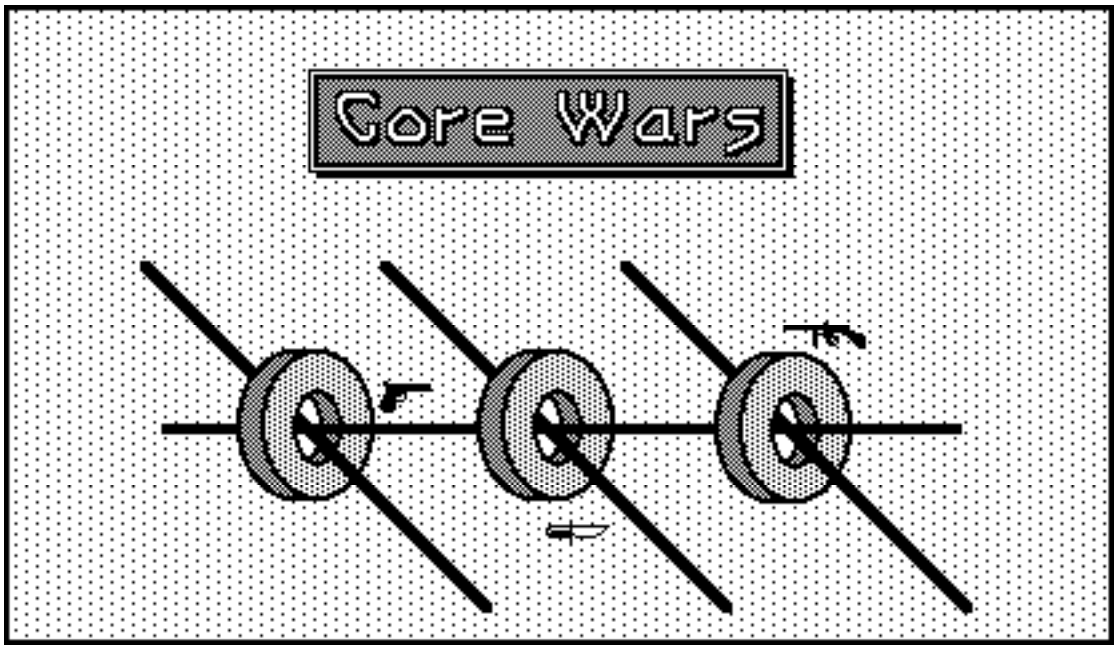


Welcome to the CORE WARS system for the Macintosh!

Core Wars is the Ultimate game for ***HACKERS***. It was designed by A. K. Dewdney, and described in his "Computer Recreations" collumns which appeared in the May '84 abd March '85 issues of Scientific American.

In Core Wars two programs battle for control of the imaginary *MARS* computer. The programs are written by the players in an "assembly language" called *REDCODE.* The players edit, assemble, and run their programs all within the Core Wars system. The programs may be stored on disk, and loaded later to fight other players' programs.

During battle, the players can see all 8000 cells of the memory in the MARS computer. The dynamics of the battle can be viewed, or the battle can be stopped, so that memory may be examined in detail.

There is no end to the interesting battle programs that can be written. Some of them are described in this manual. Others in the articles mentioned above. You may create others.

12310 Lee Ave
Waukegan, IL.
60085

Chapter I

Commands

When you load the Core Wars program by double clicking on its ICON, you will be entertained with a little rendition for whom the credit must obviously go to George Lucas. You may terminate this performance any time after the text begins to scroll by simply clicking the mouse.

Core Wars starts out with an empty screen and the menu bar. There are the 3 familiar menu items: *apple*, FILE, and EDIT, and a fourth unfamiliar one named **WAR**. It is with these menus that all of the Core Wars functions are invoked.

The File Menu

As always, the File menu contains the commands which manipulate the files which are input to the system, and output from the system. The quit
command also resides here. The user who is familiar with the Macintosh should find no surprises in the structure of this menu. The commands all have there most obvious meaning.

The Edit Menu

Once again, the user should find no surprises here, except for the fact that the UNDO command is not implemented. (So don't make any mistakes). The clipboard is managed as usual and should be transferrable into other programs as a TEXT item.

