Combat Zone - The Manual

by Mat Peck
with lots of contributions by Pat Fox

"What? A computer game that doesn't involve abusing a joystick or mangling a mouse? One that requires more than 30 seconds to learn?! Take two bottles into the shower! It's into the bin with this foul satanic little diskette"

WAIT! Before you callously consign this floppy to the dusty graveyard of disks that exists somewhere in the gap between your stylish off-white mini-tower and that stack of unread DOS manuals, take a moment to consider what it is you might miss out on...I can, without fear of contradiction, state that this is the most fun, addictive, and yet most educationally valuable piece of leisure software that has ever graced a computer. Why? Well, in order to find out, you're going to have to read on...

1. The History

Way back in the mists of time, back when computers were all limited to a miserable 8 bits and when PCs were owned by just a privileged few (and weren't very good anyway), back when Mario Brothers were just another couple of game characters and Sonic the Hedgehog was something only LSD could produce; back when Lemmings didn't have blue hair and explode on command, back when I was but a young lad bombing up and down the road on my Chopper bike, making police car noises...a long time ago, okay.....Turtle Graphics first appeared. Some bright spark obviously figured that one way to teach people to program was to reward them with terribly attractive visible output from their program. This was a great success and was bought by schools to help teach kids, but failed to capture the imagination of many young, budding programmers because it lacked one vital ingredient; the same ingredient which makes games like Street Fighter II and Syndicate so popular. Mindless violence. Fascinating and character building as it is, programming a little wedge shaped turtle to draw impossible triangles on the screen, isn't quite as much fun as slam dancing a sumo wrestler onto your best friends head!

In 1984 I discovered a program called "Robot Arena", a game that a friend found on the school 480Zs (8-bit educational things made by Research Machines). The idea of the game was this; you (the player) would, using a simple, dedicated programming language (called ROBOL - an in-joke for all you programming freaks), write a program to drive a robot that would beat the shit out of any similar efforts written by your friends. When all the robots were programmed, they would be put into an arena together, and the last one left standing would be the winner. Thus our days became an endless round of programming, interspersed with foul language as our best programming efforts were pounded to death. Unconsciously we mastered the sacred black art of programming, when we actually thought we were having a good time - strange, huh?

Since then I have seen nothing similar on the market. The idea has, however, been festering in my brain, and when lovely old PC Format announced in March (last year!) that a series of programming articles were about to appear, I, nostalgic for long-past, more naive, zit-ridden times, set to work, creating a caring, sharing, Nineties version of this amazing package.....Combat Zone.

You're now thinking: "That's all very well, but I don't know how to program and I don't have the time or the patience to learn, and anyway, I don't believe that a game that takes so long to get started could possibly be any fun."

Well, it is true that, in order to play this game, you will need the following things:

1) an attention span of more than five seconds.
2) a willingness to learn a few simple new ideas.
3) moderately more brain power than your average MegaDrive owner.

if you don't, then you're going to miss out on the most fun that it's possible to have without actually exchanging bodily fluids.

For those of you who do wish to carry on with this odyssey into creative programming, I will attempt to explain in as simple terms as possible the principles behind programming, and its bearing on COMBAT ZONE....but first a general word about programming.

It doesn't matter which programming language you try to learn, you always seem to encounter one of two scenarios. Number 1, the "Boring Manual Syndrome", is where the hard-backed, 1000 page manual, explains the structure of the language in minute detail, including fascinating example programs that produce the twelve times table, whilst containing absolutely NO information about graphics and sound capabilities, which, most of the time, is the reason you bought the language in the first place.

Number 2 is the "Borland Manual Syndrome", which is a set of 25 manuals, all of which have the following chapter structure:

Chapter 1 : How to Open the Box with the Software inside.
Chapter 2 : How to turn on your machine and install the software.
Chapter 3 : Practical object oriented assembly language and it's use in LAN based 80486 systems.

Leaving you the relatively minor task of guessing everything in between!

Magazines try to overcome this inherent problem by running series after series of articles on how to program a particular style of game in some language or another, but there is only so much you can fit into one article, and the month delay between issues often causes frustration. How many times have you seen summary lines at the end of programming articles which run something like.....

> *"Now we've created the entire game structure, with it's plots and sub-plots, and have added copy protection and written the manual, next month we'll show you how to display the characters on the screen!"*

This may seem to you an extremely cynical view of the programming world, but in my experience it is all too true. THIS story, however, does not end here...this program and manual is the answer!

Where to now?, Well, the next section describes how to program a tank in ROBOL, hopefully without falling into either of the two categories mentioned above. Section 3 tells you how to operate the Editor/Compiler, and Section 4 gives instructions on how to load tanks into the Arena and watch them fight. Jump to Section 4 now and load in some of the example tanks supplied with this program. You've done far too much reading for the moment, time to take a break. After you've watched the example tanks and seen what you're up against, make yourself a strong cup of coffee and take six Pro-Plus tablets so you'll be able to stay awake whilst reading.....

2. The Dull Bit, or, How to Program in Three Easy Steps

So, what is it we're trying to do? As I said earlier, the idea of the game COMBAT ZONE is to program a robot tank that will be able to beat any other tank that anyone else has programmed. In order to do this, we need to know what it is that constitutes a program; put simply, a program is a list of instructions. What we, as programmers, have to do, is to work out what instructions to give our tank, in what order, so that it should be able to win. And in order to do *that*, we need to decide exactly what we want the tank to do!

Confused yet? You will be!

OK, first things first. As I just said, we need to work out what we want our tank to do, and then translate it into a language that the computer can understand; obviously, we cannot simply say to the computer "Find all enemy tanks and destroy them before they destroy you", since the computer does not understand English - if it did, all this would be much easier. What we have to do is to look at the sort of instructions that the computer *does* understand, and try to use them to achieve the same result.

When the world was new and people still considered there to be scientific merit in writing about such things, two unfeasibly named chaps, Böhm & Jacopini, produced their Structure Theorem, which stated that programming could be broken down into "Three Easy Steps" (and you though I was joking about that bit didn't you!).

(i) Composition - a set of instructions executed one after the other

(ii) Alternation - is best expressed as follows: "If something is true then do one thing, otherwise do something else". A real world example might be "If you have £3.95 in your pocket, buy PC Format, otherwise steal it!"

(iii) Repetition - is simply doing something zero or more times. I know that, strictly speaking doing something zero times is NOT actually doing it at all, and doing something once cannot technically be described as Repetition, but I didn't make these rules up so I'm not apologising for them! And anyway, it's not important, so let's concentrate on the real issues here shall we!

Fascinating stuff, but is this actually getting us anywhere? How do I shoot things? Someone give me some match sticks to prop my eyes open with.

Okay, bearing all this in mind, let's see how to program a tank....

i. Composition

What is a house? No it's not a trick question...it's actually a simile. A house is a thing made up of two floors and a roof. A floor, in turn, is made up of a collection of rooms. Each room has 4 walls, and each wall is built of bricks. In other words, a house is made of a collection of smaller objects, each of which is, in turn, made up of even smaller objects, until we get to the point, the brick, where we can get no smaller. Composition in computer programs is exactly that, only we don't use bricks, we use commands in our programming language. You can use bricks if you want, but it damages your monitor!

Combat Zone tanks are programmed in a very straightforward custom language called ROBOL, which has a few simple commands (like MOVE and FIRE) and most of the standard programming structures (such as loops, which are explained below). You type the programs into the Combat Zone Editor and COMPILE them into a language which the computer can understand, but you can't. You can then load these compiled robots into the Combat Zone Arena and set them fighting one another in an arena of your choice. The nice thing about the compiled robots is that whilst your friends can watch your robot fight, they cannot see the program you actually typed in, so you can swap your best robots with your friends without fear of them being able to see just how your robot pulverises them into the floor.

A tank will understand a small (but perfectly-formed) set of instructions (shown over the page); these are limited, since the tank only has a very limited task to fulfil. These COMMANDS are as follows: ("n" and "a" signify a number that you, the programmer, must specify - we'll get to that later)

MOVE n	Moves "n" pixels in the direction the tank is currently facing.
ROTATE a	Rotate the tank to face the direction "a". "a" is the angle to which you wish the tank to face, in degrees. Like a compass, 0 is facing directly up the screen, 90 is facing to the right, 180 is facing down etc. If you specify an angle greater than 360 degrees, the angle turned to is "a"-360.
TURRET a	Like ROTATE, but turns the gun turret to face in direction "a". The turret need not face in the same direction as the tank itself.
RADAR a	Performs a radar scan in direction "a". The FIRST object found is returned in the variable OBJECT, and it's distance from the tank in the variable DISTANCE. Don't worry about variables for the moment, they are explained later.
FIRE n	Fires a bullet in the direction in which the turret is facing, which explodes at a distance of "n" pixels from the tank. Only one bullet can be fired from a tank at any time, and no further bullets can be fired until the previous bullet has exploded.
SHIELD ON/OFF	Turns on, or off, the tank shield. The shield protects your tank from bullets and collisions, but you cannot use the radar, or fire when the shield is on, and you only have a limited amount of shield energy, whose reserves are stored in the ENERGY variable (see later).

So, this is the sum total (almost) of the tank's vocabulary. It is not terribly extensive, but it is sufficient to allow the tank to do its job. So now we can write a program, by *composing* (composition, remember?) these COMMANDS into a program:
E.g.

```
ROTATE 90
MOVE 50
TURRET 180
FIRE 100
```

This program turns the tank to face right, moves 50 pixels in that direction, rotates the gun to face down and then fires a bullet which explodes 100 pixels away from the tank. Wowee! It's not going to win any battles, that's for sure. It is very unlikely indeed that this tank will even hit anything, never mind actually win a game. What we need is a tank that reacts to the circumstances of the game, and acts accordingly - for instance, it would use its radar to find an enemy tank, move towards it, point its turret in the correct direction, and blast it to pieces. In order to do this, we need our second step - *alternation* - which allows us to do one thing or another, depending on what is happening.

