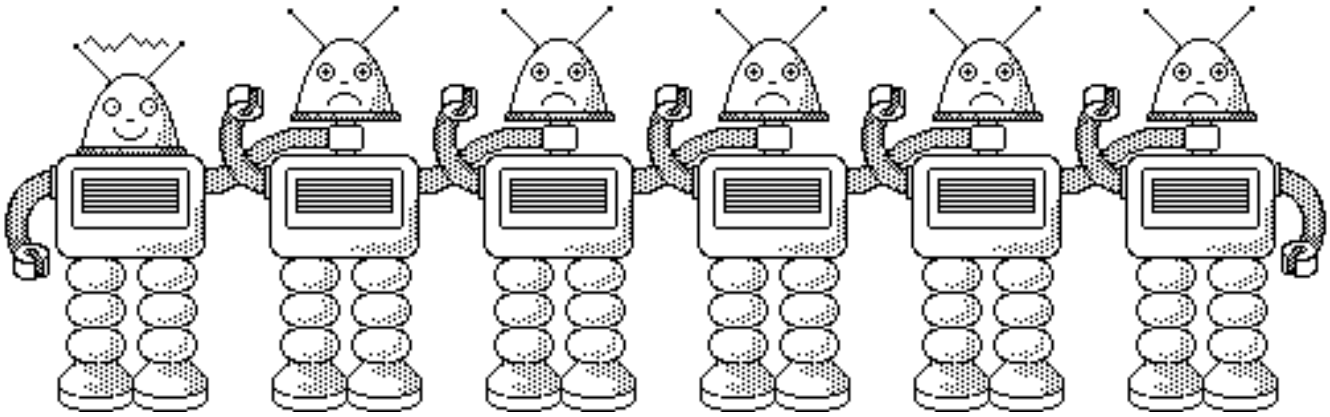
Robot Warriors 1.0.1

Program by Steven Miller and Dean Ouchida
Documentation by Steven Miller
9/11//91

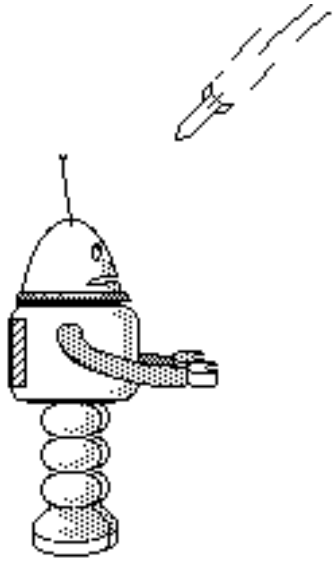


Introduction

Welcome to Robot Warriors, the game that let's you design and program your own robot to fight in a high-tech battle arena. Using a special robot programming language, you program your robot with the built in text editor and pitch it against other robots in the battlefield. A maximum of five robots can fight it out at the same time where only the best designed robot survives!

The built in text editor is all that you need to program your robot. When you're ready to try out your robot, you can compile your program to check for errors or pitch it against other robots in the battlefield to see how well you've done. The Robot Warriors debugger allows you to watch your program execute while a battle progresses allowing you to easily find problems with your robot. Up to 1000 battles can be run in succession to give you a good picture of how robots stack up against each other. You can pitch your robot against those that others have developed or use the ones supplied with Robot Warriors.

An explanation of all the menu commands of Robot Warriors is given in Appendix A for quick reference. If you are an experienced Macintosh user, you will already know how to use many of the features of this program. If you are new to using the Macintosh, you may want to complete Apple's introductory tutorials on the Macintosh before attempting to use this game which requires the use of menus, dialogs, and simple text editing. In any case, to find out more about any of the menus and commands available in Robot Warriors, refer to Appendix A for more information. The remainder of this manual is devoted to designing, programming, and testing your new robot.



Robot Warriors Registration Form

This program is being distributed as **shareware**. If you enjoy this program and would like to join the official list of "Robot Warriors" , please fill out this registration form and send it in with your registration fee of $15. Sending in a registration means you now own your copy of Robot Warriors and allows us to keep in touch with owners if a new version becomes available, if a disk of the best available robots becomes available, or to announce Robot Warriors programming competitions. If you don't find yourself using this program but would like to comment on it anyway, please send us your comments, we'd love to hear from you. Also, if you've developed robots that you'd like to share, send them in too!

- Steven Miller & Dean Ouchida

Name and Address: _____
_____
_____
_____

Phone: _____
Electronic Mail System and Address: _____

How did you obtain Robot Warriors? _____
How long have you been using Robot Warriors? _____
Have you written your own robot programs? _____
Would you consider entering a Robot Warriors programming contest? _____

What do you like most about Robot Warriors:

_____
_____
_____
_____
_____

What do you like least about Robot Warriors:

_____
_____
_____
_____
_____
_____

What features would you most like to see added to Robot Warriors:

_____
_____
_____
_____
_____
_____

Send this form and your $15 registration fee to:

Robot Warriors

14818 NE 78th St.
Vancouver WA 98682

Overview of Robot Warriors

The robot that you program in Robot Warriors has several different components:

CPU - This is the brain of the robot, it executes the instructions of your program which control the robots functions.

ENGINE- This gives the robot the ability to move giving you control of the speed and direction of the robot. Movement is often needed to escape, attack, and to avoid being easily shot by other robots. One danger of moving is the chance of colliding with other robots or with the walls of the battlefield. Your program has to take this into account and avoid collisions. Robot movement uses fuel proportional to the distance traveled.

FUEL- Robots have a limited fuel supply. Once it is used up the robot can neither move, nor cloak itself. The robot can monitor its fuel supply and choose to conserve it.

GUN- This is the only weapon the robot has. It shoots exploding shells in a high trajectory a specified distance. The shells don't collide with a robot to do damage, they hit the ground and explode and any robot that is close enough gets damaged. A direct hit on top of a robot causes the most damage. The gun has an unlimited number of shells, but it takes time to reload. Also the robot monitors the progress of the shell and therefor can have only one shot in progress at a time. The range of the gun is effectively unlimited.

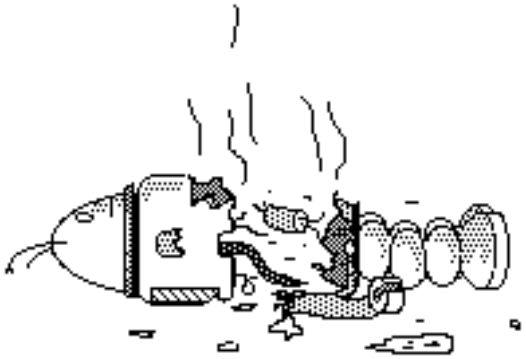
RADAR- This is equivalent to the robots eyes. The robot has two types of radar. The proximity radar "sniffs the air" and reports how many enemy robots are on the battlefield, it doesn't have the capability of determining the positions of the enemy robots though. This is done by the aiming radar which is specifically aimed in a certain direction and it reports whether a robot is spotted and how far away that robot is. This radar is used to give the gun directional and distance information.

CLOAKING- Robots have the capability to cloak themselves. This makes them invisible from all of the other robots radars. The cloaking device uses fuel to operate. Used too much, and the fuel will run out before the enemy is destroyed leaving your robot very vulnerable.

ARMOR- Robots are protected by heavy armor plate which allows them to sustain several direct hits from an enemy robot before being destroyed.

SENSORS- The robot has many sensors:

- Fuel sensor - reports the amount of fuel remaining.
- Damage sensor - reports the amount of damage from 0 to 100% damaged.
- Speed sensors - report the current X and Y velocities.
- Position sensors - report the current position of the robot.
- Gun Reload sensor - signals if the gun is reloaded and ready to fire.



Not all robots are created equal. Many of the attributes of a robot can be chosen by you, using a "shopping list" of features that you can choose from. This allows you to customize your robots and match it's capabilities with the strategy you plan to employ. There are 7 different robot attributes that you specify and you are allocated 6 points that you can "spend" to beef up any of the 7 attributes.

Attribute	**Cost**
CPU_SPEED:	0 points: 1000 robot instruction cycles per second. 1 point: 2000 robot instruction cycles per second. 2 points: 4000 robot instruction cycles per second.

A large CPU_SPEED will normally cause a robot battle to proceed more slowly due to the increased number of instructions that must be executed per second.

All robot CPU instructions take one instruction cycle to execute except for three:

1) Using the aiming radar takes 40 CPU cycles.
2) Firing the gun takes 10 CPU cycles.
3) Rectangular to polar conversion takes 50 CPU cycles

FUEL_CAPACITY:	0 points: 1850 units of fuel. 1 point: 2500 units of fuel. 2 points: 3200 units of fuel.

The rate of fuel consumption is proportional to speed of the robot: 1 unit of fuel per second for every meter per second the robot is travelling. A robot travelling at 20 meters per second will use 20 units of fuel per second.

When a robot is accelerating, fuel is used at the same rate as the speed the robot is accelerating to.

