Robot Battle

Version 1.1

by Toby Smith and Charlie Moylan

Blue Cow Software
6656 Ridgeville St.
Pittsburgh, PA 15217

Contacting us electronically:
Internet: bluecow@unix.cis.pitt.edu
CompuServe: 76174,2447
American Online: TOBYS3

Manual by Toby Smith

Introduction:

Welcome to Robot Battle. Robot Battle is a game in which pre-programmed robots enter a battlefield and slug it out to the death. The object of the game is for players to out-program their opponents, leading their robot to victory.

The Program:

Robot Battle consists of a single, MultiFinder-friendly application and an accompanying data file which contains information the program needs to run. The data file, called RBData, must be located in the same folder as the Robot Battle application itself. The program is not copy-protected in any way, but attempting to play it without this manual is a true exercise in futility.

Two versions of Robot Battle are included in this package. One runs in black and white, will run on just about anything (so far as we know), and is smaller. The other is larger and runs in 16 colors. The color version is also meant to be played on machines with a 68020 or greater.

In the color version, the white robot is gold-colored, and the black robot is blue. We here at Blue Cow Software are partial to the blue one, of course...

Ye Olde Money-Grovelling Section:

Here's where we tell you why you should be sending us your hard-earned cash. Let me first start by telling you why Robot Battle is being released electronically.

The program you have here is a successor to a Microsoft BASIC program that was written way back when when people with 512K of RAM and two floppy drives were power-users. I posted that original version of Robot Battle on CompuServe and received a favorable response. Next came a compiled C version of the same program, with some small changes. After that I planned on letting Robot Battle die gracefully. But then I wanted to do just one more thing to it. And maybe change the interface. And maybe add some real graphics (instead of black and white circles that were supposed to be robots). And then Charlie became involved. And suddenly Robot Battle was a part of my life again, like it or not. Well, five years later, Robot Battle bears absolutely no resemblance to the original program other than in concept. And we look at the program and think "Gee, maybe someone else would like to play this." So here we are. Back to the question of why RB is being published electronically, I have three answers:

1. We could have gone through a publisher, but then we'd be receiving a measly share of a measly amount of money and Robot Battle would be forever associated with the name of a publisher, rather than the names of the authors. This isn't what we wanted.

2. Robot Battle is a program that should receive a lot of user feedback. By distributing the game electronically, by the time you send your check you'll most likely have suggestions for improvements or reports of bugs. We're genuinely concerned that you enjoy this program and will listen to all reasonable requests (note the number of additions to version 1.1!).

3. By not having any packaging costs (other than for registered users) we can offer the program for substantially less than we'd be able to through other channels.

Why should you pay at all? There are several good reasons.

1. The distributed version of Robot Battle is crippled. The program is fully playable, but only the laser weapon is operational. The two most interesting weapons, missiles and the sonic beam, are present but don't do any damage. These two weapons add a completely new dimension to the program. By at least being able to use them (even though they don't hurt your opponent) you can get a feel for what the full version is like.

2. There is an annoying dialog box that appears at the start of every game. This isn't present in the registered version.

3. Future versions of Robot Battle may cost more. Paying the price of the program entitles you to free upgrades, so the earlier you register, the better.

4. You'll make us happy, and we make more games when we're happy. An example of our happiness is Strike Jets, which should be available from the same place you got Robot

Battle by the time you read this. Strike Jets is a strategic simulation of aerial combat in the modern age. Detailed data for over 90 aircraft is included in the full version, along with many built-in scenarios. Users can even create their own scenarios.

If you've read all of this, thanks. See the About Robot Battle... menu item for pricing and ordering information.

Playing the Game:

The first thing that the budding Robot Battler must do is create a robot program. This is done by simply creating a text file which contains valid robot code. Robot program files can be created with any standard word-processing program which can save files as "text only." The Robot Battle application includes a built-in text editor so that robot files can be created and modified from within the program.

Robots are programmed using a simple, BASIC-like language called RIPPLE. RIPPLE is a very easy-to-understand language, so even a complete programming neophyte should be able to send mighty steel warriors into the arena with a minimum of effort. See the RIPPLE section below for details.

