
- Implement **Test Check** UDF (FNJ) in Assembler. This will speed it all *greatly*.
- Implement some way to *edit* any entered position, to avoid having to *reenter* it to correct mistakes. Also, check the position for chess *legality* (two kings, etc).
- Implement a **Checks-Only** mode which will quickly find mates where all White moves are checks. This will find such mates in a fraction of the usual time.