```
DDDD       OO       GG     FFFFF  III     GG     H    H  TTTTT
D   D     O  O     G  G    F       I     G  G    H    H    T
D    D   O    O   G    G   F       I    G    G   H    H    T
D    D   O    O   G        FFF     I    G        H    H    T
D    D   O    O   G GGGG   F       I    G GGGG   HHHHH    T
D    D   O    O   G    G   F       I    G    G   H    H    T
D    D   O    O   G    G   F       I    G    G   H    H    T
D   D     O  O     G  G    F       I     G  G    H    H    T
DDDD       OO       GG     F      III     GG     H    H    T
```

CREATED FOR YOUR ENJOYMENT BY:

DOUGLAS P. DREES

D. THOMPSON McCALMONT

EE 315
Winter, Spring Quarters 1977
Professor Harry T. Garland

1.0 INTRODUCTION

Dogfight is a video game based upon the aerial combat of World War II. The program simulates the response of a real plane to the pilot's commands. Unlike other implementations of this type of simulation, this program actually uses Newton's laws of motion. Our goal was to design a game that was not just fun but also realistic. We wanted the planes to react much the same way a real one would. But we realized that that would not be easy considering the limited computing power of a microcomputer like the Z-1. The program makes some pretty loose assumptions but they are close enough for a reasonable simulation of true flight.

The program was written in 8080 assembly language with a good number of Z80 instructions inserted using the define word pseudo-op. (note: the program will not operate on an 8080 system). The program was designed to be operated on a system fitted with a Dazzler color graphics display module and a set of JS1 joysticks, both built by Cromemco.

2.0 INPUT ROUTINES

2.1 Data Input

One subroutine, 'Input', called once per pass, handles all the inputting of data from the joysticks. It first loads "Axms" into the B register and "Bxms" into the C register. The most significant bit of each of these values is the Crash-flag of its respective plane. If the Crash-flag is set for a plane, then all input data for that plane is destroyed. The first byte inputted is the Action-flag ("Aflag") which is the push button data. This data comes in inverted, so the data must be complemented before use. The Crash-flags are then tested and any appropriate data is destroyed. Then the data from the joysticks themselves is inputted. It also must be negated before it is useable.

The second half of 'Input' is concerned with the ability to roll the planes much like a slow roll. Two counters, "Cntr0" & "Cntr1", are used to slow down the response so that a short touch of the roll button will roll the plane only once. Using this method, only when the counter is timed out and the roll button is depressed does the subroutine 'Roll' get called. When it is called, it flips the orientation bit (bit 0 of "Angl" or "Bngl"), subtracts 180 degrees from the true angle of orientation, and sets the proper counter to zero. If the counter has not timed out at the beginning of 'Input', then it is incremented and 'Roll' is not called. If the counter has timed out but the button is not depressed, the counter is

forced to the timed out value of 16.

2.2 Thrust Loading Routine ('ThId')

Since we are using the buttons to also control the thrust of the planes, some kind of translation was needed to obtain appropriate values for the thrust from a two bit binary input from the buttons. The appropriate bits of the Aflag are rotated into the least significant positions of register A. The Ix register is loaded with the pointer to the beginning of the appropriate plane parameter area. 'Thld' is then called and it loads into the plane's thrust variable the value of thrust that corresponds to the value in the A register.

3.0 INITIALIZATION ROUTINES

3.1 'Gamst' — Game-start Initializing Routine

'Gamst' is called once for every game, and it functions to initialize all necessary registers for the next game. It begins by storing the value from the sense-switches as the "Maxsc." This can be any value between 0 and 7FH; however, anything greater than 63H (i.e., 99D) automatically defaults to 63H. Likewise, leaving the switches at zero wil force a default to the value FH (or 15D). A value of "1" will always go to "2" because of the "win-by-two" feature described in the discussion of "Score." Next in 'gamst,' Bits 7 of "Ayms" and "Byms," the "Initialization FIags," are set so that both planes uill be initialized when 'init' is called. After this the bullet parameter area is initialized; the bullet timers are set to 48D and one of the instructions is initialized for all 10 (5 for each Plane) bullets. "Cntra" and "Cntrb" (Counters-A and B) are set to 10D; these time the interval between bullet firings. Then the following variables are initialized to zero: "CntrO" and "Cntr1" (Interval Counters — used to determine the frequency of 'roll,' when the "roll" button is pushed; "Temp2" (Used to slow down the roll of the Planes in 'thup'). "Scora," "Scorb," "Pscra," and "Pscrb" (Scores — explained in "Score" description); and finally, "Pntra" and "Pntrb" (Pointers A and B — which indicate the bullet which was just created. Before returning 'gamst' sets the Replot-Flags (RPA and RPB) and thus enables the Ground to be plotted the first pass through the program (see description of 'Eplot' and Related Subroutines below).

3.2 Plane Initialization Routine ('Init')

This routine very simply loads into the parameter area of a lost or destroyed plane the appropriate values to create a new plane on the ground in the correct corner. It also resets the second crash flag used in the handling of explosions. It only initializes planes which have in some way gone off the screen and have had their Init-flags set. At the beginning of 'Init' is an input from the sense switches which restarts the game if bit 7 is a "1".

4.0 MOVEMENT SIMULATION ROUTINES

4.1 Angle Of Orientation Update

The first real calculation the program does is the updating of "Theta" and "Thetb". The incremental angle is added to the true angle every five times through the theta update subroutine, 'Thup'. A counter ("Temp2") is incremented every time through 'Thup' and when the count reaches five, the counter is cleared and the angles are updated.

'Thup' also performs the task of converting the true angle of orientation, which is a seven bit quantity, to the displayed angle of orientation, a four bit representation of the sixteen possible orientations of a plotted plane. Only the most significant bits are used in the conversion. The newly calculated values of "Theta", "Thetb", "Angl", and "Bngl" are then stored away.

4.2 Plane Flight Simulation

Flight simulation was performed by a subroutine called 'Updat'. It was designed to update the position of one plane at a time. The Ix register was used to pass the pointer to the proper parameter area. This is the only value used in 'Updat' which was unique to a particular plane. Otherwise the routine treats both planes identically.

There were certain assumptions made about the nature of flight. To simplify the simulation, the number of forces acting on the plane were reduced to four: lift, drag, weight, and thrust. Although this is simplified, it is not an oversimplification because these are the main forces and control most of the plane's responses.

