BASIC TUTORIAL

Please don't be intimidated by the size of this tutorial. The tutorial is divided into a number of different sections, each one taking you through the steps of building a progressively more complex bot. You can read only the sections you want, skipping those that seem too advanced or too simple.

This document is intended as an aid for users with little or no programming experience. It might be a good idea for you to take a quick look through the Basic portion of the manual before starting this tutorial, just so you have some idea of what the language is going to look like. You also might want to have a printed copy of the Basic Tables given at the end of the documentation file, but this is not necessary. The programs presented here are intended as learning tools rather than as efficient and effective code, so don't be surprised when more complex demonstration bots are beaten by simpler bots. For information about designing your bot's physical characteristics (its "architecture"), read the section of the manual on the Architecture Editor. This section describes all of your options. If you'd rather just play around on your own, you can type Command-H (or select "Architecture Editor" from the "Window" menu) when there is a bot loaded into the Developer. This will bring up the Architecture Editor dialog. This tutorial touches on many of the interesting weapons, defenses, etc. that you can give your bot, but it concentrates primarily on teaching you how to program bots.

Basic is a language designed to be easy for humans to read and write. Because of this, however, the bot's computer can't easily understand Basic, as Basic is just too complicated for it. So that the bot's computer can understand the program that you want it to follow, your Basic program is "compiled" into a very ugly language called Assembly. This language tells the bot's computer exactly what to do in a form that the computer can understand. To convert your Basic program into this language, you "compile" it by typing Command-K (or by selecting "Assemble 'bot" from the "Build" menu). This will save a "botname.bot" file that can be loaded into the Arena application and run.

If you like, you may write your bots in Assembly yourself, rather than have the compiler write the Assembly using your Basic programs. This tutorial does not deal with Assembly programming, and we don't really recommend that beginning programmers use Assembly, as it's a lot harder for humans to comprehend. There is a section in the manual on ’bot Assembly programming for those who wish to try. Also, Assembly language versions of all of the Tutorial bots, with comments on what they do, have been provided for you should you wish to look at them.

First let's consider a bot that sits in one place looking for robots and shooting at them when it finds them (Seeker.bot). The first section of code we need should scan around in a circle:

```
:ScanLoop
        ScanAt = ScanAt + INCREMENT
        Scan Angle ScanAt
        If ($FOUND == 0) Then Goto ScanLoop
```

:ScanLoop is a jump label (all jump labels must start with the ':' character). Whenever a GOTO statement is executed, the program continues execution at the line following the label given in the GOTO. In the example above, the Goto ScanLoop statement will cause the program to go back up to the top line of code and continue from there. Here we use INCREMENT as the amount to increment the angle by for each scan and scanAt as the current angle being scanned at (scanAt is a variable, and has some number as its value). scanAt = scanAt + INCREMENT makes scanAt equal to INCREMENT more than it was before. Thus if scanAt started with a value of 0, and INCREMENT had a value of 7, then each time this line of code was executed, scanAt's value would go from 0 to 7 to 14 to 21, etc. The line following this line executes the Scan Angle command using the current value of scanAt as the angle at which to scan.

After the Scan Angle command, the bot's computer automatically places the results of the scan into a variable called $FOUND. If $FOUND has a value of 0, then nothing was found; if it has a value of 1, then an enemy bot was found. Therefore, we have the computer to a conditional GOTO command: If ($FOUND == 0) tests to see whether the value of the $FOUND variable is equal to 0. If it is, then the statement immediately following the Then is executed next. In the example, this statement is Goto ScanLoop, so if nothing was found ($FOUND == 0), then go back up to the ScanLoop label and continue with the program from there. When this happens, the line of code to add INCREMENT to scanAt will be executed again, so scanAt will have a new value when the Scan Angle scanAt statement is executed again. Thus, this program causes your bot to scan around in a circle (adding, say, 7 to its current angle each time) until it finds an enemy. When the scanner does find an enemy, $FOUND will have a value of 1, so the test ($FOUND == 0) will be false, so the Goto ScanLoop piece will not be executed at all, and the program will continue with the next segment of code (below). Note that the value of scanAt can go above 360 degrees with no problem: scanning at 360 degrees is the same as scanning at 0 degrees; 450 degrees is the same as 90 degrees is the same as -270 degrees.

I'll digress for a while and answer some questions that you might be wondering about. The points that I'll make are mostly stylistic (that is, you don't really need to know them, but you'll be happier if you do). Then we can return to building the rest of the bot.

Notice that in the piece of code above, there one place where I use a single equals sign ("scanAt = scanAt + INCREMENT"), and another place where I user a double equals sign ("If ($FOUND == 0) Then ..."). A single equals means "set to the following value," as in "Set scanAt to the value that you get by adding scanAt's current value and the value of INCREMENT." A double equals mean "test for equality" as in "Is the value of $FOUND equal to 0?" The compiler will tell you if you mix them up, so don't worry about it too much.

You may have noticed that, in the piece of code given above, INCREMENT always has the same value, be it 7 or 13 or whatever you like. If you wanted to, you could define INCREMENT as a variable, and, somewhere before executing the piece of code above, you could set its value to 7 with a statement like INCREMENT = 7. This is perfectly fine, but it seems a little wasteful. If you're never going to change INCREMENT's value, you may as well just put the number 7 into your program everywhere where you wanted to put INCREMENT. This would also be a perfectly good way to get the same result — your scanning angle would still be incremented by 7 each time. But there is a problem with this approach.

Assume that your program is fairly large, and you use the same value (7) many times in the program as your scan increment in various situations. Then you decide that an increment of 11 would really be a lot better. So, you're now stuck with having to go through your entire program, changing all of your 7's into 11's. Then you decide that 7 was better, so you go through and change them all back. This is a pain. For this reason, we have provided a way (called an "EQU," which is short for "equivalent," and which is demonstrated below) to declare a constant value in your program. In this example, you can declare INCREMENT (or any word you wish) to be a constant with a value of 7 right at the top of your program. Then, you use the word INCREMENT instead of 7 everywhere in your code. When you compile the program, every occurrence of INCREMENT is treated exactly as if you'd written a 7 instead. This way, if you want to change all of your 7's to 11's, you need only change that top definition of INCREMENT from 7 to 11, and it is automatically changed everywhere in the program for you.

You might also have been wondering why, in the code given above, I wrote the variable scanAt with mostly lower-case letters but wrote INCREMENT with all upper-case. Since INCREMENT is a constant, not a variable, I chose capital letters as a way to remind myself that it was really just a number, so I couldn't have a statement like INCREMENT = X, because that would really be the statement 7 = X, which doesn't make any sense. I will use this convention in the rest of the tutorial: variables are written with mostly lower case letters, which constants are written in all upper-case. Also, there are some special variables like $FOUND above that the computer uses to give the program information about the bot's status. All of these start with the '$' character and are written in upper-case. I'll describe each of these as it comes up, but now you know what to look for. Remember that you don't need to follow any of these upper and lower-case conventions in your own programs if you don't want to — you can use any system you wish, but I suggest that you use *some* system, just to make the programs easier to understand. End of digression.

