[Note: This is a reproduction of the Core War Guidelines
originally produced by Jones and Dewdney in March of 1984]

CORE WAR GUIDELINES

D. G. Jones and A. K. Dewdney

Department of Computer Science
The University of Western Ontario

March, 1984

These guidelines suggest ways of implementing a simple version
of Core War, the game described in the "Computer Recreations"
department of Scientific American in May, 1984.

Programming examples are given in a style similar to that of the
Pascal and C languages. Readers unfamiliar with constructs of
these languages, such as the CASE statement, may want to consult
a Pascal manual.

The Redcode Instruction Set

Core War programs are written in an assembly-type language
called Redcode. The eight instructions included in the version
of the language presented here are by no means the only ones
possible; indeed, the original implementation of Core War, done
on a minicomputer, had a larger instruction set. If there are
many kinds of instructions, however, the encoded form of each
instruction takes up more space, and so the area of memory
needed for CORE must be larger. Mars, the program that inter-
prets Redcode programs, also grows as the size of the instruc-
tion set increases. The complexity of your Core War implementa-
tion may be constrained by the amount of memory available in
your computer.

CORE WAR GUIDELINES   2.

If you choose to a create your own Redcode instruction set, two
points should be kept in mind. First, each Redcode instruction
must occupy a single location in CORE. In many assembly lan-
guages an instruction can extend over multiple addresses, but
not in Redcode. Second, there are no registers available for

Redcode programs; all data are kept in CORE and manipulated there.

Here is a simple Redcode instruction set:

Encoded Form	Mnemonic Symbol	Argum ent		Action
0	DAT		B	Initialize location to value B.
1	MOV	A	B	Move A into location B.
2	ADD	A	B	Add operand A to contents of location B and store result in location B.
3	SUB	A	B	Subtract operand A from contents of location B and store result in location B.
4	JMP		B	Jump to location B.
5	JMZ	A	B	If operand A is 0, jump to location B; otherwise continue with next instruction.
6	DJZ	A	B	Decrement contents of location A by 1. If location A now holds 0, jump to location B; otherwise continue with next instruction.
7	CMP	A	B	Compare operand A with operand B. If they are not equal, skip next instruction; otherwise continue with next instruction.

Addressing Modes

There are various ways of specifying memory addresses in an assembly-language program. In order to make the execution of a Redcode program independent of its position in memory, a special form of relative addressing is used. Again, your version of Redcode may have different addressing modes or additional ones, al-

CORE WAR GUIDELINES   3.

though you should be aware when you choose modes that Mars will load your Redcode program at an address in CORE that cannot be predicted in advance.

The three addressing modes in our version of Redcode are identi- fied by symbols placed before the argument:

Encoded Form	Mnemonic Symbol	Name	Meaning

| ------- | --------- | ----------- | --------------------------------- |
0	#	Immediate	The number following this symbol is the operand.
1	<none>	Relative	The number specifies an offset from the current instruction. Mars adds the offset to the address of the current instruction; the number stored at the location reached in this way is the operand.
2	@	Indirect	The number following this symbol specifies an offset from the current instruction to a location where the relative address of the operand is found. Mars adds the offset to the address of the current instruction and retrieves the number stored at the specified location; this number is then interpreted as an offset from its own address. The number found at this second location is the operand.

All address arithmetic is done modulo the size of CORE. The MOD operator is the remainder after division, and so 5096 MOD 4096 is 1000. Thus if your CORE array has 4096 locations, a reference to location 5096 is taken as a reference to location 1000.

Because a program can never refer to an absolute address, some addressing modes for some operands do not make sense. For example, in the instruction MOV #5 #0 the operand to be moved is the immediate value 5, but the argument indicating where it is to be moved is not an address but the immediate value 0, which has no clear interpretation. The allowed modes can be stored in a two-dimensional table that is consulted by Mars when it interprets the instructions.

The example below should illustrate how the instructions and the addressing modes work. The code is taken from a battle program

CORE WAR GUIDELINES   4.

called Dwarf, which was described in the Scientific American article. Here, however, it has been altered to work in a CORE array of 4096 locations instead of one with 8000 locations.