Of these four forces, two are proportional to the square of the total velocity of the plane: lift and drag. Lift is also a function of the angle of attack of the wing into the wind. This angle is the difference between the angle of the plane itself relative to horizontal, theta, and the angle of the plane's velocity relative to horizontal, phi (refer to fig 2.1). It was assumed that the lift would be zero whenever the airflow over the wing was in the reverse direction. This is not quite true but the assumption does not greatly affect the flight of the plane and it does simplify the the simulation.

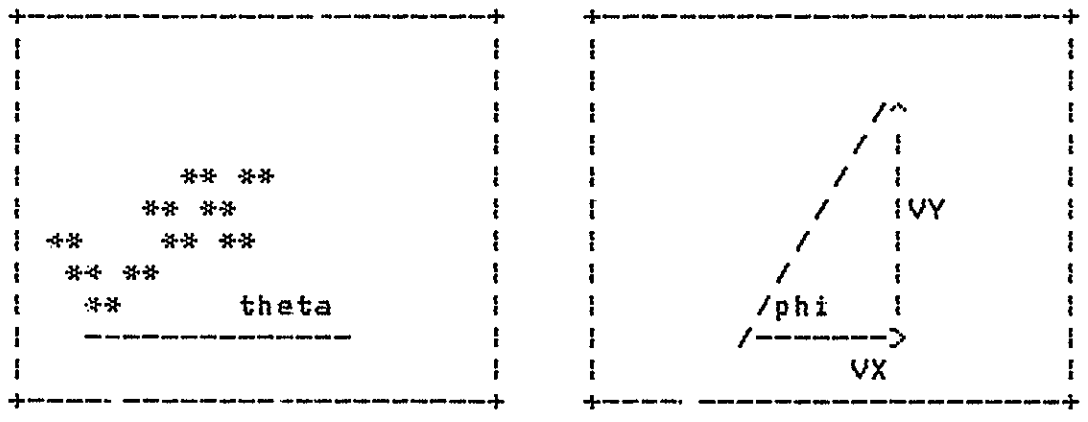

fig. 2.1

The equations for lift and drag are:

$$DRAG = KD*(Vt**2)$$

$$LIFT = \begin{cases} KL*(Vt**2)*\cos(Theta-Phi) ; & \text{for } \cos(Theta-Phi) \geq 0 \\ 0 ; & \text{for } \cos(Theta-Phi) < 0 \end{cases}$$

Newton's laws state that the sum of the forces acting on a body in any direction is equal to the acceleration of the body in that direction. The plane position update subroutine, 'Updat', first calculates the square of the total velocity. It also takes the arctangent of the ratio of the vertical velocity (VY) to the horizontal velocity (VX) which is the angle of the plane's true velocity to the horizontal, phi. Next the sine and cosine of theta, phi and (theta - phi) are computed. It is then possible to compute the sum of the forces in the X and Y direction according to

the following equations:

```
FsubX = Thrust*cos(theta) - Lift*sin(theta)
              - Drag*cos(phi)

FsubY = Thrust*sin(theta) + Lift*cos(theta)
              - Drag*sin(phi) - Weight
```

Once the forces are summed, if we assume the time increment equal to one, then the new velocity in either direction is just the sum of the old velocity and the acceleration in that direction.

```
VnewX = VoldX + FsubX

VnewY = VoldY + FsubY
```

And again since delta t is one, the new position of a plane is merely the sum of the old position and the newly computed velocity. But a word must be said here about the definition the direction of positive X and positive Y relative to the definitions of the positive directions of their respective velocities. Both X and Y were defined for easy calculation of the address of the position in the real memory space. X is therefore defined as increasing to the right and Y as increasing down:

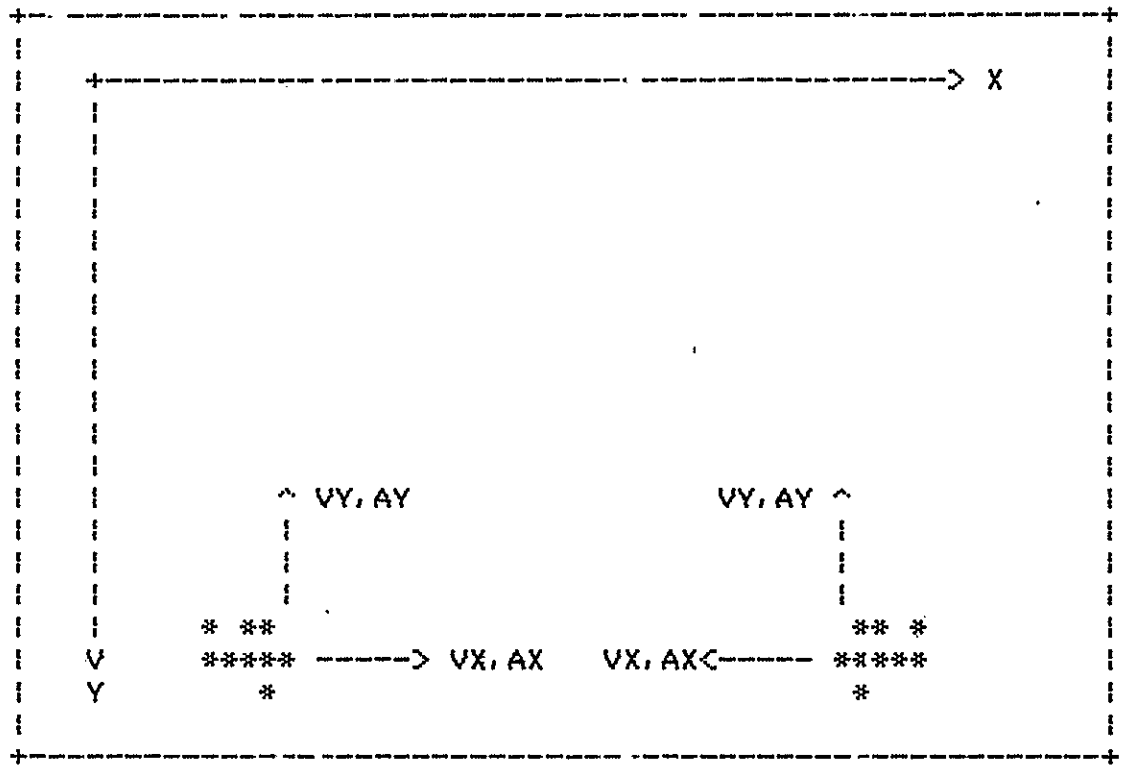

fig. 2.2

But to make it possible to treat both planes identically, it was required that the accelerations and velocities be defined by the normal flying orientation of the plane. For the plane that flies normally from left to right, VX and AX are both defined as increasing to the right. For the plane that normally flies from right to left, VX and AX are both defined as increasing to the left. VY and AY for both planes are defined as increasing upward (fig. 2.2). Therefore, before the velocity can be added to the position to get the next position, the value of VY has to be negated and the value of VX has to be negated if the plane normally flies from right to left. The latter entails the checking of a bit in the orientation angle and negating VX if the bit is set.