We had just determined that if the program's execution gets to the line after the above code, then the scanner must have found an enemy bot. When this happens, the bot's computer automatically stores the angle to the enemy in the special variable $ANGLE and the distance to the enemy in $DISTANCE. Therefore, in order to have our bullets hit the enemy, we want to fire the weapon at an angle equal to $ANGLE (for a normal gun, distance doesn't matter, so we can ignore $DISTANCE). This bot only has one weapon (you are allowed to have two), but we still need to tell the computer exactly what to fire. The two weapons that we could have are numbered 1 and 2: since we only have a single weapon, that weapon is number 1.

```
:ScanLock
        Fire Weapon 1, Angle $ANGLE
```

:ScanLock is a label that we'll need in a little bit, but you can ignore it for now. Now, since we know the location of this bot, there's no point in scanning around looking for more. So, keep scanning and shooting in the same place until the enemy dies or moves (if a bot dies, it disappears from the Arena):

```
        Scan Angle ScanAt
        If ($FOUND == 0) Then
                Goto ScanLoop
        Goto ScanLock
```

Since the value of ScanAt is still the same, the Scan Angle command scans in the same exact place, so it's checking to make sure that the enemy is still there. If not (that is, if ($FOUND == 0)), scanning is resumed at ScanLoop (just as shown above). If so (that is, if ($FOUND == 0) was not true), go back to the line following the ScanLock label (where the weapon is fired) and fire again. The complete program can be seen in the bot called Seeker. The code looks like this:

```
#DATA

DEF scanAt
EQU INCREMENT 7

#CODE BASIC

:Initialize
	scanAt = Random(360)

! This is a comment

:ScanLoop
	scanAt = scanAt + INCREMENT		! This is also a comment
	Scan Angle scanAt
	If ($FOUND == 0) Then Goto ScanLoop
:ScanLock
	Fire Weapon 1,  &
		Angle $ANGLE
	Scan Angle scanAt
	If ($FOUND == 0) Then
		Goto ScanLoop
	Goto ScanLock

#END
```

There are a few things in this code that still need to be explained. The first thing that the code does, before it enters the scan loop, is set scanAt to a random number between 0 and 359 (that's what Random(360) returns). There's no particular reason for this, but it's often good to inject a little randomness into you programs: it makes it harder for an adversary to second-guess what your bot is going to do. For example, if your program just left scanAt at 0 at the battle's start, bots that immediately moved to the top of the Arena would generally be safer — for a little while — than those that moved to the bottom (remember that 0 degrees points right and 90 degrees points down). This may seem a bit silly, but if you start at a random angle, no one could make a bot that moved to a known safe spot like that.

Why do I keep using 7 as my scan increment? Why not a nice normal number like 5 or 10? As your bot's scanner swings in its circle looking for enemies, remember that it might miss. It is quite possible that a bot some distance away from yours will fall into the “cracks” between scan beams. Now, if you have an increment of 10 degrees, then after 36 scans, your scanner has gone from 0 degrees (or wherever it started) to 360 degrees. Since 360 degrees is the same as 0 degrees, it’s right back where it started, so if it misses a stationary enemy on one sweep, it’ll miss it on *every* sweep! Now consider what happens when you use 7 instead of 10: after 52 scans, your bot is scanning at 364 degrees, which is the same as 4 degrees. On the second sweep, instead of scanning 0, 7, 14, 21, it will scan 4, 11, 18, 25. Effectively, you are scanning “in the cracks,” finding bots that might have been missed on the previous sweep around the Arena. There are many numbers that work well like this, so try some out for yourself. Prime numbers are generally best.

Okay, now that you understand the code, you’re probably wondering what those extra lines are doing there. The ones like “#CODE BASIC” and “DEF scanAt”. In order to tell the compiler the names of your variables, you must use DEF statements. A DEF statement consists of — on its own line — the keyword “DEF” followed by the name of the variable. If you wish, you may put a number (or an expression, like “INCREMENT+4”) after the variable name. When the program is run, the variable will start off with this value; if you don’t specify a number, the variable’s initial value will be 0. In order to tell the compiler the names and values of your constants (discussed above), you do the same thing as for DEFs, except that you use “EQU” instead of “DEF”. Also, you *must* specify the numeric value — it isn’t optional for EQUs. Note that you can have EQUs and DEFs whose values can be constants themselves, as long as you’ve defined the constant above the line the you use it on. For example, you could have

```
EQU INCREMENT          7
EQU BIG_INCREMENT      (INCREMENT*2) - 1
```

Where BIG_INCREMENT has value 13. However, the following would be invalid:

```
EQU BIG_INCREMENT      (INCREMENT*2) - 1
EQU INCREMENT          7
```

because the compiler can’t calculate the value of BIG_INCREMENT before it knows the value of INCREMENT. Note that (for this same basic reason) all of your DEFs and EQUs must come before the start of your actual program.

Also, there are three “directives” that indicate what you’re going to talk about next in your program. Right at the top of your bot file, you should put a #DATA directive. This lets the compiler know that what follows is the section where you declare your variables and constants. If your program doesn't use any variables or constants, then you don't need to put in the #DATA directive. After you've put in all of your DEFs and EQUs, you need the directive #CODE BASIC. This lets the compiler know that what follows is the program itself, and that it is written in Basic. If you were writing in Assembly Language (which you may do if you like), you would instead use #CODE ASM. And, at the very end of the file, put the #END directive to let the compiler know that the text of your program has ended.

Finally, remember that you can put comments — notes to yourself or other people that are ignored when the code is compiled — on any line by putting an exclamation point character ("!") in front of your comment. Everything from the "!" to the end of the line will be ignored. Also remember that if you want to continue a long line of the program onto the next line, you need to put an ampersand ("&") at the end of the first line. So if you can't fit

```
Fire Weapon 1, Angle (savedAngle - ANGLE_CONSTANT * 17 + 123)
```

on one line, then you can write it as

```
Fire Weapon 1,        &
          Angle (savedAngle - ANGLE_CONSTANT * 17 + 123)
```

if you like. The exception to this is that after the word THEN (or ELSE) in an IF-THEN statement, you can go to the next line without needing an ampersand, as in

```
If (myNumber <> A_CONSTANT) Then
          myNumber = myNumber + 1
```

This concludes the first part of the tutorial. You have now learned some of the basics of creating a bot. Take some time now (if you haven't already) and play with the bot called Seeker in the Developer and in the Arena. Experiment with changing the scan increment, reconfiguring its architecture (weapon, armor, etc), and anything else you like. Then, continue on below with the next robot in the tutorial.

One of Seeker’s flaws is that it must scan a full 360 degree arc in order to find other bots. If it were positioned at an edge of the arena, it could scan the whole field with only a 180 degree sweep. And in the corner, it would need only 90 degrees. Let’s make a bot now that moves to the upper left corner of the arena and scans from there. Be aware that although you gain a great scanning advantage when you’re in a corner, any bot that takes the time to look in the corners can easily spot you.

```
:XMoveToCorner
	Velocity -$XLOC, 0
	If ($XLOC > 10) Then Goto XMoveToCorner
	Velocity 0, 0
```

This code moves you toward the left side of the screen, slowing down as you approach. $XLOC is a special variable that always contains your bot’s current X (horizontal) position in the Arena — a value from 0 to 255. By using the statement Velocity -$XLOC, 0 to set your bot’s X velocity to its negative X position and its Y velocity to 0, you make your bot move directly left. As the bot gets closer to the left wall of the Arena, $XLOC will get closer to 0, so the bot will slow down instead of just smashing into the wall. The code here keeps looping through the Velocity statement until its X position is less than 10. That is, until the bot is 10 pixels away from the left wall of the Arena. At that point, the bot is stopped with the Velocity 0, 0 statement.

