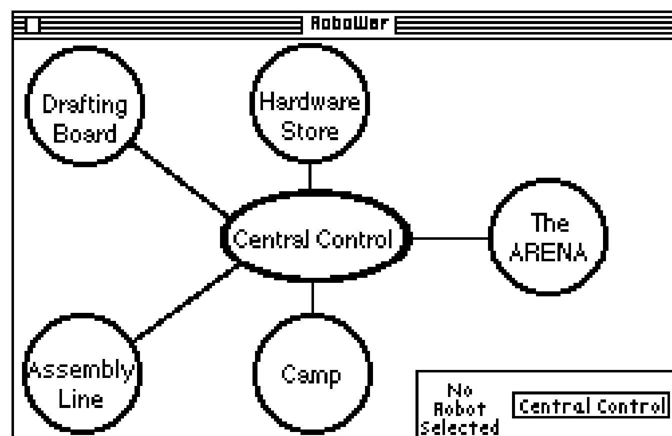
RoboWar

© 1990 David Harris

Welcome to RoboWar. In this game, you will design and program robot gladiators, then pit them against each other in a deadly battle. May victory go to the strongest!

Features of RoboWar include animated combat, color graphics on any machine with a color monitor, and a complete programming language, RoboTalk, with an editor, assembler, and interpreter.

To launch RoboWar, double click on the game or any saved robot icon. A window will appear with a number of circles connected by lines:



This window shows the Central Control station. From Central Control, you may go to any of the other five stations, the Drafting Board, Hardware Store, Assembly Line, Camp, or Arena. If you would just like to play with predesigned robots, you only need to go to the Camp (where robots are chosen for battle) and the Arena (where the battles take place). However, if you would like to try the most exciting part of RoboWar, designing your own robots, you will also need to use the Drafting Board to write your software, the Hardware Store to select various hardware additions to make your robot more potent, and to design a customized picture for your robot, and the Assembly Line, where the robot's software is assembled into executable form.

The rest of these instructions are divided into five parts. The first part is a step-by-step tutorial on writing simple robots and lead them into battle. The second part describes each of the five stations in more detail. The

third part is an introduction to RoboTalk, the language with which robots are designed. The fourth part is a reference manual on the RoboTalk assembler and interpreter, describing each instruction in more detail. The fifth part details some of RoboWar's advanced features. Finally, the appendix shows two slightly more complex robots that may serve as examples for budding RoboTalk programmers.

Robots are generally referred to as males. If you consider your robots to be female, so be it. Call me a sexist pig and substitute she for he throughout the rest of the instructions.

I. A RoboWar Tutorial

Let us begin by creating two simple robot gladiators. The first robot will just stand still and wait for other robots to shoot it. The second will be somewhat more intelligent; it will rotate its guns seeking a target. When it finds one, it will lock on and keep shooting until the target is destroyed.

If RoboWar is not running, launch it from the Finder. The Central Control window, with five circles around a central oval, should appear. The five circles are the various stations; they are where you can design and equip robots and send them into battle. In the lower right corner is a box with a message, "No Robot Selected" and a dimmed button, "Central Control." First, we must select a robot to edit.

Choose "New Robot…" from the File menu. A dialog box will appear, requesting the name of the robot. Type "Target" and click Save or press *return*. Now, a red circle and the name Target should appear in the corner where "No Robot Selected" once was. Target is the selected robot.

Now that we have created Target, we must write his software. Click on the circle labeled Drafting Board. This changes the window to show the Drafting Board, the station where robot software is designed. The left part of the window is a text editor; the right part displays some statistics about the current program.

First, type "# TARGET" and hit *return*. This line is a comment because it begins with the # symbol. It is ignored by the robot, but is useful for humans to remember which robot we're programming. Leave another blank line, then type "LOOP:" and hit *return* again. The word LOOP: is a label definition. It identifies this point in the software with the name LOOP. Indent a few spaces (one tab press is conventional) and type "LOOP JUMP". This line means that the program should jump back to the

label LOOP. Thus, the program will just go in circles, not doing any real action. The robot will just be a stationary target, a sitting duck for our next robot to track down. At this point, the completed code should read as follows.

```
# TARGET

LOOP:
  LOOP JUMP
```

If it does not, make the necessary corrections so that it does. RoboTalk, the language in which we are writing our robot in, does not care if letters are upper or lower case. However, all words MUST be spelled correctly.

When the robot's instructions are correct, click on the button, "Central Control" in the lower right corner. This takes us back to Central Control and automatically saves the robot's software. Now, click on the Assembly Line circle to go to that station.

At the Assembly Line, we assemble a robot's brain from its software. Every time we change the software at the Drafting Board, we must reassemble it at the Assembly Line. Click on the button labeled "Assemble Robot." If all goes well, the message, "Robot Assembled" will appear, along with the number of lines in the program. If there is an error, the computer will beep and mention the error and line number. If you encounter an error, return to the Drafting Board via Central Control, recheck the program, make the correction, and return to the Assembly Line to reassemble the robot. When the robot successfully assembles, return to Central Control.

Let's test Target now to see that he really just stands in one place. Click on the station Camp. A new view will appear with a roster of robots arrayed for battle. At first, there should be no robots selected. Click on the Add button to add Target to the roster. Another dialog box will appear, asking for the robot to add. Choose Target and click open or press *return*. A filled circle will appear in the first position, beside the name Target. Add another Target or two by clicking on the Duplicate button beside Add. This button adds another robot to the roster with the same name as the previously selected one. Now that some robots are ready in Camp, return to Central Control.

Now we are finally ready to go to the Arena. Click on the Arena station. A view will appear with a few robots scattered randomly around the left

and a roster on the right. The left part of the window is the actual battlefield in which combat takes place. Click on the Battle button to begin.

The robots will be rearranged in random positions on the field. They should stand still, just as our software instructs them. Beside their names, the vital statistics of energy and damage should appear. Both are set at 100. Since the robots neither expend energy for combat nor take damage from other robots' shots, both values should remain at 100. To stop the battle, click on the Halt button. The Halt button is the only button recognized during combat; all other mouse clicks are locked out. Now return to Central Control.

Target is not a very exciting robot. Let's design a more interesting one that will seek out and shoot down robots like Target. From the File menu, choose New Robot… again. Name the new robot ShotBot. (Note: the robot's name does not really matter; it could be anything you want.) Go to the Drafting Board and enter the following code:

```
# ShotBot
# Written 1/3/90 by David Harris

Main:
  Range 0 > FireSub RotateSub IFE
  Main JUMP

FireSub:
  20 fire' STORE
  RETURN

RotateSub:
  5 AIM +
  AIM' STORE
  RETURN
```

This robot is somewhat more complex than Target. The first two lines are comments, identifying the robot and his author. This is very important if somebody else was to look at your robot later; it also is a good way to feed your ego. The first label definition is Main:. Note that the name of the label definition is not important; it could be anything except a variable name or an operator (see the reference manual, section IV, for a complete list of variables and operators). However, if we changed the name Main, we would have to be sure to change the reference to it in the line Main JUMP.