This routine also tests to see if a plane has flown off the top of the screen or crashed into the ground. In both cases a flag must be set that tells the initialize subroutine, 'Init', to create a new plane. For the case of a crash the opponent's score must also be incremented. The program tests first if the new position is a negative number. If it is, then the plane has left the screen one way or the other. If VY is positive, then the plane has crashed!

4.2.1 Subroutines Called By Update Routine -

There are four routines used by the update routine to perform its calculations: 'Cosin', 'Phi', 'Mult', and 'Overf'. These routines are required because the instruction set of the Z80 processor does not include such commands as multiply, sine, cosine, or arctangent.

4.2.2 Sine And Cosine Lookup Routine -

This routine is a simple table lookup with an algorithm added which figures out the signs of the sine and cosine depending on the value of the angle. The table is arranged so that the each entry is proportional to the sine of an angle which is proportional to the displacement of the element. It is 180 degrees long and has 64 entries as the angles we chose to deal with are only seven bits in accuracy. First the angle is saved for later reference, and is then stripped of its most significant bit. The result is then used as a displacement in an indexed load instruction which has the effect of getting the absolute value of the sine out of the table. The second most significant bit of the result is then reversed and it is again used as a displacement in the table lookup. This has the effect of getting the absolute value of the sine of the angle plus 90 degrees which is equal to the absolute value of the cosine. The angle is then retrieved and the two most signicant bits are tested to determine the signs of the sine and cosine according to the following diagram:

bit 6 = 0 bit 5 = 1 cosine - sine + 180 > angle > 90	bit 6 = 0 bit 5 = 0 cosine + sine + 90 > angle > 0
279 > angle > 180 sine - cosine - bit 5 = 0 bit 6 = 1	360 > angle > 270 sine - cosine + bit 5 = 1 bit 6 = 1

fig. 2.3

4.2.3 Multiply Subroutine ('Mult') -

The algorithm used in this subroutine is flowcharted in ref. 1. It is a simple add and shift algorithm and any questions concerning it would be answered by reading pages **-** in that ref. The only difference is an added portion which allows for one of the operands to be negative. The operand in the D register is checked for sign and the absolute value is found. The multiplication is then carried out and afterward the result is negated if that operand were originally negative.

4.2.4 Arctangent Calculation Subroutine ('Phi') -

This routine takes the signed values of VX and VY, first calculates their ratio, and then looks up the arctangent of that ratio.

The process begins by saving both values and then deriving their absolute values by testing their signs and negating the values if they were found to be negative. Then the absolute value of VY is divided by the absolute value of VX to yield an eight bit result. Another subroutine, ('Tlu'), is then called which actually looks up the value of the arctangent of the calculated ratio. The last step is to add the necessary high order bits which will place the angle in the correct quadrant (fig 2.3). The signs of the original operands are required for this last step.

The division of two eight bit integers yields a 16 bit result. In most applications, the low order eight bits are usually discarded. But in order to obtain an accurate value for the arctangent, some of the low order bits were needed. In fact the portion of the 16 bit result that was actually used was the middle eight bits. This is due to the shape that the arctangent of any value over 16 is essentially 90 deg. and the arctangent of any value less than 1/16 is essentially 0. The actual division algorithm was also taken from ref. 1, but it was changed so that the peculiar result could be obtained. The basic division consists of a loop which is executed eight times; once for each bit of the dividend. The result of this division is the eight most significant bits of the ratio. Then another loop is entered for which the dividend is always taken to be zero. This loop is executed four times to get the next four bits of the ratio. The most significant four bits of the ratio are thrown away.

The arctangent lookup is performed by another routine ('Tlu'). Because it undesirable to have a table with 256 entries; one for each value of the ratio, this subroutine was written to compress the table into a much smaller memory

space. The nature of the arctangent function is such that there are ranges of values of the ratio wherein the arctangent is approximately constant. The compression is accomplished by first testing the ratio against a number of ranges and if it is not found to be in any of the ranges, it is then used as a displacement in a much smaller table. The testing for the ranges was done by comparing the ratio to a value. If the ratio was larger than the value, then the routine would return a value of the arctangent which was constant over the range above the value compared. If the ratio was smaller, it would then be compared to a smaller number. This would continue until the ratio was determined to be less than 31H. At this point it is more efficient to use a table. The ratio is used as a displacement in the table to get the proper value of the arctangent.

Once the arctangent of the ratio of the absolute values of VY to VX has been obtained, all that remaines is append the two most significant bits determined by the signs of the velocities according to fig. 2.3 (VX has the same sign convention as the cosine, and VY is same as sine).

4.2.5 Overflow Correction Subroutine ('Overf') -

The common problem of what to do when an overflow is detected is solved in this routine by first determining the proper sign the result of the operation was to have and then returning the largest integer with that sign. This leads too errors in the computation but these errors, although they can be quite large, have little detrimental effect on the simulation.

4.3 BULLET MOVEMENT ROUTINES

There are three topics to discuss concerning the handling of the bullets. First the bullet positions must be updated. They must be plotted in the B/W page. And new bullets must be created when the trigger is depressed.

4.3.1 Bullet Plotting -

The other topics depend upon an understanding of how the bullets are plotted in memory. Since the Z80 has bit set and reset instructions and since the bullets should be as small as possible, it was decided that the bullets would be plotted using these bit type instructions. The X and Y positions are used to generate an address (see 'Adgen') in the picture area and also a bit address within the addressed

byte. The address is put in the HL register pair and the set and reset instructions are performed on the memory location addressed by the HL register. There is a problem though in that there is no good way to change the bit address of the instruction during execution. Although it is poor programming practice to change instructions in the instruction stream, it seemed to be the only way to perform the necessary manipulations. Hence there is a section in the main program devoted to the generation of one byte of the required bit instructions. The address of the bit within the byte is rotated to the left three times and the number 6 is added (6 is the designator of the location addressed by the HL reg.). The most significant bit is set for a reset (or erase) instruction and both of the highest order bits are set for a set (or plot) instruction. These high order bits are set just before the instruction is stored in the instruction stream and then executed.

