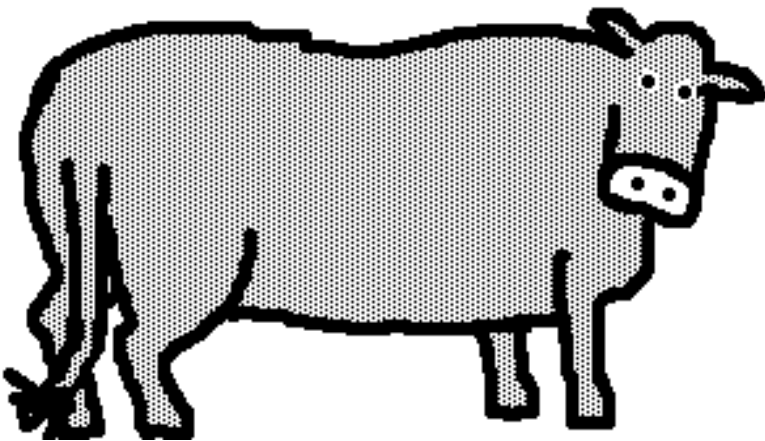


Mech Zone: Robot Battle II

Manual Version 1.0



©1991 Blue Cow Software

Chapter 1 - An Introduction	1
1.1 About Mech Zone: Robot Battle II	1
1.2 What's New in 2.0	1
1.3 Our ShareWare Policy	2
1.4 About Blue Cow Software	2
Chapter 2 - Getting Started	4

2.1 System Requirements	4	
2.1.1 Hardware	4	
2.1.2 System Software	4	
2.1.3 Using Mech Zone With MultiFinder or System 7	4	4
2.2 If Mech Zone Should Bomb (gasp!)	5	
2.3 Starting the Program	5	
2.4 Now What Do I Do?	5	
Chapter 3 - User's Guide	6	
3.1 The Menus	6	
3.1.1 The Apple Menu	6	
3.1.2 The File Menu	6	
3.1.3 The Edit Menu	7	
3.1.4 The Battle Menu	7	
3.1.5 The Misc. Menu	7	
3.2 Running a Battle	7	
3.2.1 The Display	8	
Adding Robots	8	
Removing Robots	8	
Setting Teams	9	
Setting Number of Humans	9	
Allowing POSITION Commands	9	9
3.2.2 Loading and Compiling Robots	9	
3.2.3 Watching the Battle Take Place	10	
3.2.3a The Display	10	
The Closeup View	10	
The Radar View	11	
The Status Displays	11	
Those Big Buttons	11	
3.2.3b The Debuggers	12	
Accessing the Debugging Windows	12	12
What All That Garbage Means	12	
Of Debuggers and Speed (Or Lack Thereof)	13	13
3.2.3c Stopping the Action	13	
New Game	13	
Pause	13	
Step Mode	13	
Restart Same Robots	13	
Chapter 4 - A Programming Tutorial	14	
4.1 Radar Love: a Walk-Through	14	
4.1.1 Commands and Variables	15	
4.2 Build a better robot	19	
Structured Programming	28	
Where to go from here	28	
Teams and Communications	28	
Trig Functions	28	
Chapter 5 - Programmer's Guide	29	
5.1 Introduction to This Section	29	
5.2 Rules of the Language	30	
5.3 Commands & Functions	31	
• EQUIP	32	
Lasers:	32	
Missiles:	32	
Sonic Beam:	33	
Decoys:	33	
Turbo booster:	33	
Fuel:	33	
The Equipment Calculator	33	
Choosing an Equipment Configuration	33	33
5.3.1 Logic Commands	34	

5.3.1a Variables and Assignments	34
•LET34	
The Functions	35
CHECKSHIELD	35
RANDOM	35
TRUNC	35
SQRT	36
SIN, COS, TAN	36
ASIN, ACOS, ATAN	36
HUMANSCAN, OBJECTSCAN	36
• DIMENSION	37
5.3.1b Flow Control	37
• GOTO	37
• IF...THEN...ELSE	37
• GOSUB	38
• RETURN	39
• FOR...NEXT	39
• IF...BEGIN	39
• WHILE...BEGIN	40
5.3.1c Communication commands	41
• SEND	41
• RECEIVE	41
5.3.2 Action Commands	42
• WAIT	42
5.3.2a Movement Commands	42
• TURN	42
• ENGINEON, ENGINEOFF	42
• TURBOON, TURBOOFF	43
5.3.2b The Scanning Commands	43
• HUMANSCAN	43
• OBJECTSCAN	44
• SWIVEL	44
5.3.2c The Weapon Commands	45
• ZAP	45
• LAUNCH	45
• SHOOTBEAM	45
5.3.2d Defensive commands	46
• DROPDECOY	46
• SETSHIELDS	46
5.3.2e Debugging command	46
• POSITION	46
5.4 System variables	47
MISSILES	47
BEAM	47
DECOYS	47
FUEL	47
RANGE	47
ENEMY	47
DAMAGE	47
BODYFACING	47
HEADFACING	48
BEARING	48
XCOORD	48
YCOORD	48
ENGINESTAT	48
TURBOSTAT	48
LASERS	48
DATAWAITING	48
TEAMSIZE	48

TEAMID	48
Appendix A: Changes since version 1.1	49
Interface changes	49
Start Battle Dialog Box	49
Follow robot by clicking on status area	49
New menu commands	49
Equipment calculator	49
Status display addition	49
Multi-robot battles	49
Select robot by clicking or by popup menu	49
Teams & Communications	49
Send	50
Teamsize & TeamID	50
Receive	50
Datawaiting	50
Friendly	50
Improved Graphics with animation	50
Color	51
Improved sound	51
Language improvements	51
If...then...else	51
Floating point math	51
Trig and algebraic functions	51
Trunc	51
Sqrt	51
Sin, Cos, Tan	52
Asin, Acos, Atan	52
Structured code	52
New system variable: Bearing	52
Support for teams and communications	53
Comments on same line as code	53
Optional Let	53
Makerandom and Checkshield converted to functions	53
Rules changes	53
Head turns with body	53
Sonic beam changed	54
Shields changed	54
Zap altered slightly	55
Fewer logic statements per turn	55
Debugging tools	55
Position command	55
Debugging window	55
Bug fixes	55
Appendix B: Further reading	56

Chapter 1 - An Introduction

Thank you for your interest in Mech Zone! We've been working around the clock to improve Robot Battle in response to users' requests, and we hope that you enjoy the fruits of our labors. Please take the time to read this manual: you can't program robots unless you do, and the sample robots we've included with the game won't hold your interest for long.

1.1 About Mech Zone: Robot Battle II

The object of Mech Zone is to design and program robots, then send them into the arena to duel each other. Robots can be equipped with several different kinds of weapons and other enhancements, but the most important element of robot design is the software which controls a robot's actions. Given two similarly-equipped robots, the smarter one will win most of the time. The challenge, then, is to develop the best strategy for victory and to program it into your robot.

1.2 What's New in 2.0

For those of you who have been using Robot Battle 1.1 (the last released version), here's a quick synopsis of what's new in Mech Zone: There can now be eight robots fighting at once; robots can form teams and communicate with one another; several extensions and improvements to the language have been made; the graphics have been significantly reworked, and robot movement is now animated; a huge number of bugs have been squashed; and the manual has been rewritten from scratch. A detailed list of these changes is located later in this manual, saving you the drudgery of sifting through the whole manual for the new info. There are still a few items remaining on the Robot Battle Wanna Do list, so you can look forward to yet more upgrades...

1.3 Our ShareWare Policy

The introductory version of Mech Zone that's available on the various communications services is inferior to the full version in a couple of respects. First, two of the weapons, the missiles and the sonic beam, do no damage to enemy robots. You can still use them, and you can see them firing during a battle, but they won't hurt a fly. The lasers are completely functional, though. Second, there is no robot communications mode available in the introductory version. (You can set up a battle with some robots on one team and some on another, but there's no way for the robots to communicate during a battle). Specifically, the BROADCAST and RECEIVE commands will do nothing. However, the global variable ENEMY does contain useful info, so at least when you're looking at another robot you can tell if it's on your side.

You may try out the introductory version for 30 days; after that, if you decide to keep it, we ask that you send the a shareware fee of \$10. For this, you'll have the peace of mind of knowing that you're legally using the program, you'll be supporting future Robot Battle upgrades, and you'll receive notice of all Robot Battle happenings. For \$25 (\$20 for students) you can become a fully registered user. This means that you'll receive the full version of Mech Zone along with a printed manual, and will be eligible for free upgrades as they become available. See below for our address.

1.4 About Blue Cow Software

Bluecow Software began with two erstwhile graduate students and Mac fanatics who go by the names of Charlie Moylan and Toby Smith. Now we are three: Elliot Wilen joined up to help design Mech Zone and to write the manual you're reading right now. We have two

products available for the Mac: Mech Zone and Strike Jets. Strike Jets is a tactical/strategic game of modern aerial combat. The game features detailed data on over 90 of the world's fighter and bomber aircraft, and allows you to create your own scenarios. If you've ever played board games such as Air Superiority, or enjoy the detail and strategy of Harpoon, Strike Jets is for you.

We maintain an anonymous FTP site on Internet that contains the latest versions of our programs, along with libraries of Mech Zone robots and Strike Jets scenarios. The address is zaphod.ee.pitt.edu (that's 130.49.15.93). We also post our programs to the two big Internet archives, sumex-aim.stanford.edu and mac.archive.umich.edu, as well as CompuServe and America Online.

Our electronic addresses are:

Internet: bluecow@unix.cis.pitt.edu

CompuServe: 76174,2447

America Online: TOBYS3

Or, via snail-mail (for sending postcards and Shareware fees):

Blue Cow Software

6656 Ridgeville St.

Pittsburgh, PA 15217

We're always interested in comments, bug reports and suggestions for new and better games, so please drop us a line if you have something to say (or just to say "hi" and let us know you've tried our games).

Chapter 2 - Getting Started

This chapter explains exactly what is necessary to run Mech Zone. If you're an old hand at using the Mac, you can just skim this chapter and run the program. If you experience any problems when you try to run it, turn here to make sure that Mech Zone is compatible with your system configuration.

2.1 System Requirements

This manual assumes that you're already familiar with the Macintosh operating system. If you aren't an experienced Mac user, we suggest that you refer to the manuals that came with your Macintosh computer and system software.

2.1.1 Hardware

Mech Zone will run on any Macintosh with 1 Megabyte or more of memory. The program will run in color if you're displaying 16 or more colors. If you really want a blast, hook up a pair of headphones or speakers through your Mac's sound jack.

2.1.2 System Software

Mech Zone is designed to work with version 4.1 or higher of the Macintosh system software. It has been tested extensively under System 6.0.7 and System 7. For best performance, we recommend using a RAM cache of 128K or more. (You can set the size of your RAM cache using the Control Panel desk accessory, or the Memory Control Panel under System 7.) Mech Zone is happiest of all when running under System 7 on a Mac II or better, because then it can do three-voice sound and a color battlefield, which makes it very proud (the three-voice sound is also available under System 6.0.7).

