Advanced T-Robots (AT-Robots)
version 2
(ATR2)

http://www.necrobones.com/atrobots/

For current CONTACT INFORMATION see: NBONES.DOC

CONTENTS:

INTRODUCTION:

Welcome to Advanced T-Robots, a game in which players write programs to

control robots which will battle to the death in a simulated arena, using a programming language similar in design and concept to PC assembly language.

This particular game is ATR2, or rather, the second AT-Robots. Unfortunately, though much remains the same, some fundamental changes made downward compatablility impossible. If you used AT-Robots 1.x, you will either need to make new robots from scratch or convert your old ones by hand. Further down, I will explain what has changed and what is the same.

AT-Robots stands for "Advanced T-Robots", though "AT" can also refer to AT-class PC's, since it will run on 286's and higher. T-Robots was the predecessor program to this, written originally in November of 1991. It was inspired by P-Robots (written by David Malmberg), which was in turn inspired by C-robots (written by Tom Poindexter). T-Robots and AT-Robots are designed with the idea of programming robots, without going to the lengths of learning to program in high-level languages, but to still allow fairly simple program design. In the case of AT-Robots, the language is based on PC assembly. Basically, T-Robots allowed you to use T-Robot-Basic to program the robots, and now in AT-Robots you use T-Robot assembly. Instead of having complex commands for complex tasks, you have simple commands to do simple tasks. You will have to design subroutines to access the specific devices/ports/memory areas of the robot specifically in your program. The language has a small amount of available commands, thus making it easy to learn, while offering experienced programmers an interesting challenge. Though it is not a real programming language of sorts, it can still teach the basic concepts of programming and logical thinking.

The first incarnation of AT-Robots was in February of 1992, only 3 months after the initial T-Robots program. After that, AT-Robots remained dormant from a development standpoint, but continued to be used occasionally by people around the world for fun and for educational purposes. The original versions were primarily distributed through my own personal BBS, and through a very large BBS that I often used for launching my shareware programs (The Programmer's Corner), and then eventually through AOL. Then in March of 1997, AT-Robots version 2 came on the scene, which coincided with my obtaining my own domain name on the World Wide Web. I made announcements on several usenet newsgroups, and ran a contest. Since then the game has slowly trickled out into the world.

CREDITS:

Concept, Design, Programming	- Bones (Ed T. Toton III)
Debugger	- FiFi LaRoo

TRADEMARKS & COPYRIGHTS:

C-Robots (c) 1985 Tom Poindexter
Turbo Pascal is a registered trademark of Borland International.

THINGS YOU SHOULD HAVE:

First of all, you need DOS 3.3 or higher (or Win95's DOS), some free hard drive space, a VGA or SVGA, and at least a 286 PC processor. That covers the hardware.

You will also need a text editor. DOS (starting with 5.0) comes with a built-in text editor called EDIT. Whatever you use doesn't matter specifically, as long as you save your robot programs in "DOS Text" or "ASCII format" (must be the DOS standard of CR,LF ending each line).

Files:

ATR2.EXE	- The main program.
ATR2.DOC	- This document.
NBONES.DOC	- Contact information.
READ.EXE	- A text viewer.
ATRLOCK.EXE	- Encrypts robots for distribution.
demo.bat	- Demo batch file
demo.ats	- Demo config file.
file_id.diz	- BBS auto-description file
*.at2	- These are robot programs, all ending in .AT2
.pas,.tpu,*.obj	-Source code files for AT-Robots.

RUNNING AT-ROBOTS:

Before we get into the language, let me first explain how to run the game. If you try running AT-Robots just by typing ATR2, this is what you will see:

Error #5: Robot names and settings must be specified.

Well, here's how you do just that. ATR2 must be run using command-line parameters. Here's an example:

```
ATR2 /s #myconfig
```

Any parameter that is preceded by a slash (/) or a dash (-) is a setting. Any parameter preceded by a pound sign (#) is a configuration file (in this case "MYCONFIG.ATS"). If no extension is specified in the filename, ".ATS" is asswumed.

Parameters not starting with these characters are assumed to be robot filenames.

Example: ATR2 /s /m2 myrobot yourobot

In this case two settings are specified, and two robots are loaded, specifically "myrobot.at2" and "yourobot.at2" (if a filename extension is not specified, it is assumed to be ".AT2").

If you want to load a locked robot, you can either specify the ".ATL" extension (i.e. "MYROBOT.ATL") or precede the namee with a question mark (?) (i.e. "?MYROBOT").

Settings:

```
 /S  = Do not show source code during compile (must appear before robot names)
 /Dn = Specify game-delay (timing control), [default=30], example: /D20
 /Tn = Specify time-slice for robots [default=5] (cpu cycles per game cycle)
 /Ln = Specify battle time-limit in 1000's, 0 means no limit [default=0]
 /Q  = Quiet Mode, no sound effects.
 /Mn = Specify number of matches to play. ex- /M10 is 10 matches.
 /G  = No graphics, just generate a quick result.
 /Rn = Generate a report file after battle (for use with tournament program) *
 /C  = Compile only, do not run battle (for debugging and verifying)
 /A  = Show scan arcs during battle
 /W  = Windoze=off,  Do not pause with windowed information. ***
 /E  = Turn on error-logging for robots.
 /#n = Specify maximum robot program length. ex- /#32 is 32 compiled lines **
 /!n = Insane missles (n can be 0 to 15). Never use for tournaments!
 /@  = Use old shield style (no damage or heat taken from hits).
 /Xn = Start in debug mode and set the step count 1-9 (/X is equivalent to /X1)
```

* See below for report-file format

** NOTE- parameters affecting compilation, such as /S or /# must come BEFORE the robots they are affecting. This means you can have different program length limitations for different robots.

ex- ATR2 /#10 small.at2 /#50 medium.at2 /#1000 large.at2

*** /W turns off the bout results in both graphics and text modes. Only the absolute final results will be desplayed if this parameter is used, and not the intermediate results. This can be toggled at runtime with the "W" key, whether you're in text or graphics mode at the time.

Creating a config file is very easy, simply use the same parameters that you would on the command line, except in the file they are placed one per line. For an example, see DEMO.ATS. In a config file, any blank lines or lines starting with a semi-colon (;) are ignored.

DURING THE SIMULATION:

While the simulation is running, there are several keyboard commands you can use:

```
A = Turn on/off the displaying of scan arcs
S = Turn sound effects on/off
G = Turn graphics on/off
T = Turn timing on/off
```

```
  + = Increase timing delay (game runs slower)
  - = Decrease timing delay (game runs faster)
  W = Toggle 'windoze' (pausing at displayed windows)
  C = recalibrate timing (timer might have mis-calibrated if
      a cache program such as smartdrive was still writing data).
  X = Turn the debugger on/off (Robot #1 must be unlocked).
      While in debug mode, there are several additional keyboard commands
      you can use:
        SPACEBAR = Step with current step count
        1 .. 9   = Set a new step count and step
        -,+,[,]  = Scroll the memory view
  ESC or Q  = Quit.
  Backspace = End current match.
```

If the game is running without graphics, timing and sound will also be off since they only make sense when you're watching the action. You can use these keyboard commands whether the game started in graphics mode or not.

REPORT FORMAT:

The report-file is for use with the tournament program or other shells that want to read information back from ATR2 about the results of the battle. There are multiple formats available. They're all basically the same, except for the amount of information included. The older ones are kept for downward compatability as new ones are added.

In each case, the first line is the number of robots in the report. After that, each robot gets its own line. On the robot lines, several numbers are given, seperated by spaces, and the name is listed last. All report modes are enabled with the /R parameter, possibly with a number to denote the specific one.

An example /R report:

```
2
0 1 SNIPER
1 1 RANDMAN3
```

The line formats for each report mode:

```
/R :  Wins Trials Name
/R2:  Wins Trials Kills Deaths Name
/R3:  Wins Trials Kills Deaths EndingArmor EndingHeat ShotsFired Name
/R4:  Wins Trials Kills Deaths EndingArmor EndingHeat ShotsFired Hits
DamageTotal CyclesLived ErrorCount Name
```

LOCKING ROBOTS:

When it comes time for you to use your robot in a competition, you may want to allow others access to your robot but without making it easy for them to discover its secrets. For this reason, there is a program included

with AT-Robots called ATRLOCK. It removes all remarks from the robot and then encrypts it. It's not a particularly complex encryption scheme, just enough to make it unreadable.

For all you hackers out there who are thinking of cracking it or making an un-lock program, I ask you nicely to please not do so.

To use ATRLOCK, simply specify the robot to be locked:

example:
```
ATRLOCK sniper
```

This will encrypt "SNIPER.AT2" into "SNIPER.ATL". To use the LOCKed SNIPER in a game, you could simply load it as "?SNIPER".

ACRONYMS & ABREVIATIONS & DEFINITIONS:

Here in ATR (see, there's one already) we use several acronyms that you may or may not already be familiar with. At anyrate, here they are (but don't worry about memorizing them):

Acronym:	Meaning:
ATR	AT-Robots, or a robot for AT-Robots.
ATR1	AT-Robots, versions 1.0 and 1.1
ATR2	AT-Robots, versions 2.0+
ATRA	Advanced T-Robots Assembly.
BIOS	Basic Input/Output System
ERR	Error
GSB	Gosub (a call to a subroutine)
I/O or IO	Input/Output
INT	Interrupt
JMP	Jump (like a GOTO in basic)
OP	Op-Code (numeric codes that represent instructions)
RAM	Random-Access Memory.
REG	Register
ROM	Read-Only Memory.
ROS	Robot Operating System.
SPD	Speed
VAR	Variable
LOCK	Encryption for robot distribution

WHAT'S NEW TO ATR2:

If you've never used AT-Robots 1.x, skip this section.

One of the most important changes is the memory addressing system, and how operands are stored and decoded internally. In ATR1, what an operand did was completely defined by its value. Anything over 10000 was a memory address. Now you have the full range of integer values to use as numbers, or as values in variables, since there is now seperate microcode stored to define the function of an operand.