To speed up the operation of the program, the address of the bullet and the bit instruction are stored in a parameter area until the next time through the program. There are five bullets maximum per plane and hence ten different parameter areas allocated for the bullets.

4.3.2 Bullet Position Update -

The main program contains a loop which updates all bullets that are still in existance. Their existance is determined by whether their counters have timed out. The X and Y positions are updated by a subroutine ('Bulup'), and the new address is also generated by a subroutine ('Adgen'). The new bit instruction is actually generated in the loop as well as the incrementation of the counter.

The bullet position update subroutine, 'Bulup', calculates the new bullet position by adding the X and Y components of the bullet's velocity to the respective components of its position. The X and Y positions are stored as 12 bit quantities. The least significant four bits of both components are stored in one byte. They are accessed individually by rotat-digit instructions (RRDD & RLDD). The velocity components are eight bit quantities with only five bits of accuracy. The velocity is added to the low order four bits of the position and the low order four bits of the result is stored back. The sum is then shifted right four times and the high order eight bits of the position is then added. For the Y components, the result of this addition is checked for sign. If it is negative, the bullet either hit the ground or went off the top of the screen and in either case the counter is set at maximum so that the bullet ceases existance. The X components are not checked because the bullets are allowed

to fly off one side of the screen and reenter the other side.

4.3.2.1 Test For Bullets Hitting Planes -

When each bullet is updated, it is also checked for a possible hit. The algorithm which is used is to test whether the bullet is plotted in the same byte as the nose of the opponent's plane. If it is then there is a hit! A subroutine, 'Hitem', is used to execute this algorithm and to take care of the actions that are caused by a hit. These include setting the Crash-flag, pointing the hit plane nose down, and to destroy the plane's X velocity. The Crash-flag has the effect of killing all pilot inputs so that the plane must fall under the influence of gravity.

4.3.3 New Bullet Creation -

Two conditions must be met before a bullet can be fired; 1) the bullet button on the joystick console must be depressed, and 2) a counter must have timed out since the firing of the last bullet. First the counter is checked and if there is still time left, the counter is incremented and the program skip the section that creates bullets. If it has timed out, the "Aflag" is tested to see if the trigger button was depressed. (see 'Input' for more info on "Aflag"). If it was depressed, then a new bullet is created by a subroutine called 'Fire'. 'Fire' first sets the timer to zero. It then transfers the X and Y position of the firing plane's nose to the X and Y position locations in the bullet parameter area. It then calculates the bullet's velocity by using 'Cosin' to get the sine and cosine of the plane's angle to the horizontal. The cosine divided by eight is the X velocity and similarly the sine divided by eight is the Y velocity. This ensures that all bullets travel with the same speed and that no bullet can hit the plane that fired it.

A subroutine, 'Shift', had to be used to perform a right arithmetic shift as the SRA instructions did not always perform an arithmetic shift.

5.0 PLOTTING ROUTINES

5.1 Introduction And Explanation Of Dazzler

There are a number of objects in "DOGFIGHT!" which must be plotted on the TV screen. This is done by the Dazzler circuit board, which does a DMA of the RAM plugged into the system. (The Dazzler and related calls is explained elsewhere.) The Dazzler has several different plotting modes including the ability to plot color or black and white and also to vary the size of the blocks that are plotted. The screen size may be as dense as 128x128 blocks in black and white or as coarse as 32x32 blocks in color. For our routines, we have used the 128x128 in B/W mode for the Plane plots, and the 64x64 in color mode for the Explosion, Score, and Ground plots. In these modes the face of the television screen is divided into four quadrants corresponding to four different areas of memory. If we consider the Dazzler to be accessing the first 2k bytes of memory, then these quadrants would be numbered as follows:

In decimal:

0 1 2 . . 15	512 . . . 527
16 31	528 . . . 542
: :	: :
240 . . . 255	752 . . . 767
256 . . . 271	768 . . . 783
: :	: :
: :	: :
496 . . 511	1008 . . 1023
1024 . . 1039	1536 . . 1551
: :	: :
: :	: :
1264 . . 1279	1776 . . 1791
1280 . . 1295	1792 . . 1807
: :	: :
: :	: :
1520 . . 1535	2032 . . 2047

In hexadecimal:

0H 0F	200 . . . 20F
10 . . . . 1F	210 . . . 21F
: :	: :
0F0 . . . 0FF	2F0 . . . 2FF
100 . . . 10F	300 . . . 30F
: :	: :
: :	: :
1F0 . . . 1FF	3F0 . . . 3FF
400 . . . 40F	600 . . . 60F
: :	: :
: :	: :
4F0 . . . 4FF	6F0 . . . 6FF
500 . . . 50F	700 . . . 70F
: :	: :
: :	: :
5F0 . . . 5FF	7F0 . . . 7FF

So long as the Object is plotted entirely within one quadrant, the plotting process is fairly straight-forward; but when the Object overlaps two or more quadrants, or overlaps only part of a quadrant, the process becomes considerably more difficult. (Note: Throughout this section, Object applies to either a Plane, an Explosion, or a Score.) In "DOGFIGHT" the way this is handled for Explosions and Planes is quite different. Knowing what we do now about the way the Dazzler does its DMA, we could probably rewrite these routines to work in a more efficient way. Among other things we could alter the size of the Explosions and Planes so that the same plotting routine could be used for either. Remember that since the fundamental size of the "page" is different for B/W or color mode, the Explosions or Planes would still come out as

different sizes.

5.2 Plane Plotting

5.2.1 Introduction -

It was thought that to save RAM space all the planes would be plotted from several basic positions stored in memory. The different orientations would then be derived by rotating or flipping the positions that were stored. This plan was implemented and is described in the following.