There is a special variable $YLOC too that’s just like $XLOC except that it stores the bot’s current Y position. Also, $XVEL and $YVEL always store the bot’s current X and Y velocities, respectively.

The above code can be easily modified to do Y movement the same way:

```
:YMoveToCorner
	Velocity 0, -$YLOC
	If ($YLOC > 10) Then Goto YMoveToCorner
	Velocity 0, 0
```

The bot will execute these two fragments right at the start of the combat. After they have finished executing, the bot is stopped in the upper right corner of the Arena. Now, the bot wants to scan the arena systematically, without wasting time scanning the left and top walls of the arena. The scanner, then, needs to go from roughly 0 to 90 degrees, and then repeat starting from 0 again. In BASIC, you can use a FOR-NEXT loop to accomplish this. One pass from 0 to 90 looks like this:

```
For s = 0 To 90
        Scan Angle s
        If ($FOUND <> 0) Then <fire at the enemy>
  Next s
```

This means that the variable “s” starts off with the value 0 (as if you’d put “s = 0”). Then, the “body” of the loop, that is, the code between the FOR statement and the NEXT statement, gets executed. When this code has been executed once, the variable s is incremented by one (as if you’d put “s=s+1”), and the program compares s to 90. If s is greater than 90, the program continues on after the Next s statement. If s is less than or equal to 90, then the program goes back up to the top of the “body” and repeats the execution.

What will happen in this example is that the bot will scan at 0 degrees, then at 1 degree, then at 2 degrees, etc., until it finds an enemy, at which point it will execute whatever is put where it says <fire at the enemy>. When it gets to 91 degrees, the test to see if (s > 90) will finally succeed, and the program will continue below the FOR-NEXT loop.

Of course, if you increment your scanner by 1 degree every time, it will take an awfully long time to do a single sweep of the arena. So once again, we want to use a larger scan increment INCR (a constant, let’s say “7”). If you want to have a FOR-NEXT loop add more than 1 to your variable each time through the loop, you can specify it as the “Step” for the loop in the following manner:

```
For s = 0 to 90 Step INCR
```

As the loop executes, the bot will scan at 0, 7, 14, ..., 84, at which time the loop will terminate (because (s == 91)).

When s is greater than 90, we want to start the scanning all over again at 0. To do this, we just put a GOTO after the NEXT statement that goes back up to the top of the loop. So now the code looks like this:

```
:ScanLoop
        For s = 0 To 90 Step INCR
                Scan Angle s
                If ($FOUND <> 0) Then <fire at the enemy>
          Next s
        Goto ScanLoop
```

Except now we run into a problem discussed above — enemy bots can fall into "cracks" in the scanner, as it always scans at 0, 7, 14, etc., degrees. To get around this, we can start s off with a random value each time. If INCR is 7, then we could pick a random number between -3 and +3 and start there. Eventually, all of the "cracks" would be scanned (you can't be *absolutely* sure that all of the cracks are scanned in, because Random might return 2, 2, 2, 2, 2 the first five times your program calls it, but it's a pretty good bet that it won't). Also, if a bot were sitting directly below yours, say in the bottom left corner, it might never be scanned, because your bot will never scan at any angle greater than 90 degrees, and it might be at, say, 91 or 92. So we could make the code look like this:

```
:ScanLoop
        For s = Random(INCR)-3 To 93 Step INCR
                Scan Angle s
                If ($FOUND <> 0) Then <fire at the enemy>
          Next s
        Goto ScanLoop
```

The <fire at the enemy> part of the code still needs to be written. We could just replace it with "Fire Weapon 1, Angle $ANGLE", but then it wouldn't "lock" on to the enemy it found — it would fire once and then continue with the loop, incrementing the scan angle in the process. So, a more complicated routine of some sort is needed, but there's really no room in this FOR-NEXT loop for a complex routine. Therefore, we will use a subroutine. A subroutine is a small (or large) piece of code that performs a specific function. It is called with a GOSUB statement, which is just like a GOTO statement except that a GOSUB remembers where it was when you called it. When the program eventually executes a RETURN statement, it goes back to where it was when the subroutine was called with the GOSUB. An example will help — we needed some code to fire at a scanned enemy:

```
:LockWeapon
        Fire Weapon 1, Angle $ANGLE
        Scan Angle $ANGLE
        If ($FOUND <> 0) Then Goto LockWeapon
 Return
```

This looks a lot like the scan locking mechanism we used in Seeker (above). When you call this subroutine ("Gosub LockWeapon"), your program breaks off from where it was and jumps down to execute this code. At this point, it hits the Return statement, which tells it to resume the program where it left off. Note that you can call subroutines within subroutines without any problem, even if you’re calling the very same subroutine multiple times. The most useful thing about a subroutine is that it can be called from anywhere: If you had a very complicated bot that needed similar firing routines in a number of different places, you could use a single subroutine called from all those places. You may remember that I said a similar thing about defining constants in your programs. Just like when you change a constant, if you think of a better firing routine, you only need to replace the code once rather than 3 or 4 times.

For example, since this firing routine is now nicely compartmentalized, it is easier to add a tracking system — that is, an attempt to figure out where the enemy has gone when the scan lock is lost. Before continuing with the 90 degree scan, make the bot look back a little bit to see if the targeted bot is moving in that direction (since the main scan loop naturally scans around in the positive direction, there’s no need to put in a special check for the enemy in that direction):

```
:LockWeapon
        Fire Weapon 1, Angle $ANGLE
        tempScan = $ANGLE
        Scan Angle tempScan
        If ($FOUND <> 0) Then Goto LockWeapon

        Scan Angle tempScan-20
        If ($FOUND <> 0) Then Goto LockWeapon
        Scan Angle tempScan-10
        If ($FOUND <> 0) Then Goto LockWeapon
 Return
```

This expanded subroutine checks 20 degrees and then 10 degrees less than the bot’s last known position to see if it’s moving that way (it only does this if $FOUND is 0, of course; if the bot hasn’t moved, then the subroutine immediately loops back to shoot and scan again). If it finds the bot, it re-locks and continues firing. If not, it returns back to the main FOR-NEXT loop. The tempScan variable is needed because if the scanner doesn’t find a bot (i.e. (S0 == 0)), then the value of S1 is undefined — it could be anything. Therefore, we have to save it while it’s still defined (i.e. before the scan actually occurs) so that when the lock is eventually lost, we know where we were scanning. Eventually, even these extra two checks for the enemy bot will fail, and the Return statement will execute, returning control back into the FOR-NEXT loop where it left off.