The War Menu

Here is where the new stuff starts. The War menu contains the commands which control the battle of the programs within the MARS computer.

Load Soldier: Implies that an edit window is open with a complete REDCODE program in it. The program is assembled and prepared for loading into the memory of the MARS system. When assembly is complete, a dialog box appears which allows the user to set the absolute load address of his program, and whether the program he is loading will be playing as WHITE, BLACK or neutral. Neutral is good for loading data structures which are accessories to programs.

Clear Battlefield: Erases the memory of the MARS computer. All programs which had been loaded there will no longer be there.

Fight: Implies that at least two battle programs have been loaded into the memory of the MARS computer with the "Load Soldier" command. A dialog box will allow the user to specify the absolute starting addresses of each of the battle programs, and will allow the user to set the battle time limit. Then the battle will commence.

Peace: Any battle currently in progress will be terminated. And the memory of the MARS computer will be cleared.

Debug: One of the many ways in which to bring the Debug window into the screen. The debug window allows programs to be single stepped, and disassembled. It also allows the memory of the MARS system to be examined in detail.

Chapter II

Battle Conditions

When both fighter programs are loaded into the memory of the MARS system, and the *Fight* command is invoked, then the *Core* window is put up and the battle begins. MARS separates the programs in the system by assigning them to an owning player. The players are named Black, and White. At first Black and White own a single program each; but during the course of the game, either of these programs may opt to use the SPL instruction which splits the program into two independant programs. Therfore the structure of the war can become quite complex, with each player owning many programs.

In this complex environment, MARS keeps order by enforcing a simple turn taking rule. Each player executes one instruction per turn. If a player has more than one program, then the programs execute in rotation. That is, if a player has 4 programs, then #1 will execute on the first turn, #2 on the second, #3 on the third, and so forth. This means that if a player has N programs running, each of his programs receive $1/N^{th}$ of the available turns for that player.

Programs attack each other by forcing each other to execute an illegal instruction (typically a zero). This may be done by locating the program and then storing zeroes in it. Programs can be defended by making them mobile, and writing them so that they move themselves out of the way of an attack.

Whenever a program executes an illegal instruction, that program dies. The game is won when all the programs of a particular player are dead.

It is possible to subvert an enemy program and make it execute your own code by storing a jmp instruction in it. Although this gives your side more power, the subverted program still counts as one of the enemy programs which will have to be destroyed in order to win the game.

The war has a time limit which is measured in turns. if after executing the specified number of turns there has been no winner, then the game is declared a draw.

Chapter III

MARS architecture

The memory of the MARS computer consists of 8000 cells. In this implementation, each cell consists of 3 parts: the op-code, argA, and argB.
If the cell is used to store data, then the op-code, and argA typically contain zero, and argB contains the data which must be a number between -7999 and 7999. If the cell contains an instruction, then the op-code contains the operation code of the instruction, argA contains the first argument, and argB contains the second argument.

Programs reference other cells by relative addressing. Therefore if an instruction references cell #2, that really means the cell which is two cells past the current cell. It is impossible to make an absolute reference.

Two forms of reference are possible: direct and indirect. An instruction makes a direct reference by specifying a cell number. That number is added to the absolute address of the instruction to yeild the absolute address of the referenced cell. An instruction makes an indirect reference by referencing a cell which points to the referenced cell. The absolute address of the referenced cell is calculated by adding the absolute address *of the pointer* to the contents of the pointer.

For example the instruction: mov bomb, @target moves the contents of the cell named 'bomb' into the word pointed to by 'target'. Lets us say that the address of target is 100, the content of target is 100, the address of bomb is 50 and the address of the instruction is 20. The contents of bomb would then be stored at 100 locations past 'target' or address 200. Were we to code this instruction without symbols we would write it as: mov 30,@80

Since 'bomb' is at address 50, and the instruction is at 20, then 'bomb' is 30 cells past the instruction. Similar reasoning is used to explain why target is referenced by 80.