After you have created your electronic masterpiece, you send it into battle. The robot code is checked for syntactic errors and compiled. If the program contains errors, Robot Battle will complain and bring up a text window containing your code so that you can fix it and try again. If your RIPPLE code is correct, your robot program will be loaded and you're ready to do battle.

Battles take place between two robots, one white, one black. They are functionally identical. Battles take place after valid programs have been loaded into both robots. The user then starts the battle and watches the contest. During the battle, the user can move the closeup view around, watching different aspects of the battle. The dimensions of the battlefield is 75 x 75 units. (Note that the robots are larger than one grid square; they are 4x4. This means that if you try to move your robot 75 spaces in one direction, you're going to hit a wall.) If the action is moving too quickly, the robots can be executed line-by-line in Step Mode. When in Step mode, the game will pause after each robot has executed a move. To continue to the next step, simply click the mouse or press a key. In addition, "comprehensive debugging" windows can be viewed which display the current status of all the robots' internal debugging variables, along with the current program line being executed by each robot. This is extremely handy for tracking down those annoying bugs.

Program Mechanics:

Using the Robot Battle program should be a straightforward exercise for anyone familiar with the Macintosh environment. The program is run by double-clicking on the Robot Battle icon. A title screen will appear. After the mouse button is clicked the title screen will disappear and the main Robot Battle window will appear. This window can be moved anywhere on the screen, but cannot be resized.

Other than the actual typing required to enter RIPPLE code, Robot Battle is completely

mouse-driven. All commands are accessed through the five menus which appear in the menubar.

Apple Menu:

This menu contains any desk accessories that you have installed in your system. In addition, there is an "About Robot Battle" item which will display our company's wonderful logo along with the game credits.

File Menu:

This menu contains standard commands which are relevant to the built-in text editor. With the exception of Quit, none of these commands need be accessed if the text editor isn't being used.

Edit Menu:

This menu contains the standard Mac Cut, Copy and Paste commands. These are used by the built-in text editor and some desk accessories.

Battle Menu:

This menu is used for controlling the battle. The first item, New Battle, resets both the black and white robots to their original states, i.e. empty and devoid of programs.

The next two items load robot programs into the black and white robots. Selecting one of these menu items brings up a standard Mac dialog box, prompting the user for a robot file to compile. Robot Battle will only load TEXT files. After a file is selected, the program will be loaded into the selected robot. If the file is successfully loaded, the name of the robot will appear on the main screen above the vital stats for the robot. If the program is not loaded successfully, the text editor window will appear with the file in it.

The final two items control the battle itself. Start Battle is greyed out until both robots contain valid programs. Once a battle is begun, the Start Battle item changes to Pause. Selecting this while a battle is in progress will halt the action until the item is selected again.

The final item in the Battle menu is Restart Same Robots. This allows the user to interrupt an ongoing battle and start over from scratch. This is a useful command to employ if your robot is getting unexpectedly trounced by a much-inferior program.

Option Menu:

This menu contains commands relevant to changing the Robot Battle environment. The first item, Sound On, allows you to the sounds on and off.

The second item allows the user to control the number of humans that appear on the battlefield with the robots. Humans are annoying little things which run towards the robots. They sometimes get in the way of laser fire, and have the annoying habit of lobbing grenades at robots which stay in one place for too long a time.

MultiFinder note: if Robot Battle is put in the background while a battle is in progress, the battle will continue to run, albeit at a much slower speed. If you don't want the battle to continue while you perform other tasks, select Pause before switching to another application. When Robot Battle is brought to the foreground, the battle will resume its normal speed. Sounds are turned off automatically when Robot Battle is in the background.

In addition to the menu commands, a few functions can be performed by clicking on the main screen.