Location	Inst ruc tion		Action
--------	---- --- ------		-------------------------------------
0:	DAT	0	This location initially holds 0.
1:	ADD #4	-1	Operand A is 4. The address of operand B is 1+(-1)=0; hence operand B is the content of this location in CORE. Add the two operands and store the result in location 0.

```
2:      MOV #0    @-2  Operand A is 0. The address  of operand
                       B   is   the   number   stored  at  location
                       2+(-2)=0, and so operand A is stored at
                       the  location given by 0+(content of lo-
                       cation 0).

3:      JMP       -2   Continue execution with the instruction
                       at location 3+(-2)=1.
```

Translating a Redcode Program into an Encoded Format

CORE is typically implemented as and array of integers. Suppose
the array has 4096 (or 2^{12}) elements; then exactly 12 bits are
required for each operand field in an instruction. There are
three addressing modes, and so for each mode field two bits suf-
fice. If four bits are alloted to the instruction itself, each
instruction can be stored in 32 bits, or four bytes.

Each element of the array might have the following format:

```
number of bits:     4       2           2        12   12
fields:           type  mode for A   mode for B    A    B
```

The example below suggests one way of encoding an instruction in
a 32-bit binary integer. (A somewhat different scheme, based on
decimal integers, was given in the Scientific American artice.)

Instruction	Fields		Encoded Integer
MOV #5 @20	type =	1	$1 * 2^{28}$ = 268435456
	mode for A =	0	$0 * 2^{26}$ = 0
	mode for B =	2	$2 * 2^{24}$ = 33554432
	A =	5	$5 * 2^{12}$ = 20480
	B =	20	$20 * 2^{0}$ = 20

			302010388

CORE WAR GUIDELINES   5.

The experience programmer will probably want to automate the
conversion of mnemonic instructions into integers by writing a
Redcode assembler to do the translation.

Rules of Core War

The rules of Core War are few and simple. The simpler the rules
are, the simpler the referee program needs to be. Here are the
rules we have been using:

   1. Two battle programs are loaded into CORE at randomly cho-
      sen starting locations, subject to the constraint that
      the programs cannot overlap.

2. The battle proceeds as Mars executes one instruction from
   program X, one instruction from program Y, one from X,
   one from Y, and so on, until one of two events happens:

   i) A previously specified number of instructions has
       been executed and both programs are still running.
       The battle is then declared a draw and ended.

   ii) An instruction is encountered that cannot be inter-
       preted by Mars and hence cannot be executed. The
       program with the faulty instruction is the loser.

One problem with these rules is that they reward small but un-
interesting programs such as:

    JMP 0   /execute this instruction over and over.

This program is completely unaggressive, and yet it is hard to
destroy simply because of its size. A program such as Dwarf, on
the other hand, is so destructive that it is hard to write lar-
ger, more complicated Redcode programs that can compete against
it. The larger program has too little time to launch its attack
before a "zero bomb" from Dwarf strikes somewhere in its
instructions. Other rules might be able to alleviate these prob-
lems, but remember that the referee program must be able to im-
plement the rules.

                        The Mars Program

In addition to serving as the referee in a Core War battle, Mars
is responsible for executing the battle programs. Mars first
loads two battle programs X and Y into the CORE array, putting
them at arbitrary positions but taking care that one program is
not written over the other. For each program Mars also needs to
know the address where execution is to start; this information
can be put into a file with the encoded Redcode program.

CORE WAR GUIDELINES   6.

During execution Mars must continually keep track of the current
instruction pointer for each program. If CORE is implemented as
an array of integers, the instruction pointer is simply an index
into the array. Mars then executes a simple loop:

LOOP: IF X's next instruction can be executed, THEN execute it
          ELSE declare X the loser, Y the winner; GOTO ABORT
        IF Y's next instruction can be executed, THEN execute it
          ELSE declare Y the loser, X the winner; GOTO ABORT
        count = count + 1
        IF count < limit THEN GOTO LOOP
          ELSE GOTO ABORT

The counter is kept in case neither program is able to defeat
the other. Without it Mars could be trapped in an endless loop.