Cloaking burns 80 units of fuel per second.

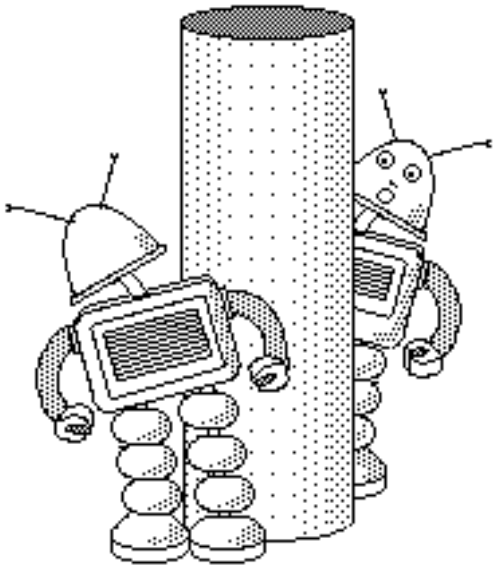
ENGINE_SIZE:	0 points: Robot can travel up to 20 meters per second. Accelerates at the rate of 2 meters/second every 1/10 of a second (20 m/s^2). Robots decelerate at the same rate at which they accelerate.
	1 point: Robot can travel up to 40 meters per second, plus accelerate at twice the normal rate (40 m/s^2).

FIRE_RATE:	0 points: robot can fire once every 2.8 seconds. 1 point: robot can fire once every 2.1 seconds. 2 points: robot can fire once every 1.5 seconds. 3 points: robot can fire once every 1.1 seconds. 4 points: robot can fire once every 0.7 seconds.

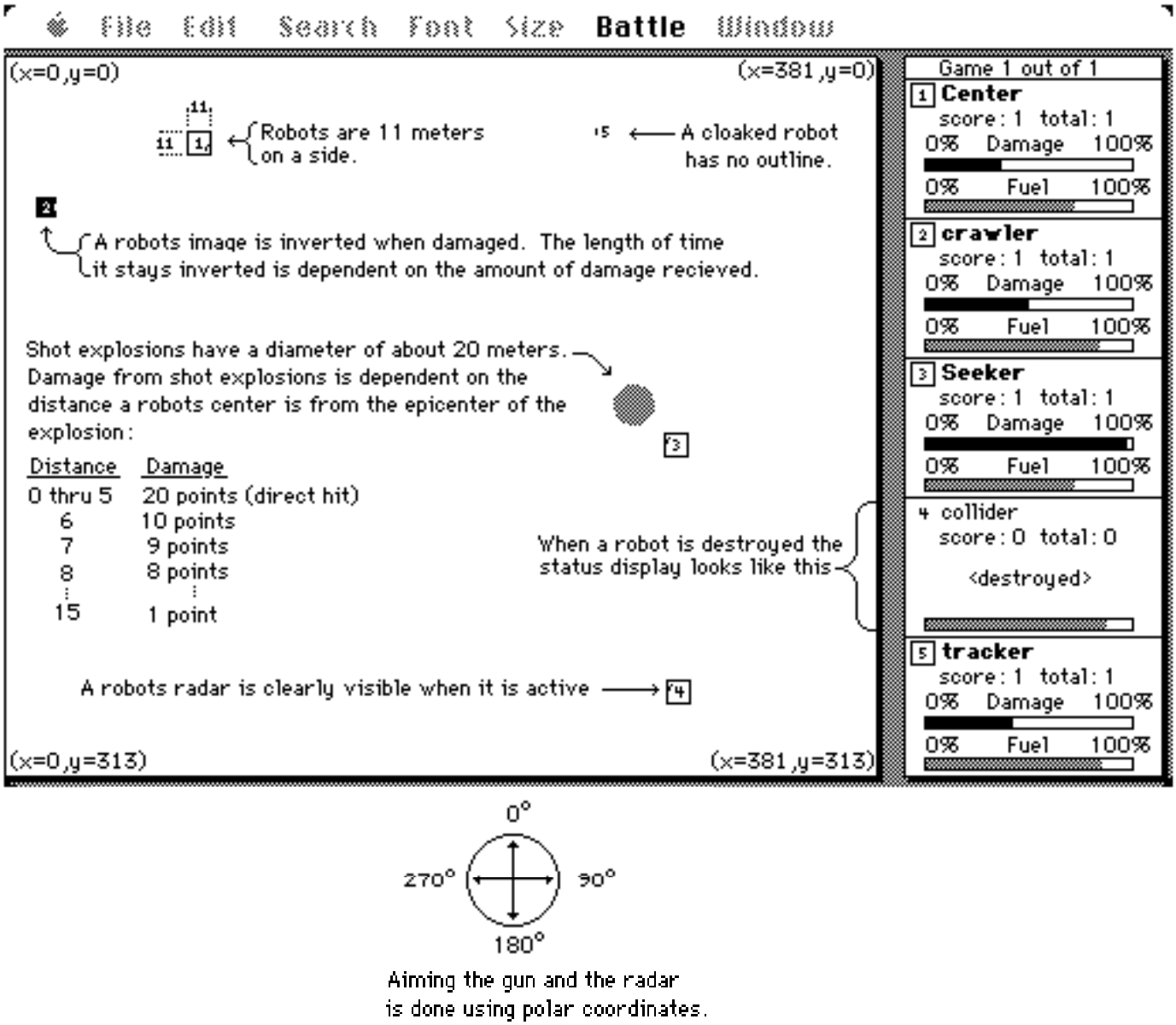
Note: Only one shot can be in the air (in progress) at one time. So if you are shooting a long distance you may not be able to re-fire at you maximum rate. It takes about 2.1 "seconds" for a shot to travel the entire length of the battlefield. A "second" is a "robot battlefield second" which is typically longer than a real second on a Mac Plus or Mac SE, and less than a second on a faster Macintosh.

CLOAKING:	0 points:	Cloaking is not available.
	1 point:	Cloaking is available. This gives you the capability to jam the enemies radar rendering you invisible to the other robots' aiming radar. You are still visible to the proximity radar. While cloaking is active, 80 units of fuel are burned per second.
RADAR_RANGE:	0 points:	Gives your aiming radar a maximum range of 200 meters.
	1 point:	Gives your radar a range of 310 meters.
	2 points:	Gives the radar a range of 500 meters, covering the whole battlefield.
ARMOR:	0 points:	A robot can sustain 100 points of damage before being destroyed.
	1 point:	gives 115 points of protection.
	2 points:	gives 130 points.
	3 points:	gives 145 points.
	4 points:	gives 165 points.
	5 points:	gives 200 points.

You have only 6 points to allocate the the features described above. You can't have the best of everything so carefully choose the attributes that your robot will have, matching them with the strategy and tactics that you program into your robot.



Before going any further, a picture of the robot battlefield with descriptions of it's components is in order:



Aiming the gun and the radar is done using polar coordinates.

Figure 1

Even though the battlefield extends from (0,0) to (381,313), a robot can only travel within the coordinates of (5,5) to (376,308) because the robots position is measured from the center of the robot and a robot's sides are approximately 5 meters from the center.

Robot scores are displayed for the current battle and the total score for the current series of battles. A robot scores a point whenever it survives the destruction of another robot. Thus in a maximum battle of 5 robots, the winning robot will receive 4 points, the runner up will receive 3 points, and 0 points is awarded to the first robot that is destroyed. When battles with more than two robots are run, robot scores will typically be fairly close unless one robot is substantially better than the others. Therefore, when running battles with several robots, it's usually necessary to run a number of battles (20 or more) before a winner can accurately be picked.

When a robot is using it's cloaking ability, the rectangular outline of the robot will disappear.

When a battle is started robots are randomly placed on the battlefield within one of 5 locations:

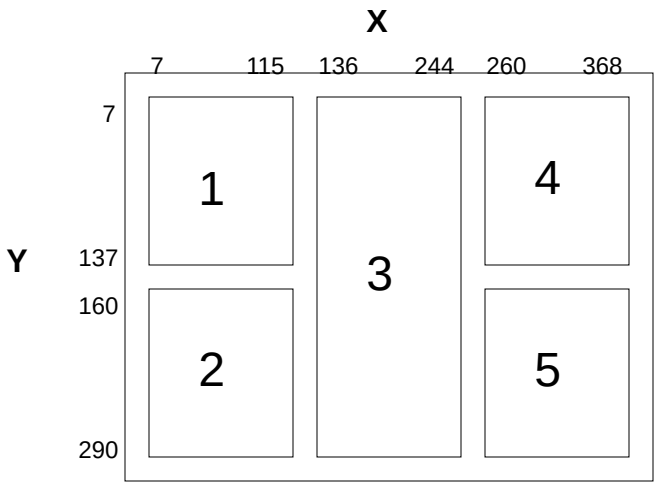


Figure 2

This figure shows the boundaries of the boxes that a robot may be placed in. At the beginning of each battle a box will be randomly chosen for a robot and the robot will be randomly placed within the box. Two robots are never placed in the same box. This assures that a robot will never be boxed in by other robots when the battle begins.