2.1.3 Using Mech Zone With MultiFinder or System 7

Mech Zone prefers a 1 megabyte partition under MultiFinder. If you don't have that much memory free, the game will run with as little as 570K, although this is not recommended as some (non-catastrophic) errors may occur. If you put Mech Zone in the background while a battle is running, the battle will continue to run in the background at a slower speed. The game sounds are automatically disabled while Mech Zone is running in the background.

2.2 If Mech Zone Should Bomb (gasp!)

If Mech Zone should bomb during execution, there's a fair chance that the program will know what caused the problem. If you are using Macsbug, you may see some sort of "Robot Battle Sez: " line in the MacsBug display. These messages can sometimes help you avoid future crashes. They normally occur when you've allocated too little memory to the program, or when the RBDData file (see below) isn't present. (If you don't know what MacsBug is, don't worry about it.)

If you run into a bomb or error, we'd appreciate it if you'd let us know, either by electronic mail or through our postal address. Please tell us the type of Macintosh and the version of the system software you're using (if you know it), and describe the error as best you can. If the error occurs repeatedly under a specific set of conditions or after you've performed a certain action, please tell us the conditions or actions which precede the error; that will help us narrow down the source of the bug.

2.3 Starting the Program

Mech Zone consists of two files: the Mech Zone program itself, and a file called RBDData. The RBDData file must be in the same directory as Mech Zone or the program will not run at all. To start the program, double-click on the Mech Zone icon. You'll soon be greeted by the Mech Zone title screen and theme song. If you ever tire of gazing at this wonder, press any key or click the mouse button, and the main Mech Zone window will appear.

2.4 Now What Do I Do?

If this is your first experience with Robot Battle, load in a couple of the sample robots (as described in the User's Guide under Running a Battle), watch them fight, then check out the source code for each of the samples. (Note that the included sample robots are intentionally pretty dumb, and none of them is a good illustration of clever robot programming.) If you know how to program, all you may need to do now is scan through the User's Guide and Programmer's Guide and start hacking. If you don't know how to program, or you have little

experience and would like a few tips, read the Programming Tutorial. The Programming Tutorial will teach you the basics of designing, coding, and debugging a robot program. In the process, it will show you how to create a robot that can beat the pants off the sample robots.

Chapter 3 - User's Guide

Mech Zone follows the Macintosh user interface guidelines fairly closely, so using it should be straightforward for anyone familiar with the Mac. However, there are several features which you'll probably miss if you don't read this section.

3.1 The Menus

The following sections describe the basic commands found in the program menus.

3.1.1 The Apple Menu

This menu contains the About Mech Zone... command, which displays a brief message about Mech Zone with credits, contact information, and a shareware notice.

3.1.2 The File Menu

This menu contains the standard commands for opening, closing, saving, and printing robot files. Mech Zone has a built-in text editor which allows you to create and edit robot programs. If you wish, you may also create and edit them using your favorite word processor or text editor. (Just make sure you save them as "Text Only".)

Note that the Mech Zone editor can only edit one file at a time. Also, you may not use the editor while a battle is in progress.

In addition to the usual file-manipulation commands, the File menu contains the Revert command, which returns the currently open file to the state it was in when last saved. The File menu also contains the Quit command, which quits Mech Zone.

3.1.3 The Edit Menu

This menu contains the standard Mac Cut, Copy, Paste, and Clear commands. They are used by the built-in text editor and by some desk accessories. (The Undo command is not currently functional, although some desk accessories use it.)

3.1.4 The Battle Menu

The commands in this Menu are used to set up and control a battle. The use of these commands is explained fully under Running a Battle, below.

3.1.5 The Misc. Menu

This menu contains four commands. The Sound command controls whether the program makes sounds while loading robots and running a battle. The second item, Ignore Position Commands, tells Mech Zone whether to allow POSITION commands during battle. (The use of POSITION commands is explained more fully under Running a Battle and in the Programmer's Guide.) The Equipment Calculator is a utility to help configure robots and is

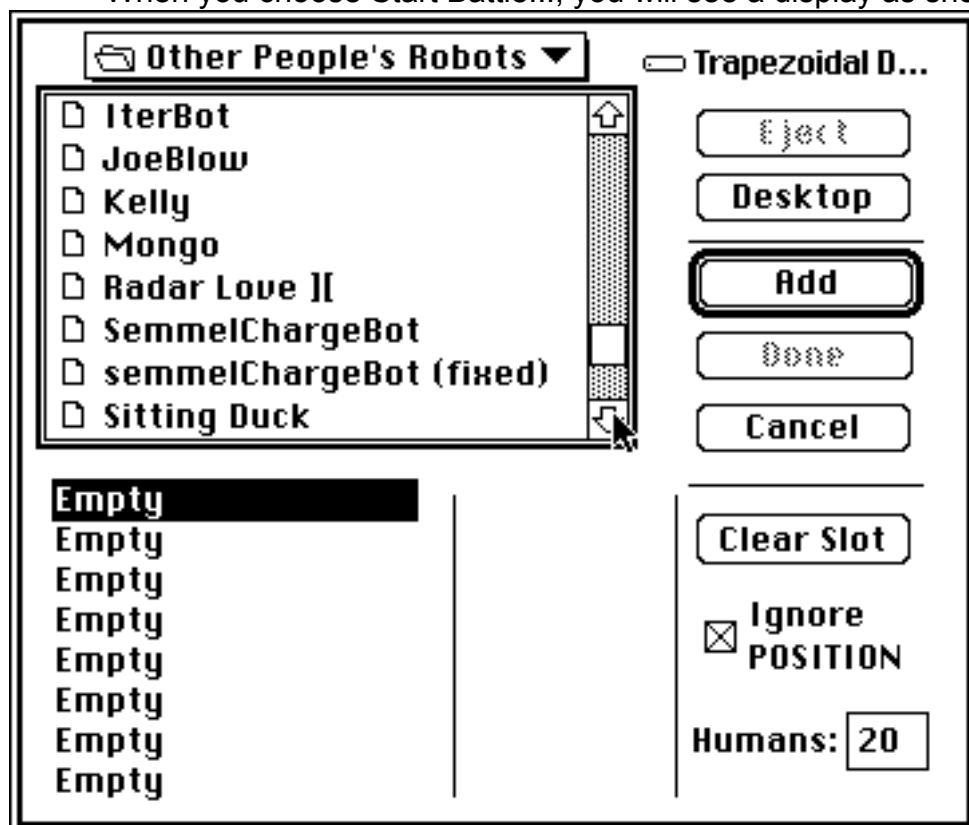
explained in the Programmer's Guide under the EQUIP command. The final item in the Misc. Menu, Set Graphics Display, controls the amount of detail which is displayed during a battle. The program ordinarily displays an animated closeup view of a portion of the battlefield. Using the Set Graphics Display command, one can turn off animation or turn off the closeup view altogether.

3.2 Running a Battle

In order to begin a battle, you must tell Mech Zone what robots you want to participate in the battle, what teams they are on, and how many humans (see below) you want on the battlefield. You do this by using the Start Battle... command located under the Battle menu.

3.2.1 The Display

When you choose Start Battle..., you will see a display as shown in the figure below.



Adding Robots

To add a robot to the battle, select one of the eight slots at the lower left, then select a robot using the normal Macintosh file selection method. When you do this, the robot's name will appear in the slot, and the next slot will automatically be highlighted. You need not use all eight slots, although you obviously cannot start a battle with less than two robots.

Removing Robots

If you wish to remove a robot from one of the slots, select the slot and press the Clear Slot button. The robot will be removed.

Setting Teams

All robots are initially considered to be independent; i.e., they regard all other robots as

enemies. This is indicated by the abbreviation Indep., which appears in the column to the right of the robot's name. To assign a robot to a team, click on the area to the right of the robot's name. A popup menu will appear allowing you to choose from five possible team assignments: Independent, or a Team from 1 through 4.

Setting Number of Humans

Humans are annoying little creatures which wander around the battlefield. They tend to congregate around robots, getting in the way of laser fire, and they have the irksome habit of throwing rocks at robots which stay in one place for too long. The number of humans on the battlefield is initially set to 20. You can change the number of humans to any value between 0 and 99 by typing in the box in the lower right corner of the Start Battle display.

Allowing POSITION Commands

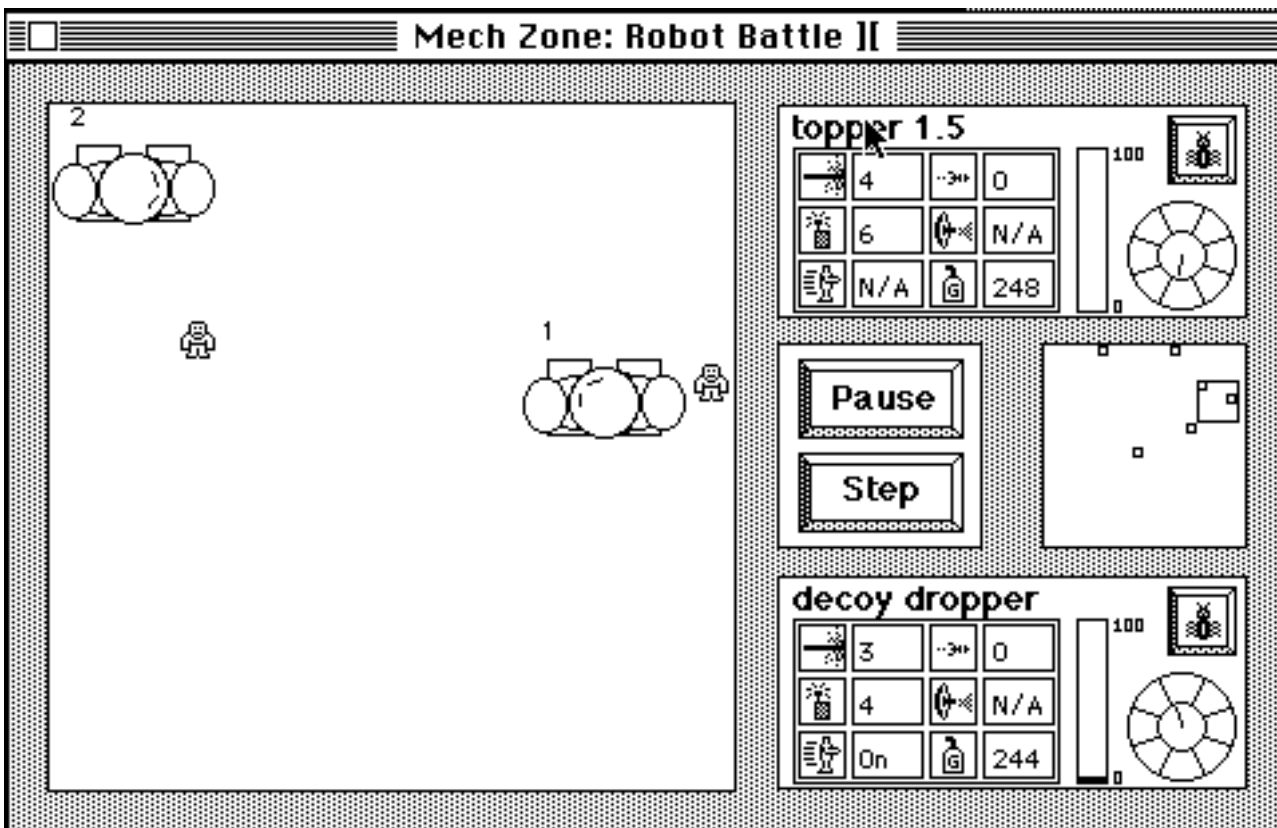
The POSITION command is a part of the RIPPLE programming language which is included for debugging purposes. It allows a robot to instantly reposition itself to any (unoccupied) point on the arena map. You can thus have teleporting robots if you so desire. But since this is cheating, there is also a way to prevent POSITION commands from being executed during battle. To do this, make sure the check box on the right side of the Start Battle display is checked on. (The box is checked on by default.) You can also disable POSITION commands using the Misc. menu, as noted above.