A few of the I/O ports have changed, but most are the same. All of the Interrupts have been redefined. Most of the instructions in the language are the same, except conditional jumps can not be used as conditional gosubs, and basic math functions use only 2 operands instead of 3.

The list of registers has also changed. However, if you were simply using some of the old ones as generic variables, why not create them as variables if you convert your robots over? The variables that are used for passing values back and forth to interrupts have changed of course.

The accuracy setting parameter that you send to the weapon port is no longer translated into degrees automatically. You must do this yourself before passing the parameter.

One more MAJOR change is that the game uses a 256 degree circle instead of 360.

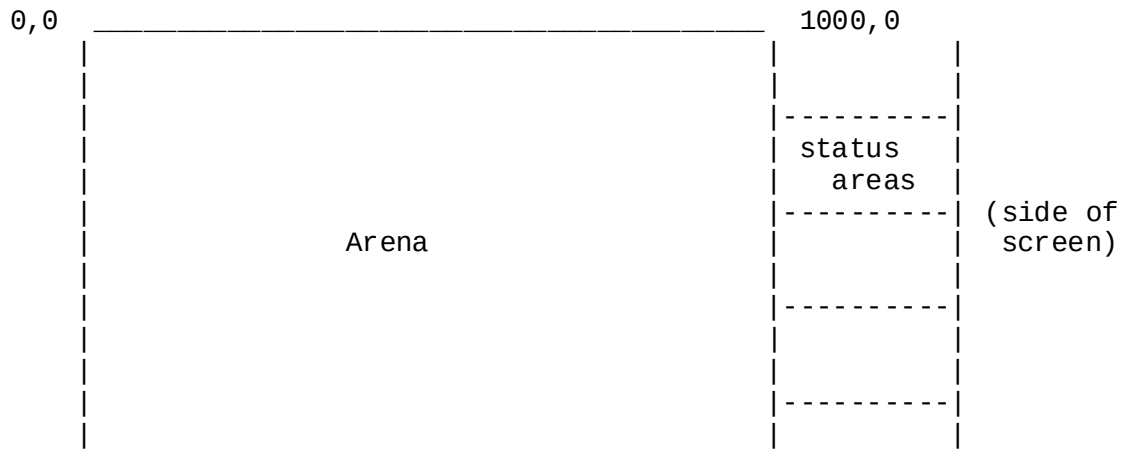
Why so many changes? Especially that circle thing? Here's why- I designed the original AT-Robots before my programming expertise really allowed me to do as good of a job as this project deserved. I was only starting to learn assembly at the time, and here I was making my own version! I was not yet used to dealing with powers of two, individual bits, bit-manipulation instructions, and hexidecimal. I ended up making the language in such a way as to steer you away from these valuable programming techniques, and was therefore teaching the wrong lessons! Now I'm going to beat hexidecimal and powers of two into everyone's skulls... :-)

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048,
4096, 8192, 16384, 32768, 65536, to name a few... :-)

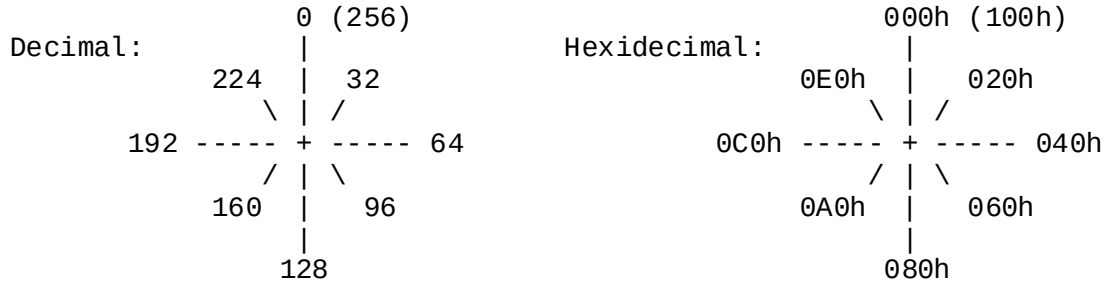
BASICS & GENERAL INFO:

The arena is 1 kilometer wide and 1 kilometer long. The coordinate system is set up such that (0,0) is at the upper-left corner and (1000,1000) is at the lower-right.

The screen is set up as follows:



Navigation is done in a 256 degree circle, with 0 being north. Here are the directions for each course:



(hexidecimal will be briefly explained later)

The Robots are all equipped with a scanner, armor, and a weapon.

Firing the weapon generates heat, which can impair the robots performance. When the projectile hits an enemy robot, the amount of damage that the target takes will depend entirely on how accurate the shot was. When a robot is destroyed, it explodes, possibly causing damage to nearby robots.

Firing the weapon at point-blank can be dangerous, since the weapon blast can hit everything within a given range. How much damage the weapon does depends on how "on-target" the shot was.

Each robot is also equipped with a feature called "overburn" which allows the robot to increase its performance by cranking more energy out of its power-supply (by disengaging certain safety measures), but at the cost of easily over-heating. When on overburn, missiles will work more effectively, and the robot will drive faster. 100% throttle is still maximum speed, as the speed of the robot is automatically scaled based upon it's current performance.

Robots don't stop, start, or turn on a dime. When you tell the robot to move at a certain speed, it will accelerate until it reaches that speed or it collides with something. Once it has reached that speed it will remain there until you change it or it collides with something. When the robot does in fact collide with something, it's speed and throttle are reduced to zero. If the velocity was greater than half of it's maximum non-overburn and non-overheated velocity, it takes a point of damage.

When you instruct the robot to turn, it will rotate until it faces the new direction, or you tell it to do otherwise. The robot will always rotate in the direction that is shortest to face the target heading. if you tell it to turn 270 degrees to the right, it will turn 90 to the left (it's

shorter, and is still the same heading). Also note that since it takes time to start/stop/turn, that the program may have executed many commands before the machine actually gets to it's new speed or heading.

Each robot has two status bars on the right-hand side of the screen. The upper one (marked with an "A"), is the robot's armor. The one below it (marked with an "H") is the robot's heat scale. Also in the robot's status area is a space dedicated to displaying error codes. These codes will be displayed in both decimal and hexidecimal format. They can be extremely useful as a means of output to see what's going on "in your robot's head".

Damage:
~~~~~~~

Collision at high speed:	1
Heat is 300 - 349:	1 every 64 game cycles
Heat is 350 - 399:	1 every 32 game cycles
Heat is 400 - 449:	1 every 16 game cycles
Heat is 450 - 474:	1 every 8 game cycles
Heat is 475 - 499:	1 every 4 game cycles
Heat is 500 or higher:	Robot explodes.
Missile blast:	14-range (i.e. 14 for dead center, and 1 at 13 meters)
Robot detonation:	25-range (i.e. 25 for dead center, and 1 at 24 meters)
Mine blast:	35-range (i.e. 35 for dead center, and 1 at 34 meters)
Overburn Missile blast:	(14-range)*1.25
Overburn Robot detonation:	(25-range)*1.30

It is assumed that the robot bodies have a radius of 4 meters, and thus the closest you can get to an enemy is 8 meters. Therefore the most damage you will actually take from another robot exploding is 17, assuming that it is not on overburn. The missiles however can go off at 0 meters from the robot if it is targetted precisely. Robot detonations cause the same amount of damage whether the robot was killed with a missile or it used a destruct command.

Robot performance:	per game cycle:
Normal missile velocity:	32 meters
Maximum non-overburn speed:	4 meters
Turn rate:	8 degrees
Acceleration	4% of max forward speed
Turret rotation	Instantaneous

WATCH YOUR HEAT!! Over-heated robots can easily become DEAD robots!

Here is the heat scale:
~~~~~~~

```
 80+   Max speed reduced to 98%
100+   Max speed reduced to 95%.
150+   Max speed reduced to 85%.
200+   Max speed reduced to 70%.
250+   Max speed reduced to 50%.
300+   Heat starts burning armor off (the hotter, the faster)
400+   Robot stops executing commands until cools below 350 (shutdown).
500+   Robot explodes.
```

Also note that during shutdown, the robot's speed is reduced to 0, making it a sitting duck.

The shutdown temperature can be altered by accessing port 20.

Normal heat exchanges: (can be altered with robot configs)

```
Firing weapon:                 20
Dissipation per game cycle:     1   ( +1/8 if throttle is -25% to 25% )
```

OVER-BURN SHOULD BE USED WITH CAUTION!!!

Overburn effects the following:

1) faster driving speed	(130%)
2) weapon does more damage	(125%)
3) weapon projectiles move faster	(125%)
4) weapon generates more heat	(150%)
5) nastier explosion from death or self-destruct	(130%)
6) worse heat dissipation	(66%)

(all percentages are of original total- i.e. 130% means 30% increase).

Note that Overburn multipliers are applied to your robot's actual status. For instance, if your driving speed has been adjusted to 150% using robot configs, and then you turn on overburn, you're actually going to go 95% over standard speed (195%).

HEXIDECIMAL:

This section is intended just as a refresher, not as a lesson in hexidecimal.

Hexidecimal is very useful to programmers since each digit perfectly represents 4 bits (a nybble, which is half of a byte). 4 bits can store a number from 0 to (2^4)-1, or 0 - 15. Since there are no single-digit symbols for 10 to 15, we use A to F:

```
0 =  0
1 =  1
2 =  2
3 =  3
```

```
 4 =  4
 5 =  5
 6 =  6
 7 =  7
 8 =  8
 9 =  9
 A = 10
 B = 11
 C = 12
 D = 13
 E = 14
 F = 15
10 = 16
```

The right-most digit in a hexidecimal number is still the one's column, but instead of 10's, 100's and 1000's we have 16's, 256's, and 4096's. To convert a hexidecimal number to decimal, simply take the digit from each column and multiply by the columns value:

```
A148 =    A * 4096     =   40960
       +  1 *  256     =  +  256
       +  4 *   16     =  +   64
       +  8 *    1     =  +    8
                          ~~~~~~~~~
                           = 41288
```

Like I said, just a refresher.