The first instructions under Main check the range to the nearest target in the sights. The variable Range is either 0 if there is no target in sight or the distance to the target if one is visible. The instructions "Range 0 >"

compare Range to 0. Note that in RoboTalk, the operator, in this case >, always follows the operands, Range and 0. This is similar to some foreign languages, where the verb always follows the nouns.

The next three words, "FireSub RotateSub IFE" check the result of the comparison. If the comparison is true, if Range really is greater than 0, meaning that a target is in sight, the robot goes to FireSub, which fires a bullet. If the comparison is false and no robot is in sight, the robot branches to RotateSub, which rotates the turret, seeking a target. The IFE instruction (meaning If-Then-Else) makes this check, always branching to the first label if the comparison is true and the second if it is false.

The next line, "Main JUMP" jumps back to Main, repeating the cycle.

The subroutine FireSub is marked by a label definition (FireSub:). It stores 20 in the variable FIRE, which causes ShotBot to launch a shot of energy 20 in the direction its turret is pointing. Note the format of the STORE instruction: 1) the amount to store, 2) the variable in which to store it, 3) the word STORE. Also note that the variable is followed by a single quotation mark. The quote means that we are dealing with the variable name itself, not the contents of the variable. This is very important; all variables used with STORE must be followed by this single quote. Conversely, when you want the value of the variable, as we did when we checked the Range variable above, you must be sure not to quote the variable. The last instruction under FireSub, RETURN, returns to the point that the IF left off.

RotateSub is similar in design to FireSub. It rotates the turret five degrees, seeking the next target. First it calculates the new value of AIM, using the instructions AIM 5 +. Note that, once again, the operator (+) follows its two operands (AIM and 5). The plus adds together the previous two operands, leaving the number AIM + 5. The next line stores this number in AIM. As with FIRE, the variable AIM must be quoted because it is the target of a STORE command. Finally, RETURN returns control to the main loop.

Return to Central Control, then assemble ShotBot at the Assembly Line. When the robot assembles correctly, go to Camp. Add a ShotBot to the list of robots. Now, go to the Arena and choose Battle.

This time, the Target robots should stay in one place as before. However, the ShotBot will rotate its turret until it sights one. It will lock on, continuing to shoot until the Target is destroyed. When all of the Target

robots are eliminated, the battle will end. Also, notice how the energy and damage of each robot changes. Target uses no energy, but takes damage from each shot that hits. This damage cannot be repaired. When it reaches 0, the robot is destroyed. ShotBot takes no damage, but his energy supply usually remains very low. This is because he is putting all of his energy into shots. Although energy slowly returns over time, ShotBot uses it rapidly, thus keeping the energy near zero.

Now you have seen how to design and assemble simple robots. You know how to choose robots from the Camp and let them battle at the Arena. Two paths await you. If there are predesigned robots on this disk, you may experiment with them. See what robots are most effective against others, then try teaming them up from Camp. Perhaps you will want to tinker with their programs a bit, or modify their equipment at the hardware store. Part II describes in more depth the various stations you will use.

If you would really like to see the true excitement of RoboWar, however, you should try designing your own robots. Designing and debugging a robot is well worth the effort if it can challenge a friend's 'bot in battle. Writing robots is somewhat more of a challenge than simply using preprogrammed ones. You will want to read the next three parts. Part II describes the various stations. Part III introduces RoboTalk. It discusses the concepts of the stack, operands and operators, comments, variables, and labels. Part IV is a complete listing of all the operators and variables defined in RoboTalk. It also gives some technical information about the assembler and interpreter that are part of RoboWar. If you are ready for an exciting challenge, read on!

II. RoboWar Stations

Central Control is the master switchboard of RoboWar. From there, you can visit the five stations where robots are designed, built, and tested. All of the stations are divided into three panels. The large square panel on the left is where most action takes place in the station. The panel on the upper right shows the name of the current station and lists other important information. The panel on the lower right shows the name of the currently selected robot (used at the Drafting Board, Hardware Store, and Assembly Line), and has a button to return to central control. Each station is described below.

Drafting Board

Robot software is designed at the Drafting Board. The left panel of the section is a text editor window that performs like most other text editors. It includes cutting, copying, and pasting text, as well as keyboard editing. The *tab* key automatically indents to the nearest multiple of four spaces, to help line up columns.

The upper right panel shows the length of the robot's software and the current position of the cursor within the document. The current position is useful if an error is present in the program. The Assembly Line reports the line number of the error it detects. Thus, you may use the current position indicator to track down the line without manually counting from the start.

The Print button, also in the upper right panel, will print the entire software listing. Be sure a printer is connected and ready, then click OK when the standard printer dialog appears.

When the Drafting Board is exited or another robot is loaded, all changes to the robot's software are automatically saved.

Hardware Store

Robots are equipped with hardware options at the Hardware Store. The left panel has two possible views: the first for choosing tradeoffs; the second for designing an icon.

In the first view, seven boxes and a list of radio buttons are shown. Each radio button lets you set the state of a given hardware option. Under the Energy Max box, the four buttons, High, Normal, Low, and Very Low, correspond to the values 150, 100, 60, and 40. These values are the highest that the robot's energy may reach at any given time. They also set the robot's initial energy when a new battle begins.

The buttons of Damage Max, High, Normal, Low, and Very Low, correspond to the values 150, 100, 60, and 30. They are the initial damage ratings of a robot. When the damage rating drops to zero, the robot is reduced to a smouldering heap of scrap, and is removed from the arena.

The buttons of Shield Max, High, Normal, Low, and None correspond to the values 100, 50, 25, and 0. The Shield Max defines the highest level at which shields may be set without causing great drain on the robot's energy resources.

The buttons of Processor Speed, Fast, Normal, and Slow, define how many instructions can be executed by the robot during a chronon, or turn. Fast allows 15 instructions, Normal 10, and Slow 5. A robot with a faster processor can get more done during its turn than a robot with a slower one. This can be very useful if the robot has to execute a very complex program.

The Bullets box defines the type of bullets that a robot shoots. Normal bullets move across the screen at a speed of 12 pixels per chronon. If they hit their target, they do damage equal to the energy put into them. Exploding bullets move like normal bullets, but explode when they impact. The explosion grows at a rate of 5 units per chronon until it detonates with a radius of 30. All robots caught in this blast radius take damage equal to one and a half times the energy put into the bullet. Rubber bullets behave just like normal bullets, but only cause half damage when they hit.

The Missiles box allows a robot to be equipped with missiles. Missiles only move at a velocity of 5 pixels per chronon. However, if they hit their target, they cause damage of one and a half times the energy put into them.

TacNukes, short for Tactical Nuclear Devices, are another form of weaponry. TacNukes are placed at the location of the owner when he fires them; they do not move at all. Instead, they expand like exploding bullets, but to a radius of 50 pixels. All within the blast radius, including the robot who set them if he is foolish enough to not move away rapidly, are affected and take damage equal to one and a half times the initial energy investment.

Two buttons at the bottom of the screen allow editing the current robot's picture, or icon. The Design Icon button brings up the second view, the Icon Editor, while the Delete Icon erases any icon that the robot might have.