This is all very well, but even if it can react to its situation, this tank will not win, because after it has executed each of its instructions once, it will simply find it has nothing else to do and die because it no longer has a reason to live. How sad. So, a tank that only does things once is not much use, really. We could write a very very long program that has all the instructions lots of times, or alternatively, to save us a lot of typing, we could simply tell the tank to repeat the same set of instructions more than once. Hey - we could call that *repetition*!

Still not particularly flexible though is it? What if I don't want to fire 100 pixels away every time? What was all that stuff about variables? How long is a piece of string?
Slow down! You are right though, we do need to examine a couple of vital concepts that will make programming a much happier, more fulfilling experience: *variables* and *expressions*. No! don't crawl into a corner mumbling "Begone, foul jargon fiend!", this won't hurt, and it is very necessary if you are to be able to truly slaughter your mates when they come round with their feeble efforts...

You could think of a *variable* as a labelled box that contains a number. The label (the name of the variable) represents the value that is held in the "box", and can be treated exactly as if it were that value - for instance we could multiply it, divide it and so on. So, we could call a variable "LIFE", and give (or *assign*, in jargon-speak) it the value 42, and then incorporate it into a program. To assign values to variables, you use the equals sign "=":

LIFE = 42 (assigns value 42 to variable "LIFE")
TURRET *LIFE*

and the tank will know that the label "LIFE" represents the number 42, so it will rotate its turret to heading 42 degrees.

But these things are called *variables* because their value can *vary*: at any time we can change the value of "LIFE" to be something else, like 100 or 25 or whatever. Is this any use? Most certainly! Let's look at an example. Here's one I made earlier:

> Let's say we want our tank to scan the whole of the arena with its radar, in increments of five degrees. We could do this by writing a program that specifies *every* angle to look at:
>
> RADAR 0
> RADAR 5
> RADAR 10
> RADAR 15...and so on...
> ...until... RADAR 350
> RADAR 355
>
> but this would be very time-consuming and not very clever. What we could do instead is to use a *variable* in place of the number - let's call it "LOOK" -and simply change its value, like this:
>
> RADAR *LOOK*
> *LOOK* = *LOOK* + 5 (increases variable "LOOK" by 5)
>
> and get it to repeat this operation until it has scanned the whole arena. Ingenious, huh?

We have just seen an *expression*, and we didn't even know it - it didn't hurt, did it? The line "*LOOK* = *LOOK* + 5" is an *expression* - or, in layman's terms, a sum. In effect, an *expression* simply works out the result of a particular sum - it can be addition, subtraction, multiplication, division (and a couple of other things). It allows you to manipulate numbers (and variables, since they can represent numbers), and then you can make decisions based on the results. You can use brackets in expressions, so all of the following are perfectly valid:

8 + 8
1 + (2 *4-3)
(20/2)*FOOBAR+15
(MONTY+PYTHON+LIFE+BRIAN)/4

Expressions can contain the following operators (the bits that actually do things, like plus and minus):

A + B	Adds the value A to the value B
A - B	Subtracts B from A
A * B	Multiplies B and A together
A / B	Whole number divides A by B, so 5/2 gives the number 2, not 2.5
A MOD B	Gives the remainder of dividing A by B, so 5 MOD 2 gives 1.
RND A	Returns a random number between 0 and (A-1).

The operators have different priorities, so, for example, multiply is more important than plus, just like on a standard calculator. Thus: 1+2*3 gives the answer 7 (i.e. 1 + (2 * 3)), not the answer 9 (which would be (1+2) *3). Using brackets is good programming style, because it makes expressions clearer and less ambiguous; you can be certain that the computer will evaluate your expression in the order that you want it to. And remember, since numeric variables represent numbers, they can be used in expressions just like any other number.

.

Back to variables a second....Can use any variable name I want, and assign any value to it?
Not quite. Here's some hard and fast rules:

1. Every separate variable has to have a different name.
2. All variable names must start with a letter of the alphabet, and can only contains numbers, letters and underscores ("_"), but NOT spaces.
3. Variable names must be less than 30 characters long.
4. The maximum value a variable can hold is 32767, and the minimum value it can hold is -32768.
5. You cannot have variable names which are also COMMANDS (e.g. FIRE or ROTATE etc. - see above) or names which are the same as SYSTEM VARIABLES (see below)

Okay, assume for the moment that I'm still following all this. What about these special variables like OBJECT and DISTANCE that you mentioned above?
Earlier, in the bit about COMMANDS, I mentioned under "RADAR" that there were two variables called "OBJECT" and "DISTANCE". These are *system variables*, that is, they are variables, in that they can hold different values, but they are assigned by the computer, not you. You can use them in expressions or whatever, but you cannot assign values to them. The system variables are listed below and over the page. The set below are continuously updated to contain the correct information. Those over the page are only set when a RADAR command is executed:

TANKX	The X co-ordinate of your tank.
TANKY	The Y co-ordinate of your tank.
MAXX	The width of the Arena in pixels.
MAXY	The height of the Arena in pixels.
HEADING	The angle in which the tank is pointing.
GUNHEADING	The angle at which the turret is pointing.
DAMAGE	The amount of damage you have sustained.
MAXDAMAGE	The maximum damage you're tank can suffer before it is killed.
ENERGY	The amount of shield energy you have left(in seconds) .
TANKS	The number of tanks currently alive in the Arena.
ACTIVE	This variable is set to 1 if the turret is capable of firing (i.e. the previous bullet has exploded) or is set to 0 if you cannot currently fire (because you've already fired your bullet and it hasn't exploded yet).
EXPLODE	The radius of an explosion caused by a bullet. Often useful in calculations to prevent your tank from blasting itself by accident.

The last three system variables are filled in by the tank radar. Whenever you call RADAR command, the values of these variables are set.

OBJECT	This variable is set to one of the following values:	
	WALL	Returned if the radar beam hits a wall
	TANK	Returned if the radar beam hits another tank
	MISSILE	Returned if the radar beam finds a bullet or explosion
	HOLE	Returned if a hole is encountered. Driving over a hole causes the tank to plummet to its immediate death. Avoid them at all costs.
	WARP	Driving onto a warp square teleports you to a random position in the arena. It's a good way to escape pursuers. You may, however, end up teleported on top of another tank so use them at your own risk.
	GATE	If the object is a Gate, the OPEN variable is also set (see below). If a gate is closed, shooting it will open it. Shooting an open gate closes it. You must score a direct hit on the gate to open/close it. The explosion itself will not effect the open/closed status of the gate.
	ROUGH	Tanks move more slowly over rough ground. In most circumstances the radar will not reflect from rough ground, but if you set the option in the Arena program (see later), you can make the radar return this value.
	BONUS	Bonus squares should be driven over to boost your tank. When you drive over a bonus square either nothing happens, or your damage may be halved, or your shield energy doubled, with equal frequency.
DISTANCE	Holds the distance in pixels of the object encountered by the radar.	
OPEN	If the object is a Gate, then this value is set to 1 if the gate is open, 0 if it is closed. Do not rely on the value of this variable if the object is not a gate.	

Yes, yes, I'm sure this is all very well, but it's getting a bit boring. How do I use all this information?
Like any variable, you can use the system variables in calculations and expressions. For example, a program block may look like this:

```
TURRET FRED
RADAR GUNHEADING
FIRE DISTANCE
```

This program turns the gun turret to the direction contained in the variable FRED and performs a radar scan in the direction the gun is now facing. It then fires at the object encountered by the radar scan, making the bullet explode on the exact spot that the object was found.

You cannot, however, assign values to the system variables with commands like:

```
GUNHEADING=20
```

That's all there is to expressions, variables and commands. It isn't too difficult when you get used to it. In order to be able to make decisions based on this information these variables give us, however, we are going to have to proceed to the next step in our guide to programming, which is...

ii. Alternation

Alternation is a technical term meaning, in effect, "decision-making": using certain instructions, we can allow the tank to "think" about what is happening in the game, and act accordingly. We've already seen how there are lots of ways of getting information about things like your tank's position, where the other tanks are, how many are left and so on. So all we have to do is go about deciding how we will use that information to our advantage.

The answer is that we use a construct called an *IF...THEN...ENDIF* structure. Basically, we use this structure to get the computer to execute an instruction only if a given condition is true. Eh? All right, an example, to make things clearer:

e.g. let's say we want the tank to use its radar, and then, if it detects another tank, to fire a bullet. First we must use the radar, so:-

```
RADAR ANGLE
```

looks in the direction contained in the variable "ANGLE", and sets the variables OBJECT and DISTANCE according to what it sees. Now we must act according to the result:

```
IF OBJECT = TANK THEN
        TURRET ANGLE
        FIRE DISTANCE
ENDIF
```

The "OBJECT = TANK" in the top line is a *condition*: the computer must decide whether or not this condition is true (the object that the radar saw was a tank), or false (it was something else): for jargon-freaks, this is known as a BOOLEAN expression, meaning that it can only be true or false. If it is true, then it will execute all the instructions that fall between the word "THEN", and the word "ENDIF" - in this case, the lines "TURRET ANGLE" (which turns the gun to face the tank detected), and "FIRE DISTANCE" (which fires a bullet towards the tank to explode at the correct distance). So the structure could be summarised as follows:

```
IF {a condition is true} THEN
        {do these commands}
ENDIF
```

There is another structure that allows you to choose between two different courses of action, depending on one condition: it is the *IF...THEN...ELSE...ENDIF* construction. It works the same way, but now, if the condition is found to be false, it will execute a second set of instructions instead of the first. For example:

```
RADAR ANGLE
IF OBJECT = TANK THEN
        TURRET ANGLE
        FIRE DISTANCE     (if the condition is true)
ELSE
        ANGLE = ANGLE + 5 (if the condition is false)
ENDIF
```

This time, if the condition is true, then the computer executes the instructions that come between the words "THEN" and ELSE" - as in the first example, if the radar detects a tank it will turn in the correct direction and fire a bullet. However, if the condition is false (it is not a tank), then it will execute the second set of instructions instead - the ones that fall in between the lines "ELSE" and "ENDIF" - in this case, it adds five to the variable ANGLE (we'll see later how this kind of construction, when combined with repetition, is an effective way to scout out enemies!). So we can summarise the construction thus:

```
IF {a condition is met} THEN
        {do something special}
ELSE
        {do something else}
ENDIF
```

You can have any *Composition* of valid program code between the THEN and the ENDIF, or the THEN and ELSE and ENDIF. The {do something special} and {do something else} need not, therefore, be one instruction, they can be Compositions, Repetitions and even other Aternations. For example:

```
RADAR MYANGLE                   - Scan in the direction MYANGLE
IF OBJECT=TANK THEN             - Did we spot a tank
        TURRET MYANGLE          - YES...so turn the gun towards it..
        FIRE DISTANCE           -..and fire at it
ELSE
        IF OBJECT=MISSILE THEN      - We didn't spot a tank, what about a missile
                SHIELD ON           - YIPES, there is one, so turn on the shield
        ELSE
                MYANGLE=MYANGLE+5   - Neither tank or missile so get next angle
        ENDIF
ENDIF
```

Placing Alternations inside other alternation structures, as we've done above, is called NESTING, and no, I don't have the faintest idea why! It might have something to do with birds, or ants, or wasps for all I know. Go ask someone else!