3.2.2 Loading and Compiling Robots

When you're satisfied with a given setup, click the Done button. (If you wish to abort the setup and do something else, click on Cancel.) At this point Mech Zone will display a status bar and begin loading the robots into memory. (This can take a while if there are a lot of robots and/or the robots are fairly complex, so be patient.) When a robot is loaded, its program is checked to make sure it contains no obvious errors, and then it is translated (*compiled*) into a form which can be understood by Mech Zone's internal processor. If there are no compile errors, the robots will load one at a time and the battle will begin. (Note that once a robot has been compiled, Mech Zone will automatically skip the compile step and load it directly. Mech Zone only re-compile robots if they have been changed since the last time they've been compiled.) If an error is found during compilation, however, a warning will sound and a message will appear describing the error. The text editor will then open with the file containing the error.

3.2.3 Watching the Battle Take Place

While a battle is going on, you have access to several displays and controls which allow you to monitor the action.



3.2.3a The Display

The main display is made up of several displays and views.

The Closeup View

On the left half of the main display is the closeup view. This view shows a detailed picture of a small portion of the battlefield. Next to each robot in this view is a number which shows the slot the robot was placed in using the Start Battle command. If there are teams, each robot also has a small symbol next to it (not shown here) indicating which team it belongs to. Within the closeup view it is possible to observe the directions that robots' heads and bodies are facing, and to see humans, decoys, and missiles.

The Radar View

At the center right is the Radar View. This view shows an overall picture of the entire battlefield. Only robots (represented by small boxes, or their team symbol, if any), missile explosions, and laser shots are displayed in the Radar View.

The relatively large rectangle which appears in the radar view represents the portion of the battlefield currently displayed in the closeup view. The closeup view is moved by simply clicking in the radar view. This causes the closeup view to center on the point in 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. You can also have the closeup view "track" a given robot. To do this, click on one of the status displays. The closeup view will then stay centered on the robot which is currently in that display.

The Status Displays

The status displays show vital information about any two robots in the battle. The six information boxes contain relevant information about the robot's six main equipment types: lasers, missiles, decoys, sonic beam, turbo booster, and fuel. The damage meter begins as a

vertical white bar (no damage) and fills with color as the robot is damaged. The shield damage meter ring shows the status of the eight shields which surround each robot. Note that the shields rotate 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. Inside the shield damage meter ring is a small vector which shows the current facing of the robot's head. There is also a small dot representing the current facing of the robot's body.

Pressing the debugging button on the status display will cause a debugging window to appear. The use of this window is discussed below. Pressing the debugging button again will cause the debugging window to disappear.

Those Big Buttons

The large buttons on the main display can be used to control the flow of the battle. Pressing a button once will cause the battle to enter Pause or Step Mode. Pressing it again will turn the mode off.

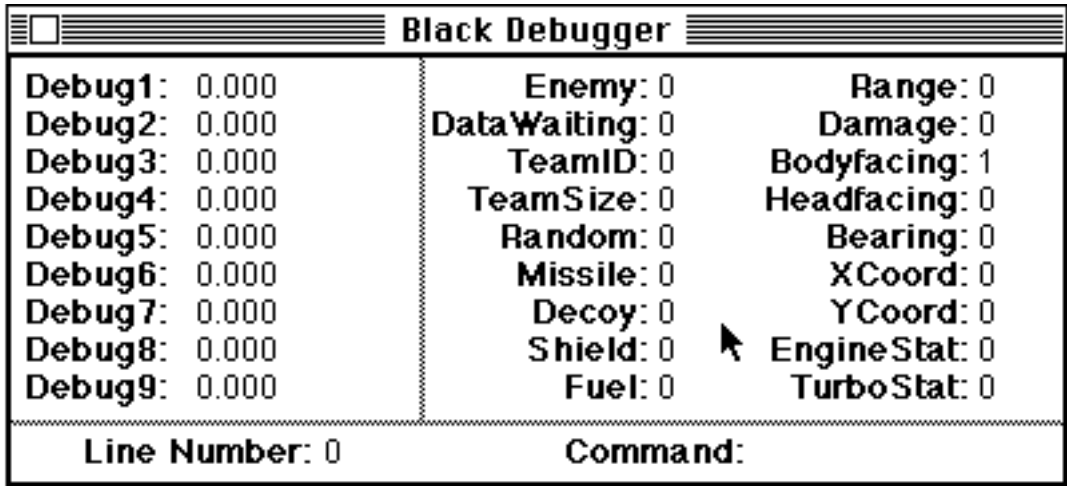
3.2.3b The Debuggers

To facilitate debugging, Mech Zone allows you to monitor the program execution of up to two robots.

Accessing the Debugging Windows

As described above, the debugging windows may be accessed by pressing the debugging buttons on the status displays. Each window is associated with the robot which is currently in its display. A window may be closed by clicking on its button or in the usual fashion by clicking in the close box.

What All That Garbage Means



The debugging window is divided into three parts. On the left are the debugging variables. These variables, called Debug1 through Debug9, can be used in a robot program as normal variables. When the robot is being monitored in the debugging window, the current value of each debugging variable is displayed and updated.

The right side of the window shows the system variables. As discussed in the Programming Guide, these variables are used to keep track of key information about the robot, such as the facing of its head, how much energy it has left, how much damage it has

sustained, and so forth.

The bottom of the window allows you to keep track of the program logic. The line number (if any) of the command currently being executed is shown on the left while the command itself is shown on the right.

Of Debuggers and Speed (Or Lack Thereof)

Note that Mech Zone slows down significantly when one or both debugging windows are open. If this isn't slow enough for you to track what's going on in your robot's program, use the Step Mode and Pause features, described in the next section.

3.2.3c Stopping the Action

New Game

Selecting this command from the Battle menu halts the battle currently in progress and clears the main display.

Pause

A battle may be paused by clicking on the Pause button on the main display, or by selecting the Pause command from the Battle menu. A battle may be unpaused either by clicking on the Pause button again or by selecting the Continue command from the Battle menu.

Step Mode

Pressing the Step button on the main display will cause the battle to enter Step Mode. When in Step Mode, the game will pause after each robot has executed a line of its program. To continue to the next line, simply click the mouse or press a key. Step Mode is particularly helpful when used in conjunction with the debugging windows.

Restart Same Robots

If this command is selected from the Battle menu, a battle which is currently in progress is interrupted and then restarted with the same robots. This is a particularly useful command when your latest masterpiece is getting trounced by an inferior piece of junk.

Chapter 4 - A Programming Tutorial

This tutorial is an introduction to programming robots using the RIPPLE language. For more information, please see the Programmer's Guide and the RIPPLE Quick Reference Guide.

4.1 Radar Love: a Walk-Through

Let's begin by looking at one of the sample robots that came with Mech Zone, Radar Love. Here is Radar Love's program. Scan over it, but don't worry about understanding the code just yet.

```
`  RADAR LOVE. Simply locks on quickly and fires.
```

```

`   TES   3-21-89
`   (modified by EMW 8-30-91)
   equip 4 0 0 0 0 300

`   Find if enemy is above or below...
10  objectscan 90
    if range > 0 then goto 20
    swivel 180
    goto 10

`   Find the 10 degree arc that the enemy is within...
20  let swivelamt = 0-85
    swivel swivelamt
30  objectscan 5
    if range > 0 then goto 40
    swivel 10
    goto 30

`   Narrow it down to the exact angle...
40  let swivelamt = 0 - 5
    swivel swivelamt
50  objectscan 0
    if range>0 then goto 60
    swivel 1
    goto 50

`   And nail him with the hard stuff.
60  zap
    objectscan 0
    if range = 0 then goto 10
    goto 60

```

4.1.1 Commands and Variables

Okay, now what does all that mean? We'll take it part by part. First of all, you've probably noticed that several of the lines in the program are written in plain English and are preceded with a backquote (`). These lines are called *comments*. They serve the purpose of allowing the programmer to explain how his program works. When writing a program, it isn't necessary to include comments, but you'll find that programs are much easier to read when you use them.

You've also probably noted that some of the non-comment lines are all preceded by numbers. These line numbers serve to identify the lines which follow them. This is important, because many robot programs need to skip forward or backward to a specific line, and there must be some way of specifying that line. There's nothing wrong with giving every non-comment line a number, but you only *have* to number the lines that the program will need to skip to.

Each non-comment line of a program is called a *command*. When a robot program is executed, each command line (or *statement*) is normally acted upon in sequence, from top to bottom. (Certain commands will modify this sequence.)

The first command line in a robot program is special:

```
equip 4 0 0 0 0 300
```

Every robot program must begin with the Equip command. The numbers after the word “equip” are called the *arguments* to the command. Not all commands have arguments; the Equip command has six of them. The six arguments to the Equip command tell the robot exactly what kind of equipment it is to carry. The arguments represent, in order, lasers, missiles, sonic beam, decoys, turbo booster, and fuel. Thus Radar Love’s Equip command tells the robot that it is to be equipped with 4 lasers, 300 units of fuel, and nothing else.

Let’s take a look at another command:

```
20 let swivelamt = 0-85
```

Line 20 contains the Let command. The Let command is used to assign a value to a *variable* called “swivelamt”. Variables provide a way of storing and referring to numbers. In effect, you can store a number away for future use by assigning it to a variable. You can then use that number in later commands by referring to the variable which represents it. Radar Love does just that in the next line:

```
swivel swivelamt
```

The Swivel command, which tells a robot to turn its head, takes one argument. In this case, the argument is a variable, swivelamt. Since the Let command has just been used to assign a value of 0-85, or -85, to swivelamt, this line has the same effect as:

```
swivel -85
```

The reason we used a variable, instead of entering the number directly, is that RIPPLE does not allow negative numbers to be written directly. (This is also why we wrote `let swivelamt = 0-85` instead of `let swivelamt = -85`.) There are many other reasons for using variables, but this example gives you the basic idea of how they work.