The second view shows the robot's icons magnified to 8x normal size. Each robot has two icons: one with shields activated and the other without. If no icon is designed, a robot uses a default picture. The second view may show either the shield or shieldless picture of the robot. A label above the picture notes this view. Three buttons at the bottom aid in editing. The first, titled Shields or Shieldless, transfers to the other picture. The second, Copy Icon, copies the present picture onto the other icon. This is useful if a robot was supposed to look similar with and without shields. The shieldless icon could be drawn. Then the Copy button could be used to copy the icon over to the shield picture. Now, the shield picture would only need a little editing, instead of recreation from scratch. The third

button, Save Icon, saves the current icons and returns to the first view of advantages and disadvantages.

The large square in the icon editor view is a thirty-two by thirty-two array of squares. The central circle shows the radius through which the turrets rotate and bullets may impact. The area outside the circle is unaffected by oncoming shots. Thus, the size or shape of a robot does not affect its chances of being hit by a bullet. Clicking in the square allows drawing or erasing pixels.

The following note is for advanced programmers; it may be blissfully ignored unless you are really unusual. Advanced programmers may wish to use a different icon editor to design their robots. Any other editor is acceptable. If another editor (such as ResEdit) is used, the final icons must be copied into the robot's resource fork. The shieldless picture must be saved as ICON ID#1000, while the shield picture must be ICON ID#1001.

The upper right panel of the Hardware Store displays status information about the selected hardware. It lists the maximum values of energy, damage, and shields. It notes the processor speed, type of bullet, and whether missiles or TacNukes are enabled. It also keeps an account of the number of advantages and disadvantages that the robot has. Each change from normal equipment (except a redesigned icon) is worth a certain number of advantage or disadvantage points. Any combination of points may be chosen. However, a robot may not have more advantages than disadvantages.

Beneath the advantages and disadvantages list is an area titled Icons. If an icon has been designed, it will appear in the Icons area. The area will also show the turret, as it rotates around a complete circle. There may be two icons. The left one is the shieldless picture; the right is the icon with shields enabled.

When the Hardware Store is exited or another robot is selected, all changes are automatically saved.

Assembly Line

At the Assembly Line, the software of a robot is assembled into a form that can execute most efficiently in the Arena. After each change at the Drafting Board, the robot MUST be reassembled for the changes to take effect during combat.

The right panel displays the dates of last changes and the length of the robot's software. A button at the bottom labeled Assemble assembles the software. If the assembly is error-free, a message appears in the middle of the panel saying that the assembly is complete and reporting the number of lines assembled. Otherwise, a message notes that there is an error. It describes the kind of error and the line of source code in which it is present.

Note that pressing ⌘-A is a substitute for clicking the Assemble button.

Camp

Teams are chosen and robots are arrayed for battle at the Camp. The left panel shows a roster of up to six robots, and a set of buttons for teaming these robots.

First robots must be added to the roster. Click on the Add button or press ⌘-A. A dialog box will appear. Select the robot to add. It will appear on the list and the position with the robot will be highlighted. This highlighted line means that that robot is selected. The other commands, Duplicate (⌘-D), Remove (⌘-R), Alone (⌘-0), Team 1 (⌘-1), Team 2 (⌘-2), and Team 3 (⌘-3), affect only the selected robot. Clicking on a different robot selects it instead; clicking on a blank line selects no robot at all.

Duplicate makes a copy of a robot. If several of the same model of robot are to be used for testing purposes, Adding one and Duplicating him multiple times may be more convenient than having to add the same robot several times. Remove deletes a robot from the list of combatants.

Double-clicking on a robot in the roster makes it the currently selected robot for the Drafting Board, Hardware Store, and Assembly Line. The name of the robot will replace the old selection in the box on the lower right beside the Central Control button.

The bottom row of buttons deal with teams. Teams of robots can work together. They do not see each other in their sights, so they will not intentionally shoot at each other. Teams of robots can also communicate with each other using the Channel and Signal variables, described in part IV. Alone places a robot on no team at all. Team 1 through 3 set robots on those teams.

At least two robots must be chosen from the Camp before they can fight in the Arena.

The Arena

Robot gladiators fight their battles at the Arena. The left panel, a 300 by 300 rectangle, is the pit in which the robot fight. They must be careful not to run into the walls; doing so can greatly damage an unwary robot. Each of the robots is drawn in the arena. When a battle starts, each is placed in a random location.

The right panel lists the robots in combat, along with their icons. If a robot is on a team, his team number appears beneath his name. Beside the robot is his current Damage and Energy ratings. If a robot is destroyed in battle, the stats are replaced by the message “Deceased.”

Clicking on the Battle button (or pressing ⌘-B) starts a battle. The robots continue fighting until only one victor is left. If you wish to terminate the battle (perhaps a team of two friends are remaining or perhaps you are tired of the carnage or lack thereof), click on the Halt button. During battle, the Halt button is the only control enabled; clicking elsewhere on the screen has no effect.

III. An Introduction to RoboTalk

RoboTalk is the language in which robots are programmed for RoboWar. The language is based on Reverse Polish notation (RPN), similar to the HP calculators. This section introduces a number of important RoboTalk concepts: the Stack, operands and operators, comments, tokens, delimiters, and programs, numbers, label definitions, labels, and variables. It also describes the hardware interactions that robots have, including the effects of energy, shields, weapons, and collisions.

The Stack

The stack is the basis of all RoboTalk instructions. It works like a stack of plates: a new plate may be placed (pushed) onto the top of the stack or removed (popped) off of it. All operands, such as numbers, variable names, and labels are pushed onto the stack. Operators may pop information off the top of the stack, act on it, and push any result back onto the stack.

The most recently pushed operand on the stack is called the top of the stack. A new operand pushed onto the stack is placed above the old top and becomes the new top of the stack. An operand is popped off the top of the stack, leaving the operand beneath it as the new top. If more than 100 operands are on the stack at once (an unlikely occurrence), the robot has a Stack Overflow error. If one tries to pop a number off of the stack when it is entirely empty, a Stack Underflow error is reported.

Tokens, Delimiters, and Programs

A token is a group of characters, usually a number or word. Every element of RoboTalk: numbers, label definitions, labels, variables, and operands, are really tokens. Each token must be separated from the next by at least one delimiter.

A delimiter is a symbol that separates two tokens. The most common delimiters are spaces and new lines. For example, in the line "Main JUMP" the space separating Main and Jump is a delimiter. Other valid delimiters include the tabs, semicolons, and commas. RoboTalk does not care which delimiters are used where. Thus ShotBot could be rewritten as:

```
Main:,Range;0 >;FireSub
RotateSub,IFE,Main;JUMP;FireSub:;;;20 fire'    STORE,RETURN,RotateSub:
 5 AIM + ,AIM' STORE;RETURN
```

However, this is very unclear for the reader. Therefore, a style similar to that shown in the original ShotBot, with spaces and new lines used for delimiters, and indentation following label definitions, is recommended.

A program is a series of tokens that work together to control a robot. Each robot's software is a single program. The segments branched to by IF and CALL operators are called subroutines. Each of the types of tokens is described below.

Comments

Comments are messages that a human may use to help understand a program, but that are ignored by RoboTalk. For example, in ShotBot, described in Part I, the line "`# Written 1/3/90 by David Harris`" is a comment. It reports to the user information about the robot's author, but does not generate any code that RoboTalk interprets.