NOTE that indenting the program in the manner shown above is not actually necessary, but it helps to illustrate which blocks are made up of which others. It is good programming style to do this. You are much less likely to make mistakes if your program is properly indented.

What a load of old Boolean!
BOOLEAN expressions like the {condition} are unlike other numerical expressions discussed earlier because they do not use the standard addition and subtraction. They have there own set of boolean operators. These are:

A = B	Returns true if A and B are equal
A<>B	Returns true if A does not equal B
A > B	Returns true if A is greater than B
A < B	Returns true if A is less than B
A >= B	Returns true if A is greater than or equal to B
A <= B	Returns true if A is less than or equal to B

Such expressions can also be joined together to produce more complex sets of conditions using:

AND Returns true only if the boolean expressions on both sides return true
e.g. IF (OBJECT=TANK) AND (ACTIVE=1) THEN
FIRE DISTANCE
ENDIF
In this case, the tank only fires if it spots another tank AND it's turret is active.

OR Returns true if either of the boolean expressions on either side is true.
e.g. IF (OBJECT=TANK) OR (OBJECT=GATE) THEN
FIRE DISTANCE
ENDIF
This tank fires if it spots another tank OR a gate.

NOT If the boolean expression associated with it evaluates to false, NOT return true, but if it is true, NOT returns false.
IF NOT(OBJECT=WALL) THEN
FIRE DISTANCE
ENDIF
This tank would fire at anything that was NOT a wall.

ALWAYS REMEMBER TO PUT BRACKETS AROUND THE SEPARATE PARTS OF A BOOLEAN EXPRESSION...e.g. "IF (OBJECT=TANK) AND (DISTANCE>50) THEN" ....BECAUSE THE BOOLEAN OPERATORS LIKE = AND >= HAVE LOW PRIORITIES.

Confused aren't you? I told you that you'd be confused! Let's look at some more specific examples and see if that makes it any clearer.

An example of the "AND" option
We want our tank to look for an enemy tank, and then, if it is able (remember, only one bullet may be fired at a time), to shoot at it, but only if the enemy tank is further away than the radius of the explosion (there's no point in shooting ourselves is there!). So, after our obligatory RADAR line...

```
RADAR ANGLE
```

...we must tell the computer our conditions...

```
IF (OBJECT = TANK)
  AND (ACTIVE = 1)
  AND (DISTANCE>EXPLODE) THEN
```

...so, if it sees a tank (OBJECT = TANK) and the gun is currently ready to fire (ACTIVE = 1) and the enemy if far enough away (DISTANCE>EXPLODE), then it will execute our command(s)...

```
    TURRET ANGLE
    FIRE DISTANCE
ENDIF
```

...that is, it will turn its gun to face the enemy tank and fire. Go tank!

An example of the OR option
We want to protect our tank from being destroyed, so when either (a) its shield ENERGY is low or (b) its DAMAGE (amount of hits it can withstand before being destroyed) is higher than the Maximum damage (MAXDAMAGE) minus 5 hits, we want it to run away. So we give it our conditions...

```
IF (ENERGY <= 5) OR (DAMAGE >=(MAXDAMAGE-5)) THEN
```

...and then, if it finds either condition (or both) to be true, it will execute the command(s) that follow...

```
        ROTATE (HEADING + 180)
        MOVE 100
ENDIF
```

...OK, it's not a very good evasive manoeuvre, I must admit - the tank will turn to face the opposite way and move 100 pixels forward regardless of its position or any obstructions that might lie in its way - but it'll do as an example.

So, have you got that? It may sound complicated, it's always difficult to explain this kind of thing on paper without pages and pages of examples, but once you get the hang of the idea, you'll be alternating with the best of them (?). If you really don't get it, have a rest and read it again later. Or alternatively, have a look at the example code that is included with the game for examples of actual usage of these constructions.

Um...okay, I might get the hang of it with a bit of practice. You were saying about the other easy (ahem) step....
Finally, before you fall asleep completely, there is one more type of building block to go....

iii. Repetition

And now, the final tool in the programmer's repertoire - repetition. As you will no doubt have guessed, this allows us to repeat our commands to avoid ridiculously long programs. It's sensible, too, since the sort of tasks that we are likely to give the tank are liable to be of a simple, repeatable nature: scanning the arena at five-degree intervals for other tanks, firing repeatedly until the enemy tank is destroyed, and so on. There are a couple of constructs that we can use in order to repeat an instruction or set of instructions:

The first Repetition construct you will most likely use in almost every program. It is REPEAT...FOREVER. Anything placed between these two commands is repeated until the end of time, or until you stop the program, whichever is the sooner. Almost all tank programs will need to use this, since it will prevent the tank from simply "running out of program", and sitting there looking stupid. For example,

```
REPEAT
        (your program)
FOREVER
```

...is pretty obvious, really.

The program immediately over the page provides a further example, and would run quite happily in an arena. It checks for tanks and missiles. If it finds neither it checks at an angle 5 degrees greater than the last. If it finds a tank it fires at it and then checks to see if it's still there. If it is, it fires again, and so on, until the tank is dead. Spotting a missile causes it to put its shield up.

```
MYANGLE=0
REPEAT
        SHIELD OFF                      - Can't scan with the shield up so turn it off
        RADAR MYANGLE                   - Scan in the direction MYANGLE
        IF OBJECT=TANK THEN             - Did we spot a tank
                TURRET MYANGLE          - YES...so turn the gun towards it..
                FIRE DISTANCE           -..and fire at it
        ELSE
                IF OBJECT=MISSILE THEN      - We didn't spot a tank, what about a missile
                        SHIELD ON           - YIPES, there is one, so turn on the shield
                ELSE
                        MYANGLE=MYANGLE+5   - Neither tank or missile so get next angle
                ENDIF
        ENDIF
FOREVER                                 - Do this until the end of time!
```

Sometimes you don't want to repeat a set of instructions forever. Sometimes you only want to repeat them until a certain condition is true, then you want to go on a do something else. In this case, you use the REPEAT...UNTIL {condition} structure. Remember, a condition is an expression that evaluates to true or false. It takes the same format as the condition in an IF...THEN statement. The tank program below illustrates this by wandering around aimlessly with it's shield up until it runs out of energy, effectively letting the other tanks start to kill each other off, before it starts hunting for them.

```
REPEAT
        ROTATE RND(360)                 - Turn the tank in a random direction
        SHIELD OFF                      - Drop the shield quickly so we can...
        RADAR HEADING                   -...scan in the direction the tank is facing.
        SHIELD ON                       - Now raise the shield again
        MOVE DISTANCE-20                - And move almost up to the nearest object
UNTIL ENERGY=0                          - Keep doing so until we run out of shield energy.

MYANGLE=0
REPEAT
        RADAR MYANGLE                   - Scan in the direction MYANGLE
        IF OBJECT=TANK THEN             - Did we spot a tank
                TURRET MYANGLE          - YES...so turn the gun towards it..
                FIRE DISTANCE           -..and fire at it
        ELSE
                IF OBJECT=MISSILE THEN      - We didn't spot a tank, what about a missile
                        MOVE DISTANCE       - YIPES, there is one, so run towards it
                ELSE
                        MYANGLE=MYANGLE+5   - Neither tank or missile so get next angle
                ENDIF
        ENDIF
FOREVER                                 - Keep doing this forever
```

The last repetition construct is useful if you want to perform the same set of actions a certain numbers of times before continuing. For example, you may wish to fire five times at the same spot if you find a tank there. This is achieved using the DO...LOOP structure. This takes the form:

```
DO {number of times}
        {some set of actions}
LOOP
```

Therefore, firing five times at a tank could be achieved using the code:

```
RADAR MYANGLE                   - Check for a tank at this angle
IF OBJECT=TANK THEN             - If we've found one then
        TURRET MYANGLE          - Turn the gun on it..
        DO   5                  - ..and repeat the following five times:
                FIRE DISTANCE   - Fire at the tank
                REPEAT          - Do nothing until the turret is ready to fire.
                UNTIL ACTIVE=0
        LOOP
ENDIF
```

The {number of times} can be a number, a variable or an expression (remember them!) e.g. DO (5+RND(3)) is a perfectly valid instruction, which would repeat the sub-program between 5 and 7 times.

You're nearly there now, don't fall asleep on me just yet... this is the last part...honest.

iv) Advanced Programming Methods - Labels and Good and Bad Programming

This part is for those who feel up to creating programs of a slightly more complex nature. This section is something you can worry about later. It isn't strictly necessary for you to read it and you don't have to use anything here if you don't want to. If this is the first time you're reading this manual, and you haven't programmed before, skip the next three pages.

