# 'bot 1.0

**General Software Information:**

## Copyright:

The included software programs, 'bot Developer and 'bot Arena (the "Applications"), and their Documentation (including Tutorials) are Copyright ©1991 by Jason A. Davis and David H. Chait and are protected by the US Copyright laws. You may not decompile, disassemble, or create derivative works from the Applications.

## Requirements:

'bot requires System 6.0 or greater on a Classic, SE, or Plus, and requires System 6.0.5 or greater on all other machines. It is fully compatible with MultiFinder and System 7.0, which allow battles and Tournaments to be run in the background. Even under MultiFinder, it can be run on a Mac with only 1 Meg of memory.

## Shareware Information:

These Applications are distributed under the Shareware system. Shareware is a user-supported distribution method: users can try out software and pay after deciding they like it and use it. You may use 'bot for thirty (30) days without charge, after which we request that you send us the registration fee. If you do not register, we ask that you delete all copies of the Applications at the end of the thirty-day period. Registered users will be notified of significant upgrades and any upcoming tournaments. All minor upgrades will be free of charge to registered users.

The software residing in the **'bot Folder**, which includes the Tutorial folder, the Applications, and this Documentation, may be distributed by user groups providing they are charging only a nominal fee for their expenses. In no case may users be charged for the 'bot software, except of course for sending the normal registration fees to us. All Public Domain and Shareware distribution companies/channels and all other for-profit corporations wishing to distribute 'bot must FIRST contact Future Generation Software at the address listed below.

## Registration Fee:

To become a registered user, send $15 (+5% for MA residents) to:
Future Generation Software
10 Thoreau Road
Lexington, MA 02173

Please only send a check, and make it payable to Jason Davis. Include your name, mailing address, electronic mail address (if any), and the version number of each 'bot Application.

## Group Registration:

As an alternative (and to make the cost for you lower), we will register a group of 2-5 users together for a single fee of $20 (+5% for MA residents). Please send a check as above along with the information requested for each user in the group.

## General Disclaimer:

Although we have taken great efforts to insure that these Applications are bug free, we cannot guarantee that they are without error. Future Generation Software (Jason Davis & David Chait) takes no responsibility for any incidental or consequential damages resultiing from the use of these Applications. In no case will our liability exceed the registration fee paid to Future Generation Software.

## Contacting Us:

We welcome all comments and suggestions for improvement. You can reach us at any of the following:

America Online: FUGESoft
CompuServe: 71121,2506
Internet: jdavis@husc3.harvard.edu
Mail Address:

Future Generation Software
10 Thoreau Road
Lexington, MA 02173

## Key to the Documentation:

All code examples are printed in Monaco. All other text is either Geneva or New York. In code examples, braces ("{" and "}") around a piece of an example is used to show that a parameter to a command is OPTIONAL, not required.

## Welcome to 'bot

As a bot programmer, it is your task to design and build winning bots, robotic gladiators that are placed in an Arena for a battle to the death. Using a careful combination of offensive and defensive hardware and software, you must build a bot that can survive the ordeals of combat in the Arena.

Thus, your objective in designing a bot is to make sure that the bot can seek out other bots in the Arena and destroy them before it is itself destroyed. Of course, the other bots in the arena will be trying to destroy your bot, so you must be clever and strategic while building and programming.

'bot comes in two pieces: the 'bot Developer module, where you design and equip your bots, and the 'bot Arena module, where you pit your bots against others. Both of these applications are completely MultiFinder compatible. For example, you can load bots into the Arena by double-clicking on them in the Finder, even when Arena application is already open. You can also run Arena battles or tournaments in the background under MultiFinder — when the combat is over, a small robot icon will flash over the apple icon on the left of the menu bar to let you know that the combat has ended. 'bot is also System 7.0 compatible.

If you are running 'bot Arena under MultiFinder or System 7.0, and you want to run a combat with a number of very large bots, you may get an "out of memory" error. If this happens, you should increase the size of the application's partition (the box in the lower right corner of the Get Info window) to 512K. Under normal operation, or under System 6 Finder, there should be no need for this change.

## The 'bot Arena Module

### Arena Overview

All combat takes place in the Arena, which is a 272x272 square. A bot itself is circular, with a radius of 8 units. Thus the logical size of the arena to a bot is 256x256 units square. The position (0,0) is the upper-left corner of the arena, while the position (255,255) is the lower-right corner. Angles in the arena are defined in a clockwise direction, with 0° pointing right, 90° pointing down, 180° pointing left, and 270° pointing up. Time in the Arena is defined in units known as 'ticks', with each bot executing 16 'Instruction Units' per tick. Different instructions for your bot will take different numbers of instruction units to be executed; adding A and B is simple and can be done fairly quickly, but scanning a 10° arc of the Arena can take quite a bit of time.

When you launch the Arena module, you will see the Arena window on the left side of the screen and the Watcher window on the right.  The Arena will display the battle from a bird's-eye view as it progresses, while the Watcher will show specific information about each of the bots in the Arena and other Arena information.   If you wish to move the Watcher window, you can move it like a normal window; if you wish to move the Arena window, hold down the Option key and click anywhere on the window, then drag it to wherever you want it.

Generally speaking, you can type Command-period to cancel any action in progress (such as a running battle) and to cancel any dialog or alert window.  If a dialog window is visible, hitting the Return key is the same as clicking on the dialog's highlighted button (usually the "Okay" button).

## Combat Details:

During a battle, the loaded bots (those visible in the Watcher Window) are randomly placed in the Arena; they are placed so that no two bots are too near each other, however.  Time then passes one clock tick at a time, with all bots moving and computing simultaneously, executing their individual programs.  A bot is destroyed when its CPU has 0 durability left.  A combat ends either when there is only one bot left alive, when no bot has taken any damage in the last 5000 clock ticks, or when a total of 25000 ticks have passed.  In the latter two cases the battle is declared a draw.

If a bot collides with one of the Arena walls it stops dead.  If two bots collide, they collide semi-elastically (they "bounce" off of one another).  In any collision, the involved bot(s) take damage proportional to the square of their velocities at the time of the collision.  Special note:  Under certain circumstances, two bots can lock their treads together, and neither will be able to move at all — the treads are destroyed.  The first case where this can happen is when two bots are touching each other, and a third bot collides with one of them.  There is a small chance that two of the involved bots will be crushed together in this collision and lock treads.  The second case is when one bot is touching an Arena wall, and a second bot collides with the first bot and the wall simultaneously (effectively wedging itself behind the first bot).  In both of these circumstances, the involved bots will be absolutely unable to move for the remainder of the battle.  The odds of these occurrences are very small, but you should be wary when writing your bots — try to intelligently move away from any bots that are colliding with you.  If you want to see what this looks like, you may be able to see bots lock treads if you run 5 Suicides (a bot provided in the Tutorial) simultaneously.

There are a number of different weapons that damage bots in varying ways.  The weapons and their descriptions (along with all other information

about what makes up a bot) can be found in the manual section on the 'bot Developer application, specifically, the section dealing with bot Architecture.

## The Menus:

### File

```
File
   Open 'bot...      ⌘O
   Reload 'bot       ⌘R
   Duplicate 'bot    ⌘D
   Close 'bot        ⌘W
   Empty Arena       ⌘E
   ..........................................
   Preferences...    ⌘P
   Tournament...     ⌘T
   ..........................................
   Quit              ⌘Q
```

**Open 'bot…**

This will present you with a standard open dialog and allow you to select assembled ".bot" files to be loaded into the Arena.  When a bot is selected and opened, you will notice that it is being added to the Watcher window in the background.  After 5 bots are in the Arena, the dialog will automatically close.  Should you want to stop loading bots at any time, simply press the Done button or type Command-Period.

**Reload 'bot**

This menu item is active only when a bot is selected in the Watcher window.  The bot currently selected will be reloaded from the disk.  This is useful when you have been running a combat in the background while modifying one of the combatants in Developer.  After recompiling the bot, you can Reload it into Arena before starting another battle.  The next version of 'bot will be able to automatically Reload bots for you, but this current feature should allow you to switch between the two 'bot applications without too much trouble.

**Duplicate 'bot**

This menu item is active only when a bot is selected in the Watcher window and less than five bots are in the Arena.  The bot currently selected is cloned and added into one of the remaining slots in the Arena.
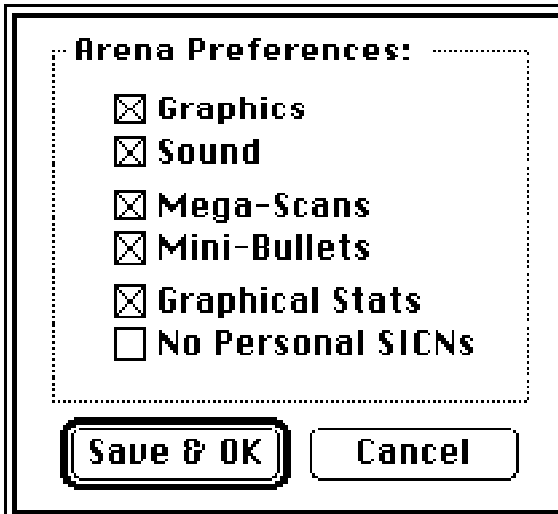
**Close 'bot**

This menu item is active only when a bot is selected in the Watcher window. The bot currently selected is removed from the Arena, and other bots will shift upward in the Watcher window to fill in a gap.