Comments come in two varieties for convenience. The first variety begins with a # sign. This comment means that the rest of the line is a comment

and should be ignored. The comment might come at the beginning of the line, or might follow some real code, as in the line:

```
AIM 5 + AIM' STORE  # Rotate Turret.
```

The other form of comment is marked by the { and } symbols (open and close braces.) The open brace indicates a start of the comment. The comment is ended by the close brace. There must be a close brace for every open brace. However, comments may be nested; that is, a pair of open and close braces may appear between another set of braces. The following program shows an example of comments marked by braces:

```
# An example robot

{   This is the beginning of a comment.
  It continues on the next line.
     {This comment is nested within the outer one
  } The previous comment is closed.

main: # this is not a label definition; it is enclosed in a comment
}

main:  # this is a valid label definition
  main jump
```

Label Definitions

Label definitions mark a point in the program so that jumps and branches may go to that point. A label definition consists of a word followed by a colon. Label definitions do not generate any RoboTalk code. However, they are used to mark the destination of labels in other parts of the program.

Label definitions should not have the same names as variables or operators. Also, there must not be two label definitions with the same names in a single program.

Operands and Operators

All tokens other than comments and label definitions are either operands or operators. An operand is a number, label, or variable name that is pushed onto the stack. An operator is a command that acts on the stack. The various operands and operators are described below.

Numbers

A number is just that: a collection of digits. Numbers are always pushed onto the stack when they are encountered. Numbers may range from -19999 to 19999. They may have a plus or minus sign in front of them; however, there must be no delimiters between the sign and the digits. (If there were, RoboTalk would interpret it as an operand, either plus or minus, followed by a positive number.)

Labels

Labels are used with the IF, IFE, JUMP, and CALL operators. They are coded as the location in the program to jump to, and thus are pushed onto the top of the stack when encountered.

Variables

Variables, also known as robot registers, contain information from the robot. They include the range to the nearest robot in the sights, a robot's current X and Y position, energy, etc. A complete list of variables and their functions appears in Part IV.

Variables may appear in two forms in a program: unquoted and quoted. An unquoted variable really refers to the contents, or value, of that variable, while a quoted variable indicates the variable name itself. Quoted variables (written as the variable name followed immediately by a single quotation mark e.g. AIM') are used with the STORE operator, as they are the location in which a number should be stored. Quoted variables must be used with every STORE operator, and should not be used with any other expression. For example, although "5 AIM' +" will successfully assemble, the results will be meaningless and will probably cause the robot to do strange things, because the number 5 is being added to the name of the variable AIM, not the contents of AIM.

Operators

Operators may pop information off of a stack, act on it, and push information back onto the stack. A complete list of operators appears in Part IV. They include mathematical functions like +, -, and *, stack manipulation functions like DROP and SWAP, and branching functions like JUMP and CALL.

Advanced programmers should note that all branches that may be returned from, namely, those generated by IF, IFE, and CALL, push the return address onto the stack. Thus if a subroutine were to be written that would

act on information on the stack, it would first have to save the return address before acting. Then, it would have to restore the return address before making a RETURN statement. The following code, which invokes a subroutine to double the number on the stack, demonstrates this technique:

```
# Demonstration Robot

Main:
  5            # the number to double
  DoubleSub CALL # double it
  DROP         # discard it
  Main JUMP    # repeat forever...

DoubleSub:
  r' STORE     # store the return address in the variable r
  2 *          # multiply the top number on the stack by 2
  r            # restore the return address to the stack
  RETURN       # and return
```

An alternative method of writing DoubleSub would use stack manipulation commands, saving an instruction:

```
DoubleSub:
  SWAP         # swap return address and number to multiply
  2 *          # multiply the top number on the stack by 2
  SWAP         # swap return address back to top of stack
  RETURN       # and return
```

<u>Sample Program</u>

Now that we have learned all the elements of a program, let us see how they act on the stack in a simple robot. The following robot remains stationary, rotating its turret:

```
# SimpleRobot

Main:
  Aim 5 +      # Rotate 5 degrees
  Aim' STORE
  Main JUMP    # Repeat
```

Suppose that the variable Aim currently contained 90 degrees, due east. Let us trace the stack through each instruction:

```
Instruction: # SimpleRobot
Type: Comment
Stack: Empty

Instruction: Main:
Type: Label Definition
Stack: Empty
```

Instruction: Aim
Type: Unquoted variable = 90
Stack: 90 <-- Top of Stack

Instruction: 5
Type: Number
Stack: 5 <-- Top of Stack
90

Instruction: +
Type: Operator
Stack: 95 <-- Top of Stack

Instruction: # Rotate 5 Degrees
Type: Comment
Stack: 95 <-- Top of Stack

Instruction: Aim'
Type: Quoted Variable
Stack: Aim' <-- Top of Stack
95

Instruction: STORE
Type: Operator
Stack: Empty

Instruction: Main
Type: Label
Stack: Main <-- Top of Stack

Instruction: JUMP
Type: Operator
Stack: Empty

The program repeats this cycle, incrementing Aim by 5 degrees each time through the loop.

Hardware Interactions

The robot interacts with his environment by reading and writing variables. For example, a robot could check his X position by reading the variable X, or could fire a bullet by writing some amount of energy to the Fire variable. Bullets, missiles, and TacNukes were described in Part II, and in more detail in Part IV. This section gives an overview of the robot's energy and damage statistics, as well as the effect of collisions with other robots and walls.

The robot's energy, listed in the upper right box during combat, is the total amount of power available to the robot. It is used for acceleration, maintaining shields, and shooting weapons. Each chronon the energy recharges by two points.

Large amounts of energy consumption may place a robot at negative energy. While a robot has negative energy, it cannot move or interpret any instructions. It is just a sitting duck until the energy returns to the realm of positive numbers.

The robot's damage, also listed in the upper right box during combat, is how much damage may still be taken before the robot is destroyed. Damage does not regenerate; once a robot takes damage, the damage is present until a new battle starts.

Collisions are the bane of any moving robot. Two types of collisions exist: collisions with walls and collisions with other robots. Collisions with walls are the most dangerous, but also the most avoidable. If a robot hits a wall, it takes five points of damage per chronon until it moves back into the main arena. However, checking the X and Y positions of a robot, and adjusting velocity if the robot nears a wall, should prevent this kind of collision.

Collisions with other robots are sensed in the Collision variable. It returns a 1 if read while the robot has hit another. Each chronon that the robots are touching causes 1 point of damage to both combatants. Furthermore, the robots cannot move through each other, so they may end up locked together until one or the other perishes. However, a smart moving robot will check the collision variable frequently. If he has collided, he may jump to some code that locates and destroys his opponent.

IV. RoboTalk Reference Manual

This manual is divided into four sections. The first describes the format of the code that the assembler produces. It is only of technical interest; it is not necessary to program a robot. The second describes each of the RoboTalk operators. The third describes each of the variables RoboTalk supports. Finally, the fourth describes the interpreter's action.