What if you want to do the same set of things at several places in the program, but don't want to type the same set of commands in numerous times?
I'm glad you asked me that at this point! You use SUBROUTINES.

Subroutines are areas of code which can be called at any other point in the program, which execute like a normal program, and then return to the point from which they were called. For example, suppose your tank was of the type that chased after other tanks when it spotted them, firing every so often. You might want to make the chasing and firing routine a subroutine, so that you don't have it type if out several times. Here's how you'd do it. Firstly you'd give the subroutine a name. We'll call it "CHASE". Then, whenever you want to use the CHASE routine, you call it using the GOSUB command from your main program. For example

```
IF OBJECT=TANK THEN
        GOSUB CHASE
        SHIELD ON
ENDIF
```

After executing the commands in the CHASE subroutine, the program would jump back to the instruction immediately after the GOSUB command, in this case the SHIELD ON command. This means that you can call the same subroutine from more than one place in the program, and it will always return to the next instruction, immediately after the GOSUB. This can lead to neater and shorter programs.

You'd define your CHASE subroutine as follows:

```
LABEL CHASE
        {the chase instructions}
RETURN
```

The LABEL command specifies a name for that particular part of the program. A label name, like a variable name, contains only alphanumeric characters, must begin with a letter, and be less than 30 characters long. You can have the same names for labels and variables if you want, but it is best to avoid doing so, if you don't want to get confused.

You should ensure that your tank only ever executes the RETURN command when the subroutine has been GOSUBed. If you let the program run into the subroutine other than by using GOSUB, the tank will die of confusion when it hits the RETURN command so it's best to place subroutines outside the main REPEAT...FOREVER loop. Your complete program may look something like the one shown over the page:

```
MYANGLE=0
REPEAT
        RADAR MYANGLE                   - Check in direction MYANGLE
        IF OBJECT=TANK THEN             - If we spot a tank then...
                TURRET MYANGLE          - ...Turn the gun towards it and ...
                GOSUB CHASE             - ...and CHASE it
        ELSE
                IF OBJECT=MISSILE THEN  - Okay it's not a tank..is it a missile?
                        DO  DISTANCE/5  -YIPES.  SHIELD on until its gone
                                SHIELD ON
                        LOOP
                        ROTATE MYANGLE  - Point the tank the correct way..
                        TURRET MYANGLE  - Point the gun the same way..
                        SHIELD OFF      - Turn off the shield so we can...
                        RADAR MYANGLE   -...scan for the nearest object
                        GOSUB CHASE     - Then chase after it
                ELSE
                        MYANGLE=MYANGLE+5  -Just increment the scan angle
                ENDIF
        ENDIF
FOREVER

LABEL CHASE                             - The subroutine CHASE
        FIRE DISTANCE                   - Fire at the object
        SHIELD ON                       - Turn on the shield and...
        MOVE (DISTANCE-EXPLODE)-5       - .. charge at the object
        REPEAT
                SHIELD OFF              - Now turn off the shield
                RADAR GUNHEADING        - Scan in the gun direction
                IF OBJECT=TANK THEN     - Is it a tank?
                        FIRE DISTANCE   - YEP, so blast it
                        REPEAT
                                SHIELD ON   - Put up the shield and keep it up..
                        UNTIL ACTIVE=1      -..until the explosion has died away
                ENDIF
        UNTIL OBJECT<>TANK          - Keep going until the tank is dead or gone
RETURN                              - Lost the target so go back to main program
```

Using subroutines is "Good" programming style. But you can also use the LABEL function with the GOTO command, something which is frowned upon in structured programming circles. Basically, it allows you to name certain sections of your program, using the LABEL command, and to jump to that position in the code using the GOTO command. This is NOT like GOSUB, which returns to the next instruction immediately after the GOSUB call when a RETURN is reached. GOTO simply jumps to the part of the program with that LABEL, never to return.

Personally I don't think there is anything wrong with this type of programming. At machine code level, this sort of thing happens all the time, in fact, if it didn't, none of the high level "structured" languages like PASCAL and C would work at all. Over the page is an example of the GOTO statement, and it's use with the LABEL command:

```
MYANGLE=0
OLDDAMAGE=0                 - Declare a variable to keep track of our damage level
REPEAT
        LABEL START                 - Label this part of the program START
        RADAR MYANGLE               - Scan for nasties
        IF OBJECT=TANK THEN         - Did we spot a tank
                TURRET MYANGLE      - YES, so rotate the gun
                FIRE DISTANCE       -..and blow it to pieces
        ELSE
                MYANGLE=MYANGLE+5       - No tank, increment our scan angle
        ENDIF
        IF DAMAGE<>OLDDAMAGE            -Have we been hit?
                GOTO GET_OUTA_HERE      -YIPES, we have, jump to
                                          the run away code
        ENDIF
FOREVER

LABEL GET_OUTA_HERE             - We jump in here when we're hit.
SHIELD ON                       - Shields up
OLDDAMAGE=DAMAGE                - Keep a track of our new damage level
ROTATE RND(360)                 - Turn in a random direction
SHIELD OFF                      - Shields down quickly so we can...
RADAR HEADING                   -.. radar in the direction we're facing
SHIELD ON                       - Shields up again
MOVE DISTANCE                   - Move as far as we can in that direction
SHIELD OFF                      - Panic over, we hope.  Shield down
GOTO START                      - Jump back to the main program loop.
```

v) We've Made It.....It's all over.....Yippee!

Well, that's all there is to know about the ROBOL language. It is quite a simple language really, although that's easy for me to say since I wrote it, and it may only seem about as clear to you as an Japanese MFI wardrobe self assembly manual.....

Let's just say, for the sake of argument, that....well I have this friend who doesn't quite understand some of the stuff you explained above and....I was just wondering if...

Don't worry if some of the programming described above seems a little complex to you now. It will become clear when you start to program some tanks. You cannot crash your machine by typing a program incorrectly. Most errors are picked up when you try to compile the program. The compiler will tell you what you did wrong, flashing up an error box with text like "IF without matching ENDIF", which means you forgot to put an ENDIF command after the IF..THEN statement (see the next section for more details). If your tank simply dies in the Arena for no reason, then a run-time of error has occurred, and you should look back at your code to try to see why.

To start with, write simple programs and build them into more complex robots one step at a time. Below is a simple starting block for your tanks. This is supplied with the program, so you can load it into your editor and build in your own routines. It is called BLOCK.ROB.

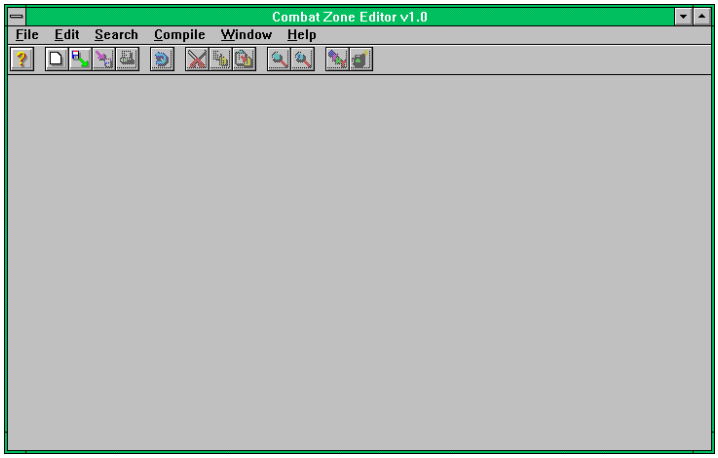
```
MYANGLE=RND(360)                    - Pick a random starting angle
REPEAT
        RADAR(MYANGLE)              - Scan for objects
        IF OBJECT=TANK THEN         - Is there a tank?
                TURRET MYANGLE      - YEP, rotate the gun
                FIRE DISTANCE       - and BLAST
        ELSE
                MYANGLE=MYANGLE+5   - Otherwise just increment the scan angle
        ENDIF
FOREVER                             - Do this forever.
```

There are three example robots supplied with the program. Two of which, AIMLESS and TRIGGER, can be loaded into the Editor, so you can see how they work. You only have the compiled form of the SMART robot, however, so you cannot see how it works. You'll know you're doing well, when your tank can beat it most of the time.

Time for a break, or as Lloyd Grossman would put it, time to Cogitate, Digest and Deliberate. Have another cup of coffee. Play a game. Clean your room....well, maybe not. Anyway, when you feel better and have forgotten everything you've just read, load up the Combat Zone Editor, read the next section, and muck about with some programs, working back through The Dull Bit to refresh your memory as you go. It's easy when you finally get the hang of it, and once you've learned the principles, you'll never forget how to do it. It's just like riding a bike really. Except you can't fall off a program and hurt yourself. Or get a puncture. Or get free Frosties reflective program stickers. Or go down to the shops on a program and get a packet of *Refreshers*. Okay, it's nothing like riding a bike, I'm sorry, just forget all about bikes. I could do with some *Refreshers*, though.

3. The Editor/Compiler

Now that you know how to program, this section describes how to use the Editor/Compiler to turn your source code into a program that the computer can understand. Everything you enter in the Editor is just in plain ASCII format, so you can use any editor (e.g. Windows Notepad, DOS Edit etc.) to enter the text, if you feel more comfortable using an editor you're already familiar with. In order to Compile that code, however, you'll have to load it into the Combat Zone Editor (which I'll call CZEdit from now on). To help explain how to use CZEdit, we'll create an example robot as we go along. In *Italics* at the end of each sub-section, I'll run through the actions we need to perform to create this robot.



When you load the editor, you'll be presented with the screen shown above. The only difference between this window and most standard windows applications is the "Toolbar", which is the line of buttons shown directly beneath the menus. These buttons are shortcuts to the most frequently used menu options, and a complete list of these buttons and their meanings is shown on the next page. Not all menu items have associated toolbar buttons.

The Editor is a Multi-Document Editor, which means that you can have more than one text file open at once, but each file will be in it's own window which is contained inside the main window. These editor windows can be opened, closed, resized and even iconised, but can never be moved outside the main window. This type of editor allows you to easily view two or more programs at once, and makes it easier for you to copy sections of source code from one program to another.