The CPU Registers

The robots CPU contains many different registers including:

- Control registers that control specific robot functions (radar, gun, engines etc..).
- Sensor registers that contain the values from the robots sensors.
- Registers that perform mathematical conversions from rectangular to polar coordinates and back.
- CPU registers that are used by the robot to execute instructions. (Accumulator, Program counter, Stack Pointer etc...).

Some registers are "read only" registers. This means that they contain data about the status of the robot that cannot be modified by the program. They can only be looked at (read) and can't be written to.

X,Y:	Current position of the robot. (X = horizontal position, Y = vertical position). These are "read only" registers.
AIM:	Aims the gun. Specified as an angle in degrees clockwise from vertical. Any angle is accepted with angles outside of 0 thru 359 being converted to an equivalent angle from 0 to 359.
SHOT:	Shoots the gun when written to. The number written to the register is the distance the shell is to travel before exploding. Reading a value from this register gives the state of the guns reloading. A zero means that the gun is ready to fire while any other number gives the approximate number of tenths of a second remaining before the gun will be ready to fire again. Shells travel at 180 meters per second. Shooting takes 10 robot instruction cycles.

DAMAGE: This read only register reports the amount of damage the robot has sustained (in %). 0% damage means no damage has been sustained. If 100% damage is reached then the robot is destroyed. Damage is sustained from a robot being shot, or colliding with a wall or with another robot. Damage from collisions is proportional to the speed of the robot during the collision. This damage is sustained at the rate of about 1 point of damage for each 6 meters/second the robots are travelling. For instance if two robots collide, one travelling at 20 meters/second and the other at 40 meters/second, they will both sustain (20+40)/6 = 10 points of damage.

SPEED: Controls the speed of the robot. Only positive values are accepted since the direction is controlled by the DIRECTION register. The value is also restricted to the maximum speed obtainable by your robot. If a value outside of this range is stored in the SPEED register, it will be reset to the closest legal value. If a robot collides with anything then the speed (this register) is reset to zero. The speed is in whole units of meters per second. The maximum speed is dependent on the amount of points allocated to ENGINE_SIZE, as described earlier.

DIRECTION: Used to controlling the direction the robot will travel. Like AIM, DIRECTION is specified as an angle in degrees clockwise from vertical. All values are accepted with numbers outside of the range 0 to 359 being translated into the equivalent angle.

FUEL: This "read only" register reports the number of units of fuel left in the tanks.

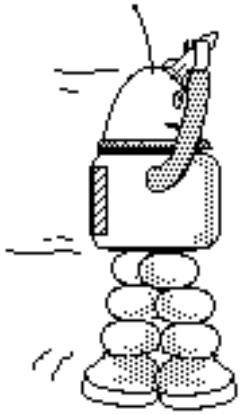
CLOAK: Turns on/off cloaking. A zero written to this register turns off cloaking, any other value turns cloaking on. Cloaking uses 80 units of fuel a second.

NUM_ROBOTS: This read only register contains the results from the proximity radar. It reports the number of robots on the battlefield. Cloaked robots are detected with this radar.

X_CONV, Y_CONV, DIST_CONV, ANGLE_CONV: These registers are used to convert rectangular to polar coordinates and vica-versa. More about these registers is described in Advanced Programming Features.

TIME_INT_MASK: A zero is written to this register to turn off time based interrupts. Any other value turns on time based interrupts. When on, an interrupt occurs 10 times a second. (Once every 1/10th of a second). Interrupts are advanced programming concepts that aren't required in order to have a competitive robot.

TIME_INT_XFER: This register contains the address of the routine that is to be executed when an interrupt occurs.



RADAR: This is the aiming radar. A value written to this radar aims it at the given angle sending a beam out at that direction with the result of what it sees written back to the register. The result can then be read from the register. A negative value signals that nothing was spotted. A positive value means that a robot was spotted at the specified distance. The register accepts the angles from 0 to 359 degrees. Any other angle is converted to the equivalent angle form 0 to 359. The radar has a limited range unless extra points are assigned to the RADAR_RANGE attribute.

The closer a robot is, the wider the angle that it can be spotted at. Robots that are right next to each other can spot each other at about a +/- 32° angle, but robots that are 400 meters apart will only be spotted if the radar is pointed to within 1 degree of it's position. The width of this angle approximately follows the function:

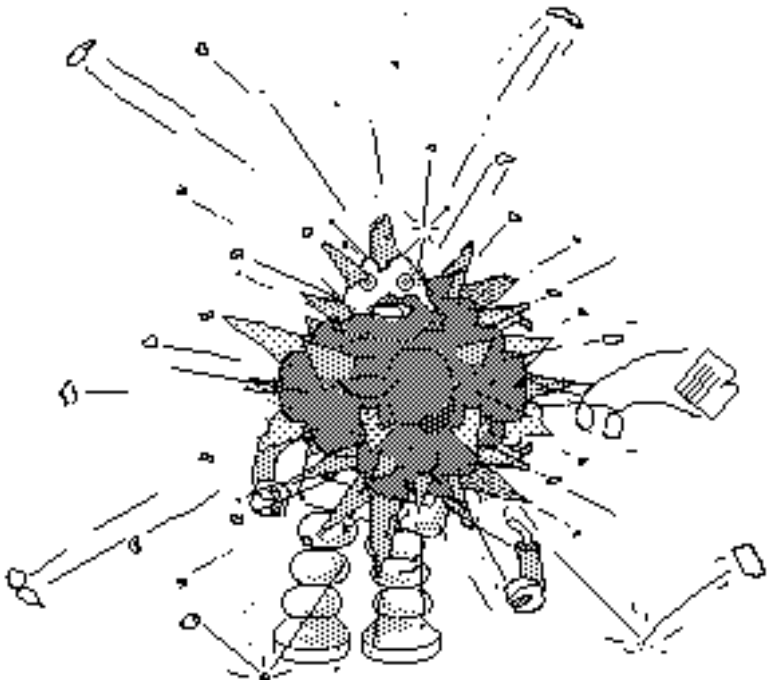
Visible Angle From Robot Center = +/- (310/distance + 1)

This means that in order to scan for robots that are 310 or more meters away and be guaranteed to spot them, the radar's angle should be incremented no more than 2° at a time.

Each use of the radar requires 40 instruction cycles. See Appendix D: "The Internal Operation Of Robot Warriors" for more information on robot instruction cycles.

RANDOM: Writing to this register sets the range of the random numbers from 0 to N (N = the number written to it). Reading from the register produces a random number from 0 to N. A new random number is produced each time the register is read.

STACK: This register should be used only by experienced programmers. It allows values to be pushed and popped off of the CPU stack. There are many things that can be done with this register. Return addresses can be popped off of the stack, or parameters to subroutines can be pushed onto the stack. Writing to the register pushes the operand onto the stack. Reading the register reads and pops a operand off of the stack.



The Robot Programming Language

While reading about programming robots, it's important to understand that it's possible to create a robot using only a small set of the robot programming language. Start with simple robots to get a feel for the game and then slowly add features to your robot. Much can be learned by studying some of the simple robots supplied with Robot Warriors.

You enter your robot program with the built in text editor. If you are familiar with text editing or word processing on the Mac at all, you should have no problem with this editor. Program's are entered by simply typing them in and using the mouse with the standard copy, cut, and paste commands you're familiar with. Opening, closing, and saving files is done in the same way as other Mac programs. You will also find several features to make entering programs a little easier such as Find and Replace functions, block commenting and indenting, and auto indentation. These features are discussed in Appendix A: "Menu Commands".

The editor allows tabs which are produces using spaces, hitting the tab key simply produces enough spaces to move to the next tab stop. Tab stops are defined at every four space interval. A reverse tab is entered by hitting shift-tab. This will back up to the previous tab stop.

The programming language used is unlike any other language available. It most closely resembles BASIC but it doesn't require line numbers. In many other ways it resembles assembly language. The language is case insensitive, it doesn't matter whether an word or letter is upper case or lower case. (Ex: "AIM" is the same as "Aim" or "aim") Using a combination of upper and lower case can make your programs more readable.

Robot Attributes:

The robot attributes are the configurable list of features described earlier and are set using the following statements:

```
CPU_SPEED  1              ;Assign 1 point to cpu_speed.  Legal range: 0 to 3
FUEL_CAPACITY  0          ;Assign 0 points to fuel capacity. Legal range: 0 to 2
ENGINE_SIZE  0            ;Legal Range: 0 to 1
FIRE_RATE  3              ;Legal Range: 0 to 4
CLOAKING  1               ;Legal Range: 0 to 1
RADAR_RANGE  1            ;Legal Range: 0 to 2
ARMOR  0                  ;Legal Range: 0 to 5
```

All unspecified attributes are assigned to zero. Up to 6 points of attributes can be allocated to a robot.

