The beginners' guide to Redcode

Version 1.22

Contents

Preface

There aren't too many beginners interested in the game of Core War these days. Of course, this is quite natural -- not that many people would consider optimizing assembly code to be fun anyway -- but one reason for the high starting threshold may be the difficulty of finding information on the very basics of the game. True, there are many good documents around, but most of them are either too technical, outdated, too hard to find or simply incomplete.

That is why I decided to write this guide. My aim is to guide newcomers from their very first contact with Core War and Redcode to the point where they can write a working (if not successful) warrior, and are able to proceed to the more technical stuff.

To be honest, I am still a beginner in this game myself. I know the language fairly well, but have yet to produce a really successful warrior. But I decided not to wait until I've gotten more experienced and instead write this guide as soon as possible while I still have a fresh memory of what it's like to be a new player struggling to comprehend the peculiarities of the game.

This guide is intended for the *very* beginners. No previous knowledge of any assembly language (or programming in general) should be needed, though knowing the general idea should help in understanding the basic terms. Redcode, especially the modern versions, may look like any assembly code, but it is more abstract than most and quite different in details from any other assembly language.

The flavor of Redcode used in this guide is (mostly) the current *de facto* standard, the ICWS '94 Standard Draft with pMARS 0.8 extensions. (Sort of like the Netscape extensions to HTML... Hmm... Luckily we still don't have a Microsoft Corewar Simulator. Maybe they think the market's too small.) The earlier '88 standard will be mentioned briefly, but this guide is mostly about the '94 standard. For those who want to learn it, there are plenty of '88 tutorials available on the Web.

Important: There is no simple way to teach Redcode -- or any programming language -- in a strictly linear way. While I've tried to organise this guide into a somewhat sensible order, *if you want to skip around, by all means do so*. That's what the contents section is for.

To maintain any coherency at all, I've often been forced to show you things and then explain them a few chapters later. If you don't seem to understand something, read on for a while. If you still can't figure it out, try browsing around to see if it's explained in some other chapter.

Everybody learns in a different way, and so any order you decide to read the chapters in is probably better than the one I've chosen. But if you think something is boring and leave it completely unread, the chances are you'll miss some important piece if information. I've tried to mark important parts with *emphasis*, so you know where to stop and think, but try to read everything carefully. I simply can't spell everything out, or this guide would grow too long to read.

Introduction to Core War

What is Core War?

Core War (or *Core Wars*) is a programming game where assembly programs try to destroy each other in the memory of a simulated computer. The programs (or

warriors) are written in a special language called *Redcode*, and run by a program called *MARS* (*Memory Array Redcode Simulator*).

Both Redcode and the MARS environment are much simplified and abstracted compared to ordinary computer systems. This is a good thing, since CW programs are written for performance, not for clarity. If the game used an ordinary assembly language, there might be two or three people in the world capable of writing an effective and durable warrior, and even they wouldn't probably be able to understand it fully. It would certainly be challenging and full of potential, but it'd probably take years to reach even a moderate level of skill.

How does it work?

The system in which the programs run is quite simple. The *core* (the memory of the simulated computer) is a continuous array of instructions, empty except for the competing programs. The core wraps around, so that after the last instruction comes the first one again.

In fact, the programs have no way of knowing where the core ends, since there are no absolute addresses. That is, the address 0 doesn't mean the first instruction in the memory, but the instruction that contains the address. The next instruction is 1, and the previous one obviously -1.

As you can see, the basic unit of memory in Core War is one instruction, instead of one byte as is usual. Each Redcode instruction contains three parts: the *OpCode* itself, the source address (a.k.a. the *A-field*) and the destination address (the *B-field*). While it is possible for example to move data between the A-field and the B-field, in general you need to treat the instructions as indivisible blocks.

The execution of the programs is equally simple. The MARS executes one instruction at a time, and then proceeds to the next one in the memory, unless the instruction explicitly tells it to jump to another address. If there is more than one program running, (as is usual) the programs execute alternately, one instruction at a time. The execution of each instruction takes the same time, one cycle, whether it is `MOV`, `DIV` or even `DAT` (which kills the process).

Starting with Redcode

The Redcode instruction set

The number of instructions in Redcode has grown with each new standard, from the original number of about 5 to the current 18 or 19. And this doesn't even include the new modifiers and addressing modes that allow literally hundreds of combinations. Luckily, we don't need to learn all the combinations. It is enough to remember the instructions, and how the modifiers change them.

Here is a list of all the instructions used in Redcode:

- `DAT` -- data (kills the process)
- `MOV` -- move (copies data from one address to another)
- `ADD` -- add (adds one number to another)
- `SUB` -- subtract (subtracts one number from another)
- `MUL` -- multiply (multiplies one number with another)
- `DIV` -- divide (divides one number with another)
- `MOD` -- modulus (divides one number with another and gives the remainder)
- `JMP` -- jump (continues execution from another address)
- `JMZ` -- jump if zero (tests a number and jumps to an address if it's 0)
- `JMN` -- jump if not zero (tests a number and jumps if it isn't 0)
- `DJN` -- decrement and jump if not zero (decrements a number by one, and jumps unless the result is 0)
- `SPL` -- split (starts a second process at another address)
- `CMP` -- compare (same as `SEQ`)
- `SEQ` -- skip if equal (compares two instructions, and skips the next instruction if they are equal)
- `SNE` -- skip if not equal (compares two instructions, and skips the next instruction if they aren't equal)
- `SLT` -- skip if lower than (compares two values, and skips the next instruction if the *first* is lower than the *second*)
- `LDP` -- load from p-space (loads a number from private storage space)
- `STP` -- save to p-space (saves a number to private storage space)
- `NOP` -- no operation (does nothing)

Don't worry if some of them seem, to put it mildly, weird. As I said, Redcode is quite a bit different from more ordinary assembly languages, which results from its abstract nature.