Before you can enter or edit a program, you must either load a previously saved file (by selecting **Open...** from the **File** menu, or by clicking the load button , or pressing F3) or start a new one (by selecting **New** from the **File** menu, or clicking the new toolbar button). Since we don't have anything to load yet, we'll start a new source code file.

*To create a new robot, we'll first open a new editor window. So click the New button ( ) or select **New** from the **File** menu. This will bring up a new window called "Untitled" in the main window. We'll type our program in this window.*

The Toolbar Buttons

 **Help** - Brings up the on line help. Only available with the registered version of the program

 **New** - Opens a new editor window for you to edit your programs in

 **Load** - Loads a saved source code file from disk (will not load Compiled robots)

 **Save** - Saves the source code to disk. This does not compile your robot

 **Print** - Prints the contents of the current editor window on the selected printer

 **Undo** - Undoes the last editing action you performed

 **Cut** - Cuts the highlighted section of text from the code and places it in the clipboard

 **Copy** - Copies the highlighted code into the clipboard

 **Paste** - Inserts the contents of the clipboard at the current cursor position

 **Search** - Allows you to look for specific text in the current editor window

 **Search Again** - Finds the next occurrence of the search string

 **Check Syntax** - Checks the validity of the program in the current editor window

 **Compile Robot** - Compiles the program and creates a robot file on disk, ready to fight

3.1. Windows

Using the editor is just like using any other text editor except that you can have lots of different text files open at once if you wish. You simply enter the text from the keyboard and it is inserted at the cursor position in the active editor window. You can move the cursor using the arrow keys , or the mouse. All actions, such as copy, paste, compile (etc.) are performed on the ACTIVE editor window. At all times, the active editor window is the one with the highlighted bar at the top, which is placed on top of all other open windows, and which has the cursor flashing in it. Clicking anywhere in an editor window makes that window active. You can also select the editor window you wish to activate by choosing it from the **Window** menu.

Always remember, if you do something and you wish you hadn't, click the undo button (), or select **Undo** from the **Edit** menu. You cannot undo actions such as load or save, but editing actions, cuts, deletions and pastes can be undone in most cases. You can only undo your last action, however, so if you do something else after you've made a mistake, you will not be able to undo the original mistake.

3.2. Blocks (Cut, Copy and Paste)

If you want to copy or move (or even delete) a large amount of text from an editor window, you first have to *highlight* the block of text you wish to manipulate. You can use the mouse to highlight sections of text by clicking in the window on the text you want to highlight, holding down the left mouse button whilst dragging the mouse until you've selected all relevant text, and then releasing the mouse button.

You can also do this using the keyboard, by placing the cursor at the start of the text you wish to highlight, holding down the SHIFT key and using the arrow keys to select the relevant text. If you wish to select all the text in an editor window, choose **Select All** from the **Edit** menu.

This highlighted text can be cut or copied into the Windows Clipboard (using either the toolbar buttons or commands in the **Edit** menu) and can be pasted into the same editor window, or another if you wish. You can use this method to transfer sections of code from one program to another. You can even copy your programs from an editor window into the clipboard, and paste them into other applications (like Word for Windows or EXCEL), or copy text from such applications back into the editor. This allows you a great deal of flexibility.

To illustrate the use of the cut, copy and paste functions, we'll type in a short Combat Zone program. First, enter the following code into the editor window we opened.

```
MYANGLE=RND(360)
REPEAT
        RADAR(MYANGLE)
        IF OBJECT=TANK THEN
                TURRET MYANGLE
                FIRE DISTANCE
        ELSE
                MYANGLE=MYANGLE+5
FOREVER
```

Now move the cursor back up to column 1 of line containing the command FIRE DISTANCE. Hold down the shift key and press the down arrow. You'll see the whole line is highlighted.

*Now click the COPY button (), or select **Copy** form the **Edit** menu, or press Ctrl-Ins. Now point the mouse cursor to the start of the line containing the ELSE statement and click the left mouse button. The green highlight should disappear.*

*Now click the PASTE button (), or select **Paste** form the **Edit** menu, or press Shift-Ins, and you should see a copy of the FIRE DISTANCE line has been inserted. The program should now looks like this:*

```
MYANGLE=RND(360)
REPEAT
        RADAR(MYANGLE)
        IF OBJECT=TANK THEN
                TURRET MYANGLE
                FIRE DISTANCE
                FIRE DISTANCE
        ELSE
                MYANGLE=MYANGLE+5
FOREVER
```

This robot would turn to face an enemy tank and fire twice.

Now highlight the second FIRE DISTANCE line again and click CUT () to remove it, restoring the original program.

3.3. How to Correctly Enter your Combat Zone Programs

To enter the programs is simply a matter of typing the instructions into an editor window. There is no need to put each instruction on a separate line (although I generally do in order to make the program clearer), nor do you need to indent the lines as I've done in my examples. All you need to do is place at least a space, a colon (":") or a line break between sets of instructions. You can have as many blank lines and spaces between instructions as you wish. The more so called "whitespace" you have in a program, the easier it is to read.

The compiler is NOT case sensitive, so, as far as it is concerned, the words "Happy" and "hAPPY" are the same. If you don't wish to type programs in capitals then don't. It's up to you.

Will all this is mind, the example shown on the previous page could be entered as shown below, and it will still compile into the same code, but it is not as easy to read and would be more difficult to debug.

```
MyAngle=RND(360) Repeat : Radar(MYANGLE)
IF object=TANK THEN turret myangle : fire DISTANCE
else MYANGLE=MyAngle+5 FORever
```

In most cases you will want to add comments to your program so that you can remember what each part does. It is good practice to add comments because you often forget what you were thinking the next time you go back to look at the code. The problem is that the computer does no know the difference between program code and comments unless you tell it, so you must enclose comments between curly brackets "{" and "}". You can put anything between such brackets and the compiler will completely ignore it when it comes to build your robot (see later). You can use curly brackets to make the compiler ignore areas of program too, and this enables you to easily add those parts back in later, by simply removing the brackets.

Example

Well add some comments to our program. Add the comment lines shown below:

```
{ ** Pick a random starting angle **}
MYANGLE=RND(360)

{ ** Main loop **}
REPEAT
        { ** Look for tanks **}
        RADAR(MYANGLE)

        { ** If we see a tank then... **}
        IF OBJECT=TANK THEN
                TURRET MYANGLE          { ** Turn to face it **}
                FIRE DISTANCE           { ** And fire...      **}
        ELSE                            { ** If we don't see a tank **}
                MYANGLE=MYANGLE+5       { ** Add 5' to the scan angle **}

FOREVER
```

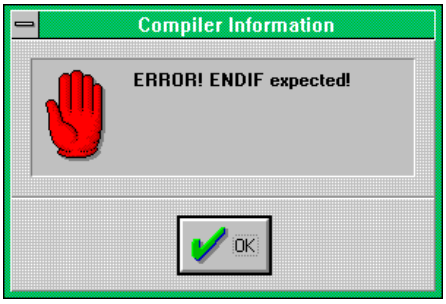
3.4. How to Compile your Programs

Before you compile your program, it is always best to save it first so that you have a copy of your latest work in case anything horrible should happen during compilation. Once you're ready to compile, make sure the window containing your program code is the current window, then click the CHECK SYNTAX button or select **Check Syntax** from the **Compile** menu. You can also press F8 if you prefer keyboard shortcuts. ().

A dialog box will open in the middle of the screen showing you the current line number being compiled. If any errors occur, a message box will open telling you of the error, and the offending piece of code will be highlighted. You must then click back in the editor window and correct your mistakes before checking the syntax again. A complete list of errors and their meanings is given at the end of this section.

Example

Now let's check the syntax of our program. First we'll save our code, so click the SAVE button and enter the name FIRST.ROB in the filename box. Then click OK to save it in the current directory. Now click the CHECK SYNTAX button. The compilation box will appear, and then the following error is received:



The line containing the ELSE command is highlighted. We have forgotten to put and ENDIF at the end of the IF..THEN..ELSE..ENDIF structure, so let's put one in.

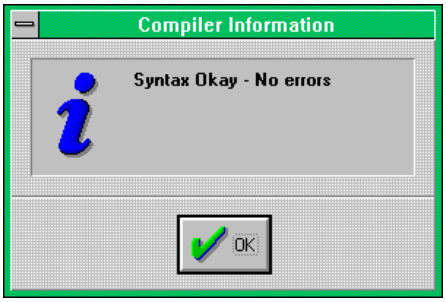
```
{ ** Pick a random starting angle **}
MYANGLE=RND(360)

{ ** Main loop **}
REPEAT
        { ** Look for tanks **}
        RADAR(MYANGLE)

        { ** If we see a tank then... **}
        IF OBJECT=TANK THEN
                TURRET MYANGLE          { ** Turn to face it **}
                FIRE DISTANCE           { ** And fire...      **}
        ELSE                            { ** If we don't see a tank **}
                MYANGLE=MYANGLE+5       { ** Add 5' to the scan angle **}
        ENDIF

FOREVER
```

Now save this code again by clicking the SAVE button (there's no need to enter the name again, the program remembers it's name from last time). Now click CHECK SYNTAX again...



This time there were no errors, so we're ready to create a compiled robot on disk, which we can then load into the Arena program.

To compile a program, click the COMPILE button (), or select **Compile** from the **Compile** menu (or press F9). A compile dialog window will appear containing four boxes. The top box, **Compile to Disk** contains the default filename for the compiled robot. This is usually the name of the program file with the .ROB file extension replaced by .CMP. You may change the filename if you wish, but you should ensure that ALL COMPILED PROGRAM FILENAMES HAVE THE ".CMP" FILE EXTENSION. If you click the checkbox to turn compile to disk off, any code which is created is not saved to disk (which is effectively the same as clicking the Check Syntax button).