Comments:

Comments are used to document your programs to make them easier to understand. A comment is started with the semicolon (;) and any text following a semicolon up to the end of the line is considered a comment and is ignored by the compiler. Comments can be on the same line as a program statement.

```
;This is a comment.  The compiler will ignore this!

   0 to SPEED   ;Comments are also allowed to follow program instructions too.
```

Variables:

Program variables (memory) provide a way of storing information to be used later in your program. Variables are defined by allocating them. Up to 200 memory locations are available for your use. Variables are defined with the following statement:

allocate <variable name> [optional: number of memory locations]
<variable name> is any alphanumeric string beginning with a letter and may also contain the underscore character. The name can be any length but only the first 15 characters are used by the compiler.

[optional: number of memory locations] must be a positive number that defines how many memory locations to allocate. The <variable name> will refer to the first memory location only. Excluding this parameter means that one memory location is being allocated.

Examples:

```
allocate  old_damage          ;Allocate one memory location for a variable called "old_damage"
allocate  small_array  10     ;Allocate ten memory locations for a variable called "small_array"
```

User Defined Constants:

A constant can be defined to be used later in the program. Constants can make a program much easier to write (and read). Use of constants is strongly advised. Constants give you the capability of giving meaningful names to numbers.

```
define max_distance 310       ; Now max_distance is equal to 310.
```

Constants do have their limitations. A constant can't refer to another constant, and a constant cannot be defined using a mathematical expression. There are some special constants already defined for you to make life a little simpler:

XMAX - The location of the right wall (approx. 381). (The left wall is at location zero.)
YMAX - The location of the bottom wall (approx. 313). (The top wall is at location zero.)
SHOT_SPEED - The speed (In meters per second) of the cannon shells.

Constants cannot have the same name as words already defined by the compiler. These predefined words are called "keywords" and are listed in the summary.

Note: Unfortunately, robot attributes (described earlier) are not constants that you can use in your program. Of course you can define your own equivalent constants if necessary.

The TO command:

Storing and retrieving from registers and variables is done using the **TO** command:

```
3 to Aim                  ;This stores the number 3 into the register Aim
Aim to Radar              ;This copies the number in register Aim to register Radar
DAMAGE to old_damage      ;Store the amount of damage into the variable old_damage.
```

When assigning a value to two or more registers or variables, it can be done more efficiently in the following way:

```
0 to Aim to Speed to Direction    ;Store zero into the registers Aim, Speed, and Direction.
```

Math Operations:

The following mathematical operations can be performed on constants, variables, and registers:

```
+       ;Add two numbers
-       ;Subtract two numbers
*       ;Multiply
/       ;Divide   (Dividing by zero results in the biggest positive/negative number available.)
mod     ;remainder of a divide.  (Ex   6 mod 4 results in 2.     6 mod 3 results in 0.)
and     ;Compute the binary "and" of two numbers.
or      ;Compute the binary "or" of two numbers.
xor     ;Compute the binary "xor" of two numbers.
```

IMPORTANT:

There is no operator precedence for any of the operators. This means that the calculation will always proceed left to right. Also, parenthesis are not allowed. Examples:

```
a  + b * c  to   d              ;add a and b, then multiply by c and finally assign to d
my_number mod d to c    ;Compute "my_number mod d"  and assign the result to c
b + 1  to  c  to  d             ;Put the result of b+1 into c and d
```

A register, variable, or user defined constant can't be negated using a unary minus (ex: -b is illegal). To negate a value subtract it from zero. Example:

```
0 - b to b      ; Negate variable b
```

Numeric Constants:

Numeric constants can be used anywhere where a variable or register or a user defined constant is used. Unary minuses can be used on the first constant in an expression. Examples:

```
-3 * b  to   c          ;multiply -3 by b and store the result into variable c
```

Labels and the GOTO statement:

The GOTO command is used to jump from one part of the program to a another, redirecting the flow of a program. The destination of a GOTO is called a LABEL. It is used to mark a location in the program and it consists of a string of characters starting with an alphabetic character followed by up to 15 alphanumeric (and underscore) characters.

Examples of labels are:

```
Start
Begin_Again
Shoot3
```

Example of GOTO:

```
Start_Label                     ;This is a label
        Goto   Do_Something     ;Jump to "Do_Something" and start executing instructions there
        a + b to c

Do_Something                    ;This is another label
        a + 1 to a
        Goto Start_Label        ;Jump back up to Start_Label.
```

Labels must start with an alphabetic character (a-z) but may contain numbers and the underscore. Labels can be any length, but only the first 15 characters are significant.

The single line IF statement:

This simple form of the IF statement is used to make decisions by comparing two operands. The general syntax is:

IF <condition> THEN <statement>

<condition> is: <expression> <comparison operator> <operand>
<expression> is: any mathematical expression. (ex: 0-b+c or c * 2)
<comparison operator> is can be any of the following operators:

=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

<operand> is: a register, variable, constant, or user defined constant. No expressions are allowed.
<statement> is: any TO, GOTO, GOSUB, RETURN, RESET, or ENDINT statement. No IF statements allowed.

Examples:

```
if a > 10  then  goto try_again
if aim + b >= 360  then  0 to aim to b
if radar > 0  then  radar + 5 to shot to save_shot
```

The block IF, THEN, ELSE, END statement:

This is a much more powerful form of the IF statement because it provides much more flexibility. The general structure is:

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

If the <condition> is TRUE then the first block of statements will be executed, otherwise the second block of statements following the ELSE will be executed. The ELSE block is optional and without it the IF statement would take on the following form:

```
IF  <condition>  THEN
        <statement>
        <statement>
        ...
END
```

In this case, if the <condition> is FALSE then the block of statements are simply skipped. IF-THEN-ELSE-END statements can be nested within each other.

The WHILE, DO, END statement:

The WHILE statement is used for continuously looping through a block of code while a condition is true. It has the following form:

```
WHILE <condition> DO
        <statement>
        <statement>
        ...
END
```

The block of statements within the WHILE loop will execute until the <condition> is FALSE. When the WHILE statement is first executed, the <condition> is evaluated and if TRUE the block of statements is executed. After the statements are executed the <condition> is evaluated again, and so on. When the <condition> finally evaluates as FALSE, the block of statements are skipped and program flow continues past the END statement. WHILE-DO-END statements can be nested within each other.

The REPEAT, UNTIL statement:

The REPEAT statement is similar to the WHILE in that it is used for continuously looping through a block of code until a condition is met. The general form of a REPEAT loop is:

```
REPEAT
        <statement>
        <statement>
        ...
UNTIL <condition>
```

In the REPEAT loop, the block of statements are always executed at least once. This is because the test of the <condition> occurs at the bottom of the loop. At the bottom of the loop, if the <condition> evaluates to TRUE, then the loop will terminate, otherwise it will continue at the top again.

The IF, WHILE, and REPEAT statements can be nested within each other up to 15 levels deep. Thus a section of a robot program may look like this:

```
if   radar > 0   then
     repeat
          if   shot = 0   then
               radar to shot
          else
               aim - 1 to aim
          end
          aim + 1 to aim
          aim to radar
     until   radar < 0
else
     while   radar < 0   do
          aim + 1 to aim
          aim to radar
     end ;end of the WHILE loop
end;end of the IF statement at the top
```

GOSUB and RETURN statements:

The GOSUB instruction is the command used for calling a subroutine. It is similar to the GOTO instruction in that program control is transferred to the labeled line. Execution continues until a RETURN instruction is reached, returning program flow back to the GOSUB. Execution then continues with the first statement after the GOSUB.

Example:

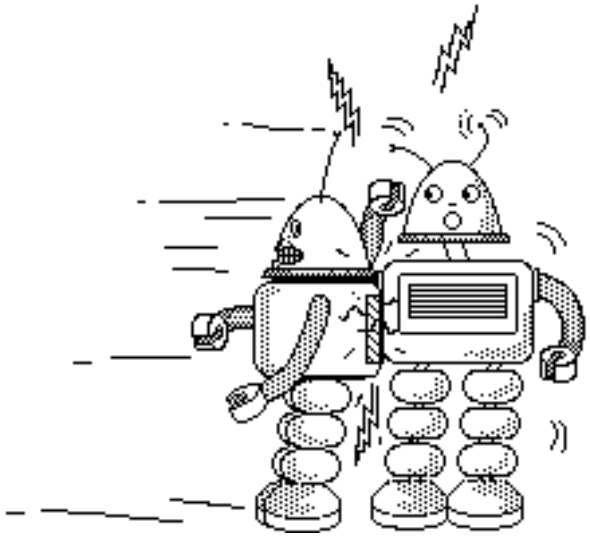
```
Again
        GOSUB  routine1     ;Call the subroutine "routine1"
        GOSUB  routine2     ;Call the subroutine "routine2"
        GOTO Again          ;Jump to Again

routine1
        a + 1 to a
        if a > 10 then 0 to a
        RETURN                      ;Return from this subroutine.

routine2
        1 - b to b
        RETURN                      ; Return from this subroutine.
```

RESET instruction:

The RESET instruction is not normally needed but you might find it useful for special cases. Upon execution the robot will set all programmable registers to zero and begin program execution at the first instruction. This is similar to what happens when the robots are first placed on the battlefield. A RESET instruction can be executed by the CPU if it detects that the program is doing something illegal (Overflowing/Underflowing the stack, jumping to unused program locations etc...)