The Imp

The truth is, the most important parts of Redcode are the easiest ones. Most of the basic warrior types were invented before the new instructions and modes existed. The simplest, and probably the first, Core War program is the *Imp*, published by A. K. Dewdney in the original 1984 Scientific American article that first introduced Core War to the public.

```
        MOV 0, 1
```

Yes, that's it. Just one lousy `MOV`. But what does it *do*? `MOV` of course copies an instruction. You should recall that all addresses in Core War are relative to the current instruction, so the Imp in fact copies itself to the instruction just after itself.

```
        MOV 0, 1        ; this was just executed
        MOV 0, 1        ; this instruction will be executed next
```

Now, the Imp will execute the instruction it just wrote! Since it's exactly the same as the first one, it will once again copy itself one instruction forward, execute the copy, and continue to move forward while filling the core with `MOV`s. Since the core has no

actual end, the Imp, after filling the whole core, reaches its starting position again and keeps on running happily in circles *ad infinitum*.

So the Imp actually creates it's own code as it executes it! In Core War, self-modification is a rule rather than an exception. You need to be effective to be successful, and that nearly always means changing your code on the fly. Luckily, the abstract environment makes this a lot easier to follow than in ordinary assembly.

BTW, it should be obvious that there are no caches in Core War. Well, actually the *current* instruction is cached so you can't modify it *in the middle* of executing it, but maybe we should leave all that for the later...

The Dwarf

The Imp has one little drawback as a warrior. It won't win too many games, since when it overwrites another warrior, it too starts to execute the `MOV 0, 1` and becomes an imp itself, resulting in a tie. To kill a program, you'd have to copy a `DAT` over its code.

This is just what another classic warrior by Dewdney, the *Dwarf*, does. It "bombs" the core at regularly spaced locations with `DAT`s, while making sure it won't hit itself.

```
        ADD #4, 3       ; execution begins here
        MOV 2, @2
        JMP -2
        DAT #0, #0
```

Actually, this isn't precisely what Dewdney wrote, but it works exactly the same way. The execution begins again at the first instruction. This time it's an `ADD`. The `ADD` instruction adds the source and the destination together, and puts the result in the destination. If you're familiar with other assembly languages, you may recognise the `#` sign as a way of marking immediate addressing. That is, the `ADD` adds the number 4 to the instruction at address 3, instead of adding the instruction 4 to the instruction 3. Since the 3rd instruction after the `ADD` is the `DAT`, the result will be:

```
        ADD #4, 3
        MOV 2, @2       ; next instruction
        JMP -2
        DAT #0, #4
```

If you add two instructions together, both the A- and the B-fields will be added together independently of each other. If you add a single number to an instruction, it will by default be added to the B-field. It's quite possible to use a `#` in the B-field of the `ADD` too. Then the A-field would be added to the B-field of the `ADD` itself.

The immediate addressing mode may seem simple and familiar, but the new modifiers in the ICWS '94 standard will give it an entirely new twist. But let's look at the Dwarf first.

The `MOV` once again presents us with yet another addressing mode: the `@` or the indirect

addressing mode. It means that the DAT will not be copied on itself as it seems (what good would that be?), but on the instruction its B-field points to, like this:

```
        ADD #4, 3
        MOV 2, @2  ;  --.
        JMP -2     ;    | +2
        DAT #0, #4 ; <--' --. The B-field of the MOV points here.
        ...                |
        ...                | +4
        ...                |
        DAT #0, #4 ; <------' The B-field of the DAT points here.
```

As you can see, the DAT will be copied 4 instructions ahead of it. The next instruction, JMP, simply makes the process jump two instructions backwards, back to the ADD. Since the JMP ignores its B-field I've left it empty. The MARS will initialise it for me as a 0.

By the way, as you see the MARS will not start tracing further chains of indirect addresses. If the indirect operand points to an instruction with a B-field of, say, 4, the actual destination will be 4 instructions after it, regardless of the addressing mode.

Now the ADD and the MOV will be executed again. When the execution reaches the JMP again, the core looks like this:

```
        ADD #4, 3
        MOV 2, @2
        JMP -2          ; next instruction
        DAT #0, #8
        ...
        ...
        ...
        DAT #0, #4
        ...
        ...
        ...
        DAT #0, #8
```

The Dwarf will keep on dropping DATs every 4 instructions, until it has looped around the whole core and reached itself again:

```
        ...
        DAT #0, #-8
        ...
        ...
        ...
        DAT #0, #-4
        ADD #4, 3       ; next instruction
        MOV 2, @2
        JMP -2
        DAT #0, #-4
        ...
        ...
        ...
        DAT #0, #4
        ...
```

Now, the `ADD` will turn the `DAT` back into `#0, #0`, the `MOV` will perform an exercise in futility by copying the `DAT` right where it already is, and the whole process will start again from the beginning.

This naturally won't work unless the size of the core is divisible by 4, since otherwise the Dwarf would hit an instruction from 1 to 3 instructions behind the `DAT`, thus killing itself. Luckily, the most popular core size is currently 8000, followed by 8192, 55400, 800, all of them divisible by 4, so our Dwarf should be safe

As a side note, including the `DAT #0, #0` in the warrior wouldn't really have been necessary; The instruction the core is initially filled with, which I've written as three dots (...) is actually `DAT 0, 0`. I'll continue to use the dots to describe empty core, since is it shorter and easier to read.