PROGRAMMING ROBOTS:

When a program is loaded into AT-Robots, it undergoes a process called "compiling" where each number, word, and instruction is converted into a number. These numbers are called OP-CODES, or OPs for short. As you will see below, the compiler is extremely flexible, and you can get away with all sorts of bizarre combinations and still get a not only a clean compile, but also a functional robot program.

The programming language consists of several basic types of statements.

1) Remarks. - These are not compiled into your program, they are simply notes that you can read when viewing programs. Use these to put comments in about what the program is doing, to make it more understanable when looking through it.
2) Directives - These are used exclusively by the compiler, and are not executed at run-time. The best examples are Var Defs (#def)
3) Labels. - These are the target locations for JUMP/GOTO and GOSUB/CALL commands. They are not executed and do not use your robots processor time.
4) Commands. - These are the functions of your robot. They command the robot to perform specified activities.
5) Variables - You can have up to 256 variables. They are 16 bits (holding

a value from -32768 to 32767). Their names must contain only letters, numbers, and underscores, and must begin with a letter. NO symbol-characters! A variable name can be up to 16 characters long.

The word "Instruction" will often be using in place of "Command". Also note that "registers" are simply special cases of variables.

Now for more detail on these types of statements...

- -

STATEMENTS:

Remarks:

Syntax:

```
        ;      <statement>
```

The semi-colon MUST come before the comments/remarks on a line, and whatever you put on the line after it will be ignored by the compiler.
OP-CODE: none, ignored

Indirect (colon) Labels:

Syntax:

```
        :####
```

The colon MUST be the FIRST character on the line (except for spaces). RIGHT after it, without any spaces in-between, you put a number anywhere from 0 to 32767. That number is the label. Do not use the same number twice within the same program. You can not use a hexidecimal number in a label.
OP-CODE: <label number>

Direct (exclamation) Labels:

Syntax:

```
        !####
```

The exclamation mark MUST be the FIRST character on the line (except for spaces). RIGHT after it, without any spaces in-between, you put an identifier, be it a word or a number. Unlike the colon labels, these are not compiled into the program, but rather a note is made of the position within the program to use as a destination for jumps, etc.
OP-CODE: <none>

Commands/Instructions:

Syntax:

```
        Command [parameters]
```

The command may or may not have parameters, but if there are, you

seperate them from the command with spaces, tabs, or commas. Only one instruction may be put on each line.
OP-CODE: defined on chart below.

Variable Definitions:

Syntax:

#def <variable name>

The definition MUST come BEFORE the first time the variable is used in the program. Don't worry about putting the definitions inside loops, or after the program starts, these can be put just about anywhere, and are not compiled into the program. They are used to figure out how to compile the program, but do not get turned into executable code.
OP-CODE: none, used by compiler only to assign memory addresses.

Variables:

Once a variable has been declared, it can be used anywhere in the program any way you like. The compiler is VERY flexible. All the things you can get away with are described later, but remember, you can do just about anything with these suckers..
OP-CODE: <mem address>

Pre-compiled Machine Code:

Syntax:

*N1,N2,N3,N4

Normally you can enter direct numeric codes for memory locations and instructions. However, these still require you to use brackets and '@'/'!' symbols in order for the microcode to be set up correctly. This directive instructs the compiler to directly enter 4 numbers in as an instruction in the program, much like the 'db' statement in PC assembly. These numbers can be decimal or hexidecimal, and must be seperated by spaces or commas or any of the other usual white-space characters. The left-most number is the instruction, the right-most is the microcode. For details on how the microcode works, you'll really need to look at the simulator's source code, since no detail will be given here. It is not recommended that you use pre-compiled code in your programs. However, this option has been included to add flexibility for those who wish to write third-party programs, such as genetic robot-evolving programs.

\- -

COMPILER DIRECTIVES:

The most important compiler directives are variable definitions, which are shown above. Here are the others currently implemented:

```
#TIME <number>     - Specifies the maximum timeslice for the robot.
                     If higher than the current robot timeslice that
                     the game is using, it will not speed up your robot
```

to give it an unfair advantage. This simply sets a limit, so that your robot will not execute its program any faster than this. i.e. TIME 5 means that the robot will never execute more than 5 CPU cycles per game cycle, no matter how high the timeslice setting is.

#MSG <message> - This displays a message in the robot's display area. Currently, the message may be 31 characters long, but only 19 will be shown in the display area. Since this is a compiler directive and not a run-time instruction, your robot may only have one overall message.

#CONFIG <config> - This directive assigns a point-value to a given device in your robot's body. By manipulating these, you can create a wide-range of configurations to custom-tweek your robot to suit your tactics. The configuration options you can use are in the section below entitled "ROBOT CONFIGURATIONS".

Remember that these are COMPILER instructions. They are executed at compile-time and not at run-time. Therefore the LAST one of each specified in the robot is the one that will be used throughout the simulation. You can not have multiple #MSG's that let you know what the robot is doing at different points in the program (text strings can not be stored in compiled robot programs).

All versions of AT-Robots-2 simply ignore (without an error) any unknown directives, so using them will not make your robot useless on earlier versions of ATR2. This allows new configs to be added easily and still maintain downward compatibility.

- -

ROBOT CONFIGURATIONS:

Your robot has many devices at its command, such as a scanner, a weapon, and an engine. You can adjust these devices, and change the emphasis in your robot's design to custom-tweek it to suit your particular choice of tactics. There are also some devices that can be added to your robot that it would not otherwise have. These adjustments are made using the compiler directive called "config".

For example: #config scanner=1 ; This reduces scan range to 350 meters.

By manipulating the number of points you have invested in each device, you can change the overall configuration of your robot. Here are the tables showing the effect and point-costs for each device: (an asterisk (*) denotes the default value for each device)

Points	Scanner	Weapon	Armor	Engine	Heatsinks	Mines	Shield

0	250	0.50	0.50,1.33	0.50	0.75	2*	None*
1	350	0.80	0.66,1.20	0.80	1.00*	4	-
2	500	1.00*	1.00,1.00*	1.00*	1.125	6	-
3	700	1.20	1.20,0.85	1.20	1.25	10	Weak
4	1000	1.35	1.30,0.75	1.35	1.33	16	Medium
5	1500*	1.50	1.50,0.66	1.50	1.50	24	Strong

Scanner: This is the maximum range your scanner can see.
Weapon: A multiplier placed upon your missiles damage, speed, and heat.
Armor: Two multipliers: 1. Armor multiplier 2. Speed multiplier
Engine: This is the multiplier applied to your maximum speed.
Heatsinks: Change in heat dissipation (multiplier).
Mines: This is the number of mines you start the battle with.
Shield: It takes at least 3 points to get one. Blocks damage

Shield damage blocking:

Weak: 2/3 damage gets through, and 2/3 converted to heat. (1/3 overlap)
Medium: 1/2 damage gets through, other half turned to heat.
Strong: 1/3 damage gets through, and 1/3 converted to heat.

These multipliers are adjustments placed upon what the standard numbers would be. For instance, Heatsinks=5 will dissipate heat 50% faster than the default of Heatsinks=1.

You have a maximum of 12 points to allocate to all these systems and devices. You can total less than 12, but not more.

Note- The speed multipliers from the engine and armor are both factored into your maximum speed. However, the throttle is still percentage based, so Throttle 100 is still maximum speed. Same goes for your armor; 100% is still maximum armor. The armor difference is actually achieved by adjusting the damage taken rather than the total number of armor points.

- -

REGISTERS:

The ROS (Robot Operating System) has a series of several memory locations that are set aside for certain functions. They are called registers. They are used for various things, including interrupt parameters, loop control, and comparisons. The following are their most common uses and their memory locations and op-codes.

Register:	Mem/Op:	Function:	Name:
Flags	64	Comparisons, conditional jumps.	Flags
AX	65	Interrupt instruction, general use	Accumulator
BX	66	General use	Base Reg.
CX	67	Loop control	Counter Reg.
DX	68	General use	Data Register
EX	69	Returns from interrupt calls	Extended Reg.
FX	70	Returns from interrupt calls	Function Reg.
SP	71	Controls the stack.	Stack Pointer

FLAGS:

  Individual bits in the flags register have specific meanings:
  R = Reserved, O = Open for your use

  Note that the lower nybble of the upper byte is available for your uses.

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00  Name:                Mask:
  R  R  R  R  O  O  O  O  R  R  R  R  |  |  |  +- Equal flag     (eq)  0001h
                                     |  |  +---- Less-than flag (lf)  0002h
                                     |  +------- Grtr-than flag (gf)  0004h
                                     +---------- Zero flag      (zf)  0008h
```

  Executing a CMP or a TEST changes the entire low nybble.

  CMP results:
  ~~~~~~~~~~~~
  Equal flag: Set when operands are equal.
  Greater flag: Set when operand#1 > operand#2
  Less flag: Set when operand#1 < operand#2
  Zero flag: Set when operands are equal AND are 0.

  All comparisons are done as SIGNED values. That means 0xF000 < 0.

  Examples: (results show flags value in binary)

```
  CMP 1, 1   ; 0001
  CMP 0, 1   ; 0010
  CMP 1, 0   ; 0100
  CMP 0, 0   ; 1001
```

  TEST results:
  ~~~~~~~~~~~~~
  Equal flag: Set when operands are equal.
  Zero flag: Set when the binary "AND" of operands #1 & #2 = 0.

  After a TEST, the Greater flag and the Less flag are always 0.

  Examples: (results show flags value in binary)

```
  TEST 1, 1  ; 0001
  TEST 0, 1  ; 1000
  TEST 1, 0  ; 1000
  TEST 0, 0  ; 1001
```

  Because the zero-flag is dependant on the "AND" result of the two operands, it is a good way to see if a bit is turned on. Let's say you want to look at the lowest USER bit in flags, the following code will TEST this bit.