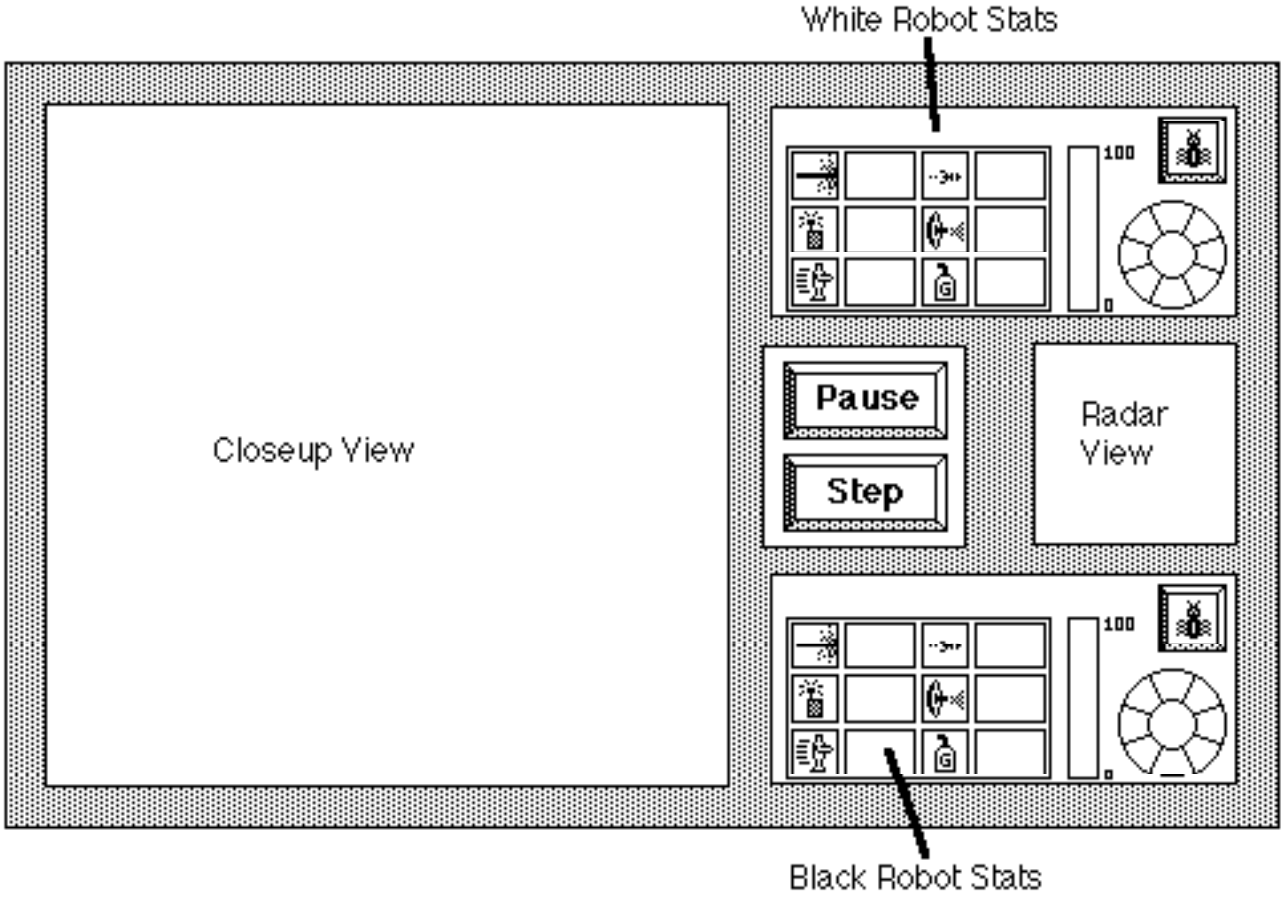


Figure 1. Main screen

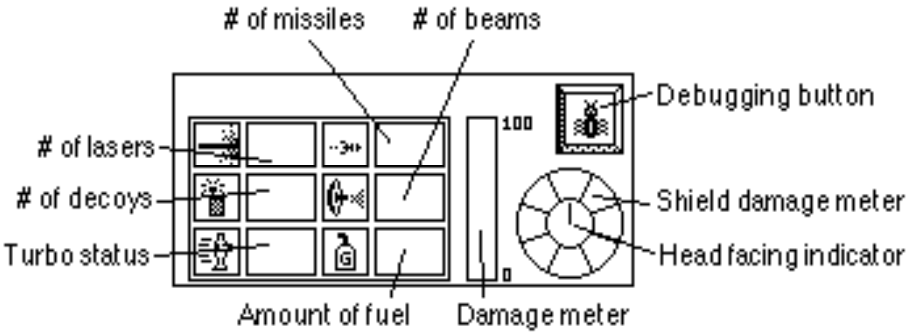


Figure 2. Vital stats box

Figures 1 and 2 show the main viewing screen for Robot Battle. In the vital stats boxes, it is possible to click on the debugging buttons to bring up the comprehensive debugging windows for each robot. These buttons act like toggles - the first click brings up the debugging window, the second click removes it. The debugging windows can also be removed by clicking on their go-away boxes.

The six information boxes in each stats box display relevant information about the robot's six main equipment types. The damage bar begins as a white bar (no damage) and will fill with black as the robot is wounded. The battle ends when one of the robots is fully damaged. The shield damage meter ring shows the status of the eight shields which surround each robot. This shield meter ring rotates with the robot. For example, if a robot's front shield is destroyed and the robot is facing up, the top shield damage meter will be greyed out. If, on the next turn, that robot turns its body to the right, the ring will rotate, and now the upper-right meter will be greyed instead. Finally, inside the shield damage meter ring is a small vector which shows the current facing of the robot's head.

Note: As a convenience, clicking on the information boxes is equivalent to selecting the "Load Robot..." menu options.

The closeup view shows a magnified portion of the battlefield. Only a small section of the entire battlefield can be viewed at once with the closeup view. The smaller radar view represents the entire battlefield. On this screen, small black and white dots represent the positions of the two robots. In addition, missile explosions and laser shots are displayed on the radar view (as well as on the closeup view).