5.2.2 Decision Of Plane To Plot -

The first thing done in 'plane' is to decide which plane will be plotted and how to erase the previous plot. This is done by a series of buffer registers. Since 'plane' is only called twice in the program, and since each time is right after an 'updat' call, then it is possible to move consecutive plane positions through the same buffer registers. First the previous X and Y positions ("Pex" and "Pwye") of the plane to be plotted (call it Plane A) are stored in the registers called "Ex" and "Wye"; then the area called "Pln" (to be explained shortly) is cleared, and 'plot' is called. This erases the previous plot of Plane A. Next "Exb" and "Wyeb" are moved into "Pex" and "Pwye" to remain until the next time 'plane' is called, when Plane B is erased. Now "Exa" and "Wyea," the present X and Y locations of Plane A are moved into "Ex" and "Wye" so that the present position of Plane A will be plotted. Finally, "Exa" and "Wyea" are moved into "Exb" and "Wyeb" so that on the subsequent time that 'plane' is called, Plane A will become Plane B. Note that Plane B is never plotted or erased; it simply "becomes" Plane A upon the subsequent time that 'plane' is called. Also, 'updat' and 'tranf' always load the position of the present plane to be plotted into "Exa" and "Wyea." The orientation of the plane is computed in 'tranf,' and the bit sequence explaining this to 'plane' is stored in register "Angle."

5.2.3 Decision Of Orientation Of Plane -

The parameter "Angle" consists of four bits describing the sixteen possible orientations of the plane on the screen. Note that in 'updat' an angle is calculated for the direction of motion of the plane, and that this can have any compass direction, not just one of the sixteen directions that the plane may be pointing. Were I to rewrite 'plane,'

I would incorporate more than sixteen positions; however, it is important to see that the plane can take on infinitely many directions despite the limitation on the number of orientations actually displayed. The four bits of "Angle" each have a logical meaning. Bit 0 determines whether the plane will roll to the left or to the right in an inside upward roll. This is the bit that is "toggled" to cause the plane to "flip over" in 'roll.' Bits 3 and 2 together determine which of the basic plane positions will be used, according to the following:

If N3N2 = 00, uses "ANDN" (meaning: angle down)
N3N2 = 01, uses "ACRS" (meaning: across the screen)
N3N2 = 10, uses "ANUP" (meaning: angle up)
N3N2 = 11, uses "UP" (meaning: point up)

Bits 1 and 0 will be explained in what follows. First, it is necessary to describe the general manner in which planes are plotted.

5.2.4 General Plotting Of Planes -

Planes are plotted using six bytes of memory. Each bit of each of these bytes will cause one square to turn white on the TV. The basic plane orientations are stored in the four memory areas: "ANDN," "ACRS," "ANUP," and "UP." If you were to write out the six bytes stored in each of these in consecutive order under each other, and then were to blacken in a square for each bit that was a "1," you would have the four basic positions of the planes. However, these positions are not in the format which the Dazzler uses. Therefore, they are reformatted by 'refor,' and plotted in a pattern by 'plot' and 'row.' There is also a six-byte parameter area called "Pln," from which all planes are plotted or erased. The pattern of bytes and bits for this parameter area, as well as their relation to the bytes and bits which the Dazzler sees is as follows:

"Pln":

		7	6	5	4	3	2	1	0
Byte 1									
:	2								
:	3								
:	4								
:	5								
Byte 6									

Dazzler:

Byte 1	0	1	4	5					Byte 2
	2	3	6	7					
Byte 3									Byte 4
Byte 5					0	1	4	5	Byte 6
					2	3	6	7	

Since the memory areas "ANDN" through "UP" are fixed values, before reformatting them for other orientations, they are transferred to the parameter area "Pln." There are

four subroutines which control how a memory area is passed: 'UL' for upper-left, 'UR' for upper-right, 'LR' for lower-left, and 'LR' for lower-right. This means that a memory area will be passed to "Pln" starting at one of its four "corners." This will be explained more fully below.

5.2.5 Further Determination Of Plane Orientation -

Next, 'plane' uses N3N2 to determine the starting address of one of the memory areas, which is saved in the BC register pair. Then Bit 0 of "Wye" is tested, and BC is decremented if it is a "1." This is because Y values grow downward on the screen. The other bits of "Wye" come into play when 'adgen' is called. Now N1N0 are used to determine which of the four subroutines: 'UL' through 'LL' is called. After this subroutine is called, Bits 1 and 0 of X are used to determine whether the plane as it is oriented in "Pln" needs to be rotated to the right (or left). This allows for four different X positions within the six-byte Dazzler parameter area described above. 'Rot' is called for each time that it is determined that the bits must be rotated (i.e., rotated as many times as X is: 00 - 11). Finally, when the plane is located in "Pln" parameter area as it will finally be plotted, 'plot' is called to actually plot the plane into memory in such a way that the Dazzler will reproduce it in the desired form. See description of 'adgen' to see how the X and Y of the plane are transformed into an address in memory.

This completes the description of the main part of 'plane'; however, it remains to explain several of the subroutines described above.

5.2.6 'UL,' 'UR,' 'LL,' And 'LR' -

5.2.6.1 'UL' -

This subroutine takes the six bytes in one of the four memory areas: "ANDN," etc. and places them in exactly the same order into the parameter area called "Pln."

5.2.6.2 'UR' -

This subroutine first calls 'UL' and then it calls 'invrs.' 'Invrs' in turn has the effect of rewriting each byte of "Pln" in "inverse " order. That is, Bit 7 becomes Bit 0, Bit 6 becomes Bit 1, etc. for all eight bits of each of the six bytes. This has the effect of reversing the plane from left to right.

5.2.6.3 'LL' -

This subroutine works similarly to 'UL'; however, the six bytes of the memory area are written into the six bytes of the parameter area ("Pln") in reverse order (i.e., Byte 1 becomes Byte 6, etc.). This has the effect of "flipping" the plane over.

5.2.6.4 'LR' -

This subroutine first calls 'LL,' and then calls 'invrs,' exactly similar to 'UR.' This has the effect of both writing the bytes into the parameter area in reverse, and also writing the bits into "Pln" in "inverse" order.

5.2.6.5 Comments On 'UL,' Etc. -

As can be seen from the above descriptions, the time it takes 'plane' to plot planes in different orientations is not always the same. In fact 'UL' takes the shortest, and 'LR' takes the longest amount of time. This I feel to be a disadvantage. Although something was gained in choosing to plot planes in this way (i.e., only four of the plane orientations had to be stored in ROM as opposed to sixteen), I feel the amount of ROM used in reformatting these basic orientations and also the execution time wasted was probably not worth the savings. Knowing what I know now about plotting routines, I feel that the best method would probably be a compromise: store more than four basic plane positions, and do a more limited amount of reformatting.