```
   mov   ax, flags   ; make a copy of "flags", just to be safe
                     ; since TEST will change "flags".
   test  ax, 0100h   ; Test bit 8 (mask of 256 or 0100h)
```

```
   jz    100         ; If bit is off, goto :100
   jnz   200         ; If bit is on,  goto :200
```

- -

INSTRUCTIONS/COMMANDS:

The instructions in a robot program tell the robot what to do. The following list shows their names, syntax, and how many CPU clock-ticks they require (some take longer than others to execute). The chart below is for your reference. To read a discussion about how to use them skip along.

The parameters that come after an instruction are called operands, and sometimes they must be of specific types. The types are as follows:

N = number, of any type. Could be a number or value from a variable.
V = variable, meaning it must be a variable, register, or memory access.

In the chart header below, T refers to the time it takes to execute an instruction

Op:	T:	Name:	Alt:	Syntax/Description:	
0	1	NOP		NOP	Simply wastes a clock-cycle.
1	1	ADD		ADD V N	Adds V+N, result stored in V
2	1	SUB		SUB V N	Subtracts V-N, result stored in V
33	1	INC		INC V	Increments V, (v=v+1)
34	1	DEC		DEC V	Decrements V, (v=v-1)
35	1	SHL		SHL V N	Bit-shifts V left N bit positions
36	1	SHR		SHR V N	Bit-shifts V right N bit positions
37	1	ROL		ROL V N	Bit-rotates V left N bit positions
38	1	ROR		ROR V N	Bit-rotates V right N bit positions
43	1	SAL		SAL V N	Bit-Shifts, same as SHL
44	1	SAR		SAR V N	Same as SHR, except preserves bit 15.
45	1	NEG		NEG V	Negates V: V = 0-V (aka "two's compliment")
3	1	OR		OR V N	Bitwise OR, V or N, result stored in V
4	1	AND		AND V N	Bitwise AND, V and N, result stored in V
5	1	XOR		XOR V N	Bitwise XOR, V xor N, result stored in V
6	1	NOT		NOT V	Bitwise NOT, not(V), result stored in V
7	10	MPY		MPY V N	Mulitplies V*N, result stored in V
8	10	DIV		DIV V N	Divides V/N, result stored in V (integer)
9	10	MOD		MOD V N	MOD's V & N, result stored in V (modulus)
10	1	RET	RETURN	RET	Returns from a subroutine (pops the ip)
11	1	CALL	GSB	CALL N	Calls subroutine at label #N (pushes ip)
12	1	JMP	GOTO	JMP N	Jumps program (ip) to label #N
20	1	CMP		CMP N N	Compares two numbers, results in flags reg.
13	0	JLS	JB	JLS N	Jumps to label N if last compare was <
14	0	JGR	JA	JGR N	Jumps to label N if last compare was >
15	0	JNE		JNE N	Jumps to label N if last compare was <>
16	0	JEQ	JE	JEQ N	Jumps to label N if last compare was =
41	0	JAE	JGE	JAE N	Jumps to label N if last compare was >=
42	0	JBE	JLE	JBE N	Jumps to label N if last compare was <=
39	0	JZ		JZ N	Jumps to label N if last compare was 0

```
40  0  JNZ           JNZ N      Jumps to label N if last compare was not 0
46  1  JTL           JTL N      Jumps to line N of compiled program.
17  3  XCHG SWAP     XCHG V V   Exchanges the values of two variables
18  1  DO            DO N       Sets CX = N
19  1  LOOP          LOOP N     Decrements CX, If CX>0 then Jumps to label N
21  2  TEST          TEST N N   Ands two numbers, result not stored, flags set
22  1  MOV  SET      MOV V N    Sets V = N
23  2  LOC  ADDR     LOC V V    Sets first V = memory address of second V
24  2  GET           GET V N    Sets V = number from memory location N
25  2  PUT           PUT N1 N2  Sets memory location N2 = N1
26  ?  INT           INT N      Executes interrupt number N
27  4+ IPO  IN       IPO N V    Inputs number from port N, result into V
28  4+ OPO  OUT      OPO N1 N2  Outputs N2 to port N1
29  ?  DEL  DELAY    DEL N      Equivelant to N NOPS.
30  1  PUSH          PUSH N     Puts N onto the stack (sp incremented)
31  1  POP           POP  V     Removes a number from the stack, into V
32  0  ERR  ERROR    ERR N      Generate an error code, useful for debugging
```

Note that some instructions take 0 time to execute. This was done so as to not discourage making complex jump-lists. Note however that 0-time instructions made it theoretically possible to send the SIMULATOR into an endless loop if you specifically designed robots to do so. Therefore, if 20 0-time instructions are executed within a single game-cycle, they count as using 1 robot cpu cycle. There is also an implied NOP at the end of each program (so that 0-length programs are impossible).

All comparisons, conditional jumps, and math functions are based on SIGNED 16-bit operations.

JTL is the only jump instruction that does not assume the older, indirect, colon-prefix labels. You can use the newer, direct labels (exclamation-prefix) with ALL jumps, but if you wish to specify a program location by number, for which there is no label, you must use JTL.

JTL jumps to the program line number specified as an operand. Remember that this uses COMPILED lines, with the counting 0-based (i.e. first program instruction is line 0). Indirect labels (:labels) also count as compiled lines.

Just remember that you can only have 256 !labels.

example:

```
       xor ax, ax    ; ax = 0,   this is position 0.
!begin               ; label is at position 1, right after position 0.

       ;...   30 instructions and :labels have passed

!start               ; Let's assume this represents compiled line 30
       XOR ax, ax    ; this is the actual line 30...  ax = 0

       JTL !start    ; Jump to !start
       JMP !start    ; Also Jump to !start
                     ;   (both cases are endless loops of ax=0)
```

```
                    ; Assuming we get past them magically...
        mov ax, 1   ; ax = 1

:1                  ; Now we get to see the difference:
        JTL ax      ; Jumps to position 1, which is at !begin
        JMP ax      ; Jumps to :1, because all jumps other than JTL
                    ;   assume the old :label system instead of the
                    ;   newer !label system.
```

All of the conditional jumps work the same way as JMP above.

- -

PORTS:

I/O ports are how your robot program accesses the various devices that the robot is equipped with. Each device is assigned a port number. It might be handy to have a print-out of this chart at the very least.

If you have used ATR1, take notice that ports 1 through 17 remain the same.

In the chart header below, once again T represents the time it takes to access the port in addition to the execution time for the port access instruction.

Num:	T:	I/O	Name:	Function:
1	0	I	Spedometer	Returns current throttle setting[-75- 100]
2	0	I	Heat Sensor	Returns current heat-level [0 - 500]
3	0	I	Compass	Returns current heading [0 - 255]
4	0	I	Turret Sensor	Returns current turret offset [0 - 255]
5	0	I	Turret Sensor	Returns absolute turret heading [0 - 255]
6	0	I	Damage Sensor	Returns current armor level [0 - 100]
7	1	I	Scanner	Returns range to nearest target in scan arc
8	1	I	Accuracy	Returns accuracy of last scan [-2 - 2]
9	3	I	Radar	Returns range to nearest target
10	0	I	Random Generator	Returns random number [-32768 - 32767]
11	0	O	Throttle	Sets throttle [-75 - 100]
12	0	O	Rotate Turret	Offsets turret (cumulative)
13	0	O	Aim Turret	Sets turret offset to value [0 - 255]
14	0	O	Steering	Turn specified number of degrees
15	3	O	Weapon control	Fires weapon w/ angle adjustment [-4 - 4]
16	40	I	Sonar	Returns heading to nearest target[0 - 255]
17	0	I/O	Scan-Arc	Sets/Returns scan-arc width. [0 - 64]
18	0	I/O	Overburn	Sets/Returns overburn status
19	0	I/O	Transponder	Sets/Returns current transponder ID
20	0	I/O	Shutdown-Level	Sets/Returns shutdown-level.
21	0	I/O	Com Channel	Sets/Returns com channel setting
22	0	I/O	Mine Layer	Lays mine or Returns mines-remaining.
23	0	I/O	Mine Trigger	Detonates/returns previously-placed mines.
24	0	I/O	Shield	Sets/Returns shield's status (0=off, else=on)

Port 13, Aim-Turret, sets the turret offset to the value given. This is an OFFSET, which means that it is RELATIVE to the robot body. (0 is straight ahead, 128 is straight back, etc). Port 12 is similar, but it is a rotation command, meaning that the turret rotates the specified number of degrees (10 is 10 to the right, -10 or 246 is 10 to the left). Port 12 is therefore much like the steering port (port 14).

There is no port that allows you to set your turret to an absolute compass heading. You will have to calculate this yourself. However, you can access port 5 to GET your absolute turret heading (this is simply a compass mounted on the turret). Port 4 returns the turret offset, which once again is the heading of the turret relative to the robot body.

The compass (port 3) returns your current absolute heading, which may not yet have caught up with the last turn command.

The spedometer (port 1) returns what your current velocity is relative to your maximum. The maximum can be affected by heat level and overburn status. i.e., setting your throttle to 100 will always cause the spedometer to report 100 once the robot reaches full speed, unless you change the throttle, shutdown, reset, or collide with something. Sending a negative number to the throttle port (port 11) will make the robot drive backwards.

The scan-arc is defined as being the number of degrees to EACH side of the center of your scan. If you set your arc to 64, you actually get a 128 total arc (half of the full circle!) (64 is the maximum).

The scanner returns a very large number somewhere over 1500 (such as MAXINT) if no target is found.

The weapon can fire as much as 4 degrees in either direction without turning the turret or the robot. If you leave off the final operand on the port access, as usual a 0 is assumed and the weapon fires straight ahead.

The Sonar has a maximum range of 250. It returns a negative number if no target is within range. Also notice that it is extremely slow to access. It also has a poor accuracy. The number returned could be as far off as 32 degrees in either direction (equivelant of 45 degrees in a normal 360 degree circle). You'll notice that sonar is SO slow, and SO innacurate, that it is almost useless. It is merely here to give you a simple method to find the nearest target, yet encourage you to create your own search routines with the scanner.

When accessing the overburn port, 0=off, and anything non zero is on.

Port 19, the transponder, simply sets or returns your robots ID number. At the beginning of the battle, the transponder starts off being a number from 1 to 6 depending on which "slot" your robot is taking up in the simulation.

Port 20, Shutdown-level, sets/returns the heat-level at which automatic shutdown will ocurr. When the robot shuts down, it will remain so until it cools down below the shutdown level minus 50 (default level is 400,

therefore the robot will start up again at 350), or when the heat drops to 0.

Ports 22 and 23 operate the mine-layer, if your robot is equipped with one. Outputting to port 22 will cause the mine-layer to place a mine, if it has any mines left. The number you output to the port will be used by the mine as its detection radius. If set to 0, the mine will most likely never go off, since it will require a robot to be perfectly centered over it. 5 will require the robot to still get very close, but will do a great deal of damage. Setting it to 15 or more may diminish the damage, but make it much more likely to go off when a robot approaches. Reading from this port (22) will give you the number of mines remaining in the mine-layer. Writing to port 23 will detonate all of the mines you have placed in the arena that have not already gone off. This is the only way to detonate your own mines, since they're smart enough not to go off on you, only on other robots. Reading from 23 will tell you how many such mines you currently have in the battlefield.

Port 24 is for the shield. If your robot is equipped with a shield, you will use this port to operate it. The shield is an energy-field that prevents the transfer of heat. Since the missiles are plasma-based, they typically damage the robot by burning it. The plasma also causes the robot firing the missile to generate a lot of heat that must be dissipated. With the shield raised, your robot will be less exposed to attacks via mines or missiles (it will not protect against physical damage, so collisions and overheating internally still hurt you). However, since the shield blocks heat, your robot will not be able to cool itself at all while the shield is in use. Unfortunately, the shield itself generates heat slowly as well, so you can not use it indefinitely. Writing a 0 to port 24 will lower the shield, any other number will turn it on. Reading from the port will return a 0 if the shield is off, or a non-zero number if it is on.

- -

INTERRUPT CALLS:

In addition to I/O ports, the robot is equipped with some built-in functions that can be used. These functions are called interrupts. In your programs you can use the INT command to execute an interrupt. When you call an interrupt, you will often need to set some of the system registers before executing it, to tell it what to do.

Unlike in ATR1, you can not specify your own interrupt vectors. Also, all of the interrupts have been redefined and are not the same as they were in ATR1.

In the chart header below, once again T represents the time it takes to execute the interrupt call, although unlike with the ports, this is the total time usage.

Num:	T:	Name:	Function:
0	-	Destruct	Detonate the robot (if I go down, you go down with me!)
1	10	Reset	Resets robot program.