The addressing modes

In the first versions of Core War the only addressing modes were the immediate (`#`), the direct (`$` or nothing at all) and the B-field indirect (`@`) addressing modes. Later, the predecrement addressing mode, or `<`, was added. It's the same as the indirect mode, except that the pointer will be decremented by one before the target address is calculated.

```
        DAT #0, #5
        MOV 0, <-1      ; next instruction
```

When this `MOV` is executed, the result will be:

```
        DAT #0, #4 ;  ---.
        MOV 0, <-1 ;     |
        ...        ;     | +4
        ...        ;     |
        MOV 0, <-1 ; <---'
```

The ICWS '94 standard draft added four more addressing modes, mostly to deal with A-field indirection, to give a total of 8 modes:

- `#` -- immediate
- `$` -- direct (the `$` may be omitted)
- `*` -- A-field indirect
- `@` -- B-field indirect
- `{` -- A-field indirect with predecrement
- `<` -- B-field indirect with predecrement
- `}` -- A-field indirect with postincrement
- `>` -- B-field indirect with postincrement

The postincrement modes are similar to the predecrement, but the pointer will be *incremented* by one *after* the instruction has been executed, as you might have guessed.

```
        DAT #5, #-10
        MOV -1, }-1      ; next instruction
```

will after execution look like this:

```
        DAT #6, #-10 ;  --.
        MOV -1, }-1  ;    |
        ...          ;    |
        ...          ;    | +5
        ...          ;    |
        DAT #5, #-10 ; <--'
```

One important thing to remember about the predecrement and postincrement modes is that the pointers will be in-/decremented even if they're not used for anything. So `JMP -1, <100` would decrement the instruction 100 even if the value it points to isn't used for anything. Even `DAT <50, <60` will decrement the addresses in addition to killing the process.

The process queue

If you looked at the instruction table a few chapters above closely, you may have wondered about an instruction named `SPL`. There's certainly nothing like that in any ordinary assembly language...

Quite early in the history of Core War, it was suggested that adding multitasking to the game would make it much more interesting. Since the rough time-slicing techniques used in ordinary systems wouldn't fit in the abstract Core War environment (most importantly, you'd need an OS to control them), a system was invented whereby each process is executed for one cycle in turn.

The instruction used to create new processes is the `SPL`. It takes an address as a parameter in its A-field, just like `JMP`. The difference between `JMP` and `SPL` is that, in addition to starting execution at the new address, `SPL` *also* continues execution at the next instruction.

The two -- or more -- processes thus created will share the processing time equally. Instead of a single process counter that would show the current instruction, the MARS has a *process queue*, a list of processes that are executed repeatedly in the order in which they were started. New processes created by `SPL` are added just after the current process, while those that execute a `DAT` will be removed from the queue. If all the processes die, the warrior will lose.

It's important to remember that each program has its *own* process queue. With more than one program in the core, they will be executed alternately, *one cycle at a time regardless of the length of their process queue*, so that the processing time will always be divided equally. If program A has 3 processes, and program B only 1, the order of execution will look like:

1. program A, process 1,
2. program B, process 1,

3. program A, process 2,
4. program B, process 1,
5. program A, process 3,
6. program B, process 1,
7. program A, process 1,
8. program B, process 1,

   ...

And finally, a small example of the use of `SPL`. More information will be available in the later chapters.

```
        SPL 0           ; execution starts here
        MOV 0, 1
```

Since the `SPL` points to itself, after one cycle the processes will be like this:

```
        SPL 0           ; second process is here
        MOV 0, 1        ; first process is here
```

After both of the processes have executed, the core will now look like:

```
        SPL 0           ; third process is here
        MOV 0, 1        ; second process is here
        MOV 0, 1        ; first process is here
```

So this code evidently launches a series of imps, one after another. It will keep on doing this until the imps have circled the whole core and overwrite the `SPL`.

The size of the process queue for each program is limited. If the maximum number of processes has been reached, `SPL` continues execution *only at the next instruction*, effectively duplicating the behaviour of `NOP`. In most cases the process limit is quite high, often the same as the length of the core, but it can be lower (even 1, in which case splitting is effectively disabled).

Oh, and as for thruth often being stranger than fiction, I recently came across a web page titled "Opcodes that should've been". Amongst some really absurd ones I found "`BBW` -- Branch Both Ways". As all the opcodes were supposed to be fictitious, I can only conclude that the author wasn't familiar with Redcode..

The instruction modifiers

The most important new thing brought by the ICWS '94 standard wasn't the new instructions or the new addressing modes, but the modifiers. In the old '88 standard the addressing modes alone decide which parts of the instructions are affected by an operation. For example, `MOV 1, 2` always moves a whole instruction, while `MOV #1, 2` moves a single number. (and *always* to the B-field!)

Naturally, this could cause some difficulties. What if you wanted to move only the A- and B-fields of an instruction, but not the OpCode? (you'd need to use `ADD`) Or what if

you wanted to move something from the B-field to the A-field? (possible, but very tricky) To clarify the situation, the instruction modifiers were invented.

The modifiers are suffixes that are added to the instruction to specify which parts of the source and the destination it will affect. For example, `MOV.AB 4, 5` would move the A-field of the instruction 4 into the B-field of the instruction 5. There are 7 different modifiers available:

- `MOV.A` -- moves the A-field of the source into the A-field of the destination
- `MOV.B` -- moves the B-field of the source into the B-field of the destination
- `MOV.AB` -- moves the A-field of the source into the B-field of the destination
- `MOV.BA` -- moves the B-field of the source into the A-field of the destination
- `MOV.F` -- moves both fields of the source into the same fields in the destination
- `MOV.X` -- moves both fields of the source into the *opposite* fields in the destination
- `MOV.I` -- moves the whole source instruction into the destination

Naturally the same modifiers can be used for all instructions, not just for `MOV`. Some instructions like `JMP` and `SPL`, however, don't care about the modifiers. (Why should they? They don't handle any actual data, they just jump around.)

Since not all the modifiers make sense for all the instructions, they will default to the closest one that does make sense. The most common case involves the `.I` modifier: To keep the language simple and abstract no numerical equivalents have been defined for the OpCodes, so using mathematical operations on them wouldn't make any sense at all. This means that for all instructions except `MOV`, `SEQ` and `SNE` (and `CMP` which is just an alias for `SEQ`) the `.I` modifier will mean the same as the `.F`.

Another thing to remember about the `.I` and the `.F` is that the addressing modes too are part of the OpCode, and are not copied by `MOV.F`

We can now rewrite the old programs to use modifiers as an example. The Imp would naturally be `MOV.I 0, 1`. The Dwarf would become:

```
        ADD.AB #4, 3
        MOV.I  2, @2
        JMP    -2
        DAT    #0, #0
```

Note that I've left out the modifiers for `JMP` and `DAT` since they don't use them for anything. The MARS turns them into (for example) `JMP.B` and `DAT.F`, but who cares?

