The hint

Replicators (part 1)

Having to make the hint of the week, I start with the kind of warriors I like more and I can do better, replicators, or paper; the sort of warrior that use the sheer number to overcome the enemy.
Paper warriors, like every other, have evoluted a lot from the beginnings of the game; presently they use almost all the so called 'silk' style, i.e. splitting before copying. This can be done only under 94 rules because requires post increment and a-field addressing. Now let's give a look at a very simple guy.

```
start   spl     1
        mov     -1,     0       ;generate 3 parallel processes

1 silk  spl.a   @0,     100     ;split
2       mov.i   }silk,  >silk   ;copy
3       jmp.a   silk,   {silk   ;repeat the thing resetting pointer
```

First two lines generate 3 processes that execute the same line one after the other, before executing the next. First line creates another process to execute line start+1, then process 1 copy start line over the mov and process two splits, adding another process to execute silk. The simpler way to generate an exact number of parallel processes is converting the number required in binary 3 -> 11, subtract one -> 10, use a spl 1 for every one and a mov -1,0 for every zero. Much simpler to do than to tell. For the warrior to work we need at least as many processes as we have lines to copy.
Let's go back toour warrior; now we have three processes executing line 1 they split, where, at the a-field address i.e. the address pointed by b-field of line 0 locations away, the b-field of the line they are executing, 100 locations away. When all three process executed this line we have three others process ready to execute line silk+100, there is nothing to execute here but we have some time because new generated processes are queued after those executing the split.
First three processes now execute line 2, they move what's pointed by a-field of line 1 to the location pointed by b-field of line 1 then they increment both a and b field of line 1. First process moves line 1 100 cells away from line 1 and leaves line 1 changed such a way:

```
1 silk  spl.a    @1,    101
```

so it copyes line 2 101 cells away from silk, just after the previous line. Process 3 does same thing copying line3.
Now it's the turn of the new processes, those created by line1, to execute, they are not more sitting on an empty cell but over the copy of line1 created by line2, they execute it and begin creating third generation copy. First three processes now reach line3, now the warrior has modifyed in such way

```
1 silk  spl.a   @3,     102
2       mov.i   }silk,  >silk
3       jmp.a   silk,   {silk
```

The a-field of line 3 is the address of the jump while b-field decrements a-field of line 1 so that the warrior can go on splitting and copying.

This one is not a real warrior, his offensive potential is too small, it's just to understand how a silk replicator works. Simple improvements are adding an add line so as copies are not packed one near the other, and adding some bombing to make it a bit nastier. The warrior following is Paperone, my first warrior to enter 94 hill, it was on top of beginner hill for some time a few months ago.
It's similar to the example in the FAQ (very similar indeed :-) but to make it run well I had to work on the many constants.

```
;redcode-94
;name Paperone
;author Beppe Bezzi
;strategy Silk replicator
;kill Paperone
;assert CORESIZE == 8000
```



```
start   spl     1,      <300    ;\
        spl     1,      <150    ;  generate 7 consecutive processes
        mov     -1,     0       ;/

silk    spl     3620,   #0      ;split to new copy
        mov.i   >-1,    }-1     ;copy self to new location
```

;this is another way to copy using multiple processes, the other one is a bit better because we can decrement the cell we are splitting to and, if we are lucky, kill an imp.

```
        mov.i   bomb,   >2005   ;linear bombing
        mov.i   bomb,   }2042   ;A-indirect bombing for anti-vamp
```

;The first bomb laid down acts as a pointer for the following stream, laying down a carpet.

```
        add.a   #50,     silk    ;distance new copy
        jmp     silk,   <silk   ;reset source pointer, make new copy
bomb    dat.f   >2667,  >5334   ;anti-imp bomb
```

This is very effective against 3 points imp rings. A lucky hit on the executing process can kill many others; other kinds of bombs are used, by me at least, we'll discuss them another time.

Another time we'll discuss more advanced questions: another replicating engine, that is better than this one, and some other paper related topics like spread constants, bombs, strategies...

For questions mail me <bezzi@iol.it> or if you think it's of general interest post to rec.games.corewar

Anyone with hints or warriors to publish is welcome.

The hint

Do you often ask yourself: Why do my warriors do well on the beginner's and get thrashed on the '94 draft hill? This hint should help all newcomers in writing more viable code.

I have asked the 15th place author to send me his warrior so we can dissect it as a case study. Thanks to Scott Manley for sharing his code. If you are in 15th place around the end of the week, you could be getting mail from me. Unfortunately, Mutagen has dropped off the hill, but I suspect that Scott will have a brand new version real soon.

```
;redcode-b
;name Mutagen
;author Scott Manley
;strategy Scan -> SPL/JMP bomb -> split to Imp gate / 2 pass  forward
;strategy travelling core clear
;assert CORESIZE==8000

plength EQU     35
inc     EQU     6
carpet  EQU     (CORESIZE-MINDISTANCE)/inc


adds    DAT     #inc , #inc
begin   SPL     clear2
start   SEQ.I   *scan, @scan
        JMP     scan1
cont    ADD.F   adds , scan
        DJN     start , count
        JMP     clear3
scan1   MOV     sbomb , * scan
scan2   MOV     sbomb2 , @ scan
```

```
        JMP     cont
scan    DAT     MINDISTANCE-10 , MINDISTANCE-9
target1 DAT     23 , -23
target2 DAT     22 , -22
target3 DAT     -1 , -21
target4 DAT     -2 , -20
        DAT     0 , 0
        DAT     0 , 0
        DAT     0 , 0
        JMP     0 , } cont
clear2  MOV.I   sbomb , } target1
        DJN     clear2 , target1
        JMP     clear4
clear3  MOV.I   target2 , } target2
        DJN     clear3 , target2
clear1  MOV.I   target4 , { target4
        DJN     clear1 , target4
        JMP     0 , > -10
clear4  MOV.I   sbomb , { target3
        DJN     clear4 , target3
        MOV     0,1
sbomb   SPL     0 , 0
sbomb2  JMP     -1
count   DAT     #carpet-1,#carpet-1
        END     begin
```

The basic concept is to scan and spl coreclear simultaneously. Once the scan is finished (one pass through core), jump to a dat coreclear to follow the spl clear. There are other startegies involved but we'll just focus on the scan engine and multi-pass coreclear.

Mutagen's scan is 11 lines long--13 lines if the stun bomb is included in the count. We can reduce the length a bit without affecting the behavior too much by using the following code:

```
        add.f split, scan
scan    sne.i loc1, loc+1
        djn.f -2, <DJNSTREAM1   ;djn.f will decrement both the a and b field.
        mov.i split, *scan
        mov.i jump, @scan
        djn.f -5, <DJNSTREAM2   ;need to have a trigger so this falls thourgh
        jmp clear               ;after scan is complete.
split   spl #step, #step        ;try to find other uses for these
jump    jmp -1                  ;can be anywhere in code
```

Remember that the bigger your executing code is, the more inviting target you make to the enemy. The split and jump lines can be moved away from the executing code to provide the smallest possible profile. Also: the scanning step is not optimal. If an enemy instruction is at location N, most likely there will be another instruction at N+1. So there is no need to scan there. There are two basic ways to spread the scan out. One is to scan N and N+step then bomb every location in between. Another is to bomb N and then check N+step in a separate step. Check out Agony and Irongate for good examples of these two methods.

Next let's talk multi-pass coreclears. Muatagen's clear is lengthy and easy prey for scanners. In a multi-pass coreclear, only a few things change--the bomb being swept through core and the pointer that does it. We can reuse the movement instructions and just change the bomb and pointer with this code:

```
org a1
ptr1    dat a1, end+100
a4      dat 0,  end+2+1
a3      dat 1,  end+2+2
a2      spl #2, end+2+3         ; spl #X, <-Y acts like a split 0.
a1      spl #3, end+2+4         ; you can use x and Y as step values
        mov *ptr1, >ptr1        ; and use the decrement in the b-field
        mov *ptr1, >ptr1        ; as part of an imp gate.
        mov *ptr1, >ptr1        ; > (post-increment) keeps adding 1 to
```



```
end     djn.f -3, <4000         ; the b-field of ptr1 to move the bomb
                                ; through core.
```

The clear starts with a1 being swept forward through core. Notice that the move instructions use a pointer (ptr1) to determine which bomb to sweep with. ptr1's b-field is also used to do the actual clearing. When the clear wraps around the core, it will eventually overwrite ptr1 with a1. Now the move instructions will look where the old ptr1 used to be to find what they should move through core. The new ptr1 (a1) points towards a2, so the move instructions will move a2 through core. The new pointer also has a new b-field. The value of this b-field ensures that the clear skips over the core clear code. Then a2 overwrites the pointer. The pointer is now points to a3. And so on. This code will continue to clear until time run out.

7 44/ 38/ 18 Hint Test M R Bremer 151 5

A simple version of these improvements has been submitted to the beginner hill. The code does reasonably well against bombers and replicators, but its large size makes it vulnerable to scanners. A good project would be to pspace it with a small fast bomber.

Extra Extra:

P.Kline usually publishes his warriors fairly quickly. However Die Hard has been a mystery for quite some time. Magnus Paulsson is one of the few authors who has had some success against the many interations of Die Hard. Paulsson has been kind enough to share his thoughts (along with his newest program--theMystery) with us:

How does Die Hard work?
That thought came to mind when some version of myVamp didn't score more than 1% wins! It was double disturbing as myVamp has a coreclear spl/spl/dat and a djn stream at the same time, no normal imp could survive it (nothing else either if lucky).

Why is that?
If you get the other program in such a clear it will fast spl it self to 8000 processes. No you have about 8000 cycles and then the imp spiral will execute. In my coreclear I change a location in core every cycle which means that when the imp-spiral execute it is (should be) overwritten/djned already.

What then?
Now if you place two spirals on top of each other, and plan in which order they and the rest of the code will be in the execution. In the coreclear the spl processes will be like 4000 processes, spiral, 4000 processes, spiral. That means the clear has to kill spiral in 4000 cycles which isn't possible in a clear.

Is that how Die Hard works?
(Actually I don't know :-) But I'll keep on guessing for a while.)
So, because there is a thing called gate which kills spirals. In order to tie you have to have something like 100 processes in a spiral to slow it down as much that it doesn't reach the gate. Now you can't launch such a monster without getting killed before the launch is complete.

How then?
Have a look at theMystery!
It is three papers working together to launch imp-rings in such a way that there will never be more that say 1500 cycles between a ring is executed.

What is the diffrence between Die Hard and theMystery?
Die Hard is a lot better :-(, I don't know how Paul did it but theMystery doesn't kill anything! (I'm totaly wortless on papers, I can't get the constants right and it's to bloody complicated to get a paper effective.)

Actually I don't belive that theMystery could ever beat anything 20%, maybee Paul has taken the idea one step further? (stone?)

(Now I resubmitted theMystery1.5 to the hill and it wins 17% :-)

15 25/ 19/ 57 Die Hard P.Kline 131 3

```
..
20  17/  7/ 76             theMystery1.5             Paulsson     127       1
```

/Magnus Paulsson
(More fun to write this than to think about Peierls substitution)
(I should heve mentioned CottonDH (J.Wilkinson) but I wont :-))

Paul responds:

Magnus' approach is intriguing. Lots of ties but not many wins, which is about where Die Hard was until I worked in some bombing. Not much but enough to pump up the score and move him up the Hill. Die Hard doesn't look much like theMystery, but his kernel is based on something I published a looong time ago.

Another tidbit about Die Hard. He starts with a very brief quick-scan and vamp. The pit does a one-pass suicidal clear incorporating a brainwash which, when it works forces programs like Jack in the Box to revert from paper to something which Die Hard can kill. Instead of going 1/0 he now goes about 12/0 against JitB :-)

theMystery sure does a lot better against myZizzor than Die Hard does :-(
(I have a pretty good idea what Magnus' missing line in myZizzor might be!)