Notice that you can put subroutines anywhere in your program file (except in the middle of other routines). However, you should probably

group them all at the top or bottom of the file so that you can locate them easily.
This subroutine finishes up this bot. The complete program can be seen in the bot called Corner. The code is also presented here:

```
#DATA

EQU INCR 7

DEF s
DEF tempScan

#CODE BASIC

:XMoveToCorner
        Velocity -$XLOC, 0
        If ($XLOC > 10) Then
                Goto XMoveToCorner
:YMoveToCorner
        Velocity 0, -$YLOC
        If ($YLOC > 10) Then
                Goto YMoveToCorner
        Velocity 0, 0

:ScanLoop
        For s = Random(INCR)-3 To 93 Step INCR
                SCAN ANGLE S
                If ($FOUND <> 0) Then Gosub LockWeapon
          Next S

        Goto ScanLoop

! SUBROUTINE -- Scan lock with tracking system

:LockWeapon
        Fire Weapon 1, Angle $ANGLE
        tempScan = $ANGLE
        Scan Angle tempScan
        If ($FOUND <> 0) Then Goto LockWeapon

        Scan Angle tempScan-20
        If ($FOUND <> 0) Then Goto LockWeapon
        Scan Angle tempScan-10
```

```
          If ($FOUND <> 0) Then Goto LockWeapon
 Return

#END
```

There are other weapon types besides the standard projectile weapon. For example, you may find a grenade launcher to be useful. Although you probably can't get many grenades into a given launcher, the few you can have can be deadly: grenades do more damage than regular projectiles, they hit everything within a large blast radius, and they damage all exposed components simultaneously. For example, a bullet might hit a bot with no armor, damaging its engine. However, a grenade exploding near that same bot would damage its cpu, engine, treads, and battery simultaneously, plus it would damage those of any other bots in the vicinity as well.

Grenades are fired at a given distance as well as at an given angle, so the command to fire a grenade is a little longer than for a bullet. Instead of the old fire command, which looks like this after a successful Scan:

```
Fire Weapon 1, Angle $ANGLE
```

you also have to specify the distance:

```
Fire Weapon 1, Angle $ANGLE, Distance $DISTANCE
```

(You can specify Distance first if you'd rather). This will launch a grenade to land at a point $DISTANCE pixels away from the center of your bot along angle $ANGLE. The grenade probably won't land exactly on that spot, as the launcher is not perfectly accurate, but it will be with a pixel or two. Also, the higher the velocity you have chosen for your grenades, the larger the error (but still not more than a few pixels from the target point). Since these are explosive grenades, it doesn't really matter if the grenade is a tiny bit off anyway. If the grenades that you launch are Impact Grenades, they will explode as soon as they hit the ground at the target point (Impact Grenades can be selected in the Architecture Editor under "Ticks until Detonation").

Be careful, though. If the enemy bot that you've scanned is right next to you, and your grenades have a large blast radius, you might be caught in the explosion. For this reason you should probably define a minimum distance for firing grenades equal to at least 10 more than your grenades' blast radius (remember that the distance is calculated from the center of your bot, which has an 8 pixel radius). That way, you'll still catch the target inside the blast radius, but your bot will be outside the blast. For example, you could do the following:

```
If ($DISTANCE >= MINDIST) &
    Then Fire Weapon 1, Angle $ANGLE, Distance $DISTANCE
    Else Fire Weapon 1, Angle $ANGLE, Distance MINDIST
```

Where MINDIST is something like 20 or 25. Remember that, just like for the Angle specification, once you've specified the distance once, you can fire again at that same distance by leaving out the "Distance $DISTANCE" portion of of the command.

Grenades generally take so long to fire and explode that there's not much point in repeatedly firing at the enemy: by the time the second grenade explodes, he'll be long gone, and you can't readily afford to waste grenades. So this firing routine won't bother to lock on. The whole scanning/firing routine, then looks like this:

```
ScanAt = ScanAt + INCR
Scan Angle ScanAt

If ($FOUND <> 0) Then If ($DISTANCE >= MINDIST) &
        Then Fire Weapon 1, Angle $ANGLE, Distance $DISTANCE
        Else Fire Weapon 1, Angle $ANGLE, Distance MINDIST
```

And we want to just keep looping through this section forever. We could use a GOTO for this: Put a jump label at the top of the code, and put a Goto LabelName at the end. However, we should try to do something when we're hit by an enemy bullet (or grenade). If the bot gets damaged, we can run away. The easiest way to tell if the bot has taken damage is to store the contents of $DAMAGE in a variable ($DAMAGE is a special variable that contains the total current durability value for the bot. If you wish to check the remaining durability of an individual component, you can check the special variables $CPU, $ENGINE, $BATTERY, $TREADS, and $ARMOR. $DAMAGE is the sum of these five). If the value in $DAMAGE suddenly becomes different than the saved value, the bot must have been hurt. So, we want to loop through the above scan routine until the bot is hit. That is, until (damage <> $DAMAGE), where damage is the name of our storage variable.

When we wrote Corner, above, we used a FOR-NEXT loop to repeatedly execute a piece of code. Then, we had a reasonably fixed number of times that we wanted to loop. Here, we want to loop until a certain condition is met (i.e. that the bot has taken damage). To do this, we can use a WHILE statement. A WHILE looks a lot like an IF-THEN (with no ELSE), except that it keeps executing the "THEN" portion of code until the condition is no longer true. For example, you could have your bot move left and then stop near the wall with the statements:

```
        While ($XLOC >= 20) Velocity -$XLOC, 0
        Velocity 0, 0
```

The velocity command will be executed over and over until finally (XLOC < 20), at which time the loop will terminate and the Velocity 0, 0 will be executed. If you want to put more than one line of code in the loop, you can surround the block of code to be repeated with the keywords Begin and End. You may also use the Begin-End construction after Then and/or Else in an IF-THEN statement. Anyway, the code loop should now look like this:

```
damage = $DAMAGE                    ! Initialize the damage variable

While (damage == $DAMAGE)
 begin
        ScanAt = ScanAt + INCR
        Scan Angle ScanAt

        If ($FOUND <> 0) Then If ($DISTANCE >= MINDIST) &
                Then Fire Weapon 1, Angle $ANGLE, Distance $DISTANCE
                Else Fire Weapon 1, Angle $ANGLE, Distance MINDIST
 end
```

If and when program execution continues on beyond this loop, the bot has been hit by something. So, the best course of action is to move out of the scan beam of whatever enemy is firing. Recall that Corner used a little routine that set his velocity equal to his negated position to move to position 0, 0. There is a generalization of this routine that moves you to any point in the Arena that you wish. I leave it to you to figure out just why this routine works the way it does. The target point is (x, y). The general routine is:

```
:MoveLoop
        Velocity (x - $XLOC), (y - $YLOC)
        If (x-$XLOC > 20) Then Goto MoveLoop
        If (x-$XLOC < -20) Then Goto MoveLoop
        If (y-$YLOC > 20) Then Goto MoveLoop
        If (y-$YLOC < -20) Then Goto MoveLoop
        Velocity 0, 0
```

To move there faster, you can multiply the velocity terms by some constant, like 2 or 3. Of course, if you do that your bot will also be more likely to overshoot the target point, unless it has really good brakes. The number “20” in the algorithm could really be any number. The larger the number, the faster and less accurate the movement will be.