```
 2   5 Locate       Sets EX,FX registers equal to X,Y coordinates.
 3   2 Keepshift    Sets Keepshift, input:     AX 0 = off, non-0 = on
 4   1 Overburn     Sets Overburn,  input:     AX 0 = off, non-0 = on
 5   2 ID           Returns robot ID number in FX
 6   2 Timer        Returns game clock in EX:FX (32-bit number)
 7  32 Find Angle   Returns angle to point specified in EX,FX; AX=result
 8   1 Target-ID    Returns ID of last robot scanned (with any scan) in FX
 9   2 Target-Info  Returns info on last scanned target (EX=dir,FX=throttle)
10   4 Game-Info    Returns info: DX=Total number of robots active,
                                  EX=Match number,
                                  FX=Number of matches
11   5 Robot-info   Returns info: DX=Robot speed (in cm per game-cycle),
                                  EX=Time since last damage taken
                                  FX=Time since a fired shot hit a robot.
                                  (time measured in game-cycles)
                                  (robot speed is current as of int call)
12   1 Collisions   Returns collision count in FX
13   1 Reset ColCnt Resets collision count back to 0.
14   1 Transmit     Transmits the data in AX on the current channel.
15   1 Receive      Returns the next item in com queue in FX
16   1 DataReady    Returns the amount of data in queue in FX (0 for none).
17   1 ClearCom     Empties the Com Queue
18   3 Kills/Deaths Returns info: DX=Kill Count (spans multiple rounds)
                                  EX=Kill Count (for this round only)
                                  FX=Deaths     (spans multiple rounds)
19   1 ClearMeters  Resets the 'meters' variable to 0.
```

Keepshift, when active, makes it so that whenever the robot body turns, the turret turns by the same amount in the opposite direction, thus the turret always faces the same way relative to the arena.

Int 5 returns your current transponder ID, as described in the PORTS section.

The timer call returns the 32-bit game clock value. Since everything in the game is handled as signed numbers, the next value above 32767 is -32768, therefore you can only reliably access the clock for 32767 cycles into the game. Most battles are completed in less than 10000 cycles though. EX holds the upper 16 bits, and FX holds the lower 16 bits of the clock value.

Interrupt 7 is very slow to execute, since it is basically doing a trigonometric function. You put the X,Y coordinates you want into EX,FX, call the int, and afterwards AX will hold the heading you should take to get there. If you specify numbers outside the range of 0-1000, they will be truncated into this range before the calculation is performed (i.e. calling as EX=12345, FX=-500 will return the results for EX=1000, FX=0).

Interrupt 8 will return whatever transponder ID the target robot has set.

Interrupt 9 returns info on the last target scanned using the primary scanner (port 7). The direction of the target (relative to your scan) is returned in EX (i.e. 64 means it is facing to the right of your scan and 0 means it is facing the same way you are scanning, which of course is away

from you). FX returns the targets current throttle (not the "desired throttle"), so that you may make some assesments of where it is going and how quickly.

Interrupt 11 has a nice feature, being able to tell how recently you've actually hit someone. This can facilitate making robots that can realize they're not hitting anything.

Interrupts 12 and 13 allow you to easily detect collisions, without relying on your spedometer or damage totals to tell you these things (especially considering that if you were already sitting still, and someone rammed you very slowly, there would be no other way to know that a collision ocurred). Int 12 simply returns the collision-count value that is stored in memory location @8. This value is incrememnted everytime a collision ocurrs, so any time it has changed you have been involved in a collision, whether it was robot-to-robot or robot-to-wall. Int 13 resets it back to zero, so that you could have some simple code that simply checks to see if the count is non-zero (and hence a collision has ocurred since the last reset with int 13).

Interrupts 14 through 16 deal with the robot's communications system. Using interrupt 14 will transmit whatever integer is stored in AX to all robots tuned to the same channel. When the data is received by a robot, it is put into a queue stored in memory. If the queue overflows, no error will be generated- instead the oldest piece of data will be lost in favor of the new data. Since the queue can hold 255 integers, a robot that doesn't check the queue often enough will always have the most recent 255 integers of data to be received. Interrupt 15 removes a piece of data from the queue, and puts it into FX. Interrupt 16 tells how much data is currently in the queue and waiting to be extracted. The channel can be set using a port access.

- -

MEMORY MAP:

The robot's memory core has 1024 memory locations, each storing a 16-bit integer value. When the robot is initialized, all memory addresses store a zero.

The first 128 addresses are the system area. You can define up to 256 variables, which are stored right after the system area. The remaining 128 memory locations are available for you to do whatever else you want, such as storing self-modifying code (if you are up to such a nasty challenge!) (self-modifying code may become more practical in later revisions, for now though you would have to have all of the instructions set up as memory assignments, like follows: @384 @385, @386 ; rewritable instruction #1)

Each memory location stores 16 bits, as stated above. A signed 16 bit number can be in the range of -32768 to 32767. If you look at it as a hexidecimal number, you will ignore the "sign" of the number. If it were unsigned, it would appear to have a range of 0 to 65535, or in hexidecimal 0000h to FFFFh.

Memory map:

Addr:	Name:	Function:
0	Dspd	Desired speed robot is trying to achieve.
1	Dhd	Desired heading robot is trying to achieve.
2	tpos	Current turret offset
3	acc	accuracy value from last scan
4	swap	temporary swap space used by swap/xchg instruction
5	tr-id	ID of last target scanned (by any scan).
6	tr-dir	Relative heading of last target scanned.
7	tr-spd	Throttle of last target scanned.
8	ColCnt	Collision count.
9	Meters	Meters travelled. 15 bits used.. (32767+1)=0
10	ComBase	Current base of the communications queue
11	ComEnd	Current end-point of the communications queue
13	tr-vel	Absolute speed (cm/cycle) of last target scanned
14-63	res	reserved
64	flags	flags register
65	ax	ax register
66	bx	bx register
67	cx	cx register
68	dx	dx register
69	ex	ex register
70	fx	fx register
71	sp	stack pointer
72-95	res	reserved for future registers
96-127	res	reserved
128-384	var	variable space
385-511	user	user space
512-767	com-queue	Communications receiving queue
768-1023	res	Reserved space
1024...	code	Robot program (code segment) (ROM)

Please note that the first 4 memory locations make for an easy and effective way to get info about your robot's status. Robots can be made to be more efficient by accessing these. However, these are here ONLY for getting info about the robot. If you try changing these values, it will have no effect. To actually cause the robot to turn or whatever, you must still access the appropriate ports.

The robot's program is technically not stored in the robot's RAM, but rather is in the robot's ROM along with the robot's BIOS (which handles the interrupt calls).

You can now access the robot program as though it were in RAM, starting at address 1024. Please remember though that you can not jump or RET or anything below 1024. As far as the robot's IP (instruction pointer) is concerned, memory address 1024 is instruction address 0.

- -

CONSTANTS:

As you will see described further down under "writing the code", the instructions are nothing more than pre-defined constants representing numbers. Here is a chart of all of the pre-defined constants that you can use in your programs, so that you don't have to memorize port and interrupt numbers.

Value:	Mnemonic:	Port-Constant:	Interrupt-Constant:	Generic Constant:
-32768				MININT
32767				MAXINT
0	NOP		I_DESTRUCT	
1	ADD	P_SPEDOMETER	I_RESET	
2	SUB	P_HEAT	I_LOCATE	
3	OR	P_COMPASS	I_KEEPSHIFT	
4	AND	P_TURRET_OFS	I_OVERBURN	
5	XOR	P_TURRET_ABS	I_ID	
6	NOT	P_DAMAGE, P_ARMOR	I_TIMER	
7	MPY	P_SCAN	I_ANGLE	
8	DIV	P_ACCURACY	I_TID, I_TARGETID	
9	MOD	P_RADAR	I_TINFO, I_TARGETINFO	
10	RET	P_RANDOM, P_RAND	I_GINFO, I_GAMEINFO	
11	GSB, CALL	P_THROTTLE	I_RINFO, I_ROBOTINFO	
12	JMP, GOTO	P_OFS_TURRET, P_TROTATE	I_COLLISIONS	
13	JLS, JB	P_ABS_TURRET, P_TAIM	I_RESETCOLCNT	
14	JGR, JA	P_STEERING	I_TRANSMIT	
15	JNE	P_WEAP, P_WEAPON,P_FIRE	I_RECEIVE	
16	JEQ, JE	P_SONAR	I_DATAREADY	
17	XCHG, SWAP	P_ARC, P_SCANARC	I_CLEARCOM	
18	DO	P_OVERBURN	I_KILLS, I_DEATHS	
19	LOOP	P_TRANSPONDER	I_CLEARMETERS	
20	CMP	P_SHUTDOWN		
21	TEST	P_CHANNEL		
22	SET, MOV	P_MINELAYER		
23	LOC	P_MINETRIGGER		
24	GET	P_SHIELD,P_SHIELDS		
25	PUT			
26	INT			
27	IPO, IN			
28	OPO, OUT			
29	DEL, DELAY			
30	PUSH			
31	POP			
32	ERR, ERROR			
33	INC			
34	DEC			
35	SHL			
36	SHR			
37	ROL			
38	ROR			
39	JZ			
40	JNZ			
41	JAE, JGE			
42	JBE, JLE			
43	SAL			
44	SAR			