Paul

```
;redcode-94
;name theMystery1.5
;author Paulsson
;strategy How does Die Hard work? (this way maybee?)
;strategy Looking gooood, could it be somthing that begins
;strategy with an i and ends with p?
;assert CORESIZE > 1
;kill theMystery

org start

step1   equ             1800
step2   equ             -1922

start   spl 1           ;\
        mov.i -1,0      ;- make 7 processes
        mov.i -1,0      ;/
        mov {ptr2,<ptr2 ; move out second paper
        mov {ptr1,<ptr1 ; move out first paper
        spl 3
        spl 4
        jmp @ptr1       ; jump to 1
        mov {ptr3,<ptr3 ; move out third paper
        jmp     @ptr3   ; jump to 3
        jmp @ptr2       ; jump to 2

pap     spl step1,0             ;\
        mov.i >-1,}-1           ;\\ Normal paper, with bad constants
        mov.i <-2,<1            ;// (I think I riped it from timescape :-) )
        spl     @0,}step2       ;/
        mov.i #0,2667           ;Impy!

ptr1 dat 5+pap,pap+5+500
ptr2 dat pap+5,pap+5+2667+500
ptr3 dat pap+5,pap+5+2667*2+500
```

```
Most losses:
Program "myZizzor" (length 58) by "Paulsson"
(contact address "mpn@ifm.liu.se"):
;strategy Cissors (or whatever way you spell it)
;strategy Let's se how hard Die Hard is this time :-)
theMystery1.5 wins: 27
myZizzor wins: 55
```



```
Ties: 168

Program "Anti Die-Hard Bevo (3c)" (length 76) by "John Wilkinson"
(contact address "jwilkinson@mail.utexas.edu"):
;strategy I didn't have a single program on my HD that could beat
;strategy my Cotton-DH, so I made this.  Let's see how it fares
;strategy against the real Die Hard. :)
theMystery1.5 wins: 60
Anti Die-Hard Bevo (3c) wins: 54
Ties: 136
```

Questions? Concerns? Comments? Complaints? Mail them to:
Beppe Bezzi <bezzi@iol.it> or Myer R Bremer <bremermr@ecn.purdue.edu>

The hint

Replicators (part 2)

Hi, happy to see you again.

Last time we spoke of basic replicator concepts, now I'll try to speak of some advanced topics.
To begin let's give a look at another replicating engine, the best one in my opinion, first introduced by Jippo Pohjalainen in its warrior Timescape.
We report slightly simplified, the way it has been proposed as White warrior by Nandor and Stefan in the tournament.

```
warrior
        spl     1,              <-200
        mov.i   -1,             0           ;this block generates 6 processes
        spl     1,              <-300


tim2    spl     @tim2, }TSTEP
tim2a   mov.i   }tim2,  >tim2

cel2    spl     @cel2,  }CSTEP  ;these four lines are the main body
cel2a   mov.i   }cel2,  >cel2
                                ;here you can insert some bombing line
ncl2a   mov.i   {cel2,  <ncl2
ncl2    jmp     @ncl2,  >NSTEP
```

All you know, having read part 1, how the first four lines work, they split away and copy the warrior body where the processes are going to execute, is worth noting that the lines cel2, cel2a don't copy the warrior from the beginning but copy two blank lines in the bottom, after ncl2.
Line ncl2a copies again the warrior, fron cel2 to ncl2+2, backward because of the pre decrements and last line jumps to the beginning of this copy resetting the pointer.
The main advantage of this structure is that all the code is executed but once, to be left as a decoy to foul scanners; this is a great advantage compared with the older structure of the first hint. Another advantage is that the warrior will continue to work, slowed, even if wounded by a bomb in its last two lines.
This guy was the harder thing to kill before Paul Kline created Die Hard.
With this structure have been made some others replicators of success, worth mention are Nobody special by Mike Nonemacher and Marcia Trionfale by...me.

Now we have a solid structure to work on, to make it deadlier we can add some other form of attack than overwriting our opponent. The original Timescape has this single bombing line inserted after cel2a:

```
        mov.i   <-FSTEP,{FSTEP
```

how it works, remember we have some processes working in papallel:
every process takes the cell -FSTEP away, decrements its b-field, take the

cell pointed by and moves it in the position pointed by the decremented
a-field of the cell FSTEP cells away. Simple? NO! :-)
OK. From the beginning:

```
        dat     0,0
-FSTEP  dat     0,0             ;will became dat 0,-1
...
        mov.i   <-FSTEP,{FSTEP  ;here we are
...
begin   mov     bomb,   nearme
...     [enemy code]            ;Our enemy is here, we are lucky :-)

end     jmp     begin
        dat     0,0
FSTEP   dat     0,0             ;will became dat -1,0
```

Now 1st process takes the cell -FSTEP and decrements its b-field, takes the
cell pointed by the decremented b-field (in the example the cell before) and
moves it; where? It takes the cell FSTEP and decrements its a-field thake
the cell pointed by it, here he hits. Missed, don't worry we have process 2
taking cell -FSTEP-2 and moving it at FSTEP-2 and so on till we have
processes executing the bombing line. At the end the enemy is no more, in
the example at least.

Bombing is useful not only to get rid of our enemy but also to get rid of
ourself ... yes, enemy scanners have the bad use to cover our poor
replicators with carpets of spl 0 and similar nasty things. Those bombs
don't kill, but cause us to generate unuseful processes slowing down our
spread. If we bomb with dat our old copies, that have a chance to be
infected, we can reduce this effect; should happen we hit a good copy don't
worry, we are so many that we can withstand a few losses.

Others warriors use different kind of bombs, more useful to kill our
enemies, the drawback is that we have to carry the bomb with us. The bombing
line will beacme:

```
        mov     bomb,   <target ;or > or { or }
```

now the first bomb laid down will become the pointer for the following carpet.
Most used bomb is the anti imp bomb

```
        dat     <-2666, <2667
```

this bomb is very good at killing 3 points imp ring, otherwise difficult to
kill by replicators.
Another bomb I used with some success, in Jack in the box, is this simple one:

```
        dat     1,      1
```

This bomb is targeted against djn streams and forward clears, two forms of
attack often used by paper enemies. The effect on streams is to make the
process go ot of the loop, wasting time; the effect on forward clears is
deadly, look at a simple forward clear

```
gate    dat     100,    1000    ;the clear is running 1000 cells away
....
clr     mov     bomb,   >gate   ;what's bomb don't matter, sure nothing with
        jmp     clr             ;a b-field of 1
```

If we hit gate with a dat 1,1 the clear will begin running inside itself,
till it reaches clr line and self destructs, very effective and very funny :-)

Like the bombing/scanning step for stones and scanners the spread constants
can make the difference beetween a good and a bad warrior. You have to
choose them so as to assure a good spread of the copies in the core.
Corestep.c by Jay Han and Mopt by Stefan Strack, available at the FTP site,
can give you a starting point, but for replicators the job is, far more
complex because they change their constants in the spread process; let me
explain with an example, same structure 4 parallel processes:

```
a       spl     @0,     100
b       mov     }-1,    >-1
c       mov     {d,     <d
d       jmp     @0,     >1000
```


First time lines a-b are executed they splits and copy 100 locations away
but, when lines c-d copy them the value of b-field is 104, and so on.
I don't know any mathematical method or optimization program to find best
values and I look at what happens using pmarsv. If I notice that modules
don't spread well I change something and so on, art more than science.
In the replicator I'm working at now I use a step modulo 200 for first
constant (anything beetween 100 and 400 is good) a mod 20/40 for second one
and ... my nose :-) for the last one.
Stefan Strack suggested a method using Pmars macros to automatize, in part
at least, the search; here is what he says:

A better way to optimize constants
is to run your warrior with pmars and use cdb macros that change code
sections and record the result. Suppose we want to optimize a slighly
"un-optimized" version of T.Hsu's Ryooki:

```
nxt_paper   equ     100 ;chosen with room for improvement

boot_paper  spl     1 ,>4000
            mov.i   -1,#0
            mov.i   -1,#0

paper       spl     @paper,<nxt_paper   ; A-fld is src, B-fld is dest
copy        mov.i   }paper,>paper
            mov.i   bomb  ,>paper       ; anti-imp
            mov.i   bomb  ,}800         ; anti-vampire
            jmn.f   @copy ,{paper
bomb        dat     <2667 ,<2667*2
```

and we want to find a better offset between copies than the "100" in the
nxt_paper EQU. First we need to come up with some good
ways to measure an even spread between paper bodies in core. Here's an
approximation that cdb can easily provide:

```
       after a few thousand cycles, a paper with a good offset
       1) has more processes
       2) covers more core locations
       than a paper with a bad offset
```

Now the idea is simply to run multiple rounds, systematically changing the
silk offset at the beginning of each round, and having cdb report process
number and number of covered core locations after 5000 cycles or so. This can
all be automated with macros, so you can have pmars find optimal constants
while you get coffee (jolt? :). Once you have a few candidate offsets, you
should make sure they're working as you expect by looking at the core
display. You can than go on to find optimal bombing constants for your
set of optimal offsets in pretty much the same manner. As an example using
Ryooki above:

```
pmars -br 1000 -e ryooki.red
00000   SPL.B  $     1, >  4000
(cdb) 0,7
00000   SPL.B  $     1, >  4000
00001   MOV.I  $    -1, #     0
00002   MOV.I  $    -1, #     0
00003   SPL.B  @     0, < 100
00004   MOV.I  }    -1, >    -1
00005   MOV.I  $     3, >    -2
00006   MOV.I  $     2, }   800
00007   JMN.F  @    -3, {    -4
(cdb) calc i=99
99
```

This sets a variable "i" to our starting constant.

```
(cdb)@ed 3~spl @0,<i=i+1~@sk 5000~@pq~ca i,$+1~@pq off~m count~@go~@st
100,987
1830
```

```
(cdb)
```

This is a bit complicated. The "@ed 3~spl @0,<i=i+1" sequence edits address 3 and writes to it the instruction "SPL @ 0, < 100", having incremented the "i" variable by 1. "@sk 5000" executes 5000 cycles silently, "@pq" then switches into "process queue" display/edit mode. "calc i,$+1" echoes the current value of the "i" variable, followed by the number of processes ("$" is the number of the last process). The output is seen on the next line: "100,987". "@pq off" then switches back into core display/edit mode. "macro count" executes a macro that is already defined in pmars.mac; the "count" macro simply echoes the number of core locations that have anything other than "dat 0,0" in them (here: 1830). Finally, "@go~@st" advance to the end of this round and to the first cycle of the next round.

When you now press <Enter>, the command sequence is repeated with an offset value of 101:

```
(cdb) <Enter>
101,1058
1971
(cdb)
```

The 101 offset results in a greater number of processes (1058) and more addresses written to (1971). If you want to run the whole thing automated, just inclose the command sequence in a loop (!!~...~!) and send the results to a file like so:

```
(cdb) ca i=99
99
(cdb) write ryooki.opt
Opening logfile
(cdb) !!~&ed 3~spl @0,<i=i+1~&sk 5000~&pq~ca i,$+1~&pq off~m count~&go~&st~!
```

To avoid sending _a_lot_ of garbish output to the log file, we have to use & in stead of @ in this macro and in the macro count in pmars.mac; just edit it.

```
count= &ca z=.~m w?~&ca x=.,c=0~!!~m w?~&ca c=c+1~if .!=x~!~ca c~&l z
w?= &search ,
```

You can easily make this more complicated by only echoing #processes/locations if the values are larger than anything so far (left as an exercise to the reader), but at this point you are probably ready to save yourself some typing by defining your own macros. Remember that you can add macros from within the cdb session using the "@macro ,user" command (a shorthand is "m="). You could even replace the rather simplistic check for #processes/locations with a more elaborate macro that calculates the variance of intervals between papers.

Now we are ready to start making a paper warrior, what we have to do is putting things together and begin working.