Advanced programming features

The TRACE instruction:

The TRACE only has an effect on a robot that is being debugged (controlling the debugging features is described later), otherwise the compiler ignores the TRACE instruction. However, when debugging a robot, the TRACE instruction causes the robot battle to pause with robot debugging automatically enabled. This will allow you to examine the robots registers and to single step on the instruction following the TRACE.

TRACE is very useful for breaking in sections of a robot program that are rarely executed. You place the TRACE instruction immediately before the line that you want to break on and run a battle with debugging enabled for the robot in question. You can run several battles unattended if you wish, but as soon as the TRACE instruction is executed, the battle will pause on the line of code immediately following the TRACE instruction. Example:

```
CollisionDetected
        trace           ;stop here because I want to see if this routine gets called whenever the robot collides.
        0 to SPEED
        0 to AIM
        return
```

When debugging a robot, the TRACE instructions can be disabled and re-enabled by using the "Disable Trace Instructions" selection under the Battle menu.

Interrupts and the ENDINT instruction:

Robots are capable of producing time based interrupts. The special interrupt registers time_int_xfer and time_int_mask plus the instruction ENDINT are used to control interrupts. When interrupts are active one will once every 1/10th second. The address of the interrupt routine is stored into the register time_int_xfer. Interrupts are then turned on by storing a non-zero number into time_int_mask. For example:

```
My_Interrupt_Routine  to  time_int_xfer  ;Store address of "My_Interrupt_Routine" into the
                                         ; special register "time_int_xfer".

1 to time_int_mask              ;Turn on interrupts by putting a non-zero value into the special
                                ;  register "time_int_mask".

Loop_Again
        if a > 100 then 0 to a  ;Anytime a is greater than 100 then set it to zero
        GOTO   Loop_Again       ;Don't do anything else, just wait for an interrupt.

My_Interrupt_Routine
        a + 1  to  a
        ENDINT                  ;Return from the interrupt
```

The above example firsts gives "time_int_xfer" the name of the interrupt routine. Then interrupts are turned on. Next, the main program loop does nothing but setting variable A to zero whenever it is greater than 100. The interrupt routine always increments a by 1. This example effectively increments variable a by one every 1/10th of a second. This can be very useful for keeping track of the amount of time elapsed in the battle, or for timing events.

When an interrupt occurs, they are temporarily shut off until a ENDINT instruction is executed restoring normal execution.

Interrupts are an advanced concept and should only be used when necessary because they can make programs difficult to debug. This is especially true if variables that are modified by an interrupt routine are also modified to by your regular (non-interrupt) code. A common error encountered when using interrupts is forgetting to use the ENDINT instruction at the end of the interrupt routine.

Rectangular to Polar Conversion:

The registers X_Conv, Y_Conv, Dist_Conv, and Angle_Conv are used to convert between coordinate systems. This is useful for calculating robot positions and firing parameters. A conversion takes 50 CPU cycles. The registers are defined in two groups:

Rectangular coordinates:

X_Conv is the distance in the X direction.
Y_Conv is the distance in the Y direction.

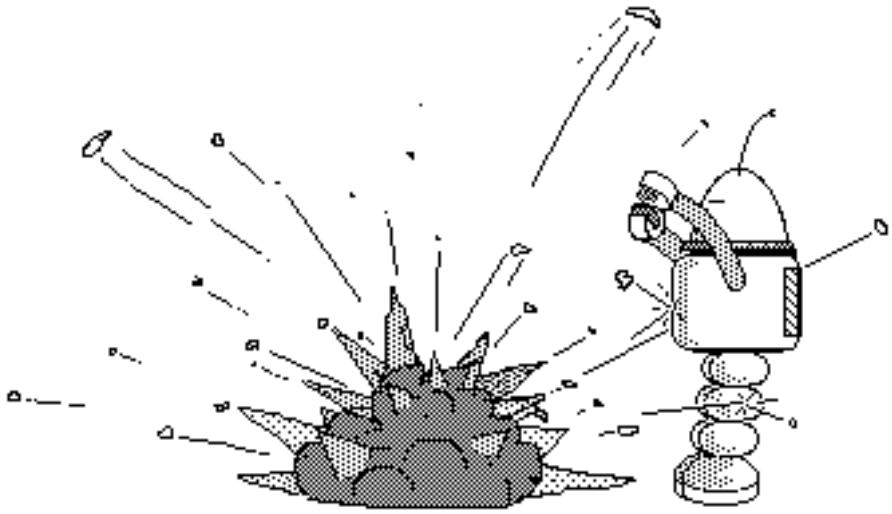
Polar coordinates:

Angle_Conv is the angle of the coordinate with 0 degrees being up just like AIM and RADAR.
Dist_Conv is the magnitude or distance.

To translate from rectangular to polar coordinates first store the X coordinate into the X_Conv register and then the Y coordinate into the Y_Conv register. Storing a number in the Y_Conv register triggers the translation with the results placed in the Dist_Conv and Angle_Conv registers.

The opposite is done to translate from polar coordinates to rectangular: The distance is stored in Dist_Conv and when the angle is stored into Angle_Conv the translation takes place with the results placed in X_Conv and Y_Conv.

Note: Both Y_Conv and Angle_Conv are the trigger registers. Storing a value into either of these registers will trigger the conversion process so make sure to store the values in the correct order.



Pointers and Arrays:

As described earlier, the allocate statement can allocate more than one memory location for a variable. This allows the programmer to create an array.

For example, the following allocate statement will create an array of 10 elements in which the first element is referenced to by "small_array":

```
allocate  small_array  10
```

While the next statement will create a simple variable that can be used to index into the array:

```
allocate  array_pointer
```

Accessing the values in "small_array" requires the use of address arithmetic. This refers to the capability of using the memory location of "small_array" as data so that an index into the array can be calculated. This is done with the "address of" operator: "&"

Example:

```
&small_array  TO  array_pointer    ;Put the address of small_array into array_pointer.
```

Array_pointer now has the address of small_array, which is actually the first element in the 10 element array. Now arithmetic can be done on array_pointer to index anywhere into small_array, and the particular element in small_array can be accessed using the de-referencing operator: ^

Example:

```
200  TO  array_pointer^     ;Put the value 200 into the memory location pointed to by array_pointer.
```

The following program segment will set all 10 elements of "small_array" to zero:

```
    0  TO  count
    &small_array  TO  array_pointer          ;Put the address of "small_array" into array_pointer
Loop_Again
    0  TO  array_pointer^                    ;Store 0 into the memory location pointed to by "array_pointer"
    1 + array_pointer  TO  array_pointer     ;Point to the next location in the array
    1 + count  TO  count
    if count < 10 then goto Loop_Again
```

Assigning program labels to variables:

Program labels can not only be jumped to, but can be assigned to variables to be treated as just another number. Also it is possible to GOTO (or GOSUB) a variable. This will cause control to transfer to the address contained in a variable. Example:

```
Start_Here
        my_subroutine TO A        ;Put address to "my_subroutine" into variable A.
        gosub A                   ;Call the subroutine whose address is in variable A.
        goto Start_Here

my_subroutine                     ;Subroutine that doesn't do anything
        return
```

Using program addresses as data allows the implementation of procedure pointers and jump tables.

CPU Stack Manipulation:

The ability to manipulate the CPU stack is a tool familiar to assembly language programmers. It is used to "pop" a return address off of the stack which effectively turns the previous GOSUB into a GOTO instruction instead. Stack manipulation is also used to pass parameters to subroutines. It isn't as simple to use as parameter passing in languages like PASCAL or C but can be just as effective. Stack manipulation is done by accessing the STACK register.

Popping a return address:

When the GOSUB instruction executes, the return address is pushed onto the stack. To remove it, just pop the address off of the stack to erase the effects of the GOSUB. Example:

```
Start
        Gosub  Change_To_Goto
        ...

Change_To_Goto
        Stack to dummy_register      ; Remove the return address off of the stack
        goto Start                   ; Now program can jump to another point rather than returning
```

Passing parameters to a subroutine requires pushing them onto the stack. The subroutine will then pop them off of the stack and use the values. Care must be taken to ensure that the stack is properly restored upon return from the subroutine. (For every push or GOSUB there must be a pop or RETURN).
Example:

```
Start
        2 to Stack                   ; Push 2 onto the stack
        Gosub Add_to_B               ; Gosub will push return address and then jump to Add_to_B.
        ...
        5 to Stack                   ; Push 5 onto the stack
        Gosub Add_to_B
        ...

Add_to_B
        Stack to Save_Return_Address     ; Pop return address off of the stack and save it.
        Stack + B to B                   ; Pop the passed parameter and add it to B.
        Goto Save_Return_Address         ; Do the equivalent of a RETURN.
```

The Debugging Window

(Refer to Appendix A: **Menu Commands** for help in starting a battle in debugging mode and using the debugging commands.)