Chapter IV

Redcode

REDCODE is the language in which all the fighting programs are written. It is a symbolic assembly language complete with tags, symbols, and expressions.

Symbols are comprised of 1-8 alphanumeric characters, the first of which must be alphabetic. Constants are signed decimal numbers between -7999 and 7999. Expressions are combinations of symbols and constants separated by + or - operators, and possibly surrounded by parenthesis.

Jmark is a valid symbol.
-423 is a valid constant.
Jmark-1 is a valid expression.

The syntax of a REDCODE statement is as follows:

[<tag>] <op-code> <arguments> [<comments>]
Or @<org-expression>
Or *<comments>

The optional tag is a symbol which takes on the value of the address of the instruction in which it appears. It can then be used as an argument in other instructions. Forward references are allowed.

The Op-code is one of the valid mnemonics of the MARS system.

The arguments may be a single expression, or two expressions separated by a comma.

The org-expression is an expression which specifies a load address relative to the beginning of the program. Thus if I want to load an instruction 5 cells before the beginning of the program I would use the statement: @-5

Addressing modes are specified by prefixing a character in front of the argument. Normal direct addressing is specified by no prefix character as in:

```
jmp     cloop
```

Where cloop is the direct address. Indirect addressing is specified by prefixing the address expression with an '@' character as in:

```
mov     bomb,@target
```

Which moves the contents the word bomb to the cell indirectly pointed at by target.

Finally, immediate numbers are represented by prefixing the immediate expression with a '#' character as in:

```
mov     #12, counter
```

Which moves the number 12 into the cell named counter.

INSTRUCTIONS:

DAT **DATA**

This is not an instruction. It is used to place data into a program. i.e. :

```
target  dat  100    points 100 cells ahead
count   dat   20     count 20 times.
```

MOV **MOVE**

Moves the contents of argA to argB. ArgA may be immediate, but argB must reference a cell either directly or indirectly. When the contents of one cell are moved to another, the entire cell is moved including the opcode and argA. When an immediate constant is moved into a cell, then the opcode and argA are set to zero.

ADD **ADD**

The contents of argA are added into argB.

SUB **SUBTRACT**

The contens of argA are subtracted from argB.

JMP **JUMP**

The program jumps to the cell referenced by argA. Reference be either direct or indirect.

JMZ **JUMP IF ZERO**

The program jumps to the cell referenced by argA if the contents of argB is zero.

JMG **JUMP IF GREATER THAN ZERO**

The program jumps to the cell referenced by argA if the contents of argB is greater than zero.

DJZ **DECREMENT AND JUMP IF ZERO**

The cell reference by argB is decremented. If the result is zero, then the program jumps to the cell referenced by argA.

CMP **COMPARE**

If argA is not equal to argB, then the next instruction is skipped.

SPL **SPLIT**

The program splits into two programs, one of which starts at the instruction pointed to by argA, the other continues at the instruction following the split.

PCT **PROTECT**

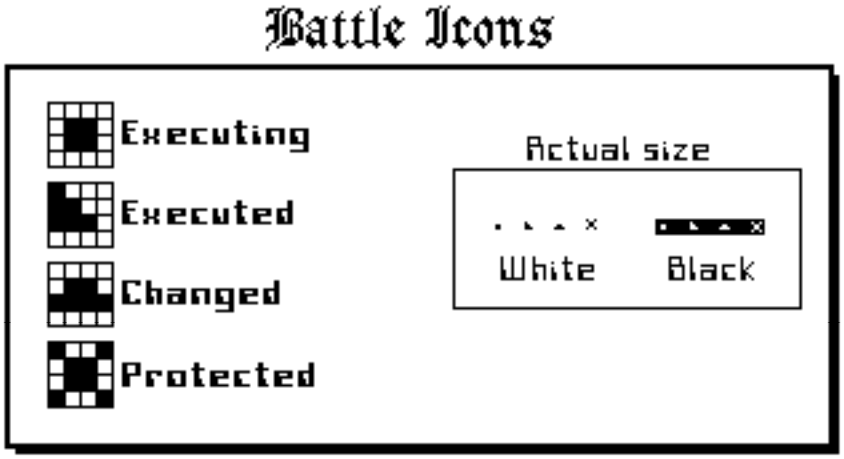
** This instruction is not an official part of the cannon of the MARS system as defined by the designer, however it is under consideration, and so it has been supplied here.