Oh, one more thing. How did I know which modifier to add to which instruction? (and, more importantly, how does the MARS add them if we leave them off?) Well, you can usually do it with a bit of common sense, but the '94 standard does defines a set of rules for that purpose.

`DAT`, `NOP`
: Always `.F`, but it's ignored.

`MOV`, `SEQ`, `SNE`, `CMP`
: If A-mode is immediate, `.AB`,

if B-mode is immediate and A-mode isn't, `.B`,
if neither mode is immediate, `.I`.

`ADD`, `SUB`, `MUL`, `DIV`, `MOD`
If A-mode is immediate, `.AB`,
if B-mode is immediate and A-mode isn't, `.B`,
if neither mode is immediate, `.F`.

`SLT`, `LDP`, `STP`
If A-mode is immediate, `.AB`,
if it isn't, (always!) `.B`.

`JMP`, `JMZ`, `JMN`, `DJN`, `SPL`
Always `.B` (but it's ignored for `JMP` and `SPL`).

Diving deeper into the '94 standard

The # is more than it seems...

The defined behavior of the immediate addressing mode (`#`) in the '94 standard is quite unusual. While the standard is 100% compatible with the old syntax, the immediate addressing has been defined in a very clever and unique way that lets it be used logically with all the instructions and modifiers, and makes it a very powerful tool.

Looking at the modifiers, you might wonder what `MOV.F #7, 10` would do. `.F` should move both fields, but there's only one number in the source?? Would it move 7 into both fields of the destination?

No, it definitely wouldn't. In fact, it would move 7 into the A-field of the destination, and *10* into the B-field! Why?

The reason is that, in the '94 syntax, the source (and the destination) is *always* a whole instruction. In the case of immediate addressing, it's simply always the current instruction, (ie. 0) whatever the actual value. So `MOV.F #7, 10` moves both fields of the source (0) to the destination (10). Surprising, isn't it?

The same even works for `MOV.I`. This way of defining immediate addressing also lets us use instructions which, even without modifiers, wouldn't make sense in the '88 standard such as `JMP #1234`. Obviously you can't jump into a number, but you can jump into the address of that number, or 0. This offers many obvious advantages, since not only can we store data in the A-field for "free", but the code will survive even if someone decrements it. We could now rewrite the earlier imp-making code to be a bit more robust:

```
        SPL    #0, }1
        MOV.I  #1234, 1
```

It still works the same, but now the A-fields are free. Just for fun I've let the `SPL` increment the A-field of the imp, so that all the imps will look different. Since `SPL`

doesn't use its B-field, that increment is also "free". It works, trust me -- or try it yourself!

Modulo math

You should already know that addresses in the core wrap around, so that the instruction one coresize ahead or behind the current instruction refers to the current instruction itself. But in fact, this effect goes much deeper: all numbers in Core War are converted into the range 0 - *coresize*-1.

For those of you who already know about programming and limited-range integer math, let's just say that all numbers in Core War are considered unsigned, with the maximum integer being *coresize*-1. If that didn't clarify everything, read on...

In effect, all numbers in Core War are divided by the length of the core, or *coresize*, and only the remainder is kept. You might try thinking of a calculator with a display of only 8 numbers that throws off any digits past that, so that 100*12345678 (1234567800, of course) is only shown (and stored) as 34567800. Similarly, in a core of 8000 instructions, 7900+222 (8122) becomes only 122.

What happens to negative numbers, then? They are normalised too, by adding *coresize* until they become positive. This means that what I wrote as -1 is actually stored by the MARS as *coresize*-1, or in an 8000 instruction core, as is common, 7999.

Of course, this makes no difference for the addresses, which wrap around anyway. In fact, it doesn't make any difference to the simple math instructions like `ADD` or `SUB` either, since with *coresize*=8000, 6+7998 gives the same result of 4 (or 8004) as does 6-2.

What's the problem, then? Well, there are a few instructions where it makes a difference. Such instructions as `DIV`, `MOD` and `SLT` always treat numbers as unsigned. This means that -2/2 isn't -1, but (*coresize*-2)/2 = (*coresize*/2)-1 (or for *coresize*=8000, 7998/2=3999, not 7999). Similarly, `SLT` considers -2 (or 7998) to be *greater* than 0! In fact, 0 is the lowest possible number in Core War, so all other numbers are considered greater than it.

The '94 standard instruction by instruction

Ok, your patience has been rewarded. Until now I've given you only some separate pieces of information. Now it's time to tie it all together by describing each instruction to you.

Of course I could've listed them at the very beginning, when I shoved you the instruction set, and it probably would've saved you from a lot of guessing. But I had -- at least in my opinion -- a very good reason to wait. Not only did I want to show you some practical code before getting into the boring theoretical stuff, but most of all I wanted you to grasp at least the basic idea of addressing modes and modifiers before describing the instructions in detail. If I had described the instructions before the modifiers, I would've had to first teach you the older '88 rules, and later teach it all

again with modifiers included. It's not a bad way to learn Redcode, but it'd make this guide unnecessarily complicated.

`DAT`

Originally, as its name shows, `DAT` was intended for storing data, just like in most languages. Since in Core War you want to minimise the number of instructions, storing pointers etc. in unused parts of other instructions is common. This means that the most important thing about `DAT` is that executing it kills a process. In fact, since the '94 standard has no illegal instructions, `DAT` is defined as a completely legal instruction, which *removes the currently executing process from the process queue*. Sounds like splitting hairs, maybe, but precisely defining the obvious can often save a lot of confusion.

The modifiers have no effect on `DAT`, and in fact some MARSes remove them. However, remember that predecrementing and postincrementing are always done even if the value isn't used for anything. One unusual thing about `DAT`, a relic of the previous standards, is that if it has only one argument it's placed in the *B-field*.