Note that you can use almost any name for a variable (as long as it contains no more than 40 characters). We could just as easily have written

```
20 let fred = 0-85
```

```
swivel fred
```

and it would have meant the same thing. But it’s generally a good idea to give variables meaningful names, names that are related to their function. As with comments, this practice helps make programs easy to follow and to comprehend. It also helps the programmer enormously if important variables have names that are easily remembered and related to their use.

There is one general exception to the rule that you can name variables as you please: you may not give a variable a name that is already used as part of the RIPPLE language. Such words are known as *keywords*. The most common keywords are *commands*, the statements which make robots function. Just about everything other than a variable is a command, for example, IF, SWIVEL, ZAP, and LET are all commands. You can get in trouble with the Mech Zone compiler if the beginning of a variable coincides with the name of a command. Creating a variable called Letterman would be a bad idea, since LET is a command. If you get some odd compile-time errors, check to make sure that your variables don’t suffer from this problem.

In addition to commands (and functions, which we’ll discuss later), RIPPLE includes a number of keywords which are *system variables*. System variables are a special sort of variable. Like other variables, they represent numerical values. Unlike other variables, they may not have values assigned to them directly by a Let command. Instead, system variables contain information about the robot’s status and the results of certain commands. Here’s an

example of a system variable:

```
10  objectscan 90
    if range > 0 then goto 20
```

The line following line 10 contains the system variable Range. Range always contains the result of the last use of the robot's scanner. When the scanner is activated via the Objectscan command, as in line 10, Range takes a value equal to the distance from the robot to the nearest object within the area scanned. If no object was found, then Range takes a value of 0. (There are a few simplifications in this description of Objectscan, but they need not concern us for the moment.) By referring to Range in the next line, the robot is able to make a decision: if Range is greater than 0, meaning that an object was found by the scanner, the robot will skip to line 20 of its program.

We now have most of the basic concepts necessary to understand Radar Love's program. All we need to do now is to see exactly how the commands work. We've already discussed the Equip command, so let's go on to the next portion of code, much of which we've seen already.

As we just mentioned, the Objectscan command found in line 10 tells the robot to activate its scanner. When it does this, it scans an area centered on the direction the robot's head is facing. The width of the area is controlled by Objectscan's argument, which is the number of degrees to either side of the head that should be scanned. Thus line 10 tells the robot to scan 180° of arc.

The next line is an example of the If command. This command evaluates a *condition*, which is found between the keywords "if" and "then". A condition is a simple comparison of two values; in this case, the comparison is ">", or "greater than". If the condition is true (e.g., if range is greater than 0), then the command found after the word "then" is carried out. Otherwise the program will continue straight on to the next line. In this case, if the condition is true, the program will execute the Goto command.

The Goto command takes a single argument, which must be a line number in the program. When a Goto is executed, the program skips to that line number and takes up execution of commands from that point. In our example this means that if the range produced by the scanner is greater than 0, the program will skip to line 20. For the time being, let's assume that the Objectscan returned (yielded) a range of 0, meaning that nothing was found. In that case, the condition in the following line would be false (0 is not greater than 0), and instead of executing the command after the "then", the program would simply continue on to the next line:

```
swivel 180
```

The Swivel command tells the robot to turn its head. The argument tells the robot exactly how far to swivel: a positive number means a number of degrees in a clockwise direction, while a negative number means a number of degrees in a counterclockwise direction.

After the line containing the Swivel comes a line with another Goto, only this time it isn't part of a condition.

```
goto 10
```

When the robot reaches this line, it **must** skip back to line 10. Thus we see that if the robot doesn't find a target in the first 180° of arc it scans, it will swivel its head to face in the opposite direction and start all over by scanning the other 180° of arc. The four lines we've just discussed:

```
10  objectscan 90
    if range > 0 then goto 20
```



```
swivel 180
```

```
goto 10
```

are what is known as a *loop*, because the program is designed to repeatedly loop back and execute them until it finds a target to home in on. (It should find one pretty soon!)

Now we can see what happens when the robot exits the loop: it goes into another loop.

```
20 let swivelamt = 0-85
```

```
swivel swivelamt
```

```
30 objectscan 5
```

```
if range > 0 then goto 40
```

```
swivel 10
```

```
goto 30
```

Since the robot knows that its target is within a 180° arc centered on its head facing, the robot narrows in on the target by breaking the arc into 10° segments. First (lines 20 and the one following) it swivels its head to the far left of the arc. Then it scans the first 10° segment (line 30). If it finds the target, it exits the current loop and skips to line 40. Otherwise it swivels 10° to the right so it will be ready for the next scan and loops back to line 30.

The next group of lines accomplish the same thing, only at an even smaller scale. Now we are scanning individual angles (`objectscan 0`) rather than arcs:

```
40 let swivelamt = 0 - 5
```

```
swivel swivelamt
```

```
50 objectscan 0
```

```
if range>0 then goto 60
```

```
swivel 1
```

```
goto 50
```

When we exit this final scanning loop, we know that the robot's head is pointing directly toward the target. Now it's time to start firing:

```
60 zap
```

```
objectscan 0
```

```
if range = 0 then goto 10
```

```
goto 60
```

Line 60 contains a new command: Zap. Its meaning is simple: fire the robot's lasers in the direction the head is facing. After each zap, the robot checks to make sure the target is still there. If the target isn't there (it may have moved or been destroyed), Radar Love will skip back to line 10 and start scanning from the beginning. Otherwise, the robot will loop back to line 60 and take another shot.

Now that you know how Radar Love works, set up a battle with one Radar Love and seven Sitting Ducks. Bring Radar Love into the status display, turn on the debugger, and watch as the commands are executed. If you wish, slow down the action further by activating the Step button. Note how you can monitor the value of Range in the debugger, and see how the result of an Objectscan affects the robot's subsequent actions.

4.2 Build a better robot

Watch how long it takes Radar Love to destroy seven enemies who don't even dodge or shoot back. Let's build something better.

The problem with Radar Love is that he has a highly *inefficient scanning algorithm*. An algorithm is a procedure designed to find a solution to a problem. One of the signs of good programming is efficiency, that is, designing algorithms which solve problems in as few steps as possible. Because robots compete directly against one another, efficiency is one of the keys

to victory in the Mech Zone arena.

Radar Love starts with a good idea for an algorithm, namely, breaking a large problem (“where is the target?”) into smaller and smaller chunks (“which arc is the target in?”). But the chunks rapidly become too small: the algorithm loses the forest for the trees. Let’s compare the problem of scanning to another problem to see if we can gain some insight.

Suppose I ask you to guess a number from 1 to 100. After each guess, I’ll tell you whether you should guess higher or lower, until you get the number. How would you go about guessing? Would you guess each number in sequence (1, 2, 3, 4,...)? If we assume that I’ve chosen my number randomly, you could expect to have to guess an average of 50 times before I stopped saying “higher” and told you you’d hit my number. What if, instead, you guessed multiples of 10 (10, 20, 30, 40,...)? Then, even if you didn’t hit on the number outright, as soon as I stopped saying “higher” you’d have narrowed the possibilities down to the numbers between your last two guesses. (E.g., if I said “higher” to 30 and “lower” to 40, you’d know that the number was between 30 and 40.) On average, you could expect to make about five guesses before narrowing the possibilities down. And then you’d only have to make at most nine more guesses, and probably less. Can you do better? Of course! What if, on your first try, you guess 50? Then, no matter what I say, you’ll have eliminated half the remaining possibilities. Split the difference by guessing 25 or 75 (depending on whether I said “lower” or “higher” to 50), and you’ll eliminate half again. If you continue in this manner, then even if you don’t hit my number by a stroke of luck, you are guaranteed to get it by your seventh try.

The method of guessing a number in a known range by always splitting the difference is known as the *binary algorithm*. Since finding the angle to a target is essentially the same as guessing a number between 1 and 360, the binary algorithm can be applied just as well to scanning as to guessing numbers. The essential difference is that because of the way Objectscan works, we cannot guess a number and be told whether to try higher or lower. Instead we must check either the right half or the the left half of any given arc. Another difference is that we won’t depend on luck to hit on the right angle: we don’t know whether we’ve found our target until we see it with an Objectscan 0. This means that we will always take about the same number of scans (approximately $\log_2 360$, or about 9) to get a lock on the target. Radar Love *could* find a target in fewer scans (as few as three), but it will usually require many more than our new robot, which we will name Binary.

The essence of Binary’s scanning algorithm is this: given an arc that we know contains an enemy robot, let us always scan one half of that arc. If the enemy robot is in the half–arc that we just scanned, we can treat that half–arc as the new arc and start over. If the enemy robot *isn’t* in the half–arc we just scanned, we’ll treat the *other* half–arc as the new arc. For simplicity’s sake, we’ll always scan the *left* half–arc of any given arc.[Begin Footnote] ---Notice how Binary’s algorithm loops back on itself repeatedly until it finds a target. No matter what happens, Binary is always asking itself “Which half–arc is the target in?”. It doesn’t matter whether the current arc is large or small; Binary will just look at the left half and check for a target. An algorithm which solves a problem by repeating the same process again and again, each time reducing the problem to a smaller version of the same problem, is called a *recursive algorithm*.--- [End Footnote]

To begin with, we need an Equip line. For the sake of comparing Binary to Radar Love, let’s give the robot the same configuration. Thus we have

- ` **Binary**
- ` **Uses binary search algorithm when scanning.**
- ` **EMW 8-28-91**
- ` **Equip with 4 lasers and 300 units of fuel.**
equip 4 0 0 0 0 300

Now we can begin searching for other robots. Since the area to be searched is 360° wide, we should never scan more than half of that. But instead of scanning 180° arcs, let's scan 128°. Here's why: as we scan smaller and smaller arcs, we'll be dividing the width of the area scanned in half. If we scan 180°, we'll eventually run into a problem when we have to divide 45 by two. So let's make life easy and choose a value which is easy to cut in half. 128 is the largest power of two that's less than 180, so it should be ideal. Even if it takes three 128° scans to get an initial lock, that's only one more than it could take with 180° scans, which is more than worth the trouble it will save us later. Now we can know what our first scanning loop should look like:

```
`  Scan 128 degree arcs until we find a target.
10  let scanwidth = 64
20  objectscan scanwidth
    if range > 0 then goto 30
    swivel 128
    goto 20
```

When Binary finds a target, it will exit the loop and go to line 30. At this point, we know that the target is within an arc 128° wide, centered on Binary's head facing. Now we want to scan the left side of the arc. To do this, we'll have to swivel the robot's head left 32° and then do an Objectscan 32. This will scan an area 64° wide. We could input the numbers literally:

```
30  let swivelamt = 0 - 32
    swivel swivelamt
    objectscan 32
```