**Empty Arena**

This menu item is active only when there are bots in the Arena. When chosen, all bots in the Arena are removed as if each had been selected individually and closed.

**Preferences…**

This menu item will bring up the Arena Preferences dialog that follows:

```
┌─────────────────────────────────┐
│  ┌ Arena Preferences: ·········· │
│  ⊠ Graphics                      │
│  ⊠ Sound                         │
│  ⊠ Mega-Scans                    │
│  ⊠ Mini-Bullets                  │
│  ⊠ Graphical Stats               │
│  ☐ No Personal SICNs             │
│  ·······························  │
│  ╔════════════╗  ┌────────────┐  │
│  ║ Save & OK  ║  │  Cancel    │  │
│  ╚════════════╝  └────────────┘  │
└─────────────────────────────────┘
```

If you change a Preference here, the 'world' setting will be changed to the same on/off value. Preferences left untouched will not affect their matching 'world' settings, even if they differ. When you next launch the application, all settings will be as last set in the Arena Preferences dialog (e.g. if Sound is off in the Preferences, but you turn it on in the world, it will still be off the next time you launch the application). See the individual world settings menu commands in the Arena menu below for details of the individual items.

The one item that **only** appears in the Arena Preferences dialog, and not in the Arena menu, is the "No Personal SICNs" option. Users can have personalized icons for their bots to be displayed in the Arena, as detailed in the Developer Overview. However, there are many cases where you might not want your personal icons to be displayed. For instance, if you want to run a bot against itself, you'd want to turn on the "No Personal SICNs" option before loading in the bot. This way, the default SICNs will be used, and the two copies of your bot can be easily distinguished.

**Tournament…**

This menu item brings up the Tournament dialog, described in full detail in the Tournament section below.

**Quit**

This menu item is used to leave the bot Arena module.  If a Tournament is running, you will be asked to confirm the cancelling of the Tournament before the application will exit.

## Edit

The edit menu is not used in the Arena, but is included for standard support of Desk Accessories.

## Arena

```
Arena
  Start Battle      ⌘/
  Stop Battle       ⌘.
  Pause Battle      ⌘,
................................
✓ Graphics          ⌘G
✓ Sound             ⌘S
................................
  Mega-Scans        ⌘N
✓ Mini-Bullets      ⌘B
................................
✓ Graphical Stats   ⌘F
```

**Start Battle**

Begins a new battle between the bots currently in the Arena.

**Stop Battle**

Halts the battle in progress in the Arena.  If a Tournament is running, you will be asked to verify that you wish to cancel the Tournament.

**Pause Battle**

Pauses the battle in progress in the Arena.  To continue, select this again.

**Graphics**

Toggles the Arena display window on/off.  May be set permanently in the Arena Preferences dialog.

**Sound**

Toggles the battle sounds on/off.   May be set permanently in the Arena Preferences dialog.

**Mega-Scans**

Toggles display of scanner 'blips' between normal (small) and full-arc (huge, takes longer to display, but allows you to see what your bot "sees").  May be set permanently in the Arena Preferences dialog.

**Mini-Bullets**

Toggles display of bullets between normal (2x2 squares) and mini (1 pixel, displays **much** faster).  May be set permanently in the Arena Preferences dialog.

**Graphical Stats**

Toggles the Watcher window between Graphical Stats display and Numeric Stats. See the Watcher Window section below for more detail.  May be set permanently in the Arena Preferences dialog.
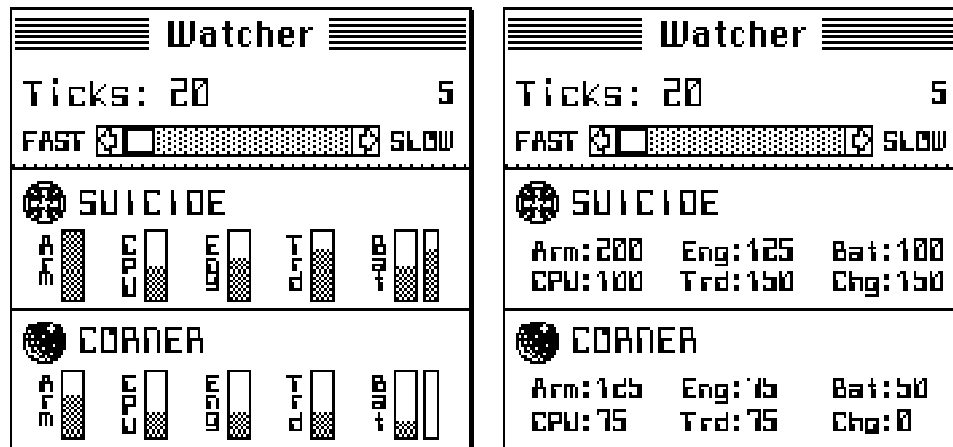

## The Arena Window:

The Arena is where you actually see the ongoing Battle.  There are a number of graphical effects in the Arena while the battle runs.  First, each bot is displayed by one of 5 small iconic representations stored in the application, and is animated as it moves around the Arena.  Also, if the bot has a Shield, the icon will change somewhat when the shield is turned on.  Second, fired bullets will fly across the Arena at their set speeds, either colliding with a bot or harmlessly hitting a wall.  Third, a bot that has a laser will be shown firing it straight out as a two-pixel wide line from the edge of the bot. The beam will damage the first bot that it hits, or, if it strikes an Arena wall, it will dissipate harmlessly.  Fourth, a grenade or explosive bullet can cause an explosion.  An explosion is drawn with the actual radius of the blast, damaging any (and every) bot it touches (see the figure below).  The blast disappears within a tick of the explosion. Fifth, a bot that is destroyed will disappear in a quick explosion.
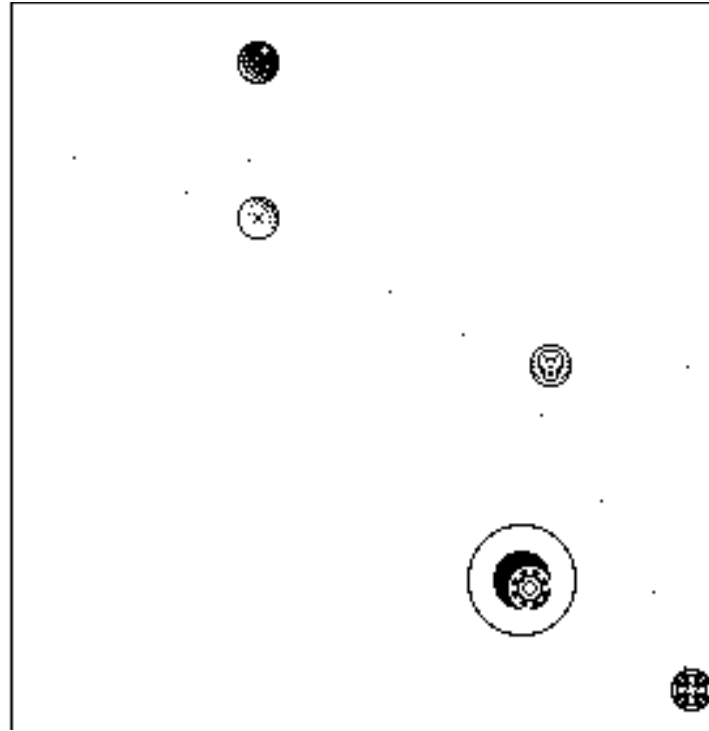
### The Watcher Window:

All information on a given bot in the Arena is displayed in the Watcher window to the right of the Arena.  The Watcher displays information on all bots in the Arena.  The small-icon representing the bot in the Arena is displayed to the left of the bot's name in its Watcher slot.  Below the name you will see current statistics of the bot, updated as needed during a battle.  There are two different statistic displays, Graphical Stats and Numeric Stats.  The Graphical Stats display (see left figure below) consists of a bar for each of the bot's vital statistics.  Each bar shows a range of 0 to 200 for that statistic.  The Numeric Stats display (see right figure below) shows the same information using the actual numbers for the statistics.  A blank space in this display indicates that the blanked component has been destroyed.  These displays show all vital information on a given bot: Armor, CPU, Engine, Tread, and Battery durability, plus the current Battery Charge (in the Graphical Stats display, the smaller bar on the extreme right).

```
┌═══════ Watcher ═══════┐   ┌═══════ Watcher ═══════┐
│ Ticks: 20           5 │   │ Ticks: 20           5 │
│ FAST [▯▯░░░░░░░] SLOW  │   │ FAST [▯▯░░░░░░░] SLOW  │
├───────────────────────┤   ├───────────────────────┤
│ ⊕ SUICIDE             │   │ ⊕ SUICIDE             │
│  A  C  E  T  B        │   │  Arm:200  Eng:125  Bat:100 │
│  r  P  n  r  a        │   │  CPU:100  Trd:150  Chg:150 │
│  m  U  g  d  t        │   │                       │
├───────────────────────┤   ├───────────────────────┤
│ ⊗ CORNER              │   │ ⊗ CORNER              │
│  A  C  E  T  B        │   │  Arm:125  Eng:75  Bat:50 │
│  r  P  n  r  a        │   │  CPU:75   Trd:75  Chg:0 │
│  m  U  g  d  t        │   │                       │
└───────────────────────┘   └───────────────────────┘
```

You may click on one of the bots in the Watcher window, and it will be selected and inverted.  You may also select a slot in the Watcher by using the Up Arrow and Down Arrow keys when the Watcher window is the frontmost window.  There are a number of operations you can perform on a selected bot (see the section on Menus above).  First, if you type Command-R, that bot will be reloaded.  Second, if you type Command-D, that bot will be duplicated.  Lastly, if you type Command-W or simply the Delete key, that bot will be closed and removed from the Arena.