So that the bot can run away immediately when it notices that it’s taken damage, we should have the velocities prepared ahead of time: it takes a few ticks to calculate 2*(x-$XLOC) and 2*(y-$YLOC), so we can save them in variables goX and goY. Also, we can write a quickie subroutine to compute this velocity, since we’ll need it more than once. At the top of the program, we put:

```
        Gosub CalcVel
        goX = goX * 20
        goY = goY * 20
```

The 20 will make sure that we accelerate a lot at first. There’s no need to be very accurate about where you go when you’re running for your life. At the bottom of the program, we can put the subroutine

```
:CalcVel
        goX = XX - $XLOC
        goY = YY - $YLOC
 Return
```

We still need to get XX and YY, the target location. A random location is often a good choice. We can put this in the same subroutine as the velocities, so that it gets computed in the same subroutine call as the initial velocities. Be sure that the point the bot chooses as its target isn’t too close to a wall, or it could easily smash itself.

```
:CalcPos
        XX = Random(215) + 20
        YY = Random(215) + 20
:CalcVel
        goX = XX - $XLOC
        goY = YY - $YLOC
 Return
```

Now the bot can do a Gosub CalcPos to compute a target point and initial velocity, and it can also do a Gosub CalcVel to just compute a new velocity for the same point. So the we can use this new subroutine to compute the velocities for the generic movement routine presented above. Here is the whole program. Try tweaking some of the position and speed constants to see what happens, and feel free to use bits of this code in making your own bots. This bot is called Bomber.

```
#DATA

EQU INCR    23
EQU MINDIST 25

DEF damage
DEF scanAt
DEF XX
DEF YY
DEF goX
DEF goY

#CODE BASIC

:Initialize
        Gosub CalcPos
        goX = goX * 20
        goY = goY * 20
        damage = $DAMAGE

        While (damage == $DAMAGE)
            begin
                ScanAt = ScanAt + INCR
                Scan Angle ScanAt

                If ($FOUND <> 0) Then If ($DISTANCE >= MINDIST) &
                    Then Fire Weapon 1, Angle $ANGLE, Distance $DISTANCE
                    Else Fire Weapon 1, Angle $ANGLE, Distance MINDIST
            end

        Goto Runaway

:RAway
        Gosub CalcVel
        goX = goX * 2                ! Move there twice as fast.
        goY = goY * 2
:Runaway
        Velocity goX, goY
        If (XX-$XLOC > 30) Then Goto RAway
        If (XX-$XLOC < -30) Then Goto RAway
        If (YY-$YLOC > 30) Then Goto RAway
        If (YY-$YLOC < -30) Then Goto RAway
        Velocity 0, 0
        Goto Initialize     ! Start all over again.


:CalcPos
        XX = Random(215) + 20
        YY = Random(215) + 20
:CalcVel
        goX = XX - $XLOC
        goY = YY - $YLOC
 Return

#END
```

One last point. You may have noticed that the scan increment for this bot is 23, much larger than it was for the previous two bots. Often, a large scan increment can be useful. It just so happens that with an increment of 23, you fill in the “cracks” in the scanner very nicely: The first loop goes 0, 23, 46, ... the second, 8, 31, 54, ... the third, 16, 39, 62. So using 23 is sort of like using 7 or 8, except that it takes three full 360-degree sweeps to see the whole Arena, rather than just one. A larger increment is also useful when scanning for bots that are fairly close to yours. Closer bots look “bigger” in the scanner and thus are easier to spot, so if your bot’s scan loop has a small increment, it may actually be (effectively) overlapping its scans when looking for close bots. For example, if an enemy is 350 pixels away, a scan increment of 7 might miss it on a given scan pass. However, if that same bot is 20 pixels away, it would be impossible to miss it with a 7 increment. In fact, it would be impossible to miss it with, say, an 11 increment. Therefore, larger scan increments may often be less wasteful than small ones.

There is another useful way to move. If you know the *angle* that you want your bot to move along, you can use trigonometric functions to calculate the velocity you want. Without going into a trig lecture, I'll just say that sin(angle) is the y-part of movement along an angle, and cos(angle) is the x-part. In 'bot, Cos and Sin return their values on a scale of -256 to 256, rather than from -1 to 1. This is perfect, coincidentally, for using in a velocity statement. A routine to move along the angle stored in the special variable $ANGLE would look like the following. Note that it doesn't stop until the bot hits something (which causes the bot to decelerate to a stop automatically).

```
        Velocity Cos($ANGLE), Sin($ANGLE)
:ALoop
        If ($XVEL <> 0) Then Goto ALoop
        If ($YVEL <> 0) Then Goto ALoop
```

Since collisions do damage, this might be a useful weapon! If the bot spots an enemy, it can move toward it at top speed and damage it. Unfortunately, it will also damage the attacking bot as well. However, our bot can put on a Shield (a force field) before striking the other bot. The Shield will absorb some of the damage of the collision, and thus, hopefully, the other bot will be hurt more than ours. To add shielding to this routine, we will add a Shield ON statement at the start and a Shield OFF statement (so the bot won't run down its battery) at the end. In the Architecture Editor, we need to add a Shield (in the "Protection" section) and a Battery (in the "Battery" section). The code now looks like this:

```
        Velocity Cos($ANGLE), Sin($ANGLE)
        Shield ON
:ALoop
        If ($XVEL <> 0) Then Goto ALoop
        If ($YVEL <> 0) Then Goto ALoop
        Shield OFF
```

"ON" and "OFF" here are just regular constants defined for convenience. If you'd rather, you could put 1 instead of ON and 0 instead of OFF.

One other kind of movement that a bot can easily do is movement *towards* a point (rather than movement *to* a point). This works just like the movement routine we used above for Bomber, but once the bot sets its initial velocity, it keeps going for a certain amount of time and then stops. This routine is a bit easier to write; it also gives the bot a quick burst of speed and then lets it continue seeking enemies after only, say, 30 ticks or so. The routine above that actually gets the bot to its target may well take longer than this to complete. Also, if that first routine happens to pick a point right near the bot's current location, the bot won't move at all! However, this new routine has the drawback that since you just set your bot's velocity and wait, your bot may well run into a wall while waiting. But that's what we'll use in this subroutine:

```
:Moveaway
	Velocity (xx-$XLOC)*10, (yy-$YLOC)*10
	Wait 20
	Velocity 0, 0
	damage = $DAMAGE
	xx = Random(200)+25
	yy = Random(200)+25
 Return
```

where (xx, yy) is the target point. It computes a new target point for the *next* movement phase immediately after it stops, and resets the variable that stores the damage value. All we need to do now is add a little initialization (like calculating initial values for xx & yy, and storing the value of $DAMAGE in damage), and we have a completed bot. Notice that this bot, called Suicide, has no weapon: the only damage it does is by collisions.

```
#DATA

EQU INCR 23

DEF scn
DEF xx
DEF yy
DEF damage

#CODE BASIC

	scn = Random(360)
	damage = $DAMAGE
	xx = Random(200)+25
	yy = Random(200)+25

	! The following few lines in fact make up the entire body
	! of this bot's code.  Everything else is done in
	! subroutines.
```

```
:Loop
        If (damage <> $DAMAGE) Then Gosub Moveaway
        scn = scn + INCR
        Scan Angle scn
        If ($FOUND == 0) Then Goto Loop

        Gosub Attack
        Gosub Moveaway
        Goto Loop

! Subroutines:

:Attack
        Velocity Cos($ANGLE), Sin($ANGLE)
        Shield ON
:ALoop
        If ($XVEL <> 0) Then Goto ALoop
        If ($YVEL <> 0) Then Goto ALoop
        Shield OFF
 Return


:Moveaway
        Velocity (xx-$XLOC)*10, (yy-$YLOC)*10
        Wait 20
        Velocity 0, 0
        damage = $DAMAGE
        xx = Random(200)+25
        yy = Random(200)+25
 Return

#END
```

Just as it is sometimes useful to move at an angle instead of toward a point, it may sometimes be useful to fire at a point rather than at an angle. When writing Suicide, we converted the given angle to its X and Y components using Sin and Cos functions. To convert the other way, we can use the ArcTan function. This function takes a Y value and an X value (Y always comes first for ArcTan), and returns the angle that you'd move along if those values were your velocity values. To get the angle to a point from your current location, then, you can use:

```
angle = ArcTan(y-$YLOC, x-$XLOC)
```

Notice that this is very similar to the way we set the Velocity in the move-to-a-point movement routine presented with Bomber (above).