The cell referenced by argA is protected. A protected cell survives one attempt to write to it without being changed. The attempt however removes the protected status so that the second attempt will succeed.

Chapter V

The CORE window

When the battle begins, the CORE window opens. The CORE window contains 8000 icons, one for each cell in the memory of the MARS system. At first each cell is colored in with a gray background, but as the battle proceeds, the appropriate cells will be filled with icons which describe the battle. There are 4 types of icon, and each comes in two colors. Black icons represent activity by the Black player, and white icons reprepsent the White player.



The *Executing* icon occurs in a cell which is currently being executed. There will be one such cell for each program in the system. The *Executed* icon occurs in a cell whose last activity was to be executed. Thus as a program executes it leaves a trail of these icons behind it. The *Changed* icon occurs in a cell whose last activity was to be altered by a program. And the *Protected* icon occurs in a cell whose last activity was to be protected by a program.

As the battle proceeds, these icons ripple in and out of cells creating a very dynamic display of the battle. It is easy to see what each program is doing, and who is winning and losing.

Chapter VI

debugging

When the battle first begins, the Debug window will appear along with the CORE window, and the battle will be in single step mode. You can start the battle going by either getting rid of the Debug window (click it's close box), or by pressing the run button.

You can retrieve the Debug window at any time during the battle either by choosing the DEBUG command from the WAR menu, or by clicking the mouse in the CORE window. Clicking the mouse in the CORE window has the extra added effect of displaying the contents of the cell which was clicked. The contents of this cell are displayed in symbolic mode in the scroll area of the Debug window. This means that the fighting programmers can halt the battle at any time and look around in the system to see what is going on.

There are 3 buttons in the debug window: STEP, RUN and QUIT. The STEP button places the current battle into single step mode. Each time the STEP button is pushed one more turn will be executed, and the executed instructions will be disassembled into the scroll area of the Debug window. Pressing the RUN button cancels single step mode and starts the battle going again. The QUIT button causes the CORE WARS system to exit to the FINDER.

Chapter VII

Examples

The examples that follow were created during the development of CORE WARS, and therefore reflect its evolution. Early on, the assembler was more primitive and would not accept symbols or comments. Later these features were added. So some of the examples look primitive, and others look more advanced.

IMP, DWARF, and GEMINI are programs which were copied directly from A.K. Dewdney's article. I added the cannon structure to IMP CANNON and GEMINI CANNON. WORM and VAMPIRE are my own creations.

IMP

IMP is the simplest of REDCODE programs. Although it is not very smart, and cannot possibly win, it is very aggressive and often forces a draw. The IMP program consists of a single MOV instruction, which copies itself to the next cell to be executed. In this manner IMP roars through the MARS memory, converting every cell it touches into a "MOV 0,1" instruction.

```
@0
   mov 0,1
```

DWARF

DWARF is another very simple program, but of a different sort. Dwarf makes a blind, but nearly saturating attack on the enemy by spraying zeroes through memory. The DWARF program is 4 cells large, so it attacks every fifth cell in the system and conveniently misses itself.

```
*
* Basic fighter program which sprays zeroes
* all through memory.
*
*
@-1
bombpt  dat 0
*
lup     add #5,bombpt
        mov #0,@bombpt
        jmp lup
```

GEMINI

GEMINI is an example of how a program can move itself in memory. This technique is useful to "step out of the way" of DWARF's killing spray, or to "jump over" a rampaging imp.

```
@-2
    dat 0           Source pointer (starts at top of program)
    dat 99          Destination pointer (points 100 cells away from first)
    mov @-2,@-1     Move the source to the destination
    cmp -3,#9       have we moved the whole program?
    jmp 4           yes: jump out of the loop.
    add #1,-5       otherwise increment
    add #1,-5       the pointers.
    jmp -5          and jump back for more
    mov #99,93      finish the new program
    jmp 93          and jump to it.
```