However, we're trying to come up with a general solution to the problem, "Which half-arc is the target in?" Given an arc, it's fairly easy to calculate where the left side of that arc is. The center of the left half-arc is exactly halfway between the center and the left edge of the current arc. This is exactly half of the current scanwidth, so we'll have to swivel counterclockwise by half the current scanwidth. (Remember that the variable scanwidth, which is being used as the argument to the Objectscan command, represents a number of degrees *to either side of the robot's head*. The absolute width of the arc scanned is thus twice the current scanwidth.) The scanwidth of the left half-arc will also be half of the current scanwidth. Thus we have

```
`  Target found. Scan left side of the current arc.
30  let scanwidth = scanwidth/2
    let swivelamt = 0 - scanwidth
    swivel swivelamt
    objectscan scanwidth
```

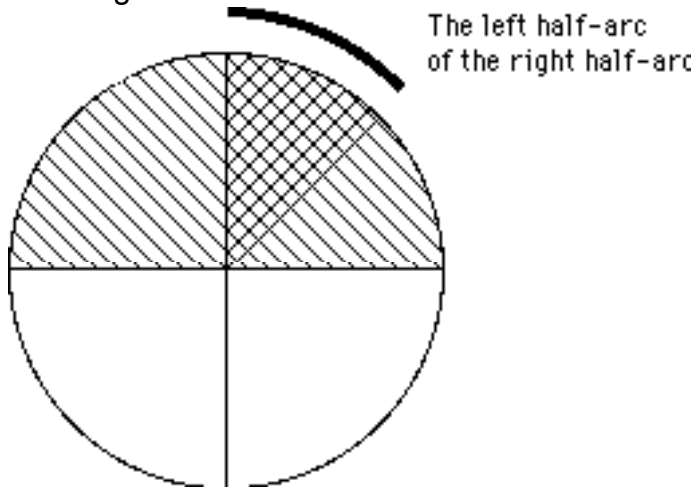
Notice how we've used variables in such a way as to avoid doing the same calculation twice. Since we're swiveling counterclockwise, we want swivelamt to be a negative number. But in absolute terms the distance we're swivelling is the same as the new scanwidth. Therefore, we first calculate the new scanwidth (`let scanwidth = scanwidth/2`), then we say that swivelamt is the negative of that value (`let swivelamt = 0 - scanwidth`).

The last line of code so far is an Objectscan. Depending on the result of the Objectscan (which will be found in the variable Range), we'll know whether or not the target is in the left half-arc. If it is, then our task is simple: loop back and treat the left half-arc as the new arc. In RIPPLE, this is written as

```
    if range > 0 then goto 30
```

If the target isn't in the left half-arc, then it must be in the right half-arc. Since we know this, there's no need to scan the entire right half-arc. Instead, we can treat it as the new arc,

meaning that our next scan should look at the *left half-arc of the right half-arc*. (See diagram.)



We could accomplish this by swiveling to the center of the right half-arc and then treating that arc as the new arc:

```
let swivelamt = 2 * scanwidth
swivel swivelamt
goto 30
```

But see what happens: after we swivel and loop back to line 30, what do we do? We swivel again! Here's what the commands would look like in the sequence they're executed:

```
let swivelamt = 2 * scanwidth
swivel swivelamt
30 let scanwidth = scanwidth/2
let swivelamt = 0 - scanwidth
swivel swivelamt
```

Swivelling is a relatively time-consuming activity: it takes an entire turn to move the robot's head from one facing to another. Wouldn't it be more efficient to figure out where the robot's head should end up, and then swivel to that point? That way we could do one swivel instead of two. If we examine the above piece of code, we see that the total distance the we should swivel is equal to $(2 * \text{scanwidth}) - (\text{scanwidth}/2)$, or $3 * (\text{scanwidth}/2)$. And the new scanwidth is, of course, $\text{scanwidth}/2$. So instead of the previous three-line piece of code, let's write:

```
let scanwidth = scanwidth/2
let swivelamt = 3 * scanwidth
goto 40
```

Before we go any further, let's take a look at the code so far:

```
` Binary
` Uses binary search algorithm when scanning.
` EMW 8-28-91
` Equip with 4 lasers and 300 units of fuel.

equip 4 0 0 0 0 300

` Scan 128 degree arcs until we find a target.

10 let scanwidth = 64
20 objectscan scanwidth
   if range > 0 then goto 30
   swivel 128
```

```

goto 20

` Target found. Scan left half-arc of the current arc.

30 let scanwidth = scanwidth/2
   let swivelamt = 0 - scanwidth

40 swivel swivelamt
   objectscan scanwidth

` If target is found, loop back to 30 and treat the current
` half-arc as the new current arc.

   if range > 0 then goto 30

` Otherwise, let's scan the left half-arc of the right
` half-arc.

   let scanwidth = scanwidth/2
   let swivelamt = 3 * scanwidth
   goto 40

```

Believe it or not, we're almost done. There's just one problem to be solved before we add a firing loop. We need to consider what happens when we do an Objectscan 1. Since each scan looks at the left half of a given arc, the result of the Objectscan 1 will tell us whether the target is in the leftmost 3° or the rightmost 2° of a 5° arc. At this point we'll have to scan each angle individually—if we just continue the loop, then the scanwidth will be 1/2, and the swivelamt will be either 1/2 or -1/2. Since fractional arguments to Objectscan and Swivel are rounded down, the robot would end up not swivelling its head at all. This means that the robot would just do an Objectscan 0 at its current headfacing, and then keep doing Objectscan 0's there forever (unless the target happens to be at that exact angle). So before we act on the results of an objectscan, we'll have to check to see whether the current scanwidth is 1. If it is, then we'll use a Goto to take us to a special final scan sequence.

As with any scan, there are two possibilities after we've done an Objectscan 1. Either we've seen the target, in which case the program will loop back to line 30, or we haven't seen the target, in which case the program will simply continue to the next line. So we should have two checks: one at line 30, and one right after the If statement. At line 30 we'll have

```

30 if scanwidth = 1 then goto 50

```

At line 50, we'll need a procedure to scan the last 3°, one at a time. First we check the left degree, then we swivel and scan twice more. After each scan, we'll check to see if we found anything, and if we have, we'll skip to our firing loop (we'll put the firing loop at line 60):

```

50 let swivelamt = 0 - 1
   swivel swivelamt
   objectscan 0
   if range > 0 then goto 60
   swivel 1
   objectscan 0
   if range > 0 then goto 60

```

```

swivel 1
objectscan 0
if range > 0 then goto 60
goto 10

```

If we haven't found anything after the third scan, that must mean that the target we've been tracking has moved. Instead of continuing to scan one degree at a time, we use a Goto to jump back to the beginning and start scanning all over again:

Now if we didn't find the target with our Objectscan 1, we'll want to scan the rightmost two degrees of the previous 5° arc. So after the check for range, we have:

```

if scanwidth = 1 then goto 55

```

At line 55, we'll want to swivel 2° to the right, then do two Objectscan 0's before giving up. Let's economize on program length by making use of part of the procedure which follows line 50. We'll give the second Objectscan of that procedure the line number 52, and write:

```

55 swivel 2
    goto 52

```

For the firing loop at line 60, we can lift the code straight out of Radar Love. Here's what our final program looks like:

```

` Binary
` Uses binary search algorithm when scanning.
` EMW 8-28-91
` Equip with 4 lasers and 300 units of fuel.

equip 4 0 0 0 0 300

` Scan 128 degree arcs until we find a target.

10 let scanwidth = 64
20 objectscan scanwidth
    if range > 0 then goto 30
    swivel 128
    goto 20

` Target found. Scan left half-arc of the current arc, unless
  we've just done an objectscan 1.

30 if scanwidth = 1 then goto 50
    let scanwidth = scanwidth/2
    let swivelamt = 0 - scanwidth

40 swivel swivelamt
    objectscan scanwidth

` If target is found, loop back to 30 and treat the current
` half-arc as the new current arc.

if range > 0 then goto 30

```

```

\ Otherwise, let's scan the left half-arc of the right
\ half-arc, unless we've just done an objectscan 1.

if scanwidth = 1 then goto 55
let scanwidth = scanwidth/2
let swivelamt = 3 * scanwidth
goto 40

\ Last swivel/scans before giving up and starting over. We
\ start at line 50 if the previous objectscan 1 was successful;
\ otherwise we start at line 55.

50 let swivelamt = 0 - 1
   swivel swivelamt
   objectscan 0
   if range > 0 then goto 60
   swivel 1
52 objectscan 0
   if range > 0 then goto 60
   swivel 1
   objectscan 0
   if range > 0 then goto 60
   goto 10

55 swivel 2
   goto 52

\ Locked on. Fire at target until it isn't there anymore.

60 zap
   objectscan 0
   if range = 0 then goto 10
   goto 60

```

Try typing Binary into the Mech Zone editor and putting it through a few rounds against Radar Love. (If you're reading an electronic version of this manual, you can just copy the code and paste it into the text editor.) You can also try setting up a battle with Binary and several Sitting Ducks, so that you can closely examine the binary scanning algorithm in action.

You'll probably agree that Binary is a major improvement in terms of scanning efficiency. Binary is still essentially a prototype, though. For one thing, he tends not to fire very fast once he's locked onto a target. For another, he never tries to dodge when he takes damage—instead he just sits there and takes it. The first problem is easily remedied: we'll just increase the number of Zaps between Objectscans. Even if a robot moves after being hit, it usually won't be able to get out of the way immediately. We don't want to have too many Zaps between Objectscans, though, or we may end up wasting time and fuel shooting at empty air. Two or three Zaps seems to be optimal. We'll err on the side of caution and go with two. So the firing loop shall be rewritten as

```

60 zap

```

```

zap
objectscan 0
if range = 0 then goto 10
goto 60

```

That was easy enough. Now for damage detection and evasive maneuvers. For this we need to introduce a few new concepts and commands. The first new concept is *functions*. A function is something that takes one or more values, or arguments, and yields a value in return. In RIPPLE, functions are always used with the Let command. For example,

```
let x = trunc 3.2
```

Trunc is a function. The Trunc function always returns the integer portion of its argument. Thus, the effect of the above example is to assign a value of 3 to the variable x. Another function is called Random. Random takes a single positive integer argument and returns a random value between one and its argument. For example,

```
let t = random 4
```

will cause the variable t to assume an integer value between 1 and 4. If you were to execute this line several times, t could have a different value each time. There are many other functions, but for the time being we're mainly interested in Random.

Another new concept we want to discuss has to do with If statements. There are some occasions when we want to do one action *or* another action, but not both. For example, suppose we have a variable, x, and we want to assign the absolute value of x to another variable, y. We might write

```

if x >= 0 then let y = x
if x < 0 then let y = 0 - x

```

(The ">=" in the condition is read "is greater than or equal to".) This works, but it's a bit inelegant: if we already know whether x is greater than or equal to 0, then surely we needn't check a second time to see if x is less than 0. The easiest way to handle alternative actions of this sort is the Else statement. An Else statement must immediately follow an If statement, and it specifies a command which is to be executed only if the condition in the preceding If statement is false. Here's how we would rewrite the last bit of code using Else:

```

if x >= 0 then let y = x
else let y = 0 - x

```

Although Else may seem like a fairly small improvement, it's really quite handy, especially when used in conjunction with *subroutines*—our next concept. A subroutine is a kind of program-within-a-program. It's a series of commands which the program is able to run at any point, and then return to where it left off. Running a subroutine involves using the GOSUB command. The GOSUB command is similar to GOTO: it causes the program to jump to a specific line number. The difference is that the GOSUB command keeps track of where it came from, so that when the subroutine is finished the program can resume execution at the line right after the GOSUB command. Here's an example of a subroutine...