Figure 3 shows a picture of a battle being run in debugging mode. Two robots are battling it out, but only the status display for the robot being debugged is displayed. The remainder of the status area is used to display the contents of the robots registers and to display the code that is currently executing.

The speed register displays contains the selected speed that the robot wants to run at but since robots have to accelerate to any given speed the actual velocity may not have reached the value in the register yet.

The direction register shows the direction in which the robot is travelling.

Unlike all the other registers, the radar register always contains the result of the radar signal, not the angle originally placed in it. In the example below the radar has the value of -1 which means that no robot has been spotted.

The shot register will contain zero when the gun is ready to fire again. Otherwise it usually contains the remaining reloading time (in the number of robot cpu cycles left) or the number 0 when the gun is just waiting for the last shell to explode.

The source code display shows 10 lines of code at time including the line currently being executed. A small square rectangle to the left of the line points to the current line of code. This rectangle will move from line to line when executing and the display of the source will be updated when necessary.

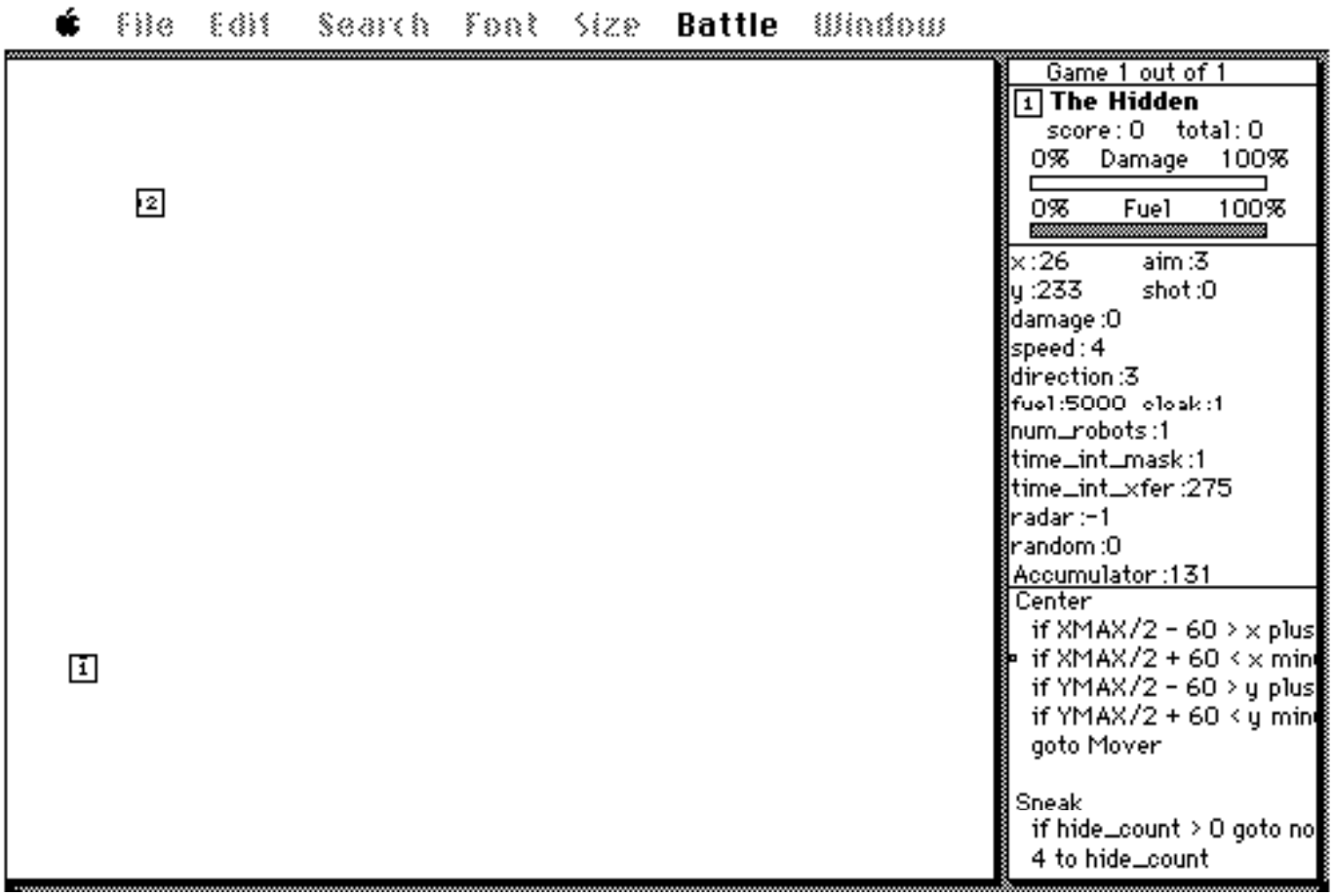


Figure 3

Appendix A: **Menu Commands**

Here is a summary of the menu commands available in Robot Warriors. It is assumed that you have a basic understanding of Macintosh operation including windows, menus, file operations, and simple text editing.

File Menu:

New: This allows you to create a new robot. An empty text editing window is opened up allowing you to type in your robot program. Up to five programs can be open at the same time.

Open: This allows you to open a previously saved robot for editing or to use in a battle. Up to five programs can be opened at one time.

Close: This closes the current window. If the file has been changed without being saved, you will be asked if you want to save it before it is closed.

Save: This will save the current file.

Save As: This will save the current file giving you the option to rename it or change the folder in which it is saved.

Revert To Saved: This command allows you to revert back to the most recently saved version of a file.

Page Setup: Using this command allows you to set the page formatting options for printing. These formatting options aren't saved with the file and will effect all files printed. The next time Robot Warriors is run the page setup options will be defaulted once again.

Print: This allows you to print the current file. Multiple robots can be printed at the same time from the Finder using the Finder's Print command.

Quit: This will exit the Robot Warriors program. You will be asked whether you want to save any unsaved files.

Edit Menu:

Undo: This command is not currently implemented.

Cut, **Copy**, **Paste**, and **Clear** are the standard text edit commands you are familiar with.

Select All: This selects all of the text in the current window.

The next four commands allow several program lines to be indented or commented at the same time. They are used by first selecting the text you want affected and then choose the operation you want performed on the current selection. If you select a partial line, and choose one of these commands, the whole line will become selected before the command operates on your selection.

Shift Left: This will shift the selected lines of text left one column eliminating one space from the beginning of each line. If no space is available at the beginning of the line then it is unaffected.

Shift Right: This will shift the selected lines of text right one column adding one space to the beginning of each line.

Comment Lines: This will add a semicolon to the beginning of each selected line so that the compiler will treat it as a comment. Lines that already begin with a semicolon will still have another added to it.

Uncomment Lines: This will delete a semicolon at the beginning of each selected line.

Auto Indent: Turning on auto indent causes a check mark to be placed next to this menu item. When auto indent is on, a new line will automatically be indented the same amount as the previous line saving you some typing.

Search Menu:

Find, **Find Next**, and **Change** should be familiar to you as the standard search and replace functions.

Goto Cursor: This automatically scrolls the window to the location of the insertion bar (editing cursor).

Home Cursor: This is a quick way of moving the cursor up to the beginning of the file.

Font and **Size** Menu:

This will change the font and size used to display the text in the current window. Files will be printed using the same font that they are displayed with. Selecting a font while holding down the option key will change the font used to display all of the files. This font attribute isn't stored in the file and will be defaulted back to 9 point Monaco whenever a file is re-opened.

Battle Menu:

Start Battle: This command is used to begin a robot battle. It will display a list of all the robots you have open allowing you to select the ones to be in the battle. Up to five robots can battle at the same time. You also have the option of running up to 1000 battles in succession. Running several battles is recommended in order to get a good picture of how the robots really compare. You also have the option of turning debugging on for a single robot. This is useful for finding bugs in your program, but it will eliminate the status display of all the other robots. When a battle is began, all robots that haven't been compiled since they were last changed will automatically be re-compiled.

A battle ends when only one robot is left on the battlefield. A battle will also be terminated if the robots are effectively idle for 30 (robot) seconds. After a series of battles is complete the final scores are displayed with some statistics on the performance of each robot. These statistics are:

Number Of Battles Won: Number of battles in which the robot was the sole survivor (or last to die).

Shot Damage: Amount of damage worth of hits this robot scored on other robots.

Average Damage Sustained: Average amount of damage sustained over the course of the battles. The damage is measured as a percent of the damage required to kill the robot. This provides an extra measure of performance when comparing robots to each other. It tells you how close each robot was to dying on the average. The average is based on the number of battles selected in the battle dialog and if a battle or series of battles is canceled, this will lower the resulting average of each robot. Therefor, this statistic is only accurate if all of the battles are allowed to complete.