The Assembler

At the Assembly Line, the RoboTalk assembler converts the source code written in the Drafting Board into a series of two byte instructions, the object code, that the interpreter can execute most rapidly. The following information describes the format of the code the assembler creates. It is only of interest for advanced programmers; others may skip to the Operators section below, which describes each of the RoboTalk operators.

Comments are left out of the object code. Label definitions (a label followed by a colon) are also not included. Labels (such as MAIN in the instructions MAIN JUMP) are converted into numbers, indicating the instruction number that the label definition preceded. Numbers in the range of -19999 to 19999 are placed in the object code as simple numbers. Variables are converted to a coded form, from 20300 to 20399. If a variable is not quoted, a RECALL operator is generated after the variable. Finally, operators are also converted into a coded form, from 20000 to 20199. Following the very last instruction, a 20110, or END instruction is generated. This alerts the interpreter if the end of the robot's code has been encountered.

As an example, let us see what kind of object code the following simple program would produce:

```
# Example
# Written 1/2/90 by David Harris

LOOP:
        5 aim + aim' store
        LOOP jump
```

The introductory comments are ignored (just like in speeches!). The label definition LOOP: also generates no code. Thus instruction 0 is the number 5. The next token is aim. Since it is an unquoted variable, it expands to two instructions: the code for the variable AIM followed by the code for the operator RECALL. They are, as noted in the sections on operators and variables below, 20330 and 20109, respectively. The next token, +, is converted into its code, 20000. Aim' is a quoted variable, so only the variable code, 20330, is generated. The token store is an operator, so its code, 20100, is produced next. The next token, the label LOOP, generates the address of the first instruction following the label definition LOOP:. This instruction is number 0, the very first one in the program. Therefore, a 0 is generated for the LOOP label. Finally, the token jump is another operator and its code, 20104, is produced. At the end of the program, the END instruction, 20110, is appended. Thus the object code consists of the following string of integers:

```
5 20330 20109 20000 20330 20100 0 20104 20110
```

This code is written to the robot's resource fork as RCOD ID#1000. Also, the length of the object code is written to the resource fork as RLEN ID#1000.

If none of this example made any sense, relax. It is not necessary to write a good robot. It is only included as a point of interest. Advanced programmers may use the knowledge to examine the robot’s resource file (with ResEdit or another resource editor) and check the code produced. A really warped programmer might work out the addresses of parts of his object code to produce a jump table instead of a set of labels, so that the jump address could be calculated, rather than using a series of IF constructs to make the jump. If anyone ever has to do this kind of exotic programming, however, this author will be very impressed.

RoboWar imposes a few limits on the size of a robot’s program. First, the source code may not exceed 32767 characters. No problem! There may be no more than 100 label definitions in any one program. Also, the assembled program may not have more than 500 instructions. The stack may have no more than 100 operands on it at any one time. If any of these limitations become a problem, a long walk in fresh air is recommended.

Operators

This section lists each operator. It gives the name, the code number generated, and the effect it has on the stack or robot state. The operands are as follows:

+	-	*
/	>	<
=	!	STO / STORE
DROP	SWAP	ROLL
JUMP / RETURN	CALL	DUP / DUPLICATE
IF	IFE	(RECALL)
(END)	NOP	AND
OR	XOR / EOR	MOD
BEEP	CHS	NOT
ARCTAN		

+: Code 20000
Adds the top two numbers on the stack, removes them, and replaces them with the result.
Ex: “4 5 +” leaves 9 on the top of the stack.

-: Code 20001
Subtracts the top number from the second number on the stack, removes them, and replaces them with the result.
Ex: “9 3 -” leaves 6 on the top of the stack.

*: Code 20002
Multiplies the top two numbers on the stack, removes them, and replaces them with the result.
Ex: “2 4 *” leaves 8 on the top of the stack.

/: Code 20003
Divides the second number by the top number on the stack, removes them, and replaces them with the result.
Ex: “22 3 /” leaves 7 on the top of the stack. (7.3333 is truncated to 7)

>: Code 20004
Checks if the second number on the stack exceeds the top number, then removes them. If the second number is greater, it places a 1 on the stack, otherwise it pushes a 0.
Ex: “5 4 >” leaves a 1 on the stack.

<: Code 20005
Checks if the second number on the stack is less than the top number, then removes them. If the second number is less, it places a 1 on the stack, otherwise it pushes a 0.
Ex: “7 3 <” leaves a 0 on the stack.

=: Code 20006
Checks if the top two numbers on the stack are equal, then removes them. If they are equal, it places a 1 on the stack, otherwise it pushes a 0.
Ex: “2 2 =” leaves a 1 on the stack.

!: Code 20007
Checks if the top two numbers on the stack are not equal, then removes them. If they are not equal, it places a 1 on the stack, otherwise it pushes a 0. (The symbol ! comes from the logical NOT command in the language C.)
Ex: “5 5 !” leaves a 0 on the stack.

STO or STORE (either token is allowed): Code 20100
Stores the second number on the stack in the variable specified at the top of the stack, and removes both the number and variable reference. The operand on the top of the stack must be a quoted variable or an error is reported. Also see the section on variables below, as values cannot be stored in some variables, such as RANGE.
EX: “20 aim’ store” stores 20 in the variable AIM and leaves nothing on the stack.

DROP: Code 20101
Drops the top element from the stack.
EX: “5 DROP” leaves nothing on the stack.

SWAP: Code 20102
Swaps the top and second elements on the stack.
EX: “1 2 SWAP” leaves 1 at the top of the stack and 2 in the second position.

ROLL: Code 20103
Rolls the second element of the stack back the number of places specified by the top operand, then removes the top operand.
EX: “1 2 3 4 5 2 ROLL” rolls 5 back 2 places, leaving 1 2 5 3 4 on the stack.

JUMP or RETURN (either token is allowed): Code 20104
Jumps to the instruction number specified by the top element of the stack, removes the top element, and resumes execution at the new instruction. The same operator, usually written with the name RETURN, returns after a subroutine call made by IF or CALL by jumping to the return address that the IF or CALL left on the top of the stack.

CALL: Code 20105
Jumps to the instruction number specified by the top element of the stack, removes the top element, places the return address (the instruction number previously being executed) on the top of the stack, and resumes execution at the new instruction. Very similar to JUMP, but leaves the return address on the stack.

DUP or DUPLICATE (either token is allowed): Code 20106
Duplicates the number on the top of the stack.
EX: “5 DUP” leaves 5 on the top of the stack and 5 in the second position.

IF: Code 20107
Checks the second operand on the stack. If it is not zero then it leaves the return address on the stack and jumps to the label specified on the top of the stack. In any case, it removes the second operand and the destination label from the stack.
EX: “32 MySub JUMP” jumps to the subroutine MySub and leaves the return address on the stack.

IFE: Code 20108

Stands for IF-THEN-ELSE. Checks the third operand on the stack. If it is not zero than it leaves the return address on the stack and jumps to the label specified in the second position on the stack. If it is zero then it leaves the return address on the stack and jumps to the label specified on the top of the stack. In any case it removes the first, second, and third elements from the stack.
EX: "0 SubA SubB IFE" jumps to the subroutine SubB and leaves the return address on the stack.