The top section of the Watcher window handles other information on the Arena. “Ticks” indicates the number of clock ticks that have gone by in the current battle.  The small number to the right of the ticks display is Tournament information, and is thus only shown if a Tournament is running.  It is the number of battles left to run in the current tournament, not including the current battle.  If the tournament is single-elimination, this number may be an over-estimation.  This is because, for example, a best-of-three series between two bots will terminate after only two battles if the same bot wins both.

The scrollbar is the Arena Speed control, and is useful for carefully examining certain bots in a running battle.  Moving the scrollbar towards slow (by clicking on the right arrow, the space to the right of the thumb, or dragging the thumb to the right and releasing) will slow the Arena down to approximately 1/2 speed, 1/3 speed, 1/4 speed, etc., all the way down to about 1/10 speed.  (For those interested, what it actually does is waits from 1/60th of a second to 1/6th of a second before it runs a tick of the battle.)
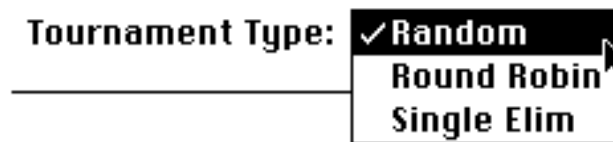
## Tournaments:

One of ’bot’s most powerful features is its ability to run large tournaments.  To set up and run a Tournament, select “Tournament…” from the File menu.  You will be presented with the following dialog:
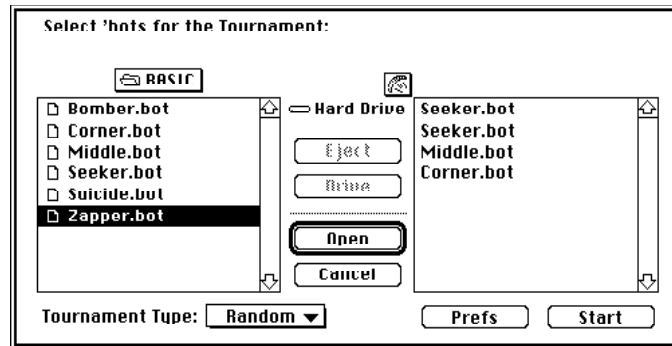
In the above example, we have already added 4 bots to the Tournament List.  This is done by finding and double-clicking on a ".bot" file in the file list on the left side, just like opening any file with a standard open dialog.  The bot will then be added to the Tournament List on the right side, and will be selected and highlighted.  A bot may also be selected in the Tournament List simply by clicking on its name.  A bot that is selected may be removed from the Tournament List by pressing the delete key.  You can also just double-click a bot in the Tournament List to remove it from the list.

To begin the selected tournament, click on the Start button or hit Command-/.  To exit the tournament dialog without running a tournament, click the Cancel button or hit Command-period.  You can access the Tournament Preferences dialog by clicking the Prefs button or by hitting Command-P.  You can also type command 1, 2, and 3 to select the various types of tournament if you do not wish to use the popup menu provided.

There are three types of Tournaments that you can run.  The first is a Random Tournament, where each battle is a random selection of from 2 to 5 of the bots in the Tournament.   The second is a Round Robin Tournament, where every possible combination of bots (2, 3, 4, or 5 at a time) is run in the Arena.  The last is a Single Elimination Tournament, in which all bots in the tournament are first run through (optional) preliminary seeding battles to determine who will make it into the elimination rounds (which start with octafinals: the top sixteen seeded bots).  Once seeded, sets of one-on-one elimination battles — a standard single elimination tournament — are run for octafinals, quarterfinals, semifinals, and finals to determine the winning bot.  The type of Tournament that will be run is chosen from the "Tournament Type" popup menu, as shown below:



To modify the settings for the tournament that you wish to run, click on the Preferences button.  You will be presented with the following dialog:

```
Tournament Preferences:
Random:   # 'bots/Round:      [ 5 ▾ ]
          # Prelim Rounds:  [◁|■▭░░░░░|▷]  10
          # Final Rounds:   [◁|■▭░░░░░|▷]   5

Round     # 'bots/Set:        [ 3 ▾ ]
Robin:    # Rounds/Set:     [◁|■▭░░░░░|▷]   5

Single    # Prelim Rounds:  [◁|■░▭░░░░|▷]  20
Elim:     Elim Rounds/Pair:[◁|■▭░░░░░|▷]   3

ALL:      Max Points/'bot:  [◁|■▭░░░░░|▷] [900]

              ((Save & OK))  ( Cancel )
```

The Tournament Preferences dialog lets you modify specific settings for each type of Tournament.  Note that a "Round" is one battle between from 2 to 5 bots.

For the Random Tournament, you can set the number of bots in a Round, the number of randomly chosen Preliminary Rounds, and the number of Final Rounds.  If the number of Final Rounds is not zero (0), then the best n bots as determined by the Preliminary Rounds (where n is the number of bots per Round) are run against each other to determine the "true" winner of the tournament.  If the number of Final Rounds is zero, then the bots are ranked solely based on their performance in the Preliminary Rounds.

For the Round Robin Tournament, you can set the number of bots in a given Set of bots, and the number of Rounds that each Set is used.  The tournament runs until a Set has been run using every possible combination of bots in the tournament.

For the Single Elimination Tournament, you can set the number of Preliminary Rounds that decide seeding for the Elimination Rounds, and the number of Elimination Rounds per pair of bots to determine the victor of the pair.  Note that in determining the victor of a pair, the computer will halt the contest between two bots as soon as it can declare one bot the winner.  For example, if you set 5 Rounds per pair, and one bot wins the first 3 Rounds in a row, that bot is declared the victor of the contest immediately — the final 2 Rounds are not run at all.

This dialog also lets you set a maximum point value for bots entered in a tournament.  You will not be allowed to add any bot to the Tournament List whose point value is greater than this defined value.  If you have already added bots to the Tournament List and you lower this maximum point value, all bots whose values are greater than this new maximum are removed from the List.  See below for more information on bots' point costs.  If the tournament point value is set to zero (0), there is no limit on the costs of entered bots.  This feature makes it easier for you to set up a tournament for, say, "800-point bots and under," or "1200-point bots and under":  You can design bots for any point limit you choose.

In a tournament, a bot earns one victory point for every enemy bot that it outlasts in a battle.  Thus, if there are 5 bots in a battle, the last bot alive earns 4 points (for outlasting the other 4 bots), the second place bot, 3 points, and so on.  The first bot to die gets no points.  If a draw is declared, none of the bots involved in the draw get any points for outlasting other bots in the draw.  For example, if there is a tie between 2 bots in a 3-bot battle, the destroyed bot will have earned no points, and the two tying bots will have earned 1 point each.  The final score listed for a bot in the tournament results chart is the average number of points earned by that bot during the tournament.

# The 'bot Developer Module

## Developer Overview

The Developer is where you actually build and program your bots. All bot programming is done in the Code Editor, a text-editor environment. Programming may be done in either 'bot Basic or 'bot Assembly. Both languages use the same constructs for defining constants and variables; only the actual bot code itself is language-dependent. We suggest that users intent on eventually showing/distributing their bot code to others use Basic, because it is much easier to read and understand. In the end, bot code generated by the Basic compiler is roughly equivalent in speed to code that can be programmed in Assembly. Thus, even experienced programmers should feel comfortable using 'bot Basic. However, because we know that there are hackers out there who will want complete control over their bots, we also allow development in 'bot Assembly. A Tutorial for the 'bot Basic language is provided with this manual. It includes 6 bots and explanations of what they do. Assembly language equivalents for these Basic bots are also provided for those who are interested.