5.2.7 'Rot' -

'Rot' (i.e., Rotate) is a short loop subroutine which has the effect of rotating each byte of the six-byte "Pln" one bit to the right. This is a right logical rotate (carry flag not included). The way it is called is explained

above.

5.2.8 'Refor' -

'Refor' takes two bytes at a time from "Pln," and Reformats them into the top two bytes in the Dazzler orientation of the bits (see chart above). The bit patterns are as shown in that chart.

5.2.9 'Row' -

'Row' works closely with 'refor' and sets up the values which 'refor' needs to work with. It also calls 'adgen' once to generate the address of the byte in the "second column," as shown in the "Dazzler chart" above.

5.2.10 'Plot' -

'Plot' calls 'row' three times, once for each "row" of the six-byte Dazzler field; it also calls 'adgen' three times, once before each calling of 'row,' to determine the address of the first byte of that row. The address of the second byte is calculated in 'row' itself as explained above. This is also the reason that 'adgen' returns with the D register still intact: this is the "y-offset" value, which must be retained if 'row' is to know where that second byte is to be plotted.

5.2.11 Final Comments On 'Plane' -

It will be seen that the above descriptions become much clearer upon comparison with the assembly listing of the program. Also, it will be apparent that it would not be a large job (at least from the stand-point of the way the PLANES are plotted) to change 'plane' so that the plane size would be different, simply by changing the way or number of times that 'row,' 'plot,' and 'adgen' are called.

5.3 Function Of "Offst" And "Oplus"

"Offst" is a one-byte value which tells both the program and the Dazzler where in RAM the planes and explosions will be plotted (i.e., the 4K "page" which the Dazzler will access). This is done in several ways:

"Offst" is used in 'dazld' to calculate the "turn-on" address which Dazzler will use, it's used in 'adgen' to provide the necessary offset to addresses which are computed there, and it is used to find "Oplus". "Oplus" is simply "Offst"+8H, providing the starting "offset" to the second 2K of the 4K being used. "Offst" is any one-byte quantity; however, only the first five bits (most significant) of it are used in 'adgen,' hence, it ranges in value from 18H to F0H. Note that the program resides in the lowest 18H bytes (along with associated variables), and the "picture" cannot be resident any higher than the top 4K bytes in memory. Thus, if 4K RAM is resident anywhere in the system, the "picture" can be relocated there simply by stopping execution and restoring a new value of "Offst"; then execution must be resumed at the start of the program. Note that the program itself is not relocatable.

5.4 'Adgen'

Although the subroutine 'adgen' is an integral part of 'plane,' it is used so often throughout our program to calculate a 16-bit address that it becomes necessary to describe it by itself. It is one of the most important programs used by DOGFIGHT!. 'Adgen' uses "Offst" and the bits stored in "Ex" and "Wye" to create a 16-bit address in the BC register-pair according to the following format:

where the Bits 10 through 0 have come from the bits of "Ex" and "Wye" as follows:

Bits 15 through 11 come from the five most significant bits of "Offst." You also have the option of "offsetting" the address created, by a specified amount without having to change the values "Ex" and "Wye"; this feature is used in plotting PIanes, say, to generate the addresses of neighboring bytes to the "primary" byte. To take advantage of this option, all that must be done is to store the X-offset (amount of X offset desired) in the C register, and store the Y-offset in the D register before calling 'adgen.' These amounts can be plus or minus. The D register is the only value which 'adgen' returns to you upon completion (except for the 16-bit address left in BC). Note: if you do not want an X- and Y-offset as described above, you still must "zero" the C and D registers.

5.5 Explosion And Score Plotting

Explosions and Scores, unlike the Planes, are both plotted in the upper 2K RAM of the "picture" and are both plotted in the color format of the Dazzler. They therefore both use "Oplus" to calculate addresses for plotting. A large portion of the Main-Program is involved with detecting whether an Explosion must be plotted and consequently whether a Score must be incremented. However, since if no changes were to take place on the color page, it would be desirous to not have explosions or scores be replotted upon consecutive passes through the program, there are provisions for jumps over large parts of this segment of the Main-Program. It would therefore be beneficial to first examine the manner in which these jumps are taken, before explaining the way in which Explosions or Scores are plotted. There are several things to be clarified first.

In 'updat' Bit 7 of "Axms" is set to a "1" as a crash flag: i.e., when Plane A is hit by a bullet. (Similar for Plane B and "Bxms.") This bit is then tested in 'ecall' and is used as an indication of whether or not to plot an Explosion for that plane. Second, there is a memory register known as "Flags," which contains six additional "Flags" which are used in plotting Explosions. In order these are: Bits 7 and 6 are not used; Bits 5 and 4 are "Crash-Flag-B" and "Crash-Flag-A" (CFB and CFA), respectively; Bits 3 and 2 are "Explosion-in-Progress-Flag-B" and "Explosion-in-Progress-Flag-A" (EPFB and EPFA), respectively; Bits 1 and 0 are "Replot-Flag-B" and "Replot-Flag-A" (RFB and RFA), respectively. These will be explained more fully below.

5.6 Explosion Calling

Explosion calling is exactly similar for Planes A and B; in fact the Explosion testing for the two planes follows each other. Hence, everything I say about Plane A below applies in following to Plane B with only the subscripts changed from "A" to "B." First, 'ecall' loads into the BC register-pair two timer values which "time" the Explosions. These values remain in the BC registers throughout 'ecall' unless they are reinitialized. Ordinarily, they are both "0," since neither Explosion is ordinarily plotted. Next, EPFA is tested and if zero, control advances to "C0.". This means that no Explosion is presently in progress. If it is a "1," then Timer B (i.e., the timer in the B register) is decremented. If this results in a non-zero value, then control passes to "Explb" (Explosion-Check B), which means that the explosion has been initially plotted and therefore there is nothing else to be done with Explosion A until

"Replc" (explained below). However, if Timer B has reached "0" following the decrement, then the present Explo A is erased, EPFA is reset and RFB is set. This is because when Explo A is erased, it erases the entire color page, and therefore, if Explo B happened to exist simultaneously with Explo A, Explo B must be replotted. This is done in "Replc."