`MOV`

`MOV` copies data from one instruction to another. If you don't know everything about that already, you should probably re-read the earlier chapters. `MOV` is one of the few instructions that support `.I`, and that's its default behavior if no modifier is given (and if neither of the fields uses immediate addressing).

`ADD`

`ADD` adds the source value(s) to the destination. The modifiers work like with `MOV`, except that `.I` isn't supported but behaves like `.F`. (What would `MOV.AB`+`DJN.F` be?) Also remember that all math in Core War is done modulo coresize.

`SUB`

This instruction works exactly like `ADD`, except for one fairly obvious difference. In fact, all the "arithmetic-logical" instructions work pretty much the same...

`MUL`

...as is the case for `MUL` too. If you can't guess what it does, you've probably missed something very important.

`DIV`

`DIV` too works pretty much the same as `MUL` and the others, but there are a few things to keep in mind. First of all, this is unsigned division, which can give surprising results sometimes. **Division by zero kills the process**, just like executing a `DAT`, and leaves the destination unchanged. If you use `DIV.F` or `.X` to divide two numbers at a time and one of the divisors is 0, the other division will still be done as normal.

`MOD`

Everything I said about `DIV` applies here too, including the division by zero part. Remember that the result of a calculation like `MOD.AB #10, #-1` depends on the size of the core. For the common 8000-instruction core the result would be 9 (7999 mod 10).

`JMP`

`JMP` moves execution to the address its A-field points to. The obvious but important difference to the "math" instructions is that `JMP` only cares about the

address, not the data that address points to. Another significant difference is that `JMP` doesn't use its B-field for anything (and so also ignores its modifier). Being able to jump (or split) into two addresses would simply be too powerful, and it'd make implementing the next three instructions quite difficult. Remember that you can still place an increment or a decrement in the unused B-field, with luck damaging your opponent's code.

`JMZ`

This instruction works like `JMP`, but instead of ignoring its B-field, it tests the value(s) it points to and only jumps if it's zero. Otherwise the execution will continue at the next address. Since there's only one instruction to test, the choice of modifiers is fairly limited. `.AB` means the same as `.B`, `.BA` the same as `.A`, and `.X` and `.I` the same as `.F`. If you test both fields of an instruction with `JMZ.F`, it will jump *only if both fields are zero*.

`JMN`

`JMN` works like `JMZ`, but jumps if the value tested is *not* zero (surprise, surprise...). `JMN.F` jumps if *either of the fields is non-zero*.

`DJN`

`DJN` is like `JMN`, but the value(s) are decremented by one *before* testing. This instruction is useful for making a loop counter, but it can also be used to damage your opponent.

`SPL`

This is the big one. The addition of `SPL` into the language was probably the most significant change ever made to Redcode, only rivalled perhaps by the introduction of the ICWS '94 standard. `SPL` works like `JMP` but the execution *also* continues at the next instruction, so that the process is "split" into two new ones. The process at the next instruction executes *before* the one which jumped to a new address, which is a small but *very* important detail. (Many, if not most, modern warriors wouldn't work without it!) If the max. number of processes has been reached, `SPL` works like `NOP`. Like `JMP`, `SPL` ignores its B-field and its modifier.

`SEQ`

`SEQ` compares two instructions, and skips the next instruction if they are equal. (It always jumps only those two instructions forward, since there's no room for a jump address.) Since the instructions are compared only for equality, using the `.I` modifier is supported. Quite naturally, with the modifiers `.F`, `.X` and `.I` the next instruction will be skipped only if *all* the fields are equal.

`SNE`

Ok, you guessed it. This instruction skips the next instruction if the instructions it compares are not equal. If you compare more than one field, the next instruction will be skipped if *any* pair of them aren't equal. (Sounds familiar, doesn't it? just like with `JMZ` and `JMN`...)

`CMP`

`CMP` is an alias for `SEQ`. This was the only name of the instruction before `SEQ` and `SNE` were introduced. Nowadays it doesn't really matter which name you use, since the most popular MARS programs recognise `SEQ` even in '88 mode.

`SLT`

Like the previous instructions, `SLT` skips the next instruction, this time if the first value is lower than the second. Since this is an arithmetical comparison instead of a logical one, it makes no sense to use `.I`. It might seem that there should be

an instruction called SGT, (*skip if greater than*) but in most cases the same effect can be achieved simply by swapping the operands of SLT. Remember that all values are considered unsigned, so 0 is the smallest possible number and *-1 is the largest*.

NOP

Well, this instruction does nothing. (And it does it really well, too.) It's almost never used in an actual warrior, but it's very useful in debugging. Remember that any in- or decrements are still evaluated.

You might notice that two instructions, namely LDP and STP are missing. They are a fairly recent addition to the language, and will be discussed... um, well right now. :-)

P-space -- the final frontier

P-space is the latest addition to Redcode, introduced by pMARS 0.8. The "P" stands for private, permanent, personal, pathetic and so on, whichever you like. Basically, the P-space is an area of memory which only your program can access, and which survives between rounds in a multi-round match.

The P-space is in many ways different from the normal core. First of all, each P-space location can only store one number, not a whole instruction. Also, the addressing in P-space is absolute, ie. the P-space address 1 is always 1 regardless of where in the core the instruction containing it is. And last but not least, the P-space can only be accessed by two special instructions, LDP and STP.

The syntax of these two instructions is a bit unusual. The STP, for example has an ordinary value in the core as its source, which is put into the P-space field pointed to by the destination. So the P-space location isn't determined by the destination *address*, but by its *value*, ie. the value that would be overwritten if this were a MOV.

So STP.AB #4, #5 for example would put the *value* 4 into the *P-space field* 5. Similarly,

```
        STP.B  2, 3
        ...
        DAT    #0, #10
        DAT    #0, #7
```

would put the value 10 into the P-space field **7**, not 3! This can get pretty confusing if the STP itself uses indirect addressing, which leads into a sort of "double-indirect" addressing system.