All architectural modifications to your bots are made in the Architecture Editor. You can define weapons, armoring and durability, defenses, and other physical characteristics of your bots there. Note that every feature added to a bot increases the bot's point-cost. Since tournaments have a set maximum point value for the bots entered in them (discussed above), you should pay close attention to your bots' total point-costs.

The one modification that cannot be made in the Architecture Editor is customizing your bots' icons. If you wish to create your own icons, use any third-party icon editor or ResEdit to create "SICN" resources. Open your bot's code file (the one with the program in it, not the ".bot" file), and save the SICN resources there. Save your bot's normal icon as ID#400, and its shield icon (if you wish to have a special icon for when the bot's shield is on), as ID#401. 'bot will take care of the rest.

There are many shortcuts available within the Developer. Generally speaking, you can type Command-period to cancel any dialog or alert window. If a dialog window is visible, hitting the Return key is the same as clicking on the dialog's highlighted button (usually the "Okay" button). Within any of the Architecture Editor dialogs, you can type Command-X to select that button or box, where 'X' is the first letter of the button's name.


## The Menus

# File

```
┌─────────────────┐
│ ▐▀▀▀▀▌          │
│ ▐File▌          │
│ ▐▄▄▄▄▌          │
│ New         ⌘N  │
│ Open...     ⌘O  │
│ Close       ⌘W  │
│ ┄┄┄┄┄┄┄┄┄┄┄┄┄┄  │
│ Save        ⌘S  │
│ Save As...      │
│ Revert          │
│                 │
│ Preferences...  │
│ ┄┄┄┄┄┄┄┄┄┄┄┄┄┄  │
│ Page Setup...   │
│ Print...    ⌘P  │
│                 │
│ Quit        ⌘Q  │
└─────────────────┘
```

**New**

Opens a brand new bot code editing window.

**Open…**

Lets you open a bot code file and Architecture settings from disk.  Note that bot code files are simply standard text files with some added resources for Architecture settings, and any text file less than 32K may be opened.  Thus, you may use any text editor you like.

**Close**

Closes the currently open bot code file and Architecture settings.  If any changes have been made since the last time you saved, the application will ask if you wish to save them.

**Save**

Updates the bot code file and Architecture settings on disk.  If the file hasn't yet been named, the application will prompt for a name under which to save the file, like Save As below.
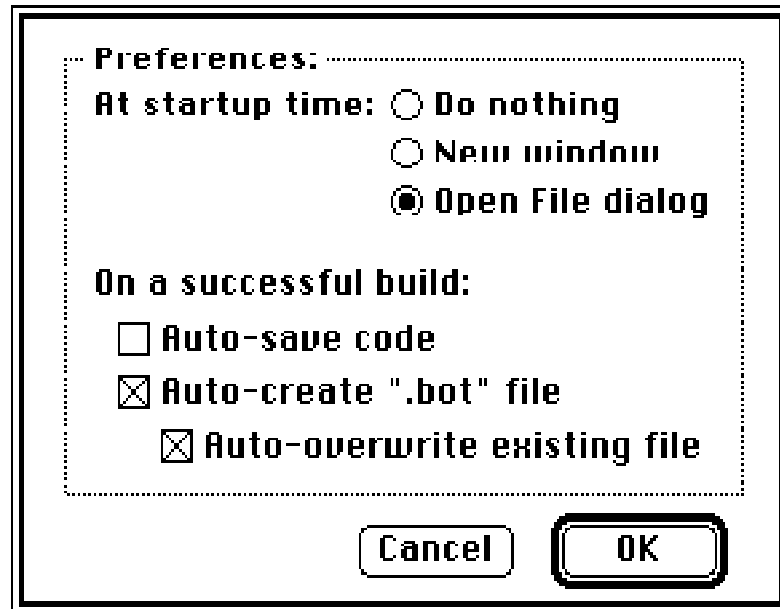
**Save As…**

Lets you save the open bot code file and Architecture settings to disk under a new name, or define the name of an untitled document.

**Revert**

Wipes out all changes since your last save by reloading the bot code file and Architecture settings from disk again.

**Preferences…**

This menu item will bring up the Developer Preferences dialog that follows:

```
┌─────────────────────────────────────────┐
│ ┌ Preferences: ·························· │
│ : At startup time: ○ Do nothing        : │
│ :                  ○ New window        : │
│ :                  ◉ Open File dialog  : │
│ :                                      : │
│ : On a successful build:               : │
│ :   ☐ Auto-save code                   : │
│ :   ☒ Auto-create ".bot" file          : │
│ :      ☒ Auto-overwrite existing file  : │
│ ·······································  │
│                                          │
│                    ( Cancel )  [[  OK  ]] │
└─────────────────────────────────────────┘
```

There are three possible choices for what will happen as soon as 'bot Developer is run.  It can do nothing, open a new (Untitled) bot, or present you with the Open File dialog (as if you had chosen Open from the "File" menu).  The "At startup time" radio buttons control which of these will occur.

"Auto-save code" means to automatically save the source code file (as if you'd typed Command-S) whenever the code is successfully assembled (Command-K).  "Auto-create '.bot' file" means to try and save the assembled bot as "botname.bot" where "botname" is the name of the currently loaded bot.  If a file with that name already exists, you will be asked whether you want to overwrite the existing file.  However, if "Auto-overwrite existing file" is selected, you will not be prompted — the new file will automatically replace the old one.  If "Auto-create" is not selected, you will be presented with a Save File dialog every time you successfully Assemble your bot.

**Page Setup…**
Standard Page Setup dialog box.

**Print…**
Standard Print dialog box.

**Quit**
Quits the Developer module.  If there have been any changes to the bot code or Architecture settings, the application will ask you if you wish to save them before exiting.

## Edit

```
┌─────────────────────┐
│ Edit                │
├─────────────────────┤
│ Undo          ⌘Z    │
│ ................... │
│ Cut           ⌘H    │
│ Copy          ⌘C    │
│ Paste         ⌘V    │
│ Clear               │
│ ................... │
│ Select All    ⌘A    │
│ ................... │
│ Find...       ⌘F    │
│ Find Again    ⌘D    │
│ Grab Find Text ⌘G   │
│                     │
│ Replace       ⌘R    │
│ Replace Every ⌘E    │
│ Grab Replace Text ⌘T│
└─────────────────────┘
```

**Undo**
Undoes the last editing action taken.  Only affects the editing window.

**Cut**
Selected text is copied to the Clipboard and then removed from the editing window.

**Copy**
Selected text is copied to the Clipboard.

**Paste**
Text stored in the Clipboard is inserted into the editing window at the insertion point.  If text is selected when Paste is executed, the selected text is deleted and replaced by the Clipboard text.

**Clear**
Selected text is deleted from the editing window.

**Select All**
Selects all text in the entire document.

**Find…**
Brings up a dialog (below) allowing text search of the open document from the insertion point forward.

**Find Again**
Continues searching from the insertion point or end of the selected text for the next instance of the same word.

**Grab Find Text**

Copies the currently selected text into the Find buffer, as if that text had been typed in at the Find dialog.

**Replace**

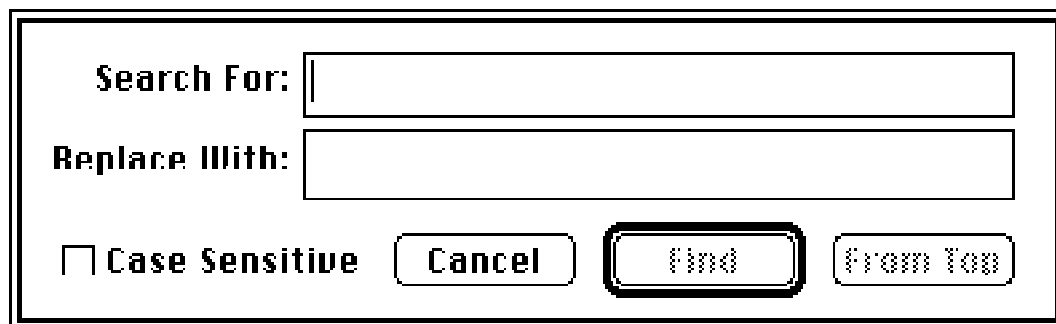Replaces the selected text with the contents of the Replace buffer.

**Replace Every**

Replaces every occurrence of the text in the Find buffer with the text in the Replace buffer.  Starts at the top of the file.

**Grab Replace Text**

Copies the currently selected text into the Replace buffer, as if that text had been typed in at the Find dialog.

The following is the Find dialog:



The text to be searched for and the (optional) text to replace that text with can both be entered here.  If the Case Sensitive box is checked, only exact text matched will be found; if not, the search ignores the difference between upper-case and lower-case letters.  From Top begins the search from the top of the file, rather than from the insertion point.

## Build



**Check Syntax**

This menu item instructs the Developer to parse the open bot code file.  If an error is found in the file, an error alert will be displayed that details the syntax error. Otherwise, the Build menu will dehilite when the Developer finishes parsing with no problems found.