NOTE! At this point our manual-writer, Elliot, rose from his trusty Mac, stared at the screen and said "Methinks I've written enough." With that, he hastily packed his bags and hopped a redeye flight to the opposite end of the country. He's now carousing with heathens and soliciting the services of shady characters with names like Bruno. Hopefully he'll return to our happy Blue Cow stable someday soon and finish this epic tutorial, since none of us have any idea where he was going with all of this. We've left his section headings intact so you can get an idea of what he might have had in mind...

Structured Programming

Where to go from here

Teams and Communications

Trig Functions

Chapter 5 - Programmer's Guide

This chapter describes in detail the commands of the RIPPLE language. This chapter is essential reading for the budding Mech Zone programmer.

5.1 Introduction to This Section

Robots are programmed using a language called RIPPLE (Robot Instructional Programming Language). RIPPLE is similar in style to BASIC, with a few modular elements that will be familiar to users of Pascal and Hypertalk.

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.

5.2 Rules of the Language

Syntax rules of RIPPLE:

- The RIPPLE language is case-insensitive; you may use upper or lowercase as you see fit.
- Blank lines are allowed within the text files.

- Comments can be included in a file by using the backquote character: ```. Any text which appears after the backquote will be ignored.

- There are nine debugging variables available: Debug1 through Debug9. These special variables have their values 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 40 chars in length). If a section of your code has a problem, use the debugging 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.

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.

- All variables contain floating point values. Floating point values may be entered directly, as:

```
Let Pi = 3.1416
```

5.3 Commands & Functions

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

During battle, 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 20 Logic commands have been performed. This system allows the robot to contain a fair amount of complex internal logic without degrading speed.

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 (including array elements except where noted)

<s> - system variable

<c> - constant (any nonnegative floating point value typed literally)

<...> - any series of command lines

We first describe a special command, Equip, then discuss the Logic commands, and lastly the Action commands. Command names are preceded by a bullet (•) for easy reference.

• EQUIP

Syntax: Equip <c> <c> <c> <c> <c> <c> <c> <c>

The Equip command must be the first command of any robot program. (It may be preceded by blank lines and comment lines, however.) It may only be executed once in the program: robots may not re-equip. The command allocates equipment to the robot. The six arguments tell how much of each type of equipment is installed on the robot. The robot has a total weight limit of 500 kg. Available types of equipment are listed below in the order that they must appear in the Equip statement:

Example:

Equip 3 10 0 0 0 200

The above line indicates that the robot is equipped with three lasers, ten missiles, and 200 fuel units.

Lasers: 50 kg

Lasers are beam weapons which are effective at any range. They fire in a straight line based on the current facing of the robot's head. Each laser does 67 points of damage to an unshielded robot, or 33 points through a shielded aspect.

Missiles: 15 kg

Missiles are fired in the same direction as the robot's head. Since a missile takes time to travel the distance to its target, it is possible that an enemy robot will have moved out of the way by the time the missile arrives. Missiles travel in a straight line and have no guidance systems. They are proximity-fused, however, and will detonate when they hit a wall or come within three units of an enemy robot. There is also a 50% chance that they will detonate while passing over enemy decoys. When a missile detonates, it destroys all humans and decoys within its blast radius. In addition, any missiles that are within its 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. Similarly, if a robot launches a missile just before being hit by one, it may find itself caught by a double blast. Beware. When a robot is caught within the blast radius of a missile, the missile will inflict 250 points of damage on the facing shield. If the shield has less than 250 points of energy, any excess damage is inflicted on the robot itself.

Sonic Beam: 75 kg (A robot may carry at most one sonic beam.)

The sonic beam has a very limited firing range, equal to about the diameter of a robot. Within this range, it affects roughly a 45 degree cone centered on the firing robot's body facing. The sonic beam is extremely lethal, inflicting one-half of the total maximum damage (i.e., 500 points) with each hit. It is not affected by shields, but it must be charged with a substantial amount of energy (costing 50 fuel units) before firing.

Decoys: 4 kg

Decoys are small transmitters used to fool enemy radars. Decoys also have a chance of causing enemy missiles to accidentally detonate.

Turbo booster: 75 kg (A robot may carry at most one turbo booster.)

Equipping a robot with a turbo booster enables it to move at twice normal speed when the turbo is activated.

Fuel: 1 kg

Robots require fuel to move, turn, fire lasers, recharge shields and charge the sonic beam. In addition, if a robot is reduced to zero fuel units it will cease to operate altogether. Therefore, the amount of fuel the robot has will affect how long it will be able to fight (assuming it isn't killed first).

The Equipment Calculator

Since figuring out exactly how much equipment a robot may carry is often time-consuming, Mech Zone provides a simple utility to do this for you. The Equipment Calculator may be accessed through the Misc. menu at any time, although you are only likely to want to use it while in the text editor.

Choosing an Equipment Configuration

To use the Equipment Calculator, simply type in the quantity of each piece of equipment and the amount of fuel you want. The amount of weight still available is automatically calculated on the fly. Clicking on the Copy to Clipboard button will generate an Equip line that corresponds to the current values in the calculator and store it in the clipboard so that it may be pasted into a robot program.

5.3.1 Logic Commands

Except where otherwise noted, all commands contained in this section are logic commands.

5.3.1a Variables and Assignments

•LET

Syntax: Let <u> = <u,s,c>

Let <u> = <u,s,c> <operator> <u,s,c>

Let <u> = <function> <u,s,c>

Assigns value on the right side of the equals sign to the variable on the left side of the equals sign. The first form simply assigns the value of a constant or other variable to the user variable. The second version allows the user to employ simple mathematical operations to

calculate a new value and assign it to the variable. Allowable operators are +, -, * (multiplication), and / (division). The third form allows the user to calculate the value of a function and assign it to the variable. The available functions are described in the next section.

Note that the word “Let” is strictly optional. For example, if you wish, you may type

A = 3

which means the the same thing as

Let A = 3

Technical note: If you use a variable in the right hand side of an assignment without first initializing it, it will automatically be initialized with a value of 0. However, you will get a runtime error if you use an uninitialized variable in any other kind of statement.

The Functions

A syntax note: Unlike most languages, RIPPLE functions do not take parentheses around their arguments. E.g.

Let A = SIN B

is correct, while

Let A = SIN(B)

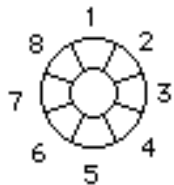
is not.

Another syntax note: the syntax of the functions below is referring to what comes on the right-hand side of the equals sign. You have to assign the function's value to a variable, of course (as in A = SIN B, above).

CHECKSHIELD

Syntax: Checkshield <u,s,c>

Checkshield expects its argument to be an integer in the range 1 - 8. [What if it isn't?] It returns the amount of energy in the corresponding shield. The shields are numbered clockwise, with shield 0 being directly in front of the robot.



Note that the shield numbering is relative to the body facing of the robot; when the robot turns, so do the shields, so that shield 1 is always in front of the robot. Shields are discussed further below, in the sections on Setshields and Powershields.

RANDOM

Syntax: Random <u,s,c>

Random expects its argument to be a positive integer. Given an argument N, it returns a random number in the range 1 through N.

TRUNC

Syntax: Trunc <u,s,c>.

Returns the largest integer which is less than or equal to its argument. For example, Trunc 2.5 returns 2, and Trunc -4.2 returns -5.

SQRT

Syntax: Sqrt <u,s,c>.

Given a nonnegative value as its argument, Sqrt returns the square root.

SIN, COS, TAN

Syntax: Sin <u,s,c>

Cos <u,s,c>

Tan <u,s,c>

Returns the sine (cosine, tangent) of the argument. The argument is considered to be in degrees. Note that Tan X is undefined when Cos X is 0.

ASIN, ACOS, ATAN

Syntax: Asin <u,s,c>

Acos <u,s,c>

Atan <u,s,c>

Returns the arc sine (arc cosine, arc tangent), also known as the inverse sine (inverse cosine, inverse tangent) of its argument. The argument of Asin and Acos must be in the range -1 to 1. (The argument of Atan may be any number.) The value returned by these functions is always considered to be in degrees, i.e. a value from 0 to 359.

HUMANSKAN, OBJECTSCAN

Syntax: Humanscan <u,s,c>

Objectscan <u,s,c>

These functions perform like the Humanscan and Objectscan commands (see below), except that instead of affecting the system variable Range, they return an argument directly, e.g.,

```
Let R = Objectscan 10
If R > 0 then Goto 100
```

is basically equivalent to

```
Objectscan 10
If Range > 0 then Goto 100
```

Technical Note: There is one difference between the function and command versions of these commands. The functional version of the scan commands will return a floating-point value, while the command version returns an integer. Note that doing an Objectscan 0 will cause the value of the system variable Enemy to be set appropriately, regardless of whether the scan is executed as a command or as a function. Also note that calling one of these functions in a Let statement *is* an action command.

• DIMENSION

Syntax: Dimension <u>(<c>)

Creates a one-dimensional array <u> of size <c>. An array must be dimensioned prior to being used. Examples:

```
Dimension someArray(50)
```

creates an array of dimension 50 for us to use.

Let A = someArray(25) * 2

assigns A a value equal to twice the value of the twenty-fifth element of someArray.

Let someArray(1) = SQR R

assigns a value (the square root of R) to the first element of someArray.

5.3.1b Flow Control

• **GOTO**

Syntax: Goto <u,s,c>

Performs an unconditional branch. Program control is passed to the line number specified by the argument. Obviously, the line number specified must exist somewhere in the program or an error will occur when the robot is compiled.

• **IF...THEN...ELSE**

Syntax: If <u,s,c> <condition> <u,s,c> Then <command>

 If <u,s,c> <condition> <u,s,c> Then <command>Else <command>

Performs conditional branching. If the condition is true, then the command following the word “Then” is executed. If the condition is not true, then nothing happens unless the second form of the command is used, in which case the command following the word “Else” is executed. Valid conditions are = (equals), < (less than), > (greater than), >= (greater than or equal to), <= (less than or equal to), and <> (not equal to). Example:

If A > B Then Zap

Goto 300

says that if the value of variable A is greater than the value of variable B, then the robot should fire its laser. Then the robot should skip to line 300 of its program. This is quite different from:

If A > B Then Zap

Else Goto 300

Note that in the first example the robot will always skip to line 300 whether it fires its laser or not, while in the second example it will fire its laser **or** skip to line 300; it will not do both.

An Else statement must follow immediately after an If statement. The following is illegal:

If A > B Then Zap

Swivel 90

Else Goto 300

Note that you cannot nest If statements directly. So you cannot write

If A > B Then If B < C <etc.>

However, you may include If statements inside of Begin...End structures and While structures (see below), which allows you to accomplish pretty much the same thing.