LDP works the same way, except that now the source is a P-space field and the destination a core instruction. The P-space location 0 is a special read-only location. Any writes to it will be ignored, and it is initialised to a special value before each round. This value is -1 for the first round, 0 the program died in the previous round, and otherwise the number of surviving programs. This means that, for one-on-one matches, 0 means a loss, 1 a win and 2 a tie.

The size of the P-space is usually smaller than that of the core, typically 1/16 of the

core size. The addresses in the P-space wrap around just like in the core. The size of the P-space must naturally be a factor of the core size, or something weird will happen.

There is one little peculiarity in the pMARS implementation of P-space. Since the intention was to keep access to P-space slow, loading or saving two P-space fields with one instruction isn't allowed. This is a Good Thing, but the result is at the very least a kludge. What this actually means is that `LDP.F`, `.X` and `.I` all work like `LDP.B`! (and the same for `STP` too, of course)

Absolutely the most common use of P-space is to use it to select a strategy. In its simplest form, this means saving the previous strategy in P-space, and switching strategies if the P-space field 0 shows the program lost last time. This kind of programs are called P-warriors, P-switchers or P-brains (pronounced *pea-brains*).

Unfortunately, the P-space isn't as private as it seems. While your opponent can't read or write your P-space directly, your processes may be captured and made execute your opponents code, including `STP`s. This kind of technique is known as brainwashing, and all P-switchers must be prepared for it, and not freak out if the strategy field contains something weird.

The parser

Labels and addresses

So far, I've written all the addresses in our example programs as instruction numbers, relative to the current instruction. But in larger programs, this can get annoying, not to mention difficult to read. Luckily, we don't really have to do this, since Redcode lets us use labels, symbolic constants, macros and all the other things you'd expect of a good assembler. All we need to do is to label the instructions an refer to them with the labels, and the parser calculates the real addresses for us, like this:

```
imp:    mov.i   imp, imp+1
```

Whoa, what happened? This is exactly the same program as the one I showed you in the very beginning. I've just replaced the numerical addressed with references to a label, "imp". Of course, in this case doing that is pretty futile. The only instruction in which the label is used is "imp" itself, in which the label is replaced by 0.

Before executing it, the parser in the MARS converts all such labels and other symbols into the familiar numbers. Such a "pre-compiled" Redcode file is called a *load file*, for whatever reason. All MARSes must be able to read load files, but some may not have a real parser. In load file format, the previous code becomes `MOV.I 0, 1`. We could've also written the same code as

```
imp:    mov.i   imp, next
next:   dat     0, 0            ; or whatever
```

In this case, the instruction labelled "next" is one instruction after "imp", so it's replaced by 1. Remember that the real addresses are still relative numbers, so the Imp will continue to be `MOV.I 0, 1` even after it has copied itself forward over "next".

Actually, the `:` in the end of the labels isn't really necessary. I've used it here to help you see where the labels are, but I usually don't use it in my own programs. It's a matter of taste.

Oh, and just in case you're wondering about it, Redcode instructions are case-insensitive. I like using lower case for the sources since it looks nicer, and upper case only for the compiled, "load file" format (mostly because it's a tradition).

The whole thing

While the examples in previous chapters might compile just fine, they're not really complete programs, but parts of one. A typical redcode file contains some extra information for the MARS.

```
;redcode-94
;name Imp
;author A.K. Dewdney

        org     imp

imp:    mov.i   imp, imp+1
        end
```

As you probably have already figured out, everything after a `;` is a comment in Redcode. The lines on the top of this program, however, aren't just ordinary comments. The MARS uses them to get some information about the program.

The first line, `;redcode-94`, tells the MARS that this really is a Redcode file. Anything above this line is ignored by the MARS. Actually, the MARS only expects a line *starting* with `;redcode`, but we can use the rest of the line to identify the flavor of Redcode used. Specially, the [KotH servers] read this line themselves, and use it to identify the hill the program is going to.

The `;name` and `;author` lines just give some information on the program. Of course you could give it in any format, but using the specific codes lets the MARS read the names and display them when the program is run.

The line with the word `END` -- surprise, surprise -- ends the program. Anything after it will be ignored. Together with `;redcode`, it can be use for example to include Redcode programs in e-mail.

The line with `ORG` tells where the execution of the program should start. This lets us put other instructions before the beginning of the program. The `ORG` command is one of the new things included in the '94 standard. The older syntax, which still works in modern programs too, is to give the starting address as an argument to the `END`.

```
;redcode-94
;name Imp
;author A.K. Dewdney

imp:    mov.i   imp, imp+1

        end     imp
```

Simple, compact, and unfortunately quite illogical. And with long programs, you have to scroll to the end just to see where it begins. In Redcode terminology, both `ORG` and `END` are called *pseudo-OpCodes*. They look like actual instructions, but they're not actually compiled into the program.

But enough of the Imp. Let's see what the Dwarf would look like in modern Redcode:

```
;redcode-94
;name Dwarf
;author A.K. Dewdney
;strategy Bombs the core at regular intervals.
;(slightly modified by Ilmari Karonen)
;assert CORESIZE % 4 == 0

        org     loop

loop:   add.ab  #4, bomb
        mov.i   bomb, @bomb
        jmp     loop
bomb:   dat     #0, #0

        end
```

The labels make understanding the program a lot easier, don't they? Notice that I've added two new comment lines. The `;strategy` line describes the program briefly. There may be several such lines in the program. Most current MARSes ignore them, so you might as well use ordinary comments like the one my name is in, but the Hills display the `;strategy` lines to others. Sending the previous program to one, something like this might be shown:

```
A new challenger has appeared on the '94 hill!

Dwarf by A.K. Dewdney: (length 4)
;strategy Bombs the core at regular intervals.

[other info here...]
```

The environment and ;assert

Another new detail in our example code is the `;assert` line. It can be used to make sure the program really works with the current settings. The Dwarf, for example, kills itself if the size of the core isn't evenly divisible by 4. So, I've used the line `;assert CORESIZE % 4 == 0` to make sure it always is.