Shot Damage To Self: Amount of damage done by shooting too close to oneself.

Collision Damage: Amount of damage sustained during collisions.

Times Out Of Fuel: Number of times the robot ran out of fuel.

CPU Error Resets: Number of times the robot's CPU was reset due to a serious program error. Errors such as jumping to undefined program addresses or overflowing/underflowing the stack will cause a CPU Error Reset.

In Figure 4 an example of the Start Battle dialog box is given. This is how it would look if four robot files were open; New Seeker, Crawler, Tracker3, and The Hidden. In this example, all of the robots are selected for battle since the square check boxes are all checked. Also debugging is turned off because the radio button for "Debugging Off" is selected. The number of battles is chosen to be three. This means that three battles will be run in succession.

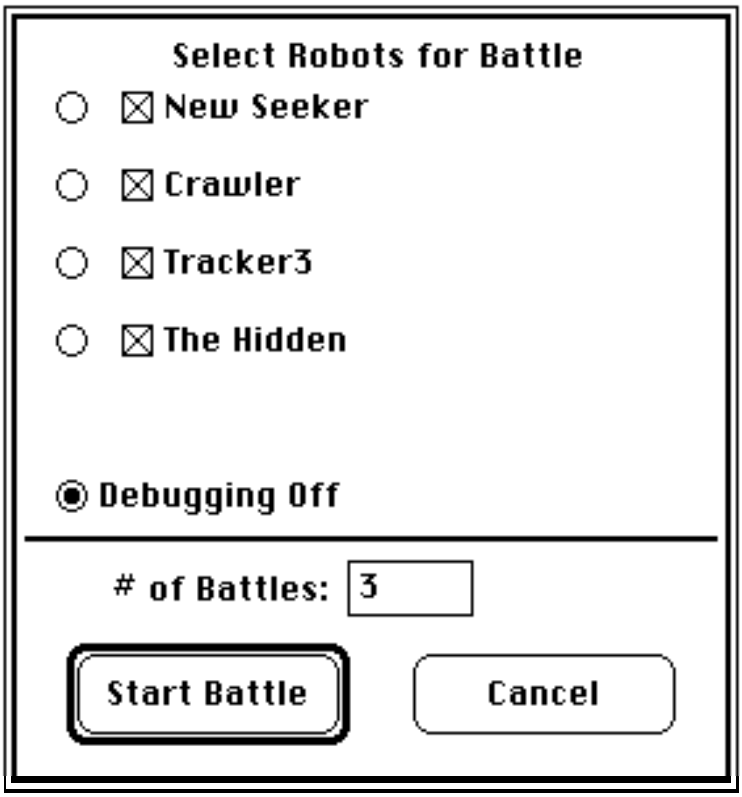


Figure 4

In Figure 5 another example of the Start Battle dialog box is given. Again, four robot files are open, but in this example, only three of the robots are selected for battle. Also notice that the debugging feature has been turned on for Tracker3 since the radio button in front of Tracker3 has been selected. Only one battle is selected to be run.

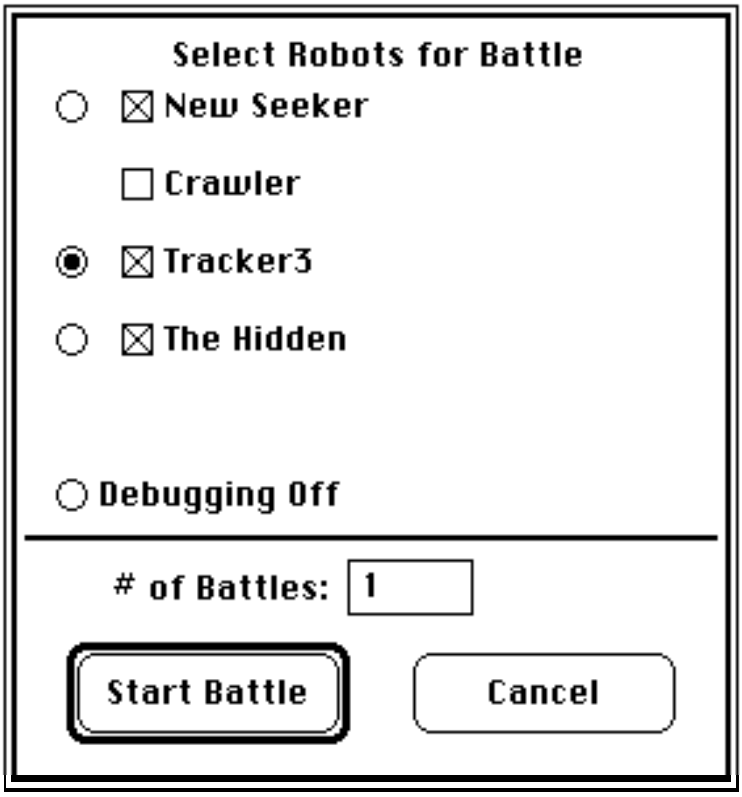


Figure 5

Pause/Continue: This allows you to pause a battle in progress. When you begin a battle in debugging mode the game will already be in pause mode.

Cancel Current Battle: This allows the battle in progress to be cancelled and the next battle will begin immediately.

Cancel All Battles: This cancels all pending battles returning you to the editing windows.

Compile Robot: This allows you to compile the robot in the current window. If a syntax error is found in the program, the compiler will inform you of the error and it will move your editing cursor to the location in your program where it found the error. It isn't necessary to compile a robot before entering it into a battle. Robot Warriors always keeps track of robots that need to be compiled, automatically doing so when a battle is started. Robots that have been changed since the last time they were compiled will be re-compiled.

Disable/Enable Debugging: This command is used when you are running a battle in debugging mode. Disabling debugging will allow the battle to proceed quickly, but without updating the debugging displays for registers and source code. Enabling debugging will allow the register and source code displays to be updated so you can view the operation of your robot.

Disable/Enable Trace Instructions: This selection is used to disable the trace instructions in a debugging robot. When disabled, the battle will not pause when a trace instruction is executed in a robot in debugging mode. With trace instructions enabled, the battle will pause and go into debugging mode whenever a trace instruction is encountered.

Single Step: This command has two modes of operation:

When battle execution is progressing normally (debugging mode is off), single stepping will allow the battle to proceed for 1/10th of a second then it will be paused.

In debugging mode, if this command is executed while the battle is running one more instruction is executed and then the battle will be put into pause mode. While in pause mode, each single step will execute exactly one CPU instruction. The small source code window will update to show you exactly which line of code is being executed. Single stepping allows you to watch your program slowly execute, stopping at your convenience to observe your program. Note: usually two or more robot instructions make up one line of source code so usually you will have to execute several single steps before your source code display is updated.

Turn Sound Off/On: This command allows sound to be toggled off and on.

Fast, **Medium**, **Slow, Slower Still, Sluggish**: This allows you to slow down a battle just in case you have a faster Mac. The Fast option will run the battle at the fastest possible speed, while the Sluggish option is the slowest speed. Battle speed is very dependent on the speed of the Macintosh and the color depth of the screen. For instance: back and white screen settings allow the battle to run much faster than monitors in 256 color mode.

Window Menu:

Stack Windows: This will re-size and reposition the windows so that they are stacked on top of each other like a deck of cards.

Tile Windows: This will re-size and reposition the windows so that they are all visible at the same time.

<Bring Window To The Front>: The next five menu items allow you to quickly bring any one window to the front. They have the command key equivalent of ⌘-1 through ⌘-5.

Clipboard: This will open the clipboard into it's own window and bring it to the front.

Hide Window: This will make a window invisible so that it doesn't clutter your desktop, but it is still open for battle. This is nice for when you are working on a single robot program, but you need other robots available to do battle with.

Appendix B: **Judging Robots**

It's important to note that determining which robot is superior can be difficult. The simplest way to judge robots is in battles of only two robots (one on one). In this case it usually only takes a few battles (around 10) to determine which robot is better and if the outcome is fairly close, more battles can easily be run.

The situation becomes more complex when running battles with several robots. When five robots are entered in a battle complex situations can arise that can cause the destruction of a very good robot (if it gets cornered by others), while allowing a poorly designed robot to survive for a long time. A crowded battlefield can be quite chaotic and this is why it's necessary to run many battles when trying to judge several robots at a time. In a contest situation, it's suggested that at least100 battles be run if 5 robots are entered at a time. Remember, that with 5 robots, even if one robot always wins, the difference in score between this robot and the runner up may be only a 25% difference in the total scores.

Obviously, the score that a robot receives when running several battles is highly dependent on the strength of the competition. What isn't so obvious is that some robots are just better at killing certain types of robots than others. It's possible to have three robots (A, B, and C) and in "one on one" competition, robot A usually defeats robot B, and robot B usually defeats robot C, yet robot C usually defeats robot A! The situation can get more complex when running battles of 5 robots each. The strengths of each robot can actually enhance each other against certain types of opponents.