• **GOSUB**

Syntax: Gosub <u,s,c>

Branches to a subroutine. Similar to Goto, except that when a Return command is encountered, control returns to the first line following the most recently called Gosub command.

• **RETURN**

Syntax: Return

When encountered, Return passes control back to the line following the most recently called Gosub command. Example:

```
Let A = 0
Gosub 500
100 Goto 100
500 Let A = 5
Return
```

In this example, when line 100 is reached, the value of A will be 5.

•FOR...NEXT

Syntax: For <u> = <u,s,c> to <u,s,c><...>Next

A For...Next clause executes the lines between the For and the Next statements a specific number of times while incrementing the variable on the left side of the equals sign.

Example:

```
Let B = 0
For A = 1 to 10
    Let B = B + A
Next
```

The addition will be performed 10 times. A is incremented by 1 each time through the loop, so that by the end of the loop B will have a value of 55.

• IF...BEGIN

Syntax: If <condition> Then Begin<...>End
Else Begin<...>End

The If...Begin command is actually a special version of the If statement. Its effect is to cause the commands contained between Begin and End to be treated as one logical command for the purpose of branching. Example:

```
If Objectscan 0 > 0 Then Begin
    If Enemy = 1 then zap
End
Else Begin
    Swivel 180
    Wait 1
End
```

The effect of the above piece of code is that if the robot's scanner detects an object, then it will fire its laser provided the object is an enemy robot or decoy. Otherwise the robot will turn its head in the opposite direction and wait for one turn.

Note: You may use Begin in either part of a conditional branch; you are not required to use Begin in the Else part just because you used it in the If part, and vice-versa.

• WHILE...BEGIN

Syntax: While <condition> Begin<...>End

The While command causes the robot to repeatedly execute the commands contained between Begin and End as long as <condition> is true. If the condition is not true at the moment the While command is encountered, the entire section from Begin to End will be ignored and skipped over. If it is true, the robot will execute the commands down to End, then

jump back to the While statement. Example:

```
While 1 = 1 Begin
    While Objectscan 0 = 0 Begin
        Swivel 41
    End
    Zap
    While Objectscan 0 > 0 Begin
        Zap
    End
End
```

Except for the lack of an Equip command, the above piece of code could actually be an entire robot program. The outermost While structure (While 1 = 1) tells the robot to execute the code contained within it forever. The first of the inner While structures tells the robot to scan until it sees a target. The robot will then fire its lasers once, then repeatedly scan and zap until the target is no longer there.

Note: Use of If...Begin and While commands will allow you to write programs entirely without Goto's. Indeed, we strongly recommend that if you choose to use the structured BEGIN...END style of programming that you not use GOTOS at all.

5.3.1c Communication commands

• SEND

Syntax: Send <u,s,c> to <u,s,c>.

Sends the first argument to a robot on the same team as the sender. The second argument is the recipient's teamID. If the second argument is 0, the message is sent to all robots on the same team as the sender (other than the sender). Robots may not send messages to themselves (attempting to do so has no effect). Each robot has a FIFO (first in first out) message queue which is ten values long. Sending a message to a robot whose queue is full has no effect (the message is lost).

Note: floating point variables are truncated before being sent.

• RECEIVE

Syntax: Receive <u>.

Assigns the first value in the robot's message queue to var. The value is then removed from the message queue. Note the number of values currently in the robot's message queue may be found in the system variable Datawaiting.

5.3.2 Action Commands

• WAIT

Syntax: Wait <u,s,c>

Given a positive argument X, Wait causes the robot to do nothing for X turns. This is useful mainly when attempting to move a set distance. For example,

EngineOn

Wait 4

EngineOff

will move the robot 5 spaces in the direction of its body facing. (It moves 4 spaces while waiting, and 1 while executing the EngineOff statement.)

5.3.2a Movement Commands

• **TURN**

Syntax: Turn <u,s,c>

Causes the robot to turn its body 45°. 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. Note that when the body turns, the head turns with it the same amount.

• **ENGINEON, ENGINEOFF**

Syntax: Engineon

Engineoff

These two commands turn the robot's engine on and off. While the engine is on, the robot will move one unit per turn in the direction its body faces. This allows the robot to carry out other actions while moving. If the robot runs into the outer wall of the arena or into another robot, the engine will automatically shut off. (The status of the engine can be determined by reading the Enginestat system variable.) Each turn that the engine is on costs the robot 1 unit of fuel (2 units if the turbo booster is also on). Note that colliding with a wall will damage a robot, while colliding with another robot is relatively benign (except that the other robot will probably blast the hell out of you).

• **TURBOON, TURBOOFF**

Syntax: 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 also turned on. Robots using turbo also hit walls twice as hard... Note that the engine and the turbo booster may be switched on and off independently, but the robot will only move when the engine is on.

5.3.2b The Scanning Commands

• **HUMANSKAN**

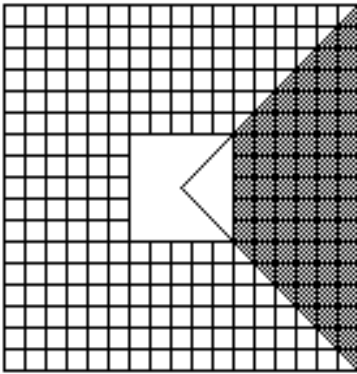
Syntax: Humanscan <u,s,c>

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.



Example of Humanscan 45
with headfacing = 90

• OBJECTSCAN

Syntax: Objectscan <u,s,c>

This command works in the same way as Humanscan, with the exception that Objectscan reports the sighting of the nearest (friendly **or** enemy) robot or enemy decoy instead of the nearest human. The distance to the nearest decoy or robot is returned in Range. If no objects are sighted, Range is set to 0. In addition, Objectscan 0 has a special effect. Not only does it cause the range of the nearest object in a straight line to be returned in Range, but it also sets the value of the system variable Enemy. If an enemy robot or decoy was detected, then the value of Enemy is set to 1; otherwise it is set to 0.

• SWIVEL

Syntax: Swivel <u,s,c>

Swivel's argument must be an integer between -360 and 360. Given an argument x, Swivel causes the robot to rotate its head x degrees (clockwise if x is positive, counterclockwise if x is negative). Example: If the robot's head is currently facing 15° degrees, then

Swivel 20

would cause it to swivel its head to 35°.

5.3.2c The Weapon Commands

• ZAP

Syntax: Zap

Zap <u,s,c>

Causes the robot to fire its lasers. If no argument is given, the robot will automatically fire all its lasers in the direction of its current head facing. If an argument is given, the robot will only fire a number of lasers equal to the argument. Each laser fired uses 1 unit of fuel; if there is insufficient fuel to execute the command then the robot will do nothing. The laser beam will hit the first item it comes in contact with, damaging it if it is a robot, destroying it if it is a decoy or a human.

• LAUNCH

Syntax: Launch

Launches a missile in the direction of the robot's head facing. If no missiles are available, the robot will do nothing.

•SHOOTBEAM

Syntax: Shootbeam

Shoots the robot's sonic beam, if it has one and it is charged. Note that the beam fires in the direction of the robot's bodyfacing, not headfacing. Also, the beam will only fire in one of the horizontal directions; if the robot's body is currently facing in a diagonal direction, it will do nothing.

5.3.2d Defensive commands

• DROPDECOY

Syntax: Dropdecoy

Assuming that the robot has a decoy available, this command causes it to drop a decoy directly behind itself. Note that the placement of the decoy is dependent on the robot's bodyfacing. Also note that decoys will be destroyed if they are stepped on.

•SETSHIELDS

Syntax: 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 control the amount of energy allocated to the robot's eight shields. The eight arguments represent the eight shields, in order, and represent the **relative** amount of energy to be allocated. For example,

Setshields 1 0 0 0 0 0 0 0

will transfer all available energy to the front shield;

Setshields 1000 0 0 0 0 0 0 0

will do exactly the same. The values of the arguments are strictly relative. 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.

Note that the Setshields command will always divide the total shield energy which is available into proportions as near as possible to those represented by the arguments. For example, if a robot has 100 points of shield energy remaining and it executes the command

Setshields 1 1 1 1 1 1 1 1

then each shield will be allocated 12 units of energy.

5.3.2e Debugging command

• POSITION

Syntax: Position <u,s,c> <u,s,c>

Repositions the robot to coordinates X,Y. Command will not execute if the designated coordinates are already occupied by a robot or if Mech Zone has been told to ignore position commands. (This is explained in the User's Guide.)

5.4 System variables

System variables are variables which may only be read by programs; they may not be

assigned values directly. They are used to keep track of information about the robot's status and the results of some commands. Many of these variables are displayed in the debugging windows.

MISSILES

The number of missiles the robot has remaining.

BEAM

This variable will be equal to 1 if a sonic beam is present, 0 otherwise.

DECOYS

The number of decoys the robot has remaining.

FUEL

The number of fuel units the robot has remaining.

RANGE

The result of the most recent Objectscan or Humanscan, that is, the distance to the nearest object or human within the area scanned.

ENEMY

Normally contains a value of 0. It will contain a value of 1 only when the last scan was an objectscan 0 and detected an enemy robot or decoy.

DAMAGE

The amount of damage the robot has sustained. When Damage equals 1000, the robot has been destroyed.

BODYFACING

The direction the robot's body is currently facing. Bodyfacings are numbered clockwise from 1 to 8, with 1 indicating straight up.

HEADFACING

The direction the robot's head is currently facing, in degrees, with 0 indicating straight up.

BEARING

The direction of the robot's head relative to its body. Measured in degrees, with 0 indicating that the head is facing in the same direction as the body.

XCOORD

The robot's x coordinate on the battlefield. Range: 1-75, with 1 being the left edge of the battlefield.

YCOORD

The robot's y coordinate on the battlefield. Range: 1-75, with 1 being the top edge of the battlefield.

ENGINESTAT

Keeps track of whether the robot's engine is on. Contains a 1 if the engine is on, 0 otherwise.

TURBOSTAT

Keeps track of the robot's turbo booster. Contains a 1 if the turbo booster is on, 0 if the turbo booster is off, and 2 if the robot is not equipped with a turbo booster.

LASERS

The number of lasers with which the robot is equipped.

DATAWAITING

The number of values in the robot's message queue.

TEAMSIZE

The number of members on the robot's team. Note that this value is *not* updated to reflect casualties.

TEAMID

The place of the robot on its team. For a team of size N, the TeamID's will be 1 through N. Like Teamsize, TeamID is not updated during battle.

Appendix A: Changes since version 1.1

Interface changes

Start Battle Dialog Box

Robots are now loaded for battle and other battle parameters are set through the Start Battle... command. The use of this command is explained fully in the User's Guide, under Running a Battle.

Follow robot by clicking on status area

If you click in the status area of a given robot, the closeup display will automatically stay centered on that robot.