The *CORESIZE* is a predefined constant which tells us the size of the core. That is,

n+*CORESIZE* is always the same address as *n*. The `%` is the modulus operator, which gives the remainder in a division. The syntax of the expressions used in the `;assert` lines and elsewhere in Redcode is the *same as in the C language*, although the set of operators is much more limited.

For those who don't know C, here's some sort of a list of the operators which are used in Redcode expressions:

Arithmetic:
- `+` addition
- `-` subtraction (or negation)
- `*` multiplication
- `/` division
- `%` modulus (remainder)

Comparison:
- `==` equals
- `!=` doesn't equal
- `<` is less than
- `>` is greater than
- `<=` is equal or less than
- `>=` is equal or greater than

Logical:
- `&&` and
- `||` or
- `!` not

Assignment:
- `=` assignment to a variable

The `;assert` is followed by a logical expression. If it's false, the program will not be compiled. In C, a value of 0 means false and anything else means true. The logical and comparison operators return 1 for true, a fact which can be useful later.

Typically, `;assert` is used to check that the size of the core is the one the constants have been designed for, like `;assert CORESIZE == 8000`. If the program uses P-space, its existence may be tested with `;assert PSPACESIZE > 0`. Since our example, the Dwarf, is fairly adaptable, I only tested the *CORESIZE* for divisibility, not for a specific size. The Imp, which runs with *any* settings, could use `;assert 1`, `;assert 0 == 0` and so on, all of which always evaluate as true. This is useful since otherwise the MARS may complain about a "`missing ;assert line -- warrior may not work with current settings.`"

Some of the predefined constants, such as *CORESIZE*, are defined by the '94 standard, and others may and have been added. pMARS 0.8 should support at least the following:

- *CORESIZE* -- the size of the core (default 8000)
- *PSPACESIZE* -- the size of the P-space (default 500)
- *MAXCYCLES* -- the number of cycles until a tie is declared (default 80000)
- *MAXPROCESSES* -- the maximum size of the process queue (default 8000)
- *WARRIORS* -- the number of programs in the core (usually 2)

- *MAXLENGTH* -- the maximum length of a program (default 100)
- *CURLINE* -- the number of instructions compiled so far (0 to *MAXLENGTH*)
- *MINDISTANCE* -- the minimum distance between two warriors (default 100)
- *VERSION* -- the version of pMARS, multiplied by 100 (80 or more)

#define? Well, almost...

The predefined constants are useful, and so are labels, but is that really all? Can't I use some variables or something?

Well, Redcode is an assembly language, and they don't really use a lot of variables. But there's something almost as good, or maybe sometimes even better. There's a pseudo-OpCode `EQU` that lets us define our own constants, expressions and even macros. It looks like this:

```
step    equ     2667
```

After this, *step* is always replaced by 2667. There's a catch, however. The replacement is textual, not numerical. In this case it shouldn't do any harm, but while it makes `EQU` a very powerful tool, it can create some problems which C programmers should be quite familiar with. Let's take an example.

```
step    equ     2667
target  equ     step-100

start   mov.i   target, step-target
```

The A-field of the `MOV` would be 2567, just as it should be. But the B-field would become 2667-2667-100 == -100, not 2667-(2667-100) == 2667-2567 == 100, as it was probably intended. The solution is simple. Just put parentheses around every expression in `EQU`s, such as "`target equ (step-100)`".

With the modern versions of pMARS it's possible to use multi-line `equ`s, and thus create some sort of macros. The way it's done is this:

```
dec7    equ     dat #1, #1
        equ     dat $1, $1
        equ     dat @1, @1
        equ     dat *1, *1
        equ     dat {1, {1
        equ     dat }1, }1
        equ     dat <1, <1

decoy   dec7
        dec7            ; 21-instruction decoy
        dec7
```

What's "rof" used for?

There are a few more features of the pMARS parser left, and this one is perhaps more

powerful (and harder to learn) than any of the above. The FOR/ROF pseudo-OpCodes not only can make your sources shorter and create complex code sequences easily, but they can be used to create conditional code for different settings.

A FOR block begins with -- yes, you guessed it -- the pseudo-OpCode FOR, followed by the number of times the block should be repeated. If there's a label before the block, it will be used as a loop counter, like this:

```
index   for     7
        dat     index, 10-index
        rof
```

The block ends, as you can see, with ROF. (Much better that the old cliché NEXT or REPEAT, I'd say.) The block above would be parsed by pMARS into:

```
        DAT.F   $1, $9
        DAT.F   $2, $8
        DAT.F   $3, $7
        DAT.F   $4, $6
        DAT.F   $5, $5
        DAT.F   $6, $4
        DAT.F   $7, $3
```

It's quite possible to have several FOR blocks inside each other. The blocks can even contain EQUs inside them, which lets us create some very interesting code. An even more useful feature is that the loop counter can be joined to a label with the &-operator. This is most commonly used to avoid declaring labels twice, but it can be useful for various other purposes as well.

```
dest01  equ     1000
dest02  equ     1234
dest03  equ     1666
dest04  equ     (CORESIZE-1111)

jtable
ix      for     4
jump&ix spl     dest&ix
        djn.b   jump&ix, #ix
        rof
```

This would, after the FOR/ROF is parsed, become:

```
jtable
jump01  spl     dest01
        djn.b   jump01, #1
jump02  spl     dest02
        djn.b   jump02, #2
jump03  spl     dest03
        djn.b   jump03, #3
jump04  spl     dest04
        djn.b   jump04, #4
```

As for what this would be useful for, well, that's up to your own imagination. The only

warriors I've seen using such complex expressions are some quickscanners. The predefined constants can also be used with `FOR`/`ROF`, like this:

```
; The main warrior body is here

decoy
foo     for     (MAXLENGTH-CURLINE)
        dat     1, 1
        rof

        end
```

This fills the remaining space in your warrior with `DAT 1, 1`. Such a decoy can misdirect other warriors' attacks, provided that you've copied (*booted*) your own program away from the decoy. Note that I've used *foo* as a loop counter even though it isn't used for anything. That's because otherwise the MARS would consider *decoy* to be a loop counter instead of the label it should be.