Executing An Instruction

The following pseudocode for a part of Mars illustrates how a
Redcode instruction can be interpreted and executed. (Note tat
the operator DIV gives the integer result of division, that is,
100 DIV 30 yields a result of 3.) In the example we assume it is
program X's turn to execute its next instruction, which is spec-
ified by the variable X-index.

Integer variables used:

```
  instruction  /current instruction (encoded as 32-bit integer)
  type         /the instruction type (encoded)
  mode-A       /the addressing mode for operand A (encoded)
  mode-B       /the addressing mode for operand B (encoded)
  field-A      /the encoded number in field A
  field-B      /the encoded number in field B
  address-A    /address of operand A (unless immediate)
  address-B    /address of operand B (unless immediate)
  pointer      /variable used in calculating indirect addresses
  operand-A    /value of operand A
  operand-B    /value of operand B
  answer       /result calculated by the instruction
```

Program statements:

```
 instruction = CORE[X-index]                    /get instruction
 type        = instruction DIV 2^28             /get first 4 bits
 mode-A      = (instruction DIV 2^26) MOD 2^2   /get next 2 bits
 mode-B      = (instruction DIV 2^24) MOD 2^2   /get next 2 bits
 field-A     = (instruction DIV 2^12) MOD 2^12  /get next 12 bits
 field-B     = instruction MOD 2^12             /get last 12 bits
```


CORE WAR GUIDELINES   7.

  CASE mode-A OF

    0: operand-A = field-A
        /immediate mode; operand given in the field itself

    1: address-A = (X-index + field-A) MOD 4096
       operand-A = CORE[address-A]
        /relative mode; address of operand A is index + field A;
        /operand A is the content of CORE at this address

    2: pointer   = (X-index + field-A) MOD 4096
       address-A = (pointer + CORE[pointer]) MOD 4096
       operand-A = CORE[address-A]
        /indirect mode; pointer to the address of operand A is
        /index + field A; address of operand A is the the value
        /of the pointer + the content of the location it points
        /to; operand A is the content of CORE at this address.

       OTHERWISE: GOTO ABORT

At this point in the program a similar CASE statement, based on

the variable B-mode, is employed to assign a value to operand B.
An error-checking routine can be invoked to make certain that
the mode of each operand is allowable for an instruction of the
kind specified by the variable type. If no errors have been en-
countered, Mars continues with the following code:

```
X-index = X-index + 1             /increment the instruction
                                  /index in preparation for
                                  /program X's next turn; the
                                  /index may subsequently be
                                  /altered by a JMP, JMZ, DJZ
                                  /or CMP instruction
```

```
CASE type OF

  0: GOTO ABORT                       /DAT statement; cannot
                                      /be executed

  1: CORE[address-B] = operand-A   /MOV instruction

  2: answer = operand-B + operand-A   /ADD instruction
     CORE[address-B] = answer

  3: answer = operand-B - operand-A   /SUB instruction
     CORE[address-B] = answer

  4: X-index = operand-B              /JMP instruction; the
                                      /next instruction is at
                                      /the location specified
                                      /by operand B
```

CORE WAR GUIDELINES   8.

```
  5: IF operand-A = 0 THEN           /JMZ instruction; if
        X-index = operand-B          /operand A is zero, jump

  6: answer = operand-A - 1          /DJZ instruction; dec-
     CORE[address-A] = answer        /rement operand A and
     IF answer = 0 THEN              /store result;  if it is
        X-index = operand-B          /zero, jump

  7: IF operand-A = operand-B        /CMP instruction; if
        X-index = X-index + 1        /the operands are equal,
                                     /skip next instruction

  OTHERWISE: GOTO ABORT

END          /An instruction of program X has been interpreted
             /and executed successfully; Mars now goes on to
             /execute the next instruction of program Y.

ABORT:       /This label is reached only if the current in-
             /struction of program X could not be interpreted
             /and executed. Program X is declared the loser
             /and program Y the winner.
```

Displaying a Battle

The author of a Redcode program would become frustrated if his
creation were loaded into the darkest regions of CORE and then,
after a short battle, pronounced dead by the referee Mars with-
out any indication of what went wrong. Was the opposing program
superior, or was there merely a bug in the program? Some trace
of the events in a battle is needed.