At "C0" Bit 7 of "Axms" is tested, i.e., the Crash Flag for Plane A. If it is zero, then control passes to "Explb"; if it's a "1," then CFA is tested. If it's not zero, control passes to "Explb"; this is because the Explosion may have "timed out" before the plane actually has touched the ground. In this case we would have a second Explosion whereever the plane happened to be at the time (because Bit 7 of "Axms" is not reset until the plane is reinitialized). To prevent this therefore, the CFA of "Flags" is not reset until the plane is reinitialized. In other words there are two Crash Flags: Bit 7 of "Axms" and CFA, one needed to test whether to plot an Explosion, and the other needed to tell the program not to plot duplicate Explosions elsewhere. Now, if CFA happens to be "0," then the program proceeds to plot the Explosion. This is done by first setting the CFA just tested, then storing the X and Y positions of the Plane into "Savxa" and "Savya." "Savxa" is stored directly from the A register; hence, Bit 7 will still be set as it must have been to get into this routine. This is important to 'eplot.' Finally, the Timer is loaded (with 45D), the EPFA is set, and 'aplot' is called to plot the Explosion. Following this, the A register is cleared to tell 'sound' to output the Explosion Sound from the speaker corresponding to Plane A, and 'sound' is called. In the case of Plane B a non-zero value is left in the A register before calling 'sound' to tell it to output the sound to speaker B.

This concludes the description of "Expla"; next I will undertake the explanation of 'eplot,' and then explain "Replc."

5.7 Explanation Of 'Eplot' And Related Subroutines

5.7.1 'Aplot' And 'Bplot' -

'Aplot' and 'bplot' are exactly similar subroutines for plotting Explosions of Planes A and B. First EPFA is tested; control returns if it is zero. Then A register is saved throughout the routine. The reason for these two things is explained under "Replc". Then "Savya" is stored in "Wyee" for use by 'eplot'; "Savxa" is loaded into the A register, and 'eplot' is called. (The above is also true for 'bplot' and subscript "B.")

5.7.2 'Eplot' -

5.7.2.1 General Remarks -

It was chosen to plot Explosions in a slightly different manner from the Planes; therefore, it is beneficial to examine this more specifically. As can be seen from 'plane' above, Planes are plotted from a six-byte field, wherein the address of the byte is calculated by a call to 'adgen.' However, I desired to plot Explosions with a slightly larger field; therefore, I chose not to calculate the address of each byte in that field. The field chosen was a 6x12 byte field, or 72 bytes. Thus, it is apparent that to call 'adgen' 72 times would be rather slow! It should be remarked at this point that the color "page" plots only two blocks per byte accessed, using four bits to determine one of sixteen possible colors. Thus, while the resolution is reduced to 64x64 "blocks" on the screen, it still requires 2K bytes of RAM per "page." The sixteen colors are in order: 0 through 7H gives black, red, green, yellow, blue, magenta, cyan, and white, all at low intensity; 8H through FH gives the same colors in the same order but at high intensity. We have used only high intensity colors, hence the values 8H through FH only. Thus, two of these values can be put together in a byte to produce two colored blocks side-by-side. Note also that these blocks go from right to left for the byte-values from left to right.

Because it was decided to plot Explosions without calculating the address of each of the 72 bytes, this greatly increased the difficulty of the plotting as far as the different quadrants were concerned.

5.7.2.2 Plotting Of Explosions -

Plotting is actually done by a series of block moves. If the Explosion lies entirely within one quadrant, the address of its first byte is calculated by 'adgen' and the bytes stored in "Explo" are transferred to the 6x12 area by means of a series of LDIR commands. However, Explosions can also be plotted under a series of other conditions: 25 to be exact! These fall into six main categories.

One of the very fortunate side effects of the way Planes are plotted is that there is "wrap-around" from side-to-side and from top-to-bottom. (The top-to-bottom wrap-around is limited in 'updat,' when tests are made to determine if the plane has gone off the top or crashed into the "ground.") Unfortunately, when using block moves to plot

Explosions, it removes this benefit; thus, the program must check when an Explosion is near a quadrant boundary so that it does not plot part of it elsewhere in memory, or plot the other half of it on the other side of the screen. The six main categories mentioned above are: an explosion entirely within one quadrant, an explosion overlapping two quadrants, an explosion overlapping all four quadrants (hence, in the "center"), an explosion in one of the four corners, an explosion along the outer edge of a quadrant but still within that quadrant, and an explosion along the outer edge of a quadrant and overlapping two quadrants. The way these different forms are handled in 'eplot' is by means of a variety of routines. Also, suppression of part of an explosion which will not be plotted if the Explosion is along an outer edge, is taken care of automatically in these routines. Perhaps the most effective way of describing these routines would be to list them along with much used variables and give a brief description of each as follows:

Routines, Subroutines, and Variables

Displ = Displacement - used several times to determine the Displacement of a portion of the Explo.

Colms = Number of columns which the Explo has in its "first" or upper-left-most quadrant. Does not change in 'eplot.'.

Lines = Number of lines which the Explo has in its "first" quadrant; does not change through-out 'eplot.'.

Tcol = Temporary storage of number of lines to be plotted in a particular quadrant; may change with change of quadrant.

Tlin = Temporary storage of number of columns to be plotted in a particular quadrant.

Ex = Begin X value - this is X value of Plane offset back to the left edge of the explosion-square; this and Wye are used to determine upper-left corner of Explo so that center of Explo is plotted where the plane was when hit. Note: Ex and Wye are the same values used by 'adgen.'

Wye = Begin Y value - this is Y value of Plane offset back to the upper edge of the explosion-square.

Bex = Same as Ex, but with value shifted right twice, and taking only four least significant bits.

Bwye = Same as Wye, but with value shifted right once, and taking only five least significant bits.

Taddr = Temporary-address - Storage of the address of the upper-left byte in the quadrant.

Texpl = Temporary-explosion-store - Storage of the address which is the quantity: "Explo" (the start of the Explosion storage in RAM) + "Displ" + 6*K, where K is a value between -1 and 11. (K is incremented in "Lin.")

Xcol = X columns - calculates the initial number of columns in the beginning quadrant; defaults to six if greater than six.

Ylin = Y lines - calculates the initial number of columns in the beginning quadrant; defaults to 12.

Cont0, Cont1, and Nuye = These are "continue control" or "now-wye," during which the above parameters or subroutines are called or initialized; the values are used throughout the rest of 'eplot.' All control passes next to "Quads," described below.