Finally, here's an example of some more creative ways of using `FOR`/`ROF`:

```
;redcode-94
;name Tricky
;author Ilmari Karonen
;strategy Some really complex warrior thingy
;strategy (A self-explanatory example of conditional code)
;assert CORESIZE == 8000 || CORESIZE == 800
;assert MAXPROCESSES >= 256 && MAXPROCESSES < 10000
;assert MAXLENGTH >= 100

        org     start

        for     0
This is a for/rof comment block.  This will be repeated 0 times, which
means that everything here will be ignored by the MARS.  This is a
perfect place for explaining the complex strategy this warrior uses.
        rof

;Of course, using ordinary comments is also possible.  You can use
;whichever alternative you like.

        for     (CORESIZE == 8000)
step    equ     normalstep
;Since a true comparison returns 1 and a false one 0, this piece of
;code is only compiled if the comparison is true.
        rof

        for     (CORESIZE == 800)
step    equ     tinystep
;And here we can put optimized constants for the smaller core size.
        rof

        for     0
;strategy Since strategy and assert lines are really comments, they
;strategy will be parsed even inside FOR 0 / ROF blocks!
        rof
```

```
;[Actual code here..]
```

Variety with variables

The problem with the constants defined with `EQU` is that they're, well, constants. Once you've defined them, you can't change their values. This is fine for most purposes, but it makes a few tricks damn near impossible.

Luckily pMARS provides a few real variables for us to use. Their use is a bit tricky and it's been a long time since I've seen anyone really using them, but they do exist.

The variable names have only one letter, effectively limiting their number to 26 (*a* through *z*). Instead of using `EQU`, the variables are assigned their values with the `=` operator. The tricky bit is that, to use the operator, one has to have an expression. And since pMARS does not recognize the comma operator, it may be necessary to write dummy expressions.

Still, the variables can be useful. For example, the following auto-generated Fibonacci sequence would probably be impossible without them.

```
        dat     #1, (g=0)+(f=1)
idx     for     15
        dat     #idx+1, (f=f+g)+((g=f-g) && 0)
        rof
```

Note how the expression `(g=f-g)` is "hidden" by ANDing it with 0. The system works because pMARS won't reorder the expression but always evaluates the left side of addition first, so that when the right side is being computed, *f* has already been increased.

PINs and needles

Okay, I almost forgot. There's one more pseudo-Op left to describe. It's almost never used, but yes, it's there. The `PIN` stands for "P-space Identification Number". If two programs have the same number as their `PIN`, they will share their P-space. This can be used to provide a sort of inter-process communication and even cooperation. Unfortunately the strategy doesn't seem to be worth the trouble it takes to create an affective and fast method of communication. Of course, if you want to try it, go ahead. You never know if it'll be a success...

If the program has no `PIN`, their P-space will always be private. Even if two programs do share their P-space, the special read-only location 0 is always private.

Climbing the hill

If you didn't already know about them, the *King of the Hill* servers (often called just *hills*) are continuous Core War tournaments on the Internet. Warriors are sent by e-mail -- or entered on a web form -- to the server, which pits them agains all the (usually 10-30) programs already on the hill. The program with the lowest total score

falls off the hill, and the new warrior will replace it (assuming it got a better score than at least one of the original programs). There are also quite a few variations of this basic setup around, like "infinite" hills, diversity hills, etc.

Note that the hills typically pre-compile the warriors into load files before actually running them to save time. This can lead to some of the predefined constants, such as *WARRIORS*, being incorrect, and thus to mysterious `;assert` problems.

There are currently (April 2012) two main KotH servers available:

KotH.org
: The oldest and most famous currently active KotH server. Currently hosts 7 hills with different settings, including two multiwarrior melee hills and two hills using the older Redcode '88 standard.

KOTH@SAL
: Also hosts 7 hills with different parameters, including a Beginners' hill where warriors are automatically pushed off after surviving 50 challenges to make it easier for new players to make it onto the hill.

Also, the Koenigstuhl server hosts 10 "infinite" hills for published warriors. Warriors sent to these hills never get pushed off, so the hills keep getting bigger and bigger. The Koenigstuhl also uses a recursive scoring argorithm that adjusts a warrior's contribution to the scores based on its ranking.

The list above is not meant to be comprehensive, and is likely to become outdated. A more detailed and up-to-date list of active KotH servers can (currently) be found at the corewar.info pages.

History

v. 0.50
: Finished the chapter on the parser. (March 25, 1997)

v. 0.51
: Fixed a bug in the for/rof examples

v. 0.52
: *The first published version*

v. 0.53
: Fixed some typos and misspellings

v. 0.54
: Added the '88 -> '94 conversion rules

v. 0.55
: Cleaned up the HTML a bit

v. 1.00
: Added info on the `=` operator. Might as well call this thing "version 1". (May 5, 1997)

v. 1.01
: Fixed a minor incompatibility with `<DD>` tags.

v. 1.02
: Fixed some typos and illogical sentences. Changed the navigation bar to have a common style with the rest of the site.

v. 1.03
: Removed some images and align attributes, changed doctype to Strict.

v. 1.10
: *Aargh!* I've got SLT backwards all this time! Fixed. (March 8, 1998)

v. 1.20
: Rewrote much of Climbing the hill to reflect the current situation. Made some other minor changes in the process. Moved the document to a new address at vyznev.net. Switched to a Creative Commons license. (October 7, 2003)

v. 1.21
: Added colors (for CSS-enabled browsers)! Made some more minor changes and typo fixes. Chose a standard spelling and capitalization for the title. This is the first published version at vyznev.net. (April 11, 2004)

v. 1.22
: Updated the license from CC-By-NC 2.0 to CC-By 3.0, removing the restrictions on commercial use. Removed the page move notification box. Updated the KotH server list again, removing defunct hills. No other content changes. (April 16, 2012)

Copyright