The relatively large rectangle which appears in the radar view represents the section of the battlefield currently being displayed on the closeup view. The closeup view is moved by simply clicking in the radar view. This causes the closeup view to center on the area of the battlefield on which you clicked. Another method is to hold down the mouse button while in the radar view and move the mouse around. This has the effect of "dragging" the closeup view around the battlefield. Additionally, the keyboard arrow keys can be used to slowly move the closeup view around.

The RIPPLE Language

Robots are programmed using a language called RIPPLE, which stands for Robot Instructional Programming LanguagE. RIPPLE is very similar in style to BASIC.

Here's a very simple example of RIPPLE code:

```
10  Let A = 1
20  Let A = A + 1
30  Goto 20
```

This example simply counts upward from one. It will never end. An important feature of RIPPLE is that line numbers are simply labels. Control of the program is controlled by which

statement comes next in the file, not which line number is next. Therefore,

```
1000  Let A = 1
500   Let A = A + 1
10    Goto 500
```

is functionally identical to the first example. Indeed, lines numbers aren't necessary at all for lines that you never want to branch to. As a final example of our counting program,

```
  Let A = 1
1   Let A = A + 1
  Goto 1
```

does exactly the same thing as our other two examples. In our subsequent examples, we will use line numbers only when they are needed as labels.

Syntax rules of RIPPLE:

•The RIPPLE language is case-insensitive, so feel free to use upper or lowercase as you see fit.

• Blank lines are allowed within the text files.

•Comments can be included in a file by starting any line with the backquote character: ` Any line with ` as the first character will be ignored.

• The user has twenty-six debugging variables for their use, the letters A-Z. These special variables have their contents displayed in the comprehensive debugging windows. Other variables that users create for their own use, such as "myVar", are not displayed in the debugging windows. You can create variables and arrays with any names you desire (up to 10 chars in length). If a section of your code has a problem, use the A through Z variables and see what's going on.

In addition, several system variables exist which may be examined but not modified.