IMP CANNON

If one IMP is a formidable weapon, then many ought to be more so. Here is a program that fires off an imp every 200 turns. Actually the reasoning is not valid, since each new imp robs the others of execution time, and they all slow down. A slow IMP is very easy to kill. (left as an exercise for the reader).

```
*
* The imp cannon fires off an imp every
* so often.
*
@-1
timer dat 200
*
lup   djz fire,timer
      jmp -1
*
fire  mov #200,timer
      spl imp
      jmp lup
*
imp   mov 0,1
```

GEMINI CANNON

The GEMINI CANNON is similar to the IMP CANNON except that it fires off a new gemini every once in a while. This can actually be a formidable enemy. It needs a bit more intelligence though.

```
@-1
    dat 200
    djz 2,-1
    jmp -1
    mov #200,-3
    spl 2
    jmp -4
    mov #0,3
    mov #99,3
    jmp 3
    dat 0
    dat 99
    mov @-2,@-1
    cmp -3,#9
    jmp 4
    add #1,-5
    add #1,-5
    jmp -5
    mov #99,93
    jmp 93
```

WORM

The WORM is a lovely little program that is very difficult to destroy. It is not very aggressive, and could be hopped over pretty easily.

```
*
* create the creeping unstoppable worm.
*
* The worm is a consecutive array of imps.
* Each imp is a separate process, and they
* are running in consecutive locations.
*
* no frontal assault of any kind can destroy
* a worm.   It is self repairing.  The only
* effective attack against a worm is to
* destroy is from left to right.
*

@-1
length  dat 5
        spl extrude
        djz 2,length
        jmp -1
        mov #0,extrude ;Stop making the worm
*
extrude spl 0     ;spin here and make a worm
        mov 0,1   ;of imps
```

VAMPIRE

The vampire is my favorite of the examples. Vampire throws jump instructions around in memory (I call them "FANGS"). If one of the fangs happens to land in the enemy program then the enemy will jump into a trap. The trap creates several hundred new programs which do nothing but jump to themselves. This of course robbs the enemy of almost all his effective power. Any other programs that he might have running will be going so slow as to be almost harmless. When vampire detects that one of its fangs has found a jugular, it begins a methodical erasure of memory. If all goes well, all the enemies programs will be erased, and VAMPIRE will win.

```
*
* Vampire -- seek out the enemy, drain the
* life out of him, and then erase him in his
* weakened condition
*
@-230
nprc  dat 500  The number of dummy processes
*              that the enemy will be forced
*              to make.
spin  spl 0
      djz 2,nprc
trap  jmp -1
*
      mov null,trap
      mov #0,spin
null  jmp 0
@-2
fang  jmp @1
bite  dat 1000
@0
start mov #(spin-bite),@bite
      sub bite,@bite  ;adjust address of fang
      sub #1,bite
      mov fang,@bite
      add #7,bite
*
      cmp #500,nprc   ;have we caught anybody?
      jmp start       ;nope.
*
* the enemy has been caught, and is currently
* being drained of energy.  So start the
* process of erasing the world.
*
      mov #100,bite  ;start near here.
erase mov #0,@bite   ;erase everything and
      add #1,bite    ;hope he dies before we
      jmp erase      ;kill ourselves
```

Chapter VIII

Source Code Availability

CORE WARS is a moderately complex program. It embodies nearly 2000 lines of well commented AZTEC C code, and would make a very effective learning tool. You can purchase a disk with the complete source code on it for $15 by sending a check to:

Robert Martin
12310 Lee ave.
Waukegan, Illinois
60085

The following list contains some of the issues handled by the source code.

- modal dialogs
- explicit bitmap operation
- sound generator
+ controls
  - scrollBars
  - radioButtons
  - Buttons
- windows
- edit records
- standard file package
- file operations
- event handling
- segments
- Alert Boxes
- Animation Techniques
- recursion
- Memory Management
- Resources
- menu management

Enjoy your CORE WARS system. I would be happy to hear about new battle programs and battle histories. Mail me a letter, or EMAIL on compuserve (70156,240)

Stay tuned for more downloads.