When judging robots in a programming contest, it is suggested to have both one on one and group competition. The one on one battles can be run until a particular robot gets 5 more points than it's opponent before moving to the next heat. Battles with multiple robots should also be handled in an elimination manner with the top two scorers making it to the next heat. Scoring can also be based on total damage done to other robots, or on the number of games won, or even some combination of these measurements.

Appendix C: **Reserved Keywords**

The Robot Warriors language uses the following words in the language. These words cannot be used as program labels or variables:

```
AIM            ALLOCATE       AND            ARMOR        CLOAK
CLOAKING       CPU_SPEED      DAMAGE         DEFINE       DIRECTION ENDINT
ENGINE_SIZE    FIRE_RATE      FUEL           FUEL_CAPACITY    GOSUB
GOTO           IF             MOD            NUM_ROBOTS   OR   RADAR
RADAR_RANGE    RANDOM         RETURN         SHOT         SHOT_SPEED
SPEED          STACK          THEN           TIME_INT_MASK
TIME_INT_XFER  TO             TRACE
X              XMAX           XOR            Y            YMAX
```

Appendix D: **The Internal Operation of Robot Warriors**

When a battle is run in Robot Warriors, a simulation of the robots CPU, mechanics, and physical environment done. This simulation is the subject of this section.

The Internal Language Of Robot Warriors

When a Robot Warriors program (source code) is compiled, it is converted into a sequence of commands that the robot understands (object code). Typically, each line of source code will translate into several object code instructions.

In order to understand how the internal language works it is important to understand the architecture of the robot. In addition to the registers that you deal with in your programs a robot has the following additional registers:

Program Counter:	Points to the next instruction to be executed.
Stack Pointer:	Points to the next available location on the stack.
Accumulator:	Almost every instruction operates on the accumulator.

Also, the memory of a robot is divided up into three partitions:

Variable Memory:	Holds variables that are allocated in your program (200 memory locations).
Program Memory:	Holds the robot's compiled object code. (room for 2000 instructions).
Stack:	Holds values pushed onto the stack such as subroutine return addresses. The stack can hold 30 values.

To get an idea of what the robots object code looks like, lets look at the translation of some simple source code statements:

Our first example simply adds one to the variable "count":

```
allocate count        ;allocate the variable "count"
        count + 1   TO  count     ;increment the variable "count"
```

For the code above, the following instructions would be generated:

```
LOAD   (memory,COUNT)      ;load the accumulator with the contents of memory location "COUNT"
ADD    (constant, 1)       ;add 1 to the accumulator
STORE (memory,COUNT)       ;store the value in the accumulator into memory location "COUNT"
```

As you can see, each instruction has three components:

<instruction> (<data type>, <value>)

In the example above, COUNT represents the memory location number assigned when COUNT was allocated. The <value> can represent register number, constant, memory location number, or a program address.

Now lets look at a slightly more complex example:

```
allocate   old_dist
repeat
   aim + 3 to aim
   if aim > 90 then
       0 to aim
   else
       aim + 1 to aim
   end
   aim to radar
until radar > 0
if radar > 20 then radar to shot to old_dist       ;shot at robot if it is more than 20 meters away.
```

This will translate into the following instructions (with the instruction number explicitly labeled with the memory location of the instruction):

```
      repeat
        aim + 3 to aim
1  LOAD       (register, AIM)       ;Load the accumulator with the contents of the register "AIM"
2  ADD        (constant, 3)
3 STORE       (register, AIM)
      if aim > 90 then
4  LOAD       (register, AIM)
5  LTE        (constant, 90)        ;Compare the accumulator with 90 (accumulator <= 90 ?)  If the
                                     ;result of the compare is false, then the next instruction is skipped,
                                     ;otherwise the next instruction is executed.
6  GOTO       (constant, 10)        ;Jump to instruction 10. (ie. Put 10 into the program counter)
       90 to aim
7  LOAD       (constant, 0)
8  STORE      (register, aim)
9  GOTO       (constant, 13)        ;Goto instruction 13, skipping around the ELSE section.
    else
       aim + 1 to aim
10  LOAD       (register, aim)
11  ADD        (constant, 1)
12 STORE       (register, aim)
      aim to radar
13  LOAD       (register, aim)
14  STORE      (register, radar)     ;Place the value in the accumulator into the radar register.  This will
                                     ;trigger the radar to fire with the radar result being placed back into the
                                     ;radar register.  Using the radar uses 40 robot instruction cycles.
                                     ;Firing the gun also takes multiple instruction cycles.  All of
                                     ;the other instructions in this example only take 1 cycle each.

      until radar > 0
15  LOAD       (register, radar)     ;Get the result in the radar and place it into the accumulator.
16  GT         (constant, 0)         ;Compare the result with 0 (accumulator > 0?)
17  GOTO       (constant, 1)         ;If the accumulator IS greater than 0, then go to instruction number 1.

      if radar > 20 then radar to shot to old_dist   ;shot at robot if it is more than 20 meters away.
18  LOAD       (register,radar)
19  LTE        (constant, 20)        ;If radar <= 20 then skip next instruction
20  GOTO       (constant, 24)        ;Goto instruction 24 skipping the assignment of radar to shot etc...
21  LOAD       (register, radar)
22  STORE      (register, shot)      ;Shoot the gun the distance stored into register "shot".
                                     ;Firing the gun uses 10 robot instruction cycles.

23  STORE      (memory, 0)           ;variable "old_dist" is stored in memory location 0.
```

As you can see, the 11 lines of source code translated into 23 object instructions. Also you may have noticed that the object instructions aren't optimal: Notice that instructions 4, 13, and 21 are unnecessary since the accumulator already contained the correct value. However instruction 15 is necessary since the value of the radar register contains the result of the radar being fired, not the angle placed into the radar register. A more efficient compiler would notice this and optimize the code by eliminating instructions 4, 13, and 21. The current Robot Warriors compiler currently doesn't optimize to this level of efficiency.

Before going any further, a list of the instructions and data types is in order:

Instructions:

```
    LOAD            ;Load a value into the Accumulator.
    STORE           ;Store a value from the Accumulator into a register or memory location.

;These mathematical operations operate on the Accumulator
    ADD
    SUBTRACT
    MULTIPLY
    DIVIDE
    MOD
    AND             ;Logical "AND"
    OR
    XOR

;Logical operations.  If result is true the next instruction is executed, otherwise the next
;instruction is skipped.
    EQUAL           ;Is value equal to that contained in Accumulator?
    GT              ;Greater than?
    LT              ;Less than?
    GTE             ;Greater than or equal?
    LTE             ;Less than or equal?
    NEQUAL          ;Not equal?

;These instructions control the flow of your program
    GOSUB
    GOTO
    RETURN
    INT_RETURN      ;Interrupt return
    RESET
```

Data Types:

```
register            ;Data represents a robot register
constant            ;Data represents a numeric constant
memory              ;Data represents a memory location
D_memory    ;Data represents a memory location that should be used as a pointer to another
                    ;memory location.  (One level of indirection)
```

Currently there is no way for you to view the actual robot instructions generated by the compiler.

The Battlefield Simulation

Initially the robots are placed on the battlefield so that they are randomly placed within cells that guarantee that no two robots will be placed right next to each other. All of the robots registers are initialized to the following values:

X,Y:	Initial robot position
AIM:	0
SHOT:	0
DAMAGE:	0
SPEED:	0
DIRECTION:	0
FUEL:	1850,2500, or 3200 depending on allocation of FUEL_CAPACITY
CLOAK:	0
NUM_ROBOTS:	Number of enemy robots on the battlefield
TIME_INT_MASK:	0
TIME_INT_XFER:	0
X_CONV:	0
Y_CONV:	0
DIST_CONV:	0
ANGLE_CONV:	0
RADAR:	-1
RANDOM:	0
STACK:	Points to the first location in the stack: 0
ACCUMULATOR:	0

After the robots are initialized, the simulation of the battle begins. The battle simulator divides the battle up into small time slices (1/10th of a second per time slice) in which it does two things:

1) Execute the robot programs.

If a robot executes 1000 instructions per second then it will execute 100 instruction cycles during one time slice. All instructions take exactly one instruction cycle to execute except using the aiming radar which takes 40 cycles, firing the gun which takes 10 cycles, and doings a rectangular to polar conversion which takes 50 cycles. At the beginning of the robots time slice, the time_interrupt register is checked to see if the interrupt routine needs to be called.

2) Update the physical environment.

The following events occur when the physical environment is updated:

Robot and shot positions are updated
Shots explode and shot damage is calculated
Robots collisions are detected
Robot speeds are updated (robots are accelerated)
Robot damage is updated (and robot destruction is determined)
Fuel register is updated.
Robots shot register is updated.
Battlefield and status windows are updated. (status window every other or fourth time slice)

Since the robot positions are only updated every 1/10th of a second it is possible that a robot with a fast CPU can track another robot with the radar several times before the robot moves.