• Negative numbers cannot be entered directly into programs. Negative numbers can be used, but indirectly. For example,

Let NegNum = -10

is not be a valid statement, but

Let NegNum=0-10

would be. Therefore, if you wish to have a negative value for an argument to a command (such as SWIVEL), you must place that negative value in a variable and use that variable as the argument. Another example:

Swivel -10

is bad, but

```
Let negTen=0-10
Swivel negTen
```

is fine.

• Complex arguments are not allowed. Only constants or single variables are allowed as

arguments to commands. This rules out something like:

Swivel 0 - 10

• The compiler is reasonably good at sorting out what you mean on a line. For example, it will handle the line

500forgeorge=fredtosam

without any problems. The exceptions to this "I don't need spaces" tough-guy stance are the EQUIP and SETSHIELDS commands. The arguments to these two commands must be separated by spaces.

• FOR-NEXT caveat - you cannot use an array element as a looping variable:

for myArray(2)=1 to 10

is right out. No way. Sorry.

The RIPPLE Commands

The RIPPLE commands can be grouped into two categories: Logic and Action.

During battle, the robots work on the concept of turns. Each robot executes its program until it does something to end its turn. A turn is ended when: any Action command is performed or after 99 Logic commands have been performed. This system allows the robot to contain much complex internal logic with no speed penalty.

The commands are described in detail below, followed by a list of the system variables and their contents.

The following notation is used throughout the command descriptions:
U - user variable (A-Z, or any variables you've created, including array elements)
S - system variable
C - constant

Logic Commands:

GOTO <u,s,c> - performs an unconditional branch. Program control is passed to the line number specified by the argument. Examples:

Goto 500
Goto D

IF <u,s,c> <cond> <u,s,c> THEN <command> - performs conditional branching. If the condition is true, then the command following THEN is executed. Valid conditionals are <, >, >=, <=, =, and <>. Examples:

If A > theArray(7) then Goto 100
If B <> XCOORD then Zap

GOSUB <u,s,c> - branches to a subroutine. Like GOTO, except that when a RETURN command is encountered, control returns to the first line following the most recently

called GOSUB command. Note that GOSUBs may be nested up to 75 levels deep. Examples:

Gosub 500
Gosub D

RETURN - when encountered, RETURN passes control back to the most recently called GOSUB command. Example:

```
  Let A = 0
  Gosub 500
4 Goto 4
500 Let A = 5
  Return
```

In this example, when line 4 is reached, the value of A will be 5.

FOR <u> = <u,s,c> TO <u,s,c> - A FOR-NEXT clause performs the operations between the FOR and NEXT statements repeatedly. An example:

```
For Looper=1 to 50
  Swivel Looper
Next
```

The swivel operation will be performed 50 times. Looper is incremented by 1 each time through the loop. FOR-NEXT loops can be nested.

LET <u> = <u,s,c>

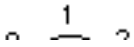
LET <u> = <u,s,c> <oper> <u,s,c> - LET assigns values to the user variables. The first version simply assigns the value of a constant or other variable to the user variable. The second version allows the user to use simple math functions: +, -, *, /. All results of math operations are expressed as integers (any fractional portion of the result is ignored). Examples:

```
Let Chuck = 4
Let SmallY = YCOORD / 6
Let Tob = D * Mary(4)
```

DIMENSION <u>(<c>) - DIMENSION creates a one-dimensional array of the specified size. An array must be dimensioned prior to being used. Examples:

Dimension ourArray(50) Creates an array for us to use
Let Chuck = ourArray(6) * 4 Accesses the 6th element

CHECKSHIELD <u,s,c> - CHECKSHIELD expects an argument in the range of 1-8. Given a valid argument, it then returns the amount of damage done to the corresponding shield into the system variable SHIELD. The shields are numbered clockwise, with shield 1 being directly in front of the robot.



Note: the shield numbering is relative to the body facing of the robot; when the robot

rotates, so do the shields, so that shield 1 is always in front of the robot.

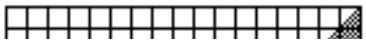
MAKERANDOM <u,s,c> - given a positive argument x, MAKERANDOM generates a random number in the range 1 through (x-1) and places the result in system variable RANDOM.