First the structure, we'll make a mid-size warrior, 8 lines, so we need 8 processes.

```
start   spl     1,      <300    ;so we make 8 parallel processes
        spl     1,      <400    ;the <### are not needed to make it work
        spl     1,      <500    ;but may damage something and cost nothing

silk    spl     @0,     {dest0
        mov.i   }-1,    >-1
silk1   spl     @0,     <dest1
        mov.i   }-1,    >-1
        mov.i   bomba,  }range
        mov     {silk1, <silk2
silk2   jmp     @0,     >dest2
bomba   dat     <2667,  <1
```

Now the constants: dest0 is the less used, let's take a modulo 200 value, for dest1 we take a mod 20 one. Now we begin optimization using Stefan method. I have a rather slow computer so I choosed to analyze but values ranging from -2000 to -1000. Before doing so I changed the mov bomb line in

a nop instruction, optimizing bombing will come later.

Running Stefan's macro I got -1278 as best value.

Then I replaced the nop with a mov and runned again the macro, choosing a range for bombing beetween 500 and 1000. Best value 933

I put values in the warrior and submitted it to 94 hill: score: 125.98
Not bad, a little better than hand made one.

For you to enjoy here is the code to play with.

Boyz on the hill, ready your scanners. They are coming :-)

```
;redcode-94
;name paper01o
;author Beppe Bezzi
;strategy paper module, partially optimized with pmars

;assert CORESIZE == 8000

dest0   equ     2200
dest1   equ     3740
dest2   equ     -1278   ;pmars optimized
range   equ     933     ;pmars optimized

paper
        spl     1,      <300    ;\
        spl     1,      <400    ;-> generate 8 consecutive processes
        spl     1,      <500    ;/

silk    spl     @0,     {dest0
        mov.i   }-1,    >-1
silk1   spl     @0,     <dest1
        mov.i   }-1,    >-1
        mov.i   bomba,  }range
        mov     {silk1, <silk2
silk2   jmp     @0,     >dest2
bomba   dat     <2667,  <1

end     paper
```

For next hint I would like a little input from you about the argument to be treated; my first choice is p-space followed by bombers, two arguments I know at least a little, having made some successful warriors, but I wish to hear from you.

For questions mail me <bezzi@iol.it> or if you think it's of general interest post to rec.games.corewar

Anyone with hints or warriors to publish is welcome.

The hint

P-space

Hi,
this time we'll speak of p-space, the last tool, implemented by pmars08, that allows our warrior to change strategy according to the history of the match.
P-space is a protected area of memory, i.e. every warrior has its own p-space and cannot read or write opponent's one. P-space hold but values, not whole instructions, and is accessed by two specific instructions LDP and STP load and store Pspace.

At the beginning of every round p-space cell 0 holds the result of previous round, -1 at the very beginning, others cells hold the value they had at the end of previous round, 0 at the start of the match. The value is 0 if we lose, and the number of alive warriors if we survive; in standard one against one matches those values are 1 for the win and 2 for the tie.

Warriors using p-space are called p-warriors or p-switchers; they store in a location of p-space informations on the strategy they are using, at the end of the round they evaluate the result of prvious round and, according to it and, sometimes, the result of others rounds, continue with current strategy or change to another, in the hope of doing better. In practice, if you are a general, planning long term strategies, the switcher is your colonel, deciding on battlefield what to use against your opponent.
It's important to say that even the best switching routine is worthless if you don't have sound combat routines; if all you components lose against your enemy, the mix will lose too, sometimes even worst because you lose some time at the beginning to pplan the round, and to boot components away from the big warrior body.
P-space is a tool for intermediate players, not for beginners; until you don't have at least two average level different warriors, a stone and a paper for example, you cannot get anyhing good from it.

Now let's see how to assemble a p-warrior; first we need good components, able to score points against different kind of enemies; we have them:
Paper01, to score against enemy bombers, juliet storm, to kill enemy scanners; against enemy paper we are not defenceless, a paper usually cannot beat another paper, so we should score:
Well against bombers, thanks Paper01
Well against scanners, thanks juliet
Ties against replicators, thanks Paper01.

Once chosen our hands, we have to assemble the brain; unless you want to do something very complex, the switcher is not a difficult thing to do. Let's give a look at a very simple one, and BTW successful, the switcher of Jack in the Box, for three main reasons: it's one of my warriors, is doing well, is the only one published :-)
Jack's has two components, a very heavy replicator, four times Paper01, and a very fast bomber, Tornado.
Its strategy is simple: the replicator scores lots of points against enemy bombers but, because of the size, is rather vulnerable to scanners; well if we are winning or tieing all right, we continue with the same strategy; if a bad scanners happens to kill the paper, BOOH, Tornado pop up and with its high speed and colored bombs kill him.
Here it's, very simple indeed.

```
        _RES    equ 0           ;here pmars loads results
        _STR    equ 1           ;here I store my strategy

res     ldp.ab  _RES,   #0      ;load result last match
str     ldp.a   _STR,   str1    ;load strategy in use
        sne.ab  #0,     res     ;check result, win or tie OK
lost    add.a   #1,     str1    ;lost change
        mod.a   #2,     str1    ;secure jump
win     stp.ab  str1,   _STR    ;save strategy
str1    jmp     @0,     juliet
        dat     0,      paper
```

We load in res.b the result of last match, in str1.a the strategy we used, then we compare res to 0, if it's zero we add one to the strategy, if it's different, tie or win, we don't. The mod instruction assure us to have a value of 0 or 1.
At last we save new strategy for the round, and we jump at bomber or paper, according to str1.a
In 7 cycles we have finished, so even a Qscan has hard times to hang on.
Now the code, nothing more than taking the waariors, the switcher an putting all together.

Last note, near to forget it, P-space has a 'dark side', brainwashing.
You cannot access your opponent p-space, but if you mage to force your opponent, with a vampire attack, to execute these lines of code, or

something similar:

```
bwash   spl     0,>1
        stp.ab  #0,#0
        jmp     -2,{-1
```

(usually this code is together with others spl and a core clear)

its p-space will soon fill of garbish and it's rather difficult that, in the following round, its switcher will found what it needs to make a correct decision. So, when you make your switcher, don't forget to think at what will happens if something goes wrong in your p-space, and, _most_important_, never forget to mod you STR value before executing the jump.

```
        mod     #2, 1
str     jmp     @0, paper       ;a field holds strategy
        dat     0,  juliet
```

If you forget it, may happens your warrior will have to execute something like

```
str     jmp     @1234,paper
```

and you will score something like 0/249/1 :-(

Here is the code. I submitted the warrior at both -94 and beginners hill, if you have any question, or you are interested in results, mail me <bezzi@iol.it>

```
------------------------------
;redcode-b quiet
;name juliet and paper
;author M R Bremer, B. Bezzi
;strategy p-warrior for C.W. n.5 hint
;strategy switches juliet storm and Paper01
;kill juliet and paper
;assert CORESIZE == 8000

ptr     EQU     -1333
dest0   equ     2200
dest1   equ     3740
dest2   equ     -1278
range   equ     933

        RES     equ 0           ;here pmars loads results
        STR     equ 1           ;here I store my strategy

imp_sz  equ     2667

org     start

gate    dat <-445, <-446
s       spl #445, <-445
        spl #0, <-446
        mov {445-1, -445+2
        add -3, -1
        djn.f -2, <-2667-500
        mov 32, <-20
go      dat     #0,     #ptr
juliet  mov     {-1,    <-1
        mov     {-2,    <-2
        mov     {-3,    <-3
        mov     {-4,    <-4
        mov     {-5,    <-5
        mov     {-6,    <-6
        mov     gate,   ptr+24
        mov     gate,   ptr+24
        spl     @go,    <4000
        jmp     boot,   <4013
start

res     ldp.ab  RES,    #0      ;load result last match
str     ldp.a   STR,    str1    ;load strategy in use
        sne.ab  #0,     res     ;check result, win or tie OK
```

```
lost    add.a   #1,     str1    ;lost change
        mod.a   #2,     str1    ;secure jump
win     stp.ab  str1,   _STR    ;save strategy
str1    jmp     @0,     juliet
        dat     0,      paper



paper
        spl     1,      <300    ;\
        spl     1,      <400    ;-> generate 8 consecutive processes
        spl     1,      <500

silk    spl     @0,     {dest0
        mov.i   }-1,    >-1
silk1   spl     @0,     <dest1  ;split to new copy
        mov.i   }-1,    >-1     ;copy self to new location
        mov.i   bomba,  }range
        mov     {silk1, <silk2
silk2   jmp     @0,     >dest2
bomba   dat     <2667,  <1

for MAXLENGTH-CURLINE-9
        dat     0,0
rof

boot    spl     1      ,#0
        spl     1      ,#0
        spl     <0     ,#vector+1
        djn.a   @vector,#0

imp     mov.i   #0,imp_sz

        jmp     imp+imp_sz*7,imp+imp_sz*6
        jmp     imp+imp_sz*5,imp+imp_sz*4
        jmp     imp+imp_sz*3,imp+imp_sz*2
vector  jmp     imp+imp_sz  ,imp

        end
```

Planar's corner:

Next week, you'll get the sequel to my article about imp spirals.
Today, I have a short hint for beginners and a call for volunteers.

The short hint:

I have written the following program:

```
        ;redcode
        foo     equ     1+2
                nop     foo*2
```

Giving it to pMARS, I get this load file:

```
        START  NOP.F  $     5, $     0
```

Hey ! What's going on here ? If foo is 1+2 and the argument to NOP is foo*2, then it must be 6, right ? Wrong. The argument is 1+2*2 = 5, because EQU does a textual replacement of the label with its argument, not a numerical evaluation of its argument (there is a good reason for this).

The solution is the same as in the C language: use parentheses generously:

```
        foo     equ     (1+2)
```

Now the argument to NOP is (1+2)*2 and I'm happy. Maybe this was the reason why my warrior failed on the hill. But then again, maybe not.

The call for volunteers:

I have started updating the ICWS'94 draft standard. My new version includes the new addressing modes and opcodes we are all using every day. Who will add p-space into it ? We have missed the '94 deadline by a long time now, and I think it's time to turn the draft into a standard (does the ICWS still exist, by the way ?) Or at least the draft should describe the language that we are using.

Before we start discussing read/write limits, Stefan asked that I post a summary of all the good arguments against them, so I have to find the postings of one and a half year ago (does anybody know where to find an archive of recent postings to r.g.cw ?) Please no flame war before I declare the season open.

The new version of the ICWS'94 draft standard is available at
http://pauillac.inria.fr/~doligez/corewar/icws94.95

<Damien.Doligez@inria.fr>

Extra Extra:
Thermite

With impeccable timing I re-arrive on the internet just as Thermite is knocked off the hill, appropriately enough by Michael Constant. I don't think I ever published the code, so here it is with explanations as accurate as the mists of time permit. I'm not really sure why it worked for so long, but I guess it was a lack of decisive weaknesses rather than any single strength. Maybe P-space even helped, making targets bigger...

Thermite was standard quickscan, followed by a Torch-like incendiary bomber. The quickscan was originally developed from Michael Constant's Sauron (94 tournament) and ended up almost exactly like his Pyramid. I experimented with various warriors to follow the scanning stage: a vampire, Midge, sadly couldn't bite as fast as Silks could grow, and Queasy (4-word Mod-1 MOV <A,B bomber) was good against scanners but otherwise weak.

Then Paul Kline published Torch which looked too good not to steal: fast incendiary bombing, and multi-process -- so tending to draw if damaged. The only constructive change I made removed the anti-scanner gaps from the code, to make it more resistant to Silk-type strip bombing. I called the result Phosphorus and it was modestly effective (somewhat less so than Torch :) but it proved a great partner to the quickscan.

The Phosphorus code may appear puzzling if you don't know Torch. The key idea, is that the instructions in the loop are executed in _reverse_ order because the SPL #0 instruction keeps feeding new processes into the loop.