(RECALL): Code 20109
This instruction cannot be entered from the source code. Instead, it is automatically appended after each unquoted variable. It gets the value of the variable specified on the top of the stack, removes the variable name, and places the value on the stack. See the section below on variables for information about their values.

(END): Code 20110
This instruction cannot be entered from the source code. Instead, it is automatically placed at the very end of a program. If it is reached, the interpreter reports that the end of the robot's code has been reached and ceases the battle.

NOP: Code 20111
No OPeration. Does nothing whatsoever, except take up time and space. May be used when some timing loop is necessary.
EX: "NOP" leaves nothing on the stack.

AND: Code 20112
Checks if the top two numbers on the stack are both not zero, then removes them. If they are both not zero, it places a 1 on the stack, otherwise it pushes a 0.
EX: "2 3 AND" leaves a 1 on the top of the stack.

OR: Code 20113
Checks if either of the top two numbers on the stack is not zero, then removes them. If either is not zero, it places a 1 on the stack, otherwise it pushes a 0.
EX: "0 4 OR" leaves a 1 on the top of the stack.

XOR or EOR (either token is allowed): Code 20114
Checks the top two numbers on the stack, then removes them. If one or the other, but not both, are not zero, it places a 1 on the stack. Otherwise, it pushes a 0.

EX: “1 2 XOR” leaves a 0 on the top of the stack.

MOD: Code 20115
Performs a modulus operation (remainder of integer division) on the top two elements of the stack. Removes them and returns the result on the stack.
EX: “10 3 MOD” leaves 10 mod 3 = 10-3*Trunc(10/3) = 1 on the stack.

BEEP: Code 20116
Beeps once. Most useful in debugging a robot.
EX: “BEEP” leaves nothing on the stack.

CHS: Code 20117
CHange Sign. Multiplies the top operand on the stack by -1, removes it, and returns the result on the stack.
EX: “3 CHS” leaves -3 on the stack.

NOT: CODE 20118
Logical Not. Checks top operand, removes it. Returns 1 if it was 0, 0 otherwise.
EX: “4 NOT” leaves 0 on the stack.

ARCTAN: CODE 20119
Inverse Tangent. Computes the inverse tangent of the ratio of the top two numbers. The y value must be the top operand; the x value must be the second operand. ARCTAN removes the top two operands and returns the arctangent of y/x. The result is in degrees between 0 and 359, with 0 degrees pointing up, just as with AIM angles.
EX: “-5 0 ArcTan” leaves 270 on the stack.

Variables

Each robot has a number of registers or variables. They are initialized to their appropriate values, or 0 if none is appropriate, when the battle starts. This section lists each variable, its code number, its use, and whether it can be read or written. The registers are:

A-Z	FIRE	ENERGY
SHIELD	RANGE	AIM
SPEEDX	SPEEDY	DAMAGE
RANDOM	MISSILE	NUKE
COLLISION	CHANNEL	SIGNAL
MOVEX	MOVEY	RADAR

A-Z (except X and Y): Codes 20300-20325 (except 20323 and 20324)
User-defined variables. They may be used for any temporary storage that the robot needs. They may be read or written.

X: Code 20323
X position of robot. May range from 0 to 300 (the boundaries of the board). 0 is the left side; 300 is the right. X may be read but may not be written (no unrestricted teleporting!).

Y: Code 20324
Y position of robot. May range from 0 to 300. 0 is the top; 300 is the bottom. Y may be read but not written.

FIRE: Code 20326
Used to shoot bullets. Returns 0 if read, shoots bullet with energy investment equal to amount written. This energy investment is removed from the robot's energy supply. It may exceed the robot's current energy value (placing the robot at negative energy and immobilizing it), but may not exceed the robots energy maximum. Depending on the settings from the Hardware Store, bullets may be normal, rubber, or explosive. Explosive bullets explode like TacNukes in a 30 pixel radius when they hit their target. Whey they detonate (6 chronons after impact) they do damage of 1.5*energy investment to all robots in the blast radius. Normal bullets do damage equal to the energy investment when they hit their targets. Rubber bullets only do half damage if they hit. Bullets move across the screen at a speed of 12 pixels per chronon, heading in the direction that the robot's turret pointed when the shot was fired.

ENERGY: Code 20327
Robot's current energy. May be read, but not written. ENERGY returns the amount of energy the robot currently has. If not used for other purposes, energy is restored at 2 points per chronon. However, if the energy ever drops below 0, the robot does not interpret any more instructions or perform any more actions until the energy exceeds 0 again. When the battle begins energy is set to the maximum energy value specified in the Hardware Store.

SHIELD: Code 20328
Robot's current shield level. May be read or written. If read, it returns the current level of the shield, or 0 if no shields are up. If written, is sets the shield level to the value written. If the current level is less than the level written, a point of energy is used for each point added to the shields.

If not enough energy is available to set the shields, the shields are only strengthened as far as remaining energy permits. If the current level is greater than the level written, a point of energy is regained for each point of difference, although energy cannot exceed the maximum energy value set in the Hardware Store. Shields can absorb damage from bullets, missiles, or TacNukes that otherwise would have been deducted from a robot's damage score. Each point of damage that is done deducts one point from the shield level, until no power is left in the shields. The remaining damage is then done to a robot's damage score. Even if shields are not hit, they decrease by one point each chronon from natural energy decay. Shields may be charged above the maximum shield value set in the Hardware Store (although they may never exceed 150), but if they are above maximum, they decrease by two points instead of one per chronon. Shields are set to 0 when the battle begins.

RANGE: Code 20329
Range to nearest target in sights. May only be read. If there is a target in the direction the robot's AIM points, RANGE returns the distance. Otherwise, it returns 0.

AIM: Code 20330
Angle turret points. May be read or written. The angle is in degrees, oriented like a compass with 0 degrees pointing upward and 90 degrees pointing to the right. All bullets and missiles are fired in the direction that the turret is pointing.

SPEEDX: Code 20331
Speed of robot in left-right direction. May be read or written. Positive speeds move right, while negative speeds move to the left of the screen. If SPEEDX is read, it returns the current velocity; if it is written, it sets the velocity. Speeds must be in the range of -20 to 20. Each point of change in speed costs 2 points of energy; thus going from 10 to -2 costs 24 energy.

SPEEDY: Code 20332
Speed of robot in up-down direction. May be read or written. Positive values move down, while negative values move up. SPEEDY has the same limits and characteristics as SPEEDX.

DAMAGE: Code 20333
Robot's current damage rating. May only be read. When the battle begins, the damage rating starts at the maximum value set at the Hardware Store. Damage caused by bullets, missiles, and TacNukes that is not absorbed by

the robot's shields is removed from the damage rating. When it reaches 0, the robot is dead.

RANDOM: Code 20334
A random number from 0 to 359. May only be read.

MISSILE: Code 20335
Used to shoot missiles. Returns 0 if read, shoots bullet with energy investment equal to amount written. This energy investment is removed from the robot's energy supply. It may not exceed 50; if it does, only 50 energy is used. Missiles do 1.5*energy investment in damage if they hit their targets. Bullets move across the screen at a speed of 5 pixels per chronon, heading in the direction that the robot's turret pointed when the shot was fired. Missiles cannot be used unless they were first enabled at the hardware store.