In the **Robot Name** box, you can type the name of this robot (as you would like it to appear in the Combat Zone Arena information boxes, see the next section). This defaults to MARVIN, but you can choose any name up to 20 characters long.

You can enter your name in the **Authors Name** box. This is also saved in the compiled file, suitably encoded so that it cannot be removed. This means that anyone who receives a copy of your compiled robot will know who wrote it when they load it into the Arena. If you have a registered version, the name of the registrant will be in here as a default, otherwise it will default to "unknown".

The last box, **Password** is used to further encode the compiled file. The Combat Zone Arena is still smart enough to load your robot without any problems, so you can still give a copy to your friends, but you will not be able to use the Combat Zone Debugger (which you can receive if you're registered) on the robot unless you know the password. This will stop other people looking at your robot using the debugger. If you want to password protect your robot, enter the password in the box and click the check box so that it shows a tick mark. If you don't select the check box, a default system password is used to encode the file.

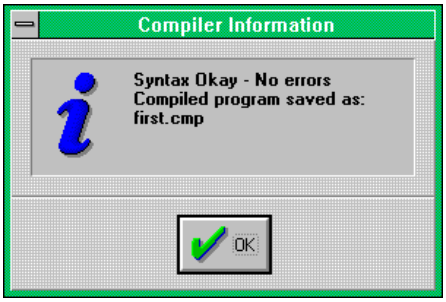
When you've filled out the boxes, click OK to compile. As with CHECK SYNTAX, the compiler box will appear to show you which line number is being processed. If there are any errors, an error message is displayed (see the end of this section for a complete list) and the compiled robot is not saved to disk. Correct the error and compile again. If there a no errors, a compiled robot file is created on the disk with the name specified in the **Compile to Disk** box.

If you don't want to compile the robot, you can click the CANCEL button. The HELP button will only produce help if you have a registered version of the program.

Example

Click the compile button and the compile dialog box will appear, as shown below.

If you wish to change the name of the robot to something other than MARVIN then click in the box and do so. If you wish to password protect your compiled file, click in the Password box, enter a password, and then click the Check box to the left of the word "Password:" to select the option. Now click OK and everything should compile. If you get any errors, correct them and click compile again. If everything has worked correctly, you should receive the following message:



You robot is now ready to fight. Section 4 explains how to use the Combat Zone Arena program to finally see your robot in action. The final two parts of this section give a brief summary of all the Editor menu commands, and list all possible compiler errors. They are provided for reference purposes, you do not need to read them through.

3.5. CZEdit Menu Commands

This section contains a list of all the editor menu options, with a brief description of their functionality.

The File Menu

New
Keyboard Shortcut : NONE — Toolbar Button: 
Opens a new document editor in the main window

Open...
Keyboard Shortcut : F3 — Toolbar Button: 
Brings up a standard file selection dialog box allowing you to select a file to load

Save
Keyboard Shortcut : F2 — Toolbar Button: 
Saves the file in the current active editor window. If the file already has a name, the information is saved, replacing the existing file. If the file has no name, a dialog box is produced, asking you to enter a filename.

Save as...
Keyboard Shortcut : NONE — Toolbar Button: NO
This command allows you to save a file under a filename you specify. A standard save filename dialog box appears and you have to type the name of the file in the top edit box.

Print
Keyboard Shortcut : NONE — Toolbar Button: 
This command brings up the print dialog box, allowing you to select the number of pages to print and the number of copies printed. You can use the Setup button to select a printer or edit it's settings.

Printer setup...
Keyboard Shortcut : NONE — Toolbar Button: NO
This options allows you to select which printer you would like to use to print your program files on. It also allows you to edit the printer settings if you click the Setup button.

Exit
Keyboard Shortcut : Alt - F4 — Toolbar Button: NO
Exits CZEdit, closing all windows. You will be asked if you are sure you wish to quit, and informed if the contents of any open editor windows need to be saved before you exit.

The Edit Menu

Undo — Keyboard Shortcut : Alt - Bksp — Toolbar Button: 
Undoes the last editing action

Cut — Keyboard Shortcut : Shift - Del — Toolbar Button: 
Cuts the highlighted text from the document and places it in the Windows clipboard.

Copy — Keyboard Shortcut : Ctrl - Ins — Toolbar Button: 
Copies the highlighted text into the Windows clipboard.

Paste — Keyboard Shortcut : Shift - Ins — Toolbar Button: 
Pastes the contents of the Windows clipboard into the document at the current cursor position.

Select All — Keyboard Shortcut : Ctrl - End — Toolbar Button: NO
Highlights the entire contents of the active editor window.

Clear All — Keyboard Shortcut : Ctrl - Del — Toolbar Button: NO
Deletes everything in the active editor window. USE CAREFULLY!

The Search Menu

Find... — Keyboard Shortcut : F4 — Toolbar Button: 
Allows you to search the active program for some specified text.

Find Next — Keyboard Shortcut : F5 — Toolbar Button: 
Finds the next occurrence of the last Search string.

Replace... — Keyboard Shortcut : F6 — Toolbar Button: NO
Allows you to replace all occurrences of one set of characters with another.

The Compile Menu

Check Syntax — Keyboard Shortcut : F8 — Toolbar Button: 
Checks the validity of the program in the active editor window and report on any errors. This does NOT generate program code.

Compile — Keyboard Shortcut : F9 — Toolbar Button: 
Opens the Compile Dialog, where you can specify the Robot Name, the Author name, the compiled filename and optional password. The program in the active editor window is then compiled and any errors reported.

The Window Menu

Tile Keyboard Shortcut : NONE Toolbar Button: NO
Arranges all open editor windows on the main window background so that all Windows are equally visible.

Cascade Keyboard Shortcut : NONE Toolbar Button: NO
Arranges all open editor windows on the main window background, with one laid on top of another. The active window is placed on the top.

Close All Keyboard Shortcut : NONE Toolbar Button: NO
Closes ALL open editor windows. USE CAREFULLY!

Arrange Icons Keyboard Shortcut : NONE Toolbar Button: NO
When an editor window is minimised, it's icon is placed in the main window. This command arranges all such icons in a straight line along the bottom of the main window.

Horizontal Toolbar, Right Vertical Toolbar, Left Vertical Toolbar
Keyboard Shortcut : NONE Toolbar Button: NO
As a default, the toolbar is horizontal, underneath the menus. You can select to have it running vertically down the left or right sides of the window if you wish. You can only select one of these settings at a time.

The Help Menu

Most functions on the Help menu only work in the registered version of the editor, the only one that does work here is.....

About Keyboard Shortcut : NONE Toolbar Button: NO
Gives information about the program. If you have a registration number, you must open the about window and click the register button to enter your number. Clicking the main logo opens a credits window. The screen also contains information about registering the product (see the end of this manual for more details).

3.6. Compiler Errors

The following two pages contains a list (including brief explanations) of all the errors you can generate when compiling a combat zone program. I've tried to split them into sections to make it easier to find your error.

Errors in Expressions

Operator expected!

You have missed out the operator (i.e. the +, *, - etc.) in an expression. You may have typed "5 5" instead of "5+5" for example.

Number or variable expected..cannot have two operators in a row!

You've missed out a number or variable between two operators (e.g. "5+*6").

Illegal use of the RND function

RND is a function not an operator. You'll get this error if you try something like "5RND(6)" instead of "5+RND(6)".

CLOSE bracket without OPEN bracket -Self explanatory!

Closing bracket expected! -Self explanatory!

This command does not take a BOOLEAN expression

This error occurs if you've used a boolean expression with a command that requires a numerical expression i.e. done something like "MOVE 5=5".

Errors in Commands

SHIELD requires either ON or OFF

The SHIELD command has only two parameters, ON and OFF. You cannot do "SHIELD 10".

BOOLEAN expression required with IF!

The IF command requires a condition which is boolean. A numerical expression won't work. You cannot have an expression like "IF 5+6 THEN". It would have to be "IF X=(5+6) THEN", or something of a similar nature.

THEN expected -Self explanatory!

ENDIF expected! -Self explanatory!

No matching IF for this ELSE

You have inserted an ELSE statement which does not relate to any IF statements in the program.

No matching IF for this ENDIF

You have inserted an ENDIF statement which does not relate to any IF statements in the program.

No matching IF for this THEN

You have inserted a THEN statement which does not relate to any IF statements in the program.

UNTIL or FOREVER expected

You have a REPEAT statement in the program that does not have a matching FOREVER or UNTIL command. You must go back to the program and insert one. Use indenting to make programs easier to read and avoid errors such as these.

BOOLEAN expression required with UNTIL!

Like the IF statement (see error above) the UNTIL statement requires a boolean expression (i.e. you cannot have "UNTIL 5").

UNTIL without matching REPEAT
You have inserted an UNTIL statement which does not relate to any REPEATs in the program.

FOREVER without matching REPEAT
You have inserted a FOREVER statement which does not relate to any REPEATs in the program.

DO requires a number, NOT a BOOLEAN!
The DO command requires the number of times the loop is to be done as a parameter. You cannot, therefore, have a boolean expression as the DO parameter. "DO A=B" is not valid.

LOOP expected
You have forgotten to close the DO loop with the LOOP command.

LOOP without matching DO
The is a LOOP command in the program that does not match any DO statements.

RETURN expected
You have GOSUBed a routine, but that routine has no RETURN statement at it's end.

Errors in Variables and Constants

Cannot assign a value to a constant
You have tried to assign a value to a reserved word (see SYSTEM VARIABLES in main manual). You cannot do things like "WALL=10".

Cannot assign a value to a system variable
You have tried to assign a value to a system variable e.g. "OBJECT=20" or "DISTANCE=100". This is illegal. Only the computer can set the values in system variables.

Cannot assign a boolean value to a variable
Variables can only hold numerical values e.g. "X=5+6". You cannot assign the result of a boolean expression to a variable, e.g. you cannot have "X=5>6".

Assignment ("=") Expected!
You have neglected the assignment sign e.g. "FRED 5+6" instead of "FRED=5+6".