:(No-one "hides" near large decoys any more... :(

```
;redcode-94
;name Thermite 1.0
;kill Phosphorus
;author Robert Macrae
;strategy  Quick-scan -> incendiary bomber.
;assert CORESIZE == 8000

; Since I don't launch phosphorus, vulnerable to carpet bombers. May
; pay to put it at start? I should make better use of DJN stream
; (nascent). Either use <, or else start it somewhere which gets bombed
; by mov fairly quickly. What happens if I fall through early, due to
; DAT 1,1s? Should check this doesn't hurt...

SPC     equ 7700            ; (CORESIZE-MAXLENGTH-MINDISTANCE*2)
STP1     equ 81              ; (SPC / (RAM/2) / 2)
```

```
Lookat  equ look+237+8*(qscan-1)*STP1

; First scan at 237; last at -67?

traptr  dat     #0,     #trap
bite    jmp     @traptr,    0         ; Vampire bite.

; Lots of pointers to these, so keep them away from trap!

look
qscan   for     6
        sne.i   Lookat+0*STP1, Lookat+2*STP1
        seq.i   Lookat+4*STP1, Lookat+6*STP1
        mov.ab  #Lookat-bite-2*STP1, @bite
        rof

        jmn     test+1, bite   ; Save a few cycles

qscan   for     6
        sne.i   Lookat+48*STP1, Lookat+50*STP1
        seq.i   Lookat+52*STP1, Lookat+54*STP1
        mov.ab  #Lookat-bite+46*STP1, @bite
        rof

        jmn     test+1, bite   ; Save a few cycles

qscan   for     6
        sne.i   Lookat+1*STP1, Lookat+3*STP1
        seq.i   Lookat+5*STP1, Lookat+7*STP1
        mov.ab  #Lookat-bite-STP1, @bite
        rof

        jmn     test+1, bite   ; Save a few cycles

qscan   for     6   ; Should be 7 if I had space...
        sne.i   Lookat+49*STP1, Lookat+51*STP1
        seq.i   Lookat+53*STP1, Lookat+55*STP1
        mov.ab  #Lookat-bite+47*STP1, @bite
        rof

; Intention is to place points evenly through the target area.

test    jmz.b   blind,  bite            ; if no address stored, no hit.
        add     #STP1*2, bite            ; Smaller than pyramid, as fast.
        jmz.f   -1,     @bite           ; find nonzero element.

        mov     spb,    @bite           ; Quick pre-bomb...

        add     #49,    bite            ; aim 51 past the hit
attack  sub.ba  bite,   bite            ; bite(b) contains target-bite
loop    mov     bite,   @bite           ; (a) contains the bite addr.
        add.f   step,   bite
        djn     loop,   #24             ; 6 spacing => 72 cycles...


; Incendiary bomber based on Phosphorus 1.0 (from Torch).

bstp    equ     155       ; Mod 5, as too big for mod 4 to miss!
gap     equ     15        ; Gap between mov and spl.
offset  equ     130       ; Chosen with step and gap to give long bombing run.
count   equ     1500

blind
spb     spl     #0,         <-gap+1    ; spl half of the incendiary
        add     #bstp,      1
        mov     spb,        @tgt-offset ; Gives longest run, given gap & step.
        mov     mvb,        @-1
tgt     djn.f   -3,         >300       ; gets bombed with spl to start clear
        mov     ccb,        >spb-1     ; Uses copied mvb for CC.
        djn.f   -1,         <spb-18    ; Aids clear.
```



```
mvb     mov     gap,        >gap       ; mov half of the incendiary
ccb     dat     0,          10         ; Core Clear.

; Bit worried about having trap so close to my code...

trap    spl     0,      >-200          ; Lackadaisical attempt at gates.
        spl     -1,     >-200+2667     ; Each increments many times between
        jmp     -2,     >-200+2*2667   ; imp steps, but then the whole imp
                                       ; moves! I only blow away rings...

step    dat     #6,     #-6            ; QS step size. Up from 5 for speed.

        end     look
```

(Editor note: I had to change it a little, because Robert used STP as a label, and now it's not allowed being it an opcode. Just hope I didn't introduce bugs; I tested it and all seems OK. - B.B.)

Extra Extra Extra:
Phq

Well! I have received lots of requests about Phq so I have decided to publish it... (in other words CoreWarrior' staff has payed me enough ! ;-)
Phq is one of my first programs written in 94 standard (I have started redcoding with the 86 standard...)
First of all two words about the name...
Phq recalls a formula of quantum mech: the original name had been Emc2 (reference to 2c initial qscan) but Emc2 was also the name of another my QScan-->Stone warrior that never worked very fine...
When I began making Phq, another program behaved very well:
Marcia Trionfale of our friend Beppe Bezzi; so I decided that my warrior should contain a paper module!
At those times most of the programs on the hill reached the limit of 100 istructions, this made me choose for the initial QScan pass.
The initial QScan is also very useful to solve (without any PSpace routines) the problem of papers in gaining points during self fighting:
QScan makes Phq able to kill itself in self fighting about 83% times.
So I took the decision: my program will have to be a Qscan --> Paper !
Well! At this point I had to decide what could I do if QScan found an enemy, or simple a decoy :(!
The first attempt was to bomb the neighborhood of the cell differing from dat.f 0,0 (blank) with simple dat bombs, using a series of "mov bomb,<ptr" istructions.
The problem is finding the size of the neighborhood.
I had obtained just slight better results using incendiary bombing.
I'd have liked to use spl 0 bombs but this wasn't possible for after bombing I had to start with paper, and a paper isn't the best to kill lots of processes executing a "spl 0 jmp -1" loop (or similar)!
Maybe I could have tried some vamp bombing, but I didn't do this...
I choosed to copy some suicide 3-instr core-clear routines "around", hoping that the enemy executes one of them!
This solution was quite good: if even only one enemy task executes this 3-instr code it may kill other eventually (near) enemy tasks; note that its bombing is harmless against my paper!
An interesting question is: ...and what about against Pspace programs?
I'm mainly a paper and if the enemy switches on a scanner module, it's a slaughter for me! :(
So I'd liked to have some brainwashing routines: I added to my core-clear a brainwashing stp line (who fills enemy PSpace with non-sense values).
Well! This program worked quite fine but another little change made it even better!
After bombing my initial strategy was simple to jump to paper routines...
...and what about using a spl to have in any case a task who executes my core-clear?
Well! I added this spl line and results were 3 points more then previous version!
I'm not sure why this works better...
maybe because I bomb more wide around the non-blank cell and sometimes I reach the enemy before the paper itself kills my own task.

Note that I’m brainwashing myself! (as you can read in the initial strategy line;-), but this is unrelevant ’cos I don’t use PSpace in "active mode", meaning with "active mode" that I don’t use PSpace for switching.
That’s all!
What?
The program?
Yeah! Of course! Here you are!
(Hoping that Phq will reach at least Thermite in old scored programs ;-)
Note that it’s quite full of bugs ;-)
I leave to find them out, to my "25 readers" as exercise (as my prof. of geometry always said...)

For any questions, flaming etc. your mail is welcome!
Mail to pan0178.iperbole.bologna.it

```
;redcode-94
;author Maurizio Vittuari
;name Phq
;assert CORESIZE==8000
;strategy New version! This one likes brainwashes...

step1   equ     3743    ; unoptimized replicator costants
step2   equ     1567    ; see CoreWarrior issue 3
step3   equ     1349


;       ****** QSCAN ROUTINES ******

start
s1 for 4
        sne start+400*s1,start+400*s1+100
        seq start+400*s1+200,start+400*s1+300
        mov #start+400*s1-found,found
rof
        jmn which,found
s2 for 4
        sne start+400*(4+s2),start+400*(4+s2)+100
        seq start+400*(4+s2)+200,start+400*(4+s2)+300
        mov #start+400*(4+s2)-found,found
rof
        jmn which,found
s3 for 4
        sne start+400*(s3+8),start+400*(s3+8)+100
        seq start+400*(s3+8)+200,start+400*(s3+8)+300
        mov #start+400*(s3+8)-found,found
rof
        jmn which,found
s4 for 4
        sne start+400*(s4+12),start+400*(s4+12)+100
        seq start+400*(s4+12)+200,start+400*(s4+12)+300
        mov #start+400*(s4+12)-found,found
rof

; just missed a line... :(

s5 for 3
        sne start+400*(s5+16),start+400*(s5+16)+100
        seq start+400*(s5+16)+200,start+400*(s5+16)+300
        mov #start+400*(s5+16)-found-100,found
rof

found   jmz rabbit,#0
        add #100,-1
which   jmz -1,@found
        add #10,found
for 4                           ; bombing enemy
        mov m3,<found
        mov m2,<found
        mov mp,<found
```



```
        mov m1,<found
rof
        spl @found,{100         ; So... why not ?

;       ****** REPLICATOR ******

rabbit  spl     1,      <200    ;create 11 processes
        mov     -1,     0
        spl     1,      <300
        mov     -1,     0
s1      spl     step1,  #0
        mov.i   >-1,    }-1
        mov.i   bomb,   }1942
s2      spl     step2,  #0
        mov.i   >-1,    }-1
        mov.i   bomb,   }1842   ;I’ve changed > with } so many times
        mov.i   bomb,   >1900   ;that I can’t remember if this version
        mov.i   bomb,   }2000   ;is the one actually on the hill...
        mov.i   <s2,    <s3
s3      jmp     @0,     }step3
bomb    dat     <2667,  <5334   ;anti-imp bomb

;       ****** BRAINWASHING CORE-CLEAR ******

m1      mov     m3,     {m3
mp      stp     <0,     #20     ; brainwashing instruction
m2      djn.f   m1,     }m3+1
m3      dat     }bomb,  <2667
end start
```

For questions and congratulations mail me <bezzi@iol.it>, flame Myer <bremermr@ecn.purdue.edu> or if you think it’s of general interest post to rec.games.corewar

Anyone with hints or warriors to publish is welcome.

The hint
How to improve your beginner’s warrior.

This week hint is again an how to improve a warrior; having received no warrior I was able to improve :-), I toke a warrior of a six months ago beginner, good in the -b hill but unable to enter 94: Provascan 2.0 by ... me :-)

Here is Provascan code, ’prova’ in italian means test, a tweaking of XTC a very successful warrior of a few years ago, and a classic sample of beginner’s coding (Provascan not XTC :-)

```
;redcode-94
;name Provascan 2.0
;author Beppe Bezzi
;strategy B-scanner
;strategy a six months ago beginner’s warrior :-)
;kill Provascan
;assert CORESIZE == 8000
;
step    equ #3364
loop    add.ab step,    ptr ;scanner modulo 4
ptr     jmz loop,       trap
        mov ptr,        dest
cnt     mov #17,        cnt     ;0
kill    mov @trap,      <dest
```

```
        djn kill,       cnt
        jmn loop,       trap
        jmp cocl                ;0
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0                  ;0
        dat 0,0
dest    dat 0,0
        dat 0,0
        dat 0,0                  ;0
        dat 0,0
        dat 0,0
        dat 0,0
trap    dat #1                  ;0
bomb    spl 0
        dat 0,0
        dat 0,0
        dat 0,0                  ;0
        dat 0,0
        dat 0,0
gate    dat 0,0
        dat 0,0                  ;0
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0                  ;0
cocl    sub #15,        cont
        mov cocl-4,     <cocl-4
        djn -1,         cont
cont    spl 1 ,<0               ;0
        spl 0,<gate
        mov mark,<cocl-1
        jmp -1,<gate
        dat 0,0                 ;0
mark    dat <-11, <-11
void    for 35
        dat 0,0
        rof
esca    for 4
        dat 0,2
        dat 0,2
        dat 0,2
        dat 0,0
        rof
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0
        dat 0,0
end loop
```

lines with ;0 must have a zero b-field to avoid self bombing.