```
 45          NEG
 46          JTL
```

- -

WRITING THE CODE:

Ok, now that you've seen the reference charts, how 'bout learning the syntax of the language, and a few of the subtleties? Below is a crash course, emphasizing CRASH. It is assumed that you have some familiarity with programming concepts. Even though the explanations start out being extremely basic, it gets into the more specific nuances of the language very quickly. Everything else you may need to teach yourself, perhaps through trial and error.

Every command line consists of an instruction (sometimes called a command, or an operator), and usually one or two operands. The compiler is not too picky about what is used where, so most things are interchangable. Also remember that capitalization is totally irrelevant.

Let's look at an example:

```
Add   ax,  2
```

This takes the value in the register AX, adds 2, then stores it back into the AX register.

You can create remarks/comments by using the semi-colon. From now on you will see comments next to the instructions explaining them.

```
        Add     ax,      2      ; ax = ax+2
```

If you wish, you can use loop commands to repeat something several times:

```
        do      5               ; cx = 5
:1
        add     ax,     2       ; ax = ax+2
        loop    1               ; decrement cx, if >0 goto :1
```

In this previous example, it would have the effect of simply adding 10 (2 added 5 times) to AX. Make sure the loop is structured properly though. Here is what NOT to do:

```
:1
        do      5               ; cx = 5
        add     ax,     2       ; ax = ax+2
        loop    1               ; decrement cx, if >0 goto :1
```

In this past example, CX is reset to 5 everytime through the loop, therefore never reaching 0. What you have is an infinite loop. Of course, making infinite loops is easy:

```
:1
```

```
        jmp     1
```

  Or to make it a little more wierd, try it this way:

```
        mov  ax,  1
:1
        jmp  ax
```

  Or you can make it even more weird:

```
        mov  ax,  70      ; AX = address of FX
        mov  fx,  1       ; FX = 1
:1
        jmp  [ax]         ; Jump to line label specified in FX.
```

  To put a specific number into a variable/register, simply use SET/MOV:

```
        mov     ax,     15      ; ax = 15
```

  Or you could use the memory address:

```
        mov    @65,    15       ; ax = 15
```

  Or copy a variable:

```
        mov    @65,    @66      ; ax = bx
        mov    ax,     bx       ; ax = bx
        mov    [65],   [66]     ; ax = bx
```

  In this past example, as far as the compiler is concerned, the first two lines are identical. Variable names simply get replaced with the addresses that they represent. For that matter, so do the instructions themselves, as you will see further down. The third line has the same result, but is compiled slightly differently.

  You can also use indirect addressing:

```
        mov    bx,     70       ; bx = 70 (the memory location for FX)
        mov    ax,     [bx]     ; ax = fx
```

  Now let's look at how you would define your own variable:

```
#def x
#def my_var

        xor     x,      x       ; x = 0 (anything xor'ed with itself is 0)
        mov     my_var, x       ; my_var = x (and therefore equals 0)
```

  Now, as stated above, the instructions are nothing more than words that represent numbers, just like everything else:

```
       add      ax,     ax      ; ax = ax+ax (in other words, ax = ax*2)
```

```
      1        ax,     ax      ; Same thing! (add = 1)
      1        @65     @65     ; Same thing!!
```

  In fact, you can use variables in place of instructions:

```
#def inst

        mov     inst,   1       ; inst = 1
        inst    @65,    @65     ; same as "ADD AX, AX"
```

  If you leave off an operand, a 0 is assumed:

```
        mov     ax              ; ax = 0
```

  If you execute an interrupt this way, you get an interesting effect:

```
        int                     ; call int #0, kaboom! farewell cruel world!
```

  You could make a destruct deciding routine that would simply change a variable. INT is 26, by the way

```
        xor     dx,     dx      ; dx = 0

        ipo     6,      armor   ; armor = armor value from armor sensor
        cmp     armor,  10      ; campare armor to 10
        ja      1000            ; if greater than 10, skip to :1000
        je      2000            ; if equal to 10, skip to :2000
        mov     dx,     26      ; dx = 26
:1000
        dx                      ; if dx=26 then kaboom! if 0, then nothing.
:2000
```

  You can also use the stack to store numbers if it is convenient:

```
        push    ax              ; put ax on the stack.
        pop     bx              ; pop a number off the stack into bx,
                                ;    since the last thing pushed was ax,
                                ;    whatever was in ax is now in bx.
```

  Be careful with the stack though, since it is the same stack used for CALL/GOSUB/GSB and RET instructions. If you use PUSH and POP in a subroutine, be sure to pop everything back off before you use RET.

```
:1
        CALL    1000    ; call subroutine at :1000
        jmp     1       ; loop back to :1, thus endlessly repeating the CALL

:1000
        opo     15      ; FIRE! (Output to port 15 fires weapon)
                        ;   no second operand, so 0 is assumed. Fire straight.
                        ;   (weapon can fire up to 4 degrees off center)
        ret             ; return to source of CALL.
```

  As shown above, you can easily create subroutines, which of course

facilitate making code that you can access from multiple places in the program, and still have it always return to where you called it from.

Now let's discuss nested loops. Since CX is used for loops, CX has to be temporarily stored elsewhere during inner loops:

```
        DO      5               ; cx = 5
:1
        opo     13,     8       ; turn turret 8 degrees right

        mov     bx,     cx      ; bx = cx
        DO      5               ; cx = 5
:2
        opo     15              ; fire!

        LOOP    2               ; loop up to :2
        mov     cx,     bx      ; cx = bx

        LOOP    1               ; loop up to :1
```

This would fire the weapon 25 times, 5 times at each angle. Perhaps a better and less confusing method would be to use PUSH and POP, that way we don't accidently screw up by doing something with bx.

```
        DO      5               ; cx = 5
:1
        opo     13,     8       ; turn turret 8 degrees right

        push    cx              ; put cx on stack
        DO      5               ; cx = 5
:2
        opo     15              ; fire!

        LOOP    2               ; loop up to :2
        pop     cx              ; get cx from stack

        LOOP    1               ; loop up to :1
```

Also keep in mind how long it takes to do something. The following three instructions all double the value of ax.

```
        MPY     ax,     2       ; ax = ax*2           time used: 10
        shl     ax,     1       ; ax = ax shl 1       time used: 1
        add     ax,     ax      ; ax = ax + ax        time used: 1
```

Also compare these two: (both quadruple AX)

```
        MPY     ax,     4       ; ax = ax*4           time used: 10
        shl     ax,     2       ; ax = ax shl 2       time used: 1
```

Also note that by breaking any multiplication or division down into powers of two you can also exectute fast math. For instance, a multiply by 3 takes 10 clock cycles. The following example uses only 3, and achieves

the same result:

```
        MPY     ax,     3       ; ax = ax*3   (10 cpu cycles)
       ; or try the following instead
        MOV     bx,     ax      ; bx = ax       (1 cpu cycle)
        SHL     ax,     1       ; ax = ax * 2   (1 cpu cycle)
        ADD     ax,     bx      ; ax = ax + bx  (1 cpu cycle)
                                ; net result is:  ax = (ax*2)+ax
                                ;             or  ax = ax*3
```

Please note however that bit shifts don't work so reliably with very large numbers or negative numbers.

Now that you've seen how to do loops, and some of the slick little tricks you can use, I'll end this segment by mentioning the use of the pre-defined constants to help with all those confusing port numbers. Back with the other charts you will find a chart of these constants, and below are a few examples:

```
        opo     p_fire          ; same as "opo 15"   - FIRE!