### Assemble 'bot

This menu item instructs the Developer to check the syntax of the open bot code file, like Check Syntax above, but if no errors are found it will prompt you for a file name under which to save your fully Assembled bot. After supplying a name, or leaving the name already there, the Developer will generate a ".bot" file that may be loaded into the Arena, ready for battle.

## Window

```
Window
  Architecture Editor   ⌘H
  Zoom Code Window      ⌘/
```

### Architecture Editor

This menu item brings up the main Architecture Editor dialog. This dialog is described below.

### Zoom Code Window

This menu item is the same as clicking in the Zoom-Box of the editing window; it will zoom the window between its last set size and the largest possible size the window can be made.

## The Architecture Editor

The Architecture Editor can be brought up by pressing Command-H; the main architecture dialog will be presented. From this dialog you may edit the different characteristics of your bot and see your bot's overall point-cost. By pressing the various buttons in the main dialog, sub-dialogs will be brought up for editing particular sets of architectural characteristics. Note that if you make changes in a sub-dialog and decide that you don't want them, you may press the Cancel button to undo those changes and return to the main dialog. However, once you press the OK button in a sub-dialog, the changes you make become permanent, and you must manually change them back, or Revert the file, if you later decide against the changes. To leave the Architecture dialog, simply press the Okay button.

The following sections contain descriptions of the various pieces that make up your bot's architecture, but, in general, the best way to learn the architecture choices is to simply go and play with it.

## Weapons:

A given bot may have two distinct weapons, found under the "**Offense**" section of the Architecture Editor. There are three weapon types:

### Projectile:

Fires a bullet at a specified angle.  The bullet has damage value, speed, etc, depending on the definitions of the particular weapon (see below).  The bullet will strike and do damage to the first thing that it encounters, be it a bot or an arena wall.  The weapon takes a certain amount of time to reload between shots.  Relevant characteristics:

Damage/Projectile: The number of damage points inflicted upon an enemy bot hit by a bullet from this weapon.

Explosive:  On impact, bullet explodes like a grenade (see below for an explanation of grenade explosions).

# of Projectiles:  The number of bullets held in the weapon's magazine.  When all of the bullets have been used up, the weapon will no longer fire.  You may choose to have unlimited bullets.

# of Ticks to Reload:  The amount of time it takes for the weapon to reset itself after firing a shot.  If the bot attempts to fire this weapon again before this time has elapsed, the weapon will not fire.

Velocity:  The speed of a fired bullet, in pixels per tick.

**Grenade:**
Flings a grenade at a specified angle, targeted to land at a specified distance.  The grenade will not hit any robots — it goes over them — and will stop once it has travelled the specified distance.  After a pre-defined time, the grenade will explode, damaging all robots within its blast radius.  If the grenade is an impact grenade (see "Ticks Until Detonation," below), the grenade will explode upon reaching its target distance, or upon striking a wall.  If an affected bot's internal components are exposed (i.e. by a lack of armor), **all** of the bot's components will take damage. Note that a bot can be hurt by its own grenade.  The weapon takes a certain amount of time to reload, and it is generally slower to reload a grenade than a bullet.  Relevant characteristics:

Damage/Grenade: The number of damage points inflicted upon an enemy bot that is caught in the grenade's blast.

Blast Radius:  The radius of the grenade's explosion.  All bots intersecting or within the blast radius at the time of the explosion take damage.

# of Grenades:  The number of grenades held in the weapon's magazine.  When all of the grenades have been used up, the weapon will no longer fire.  Note that the point costs for Damage/Projectile and Blast Radius (above), as well as for selecting Impact Grenades (below) are the point costs **per grenade**, so the number of grenades can drastically affect the weapon's cost.

# of Ticks to Reload:  The amount of time it takes for the weapon to reset itself after launching a grenade.  If the bot attempts to fire

this same weapon again before this time has elapsed, the weapon will not fire.

Maximum Distance:  The maximum possible distance to which the weapon can fling a grenade.  If the bot attempts to fire at a distance greater than this maximum, the grenade will fall to the ground at this maximum distance.

Velocity:  The speed of a launched grenade, in pixels per tick.

Ticks Until Detonation:  The number of ticks between the launching of the grenade and its explosion.  There is no point cost associated with this characteristic: you may set any delay time you like without cost.  However, you may also select "impact" grenades, which explode upon reaching the specified distance (or upon striking a wall).  Impact grenades add a small point cost per grenade.

**Energy:**

This weapon fires a laser at the specified angle.  The beam hits instantaneously when it is fired — there is no "projectile velocity."  The first bot it hits takes damage as follows:  If the bot has armor, some of the beam's energy is scattered based on the armor's reflectance value and some of the remaining energy is then absorbed by the armor — the more armor a bot has, the more energy is absorbed before the beam cuts through to the inside of the bot.  One-quarter of the energy absorbed by the armor damages the armor.  All remaining energy damages an internal component of the bot.  Note that shields also absorb energy from energy weapons just as they absorb the damage done by other types of weapons.

An energy weapon never runs out of shots, but each shot drains the bot's battery.  If the battery's charge is not sufficient, the weapon simply will not fire.  When fired, the weapon takes a certain amount of time to charge before the laser is generated, but it can be fired multiple times in immediate succession without having to wait to "reload."  While the weapon is charging, the CPU is locked — no instructions are executed.

Energy Consumption:  The amount of energy drained from the battery every time the weapon is fired.  If the weapon requires more energy than is stored in the battery at firing time, it won't fire.

Energy Beam Focus:  A Wide Focus beam does damage equal to half of the weapon's Energy Consumption; a Medium Focus beam does equal damage; a Narrow Focus beam, double damage.

# of Ticks to Fire:  The number of ticks it takes for the weapon to charge before the beam is released.  This is effectively the delay between the instruction to fire the weapon and the time the weapon actually fires.  However, energy weapons have no cycle times

("Ticks to Reload") like the other weapons do: they may be fired multiple times in immediate succession.

## Damage:

Your bot is encased in armor.  Only when the external armor has been destroyed can your bot's internal components be damaged by projectile/grenade weapons; energy weapons bore through armor and damage internal components as well as weakening the armor.  A bot has four internal components: CPU, Engine, Battery, and Treads.  Each component can withstand a certain amount of damage before it is destroyed, called the component's Durability.  If the Treads are destroyed, the robot will slow to a stop and cannot move further.  If the Battery is destroyed, energy weapons and shields will not function.  If the Engine is destroyed, you cannot accelerate, and mechanical (i.e. projectile and grenade) weapons will not function.  And, if the CPU is destroyed, your robot has been eliminated.  If the CPU has taken damage equal to half of its initial Durability, the Scanner and Repair mechanisms will begin to have random failure.  They will still work correctly, but they will take an extra amount of time to successfully complete their function.  The average delay for a damaged Scanner is 4 ticks; for a damaged Repair system, 8 ticks.  If you purchase Repair capabilities, your robot can repair damage that has been done to its internal components.

Your robot may also be equipped with a shield (also known as a force field) that absorbs all forms of damage.  However, the more powerful the shield is, the faster it drains your battery's energy.  It may be turned on or off by a single command, and will turn off automatically if the battery is drained or destroyed.

Within the Architecture Editor, damage characteristics are divided into two categories: Durability and Protection.

**Durability** refers to the amount of damage that can be sustained by an internal component before it is destroyed.  The four internal components are Main CPU, Engine, Battery, and Treads, and you may set the durability value for each of these independently.

**Protection** characteristics are those that prevent damage to the internal components.  There are two basic types of protection, Armor and Shield.  Armor has two relevant characteristics:

Damage Points:  The total number of damage points that the armor can withstand before it is destroyed.  Remember that energy weapons can bore through armor to the bot's interior without damaging the armor itself.

Reflectance %:  The percentage of the energy (and damage) of a laser beam that is reflected away from the bot (doing no damage) if it is hit by an energy weapon.  If the armor is destroyed, the reflectance value immediately drops to 0.

A Shield absorbs a certain number of damage points from every hit (including energy weapons and collisions) sustained by your bot.  It must be turned on at the time of the hit to be effective.  Relevant characteristics:

Damage/Tick:  The amount of damage that the shield will absorb from each hit.

Efficiency:  The amount of energy that the shield drains from the battery every 10 ticks (while it is turned on) is proportional to the amount of damage that the shield can absorb.  A Low Efficiency shield drains twice as much energy as it absorbs damage; a Medium Efficiency shield drains an equal amount; and a High Efficiency shield drains half the amount.

## Sensors:

You can use your scanner to look for enemy bots.  You scan at a given angle and the closest bot located within scanwidth/2 degrees of that angle will be seen by your scanner (scanwidth is the width of the scanner you purchased).  This scanner will find a bot within the angle if the scan beam hits anywhere in the center 10 pixels of an enemy (16-pixel diameter) bot.  The angle and distance returned are to the center of the scanned bot.  If the center of the bot is not within the scan beam (this is possible because the scan will lock onto any of the bot's center 10 pixels), then the angle will be to the scanned point closet to the bot's center.  Information on the direction and distance to closest scanned bot is placed into the Sx registers (detailed in the tables at the end of this document).  Relevant scanner characteristics:

Degrees/Scan:  The size of the arc swept by the scanner in one activation (also called "scan width").  The arc centers on the angle specified in the Scan instruction.  Note that a "0 degree" scanner is not the same as no scanner: the beam simply must strike a target directly in order to successfully spot it.