General and System Errors

Unknown command xxxxx
Most commonly you've mistyped a command e.g. "MVOE" instead of "MOVE".

Duplicate label found!
There can only be one label with each name. You have two (or more) LABEL statements with the same name. Change one of them.

Cannot find label xxxxx
You've made a GOTO or GOSUB call to a label and forgotten to use the LABEL command to name that part of the program. Go back and insert the label in the correct place.

SYSTEM ERROR! Cannot allocate display memory
There is too little memory to handle the CZEdit window and compiler. Shut some other programs down before trying to compile again. You may have to restart Windows if this happens.

Cannot open file xxxxxx
The filename you specified for the compiled file is invalid for some reason. Try compiling the program again with a different compiled robot filename.

4. The Combat Zone Arena

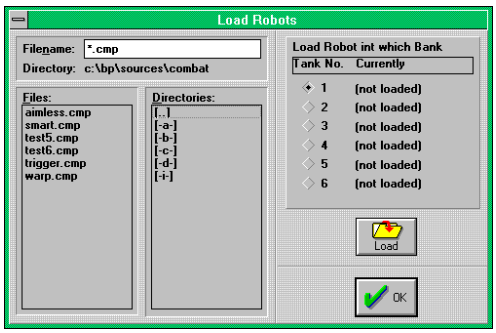
Now that you know how to program, and have created a robot (or maybe not, you could be reading this manual in some funny sort of order!) this section describes how to load your compiled robots into an Arena and watch them kick the living daylights out of one another! As with the Editor instructions, a complete list of all available menu options is given at the end of this section, but the most important options are explained in more detail as we go along.

If you have not written any robots of your own, there are three precompiled robots supplied with the program. AIMLESS.CMP is a very simple robot (whose program can be found in the AIMLESS.ROB file, supplied with the package). It is very easy to kill, you can use it for target practice when you develop your own tanks. TRIGGER.CMP (whose program code is also supplied) is slightly tougher because it shoots a lot (trigger happy, see!), but it is still not a very clever program. SMART, as it's name suggests, it a bit more clever. I have not supplied the program code for this, just the compiled robot, so you can't see how it works. When a robot of yours can consistently beat this chap, then you know you're getting the hang of it (but believe me, there are smarter robots than this one!). Follow the sections below and watch the robots fight so you can get a feel for what Combat Zone is all about.

4.1 Loading Robots

There are six tanks into which you can load your programs. There is no restriction on which tanks you can load programs into, and every tank can run a different program (or the same one if you wish. Loading all six tanks with the same program is a good way of debugging your own programs during tank development.). So, for example, you may wish to load programs into only 4 of the six tanks. Two of the tanks will run the SMART.CMP robot, one will run AIMLESS.CMP, and one TRIGGER.CMP.

To load some robots, click the Load Robots button (), or select **Load Robot** from the **File**, or press F3. The following dialog box then opens:



On the left-hand side of the box is a list of the compiled robots in the current directory. You can change the directory by clicking on the directory list immediately to the right, or by typing the new directory in the Filename box. When you have found the robots you wish to load, first select a tank you wish to load the program into, by clicking on one of the radio-buttons on the right-hand side of the screen. Any programs already loaded into the tanks are displayed here. You can overwrite them if you wish (in which case you will be asked if you are sure you wish to do so). Then select a filename from the list and click the Load button. The box on the right will show the name of the program you have just loaded. Load as many robots as you want, and click the OK button when you are finished.

NOTE: YOU DO NOT HAVE TO PUT ALL THE TANKS IN THE ARENA IF YOU DON'T WANT TO, SO YOU CAN LOAD AS MANY PROGRAMS AS YOU LIKE AND STILL ONLY HAVE TWO OR THREE TANKS FIGHTING (see Starting a Fight, section 4.5).

NOTE ALSO: You cannot Unload a robot program. If you do not wish a tank to fight in the Arena, simple don't include it in the battle (see Starting a Fight, section 4.5).

4.2 Loading an Arena

Now we need to load an Arena for the robots to fight in. Two are supplied with the package: SIMPLE.CZA is a simple square arena with no obstacles or hazards. This is a good testing ground for robots during development and leads to good fights since the tanks are not distracted by other hazards. TOUGHER.CZA has rough ground, holes, warp squares, gates and lots more walls. The special arena features are:

HOLES Avoid these at ALL costs. If your tank drives over a hole it dies INSTANTLY, falling to it's doom into the bottomless pit beneath every arena.

WARP SQUARES
These randomly warp your tank to another point in the Arena. You are always warped to a simple floor tile, but NO CHECKS are made as to the location of other tanks, so you may warp right on top of one of them. Warp gates can help you avoid a tank that is chasing you, but could also kill you if you get an unlucky warp.

GATES Gates are either OPEN or CLOSED. If you shoot an open gate, it shuts, and if you shoot a closed gate, it opens. It requires a DIRECT hit to open or close a gate. You can drive over open gates, but will crash into closed ones.

GUN EMPLACEMENTS
Very much like fixed tanks. Gun emplacements are ONLY available in the registered version of Combat Zone

ROUGH GROUND
Slows the progress of your tank. The RADAR command does NOT return rough ground as a feature unless you select "Return Rough from Radar" in the Options box (see later).

BONUS A bonus square does one of the following things, randomly and with equal frequency:
(i) Nothing.
(ii) Halves the amount of damage your tank has taken.
(iii) Completely restores your shield energy.
Once you have driven over a bonus square, it simply becomes a normal tile again, so only the first tank to drive over it gets a bonus.

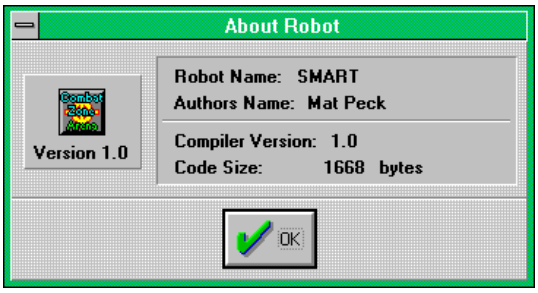
If you want to design your own Arenas, you'll need the Arena Designer, which is sent to you when you register (see section 5).

To load an Arena, click the Load Arena button (), or select **Load Arena** from the **File** menu, or press F4. A standard file dialog box opens. Select the name of the combat zone arena you wish to load and click OK. The Arena will take a few seconds to load and then a box will be displayed giving you information about the Arena size and tile file.

4.3 Viewing Robot Information

Every compiled robot program holds information about itself and it's author. If you want to see who the talented author of the most unbeatable robot is, click one of the View Robot buttons, or select **View Robot** from the **View** Menu (in which case you have to select the robot you wish to view when asked).

The view robot buttons are :  . The buttons are greyed out when no robot is loaded into that tank. The robot information for the SMART robot is shown below:

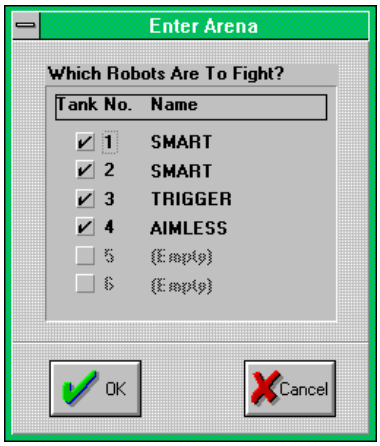


4.4 Viewing an Arena

To view an Arena (after you have loaded it), click the View Arena Button (), or select **View Arena** from the **View** Menu, or press F6. A window will open, inside the main window, showing you the Arena. If you move the mouse into this window, you will see it changes shape, becoming a magnifying glass. At the bottom of the window, the location of the cross in the centre of the mouse cursor is displayed, along with the contents of that tile. This enables you to point the mouse cursor at something to find out what it is.

4.5 Starting a Fight

Now that you've loaded your robots, and an arena for them to fight in, click the Start Combat Button () (the one with the big green light), or select **Start Combat** from the **Combat** menu, or press F9. The following dialog box will appear, listing the robot programs you currently have loaded into the tanks:



As a default, all the tanks are selected. Any you do not wish to place in the Arena, deselect by clicking the relevant check-boxes. When you've picked which tanks you want in the Arena, click OK to start the combat, or Cancel to, well, cancel I suppose.

4.6 The Info Windows

Now this is more like it. The robots are charging around, firing at each other. Violence abounds. What's more, information about the state of play is also readily available. On the right hand-side of the Arena screen are the Info Windows (one of which is shown below). These contain loads of interesting facts about each tank.

A picture of the tank is shown in the top left corner of the box, so you know which tank the info is for. Everything else is pretty self-explanatory really, except maybe **Action**, which tells you the last Tank Command which was performed.. Keeping an eye on this screen will let you find errors in your tank program if it doesn't appear to be working properly.

The Info Windows do slow the game down though, and on less powerful machines, you may like to turn them off (see Setting your Options, later). You can still move the mouse cursor into the playing window to see what you're pointing at, but now you can point to a tank and the tank name and current damage are displayed at the bottom of the screen. This means you can still see how well your tank is doing, even with the Info Windows turned off.

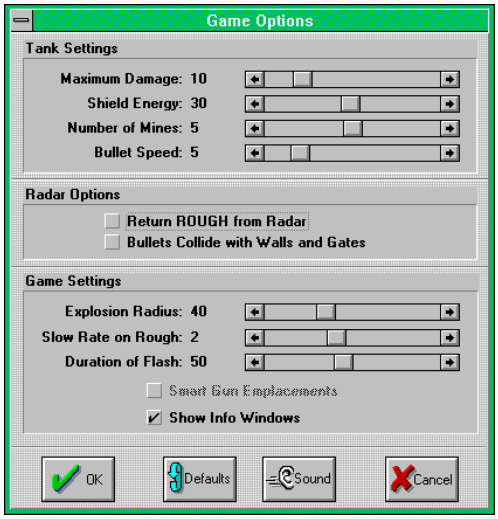
You can pause the combat at any time by clicking the pause button (), or by selecting **Pause/Restart Combat** from the **Combat** Menu, or pressing F11. You can restart by clicking the same button again.