NUKE: Code 20336
Used to place TacNukes, or Tactical Nuclear Devices. Returns 0 if read, places TacNuke with energy investment equal to amount written. This energy investment is removed form the robot's energy supply. It may exceed the robot's current energy value (placing the robot at negative energy and immobilizing it), but may not exceed the robots energy maximum. TacNukes begin to explode as soon as they are placed, increasing in radius by 5 pixels each chronon. At the tenth chronon, when they have a radius of 50, they detonate and cause 1.5*energy investment in damage to all robots in the radius. Robots who lay TacNukes are well advised to hasten away and be out of the blast radius when the devices explode. TacNukes cannot be used unless they were first enabled at the hardware store.

COLLISION: Code 20337
May only be read. If another robot has collided with the current robot, the COLLISION variable returns 1; otherwise it returns 0. When a collision with another robot takes place, both robots take one point of damage each chronon until they separate. They may either separate by changing direction, or by blowing the rival to little pieces.

CHANNEL: Code 20338
The robot's broadcasting and receiving channel. May be read or written. If it is read, it returns the current channel. If it is written, it sets the channel. The channel must be in a range of 1 to 10. Communications over a given channel only affect robots on the same team. Thus, a robot must be

placed on a team with at least one other robot at Camp if communications are to have any effect.

SIGNAL: Code 20339
The signal value on the robot's current channel. May be read or written. If it is read, it returns the last value broadcast over the channel by any robot on the same team. If it is written, the value written is broadcast over the channel and may be read any time in the future by any other robot on the same team. Typically signals and channels are used by two or more robots to coordinate movement or team up against another set of robots.

MOVEX: Code 20340
Used to move the robot a given distance in the X direction without changing SPEEDX. Returns 0 if read, moves the robot the specified distance if written. Movement costs two points of energy per unit moved. The distance must be between -20 and 20 units.

MOVEY: Code 20341
Used to move the robot a given distance in the Y direction without changing SPEEDY. MOVEY has the same characteristics and restrictions as MOVEX.

RADAR: Code 20343
Range to nearest bullet, missile, or TacNuke in the path of AIM. May only be read. RADAR checks a path 40 degrees wide centered on the AIM. It returns the distance to the nearest bullet, missile, or TacNuke in this path. If there are none, it returns 0. Note that the weapon detected might be moving perpendicular to the aim, not toward the robot.

Interpreter

The RoboTalk Interpreter, which interprets the assembled code as the battle takes place in the arena, looks up and executes several instructions each chronon. Depending on the processor speed, set through the Hardware Store, it may complete 5, 10, or 15 cycles each chronon.

Each cycle, the interpreter fetches and handles one instruction. The one apparent exception is the unquoted variable, which takes two cycles because it really consists of a variable name followed by a RECALL operator. Operands, such as numbers, labels, and variable names, are just pushed onto the stack. Operators perform actions on the data on the stack and may return information to the stack, as discussed above.

Robots with faster processors can perform more actions each chronon, and thus interpret complicated code at reasonable speed. However, they still only recover energy at the rate of 2 points per chronon, so they may find themselves consuming power faster than is practical.

Part V: Advanced Features

With the introduction of RoboWar version 1.5, two advanced features have been added: automated combat and password protection.

Password protection is quite simple to use. In the Drafting Board, choose the Add Password button. A dialog will prompt you for the password; a second one will verify that you typed your password correctly. Once a robot has a password, the password must be entered to open the robot. This way, other people may watch your 'bot in combat but may not view the source code. If you forget your password, send me your disk with the robot and five dollars. I'll fix the robot if it is possible.

Robots with passwords are stored with their source code encrypted. This prevents people with a sector editor from viewing your code illegally. Furthermore, the password is encoded to make breaking into robots an even greater puzzle. I challenge all the intrepid RoboWar hackers to try breaking in; I will recognize in the RoboWar hall of fame the first person to break the code. Be careful, though. You are likely to corrupt the robots if you play around with their passwords from ResEdit or another resource editor. Thus I would suggest working only on backup copies. I personally dislike passwords and feel that anyone who messes up his robot while fooling with passwords deserves his fate.

The second advanced feature is automated combat under the Tournament menu. This command prompts the user for an ASCII TEXT file with a list of combatants. The text file must have been created with a separate text editor such as your favorite word processor or programming environment. It lists the file to save the results in, the number of times each group of robots should fight, and the individual robots to battle. A sample file appears below:

```
SAVE MY BATTLE RESULTS

3
TimBot
Stationary
DumBot

Pearl
```

Lich
Aeneas III
Silo IV

2
Blade
Freud

The first line, “SAVE MY BATTLE RESULTS” is optional. If you want a record of the results, type “SAVE” followed by the file name to which RoboWar should write the results. Note that RoboWar will overwrite an old file with the same name. Leave a blank line after the SAVE directive, then specify the number of times that a group of robots sould fight. For instance, the first group, TimBot, Stationary, and Dumbot, will battle 3 times. If you ommit the number of times, as is done in the second group, RoboWar defaults to one combat. Note that the combats must involve from two to six robots each.

Automated combat is strictly an advanced feature for people trying to run large tournaments. The error checking is of marginal quality, so be sure that your battle lists are in the proper format.

Appendix A: A Brief History of RoboWar

Many moons ago, the world was lacking RoboWar. Then one crisp spring night a bunch of fanatical teen-age computer programmers were sitting around at a meeting of the Ridgecrest IEEE Student club tossing out programming ideas and dodging the shrapnel. Sick of hand-eye arcade games, the concept of robots fighting without user intervention grabbed their attention. Thus was born RoboWar.

The computer club members debated many ideas. At first they considered placing robots on a board full of obstacles and challenges. They thought about building robots by dragging icons about to form various subsystems. Jon Richards was the first to prototype RoboWar code on a foul MS-DOS clone. Later that summer at Caltech, David Harris saw an early version of another robot-battle style game being developed by students there. He liked the concept of the programming language for robots and saw a number of other ideas for improvement.

In the fall, while climbing Dragon Peak out of Onion Valley in the Sierra Nevada, David was hit across his head with a fascinating, efficient, and easy to implement programming language based on Reverse-Polish notation. Fortunately, another computer club veteran, Ralph Giles, was also on the

hike and the two worked out the details of the language, with inspiration from the HP-28 calculator language, C, Pascal, and assembly language in addition to their own foul concoctions. Fortunately, they wrote a complete synopsis of the interpreter, for, although Ralph spent a bit of time working on it on another clone, it wasn't until December that David began to implement it on a real machine.

David first designed an interpreter and compiler that ran, albeit clumsily, on a Silicon Graphics workstation. Then he ported it to the Macintosh, built up a user interface, and spend most of Christmas vacation chasing bugs. When the program worked reliably, progress greatly slowed, for he spent extensive time "playtesting" instead of coding.