Ticks/Degree:  The speed of the scanner.  One activation of the scanner takes roughly 2 ticks plus another amount.  For Slow scanners, this amount is twice the number of Degrees/Scan; for Average scanners, it's equal to the number of Degrees/Scan; for Fast scanners, half the number.

Range:  The maximum distance at which the scanner will function.  Enemy bots further than this amount away from the scanning bot will not be picked up by the scanner.  If you do not wish to have a scanner, select a range of 0 pixels.

You may also purchase Intelligent Damage Recognition (IDR) for your bot.  IDR for Damage Type places in register D1 the type of damage last taken (see below for details on damage types).  IDR for Damage Direction places in register D2 approximately what direction the damage came from,

rounded to the nearest axis.  That is, it will return 0, 90, 180, 270 as a rough estimate of the direction from which the bullet (or laser) was coming.

The Ax registers, A0–A4, contain the Durability left for your Armor, CPU, Engine, Battery, and Treads (in that order).  Register D0 contains the sum of the five Ax registers, and is thus your bot's current total Durability value.  When programming in Basic, you may use mnemonics for all of these registers.  The mnemonics are given in a table at the end of this document.

## Maintenance:

Your bot may repair damaged internal components using its Repair mechanism.  When activated, the repair system takes over control of the CPU for the requested number of 10-tick time units and repairs the requested internal component.  You can set the rate of repair (in damage points per 10—tick unit), and you can specify which of the four internal components (CPU, Battery, Engine, Treads) the repair unit can fix.  The repair mechanism cannot restore the Durability of a component to a level above that component's original Durability value.

The repair mechanism will fail immediately (i.e. the program will continue and nothing will have been repaired) in the following circumstances:  the bot is moving and Repair Engine or Repair Treads is requested; the shield is on and Repair Battery is requested; the Repair mechanism is not capable of repairing the requested component.

## Movement:

To move, you set your bot's target X and Y velocities, and your bot will accelerate or decelerate to reach the target velocity based on the acceleration and deceleration rates purchased.  Velocity range is from -255 to 255, and values higher or lower are truncated to -255 and 255 respectively.  A velocity of 64 corresponds to a 1 pixel per clock tick movement rate.

Should a bot collide with a wall or another bot, it will take damage proportional to the square of the collision velocity and collide semielastically.  The braking system of a bot kicks in immediately after a collision and slows the bot to a halt, as if a Velocity 0, 0 command had been executed. A VEL (or Velocity) command from the bot's program will override this braking, and the bot's velocity will be changed normally.

To use the command-key shortcut to access the Movement sub-dialog, type Command-V (where you would normally use M, the first character of the word "Movement").

## Battery:

The Battery provides any energy that the bot needs beyond that used by the CPU.  Specifically, it is used by Energy Weapons and by the Shield.  If

your bot has neither of these installed, it does not need a Battery.  Relevant characteristics:

Maximum Capacity:  The maximum amount of energy that can be stored in the Battery at any given time.  At the beginning of a battle, the Battery is charged to its maximum capacity.

Recharge:  The Battery will recharge by this many units every 20 ticks that the battle runs.  It cannot recharge to a level above its maximum capacity.

## The Code Editor

For flexibility of style and level of programming, users can program in two supplied languages: 'bot Assembly and 'bot Basic.  If you write your code in Basic, it will be compiled into Assembly before it is saved to disk.  As in any compiled language, 'bot Basic in certain cases will not be as well-optimized as carefully written 'bot Assembly, but we have worked to make Basic fast enough that bots written in Assembly will only have a slight advantage, if any.  However, just because a bot is programmed in Assembly does not mean that it has any better chance of winning; it is a combination of strategy and architecture that will make a good bot.  Overall, we highly recommend that inexperienced programmers write their robots in Basic rather than in Assembly, simply because Basic programs are easier for people to read and write.  If you have little or no programming experience, we also highly recommend that you read through the Basic Tutorial provided with 'bot.

Two quick comments need be made about the Developer.  The code for your bot can be as long as 32K in size, although that is more than should ever be needed.  A limit of 8192 is placed on the number of Assembly/object instructions in your code.  Again, this is far more than should ever be needed.

## Compiler Directives:

A given program is delimited by three directives that tell the compiler where data begins, where data ends and code begins, and where the code ends.  The program should start with the "#DATA" directive.  This is followed by the declaration of all variables (DEFs) and constants (EQUs) used.  A variable is declared as follows:

    DEF VARIABLENAME {INITVAL}

The INITVAL is an optional initial value for the variable to start at when the program runs.  If no INITVAL is given, the variable is set to default to 0.  Further, you may define constant identifiers which will be substituted into your program at compile-time.  A constant is declared as follows:

    EQU CONSTANTNAME <NUMBER>

When you compile your program, the compiler will search for all instances of CONSTANTNAME and replace them with the number '<NUMBER>'.  Thus, you may declare a constant like SCAN_INCREMENT to be equal to 7, and use it everywhere in a program.  If you later decide to change the increment to 8, you need only change the value in the EQU statement, and the new value will be used throughout the program when you recompile it.  Additionally, EQU directives may be combined in simple expressions, such as:

  EQU SCANRANGE 14
  EQU SCANWIDTH 5
  EQU SCANJUMP (SCANRANGE+SCANWIDTH)/2

You may use complex values like the above as long as every constant that you reference has been declared above the one being defined.  For example, you would get an error above if you defined SCANWIDTH *after* defining SCANJUMP instead of before.

Note that if you do not use any variables or constants, you need not start with the #DATA directive.  After the data segment (if there is one) comes the code segment, delimited by the "#CODE" directive and a language name, as in:

  #CODE ASM

The keyword "BASIC" should replace "ASM" if your code is in Basic and not in Assembly.  Following the #CODE comes your program, which will control the operation of your bot in the Arena.  A given line in a program can be a blank line, a comment, a jump label, or an instruction followed by its arguments.  Blank lines, extra spaces or tabs, and anything between a comment and the end of the line that it's on are ignored by the parser.

At the end of your program, after your last line of code, place the "#END" directive to tell the compiler that it has reached the end of your program.

You can define a jump label in your code by putting ":LABEL" in your code on a line by itself (although it may be followed by a comment), where LABEL is the identifier for that jump label.  Note that the label is immediately preceded by the colon (':') character, with no intervening spaces.  See the JMP instruction (in Assembly) and the GOTO command (in Basic) below for more information on what jump labels are for.

The comment delimiter is the exclamation-point ('!') character, which can be placed anywhere on a line.  Anything after the '!' will be ignored by the compiler, and thus you can use it to describe the purpose of a variable or a line of code in your program for your own future reference.

The bot Assembly and bot Basic languages are entirely case-insensitive; all lower-case characters are converted to upper-case during the compilation process.  For example, ScanWidth and SCANWIDTH reference the same thing, and both are perfectly valid.

Identifiers (variable names, constant names, and jump label names) must consist of alphanumeric characters ('A'-'Z', '0'-'9'), underscores ('_'), and dollar signs ('$') only. The first character of an identifier must be alphabetic ('A'-'Z' only).  The first twelve characters of the identifier are used by the parser for identification, and the rest are ignored.   Therefore,   be   aware   that   the   identifiers   DAMAGE_TAKEN   and DAMAGE_TAKEN_RECENTLY are the same as far as the parser is concerned (but "DamageTaken" and "DamageTakenRecently" are different).   Also, no two-character identifiers of the form LN, where L is a letter and N is a number, are allowed — these are reserved for register names.

## 'bot Basic:

The bot Basic language is a melding of standard Basic and newer programming styles.  When programming in Basic, you begin with your #DATA section as you would in Assembly, using the DEF and EQU statements.  To start your bot's code section, use '#CODE BASIC' to tell the compiler you will be programming in Basic.  End the program with '#END'.  The 'bot Basic language uses the same ':LABEL' directive for its GOTO labels as Assembly uses for its JMP labels.  Also, long lines may be extended onto a new line by placing an '&' at the end of the line and continuing to code on the succeeding line.  The following is a list of Basic commands.  There is a less detailed version of this list in a Table at the end of this document.

### Flow Control:

Goto <LABEL>　　　Executes code starting at the line of code right after the jump label LABEL appears.

Gosub <LABEL>　　　Executes code starting at the line after LABEL and ending at the next Return, at which point execution returns to the line immediately following the Gosub.

Return　　　　　　Returns and executes code following the GOSUB most recently executed.  Note that GOSUBs within GOSUBs are allowed.