It’s a classic b-scanner that covers nonzero locations with a wide carpet of spl 0,0 until it covers the label trap; at such point it begins a coreclear with spl and then dats, ugly indeed, but I was a beginner :-)

I resubmitted it to 94 hill and I scored a nice 109, just what I needed to start with. The idea of a carpet bombing b-scanner isn’t that bad, the implementation is really nasty, we can cut it _a lot_ and have it make the same work, 17 is a too big number for the carpet, a smaller one, 7, is sure better, so we are not delayed too much by decoys. Another improvement is using the post increment for bombing, better with imps and allowing us another trick we’ll speak of later, and adding a forward running perpetual clear;
The new code:


```
name author blah blah ...
;
step    equ     #3364

trap    dat     0,      1       ;0
        dat     0,      0
dest    dat     0,      0
        dat     0,      0
        dat     0,      0       ;0

;the use of postincrement allow us to put dest before our code

loop    add.ab  step,   ptr
ptr     jmz     loop,   trap
        mov.b   ptr,    dest
cnt     mov     #7,     0       ;0
kill    mov     bomb,   >dest
        djn     kill,   cnt
        jmn     loop,   trap
bomb    spl     #0,     0       ;0
        mov     2,      >ptr
        djn.f   -1,     {ptr
        dat     -5,     #15
end loop
```

Much better, isn’t it, now we are but 10 lines long plus two dats nonzero. We have not to worry of self bombing when we bomb dest to end scanning, because moving a spl 0,0 at a cell addressed with > is uneventful, the cell is overwritten with zero after the increment, because of ’in register’ evaluation.

With a bit more confidence I resubmitted it to pizza 94 to score a 121, better but not enough, something is still going wrong.
We lose a lot from Frontwards, Porch Swing and others once through scanners, and we cannot stop Impfinity and Night Train’s imps. To solve the first problem the solution is simple, boot away and leave a decoy behind; for second problem the solution is more subtle. Let’s give a look at our clear: we are using ’ptr’ as clear pointer, when imps get incremented and attacked, they stop being imps but begin executing our code, so we cannot kill them; to do so we must have a dat after the clear pointer; we can use the line ’dest’, it will be split covered but it’s goo for our job.

New version:

```
;redcode-94
;name Provascan 2.0d
;author Beppe Bezzi
;strategy B-scanner
;strategy a six months ago beginner’s warrior :-)
;strategy trying to improve it for the hint
;kill Provascan
;assert CORESIZE == 8000
;
step    equ     #3364
away    equ     3198

trap    dat     0,      0       ;0
        dat     0,      0       ;we can use equs for those dat 0,0 they are left
dest    dat     0,      0       ;for clarity
        dat     0,      0
        dat     0,      0       ;0
loop    add.ab  step,   ptr     ;
ptr     jmz     loop,   trap
        mov.b   ptr,    dest
cnt     mov     #7,     0       ;0
clear   mov     bomb,   >dest
        djn     clear,  cnt
        jmn     loop,   trap
bomb    spl     #0,     0       ;0
        mov     2,      >dest
```

```
        djn.f   -1,     {dest
kill    dat     -5,     #kill-dest+2
        dat     0,      0       ;0

boot    mov     kill,   away
for 10
        mov     {boot,  <boot           ;the faster way to boot away
rof
        mov     #0,     boot+3          ;we have to set those b-fields to zero
        mov     #0,     boot+7          ;to save time later
        mov     #0,     boot+11
jump    jmp     @boot,  >away-29        ;> is to set trap b-field non zero

a for (MAXLENGTH-CURLINE)/4
        dat     jump,   0               ;this decoy doesn't have two equal cells
        dat     bomb,   boot            ;and still has all fourth b-field at zero
        dat     boot,   kill
        dat     clear,  boot
rof
end boot
```

Results are good now: 136.5 and 11th place in 94 hill, yuppee :-)

We beat Frontwards and La Bomba, we tie Impfinity, Porch Swing2, and Torch and we score an high nuber of wins that boost our score, it's better losing 45/55/0, as we did against Derision, than scoring 100 ties. We are still losing bad against quiz, solving this problem is left as an exercise for the reader :-)
BTW it's my best ever result with a scanner, were it not the hint test I'm not sure I had published it. ;-)

Detailed 94 scores are available on request (mail me), I haven't tested the warrior against beginners hill, feel free to do it and make public domain the results.

Next hint will be made by Maurizio <pan0178@comune.bologna.it>, mail him with your requests.

Extra Extra:
La Bomba
by Beppe Bezzi

La Bomba is the first program allowing me to become King of the Hill and to remain in such position for some time, Jack in the box was King for but few challenges. The reasons of its success was its very high speed together with the very favourable environment it found; looking at it now I can see some ways to improve it, like using the faster decoding of Pyramid, but now it's no time for La Bomba 2.

La Bomba is a qscan followed by a simple replicator, the same of the hint of CW #3, the innovative part is the Tornado bombing engine used to drop a cluster of dat bombs on the opponent, in the hope of catching it during boot; this proved very effective against p-warriors and stationary warriors using a decoy.

```
;redcode-94
;name La Bomba
;author Beppe Bezzi
;assert CORESIZE == 8000
;kill La Bomba

org     start
qstep   equ     5
qrounds equ     7
bigst   equ     99

qst     equ     qstart -(4*bigst)
qstart  equ     start+145
```



```
dest0   equ     2200
dest1   equ     3740
dest2   equ     -1278
range   equ     933


start
s1 for 5
        sne.x   qst+4*bigst*s1, qst+4*bigst*s1+bigst*1  ;check two locations
        seq.x   qst+4*bigst*s1+bigst*2, qst+4*bigst*s1+bigst*3
        mov.ab  #qst+4*bigst*s1-found, found  ;they differ so set pointer
    rof
        jmn  which,   found
s2 for 5
        sne.x   qst+4*bigst*(s2+5), qst+4*bigst*(s2+5)+bigst*1
        seq.x   qst+4*bigst*(s2+5)+bigst*2, qst+4*bigst*(s2+5)+bigst*3
        mov.ab  #qst+4*bigst*(s2+5)-found, found
    rof
        jmn  which,   found
s3 for 5
        sne.x   qst+4*bigst*(s3+10), qst+4*bigst*(s3+10)+bigst*1
        seq.x   qst+4*bigst*(s3+10)+bigst*2, qst+4*bigst*(s3+10)+bigst*3
        mov.ab  #qst+4*bigst*(s3+10)-found, found
    rof
        jmn.b   which,  found
s4 for 5
        sne.x   qst+4*bigst*(s4+15), qst+4*bigst*(s4+15)+bigst*1
        seq.x   qst+4*bigst*(s4+15)+bigst*2, qst+4*bigst*(s4+15)+bigst*3
        mov.ab  #qst+4*bigst*(s4+15)-found, found
    rof
        jmn.b   which,  found

found   jmz.b   warr,   #0      ;skip attack if qscan found nothing
        add     #bigst, -1      ;increment pointer till we get the
which   jmz.f   -1,     @found  ;right place
qattack                         ;found.b points target
        mov     bomba,  @found  ;in case small and dangerous
for 0
        After decoding enemy position it checks the location found+32 and, if it
        proves not to be empty, shifts 30 cells the bombing zone to the right
        This added near 5 points to my score
rof

        add.ba  found,  qstone
        add.b   found,  qstone
        seq     *qstone,-100
        add.f   shift,  qstone

qst1    mov     qbomb,   *qstone        ;Tornado bombing engine the faster way
        mov     qbomb,   @qstone        ;to fill your enemy with hot lead
qstone  mov     32,     *32-qstep
        sub.f   qincr,   qstone
        djn.b   qst1,   #qrounds

warr                                    ;Paper01 the hint warrior
paper   spl     1,      <300
        spl     1,      <400
        spl     1,      <500
silk    spl     @0,     {dest0
        mov.i   }-1,    >-1
silk1   spl     @0,     <dest1
        mov.i   }-1,    >-1
        mov.i   bomba,  }range
        mov     {silk1, <silk2
silk2   jmp     @0,     >dest2
bomba   dat     <2667,  <1

qbomb   dat     #-qstep, #-qstep
qincr   dat     #3*qstep,#3*qstep
shift   dat     #30,    #30
```

```
for MAXLENGTH-CURLINE-9
        dat     0,0
rof

for 9
        dat     1,1
rof
end
```

Extra Extra:
Armory - A5
by J.K. Wilkinson

Ok, Armory was my first succesful Hill challenge.
It’s based on a simple idea: beat scanners/scissors with a stone,
beat stones with an imp-stone, and beat papers with scissors.

The only thing really new/interesting is the boot method and
pspace. All the components are highly standard and well-known
warriors.

In order to squeeze all this code in, (and do it fast, I wanted
to make a splash on the Hill!) I had to throw together some stuff
like this:

```
sboot:   mov.a  #cgate-2-tboot, tboot
         mov.a  #T+18-goboot, goboot
         jmp    2

tboot:   dat    gate,   T
tornado: mov    }tboot, >tboot
         djn    -1,     #8
```

That "jmp 2", for instance, is 100% pointless. :)
100%! I mean, I’ve seen code where I thought something _could_
be trimmed down... but placing your own dats so your jumping
over them???
As you can see, there’s much room for optimizing, but when A5
hit the big time, I decided to leave well enough alone. :)

As for the pspace, it’s perhaps the smartest one that’s been
attempted on the Hill. You can’t be too smart, or you get you
ass kicked while your trying to decide what to do. Heh.
I think I struck I happy medium with my system.

Basically, if I’ve just been brainwashed, I reinitialize
pspace and just keeping going with what ever is in my NUM_STR
(my strategy pspace.) That means if I’m washed with 0 I go
to scissors when I lose. Is that a good idea?

Well, think about this: If I’m brainwashed and I don’t lose
I go to tornado, because on a loss I add 1 to the strategy.
This means that q-brainwash->papers can’t really lock on.
I’ve still got a shot do bounce out of the "just lost, now
you’re brainwashed->your screwed" cycle. :)
In retrospect, a better system might be just the opposite,
but it’s difficult to predict your opponent’s methodology.

It seemed to work fairly well though, until I went and killed
it. :(<g>

Oh, and if you’re wondering what the "Major changes" were,
I removed a paper module. The paper just couldn’t launch
in enough time, from that much space... so I redesigned all
the boots. You’ll notice they aren’t in-line boots (they
use djn.b to loop the boot.)