If you've run combats with any of the Tutorial bots presented thus far, you've probably noticed that Corner always seems to win. This is because there is a great advantage to sitting in the corner of the Arena. However, this makes it much easier to find the bot, as long as its competitors are smart enough to check the corners once in a while. We'll make a fairly dumb bot that behaves a lot like Bomber, except that after it recovers from taking damage, it fires an energy blast into each of the four corners of the Arena.

The ArcTan instruction takes a fairly long time to execute, so use it sparingly if possible. Here, we'll add an extra check for damage after every shot, just in case an enemy locates our bot while it's calculating ArcTans. These checks are indented (below) just to make the program easier to read.

The energy weapon will be the bot's secondary weapon, not its primary one, so that the battery doesn't drain too much. Energy weapons strike the target immediately and take no time to reload, so they have some definite advantages over regular bullets. However, if the battery drains due to excessive firing, the weapon is useless until the battery has time to recharge. The code to fire into the four corners from the bot's current location is:

```
damage = $DAMAGE
Fire Weapon 2, Angle ArcTan(-$YLOC, -$XLOC)
    If (damage <> $DAMAGE) Then Goto Runaway
Fire Weapon 2, Angle ArcTan(-$YLOC, 256-$XLOC)
    If (damage <> $DAMAGE) Then Goto Runaway
Fire Weapon 2, Angle ArcTan(256-$YLOC, 256-$XLOC)
    If (damage <> $DAMAGE) Then Goto Runaway
Fire Weapon 2, Angle ArcTan(256-$YLOC, -$XLOC)
    If (damage <> $DAMAGE) Then Goto Runaway
```

Runaway is the same running routine used by Bomber. We can be a bit more intelligent about this. As I said above, if the battery charge is low, or if the battery has been destroyed, the weapon won't fire. If this happens, there's no point in wasting many, many ticks calculating angles that will never be used. So put in a quick check beforehand:

```
	If ($BATTERY == 0) Then Goto Initialize
```

Where Initialize is back at the top of the scan loop. $BATTERY is the special variable containing the number of durability points remaining for the Battery. If you want to add a check to see if the battery is too drained to fire, the battery's current charge is stored in $CHARGE, and you can write the code for this yourself as an exercise. Useful Architecture characteristics to consider for this are Battery Maximum Charge, Battery Recharge Rate (perhaps), and Energy Weapon Consumption. Notice that when you design an Energy Weapon, you set the weapon's *consumption*, not its damage. The amount of damage that it does to the enemy using this consumed energy depends on the Focus of the weapon. All of these can be set in the Architecture Editor.

Other than the above changes, this bot will be pretty much like Bomber was (except that it uses a regular gun instead of grenades). Here is the code for Zapper; notice that the scanning routine uses a double WHILE loop: one for scanning and one for firing at a scanned bot.

```
#DATA

EQU INCR    23
EQU MINDIST 25

DEF damage
DEF scanAt
DEF xx
DEF yy
DEF goX
DEF goY
DEF temp

#CODE BASIC

	Gosub CalcPos
	goX = goX * 20
	goY = goY * 20
```

```
:Initialize
        damage = $DAMAGE

        While (damage <> $DAMAGE)   ! Until I get hit
            begin
                scanAt = scanAt + INCR
                Scan Angle scanAt
                While ($FOUND <> 0)     ! Until I lose sight of enemy
                    begin
                        Fire Weapon 1, Angle $ANGLE
                        If (damage <> $DAMAGE) Then Goto Runaway
                                ! Jump out if damage is taken while locked
                        Scan Angle $ANGLE
                    end
            end

        Goto Runaway

:RAway
        Gosub CalcVel
        goX = goX * 2
        goY = goY * 2
:RUNAWAY
        VELOCITY GOX, GOY
        If (xx-$XLOC > 30) Then Goto RAway
        If (xx-$XLOC < -30) Then Goto RAway
        If (yy-$YLOC > 30) Then Goto RAway
        If (yy-$YLOC < -30) Then Goto RAway

        Velocity 0, 0

        Gosub CalcPos
        goX = goX * 20
        goY = goY * 20

        ! Before returning to the scan loop, zap.

        If ($BATTERY == 0) Then Goto Initialize

        damage = $DAMAGE
        Fire Weapon 2, Angle ArcTan(-$YLOC, -$XLOC)
                If (damage <> $DAMAGE) Then Goto Runaway
```

```
          Fire Weapon 2, Angle ArcTan(-$YLOC, 256-$XLOC)
                    If (damage <> $DAMAGE) Then Goto Runaway
          Fire Weapon 2, Angle ArcTan(256-$YLOC, 256-$XLOC)
                    If (damage <> $DAMAGE) Then Goto Runaway
          Fire Weapon 2, Angle ArcTan(256-$YLOC, -$XLOC)
                    If (damage <> $DAMAGE) Then Goto Runaway

          Goto Initialize     ! Start all over again.


:CalcPos
          xx = Random(215) + 20
          yy = Random(215) + 20
:CalcVel
          goX = xx - $XLOC
          goY = yy - $YLOC
 Return

#END
```

The last bot in this tutorial will combine a few of the strategies presented above. We want a bot that will run away when it gets hit, and that can recover intelligently if it collides with another bot while running. We want a bot that checks the corners occasionally. We want a bot that can efficiently scan the Arena.

This bot, called Middle, will sit in the center of the Arena and scan in a circle. Since no enemies can be more than about 180 pixels away from this center position, the bot won't even need a long-range scanner. Since all bots are fairly close to Middle, they all appear larger in its scanner. Thus, a wide scan increment can be used efficiently (see the discussion of this with Bomber, above). I'll present each chunk of Middle's code here. You will probably recognize many of the pieces from earlier bots. First, here is a routine to move to the middle of the Arena (location (128,128)). This routine is a bit different (actually, it's simpler) in that first it moves horizontally, then it moves vertically (rather than both at once). Also, it picks a random angle and fires at that angle while moving. Since it has the weapon, why not try to land a few "accidental" shots on the enemy?

```
:RunawayStart
        temp = Random(360)
        Fire Weapon 1, Angle temp
:RunawayX
        Velocity (128-$XLOC)*4, 0
        Fire Weapon 1
        If ($XLOC > 136) Then Goto RunawayX
        If ($XLOC < 120) Then Goto RunawayX
        Velocity 0, 0

:RunawayY
        Velocity 0, (128-$YLOC)*4
        Fire Weapon 1
        If ($YLOC > 136) Then Goto RunawayY
        If ($YLOC < 120) Then Goto RunawayY
        Velocity 0, 0
```