If (<EXPR> <CMP> <EXPR>) Then <CODE> { Else <CODE>}

　　　　　　　　<CMP> is one of ==, <, <=, >, >=, or <>

　　　　　　　　<EXPR> is an expression as described below

　　　　　　　　<CODE> is any legal Basic command, including another If or a FOR-NEXT loop.

　　　　　　　　If the comparative is true, then the code following the Then is executed.  If the comparative is false, the code following the Else (if specified) is executed.  To have more than one line of code after a Then or an Else, use the BEGIN-END construction (discussed below).

You may go to a new line before the Else section if you like without adding an '&' character to the end of the line. For example:

```
If (X <> Y)     Then X = Z
                         Else Y = Z
```

For <VAR> = <EXPR> To <EXPR> { Step <EXPR> }

Iterates through the code between the FOR statement and its matching Next statement (below), incrementing <VAR> by 1 at the end of each iteration (or by the amount specified by the Step term if Step is specified), until <VAR> is greater than the expression specified after the To. The code between the For and the Next can be of any length and can include other FOR-NEXT loops. Negative Steps are not allowed in this version of 'bot Basic.

Next <VAR>          Marks the end of the For loop on variable <VAR>; it increments <VAR> by the amount specified in the For statement, and loops back to the For statement to test if another iteration should be performed.

While (<EXPR> <CMP> <EXPR>)
  <CODE>

(<EXPR> <CMP> <EXPR>) is exactly the same as for IF-THEN statements (above). If this comparative is true, the code immediately following the WHILE statement is executed. The comparative is then tested again, and this process repeats until the comparative is false, at which time program execution continues on normally. To have more than one line of code in a WHILE loop, use the BEGIN-END construction (discussed below)

Begin          A Begin statement marks the beginning of a block of code that is to be treated as if it were a single line. For example, in the statement If (A==0) Then <code>, <code> must be a single Basic statement. To expand <code> to more than one line of Basic, place a Begin statement before the lines of code, and a matching End statement after them. In this example, the result would be:

```
  If (A==0) Then          or     If (A==0) Then Begin
     Begin                             <code>
        <code>                         <more code>
        <more code>                 End
     End
```

These two examples are exactly equivalent, and simply illustrate two different formatting styles. The

various indentations are optional, of course. This construction is also useful for WHILE loops.

End                          An End statement marks the end of a block of code started with a Begin statement (above). Every Begin must be followed by a matching End, and vice versa.


**'bot Control:**

Scan Angle <EXPR>   Send out a scanner pulse at the angle <EXPR>. Register S0 is set to 1 if a bot is found, and in that case register S1 is set to the angle to the enemy (in degrees) and S2 to the distance to the enemy (in pixels). If no bot is found, S0 is set to 0 and S1 and S2 are undefined.

Repair <EXPR1> For <EXPR2>
                             <EXPR1> is the system to be repaired (see constants defined below)
                             <EXPR2> is how long to repair in 10-tick units. If <EXPR2> is less than 0 it is assumed to be 0, and if it is greater than 25, it is assumed to be 25 (which equals 250 ticks of repair).
                             If you don't have the repair mechanism installed for the system you wish to repair, nothing happens. Also, if you try to repair your engine or treads while moving, or if you try to repair your battery while your shield is on, nothing happens.

Fire Weapon <EXPR> {, Angle <EXPR1>} {, Distance <EXPR2>}
                             Fire weapon #<EXPR> at angle <EXPR1> and at distance <EXPR2> (which only needs to be specified if using a Grenade-type weapon). If <EXPR> is not equal to 1 or 2, this command does nothing. If Angle or Distance is not specified, it will default to whatever the last value was for that term (Thus to fire repeatedly you could specify all of the arguments on the first shot, and only "Fire Weapon 1" on the subsequent shots).
                             If you can't fire the specified weapon (if, for example, you didn't have that weapon, or if you were out of bullets or energy), then nothing happens. If the weapon requires a cycle time before firing twice in a row (projectile weapons and grenade launchers), and that time has not yet elapsed, then nothing happens. You may use Wait Weapon (below) to prevent misfires of this type.

Shield <EXPR>        Turns the shield On or Off (1 or 0, respectively). If you don't have a shield, nothing happens.

Velocity <EXPRX>, <EXPRY>

Accelerates (or decelerates) the bot as quickly as possible to <EXPRX>, <EXPRY> velocity.  Both arguments must be specified.

Wait <EXPR>        Waits for <EXPR> ticks to go by.  **OR**:

Wait Weapon <EXPR>Waits until Weapon #<EXPR> has reloaded.  If <EXPR> is not equal to 1 or 2, this command does nothing.  This instruction is useful for synchronizing a firing routine with the weapon cycle time.

**<EXPR> definition:**

Any variable (DEF), number (EQU or literal), or mnemonic (register name — see below) is an expression.

If A, B are expressions, then the following are also legal expressions:

| | |
|---|---|
| (A) | (order of operation) |
| -A | (negation) |
| Sin(A) | (Sine(A) * 256  (an integer)) |
| Cos(A) | (Cosine(A) * 256 (an integer)) |
| Abs(A) | (the absolute value of A) |
| Random(A) | (random number between 0 and (A-1)) |
| A + B | (addition) |
| A - B | (subtraction) |
| A / B | (division) |
| A * B | (multiplication) |
| ArcTan(A, B) | (ArcTangent(A/B) where A = Y, B = X) |

The following constants are also defined for your use as expressions:

    **For Shield:**
      ON
      OFF

    **For Repair:**
      C_CPU
      C_ENGINE
      C_BATTERY
      C_TREADS

    **For Damage (test versus $DAMAGETYPE):**
      C_ENERGY
      C_BULLET
      C_GRENADE
      C_COLLIDE

For example:

```
Shield ON
Repair C_CPU For 5
If ($DAMAGETYPE <> C_COLLIDE) Then Goto RunAway
```

All read-only registers (as specified in Assembly section of the manual, which should be read for full details) are accessible from BASIC, but R0-R9 are not. For specifications and complete definitions of the Register variables that you can (and need to) use, read the "Assembly Registers" section in the tables below. However, you are not required to use these register names if you do not wish to. The Basic Mnemonics Table (also below) lists equivalent mnemonics that you can use instead, although you should still look in the Assembly Registers section for descriptions of the information stored in each register.

The BASIC compiler is fairly smart, and it will produce optimized assembly code wherever possible. A statement like "X = Y+X+3*5" is made into the fastest possible assembly code (in this case, the instructions created are "ADD Y, X" followed by "ADD 15, X"). However, be aware that a statement like "X = Y+(2*X*Z)+(2*X*Z)" is not fully optimized, and you should probably use two statements, "TEMP = 2*X*Z" and "X = Y+TEMP+TEMP" (or similar), instead if you want to have fully optimized code.

Double negatives are removed from the code (--X simply becomes X), constant expressions are evaluated where possible (3+X+5*2 becomes X+13), additions of 0, and multiplications by 1 are removed, and the resulting assembly code has "CLR X" rather than "MOV 0, X" where possible (CLR takes less time to execute than a MOV). A few other optimizations are made as well. Generally speaking, there is no great disadvantage to using BASIC against enemy bots written in Assembly, and BASIC programs, for most people, are much easier to read and write than assembly language programs.

## 'bot Assembly:

While many of the 'bot Assembly commands are 'standard' assembly code commands, many of them are specific to bot operations. All Assembly code is written one line per instruction, in the form:

    MNEMONIC {ARG1 {, ARG2}}

The arguments ARG1 and ARG2 are dependent upon the particular instruction; some instructions have no arguments, some have one and others have two. The list of Assembly instructions, what they do, and what argument(s) they take can be found in the tables presented at the end of this document.

For those who know Assembly language in one form or another, note that no indirect addressing is allowed. That is, you can only directly address a variable by using a DEF command described later. In addition, no arithmetic may be performed on an argument in an instruction, so the

following would be invalid, assuming that LAST_ANGLE is a variable rather than a constant:

    MOV LAST_ANGLE+5, SCAN_ANGLE

Instead, you'd do the following:

    MOV LAST_ANGLE, SCAN_ANGLE
    ADD 5, SCAN_ANGLE

Various instructions of bot Assembly place return values in special read-only registers which are described below.  Further, ten scratch registers, R0–R9, exist for your use and are both readable and writeable.  The instruction to fire a weapon (FIR) requires that its parameter(s) be placed into certain of these registers before they are executed (again, documented in the tables below).

## Assembly Instructions:

In the following instructions, the letters A and B are used to denote an EQU, DEF, or register label, or a literal constant; the letter C is used to denote a jump label.  Note that although instructions like ABS take two parameters, a statement like "ABS A, A" is perfectly legal, and A will be set to its own absolute value.

There are 16 Instruction Units (I.U.'s) per clock tick.  Different instructions take varying amounts of time to execute, based on how complicated the instruction's function is.  The number of I.U.'s for each instruction is given in boldface after the instruction in the table below.  Note that this information is really only useful to people trying to synchronize/optimize their bot Assembly code to an extreme degree.  In normal usage, this information should be used just to get an idea for how long some operations are going to take during a battle.  The time for the SCN instruction does not include the time that the scanner itself uses to effect the scan; details on scanning times can be found in the section on bot Architecture, above.