Here’s Armory...


```
----------c-u-t---o-n---t-h-e---d-o-t-t-e-d---l-i-n-e--:-)----------
;redcode-94
;name Armory - A5
;kill Armory - A4a
;author Wilkinson
;strategy use pspace to go to battle
;strategy v 5 - well, I’m still losing to Brain Wash... I may still
;strategy have a pcode bug. :/
;strategy Major changes.  Hoping for more wins, and less ties...
;assert 1

i       equ imp+100
NUM_STR equ #3
_RESULT equ #0
_LOSS   equ #222
_STR    equ #333
BOUND   equ #800
CDIST   equ 12
IVAL    equ 42
FIRST   equ scan+OFFSET+IVAL
OFFSET  equ (2*IVAL)
DJNOFF  equ -431
BOMBLEN equ CDIST+2
GATE    equ tie-4000
stinc   equ 190
d       equ 2667
S       equ stone+2537
T       equ gate+5500
step    equ 52
count   equ 665

res:  ldp _RESULT,  #0      ;load last result into B-field
loss: ldp   _LOSS,  #0
      jmz goloss,   res          ;a zero indicates a loss in the last round

      djn tie,res

win:     add    #-1,    loss
         stp.b  loss,   _LOSS
tie:
go:      ldp    _STR,   #0

         slt    BOUND-100, loss   ;check for illegal _LOSS record
         stp    BOUND,     _LOSS

         mod.ab NUM_STR,   go      ; in case _STR ever gets screwed up
         mov.ba go,      case
gojmp:   jmp    case               ;after this gojmp, we jump again from case

goloss:  add    #1,     loss
         slt    loss,   BOUND+2;if we’ve lost more than we won, then switch
         jmp    switch

         slt    BOUND-100, loss   ;check for illegal _LOSS record

         jmp    switch
         stp.b  loss,   _LOSS
         jmp    go

switch:  ldp    _STR,   #0
         add    #1,     -1
         slt    -2,     NUM_STR
         mov    #0,     switch
         stp    BOUND,  _LOSS
         stp.b  switch, _STR
         add.ba switch, case

case:
         jmp    @0,     tornado    ;3
```

```
         jmp    @0,     sboot      ;3
         jmp    @0,     stonespir  ;3

;***Cannonade
stone:   mov   <1+5+(stinc*800),1
         spl   -1,    <2
         add    3,    stone
         djn   -2,    <5141+1
         dat    0,    0
         mov   stinc, <-stinc

           dat stone,   S
stonespir: mov }-1,     >-1
           djn -1,      #6
           spl S+1

spir:      mov.i  imp,   i
           spl.a  1,     <GATE-200
           mov.i -1,     0          ;2
           spl.a  1,     <GATE-300  ;3
           spl.a  1,     <GATE-400  ;6
           spl    2                 ;12
           jmp.a @imp-1, {0
           jmp.a *imp-1, {0

           dat   #i+2*d+7, #i+1*d+7
           dat   #i+7,     #i+2*d+6
           dat   #i+1*d+6, #i+6
           dat   #i+2*d+5, #i+1*d+5
           dat   #i+5,     #i+2*d+4
           dat   #i+1*d+4, #i+4
           dat   #i+2*d+3, #i+1*d+3
           dat   #i+3,     #i+2*d+2
           dat   #i+1*d+2, #i+2
           dat   #i+2*d+1, #i+1*d+1
           dat   #i+1,     #i+2*d
           dat   #i+1*d,   #i
imp:       mov.i #1,    2667

sboot:   mov.a  #cgate-2-tboot, tboot
         mov.a  #T+18-goboot, goboot
         jmp    2

tboot:   dat    gate,   T
tornado: mov    }tboot, >tboot
         djn    -1,     #8
         add.ab #10,    tboot
         mov    }tboot, >tboot
         djn    -1,     #4
goboot:  jmp    T+1,    {0

;***Tornado
gate    dat     #step,       #-step           ;step equ 52
start   mov     bombd+10,    *tstone
        mov     bombd+10,    @tstone
tstone  mov     *(2*step)+1, *(3*step)+1
        add     incr+10,     tstone
jump    djn.b   start,       #count          ;count equ 665
        spl     #step,       #0
clr     mov     gate,        }gate-5 ;jump ;gate-3

; 10 "dat 0, 0"'s need to be inserted here

incr    dat     3*step, 3*step
bombd   dat     #52   , #1         ;hit dat
dat 0, 0
dat 0, 0

;***Scissors
         dat    #cgate-10, clear-cgate+8+10    ; just in case clr is decremented
```



```
cgate    dat    #4000,    3000
wipe4    dat    #4000,    clear-cgate+8+10
wipe3    dat    #4000,    clear-cgate+8+10
         spl    #6000,    clear-cgate+8+10   ; redundant wipers
wipe2    spl    #6000,    clear-cgate+8+10   ; redundant wipers
wipe1    spl    #3050,    clear-cgate+8+10

; 10 "dat 0, 0"'s need to be here
clear    spl    #0, >-20
         mov    @2, >cgate-10
         mov    @1, >cgate-10
         djn.b  -2, {wipe1-10
end
```

As you can see from the Armory ;strategy lines, I thought I was losing to brainwashes. It turns out that my scissors was so pitiful that any decent paper could thrash me, and I don't think Brainwash's brainwashing was really much of a factor. :/

Planar's corner

CDB tutorial, part 1

This is the first article of a long series that we will write, Stefan Strack and I. We will start at the level of "I haven't even read the docs yet" and we'll hopefully end up at the level of "why don't we have a CDB macro programming contest ?"

Because Stefan has written CDB itself, he probably wouldn't see much difference between a three-line macro and a one-character command, so I get to write this first article. We're writing this in the hope that it will be useful to you, so your feedback is vital: tell us what is missing, what is unclear, what you would like to see explained in more detail, etc.

After this introduction, we can start learning CDB. The hardest step is the first one: you must realize that CDB is at the same time extremely powerful and quite easy to use. I'll take myself as an example. I was reading the docs for the first time less than three months ago. Have a look at Core Warrior 8 to see what kind of macros I can write now. And I'm still far from Stefan's level.

So CDB is easy to use, and by learning it you can greatly speed up your warrior development: once you master CDB, you'll be able to do such things as:

+ explore the functioning of a warrior in a single-warrior "fight" and discover any unexpected problems (i.e. debug a program)
+ find the right set of constants for a stone to remove its suicidal tendencies
+ optimize a paper's constants to cover the most core locations in the smallest time
+ gather statistics on how much of a spiral is still alive at the end of a typical fight
+ automatically find the best constants for a warrior against a given "White warrior"
+ many other things that I haven't thought of yet: the only limit is your imagination.

Let's go on to the technical stuff. We will use the following program for the examples. Save it in a file named "fahr.red".

```
;redcode-94
;name Fahrenheit 0
;author Planar
```

```
;assert CORESIZE == 8000

steps   spl     #2044, <3039
ptr     mov.i   <100, <1000
attack2 mov.i   <-2000, *-1
        add.f   steps, ptr
        djn.f   -3, {attack2
        end
```

This is a fast stone with a strong suicidal tendency. We'll try to find a good set of constants to replace those 100 and 1000.

General description

CDB is a line-oriented debugger. CDB controls the pMARS simulator and you control CDB by typing commands to the (cdb) prompt. The best way to get this prompt is using the "-e" option to the pMARS command line. We will use the following command line. Type this at your shell, or (if you use a Macintosh) use the "command line" item in the "file" menu:

```
    pmars -e -b -r 100 fahr.red
```

This launches pMARS and immediately enters CDB. CDB displays the next instruction to execute (i.e. the first instruction of the program), and our good friend the (cdb) prompt. Now we get to decide what happens next simply by typing a command.

You can also get the prompt by typing 'd' (in the DOS version), or control-C (in the Unix versions) or command-. (in the Mac version) when pMARS is running. If you're looking at a battle and you see something strange happen, you can stop the battle and use CDB to investigate.

The best way to read this tutorial is to launch your pMARS on your own computer and try the commands when you read their description. I was going to show the example commands and CDB's answers in the following format:

```
00000   SPL.B  #  2044, <  3039
(cdb) echo coucou
coucou
(cdb)
```

CDB displays the first line when it is triggered by the "-e" option on the command line. If you enter CDB with control-C, you'll get a different display. The (cdb)'s are the prompts, "echo coucou" is what I typed, and "coucou" is CDB's answer. After printing "coucou", CDB is ready for a new command.

Because this tutorial is mostly useless if you're not trying the commands on your pMARS, and because Beppe won't be happy if Planar's corner gets bigger than the rest of Core Warrior, I'm not going to show CDB's answers to my example commands. Try them and see. If you can't view this file and use pMARS at the same time, print this file. If you don't have a printer, contact me and I'll send you a paper copy.

The first thing to learn is of course how to get out of CDB. Here is one command to do that:

```
(cdb) quit
```

After this command, CDB is not ready for a new command because it has exited, along with pMARS itself. Here is the other command to exit CDB:

```
(cdb) continue
```


With this command, CDB exits and lets pMARS run as if CDB had not been invoked, until the end of the 100 rounds we specified on the command line. If you type control-C again, you get back in control.

Displaying the current state

The most useful command of CDB is "list". It takes as argument a core address or a range of addresses, and it displays the contents of the core at that address or range of addresses. For example, you can get a listing of Fahrenheit by typing:

```
(cdb) list 0,4
```

If you want to see the next core cell:

```
(cdb) list 5
```

This cell contains "DAT 0, 0", which CDB displays as blank, because it is the default cell contents at the beginning of a battle.

You can abbreviate "list" to "l", or even nothing at all, like this:

```
(cdb) 1,3
```

Instead of numbers, you can use expressions as arguments to "list" (this is true for all cdb commands that take numeric arguments). I won't describe in detail the syntax of expressions, they are the same as what you use in redcode programs. There are a few special values that you can use in CDB expressions.

The most important of these special values is the dot address; it is entered as a period, and it represents the address of the last core cell displayed by CDB. For example, after the last example above, you can type:

```
(cdb) .
```

And to see what the B-field of this instruction points to:

```
(cdb) .-2
```

The variables A and B contain the A-field and B-field of the instruction at the dot address. And as a further shortcut, CDB accepts "+expr" and "-expr" for ".+expr" and ".-expr", so the above example could be entered as:

```
(cdb) 3
(cdb) +b
```

You can also type:

```
(cdb) +1,+10
```

Then press <Enter> at the CDB prompt. When you enter a null command like this, CDB will repeat the previous command, so you can list the whole core 10 cells at a time by simply pressing <Enter> repeatedly.

That's all for the "list" command. It took a long explanation because it is the most often used. With some graphical interfaces, you can also activate it by clicking in the core with the mouse.

To get a general view of what's going on, type:

```
(cdb) registers
```

This will display a the number of the current round, the number of cycles remaining before the end of this round, and a listing of each

warriors with its name, its number of processes, and a summary of its process queue and p-space.

The process queue is a list of core addresses. The process that will execute next is in [brackets], and the following process is pointed by the arrow.

Running the program

To run the program until completion, just type:

```
(cdb) go
```

pMARS will run until the program dies or the cycles have run out. While the program is running, you can still press control-C to get the prompt back. To know which condition ended the "go", use "registers".

To execute one instruction and get back to the (cdb) prompt, type:

```
(cdb) step
```

To step again, just press <Enter>.

Executing one instruction at a time is useful, but you'll often want to go forward in time at a faster pace. To this end, CDB provides the "skip" command, which takes an argument. Just type

```
(cdb) skip 999
```

The simulator will execute 1000 steps before giving back the (cdb) prompt. Use "registers" to check the number of cycles left: 1000 fewer than before you used "skip". Note that "skip 0" is equivalent to "step".

With "skip" commands, we can already zero in on the most interesting parts of a fight. For example, let us find out when Fahrenheit bombs itself. Use "go" to finish the current round, then "step" to get to the first cycle of a new round. Now type

```
(cdb) skip 999
(cdb) 0,4
```

You'll see that Fahrenheit is still intact after 1000 cycles. Type that sequence a second and a third time. Fahrenheit has bombed itself between the 4000th and the 5000th cycle. Use "go" to skip the rest of this round, skip to cycle 4000, and then 100 by 100 until the self-bombing. Then 10 by 10, then just step until the self-bombing. If you lose track of how many cycles have elapsed, "registers" will tell you.

This is a bit tedious, and CDB provides much better ways for doing this, but I have to keep something for next time, or I'll lose all my readership. We already have a useful piece of information: after 4136 cycles, Fahrenheit destroys its own SPL instruction.

Use "go" to reach the end of the battle, step once to start a new battle, and "skip 4135". Type "registers" and look at the process queue. Between the brackets is the address of the instruction about to execute: 1. type "1" and you'll see how this instruction bombs the SPL. (Note that the instruction at 1 was already displayed by CDB right after your "skip".)

Exercise: change the constant 1000 in Fahrenheit to avoid self-bombing at cycle 4136. Find out when the self-bombing occurs with your new constant, and which instruction of Fahrenheit is bombed.

There is one more way to do big steps. You can step all processes once by typing:


```
(cdb) thread
```

CDB will step once for every process in the process queue, and you'll get right back at your current process, one step later. This is very useful when you're debugging multi-threaded programs (which is not the case of Fahrenheit, but you can try it on the bootstrap sequence of Impfinity v3i, for example).

To conclude this first part, here is the great debugger classic, breakpoints. Type:

```
(cdb) trace 0
```

CDB will place a breakpoint on the SPL instruction of Fahrenheit. Every time a traced instruction is about to be executed, CDB will stop and display the instruction. If you repeatedly enter

```
(cdb) go
(cdb) registers
```

you'll notice how often the SPL injects a process into the loop. You'll also notice that the listing of the traced instruction has a "T" at its right. To remove a breakpoint, use "untrace".

For your homework, look up the "moveable" command in the docs, and think up a use for moveable breakpoints, and one for non-moveable breakpoints.