There is a problem with this set of routines. If the bot collides with a stationary (or moving) enemy, they could get "stuck." That is, the enemy might permanently block the chosen path to the center. To deal with this, we'll create a routine that backs away from the enemy, scans, and fires. It will be called from the above movement routines, so we really need both an X version and a Y version:

```
:XDamage
        d = $DAMAGE
:XTestDirection
        If ($XLOC > 130) Then              ! Are we moving left or right?
          begin
                Velocity 10, 0          ! (Back up slowly while firing)
                scn = 180
          end
        Else
          begin
                Velocity -10, 0
                scn = 0
          end
        Scan Angle scn          ! Scan along the x-axis
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn-30       ! Scan -30 degrees from the x-axis
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn+30       ! Scan +30 degrees from the x-axis
        If ($FOUND <> 0) Then Goto XQuickShot
        d = $DAMAGE
        Goto ContinueX

:XFireAt
        Scan Angle $ANGLE
        If ($FOUND == 0) Then
          begin
                d = $DAMAGE
                Goto ContinueX
          end
:XQuickShot
        Fire Weapon 1, Angle $ANGLE
        Goto XFireAt
```

Where ContinueX is a label back in the X movement routine above.

Notice that XFireAt is a routine of its own. A Goto XQuickShot command immediately fires at the recently scanned angle and then loops, scanning and firing at the last known angle to the enemy until the lock is lost. The Y equivalent of this code is not presented here, but it can be seen in the bot Middle if you want to look at it.

If we add a simple check for damage to the movement routines (the standard If (d <> $DAMAGE) Then…), then we have a problem. If Middle is hit by a bullet while moving, it will go into this collision routine. But since Middle is moving, it's fairly safe to assume that the attacking bot does not have a lock (as it's difficult to lock on a moving bot). So, the best thing to do would be to keep moving, not to stop and look for enemies. But how can you tell what caused the damage?

Under the "Sensors" heading in the Architecture Editor is a check box for Intelligent Damage Recognition (IDR) — Identify Cause. If this is selected, the bot's damage computer places a value into the special variable $DAMAGETYPE. This value is one of C_BULLET, C_GRENADE, C_ENERGY, and C_COLLIDE, which are constants defined for you to represent the type of damage last taken. So, we will add the following check right at the top of the backup/scan routine above:

```
        If ($DAMAGETYPE <> C_COLLIDE) Then Goto ContinueX
```

Now we know that there's really something out there to fire at. The X version of this whole routine, then, looks like the following:

```
:RunawayStart
        temp = Random(360)
        Fire Weapon 1, Angle temp
:RunawayX
        If (d <> $DAMAGE) Then Goto XDamage
:ContinueX
        Velocity (128-$XLOC)*4, 0
        Fire Weapon 1              ! Automatically fires at angle temp
        If ($XLOC > 136) Then Goto RunawayX
        If ($XLOC < 120) Then Goto RunawayX
        Velocity 0, 0


:XDamage
        d = $DAMAGE
        If ($DAMAGETYPE <> C_COLLIDE) Then Goto ContinueX

        If ($XLOC > 130) Then
            begin
                Velocity 10, 0
                scn = 180
            end
        Else
            begin
                Velocity -10, 0
                scn = 0
            end
        Scan Angle scn
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn-30
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn+30
        If ($FOUND <> 0) Then Goto XQuickShot
        d = $DAMAGE
        Goto ContinueX

:XFireAt
        Scan Angle $ANGLE
        If ($FOUND == 0) Then
          begin
                d = $DAMAGE
                Goto ContinueX
          end
:XQuickShot
        Fire Weapon 1, Angle $ANGLE
        Goto XFireAt
```

When the bot finishes executing this code and the Y version of this code, it is in the center of the Arena. Now we'll have the bot execute a fairly standard scan loop:

```
:Start
        d = $DAMAGE
        scn = Random(360)

:ScanLoop
        If ($DAMAGE <> d) Then Goto GotHit
        scn = scn + INCR
        Scan Angle scn
        If ($FOUND == 0) Then Goto ScanLoop
:ScanLock
        Fire Weapon 1, Angle $ANGLE
        If ($DAMAGE <> d) Then Goto GotHit
        Scan Angle $ANGLE
        If ($FOUND <> 0) Then Goto ScanLock

:LockOn                 ! Try to locate the moving enemy
        Scan Angle scn-43
        If ($FOUND <> 0) Then Goto ScanLock
        Scan Angle scn-43 + FIND_INCR
```

```
    If ($FOUND <> 0) Then Goto ScanLock

    If ($DAMAGE <> d) Then Goto GotHit

    Scan Angle scn-43 + FIND_INCR*2
    If ($FOUND <> 0) Then Goto ScanLock
    Scan Angle scn-43 + FIND_INCR*3
    If ($FOUND <> 0) Then Goto ScanLock

    Goto ScanLoop        ! Couldn't find him.
```

LockOn is a routine similar to that used by Corner: when the scanner loses the enemy, this routine does a quick check to try to find him again. Middle has INCR defined as 37, and FIND_INCR (for more careful searching) as 11. Remember that no efficiency is lost when you have a statement like Scan Angle scn-43 + FIND_INCR*3 : the compiler computes all of the constant calculations ahead of time, and it's the same (when FIND_INCR is 11) as using Scan Angle scn-10.

When damage is taken in this routine, it jumps to the GotHit routine. This routine will move the bot away from the center, and then bring it back to the center so that it can continue scanning. Since we already have a routine to move *to* the center, we need one to move away. This routine will use the Repair Mechanism to time its movement, rather than a Wait statement like Suicide Used.

The Repair mechanism, when activated by a Repair statement, takes control of your bot's computer for a certain amount of time (10 ticks * the number used in the Repair statement). At the end of this time, the specified system has a certain amount of its lost durability restored. See the section on Repairing in the main documentation for more details. There are constants provided for you for the names of the systems to be repaired. They are C_CPU, C_ENGINE, C_BATTERY, and C_TREADS. You can only repair a system that is checked in the Architecture Editor under the Maintenance section.

Here, instead of simply waiting for 20 ticks while we move, we will repair the CPU for 20 ticks. The following routine moves the bot away from the center by first moving it to the right and down, then up and to the left:

```
:GotHit
    Velocity 220, 70
    damageLoc = $DAMAGEDIR

    Repair C_CPU For 2     ! Repair the CPU for 20 = 10*2
    Velocity -70, -220
    Repair C_CPU For 2      ! Repair some more
    d = $DAMAGE
```

There is a problem with the repair mechanism, however. If the CPU is heavily damaged (fewer than half of its original durability remains), the repair mechanism might jam; that is, it might take a few extra ticks before it can successfully repair the specified system. With the above routine, the bot will occasionally smash into the right wall of the Arena, because it repairs for, say 36 ticks instead of 20. So, we need a check to see if the CPU is very damaged, and we'll write a second routine that uses a Wait instead of repairing. At the very top of the program, we'll add the line:

```
    cpuThresh = $CPU/2
```

Remember that $CPU is the current durability left for the CPU. Right at the beginning of the combat, it is the maximum durability. Now we can modify the move away routine as follows:

```
:GotHit
    Velocity 220, 70
    If ($CPU >= cpuThresh) Then
      begin
        Repair C_CPU For 2      ! Repair the CPU for 20
        Velocity -70, -220
        Repair C_CPU For 2      ! Repair some more
        d = $DAMAGE
      end
    Else
      begin
        Wait 20
        Velocity -40, -150
        Repair C_CPU For 2
        Velocity 0, 0
        d = $DAMAGE
      end
```

After executing the above code, the bot executes the Move-to-Center code we discussed above, and we have completed the damage control routine. Not only does it move the bot out of danger, but, if the CPU has been damaged, it repairs it.