Memory Instructions:

| | | |
|---|---|---|
| CLR B | **4** | Sets A to 0 (functionally equivalent to MOV 0, A). |
| MOV A, B | **8** | Sets B to the value of A. |

Mathematical Instructions:

| | | |
|---|---|---|
| ADD A, B | **16** | Adds the value of A to B. |
| SUB A, B | **16** | Subtracts the value of A from B. |
| MUL A, B | **20** | Multiplies B by the value of A. |
| DIV A, B | **20** | Divides B by the value of A.  B is then rounded down. |
| NEG A, B | **8** | Sets the value of B to be –A. |
| ABS A, B | **8** | B is set to the absolute value of A. |
| RND A, B | **32** | B is set to a random number between 0 and (A-1), inclusive. |
| SIN A | **80** | R0 is set to SINE(A°) * 256. |
| COS A | **80** | R0 is set to COSINE(A°) * 256. |
| ATN A, B | **160** | R0 is set to ARCTAN(A/B). |

Unconditional Jump Instructions:

| | | |
|---|---|---|
| JMP C | **4** | Jump to instruction following label C. |
| JSR C | **6** | Jump to subroutine following label C. |
| RET | **4** | Return from subroutine jumped to by a JSR.  Program execution continues with the instruction following that JSR. |

Conditional Jump Instructions:

| CMP A, B | **16** | Do a comparison of A and B.  Branch statements below jump based on the last CMP performed: |
| BEQ C | **5** | Branch (jump) to instruction following label C if (A=B). |
| BNE C | **5** | Branch to label C if (A ≠ B). |
| BLT C | **5** | Branch to label C if (A < B). |
| BLE C | **5** | Branch to label C if (A ≤ B). |
| BGT C | **5** | Branch to label C if (A > B). |
| BGE C | **5** | Branch to label C if (A ≥ B). |

Miscellaneous & bot Specific Instructions:

| NOP | **16** | Does nothing for that tick. |
| WAI A | **16** | Wait until Weapon #A has reloaded.  If A is not equal to 1 or 2, this instruction does nothing.  This instruction is useful for synchronizing a firing routine with the weapon cycle time. |
| FIR A | **16** | Fire Weapon #A.  If A is not equal to 1 or 2, this instruction does nothing.  Register R1 should contain the angle at which to fire.  If the weapon is a Grenade Launcher, register R2 should contain the target distance.  If the weapon has not had time to reload, this instruction will do nothing. |
| SCN A | **8** | Scan at A°.  Results are placed into the S registers (detailed above). |
| SHL A | **32** | Turn shield off if (A=0) or on if (A≠0).  If the battery has no charge left or has been destroyed, this instruction does nothing.  If the bot has no shield installed, this instruction does nothing. |
| RPR A, B | **16** | Repair system A for B cycles. |

A: System:
0 CPU
1 Engine
2 Battery
3 Treads

A "cycle" is 10 ticks or 160 Instruction Units.  Valid values for B are 1-25 (i.e. 10 - 250 ticks).  Any value over 25 is taken as 25.  The bot will lie dormant for the given number of ticks, and then any damage that the component has taken, up to (B * RepairRate) points, will have been repaired.  If the bot is moving, attempts to repair the Engine or Treads will fail immediately.  If the shield is on, attempts to repair the Battery will fail immediately.  Note that you can, for instance, start moving and then repair

your CPU while your momentum carries your bot across the Arena.

Appendix B:

## Assembly Registers:

The sets of memory registers are broken into subsections by specific use. The user scratch registers, R0–R9, may be read from and written to, just like DEF variables. The other register banks, however, are read-only, and should not be written to. If you do change the value of one of these read-only registers, that register's value is undefined from that point on. That is, the value could be anything at all.

Scratch Registers:

| | |
|---|---|
| R0–R9 | Anything you want… (R1/R2 used for weapon firing, R0 used by mathematical functions to return a value; not accessible in bot Basic) |

Scanner Registers:

| | |
|---|---|
| S0 | Scanner result:  0 = No bot scanned.  1 = bot scanned. |
| S1 | Angle to target |
| S2 | Distance to target |

Selective Damage Registers:

| | |
|---|---|
| A0 | Damage points left in Armor |
| A1 | Damage points left in CPU |
| A2 | Damage points left in Engine |
| A3 | Damage points left in Battery |
| A4 | Damage points left in Treads |

Overall Damage Registers:

| | |
|---|---|
| D0 | Sum of A0, A1, A2, A3, and A4. |
| D1 | Type of damage last taken was from a: (if bot has damage type recognition) |

D1 Type
1  Energy Weapon
2  Bullet
3  Grenade
4  Collision.

| | |
|---|---|
| D2 | Direction of damage last taken: (if have damage direction recognition) |

D2 Direction (approximate)
0     0° ± 45°
90    90° ± 45°
180   180° ± 45°
270   270° ± 45°

Weapon & Battery Registers:
 B0              Current Battery charge.
 B1              Number of shots left for Weapon 1.
 B2              Number of shots left for Weapon 2.

bot Position Registers:
 X0              Current X position (0 to 255).
 Y0              Current Y position (0 to 255).
 X1              Current X velocity (-255 to 255).
 Y1              Current Y velocity (-255 to 255).

Appendix C:

# Basic Mnemonics for Registers:

| Register | Mnemonic | Description |
|----------|----------|-------------|
| A0 | $ARMOR | Armor durability remaining |
| A1 | $CPU | CPU durability remaining |
| A2 | $ENGINE | Engine durability remaining |
| A3 | $BATTERY | Battery durability remaining |
| A4 | $TREADS | Tread durability remaining |
| | | |
| B2 | $CHARGE | Current battery charge |
| B1 | $BULLETS1 | Bullets left for Weapon 1 |
| B2 | $BULLETS2 | Bullets left for Weapon 2 |
| | | |
| D0 | $DAMAGE | Total durability value |
| D1 | $DAMAGETYPE | IDR — type of damage last taken |
| D2 | $DAMAGEDIR | IDR — direction of " " " |
| | | |
| S0 | $FOUND | Scan result |
| S1 | $ANGLE | Angle to scanned bot |
| S2 | $DISTANCE | Distance to scanned bot |
| | | |
| X0 | $XLOC | Current X location |
| X1 | $XVEL | Current X velocity |
| Y0 | $YLOC | Current Y location |
| Y1 | $YVEL | Current Y velocity |

Appendix D:

# Basic Statements Quick Reference:

Goto <label>
Gosub <label>
Return
If (<term> <cmp> <term>) Then <CODE> { Else <CODE> }
For <var> = <term> To <term> { Step <term> }
Next <var>
While (<term> <cmp> <term>) <CODE>
Begin <CODE> ... <CODE> End

Scan Angle <term>
Repair <system> For <ticks/10>
Fire Weapon <term> {, Angle <term> } {, Distance <term> }
Shield <term>
Velocity <termX>, <termY>
Wait <ticks>
Wait Weapon <term>

| | |
|---|---|
| Sin(A) | (Sine of A) * 256 |
| Cos(A) | (Cosine of A) * 256 |
| ArcTan(Y, X) | ArcTangent of Y/X. |
| Abs(A) | Absolute value of A |
| Random(A) | Random number between 0 and A-1 |

## <u>Constants:</u>

| | | | | |
|---|---|---|---|---|
| For Shield: | ON | OFF | | |
| For Repair: | C_CPU | C_ENGINE | C_BATTERY | C_TREADS |
| For Damage: | C_COLLIDE | C_BULLET | C_GRENADE | C_ENERGY |

Appendix E:

## Helpful Arena Information:

0 degrees points right
90 degrees points down
180 degrees points left
270 degrees points up

The Arena is physically 272*272 pixels.  To a bot, the Arena is only 256*256 pixels, as bots are 16 pixels in diameter.

(0, 0) is in the upper left of the Arena.  (256, 256) is in the lower right.

A bot executes 16 "Instruction Units" every tick.  Therefore, a bot could execute multiple instructions in a given tick, or none at all.

A velocity of 64 corresponds to 1 pixel/tick.  Velocity range is -255 to 255.

The scanner will lock on to a target if any of the center 10 (of the bot's 16) pixels fall within the scan arc.

A battle is declared a draw if either a total of 25,000 ticks have passed, or 5,000 ticks have passed with no damage occurring to any bot.

To move the Arena window, hold down the option key and drag the window with the pointer.

**Coming in future releases:**

**Arena**
Color
Resizable Arena
More bots in Arena at one time
GUI Assembly Debugger
Animated SICN Support
Terrain & Obstacles

**Developer**
Multiple bots open at once
More languages
Bigger & better compiler support of language constructs
Identification of scanned enemies
Arrays
SICN Editor
More complex Architectures & more precise editing

**Both:**
System 7.0 Studliness