Action Commands:

HUMANSCAN <u,s,c> - this command, surprisingly enough, scans for humans. This is accomplished by looking in the direction that the robot's head is currently facing. The positive argument tells how many degrees on either side of the robot's line of sight to scan. This sounds confusing but is clarified with an example:

20 HumanScan 10

Assume the robot's headfacing = 90 (0 degrees is up, therefore headfacing = 90 means the robot's head is facing to the right). The argument, 10, tells the robot to scan in an arc ranging from 10 degrees on either side of the current headfacing. Therefore, the area scanned is the cone from 80 degrees to 100 degrees. If any humans are within this cone, the distance to the nearest spotted human is returned in system variable RANGE. If no humans are spotted, RANGE is set to zero. The figure below illustrates the range covered by Humanscan 45 when the headfacing is equal to 90.



OBJECTSCAN <u,s,c> - this command works in the same way as HUMANSCAN, with the exception that OBJECTSCAN reports the sighting of the enemy robot or decoys instead of humans. The distance to the nearest decoy or robot is returned in RANGE. If no objects are sighted, RANGE is set to zero.

EQUIP <c> <c> <c> <c> <c> <c> - EQUIP is a special command. It must be the first command of any robot program (not including blank lines and comment lines), and may be called only once for any robot program. This command allocates equipment to the robot. The six arguments tell how much of each type of equipment the robot is to hold. The robot has a total weight limit of 500 kilos. The constants represent the following items, in order:

Lasers - how many lasers the robot is to carry. A beam shot by a 3-laser robot is three times as powerful as a single laser robot's. Lasers weigh 50 kilos each.

Missiles - the number of missiles to be carried. 15 kilos each.

Sonic Beam - this is an extremely powerful weapon that can be used at very short range. Each hit with the sonic beam causes damage equal to one third of the total damage meter range. Only one sonic beam may be on-board, and the weight is 75 kilos.

Decoys - the number of decoys to be carried. Decoys are useful for confusing the other

robot's OBJECTSCANs and have a chance of causing enemy missiles to accidently detonate. Decoys weigh 4 kilos each.

Turbo - equipping robots with a turbo booster enables them to move twice normal speed when the turbo is activated. The turbo unit weighs 75 kilos. Arguments greater than one do not cause more turbo boosters to be added.

Fuel - Action commands that actually move the robot or fire the laser use fuel. Therefore the amount of fuel the robot has will dictate how long it will be able to fight (assuming it isn't killed first!). Fuel weighs 1 kilo per unit.

TURN <u,s,c> - TURN rotates the robot's body in 45° increments. If the argument to TURN is any positive number, the robot's body is rotated 45° clockwise. If the argument is negative, the body is rotated 45° counterclockwise. Turning the body has no effect on the facing of the head.

SWIVEL <u,s,c> - given an argument x, SWIVEL rotates the robot's head x degrees. The argument may be positive or negative, but must be in the range of -360 to 360. SWIVEL is a relative command; SWIVEL 10 will not place the robot's head at a 10° orientation. Rather, it will increment the robot's head facing by 10 degrees. So if HEADFACING was equal to 45° before SWIVEL 10 was called, it would be equal to 55° after the call. Turning the body has no effect on the facing of the head.

ZAP - this command fires a laser beam in the direction of the robot's head facing. The laser beam will hit the first item it comes in contact with, with one exception: a robot's laser will not destroy any decoys that the robot has placed. It can, however, destroy the other robot's decoys. Lasers automatically destroy any humans or decoys that they hit. If the laser hits the enemy robot, damage is afflicted to the robot's main damage meter. A single laser hitting an unshielded robot will damage the robot by 67 points. Each robot has 1000 damage points, so a single laser inflicts 1/15 of the total damage. Having more lasers multiplies this damage amount by the number of lasers used. If the laser must pass through an intact shield before striking the enemy robot, damage is reduced by half. Lasers do not affect shield damage in any way.