Since that point, robots have evolved through a number of stages. This history details the development of code by the IEEE club members. Doug Harris, David's demented brother, also built a number of robots, ranging from unreliable to excellent in quality, that followed some convergent and some divergent branches of evolution. The IEEE club robots have come through six generations:

First generation robots include MoveBot (listed below) and DumBot. They were written to test the interpreter on the Silicon Graphics. They duel nicely but fairly randomly. Each was generally less than a page in length and wasn't smart enough to kill its opponent in a single shot. TimBot (also listed below), a simple but effective robot, lead the second generation of robots into existence. Written by Tim Seufert on the Macintosh, it locked onto stationary robots and blew them to tiny bits. Originally, it used less energy in its shots and took longer to eliminate its opponents, but still, TimBot made completely stationary robots unviable.

In response to TimBot, drooling hackers started to produce a third generation of robots. These typically moved about the board. One interesting pair of third-generation robots were Coroner 1 and 2. These 'bots moved to opposite corners before shooting at any targets they saw. If they began taking damage through their shields, they would flee to the opposite corners.

Matt Sakai changed the entire course of robot evolution when he brought Silo IV to a meeting on a nondescript floppy disk. Silo IV, the one and only fourth generation robot, moved about the board and almost unfailingly destroyed its first, second, and third generation opponents. For a time, the IEEE group was stunned; David even added some more features

to the interpreter that robot designers might try because Silo IV seemed to be the ultimate robot in the evolutionary tree.

Nonetheless, after a few weeks, numerous other robots using similar tactics sprung from the silicon mind. These fifth generation robots, the so-called “Silo clones”, eventually managed to equal, then best Matt’s Silo IV. Among the Silo clones were Robot B2 (the product of David Wasserman’s nightmares), TimBot IV (at first the joke of the IEEE group, later a reasonably effective robot), Freud (most famous for his remarkable icon), and Blade (the best Silo clone yet developed). To test these robots, the IEEE members ran extensive combats and developed the “TimBot Test:” how many plain-vanilla TimBots on one team can one robot defeat at one time in at least 60% of the combats? Most Silo clones could almost always defeat a single TimBot and could defeat two more than 60% of the time.

Fortunately, before things became too dull, robots using other strategies began to appear. Among these sixth generation robots were Lewis Girod, Aeneas II, and Pearl. Aeneas broke the three TimBot barrier. Unfortunately for his creator, Pearl appeared soon after, usually capable of overcoming five other TimBots teamed together in a battle! Pearl, another product of Matt Sakai, is the current champion.

What is the future of RoboWar? Pearl appears to be very good but Silo at one point also seemed unbeatable. Few robots have been written to work effectively together on teams; few use the communications or advanced projectile detection capabilities. Perhaps somebody will develop tracking algorithms or discover another excellent algorithm lurking just out of sight. The IEEE group has proposed a number of questions about robot behavior that they still have not answered, so there is still ground to explore. For the novice, RoboWar has much excitement waiting as one learns to program and to overcome each generation. For the expert, too, the horizons of RoboWar are still beckoning.

In such a state RoboWar rests April 16, 1990. I will be sponsoring a RoboWar tournament in early June in which the best robots of each contestant may compete for glory and riches. See the about box for more details. Also, please pay your shareware fees if you enjoy RoboWar. I put a great deal of time and effort into programming. Since the $10 fee is so much less than the cost of a professionally marketed game, I would appreciate the money.

History Continued. On June 1st, (tonight, as I write) we are collecting the robots for the tournament. We received 11 entries at $2 each. Matt introduced his reputedly fearsome seventh generation “Religion” robots-Hinduism, Judaism, and (The Great God) Anything, replete with wild comments as well as powerful new designs. Yes, there are robots better than sixth generation Pearl clones! We are waiting for the robots from Northwestern University (hopefully Jon will send some) and plan to hold the competition very soon. YAWP!

Appendix B: Version History

Robots are not the only beast to be evolving with time. The RoboWar application has gone through numerous versions, expanding features, patching bugs, and introducing new and wonderful bugs. This version history records the changes from version 1.5 onward.

Version 1.5 (May 18, 1990)

This version introduces two major new features: automated combat and password protection. In anticipation of the upcomming tournament, I added these features to help run the large number of combats and to keep everyone’s code secret. Notice: I hate passwords, so if anyone has their robot garbled by the password protection, I’ll just laugh. Version 1.5 also adds the “Don’t show battle” menu option. This speeds up battles significantly, especially on a color monitor. After extensive testing I found that 256 color mode slowed RoboWar down the greatest amount during most combats; however, a large number of bullets on the screen at one time (as used by Matt’s Pearl and his seventh generation robots) slows the game even further. Worse yet, not showing the battle scarcely improves the performance of missiles. I dread running large numbers of combats with his robots. They usually take at least ten minutes. A final change: a number of intrepid RoboWar hackers discovered a method of cheating to create nearly invincible robots by going through a back door in the program. Unfortunately for them, I now check for this method of cheating and tweak any robots who violate the rules. Don’t worry: you won’t stumble upon this method of cheating by accident.

Version 1.5.1 (May 28, 1990)

This version is a bug fix on version 1.5. Most significantly, I found that the DUP command does not work in the old versions. This causes great problems and confusion for robots using DUP. Now DUP behaves correctly. I also fixed some glitches in the password protection, regarding opening robots. I also added tallying features for group battles to the

automated combat menu. Finally, I updated these instructions to include some of these advanced features.

Appendix C: Sample Robots

This appendix lists two robots that might provide helpful examples. The first one demonstrates wall detection in a moving robot. At the hardware store, its Damage Max should be set high, Shield Max low, and rubber bullets and missiles should be chosen.

```
{ MoveBot
 Created 11/21/89 by David Harris.
}

START:
   3 speedx’ store
   2 speedy’ store
   25 shield' store

MAIN:
   aim 5 + aim' store                # Rotate Turret
   x 50 < xmin if                    # X minimum
   y 50 < ymin if                    # Y minimum
   x 250 > xmax if                   # X maximum
   y 250 > ymax if                   # Y maximum
   range 0 > shoot if                # Shoot if range >0
   25 shield' store
   main jump

XMIN:
   random 3 mod 1 + speedx' store
   return

YMIN:
   random 3 mod 1 + speedy' store
   return

XMAX:
   -1 random 3 mod - speedx' store
   return

YMAX:
   -1 random 3 mod - speedy' store
   return

SHOOT:
   energy 2 / missile' store         # Missiles must be enabled
   energy fire' store
   return
```

The second example was the first robot my friend Tim Seufert designed. It smacks of too much Douglas Adams science fiction, but nonetheless is quite effective against stationary targets, as it locks on, shooting until its target is

destroyed. Note that the following changes should be made in the hardware store: Shield Max None, Explosive Bullets Enabled, Fast Processor.

```
#  Tim's Robot
#  Designed by the same person
#   Who, at the present, due to the presence
#   of altogether too many Electric Monks,
#   believes that he is a banana and that Dodo
#   is more powerful than Mac II, thereby
#   causing his programming ability to
#   deteriorate.  Oh well.

   random aim' store

Main:
   range 0 = rotate shoot ife
   main jump

rotate:
   aim 17 + aim' store
   return

shoot:
   energy 20 > reallyshoot if
   return

reallyshoot:
   energy fire’ store
   return
```