Abbreviations

CDB provides a good way of saving your fingers: you may abbreviate a command to its first few letters. Here is a complete list of the shortest abbreviations accepted by CDB. Each command is listed with its optional part in (parentheses).

c(ontinue)	g(o)	r(egisters)
ca(lc)	h(elp)	res(et)
cl(ear)	if	s(tep)
cl(s)	l(ist)	se(arch)
clo(se)	m(acro)	sk(ip)
d(isplay)	mo(veable) of(f)	sw(itch)
d(isplay) c(lear)	mo(veable) on	t(race)
d(isplay) o(ff)	p(rogress)	th(read)
d(isplay) on	pq(ueue)	u(ntrace)
e(dit)	pq(ueue) of(f)	w(rite)
ec(ho)	ps(pace)	wq(ueue)
ex(ecute)	ps(pace) of(f)	wq(ueue) of(f)
f(ill)	q(uit)	

Next time, we'll yield much more power with ~! !!@& and the commands for changing the core.

-- Planar <Damien.Doligez@inria.fr>

Many thanks to Stefan for the very useful remarks he made on a draft of this text.

The hint

The very basics of Core War game strategy.

(How to improve your beginner's warrior)

In this issue we shall try to discuss about improving warriors.
First of all a little preface for the beginner: we want to clarify the meaning of the sentence "a warrior better than another one".
At the moment, as you know, our "work" is to try to make our warriors enough powerful so that they can enter in a 25 warrior populed heap, that we usually call hill ;-).
One of the first things that a novice (apart the redcode language ;-) should

know about this game, is that for each given warrior, there exists a warrior that can defeat it in the long distance.
Maybe one of the main differences between a beginner's and a veteran's program is the range of enemy types it can succesfully face: this is often the reason why a warrior going very well in the beginner hill, often finds many problems if subscribed on the 94 hill.
Let's see some examples...
In the last number Beppe showed us how to improve a beginner's scanner, and he put it into the 94-hill (Provascan 2.0); then he modified again the warrior and reached the second place (Provascan 3.0)!
I tried to make some changes to version 2.0, and even if I couldn't score as high as Beppe (Provascan 3.0 is in the top ten, and it can be considered, as his previous version, no more a beginner's warrior!), I think it's an useful challenge for the beginner trying to improve it (not such an easy exercise!).
Now we discuss some basic and reasonable moves that can be attempted to modify it succesfully.
I remind you that Provascan is a 50%c B-scanner with boot and relative decoy, and spl0-dat core clear (for further details see Core Warrior 9); it plays conventionally the role of scissors, in the well known scissors-paper(replicators)-stone(bombers) analogy.

We start analizying some interesting results of Provascan 2.0:

```
Provascan 2.0d vs Frontwards    95/72/33        (one shot scanner)
Noboot vs Frontwards            55/129/16

Provascan 2.0d vs La Bomba      87/70/43        (QScan--> Paper)
Noboot vs La Bomba              97/76/27

Provascan 2.0d vs Impfinity4g1  93/93/14        (Imp stuffs! ;-)
Noboot vs Impfinity v4g1        90/88/22

Provascan 2.0d vs Torch t18     85/88/27        (incendiary bomber)
Noboot vs Torch t18             91/77/32

Provascan 2.0d vs Porch Swing 2 81/80/39        (one shot scan/bomber)
Noboot vs Porch Swing 2         59/111/30
```

Question: but booting is truly useful ?
Let's see the result!
Noboot (the same version but without boot) is faster, because doesn't have to spend some initial cycles executing the copy, and can immediately start scanning.
It's reasonable to think that Noboot will gain something against warriors not scanning the core.
If we neglect fluctuations, we see that Noboot works quite better against Torch (a bomber) and also against La Bomba (maybe Noboot can reach and hit, with a spl0 carpet, the paper before it spreads away.
Against Impfinity we can't see any obvious difference.
But take a look of what happens with one-shot scanners!
Against these guys the boot is highly recommended, as you can see from the scores: they stop scanning and start to fire from your decoy (with the core clear), while you are safe in another part of the core, scanning and ready to bomb them later.
Overally we understand that isn't a very good idea to take away the booting section (at least in the current hill) from our 50%c scanner!

Now, extablished that booting is a quite pretty idea ;-), we want to try another stuff: who of you, did never hate (maybe even for a while) those little, ugly and hard-to-die imps ;-) ?
(answer: maybe only our dad_of_imp_Planar never hated his little and *pretty* "creatures"...;-)
So what about on installing an anti-imp core clear on our favourite scanner ?
Let's take a look at this version of ProvaScan (I left the original comments of Beppe and changed just the clear and very very other little things):
BTW Prova_and_Riprova means Try_and_Try_Again!

```
;redcode-94
;name Prova_e_Riprova
;author Maurizio Vittuari
```



```
;strategy B-scanner
;strategy This is a personal defiance to Beppe ;-)
;strategy hoping to improve Provascan for the hint
;strategy on the New Year's Day issue
;assert CORESIZE == 8000

step    equ     3364
away    equ     3198+1

trap    dat     0,      0       ;0
        dat     0,      0    ;we can use equs for those dat 0,0 they are left
dest    dat     0,      0    ;for clarity
        dat     0,      0
        dat     0,      0       ;0
loop    add     #step,  ptr
ptr     jmz     loop,   trap+step
        mov.b   ptr,    dest
cnt     mov     #7,     0       ;0
clear   mov     bomb,   >dest
        djn     clear,  cnt
        jmn     loop,   trap
bomb    spl     #6,     0       ;0
        mov     kill,   }bomb   ;1st pass: spl0-dat carpet
        mov     kill-1, }bomb   ;and then 2nd pass: only dat <2667,<-2666
        jmp     -2
        spl     #4,     0       ;0
kill    dat     <2667,  <-2666

boot    mov     kill,   away
for 12
        mov     {boot,  <boot        ;the faster way to boot away
rof
        mov     #0,     boot+2       ;we have to set those b-fields to zero
        mov     #0,     boot+6       ;to save time later
        mov     #0,     boot+10
        mov     #0,     boot+14
jump    jmp     boot+away-11,>away-34   ;> is to set trap b-field non zero

for (MAXLENGTH-CURLINE)/4
        dat     jump,   0           ;this decoy doesn't have two equal cells
        dat     bomb,   boot        ;and also has all fourth b-field at zero
        dat     boot,   kill
        dat     clear,  boot
rof
end boot
```

```
Prova_e_Riprova vs Impfinity v4g1       10/96/94    (AArgh!)
Prova_e_Riprova vs Torch t18            81/86/33
Prova_e_Riprova vs Provascan 3.0        93/94/13
Prova_e_Riprova vs Frontwards v2        86/74/40
Prova_e_Riprova vs La Bomba             58/85/57
Prova_e_Riprova vs Night Train          98/36/66
Prova_e_Riprova vs DoorMat              99/36/65

Provascan 2.0d vs Night Train wins:     44/39/117
Provascan 2.0d vs DoorMat v0.1 wins:    46/48/106
```

Two words about these results:
They follow our expectations when fighting against Torch and Provascan 3.0; this version is quite better then 2.0d against imp stuffers as Doormat and Night Train.
On the contrary I can't understand what actually happened against La Bomba (at home my tests gave very different results...); against Frontwards the performance was slightly worse than the one by version 2.0d, maybe for the couple of istructions added for the clear (now P_e_R is no more so tiny, and also a bit slower in core cleaning!).
What is quite unexpected is the score against Impfinity; maybe here the bi-directional core clear is much better, and probably P_e_R is very often stoned!

Some interesting changes, left as an useful exercise to my twenty-five readers, can be:
-try to change the scan step
-try to change the carpet dimension
-try to change the starting offset of the core clear
-try to change boot distance
-try to write a better coreclear: smaller than mine [not so hard] and possibly bidirectional and anti-imp [not so easy]
-reengineer the program and reduce the number of istructions [quite hard]
-find the changes that make this version score better than Provascan 3.0 ;-)

Well, I think that's all for this issue, hoping to have been clear...
anyway you can mail me at pan0178@bologna.iperbole.it
Now I leave you with Planar and his interesting tutorial.

CDB tutorial, part 2

In the first part of this tutorial, we explored the most basic commands of GDB. Now we are ready for much more powerful features, features that give most of its power to CDB (and make its input look like line noise). We won't write macros yet, but we'll write CDB programs in the form of complex command lines.

Sequence

Last time, we found the point where Fahrenheit bombed itself with the "skip" command. It was a bit tedious because we were typing these two commands over and over:

```
(cdb) skip 999
(cdb) 0,4
```

There is a better way. Just type:

```
(cdb) skip 999 ~ 0,4
```

CDB will execute the two commands and display their results before giving the prompt back. So you can put a tilde instead of pressing <Enter> between two commands. Big deal.

Indeed, this feature will save a lot of your time. Press <Enter> now, and CDB will repeat the entire command line, not just the last command. You don't have to retype the two commands over and over.

Suppose you lose track of the time and you want to use "registers" to see the current cycle. If you type

```
(cdb) registers
```

you'll see the cycle count, but you lose your "previous command" buffer and you have to type "skip 999~0,4" again. But if you type

```
(cdb)  registers
```

with a space before the "registers", CDB will execute "registers" but keep the old "previous command". Now if you press <Enter>, CDB will execute the "skip" and the "list".

Notice that, when you type "skip 999 ~ 0,4", CDB will execute both commands and display their results. "skip" executes first and displays the next instruction to execute, then "0,4" displays the first five core locations. This is messy. If you type instead

```
(cdb) @skip 999 ~ 0,4
```


the "skip" command will execute without display. The "@" operator, placed before a command, will tell CDB to suppress the display of this command.

Tests

Use "go" to skip to a new battle, then type

```
(cdb) @skip 999 ~ @list 0 ~ if B!=3039 ~ 0,4
```

Here is what CDB will do:
1. silently step 1000 times (@skip 999)
2. silently list the first core location (@list 0)
   What's the point in listing the first core location if you don't display the listing ? It's to set the dot address (and the A and B values). You'll do it all the time in CDB programming.
3. Test if the B-value of location 0 is different from 3039. If so, CDB will execute the next command (0,4); if not, CDB will skip the next command and give the prompt back.

Now, you can press <Enter> until CDB displays the listing of Fahrenheit. You'll see that Fahrenheit has bombed itself.

So this is how the "if" command works: execute the following command only if the condition is true. The condition is any expression, it is true if not equal to zero, false if equal to zero. Easy, isn't it ?

Loops

Try this:

```
(cdb) @step ~ !1000
```

and notice that it is exactly equivalent to "@skip 999". What does it mean ? The "!" command means "repeat the beginning of the command line up to this (!) command". Here, this will repeat the "@step". The argument to "!" is the number of times that CDB must repeat the line. (Warning: for some strange reasons, sometimes MacpMARS will not accept a space between the ~ and the !)

So we told CDB to repeat 1000 times "@step". If we use

```
(cdb) @step ~ !999 ~ step
```

this is equivalent to "skip 999": step 999 times silently and once with display.

If you omit the argument to "!", CDB will loop forever. Typing

```
(cdb) @step ~ !
```

is equivalent to "continue", but slower. You'll have to press Control-C to get back to the prompt. (In this case, Control-C doesn't work in the X-window version. I have to type Control-C in the terminal window where I launched pMARS from. This will be fixed in the next version of pMARS.)

In addition to the loop count, you can use "if" to break out of a loop: if you use "if" to skip over the "!" command, the loop will stop. Use "go" to skip to a new battle and type:

```
(cdb) @step ~ @list 0 ~ if B==3039 ~ !
```

CDB will step until Fahrenheit has bombed itself. Use "registers" to get the exact cycle when this happens, subtract the remaining cycles from 80000, use "go" to start an new battle and skip to just before

the bombing. Check that the next instruction will bomb core location 0, step over it and check that it does bomb that core location.

So this is finally the right way of finding when and how Fahrenheit bombs itself. In just one short command line, we can do the same work as the tedious process of skipping in decreasing increments that we used last time. Note that we can use this technique only if we know which of Fahrenheit's instructions is bombed. I think Stefan has some predefined macros that will help us further. We'll keep them for part 3.