New menu commands

Equipment calculator

This utility allows the user to configure robots by typing in the quantity of each piece of equipment and the amount of fuel he wants. The amount of weight remaining is automatically calculated on the fly. Clicking on the Copy to Clipboard button will generate an Equip line that corresponds to the current values in the calculator and store it in the clipboard so that it may be pasted into a robot program.

Status display addition

A small dot has been added to the display to indicate the robot's current bodyfacing.

Multi-robot battles

Battles may now involve up to eight robots. Each robot has a number associated with it on the popup menu (see below) and in the closeup view.

Select robot by clicking or by popup menu

To change the robot which is being monitored in either status display, click on the name of the robot in the display. A popup menu will appear allowing you to select another robot from among those which are still alive.

You may also change the robots in the status display by clicking directly on a robot in the closeup view. Doing so will cause that robot to be monitored in the upper status display. The robot which was in the upper display is moved to the lower display.

Teams & Communications

Up to four teams, as well as any number of independent robots, may participate in a battle. Teams are assigned through the Start Battle... command. All robots on the same team have a distinct symbol in the radar view and next to their number in the closeup view. (Independent robots are all given the same symbol, which is different from those given to teams.)

To facilitate coordination among teammates, several enhancements have been added to the RIPPLE programming language.

Send

Syntax: SEND <varA> to <varB>. Sends varA to a robot on the same team as the sender. varB is the recipient's teamID. If varB is 0, the message is sent to all robots on the same team as the sender, except for the sender. Robots may not send messages to themselves (attempting to do so has no effect). Each robot has a FIFO (first in first out) message queue which is ten values long. Sending a message to a robot whose queue is full has no effect (the message is lost).

Note: at least for the time being, floating point variables are truncated before being sent

Also note that both SEND and RECEIVE are Logic commands; i.e., they do not end a robot's turn when executed.

Teamsize & TeamID

TEAMSIZ is a system variable which contains the number of robots on a robot's team (at the **start** of the battle—TEAMSIZ is not updated to reflect casualties). TEAMID is a system variable which identifies a robot by its place on its team. For a team of size N, the teamID's will be 1 through N. Like TEAMSIZ, TEAMID is not updated during battle.

Receive

Syntax: RECEIVE <var>. Assigns the first value in the robot's message queue to var. The value is then removed from the message queue.

Datawaiting

This system variable contains the number of items currently in the robot's message queue.

Friendly

This system variable has a value of 1 whenever the last scan was an objectscan 0 **and** the object detected (if any) was an enemy robot or decoy. Otherwise it has a value of 0.

Improved Graphics with animation

Off-screen bitmaps have been implemented, eliminating (hopefully) "ghost" objects.

The images of robots in the closeup view have been changed and animated. With All Graphics set in the Graphics Display submenu, robots walk when they move, and their heads turn when they swivel. In addition, missiles now look like missiles instead of pinky balls.

Color

There is now just one version of Mech Zone, which operates in both color and black &

white mode.

Improved sound

On a Mac II or better running System 7 (or higher, one supposes...let's not get ahead of ourselves, shall we?), three-channel sound is implemented. This means that sounds will literally overlap instead of cutting each other off.

Language improvements

If...then...else

Syntax: IF <condition> THEN <commandA>ELSE <commandB>

Meaning: If <condition> is true, then execute commandA. Otherwise, execute commandB

Floating point math

Variables are no longer truncated in calculations. Instead, they contain floating values accurate up to <number> places. When passed to an action command or to SEND, variables are automatically truncated to integers. Note that floating point numbers may be entered directly; i.e. commands of the form:

LET X = 2.5

are legal. However, unary minus is still not available. You may not do:

LET X = -3

Instead, you should use LET X = 0 - 3.

Trig and algebraic functions

A number of trigonometric and algebraic functions have been implemented. They are described briefly below. Note that these functions do not take parentheses around their argument.

Trunc

Syntax: LET <varA> = TRUNC <varB>. Returns the integer portion of var. (There's a technical term for this...the opposite of mantissa, I think.)

Sqrt

Syntax: LET <varA> = SQRT <varB>. Returns the square root of varB. (Note: I haven't checked, but taking the square root of a negative number should produce a runtime error.)

Sin, Cos, Tan

Syntax: LET <varA> = SIN <varB> (etc.). Returns the sin (cosin, tangent) of varB. The argument is considered to be in degrees. (Note that TAN 90 and TAN 270 should produce runtime errors.)

Asin, Acos, Atan

Syntax: LET <varA> = ASIN <varB> (etc.). Returns the arc sin (arc cosin, arc tangent), also known as the inverse sin (etc.) of varB. The argument of ASIN and ACOS must be in the range -1 to 1. Any argument outside this range will (should) produce a runtime error. (The argument of ATAN may be any number.) The value returned by these functions is always considered to be in degrees; i.e. a value from 0 to 359.

Structured code

The syntax is as follows:

BEGIN

<statement>

<statement>

<...>

END

All statements contained between a given set of BEGIN and END statements are

considered to be part of one logical statement. Thus structures such as the following are possible:

```
IF <condition> THEN BEGIN
<...>
END
ELSE BEGIN
<...>
END
```

The above would mean that if the condition is true, the statements between the first BEGIN and END should be executed; otherwise, the statements between the second BEGIN and END should be executed.

We have also added a WHILE statement. For full details see the Programmer's Guide.

New system variable: Bearing

Because of a change in the rules concerning head position relative to body movement, a new variable has been included which we hope will prove helpful. BEARING always contains the **relative** facing of the head with the respect the body. Thus if BODYFACING is 4 and HEADFACING is 180, BEARING will contain the value 45. Note that, like HEADFACING, BEARING ranges in value from 1 to 359.

Support for teams and communications

This is discussed above under Teams and Communications.

Comments on same line as code

Comments may be placed on the same line as a line of code. All you have to do is put a backquote (`) before the comment. Everything after the backquote up to the end of the line is ignored when the robot is compiled

Optional Let

Assignments to variables no longer require the word Let. Both "LET X = Y" and "X = Y" are valid statements and mean the same thing.

Makerandom and Checkshield converted to functions

Instead of having MAKERANDOM and CHECKSHIELD assign a value to a system variable, they have been replaced by functions. LET var = RANDOM N assigns a random integer between 1 and N to var. LET var = CHECKSHIELD N assigns the current strength of shield N to var. (If N is 0, the function returns the amount of energy currently in the shield reserve. See below, rules changes.)

Rules changes

Head turns with body

Partly to make the animation of the robots in the closeup view more credible, and partly because it seems more fair, robots' heads no longer "float" independently of their bodies. When a robot turns 1 to the right, its head turns with it 45 degrees. Conversely, when a robot turns 1 to the left, the head turns -45 degrees. The HEADFACING system variable still returns the absolute facing of the robot's head in degrees; however, a new variable, BEARING, has been added, which contains the facing of the robots head relative to the body, where the current BODYFACING is considered equivalent to 0 degrees. Note that BEARING remains constant when a robot turns, while HEADFACING does not.

Because of this change, it is suggested that programmers take care when executing SWIVEL and TURN commands. If you want to turn a robot's head to a specific direction (either absolute or relative), it is best to find that direction, subtract the current HEADFACING or BEARING from it, and then swivel the resulting number of degrees. If you want to maintain a constant headfacing while manoeuvring, you must do a SWIVEL 45 after each turn to the left,

and a SWIVEL 315 (=SWIVEL -45) after each turn to the right. (If you're making several turns, it will save time if you make just one swivel adjustment at the end of your maneuver.)

Sonic beam changed

The sonic beam currently will only operate when the robot's body is facing in one of the four horizontal directions (BODYFACING 1, 3, 5, and 7). We hope to reintroduce diagonal firing of the sonic beam in the near future.

In addition, the sonic beam has been improved so that it now does 500 points of damage per blast. This will allow it to destroy its target in two blasts. However, the beam must now be charged using the Charge command, which takes a turn.

Shields changed

We have altered the way that shields interact with missiles, and we've also made it possible for robots to augment their shields using their fuel reserves. The new rules are described below.

A robot begins the battle with 1200 points of shield energy divided equally among its 8 shields. The allocation of energy may be changed via the SETSHIELD command, which is unchanged from version 1.1. (SETSHIELD 0 0 0 0 0 0 0 0 is allowed, although it isn't very useful.) There is no limit to the amount of energy which may be allocated to a single shield, except that the total energy in all shields and the reserve (see below) may never exceed 1200 points.

When a missile explodes next to a shielded robot, the missile does 250 points of damage to the facing shield. This damage is absorbed by the shield at a cost of 1 point of shield energy per point of damage. Any excess damage is immediately inflicted on the robot.

When a robot executes a SETSHIELD command, any shield energy left over after apportionment is allocated to a shield reserve. (For example, if a robot has a total of 10 points of shield energy and does a SETSHIELD 1 1 1 1 1 1 1 1, 2 points will be left in the shield reserve.) The next time a SETSHIELD is executed, any energy in the shield reserve is considered part of the total shield energy. (In other words, excess shield energy is never wasted.)

The amount of energy in the shield reserve can be found by executing a CHECKSHIELD 0 command. (Or we may have a new system variable for the reserve.)

Robots may increase the amount of energy in the shield reserve using the POWER command. (I'm not sure of the exact name yet.) POWER N will expend N points of fuel and add 5*N points of energy to the shield reserve. If doing so would cause the total shield energy to exceed 1200 points, then any excess is lost.

Zap altered slightly

It is now possible to have commands of the form ZAP N, where N is the number of lasers which you wish to fire. If N is larger than the number of lasers available, then only the number of available lasers will fire. If N is 0, the laser will still fire (make noise and produce a beam) but it won't do any damage, not even to decoys or humans. ZAP 1 is particularly useful as a tool to clear out enemy decoys, especially when dealing with limited fuel, or to kill pesky humans.

You can still use the ZAP command without an argument, in which case it will behave as it always did.

Fewer logic statements per turn

Robots now execute a maximum of 20 logic statements per turn instead of 99. A robot still automatically ends its turn after executing an action statement.

Debugging tools

A command has been added to assist in debugging, and the debuggin window has been

changed.

Position command

Syntax: POSITION X Y. Repositions the robot to coordinates X,Y. Command will not execute (is ignored—but does it still take up a turn?) if the designated coordinates are already occupied by a robot or if Mech Zone has been told to ignore position commands. (This is explained in the User's Guide.)

Debugging window

The debugging windows have been changed to include the new sytem variables. To make room, the number of debugging variables has been reduced to 9, called DEBUG1 through DEBUG9.

(At some point in the future we may add the capability to specify variables to be tracked by name.)

Bug fixes

A large number of bitmapping errors have been fixed.

The debugging windows now work properly.

A bug which caused robots to appear to occupy a discontinuous set of points (with respect to OBJECTSCAN and ZAP) has been fixed.

Appendix B: Further reading

Lem, Stanislaw.

The cyberiad; fables for the cybernetic age. Translated from the Polish by Michael Kandel. Illustrated by Daniel Mroz. New York, Seabury Press [1974]