You can stop the combat by clicking the Stop button (), (the one with the big Red light) or by selecting **Stop Combat** from the **Combat** Menu, or pressing F10. YOU CANNOT RESTART A GAME ONCE YOU HAVE STOPPED IT. If you stop a game with only one robot left alive, the robot name and author name are displayed in a congratulations box.

NOTE: When a game is running, you cannot do anything except Stop or Pause the game. All other functions are disabled.

4.7 Setting your Options

If a game is not running, you can open the Options window by selecting **Options** from the main menu. The following dialog box appears:



The dialog box lets you change most of the major settings of the game. You can alter the Maximum damage a tank can take, the amount of shield energy it has (in seconds), the number of mines available to it (mines are only available with the registered version of the game) and the speed of the bullets.

Return ROUGH from Radar will cause your tank radar to pick up rough ground (rather than ignoring it which is the default behaviour). Selecting **Bullets Collide with Walls and Gates** means that you cannot fire over walls and gates, as you can normally, because the bullets will explode on contact with either of them.

The **Explosion radius** is the size of the bang caused by a bullet. If you set this number to a low value, the game becomes more difficult because your shots must be more accurate if you want to hit anything. Setting this value too high may slow the game down. The **Slow Rate on Rough** setting dictates the number of times slower a tank will move over rough ground than on normal ground, and the **Duration of Flash** dictates how long your tank will flash for when it takes damage. If you set this very low, the same explosion may hit you twice!

Smart Gun Emplacements causes the Gun towers to run a smart program, rather than their dumb basic routine (only available in the registered version).

Show Info Windows allows you to turn off the Info Windows if you wish. This makes the game considerably faster, especially on less powerful machines.

The **Defaults** button sets everything back to the start-up settings, and the **Sound** button selects the sound output (only in the registered version, again).

When you've finished changing the settings, select OK to accept your changes or CANCEL to ignore them all.

4.8 What Now?

Well, now you know how to use everything, it's up to you to create the best tank ever, just to show your mates how smart you are. The rest of this section lists the toolbar buttons and menu options for your reference (aren't I generous). If you like this game, and use it a lot, please read Section 5 which tells you how to register the game, and what's in it for you when you do. Happy coding!

Mat Peck. December 1993

The Toolbar Buttons

Help - Brings up the on line help. Only available with the registered version of the program

Load Arena - Allows you to load in a Combat Zone Arena for your robots to fight in

Load Robot - Loads Compiled Robots to fight in the Arena

View Arena - Displays the Arena. Moving the mouse in the display window allows you to examine the contents of the tile beneath the mouse pointer.

View Robot 1-6 - Displays information about the Robot program. These buttons are greyed out when no program is loaded into that particular tank.

Start Combat - Asks you to load robots and an arena (if you have not already done so). You then select the Robots you wish to place in the Arena (see below). The combat then starts.

Stop Combat - Stops the current battle. YOU CANNOT RESTART A BATTLE ONCE YOU HAVE STOPPED IT!

Pause/Restart Combat - Allows you to pause a combat in mid game and restart it again from that point. This allows you to look at the information in the info windows etc. without being distracted by the live combat.

4.9. Combat Zone Arena Menu Commands

This section contains a list of all the Arena menu options, with a brief description of their functionality.

The File Menu

Load Robot — Keyboard Shortcut : F3 — Toolbar Button:
Opens the load robot dialog box, allowing you to load compiled robots into one of the six available tanks.

Load Arena — Keyboard Shortcut : F4 — Toolbar Button:
Allows you to load an Arena for the robots to fight in. Select the Arena name from the standard windows file dialog box.

Exit — Keyboard Shortcut : Alt - F4 — Toolbar Button: NO
Exits CZArena, closing all windows and unloading all robots and the Arena.

The View Menu

View Robot Keyboard Shortcut : F5 Toolbar Buttons: 
Shows information on the robot program loaded into the selected tank.

View Arena Keyboard Shortcut : F6 Toolbar Button: 
Opens a window showing the Arena you have loaded. Moving the mouse around in this window allows you to examine the Arena contents.

The Combat Menu

Start Combat Keyboard Shortcut : F9 Toolbar Button: 
Starts the fight if both robots and Arena are loaded, otherwise it prompts you to load them before starting the battle.

Stop Combat Keyboard Shortcut : F10 Toolbar Button: 
Stops the battle. YOU CANNOT RESTART A FIGHT ONCE YOU HAVE STOPPED IT!

Pause/Restart Combat

Keyboard Shortcut : F11 Toolbar Button: 
Clicking this button when a fight is on, pauses the fight. Clicking it again, restarts the battle.

The Options Command

Opens the options dialog box (see section 4.8).

The Window Menu

Horizontal Toolbar, Right Vertical Toolbar, Left Vertical Toolbar

Keyboard Shortcut : NONE Toolbar Button: NO
As a default, the toolbar is horizontal, underneath the menus. You can select to have it running vertically down the left or right sides of the window if you wish. You can only select one of these settings at a time.

The Help Menu

Most functions on the Help menu only work in the registered version of the editor, the only one that does work here is.....

About Keyboard Shortcut : NONE Toolbar Button: NO
Gives information about the program. If you have a registration number, you must open the about window and click the register button to enter your number. Clicking the main logo opens a credits window. The screen also contains information about registering the product (see the end of this manual for more details).

5. Combat Zone - The Registration Fee - What's in it for you - And other stuff....

This game is the culmination of nearly 8 months of infuriating work. PC Format originally announced in March 1993 that a series of programming articles were about to appear and I thought it would be a good idea to write Combat Zone and send it to them as a coverdisk feature. Originally, the game was going to be a lot less complex and not as attractive or easy to use, but once I got started it just kept sort of growing (as the proverbial actress said to the bishop...). So, after all that effort I decided I'd make it SHAREWARE.

For those of you who don't know what SHAREWARE is, I'll explain. It basically means that you can use this game for a while (two weeks) and copy it to pass on to as many people as you like. You can put it on bulletin boards and send it to your mate in Papua New Guinea, if you like, but if you enjoy using it and you want to keep it forever and ever, you have to register it with me. There's no thought police here, or nasty viruses that chew up your hard-drive if you don't register, just your conscience to stop you sleeping at night! Also, if you do register you get lots of extra goodies. What goodies? Well.....

An Arena Designer A natty little program that lets you create Arenas for your tanks to fight in. It also lets you edit the graphics which make up the tiles and even the tanks themselves.

Help Files On-line help in the Editor and Arena. In the Editor, you can highlight program command and hit Ctrl-F1 which produces specific help for that command, along with example code. The Entire manual is also available on the computer, at your request.

Some New Arenas A sample of some nice and nasty Arenas for you to battle in.

Gun Emplacements New fixed gun emplacements make Arenas even more dangerous for your tanks. You can even make them smart, to make it a real challenge.

Sound Gripping digital sound (if you have a Soundcard and device driver that can play .WAV files) to add that extra dimension to the battles.

Robots Galore! Access to the Compiled Robot library, so you can get the latest deadly robots written by people like yourself all over the world. If you have a robot you wish to include in the library, send it on a disk to me (with return postage if you want your disk back, or without postage if you send it with your registration) and I'll add it to the archives.

Competitions I'll run The Best Robot Award competitions for registered users, where the winning robot could earn it's writer up to £100 (if enough people enter).

A Debugger??? If enough of you want a robot debugger program (with which you can actually see your program running) then I'll write one. This won't come as part of the registration (but it will only cost a couple of pounds at the most if it do write it), but let me know if you'd be interested when you register.

YES, YES, BUT HOW MUCH WILL IT COST ME?

£10. I don't believe in charging the earth for a game, despite the fact it took so long to write. Games are overpriced as it is without Shareware costing nearly thirty quid too. What's more, if you send a disk (high or low density, it doesn't matter) it only costs **£9**. Outside of the EC, add **£1** because it costs so much more to send everything to you, since you're so far away and all that. Only send cheques or Postal Orders made payable to **M.J.Peck**. Don't send money, because it isn't safe. When you register, you'll get the latest stuff (described above) and your own Registration ID and Number, which you type in in the box produced when you click the Register button (shown over the page) in the About Window.

Send your stuff to the address shown below, and don't forget to include a note telling me whether you'd be interested in a debugger. If you're sending a disk and you have some robots you would like included in the robot archive, then send them along too (this won't affect any entries you make in later competitions), but ONLY send compiled robots, NOT source code, unless you want that given away too!

IMPORTANT NOTICES - READ THESE

General Correspondence

I am quite happy to write to people who have general questions about combat zone, or anything else really. If you have ideas for improvements, or if you find a bug (god forbid), or if you want to know about other Paranoid releases, or if you want a chat about computers in general, or want advice about what to do about your girlfriend Debbie.... BUT I'LL ONLY REPLY IF YOU INCLUDE A STAMPED ADDRESSED ENVELOPE with your letter, because it will cost me a fortune otherwise.

Credits and Thanks

I have to thank the following people for their help during this arduous project:

Sarah Gould For putting up with me whilst I sat here, night after night, finishing this damn program. All my love. I'm sorry that I can't get more mushy and sickening in public!

Pat Fox This man virtually play tested the game by himself. And he added his particular cynical sense of humour to the manual. The man's a saint. And he's tall. A tall saint.....with curly hair. What more can I say? (Is this enough Pat?)

Kenji The other chap who play tested this. I don't know what he looks like, but he's a saint too, and he might have curly hair for all I know, but I don't think he's as tall!

Robert Cray, Gary Moore, Stevie Ray Vaughan, BB King, Buddy Guy and Albert Collins
There's nothing like a good bit of blues to code to!

The Address

Here's the address for everything. Don't forget to include an SAE if you send general correspondence (not necessary when you register).

Mat Peck
"Rivendell"
16 Highdowns
Hatch Warren
Basingstoke
Hampshire
ENGLAND
RG22 4RH