Changing the core

We'll want to experiment by changing some of Fahrenheit's constants to see which one makes it bomb itself the latest. For example, to set the B-value of location 1, we'll set the whole location with the command "edit". It takes an address as argument. In interactive mode, "edit" will prompt you for a line of redcode to put at this address. In a command line, "edit" will take the next "command" (whatever is between the next ~ and the following one) as the line of redcode. For example, if you type:

```
(cdb) edit 1
mov.i <100, <2000
```

CDB will set the contents of core location 1 to this instruction, in effect changing the B-value of the instruction to 2000. You could instead type:

```
(cdb) edit 1 ~ mov.i <100, <2000
```

If we want to see the effect of a big decoy on Fahrenheit's DJN stream, we can use:

```
(cdb) fill 5700,5800
dat 1, 1
```

This will set the 101 core locations between 5700 and 5800 to "DAT 1, 1". As with "edit", we can give the instruction on the command line after a ~. Determine on which cycle Fahrenheit bombs itself when this decoy is present.

Computing

CBD can also do arbitrary calculations (and display the results). The command to use for this is "calc". Try:

```
(cdb) calc 2+2
(cdb) calc B+1, .+A
(cdb) calc D=D+1
```

CDB provides 24 variables for your use: C...Z (remember that A and B contain the A-value and B-value of the dot address; you cannot assign to them). An expression can contain an assignment, as in the C language. You will often use "@calc" to do an assignment and avoid printing the result.

Nested loops

Let's make CDB work hard to determine the constant that will make Fahrenheit bomb its SPL line as late as possible. We'll try the values of the interval [1004,1007] for the B-field of the first MOV.

We'll use two nested loops: the outer loop will enumerate the 4 values for the constant, and the inner loop will execute the battle step by step, checking the SPL after each instruction.

For nested loop, we use the "!!" instruction. It marks the opening of the loop, which is closed by a matching "!". Our command lines will be (I've broken the long line, but you must type it in one line without the \):

```
(cdb) ca C=1003,X=3039
(cdb) !!~ca C=C+1,D=0~@ed1~mov.i<100,<C~!!~@ca D=D+1~@s~@0~if \
B==X~!~ca d~@g~@s~!4
```

This is a detailed explanation:

```
!!                      start the outer loop
   ca C=C+1, D=0           increment the outer loop counter, reset
                           the inner loop counter (and print the
                           constant)
   @ed 1~mov.i <100,<C     set the first MOV instruction
   !!                      start the inner loop
      @ca D=D+1              increment the inner loop counter
      @s                     step the warrior once
      @0                     set the dot address to 1
      if B==3039~!           end the inner loop if self-bombed
   ca D                    print the constant and number of cycles
   @g~@s                   skip to the first cycle of the next battle
!4                      loop 4 times for the outer loop
```

Four values is not a lot, but I couldn't do better with the strong suicidal tendency of Fahrenheit (see exercise 6 below). What would you do to reduce this tendency ?

There's no reason I should be the one doing all the work, so here are a few exercises. Note that you won't be able to test your solutions because my command line is very close to the maximum size allowed by pMARS. We'll have to define macros to overcome this limitation.

1. The above loop will take a lot of time. Accelerate it by stepping 10 by 10 instead of 1 by 1.
2. We're only interested in the value that gives the greatest number of cycles. Change the command line to display only that value and the corresponding number of cycles.
3. Use "fill" to abort the current battle instead of finishing it with "go". How much faster is this ?
4. Even faster: instead of doing the inner loop step by step all the way, skip to the greatest number of cycles so far, and start stepping from there if the SPL is still not bombed. (Is that clear ?)
5. Figure out a way of putting a sequence of instructions under an "if" instead of a single instruction, so that the sequence is executed only when the condition is true.
6. [hard] The inner loop fails to stop if the self-bombing doesn't occur on the SPL. How do you make it work in all cases ?

The end

This is all for this part. In the next one, you'll have the answers to the exercises, and Stefan will tell us about macros. There's a whole lot of useful predefined macros that come with pMARS.

Stefan is on vacation, so he didn't proofread this part. Blame him if it is not as good as part 1 (-:

I want to greet you with a few lines written by Myer R Bremer on issue 4...

```
################################################################################
EDITOR'S NOTE:  Damien is much too modest.  Imp spirals are one of the most
difficult subjects to understand.  Impfinity missed the '94 draft hill by
only a fraction of a point.  I'm sure he'll breach the hill very soon.  Spoke
too soon.  Impfinity has just entered the hill as I am posting.
################################################################################

Well, I think that now Impfinity is going very well!
Myer has a very good sight! ;-)
```

Questions? Concerns? Comments? Complaints? Mail them to:
Beppe Bezzi <bezzi@iol.it> or Myer R Bremer <bremermr@ecn.purdue.edu> or
for this issue to Maurizio Vittuari <pan0178@iperbole.bologna.it>

The hint
A new p-switcher
by Paul Kline

A colorful variety of p-switching mechanisms are sprouting this Spring, and it would be nice if someone would round them all up for comparison. Also nice if people would POST a few :-)

A simple, fast switch-on-loss routine for two p-components might look like this:

```
pflag     equ    (somenumber.lt.500)
pGold     ldp.ab #0,#0               ; get results of last battle
          ldp.a  #pflag  ,pGold      ; retrieve attempted strategy
          add.a  #1      ,@pGold     ; if a loss, increment strategy
          mod.a  #2      ,pGold      ; safeguard against brainwashing
          stp.ab pGold   ,#pflag     ; store current strategy
          jmz.a  select1 ,pGold      ; select strategy 1
          jmp    select2             ; select strategy 2
```

(The last jmp is unnecessary if strategy 2 immediately follows)

A powerful adaptation of the routine can be made with no extra instructions. By increasing the MOD number we have an assymetric switcher, by which the second strategy is selected more often than the first. This can be very helpful in pairing up a strong all-purpose warrior like Torch, with a special-purpose warrior like Clisson. Like many fast programs with spl-dat clears, Torch is vulnerable to a stone, which is in turn highly vulnerable to Clisson. Using an assymetric switcher to select Torch most frequently gives the best results against a variety of opponents, and the infrequent Clission strategy breaks up a protracted series of stone attacks.

This is the switcher used by Goldfinch which pairs a one-shot scanner w/multipass clear, with Clisson's dodger.

Paul Kline
pk6811s@acad.drake.edu

Extra Extra
Twister
by Beppe Bezzi

Tornado is, beetween my warriors,one my favourites and, being a very flexible bombing engine, I like a lot to tweak and improve it, testing new bombs and variations. When Tornado 3.0, that had a success beyond my expectations, was near the bottom of the hill, a long time indeed :-), I tried to fix some problems that caused its fall and I coded v 3.3 that's the one included in Twister. Jack uses a slightly different version but you can fit this one in the old Jack, using but one paper module, and results won't differ too much from those on the hill (let me something to publish next week :-)
The bombs are common dat <1,{1 deadly against clears and slowing djn stream

users, intermixed with one spl #xx to allow self bombing to enter the core clear.
Worth noting are the pattern, not exacly mod 5 but slightly translated, such way it's more difficult for a one shot scanner to slip through my bombs without noticing them, and the djn protection, jmz.b start,#0 stolen from Torch.

The qscan is rather similar to the one in Stepping Stone, being only a bit slower overall even if with a better bomb distribution. I coded it from the warrior I sent to J K Lewis tournament, a thing that proved deadly, I won 13 rounds alone, but too weak. It's a 50%c vamp engine dropping one far jump to the pit and two near jumps through and to the far jump in a six instructions loop, something like that:

```
jn   jmp     *qqstep,  qqstep                        ;jump near
jf   jmp     -bombn+pit-(3*qqstep)-qdisp,-qqstep     ;jump far

jn ..[qqstep cells].. jn .. [qqstep cells] .. jf
```

The first jn jumps to jf using a-field of the second jn, the other jumps go to the pit.
The pit is a standard self destructing, brainwashing pit.

```
;redcode-94
;name Twister
;author Beppe Bezzi
;strategy qscan -> Tornado bomber
;assert CORESIZE == 8000
;kill Twister

step    equ     -45
away    equ     4000+2  ;mod 5 +2
gate1   equ     (gate-4)

org     startq


qstep   equ     6
qrounds equ     8
qdisp   equ     -qstep*(qrounds*3/2-1)-60
qqstep  equ     qstep*qrounds
bigst   equ     100                     ;or something more :-)
qstart  equ     startq+145
qst     equ     qstart -(4*bigst)

pstep   equ     40
spacer  equ     4
cldst   equ     (bclr-bgate+spacer+5)


pit     spl     4
pit1    mov     -3,     <1300
        spl     pit1
        spl     pit1
        stp.b   #0,     @pit1
        jmp     pit1

;----Qscan

;don't ever think it's the right qscan pattern :-)

startq
s3 for 4
        sne.i   qst+4*bigst*(s3+0), qst+4*bigst*(s3+0)+bigst*1
        seq.i   qst+4*bigst*(s3+0)+bigst*2, qst+4*bigst*(s3+0)+bigst*3
        mov.ab  #qst+4*bigst*(s3+0)-found, found
    rof
        jmn.b   which,  found
s2 for 4
        sne.i   qst+4*bigst*(s2+5), qst+4*bigst*(s2+5)+bigst*1
```

```
        seq.i   qst+4*bigst*(s2+5)+bigst*2, qst+4*bigst*(s2+5)+bigst*3
        mov.ab  #qst+4*bigst*(s2+5)-found, found
     rof
        jmn.b   which,  found
s1 for 4
        sne.i   qst+4*bigst*(s1+10), qst+4*bigst*(s1+10)+bigst*1
        seq.i   qst+4*bigst*(s1+10)+bigst*2, qst+4*bigst*(s1+10)+bigst*3
        mov.ab  #qst+4*bigst*(s1+10)-found, found
     rof
        jmn  which,   found

which
found   jmz.b   boot,   #0      ;Pyramid decoding
        add.b   found,  pt2
        sne.i   @found, @pt2
        add.ab  #(bigst*2),found
        sne.i   -100,   @found
        add.ab  #bigst, found
qattack                         ;found.b punta il bersaglio
        mov     bombm,  @found    ;dat bomb found position

        add.ba  found,  qstone    ;\
        add.b   found,  qstone    ; >setup vamp pointers
        sub.ba  found,  bombf     ;/

;---vamp attack ---

qb1      mov     bombn,  *qstone
qb2      mov     bombf,  @qstone
qstone   mov     (1*qqstep)+bombn+qdisp,@(3*qqstep)+bombn+qdisp
qstart1  sub     qincr,   @qb1
        add.a   qincr,   bombf
qjump   djn.b   qb1,    #qrounds
        jmp     boot

bombn   jmp     *qqstep,  qqstep
qincr   dat     >-1*qstep,>-1*qstep

bombf   jmp     -bombn+pit-(3*qqstep)-qdisp,-qqstep

for 5
        dat     0,0
rof

;--- Tornado

start
boot

        mov     gate,   }pt2
        mov     gate,   *pt2
        mov     last,   <pt1
        spl     1,      1
        mov     {pt1,   <pt1
        mov     {pt1,   <pt1
        mov     {pt1,   <pt1
        mov     {pt1,   <pt1
        mov     {pt1,   <pt1

go      djn.b   @pt1,   #2      ;start Tornado
        mov     gate,   <pt1
pt1     div.f   #last,#last+1+away
pt2     div.f   #gate+away-5,#bigst+found

        dat     -25,    last-gate1+5
warr
gate
        dat     -25,    last-gate1+15
bombs   spl     #(step+1),  -step           ;hit spl
start1  sub     incr,   @b1
```



```
stone   mov     (0*step)+jump,*(1*step)+jump
b2      mov     bombs,  @stone
b1      mov     bombm,  *stone
jump    jmz.b   start1, #0          ;hit by spl
clr     mov     @djmp,  >gate1
        mov     @djmp,  >gate1
djmp    djn.b   clr,    {bombs
incr    dat     >-3*step,>-3*step
last

bombm   dat     <1,     {1

shift   dat     #40,    #40
```

Questions? Concerns? Comments? Complaints? Mail them to people who care.
authors: Beppe Bezzi <bezzi@nemo.it> or Myer Bremer <bremermr@ecn.purdue.edu>