Once we get back in the center, we want to find our attacker as quickly as possible. There is a second kind of IDR that we can use to do this: IDR (also under the Sensors section of the Architecture Editor) that identifies the direction that the damage came from. This is not perfectly accurate. Basically what it does is tell the bot which side the damage hit on: it returns either 0, 90, 180, or 270 as an angle approximation of where the damage hit in the special variable $DAMAGEDIR. We will add the line

```
     damageLoc = $DAMAGEDIR
```

to the top of the GotHit routine so that the bot will remember where it was hit (because if it gets hit while moving away, the old value in $DAMAGEDIR is lost). Once the bot gets back into the middle, we can execute a special section of code that sweeps the 90-degree area around the angle in damageLoc looking for the attacker. The bot will use a smaller increment (FIND_INCR) here so that it's less likely to miss the enemy on the sweep. If the attacker was in the corner, it will be included in this sweep:

```
     d = $DAMAGE
     For scn =    (Random(4)+damageLoc-47)            &
                  To   damageLoc + 43                 &
                  Step FIND_INCR
:FindScanLock
          If (d <> $DAMAGE) Then Goto GotHit
                                        ! Jump out of For-Next!
          Scan Angle scn
          If ($FOUND <> 0) Then
           begin
                  Fire Weapon 1, Angle $ANGLE
                  Goto FindScanLock      ! Still locked on
           end
      Next scn               ! No lock.

     Goto Start
```

It randomizes the angle slightly so that if it misses the enemy on one sweep and gets hit again, it will have a chance to get him on the next sweep. Once this routine finishes, the Goto Start statement starts the process all over again with the wide scan loop.

This completes the code for Middle. I'll give you the whole program below. I hope that this tutorial was useful for you, and that you now feel ready to go out and write bots of your own. Good luck!

```
#DATA

EQU INCR       37
EQU FIND_INCR   11

DEF scn
DEF d
DEF damageLoc
DEF temp
DEF cpuThresh
DEF firstTime 0

#CODE BASIC

        cpuThresh = $CPU/2
        d = $DAMAGE
        Goto RunawayStart   ! Move to center

:Start
        d = $DAMAGE
        scn = Random(360)

:ScanLoop
        If ($DAMAGE <> d) Then Goto GotHit
        scn = scn + INCR
        Scan Angle scn
        If ($FOUND == 0) Then Goto ScanLoop
:ScanLock
        Fire Weapon 1, Angle $ANGLE
        If ($DAMAGE <> d) Then Goto GotHit
        Scan Angle $ANGLE       ! Center the scanner on the enemy
        If ($FOUND <> 0) Then Goto ScanLock

:LockOn
        Scan Angle scn-43
        If ($FOUND <> 0) Then Goto ScanLock
        Scan Angle scn-43 + FIND_INCR
        If ($FOUND <> 0) Then Goto ScanLock

        If ($DAMAGE <> d) Then Goto GotHit

        Scan Angle scn-43 + FIND_INCR*2
        If ($FOUND <> 0) Then Goto ScanLock
        Scan Angle scn-43 + FIND_INCR*3
        If ($FOUND <> 0) Then Goto ScanLock
        Goto ScanLoop       ! Couldn't find him.


:GotHit
        Velocity 220, 70
        damageLoc = $DAMAGEDIR
        If ($CPU >= cpuThresh) Then
          begin
                Repair C_CPU For 2      ! Repair the CPU for 20
                Velocity -70, -220
                Repair C_CPU For 2      ! Repair some more
                d = $DAMAGE
          end
        Else
          begin
                Wait 20
                Velocity -40, -150
                Repair C_CPU For 2
                Velocity 0, 0
                d = $DAMAGE
          end


:RunawayStart
        temp = Random(360)
        Fire Weapon 1, Angle temp
:RunawayX
        If (d <> $DAMAGE) Then Goto XDamage
:ContinueX
        Velocity (128-$XLOC)*4, 0
        Fire Weapon 1
        If ($XLOC > 136) Then Goto RunawayX
        If ($XLOC < 120) Then Goto RunawayX
        Velocity 0, 0

:RunawayY
        If (d <> $DAMAGE) Then Goto YDamage
:ContinueY
        Velocity 0, (128-$YLOC)*4
        Fire Weapon 1
        If ($YLOC > 136) Then Goto RunawayY
        If ($YLOC < 120) Then Goto RunawayY
        Velocity 0, 0
        firstTime = firstTime + 1
        If (firstTime == 1) Then Goto Start
                                        ! Don't do the following routine
                                        ! the first time we move.
        d = $DAMAGE
        For scn =       (Random(4)+damageLoc-47)                &
                        To   damageLoc + 43                     &
                        Step FIND_INCR
:FindScanLock
                If (d <> $DAMAGE) Then Goto GotHit
                                                        ! Jump out of For-Next!
                Scan Angle scn
                If ($FOUND <> 0) Then
                  begin
                        Fire Weapon 1, Angle $ANGLE
                        Goto FindScanLock
                  end
          Next scn

        Goto Start


:XDamage
        d = $DAMAGE
        If ($DAMAGETYPE <> C_COLLIDE) Then Goto ContinueX
        If ($XLOC > 130) Then   ! Are we moving left or right?
          begin
                Velocity 10, 0
                scn = 180
          end
        Else
          begin
                Velocity -10, 0
                scn = 0
          end
        Scan Angle scn
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn-30
        If ($FOUND <> 0) Then Goto XQuickShot
        Scan Angle scn+30
        If ($FOUND <> 0) Then Goto XQuickShot
        d = $DAMAGE
        Goto ContinueX

:XFireAt
        Scan Angle $ANGLE
        If ($FOUND == 0) Then
          begin
                d = $DAMAGE
                Goto ContinueX
          end
:XQuickShot
        Fire Weapon 1, Angle $ANGLE
        Goto XFireAt


:YDamage
        d = $DAMAGE
        If ($DAMAGETYPE <> C_COLLIDE) Then Goto ContinueY                        If
($YLOC > 130) Then   ! Are we moving up or down?
          begin
                Velocity 0, 10
                scn = 270
          end
        Else
          begin
                Velocity 0, -10
                scn = 90
          end
        Scan Angle scn
        If ($FOUND <> 0) Then Goto YQuickShot
        Scan Angle scn-30
        If ($FOUND <> 0) Then Goto YQuickShot
        Scan Angle scn+30
        If ($FOUND <> 0) Then Goto YQuickShot
        d = $DAMAGE
        Goto ContinueY

:YFireAt
        Scan Angle $ANGLE
        If ($FOUND <> 0) Then
          begin
                d = $DAMAGE
                Goto ContinueY
          end
```

```
:YQuickShot
        Fire Weapon 1, Angle $ANGLE
        Goto YFireAt

#END
```