The simplest display of an ongoing Core War battle is a listing
of both programs' execution on a split screen. The address of
each instruction and its mnemonic symbol ought to be given. A
typical display might look something like this:

```
     1135: MOV       0    1        202: ADD   #4    -1
     1136: MOV       0    1        203: MOV   #0    @-2
     1137: MOV       0    1        204: JMP         -2
     1138: MOV       0    1        205: ADD   #4    -1
     1139: MOV       0    1        206: MOV   #0    @-2
     1140: MOV       0    1        207: JMP         -2
     1141: MOV       0    1        208: ADD   #4    -1
     1142: MOV       0    1        209: MOV   #0    @-2
     1143: MOV       0    1        210: JMP         -2
     1144: MOV       0    1        211: ADD   #4    -1
     1145: MOV       0    1        212: MOV   #0    @-2
           PROGRAM X                      PROGRAM Y
```

CORE WAR GUIDELINES   9.

The instruction currently being executed would always be dis-
played at the bottom of the screen, along with the preceeding 23
instructions (on a 24-line screen). The information for the dis-
play might be generated as each instruction is interpreted and
the values of the various fields are determined. A subroutine
would output the mnemonic or the numerical value corresponding
to each encoded field. It must be recognized, however, that this
output will slow th execution of Mars. In a battle lasting
several thousand steps you may want to suppress the output.

Although the display described above is useful for debugging
programs by stepping through their instructions, it gives no
clear idea of the overall action. It is rather like seeing only
two televised close-ups of the gladiators' feet in the Roman
arena.

Another method we intend to try creates a circular display on a
graphics terminal. Two large concentric circles represent the
CORE array, and symbols within the annulus formed by the circles
represent the two battle programs.

The extreme right-hand point of the circles can be taken as
address 0. The symbols could be any simple but distinguishable
shapes (say a dot and a line). Additional information could be
displayed, such as the elapsed time or a key identifying the
programs corresponding to the symbols. One could even have a

momentary flash (an asterisk?) appear at addresses altered by
MOV commands; the flashes would be interpreted either as artil-
lery or relocation. Of course the more complicated the display
is, the slower the battle will proceed.

CORE WAR GUIDELINES   10.

The position of each symbol in the circular display can be cal-
culated from the instruction index for the corresponding pro-
gram. Suppose a program is executing at location I in a CORE
array with 4096 elements and its position is to be shown on a
screen that has 1000 distinguishable points vertically and hori-
zontally. The coordinates on the display screen are then given
by the expressions 400 * cos(2piI/4096) and 400 *
sin(2piI/4096).

With a lower-resolution graphics system the circular display
might not be feasible, but the CORE array could still be repre-
sented as a rectangle. Again symbols would indicate the location
of each program, although they might appear to jump or flicker
slightly even tough execution moved "continuously" from address
to address. A crude version of such a display might even be im-
plemented on a purely character-oriented screen.

Extensions to Redcode and Mars

The version of Mars presented here is easy to implement, and
many of you may want to see how you can expand the Core War sys-
tem. For example, in this version of Mars only two battle pro-
grams can be run at once. Would it be hard to let more programs
execute? How about a new Redcode instruction that allows a run-
ning program to start up another program it has copied into a
free area of CORE, and thereby increase the chance that at least
one program from its "team" will survive the Core War battle?

The Redcode instruction set given here is a simple one. Those
of you with access to a larger computer may want to experiment
with new instruction sets and addressing modes, possibly making
Redcode more like a real assembly language. Instructions that
protect a larger program from a small, hard-to-defeat one would
help to elevate Core War to a higher, more interesting level.

We would welcome documentation and listings of Core War systems
and Redcode battle programs from anyone who thinks he has come
up with a particularly interesting or innovative idea. Send
correspondence to:

                        D. G. Jones or A. K. Dewdney
                        Department of Computer Science
                        The University of Western Ontario
                        London, Ontario
                        Canada
                        N6A 5B7