LAUNCH - launches a missile in the direction of the robot's head facing. Missiles can take several turns to reach their targets, so it is possible that the enemy robot will have moved out of the way by the time your missile arrives. Missiles travel in a straight line and have no guidance systems. They will detonate when they hit a wall or come within three units of the enemy robot. There is also a chance that they will detonate accidentally while passing over enemy decoys. When missiles detonate, they destroy all humans and decoys within their blast radius. In addition, any other missiles that are within their blast radius will also detonate. Therefore, if missiles are launched rapid-fire

one after another, a chain reaction can occur, causing all of the missiles to detonate at once. Beware. When either robot is caught within the blast radius of a missile, one of two things will happen: if there is an intact shield facing the blast, that shield will absorb the missile's damage, and no damage will be done to the robot itself. That shield, however, will most probably be destroyed. If no intact shield is facing the blast, the damage of the blast is inflicted upon the robot itself.

SHOOTBEAM - shoots the sonic beam, if one is available. The sonic beam has a very small firing range, equal to about the diameter of the robots. You must be extremely close to the enemy to do damage with this weapon. Each hit with the sonic beam is extremely lethal, however, inflicting one-third of the total maximum damage with each hit. The sonic beam is not affected by shields. Finally, make sure that you have enough fuel around; each firing of the beam uses 50 fuel units.

SETSHIELDS <u,s,c> <u,s,c> <u,s,c> <u,s,c> <u,s,c> <u,s,c> <u,s,c> <u,s,c> - This command is used to transfer energy to your individual shields. The eight arguments represent the eight shields, in order. The arguments list relative amounts of energy to be transferred. For example,

Setshields 1 0 0 0 0 0 0 0

will transfer all available shield energy to the front shield;

Setshields 1000 0 0 0 0 0 0 0

will do exactly the same thing. The values of the arguments are only relative to each other. Using bigger numbers doesn't by itself gain you anything. Another example:

Setshields 5 5 15 5 5 5 15 5

will cause the left and right shields to receive three times as much energy as the other shields.

DROPDECOY - assuming that the robot has some decoys installed, this command will drop one of the decoys behind the robot. Note that DROPDECOY always drops the decoy immediately behind the robot, so the placement of the decoy is dependent on the robot's body facing.

WAIT <u,s,c> - given a positive argument x, WAIT causes the robot to do nothing for x number of turns. This is sometimes useful when attempting to move a set distance. For example,

EngineOn
Wait 4
EngineOff

will move the robot 5 units in the direction of its body facing.

ENGINEON

ENGINEOFF - these two commands turn the robot's engine on and off. While the engine is on, the robot will move one unit in the direction of its body facing every turn until the engine is turned off again. This allows the robot to carry out other actions while moving. Note: running into the borders of the battlefield or the other robot automatically shuts off the engine. The status of the engine can be determined by reading the ENGINESTAT system variable.

TURBOON

TURBOOFF - if the robot is equipped with a turbo booster, these two commands toggle the turbo effect on and off. When the turbo is on, the robot moves twice as fast when the engine is on. Robots using turbo also hit walls twice as hard... Note that simply turning the turbo on has no effect by itself; the engine must be on also.

System variables:

The following are the robot's system variables: variables which may be examined but not changed directly. A brief explanation is provided for each:

MISSILES - the number of missiles remaining
BEAM - whether we have a beam weapon on-board or not
DECOY - the number of decoys remaining
SHIELD - the strength of a given shield is returned here by CHECKSHIELD
FUEL - the number of fuel units remaining
RANDOM - result of MAKERANDOM is returned here
RANGE - result of HUMANSCAN and OBJECTSCAN calls
DAMAGE - current amount of damage done to the robot itself
BODYFACING - current body facing of the robot (number from 1 to 8)
HEADFACING - current head facing of the robot, in degrees
XCOORD - the robot's x coordinate on the battlefield grid - range 1-75
YCOORD - the robot's y coordinate on the battlefield grid - range 1-75
DAMAGELIMIT - the maximum amount of damage the robot can sustain
SHIELDLIMIT - the maximum amount of damage any shield can sustain
ENGINESTAT - contains a 1 if the engine is on, 0 otherwise.
TURBOSTAT - contains a 1 if the turbo is on, 0 otherwise.