        p_spedometer   ax, ax   ; same as "1 ax ax" or "add ax ax"
```

I hope that was enough to get you started. Believe me, if you can do this stuff (and this isn't too hard), then you've tackled the basics necessary for doing REAL programming in assembly on the PC. Just remember that the PC is not as forgiving as ATR2!

Now don't go away yet! Below are the error codes your robot might generate...

- -
ERRORS:

Your robots should never generate a runtime error (either in terms of the robot, OR the main game itself, but in this case we mean in terms of the robot). If they do make one, and you have the graphics and status bars turned on, you'll be made aware of it. If the sound is on, then you'll hear it as well. If you have the graphics on, the bottom row of the status display shows the last error code genereated by your robot. If your robot works properly, and you aren't using the ERR command, it should always be "None".

When a robot has a runtime error, the program will continue on and simply not do the command that created the error. Note that the command that had the error STILL uses your robot's time.

If you use the ERR instruction to output info, you might want to display numbers that aren't listed below, so you can tell the difference between your error and a real robot error.

Here are the error codes:

Code: Meaning:

1 Stack full - This means that you have used too many PUSHes and/or CALL/GOSUB/GSB commands without using the appropriate POPs and RETs. This is a common problem for novice programmers.

2 Label not found - This means you're trying to jump or call or something to a label that does not exist. Go make sure all of the labels you call actually exist somewhere. If you are using JTL, the location you are jumping to is outside of the program.

3 Can't assign value - This means you're trying to store the result of an operation into something that is not a variable of some sort. Maybe you got the operand order confused on an IPO command.

4 Illegal memory reference - You're accessing a memory address that does not exist! Remember, there are only so many memory addresses in your robot!

5 Stack empty - This is the reverse of error #1, meaning you're calling too many POPs and RETs compared to your use of PUSH and GOSUB/CALL/GSB commands. This is a much less likely mistake to make.

6 Illegal instruction - Hey! Trying to invent a new instruction eh? This error ocurrs when you try to execute an instruction that does not exist. If you program with actual instruction mnemonics instead of using variables and numbers in their place you will not get this error, unless you redefine a mnemonic as a variable.

7 Return out of range - Your RET command yielded a number that is outside your program. You must have called a RET when the last thing still on the stack was PUSHed there.

8 Divide by ZERO - You executed a DIV or MOD instruction with the second operand being a 0.

9 Unresolved !label - You should never see this happen. No unresolved !labels should make it past the compiler.

10 Invalid Interrupt Call - This simply means that the interrupt you are trying to call does not exist.

11 Invalid Port Access - Either the port you are accessing does not exist, or you are writing to an input port, or you are reading from an output port.

12 Com Queue empty - You're trying to receive data when there's none to receive. You should use Int 16 to check for

data ready.

13 No mine-layer - You're attempting to access mine-layer functions but you don't have a mine-layer installed. You won't get this error just checking to see if you have mines on-board though.

14 No mines left - You're trying to place mines, when in fact you've run out of them.

15 No shield - You're attempting to use a shield but you don't have one.

16 Invalid Microcode - Somehow your robot is trying to execute an instruction with invalid microcode. This should never happen, though it is possible if you put inline machine-code in your programs.

- -

DESIGNING THE PERFECT ROBOT:

Ok, I'll bet you just glanced through the above information and are thinking, "Ok, great. It looks nice, but how do I use it?" I hope to explain a few details here, but I think the sample robot programs are by far the best source of information. By looking through the samples robots, you can pick up a few tricks and tips. Below are a few charts to refer back to at later times, and a few general rules to go by. Afterwards is some more text to help teach you about those strange little features that make the robots so much more deadly, but require some understanding first.

As you create your robot, you will want to first make sure it can handle a stationary target. Afterwards you will want to slowly increase the difficulty. Here is the order in which you should try some of the sample robots as opponents:

1. SDUCK
2. CIRCLES
3. TRACKER
4. SNIPER and/or RANDMAN3

General rules:

1. Try to make all the parameters line up in single columns, they're easier to read that way (hint: use the tab key).
2. You MAY leave blank lines. The compiler ignores them.
3. One command is executed each cycle unless otherwise noted.
4. Examining the sample robots is the best teacher.
5. Commands may be upper-case, lower-case, or any combination.
6. Never make multiple labels of the same ID number within the same program.
7. All robots MUST have the extension AT2.
   ex- MYROBOT.AT2, or THISGUY.AT2

When you perform a scan, the number that is returned is the distance to the nearest target within the scan arc. If no target is found, a huge number

is returned (specifically MAXINT, or 32767). This sets an internal variable as well which is referred to as the "scan accuracy". This number can be accessed by reading from the "accuracy" port. It will be a number from -2 to +2. If the target was all the way on the left side of the scan, this number will be -2. If it were on the right, +2. Dead center is 0, half-way to the left is -1 and half way to the right is +1. Based upon the size of your scan arc, you can use this number to figure out where the target is. The weapon is capable of firing up to 4 degrees in either direction without turning the turret or the whole robot, and so using the accuracy setting can be very useful.

There are several "setting" ports, such as the turret, and arc-width ports. The default settings are as follows:

Item:	Value:
Turret Offset/shift	0
Throttle	0
Scan arc radius	8
Overburn	off
Keepshift	off

All variables are set to 0 at the beginning of the battle. These settings are all restored to default if the reset interrupt is called.

Note that when you rotate the turret, EVERYthing that is on it turns with it. This includes the scanner AND the weapon.

Remember to be careful with the stack. A program can get very mixed up if you don't call the approprite RETs and POPs, in proper order, for all of your CALL/GOSUB/GSB and PUSH commands.

Also remember that if you aren't sure whether a particular section of code is doing what you want, or you're not sure whether it is even executing a section of program code, then try using the ERR instruction to display the results of your calculations. The ERR command does not use up your robot's CPU time, and can be very useful in tracking down bugs in the robot (or in the game itself!) You may also want to turn on the display of scan-arcs (/A parameter, or "A" key during battle) so that you can see where your robot is looking.

THE PHYSICS OF AT-ROBOTS:

This section is purely theoretical and hypothetical, as it hopefully will remain... :-)

Assuming that the arena is a square kilometer, and that the actual simulated time runs at 10 cycles/second, we can extrapolate and extract the following information: (you can see the game run at roughly this speed with the /D100 parameter):

Robot Technical info:

Length:	8 meters or 26.25 feet (in other words, a reasonably sized TANK)
Max velocity (normal):	144.0 kph or 089.48 mph
Max velocity (overburn):	187.2 kph or 116.32 mph
Time from 0 to max speed:	2.5 seconds (0 to 116 in 2.5 seconds!?)
Time to turn fully about:	1.6 seconds (even at full speed!? and without skidding!?!)
Normal missile velocity:	1152 kph or 715.84 mph (nearly 3/4 of mach 1)
Processor speed:	50 hertz
Maximum ROM program size:	8k
RAM capacity:	2k
Ammunition supply:	Unlimited (missiles are generated from the on-board fusion power-plant).

scary, huh?

If you wanted to assume that the game runs at the default of about 30 cycles per second, triple the speeds above. Even more scary, eh?

And an even more scary thought is that these huge, fast tanks are controlled by processors whose speed is measured in hertz, not megahertz.

STAGING A COMPETITION:

When staging a competition, be sure to specify the rules and the settings that will be using in the game (such as timeslice, time limit, and number of matches to be fought). These things must be decided ahead of time, so that the players may optimize their robots for these settings.

Make sure that all players involved are aware of the ATRLOCK program, so that they don't have to give away their secrets easily.

For specific details on configuration, be sure to see the sections on command-line parameters and report-file formats. You can also go to the website for more information (URL is at the top and bottom of this document).

Also be sure to view the ATRT.DOC file for information on using ATRT, the AT-Robots Tournament program.

TROUBLE-SHOOTING & SPEED-CONTROL:

The AT-Robots program is basically fully automatic and self-contained. It will probably either work or it won't. If there is a problem, most likely

(but not definitely) you are out of luck. Here are some possible causes of your difficulty:

1. Not enough memory.
2. Incompatable graphics adapter.
3. Damaged copy of AT-Robots.
4. Computer is not sufficiently IBM compatable.
5. You haven't followed any of the instructions.
6. Computer is damaged or not performing properly.
7. You're using too early a version of DOS.
   (suggest DOS 3.3 or higher)
8. Hardware/software conflict

If you HAVE gotten the program to run, but it's too slow, there are some things you can do to speed it up. First, You can try putting only 2 or 3 robots in the arena at once. You will find that the program slows down a lot when you put 5 or 6. Another thing you can do is put the /D0 setting in. That will turn off the delay that is placed in the program to keep it running at the proper speed on faster computers. Usually the delay won't need to be cranked down unless you're using an older computer (286 or 386).

LEGAL SHTUFFS:

This game was originally distributed as shareware. This has changed to "Freeware", which basically means that you are granted free license to make use of this software without paying for it, but the author retains all rights (including copyrights) to it, including the source code, executables, documentation, and associated files, as per the license displayed below. You are also encouraged to give copies of it to your friends, and upload it to bulletin boards.

- -

LICENSE & DISCLAIMER:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

All advertising materials mentioning features or use of this software must display the following acknowledgement:

> This product includes software developed by Ed T. Toton III & NecroBones Enterprises.

- -

REGISTRATION:

Registering is not required. In fact, registration codes are only a throw-back to the days when this was distributed as shareware instead of freeware. These days, registration codes will rarely be handed out, but here's what to do should you get one:

Once you register you will get a registration code that will disable the nag-messages. To use it, either enter the info using the ATROBS SHELL or enter it into a data file manually.

To do it manually, simply do the following from the DOS prompt:

```
COPY CON ATR2.REG<enter>
<reg name goes here> <enter>
<reg number goes here> <enter>
^Z
```

In the above, anything inside angle brackets indicates what to put. Also, ^Z means Ctrl-Z or F6.

Another way is to simply enter the two lines in a text editor and save it under the filename of ATR2.REG.

example:

```
John A. Smith
1234567
```

The capitalization COUNTS! It must appear precisely as it does on your registration response.

REVISION HISTORY:

All of the 2.00 versions were beta versions.
The first non-beta was version 2.01.

2.00a - In-house test version.

2.00b - First public release, beta test.

2.00c - INC and DEC instructions fixed (used to be backwards)

2.00d - Fixed a documentation error: Port 12 & 13
- General DOC updates, spelling fixes, & clarifications added
- New LOCK feature to encrypt/scramble robots for distribution.

2.00e - More Documentation clarifications
- 'S' key to toggle sound on/off now works.
- LOCK program now copies header of comments over to LOCKed copy before encoding.
- Heat generation/dissipation is now itemized in DOCs.
- Reverse throttle added
- You can now add comments after labels and compiler directives (on the same line).
- Find-Angle interrupt added.
- Find enemy ID interrupt added.
- Registration codes added.
- #MSG compiler directive added (message display was already implemented in version 2.00a, but there was no way to define the message).
- #TIME directive added.
- AT-Robots Shell now included.
- TABs are handled more effectively by the compiler now.

2.00f - New error code: Divide by 0. (It would have previously caused the GAME, not the robot, to have a runtime error).
- Shell is now complete :-) (was missing a file)

2.00g - Bug fixed with robots taking missile damage. (bug was introduced in 2.00e)

2.00h - JLE/JBE and JAE/JGE instructions now work (they used to see if something was less/greater than AND equal instead of OR, and would therefore never jump).
- SAL and SAR instructions added, for signed bit-shifts.
- NEG instruction added
- Heat-bar graphs are now only re-drawn when a visible change ocurrs. (for the sake of run-time efficiency)
- Robots now dissipate an extra point of heat every 8 game cycles if abs(throttle)<=25.
- Interrupt 9 (target info) added.
- Collision distance reduced (robots now have a radius of 4 meters instead of 6).

2.00i - "The Physics of AT-Robots" added to DOC file.

- RAM address 3 "accuracy", now works
- RAM addresses 0 and 1 were reversed in DOC file, now fixed.
- New GIF to replace old robot schematic.
- LOCK program renamed ATRLOCK to avoid confusion with
  LOCK command for locking files in Win95.
- More documentation clarifications and explanations.
- Program code can now be accessed (but not changed) by
  normal pointers ("@" codes).
- Constants added for interrupts 7 through 9.
- Range-checking on GET and PUT added.
- New label-addressing mode added.
- JTL instruction added.
- Jump/CALL range-checking improved.
- Find-angle interrupt sped up.
- New sample robots: SWEEPER, TRACON
- Scan "accuracy" fixed. Used to return -2,0,+2,
  but never -1 or +1.
- Minimum scan-arc reduced from 1 to 0.
- Interrupt 10 (Game-info) added.
- Interrupt 11 (Robot-info) added.
- Missiles now have a little more room along the walls
  (they don't dissappear within the viewport anymore)
- Number of wins is now displayed in robot status display.

2.00j - Compiler is even more accepting of TABs and other
  extraneous control-characters.
- Collision count (interrupts 12 and 13) added.
- Error #10 added (invalid interrupt call).
- Error #11 added (invalid port access).
- SDUCK now accesses the correct port to randomly
  aim its turret... ooh yay.
- RANDMAN has been slightly improved (so as not to get
  stuck on the walls as much)
- Minor adjustment to the missile blast algorithm.

2.00k - Minor adjustment made to the movement routine. A rounding
  error made it impossible to go at heading 192 (left)
  at a Y of 0 (north wall). This has been fixed.

2.00l - Minor internal re-arrangements to make the code a little
  more organized.
- Transponder functions added- Port 19.
- Robots can no longer start a battle inside of each
  other (I never actually saw this happen, but it was
  theoretically possible).
- Port 20, shutdown level added.
- Robots now take damage faster in the heat range
  of 475 to 499.
- Minor adjustment to the rounding in the scan accuracy returns.

2.01 - First NON-Beta release.
- ERROR now works as an alternate for ERR.
- Windoze setting (/W)
- Minor clean-ups to increase game stability.

2.02 - Adjusted Sniper to not intentionally generate errors.
- Robot Software-resets now reset the entire stack and RAM memory for the robot. It used to be that from one round to the next in tournaments, the values of the stack, registers, and collision count, etc were preserved.
- Robot communication system added. Also added two sample robots, RECEIVE.AT2 and TRANSMIT.AT2 to demonstrate it. Interrupts 14, 15, & 16 added, and port 21 added for communications.

2.03 - Kills/Deaths are now tracked and displayed.
- Display area slightly modified (especially in color) to accomodate the increasing crowding of displayed data.
- New report mode /R2 added, so that third-party tournament programs can track kills/deaths. /R report mode still works the same way it always did, for compatability with the existing ATRT program.
- Documentation fixes (some spelling errors corrected, as well as a few corrections to the sample code)
- Interrupts 17 & 18 added (clearcom, and kill/death count)
- Finally, that bug that screwed up the results in the first tournament has been found and stamped out. Robot's scanning with an arc of 0 with the simulation graphics off will now actually find their targets.
- Indirect addressing. Example: MOV AX,[BX]

2.04 - /R3 report mode added: Also gives info on robot armor & heat at the end of the last round (or at the time of last death, as the case may be). It also gives the shots-fired count, which is a total over all the matches played out.
- Trademarks & Copyrights section put back into documentation (I must have misplaced it somewhere along the way)
- Added more detail to the Introduction section of this doc.
- Robot hardware configurations added! This includes a shield and mine-layer, which use ports 22, 23, and 24. Configs are done with compiler directives. Robot TRAPPER.AT2 added to demonstrate the shield, minelayer, and config directives.
- New "meters" counter in memory. You can use this to keep track of distance travelled, perhaps for spacing mines. 'Meters' is a variable you can call by name. Int 19 resets it to 0. Writing to the variable doesn't do anything.
- 'W' can now be pressed during the simulation to toggle 'windoze'
- Backspace key can now be used to end a single match, without ending the entire sequence of matches.
- Fixed a bug in the scan-accuracy returns. It would sometimes return some very screwy data for large scan-arcs (20 and lower seemed to work well before). This quirk has been fixed.
- Error-logging for robots now available. /E parameter activates it. Logs will be stored as the same filename as the robot, except with a .ERR extension. May be useful for debugging robots.
- ATRT Tournament program is now included (it has also been slightly modified). See ATRT.DOC for details.
- All match-results now display shots-fired. The count for the

match is displayed for each match, and the total count over all the matches is displayed in the final results chart.

2.05 - Fixed a bug in ATRLOCK in which the first command or compiler directive in the robot to be locked would not be included in the locked copy.
- Automatic shutdown now turns off the shield, if it's on. TRAPPER.AT2 has been modified to adjust its shutdown level.
- Mine detonations now actually damage the robot that dropped them.

2.06 - ATR2 no longer displays the source code for locked robots during compile. (ooops!!) I'm not sure when this bug first appeared. (Well, it wasn't rediculously secure anyway, but still)
- Both ATR2 and ATRLOCK now use a slightly modified encryption so that robots locked from now on will not be viewable in older versions of ATR2. However, older locked robots will still be usable (it's downwardly compatible).
- ATRLOCK has been made more tolerant of Tabs and other control characters (turns them into spaces).

2.07 - Some miscellaneous clean-ups in the compiler and LOCK decoder.
- Shield has been severely castrated to restore game balance. This is supposed to be more of a sharp-shooting game, so the shield had to be weakened. Using it effectively will now be more of a challenge. :-)

2.08 - Fixed a bug that allowed you to get more than the maximum number of config points.
- Int 19 actually works now (man, that's embarrassing).
- /# parameter added to specify max robot program length.
- Changed to a variant of the BSD license, thus allowing the source code to be distributable.
- LOCK3 encryption added. The key does not reset on each line, and it propogates errors, so single character changes should ruin the decryption process.
- Memory location 13, last target's absolute speed (in cm/cycle)
- Added /@ switch to allow old-style shields.
- Documented the previously undocumented /! parameter (insane missiles).
- Fixed some confusion in TRAPPER.AT2, improving it slightly.
- Added report file formats to the docs.
- /R4 report mode added. (now documented)
- Fixed a huge security hole via range-checking.
- Fixed documentation for port 11, throttle setting.
- /W parameter (and "W" key too) Windoze setting has been extended to also affect the display of intermediate battle results in text mode (and no longer just in graphics mode).
- Added SNIPER2.AT2 and WALLBOMB.AT2 sample robots.
- #msg now allows use of punctuation and lowercase.
- Some ATRT enhancements.
- Lowered the max cycle limit and max number of matches to 100,000 each (to help reduce the chances of variable overflows).

2.09 - ATRT updated to be more compatibile with /# parameter.
- Maximum number of robots increased to 32. The status display now automatically condenses to fit more robots in when necessary.
- "Q" key has been changed to toggle sound along with "S", rather than quit. ESCAPE can still be used to quit, however. This was done so that the keyboard commands would more closely match the command-line parameters.
- Default time limit changed to 100,000 cycles (instead of defaulting to no limit at all).
- Status area now displays amount of left-over DOS memory in bytes, so you can see how close to the limit you are.
- "Chaos" batch file and config file added to demonstrate 32-robot battles using the sample robots. Just run "Chaos" to see.
- Fixed an array-overflow problem with mines.
- Increased number of mines. Updated WALLBOMB.AT2 and TRAPPER.AT2 to accomodate.

2.10 - ATRT enhancements, particularly to facilitate unattended runs.
- You can now enter pre-compiled machine code into your programs.
- Simulation now checks for invalid microcode, and will not execute instructions that are thus invalid (invalid instructions as such will use up 1 cpu cycle when skipped, regardless of what the instruction was).
- Source code re-arranged slightly to overcome code-segment limitations.
- Debugger added (see command-line parameter and keyboard command sections for details). Thanks to FiFi LaRoo for making the debugger.
- Fixed a bug in SAR and SAL commands that made them more or less ignore the second operand.

Well, that's all! Have fun!!!

For current CONTACT INFORMATION see: NBONES.DOC

-Ed T. Toton III
"Bones"
NecroBones Software.