Quads = Examines Bit 6 of Wyee (address-bit 10) and determines whether Explo begins in Top or Bottm half of screen.

Top and Bottm = Examine Bit 6 of Ex (address-bit 9) and determine whether Explo begins in Left or Right half of screen, i.e., they determine the beginning quadrant.

Qud1b = Quadrant-1-Begin - This is the most difficult quadrant to begin from, because the Explo may overlap into any of the other three quadrants. (Note: quadrant 1 is the upper-left quadrant.) Hence, Q1234, Qud12, Qud13, Q12, Case-1, Case-2, Case-3, and Case-4 all plot different portions of the Explosion for different conditions of overlap. Also, each of these routines tests for completion of the Explo, and exits if it is complete.

  Q1234 = Quadrants-1-2-3-4.

  Qud12 = Quadrants-1-2.

  Qud13 = Quadrants-1-3.

  Q12 = Also Quadrants-1-2; however, it actually plots quadrants-1-3 at the far left edge using routines for quadrants-1-2. Note: quadrant 2 is the upper-right quad, quadrant 3 is the lower-left quad, and quadrant 4 is the lower-right quad.

Qud2b = Quadrant-2-Begin - Explo will begin to be plotted in quadrant 2, and may be plotted entirely within that quadrant or plotted in both quadrants 2 and 4.

Qud3b = Quadrant-3-Begin - Explo will begin to be plotted in quadrant 3, and may be plotted entirely within that quadrant or plotted in both quadrants 3 and 4.

Qud4b = Quadrant-4-Begin - Explo will begin in quadrant 4, and will be entirely plotted there; however, only part of it may be plotted if the Explo is along one of the edges of the quadrant. Note that this is the simplest of the plotting routines because it need only plot for one quadrant; no others are possible.

Xoffs = X-offset - Used by 'adgen' to provide an offset to the address it calculates. See 'adgen' for a more complete description of this feature.

Yoffs = Y-offset - Used by 'adgen' to provide an offset to the address it calculates.

Linad = Line adjust - Stores "Lines" in "Yoffs," and finds "Tlin."

Colad = Column adjust - Stores "Colms" in "Xoffs," and finds "Tcol."
Qudr1 = Quadrant-1 - Used any time that it is desired to have the first portion of an Explo plotted in a particular quadrant. Since this often occurs in quadrant 1, it gets that name.
Qudr2 = Quadrant-2 - Used to plot the part of an Explo which occurs in quadrant 2; if called after "Q12," it plots this portion in quadrant 3.
Qudr3 = Quadrant-3 - Used to plot the part of an Explo which occurs in quadrant 3.
Qudr4 = Quadrant-4 - Used to plot the part of an Explo which occurs in quadrant 4; this is called only by "Q1234." This means that it is <u>not</u> for plots which begin in quadrant 4.
Remvx = Remove-X-displacement - This removes the displacement put into "Displ" to enable plotting of quadrant 4 prior to plotting of quadrant 3. Note that in "Q1234" the quadrants are plotted in the order: 1, 2, 4, and 3.
Xdisp = X-displacement - Calculates the displacement to the next quadrant to the right in order to resume plotting there.
Ydisp = Y-displacement - Calculates the displacement to the next lower quadrant in order to resume plotting there. Both this and "Xdisp" are called by the routines explained above.

5.7.2.3 'Quad' -

This is the most important subroutine of 'eplot'; therefore, it will be explained separately. 'Quad' is responsible for doing the block moves and calculating the starting addresses for them. It first calculates the starting address (upper-left corner of portion of Explo to be plotted in a particular quadrant) by using "Xoffs" and "Yoffs"; it also uses "Oplus" to generate the address for the color page. The values "Ex" and "Wye" it uses have been shifted by the right amount to cause the address to be properly generated, despite the reduction in resolution of the 64x64-square page. Next, it calculates "Texpl" and uses it as a line-address throughout "Lin." "Lin" loops once for each time that a "line" of bytes is to be plotted, and uses "Tlin" to compute the number of remaining "lines" to be plotted. It returns when this value reaches zero.

5.7.3 "Erase" And "Clear" -

Because of the large size of the Explosions (72 bytes), to erase them in the same way that 'plane' does would entail having 72 bytes of "0" stored somewhere in RAM. To avoid this waste of memory space, it was decided that to erase explosions, the entire color page would be erased by a block move of 2K times of moving "0." This is done by "Erase," using "Oplus" to tell it to use the color page. A very fortunate additional bonus is that this same move of zeroes can be used to erase the "pages" initially in the program, and also after the words "DOGFIGHT!" are plotted. The way to tell 'eplot' to erase is to clear the A register; this resets Bit 7, which is the signal to 'eplot' to skip to "Erase."

"Clear" uses this feature to clear the two pages of memory. "Oplus" is first set to the same value as "Offst," and 'eplot' is called to erase the B/W page. Then "Oplus" is increased to its value throughout the rest of the program, and the second 2K of RAM is erased. Also, "Clear" initializes the stack pointer each time a new game is started.

5.8 "Replc"

This routine ("Replot-Check") is the closing segment of "Ecall," and is needed because of the nature of the erasing of the Explosions. If an Explosion is erased, the Ground, the Scores, and possibly the other Explosion will have to be replotted again ("Erase" erases the entire color page). Thus, if either RFA or RFB is set, the appropriate subroutine: 'aplot' or 'bplot' is called. Now it becomes clear why the EPFA (or EPFB) is tested in these subroutines, and also why the A register is saved, as mentioned above. In "Gamst" both RFA and RFB are set; this tells "Replc" to plot the Ground (call 'gplot'), but not re-plot or plot either Explosion. If the RF's are both zero, control automatically passes over these checks. Finally, the Timers which have been preserved in the BC register-pair all this time are restored.

5.9 Score Calling

"Score" is similar to "Ecall" in that identical routines are used for both Plane A and Plane B; thus, it will suffice to explain only the checking routines for Score A. First, the D register is cleared and this value is stored in "Displ"; this tells 'numpl' to plot the Score for Plane A. ("Displ" is used as a temporary storage register

to preserve the value in "D" for use in case of "Flash"; it is either a "0" for Plane A or a "1" for Plane B.) Next, "Maxsc" (Maximum Score) is loaded and compared to "Scora"; if "Scora" is less than "Maxsc," then "Pscra" (Previous-Score-A) is compared to "Scora." If "Pscra" is equal to "Scora," then control passes to "Pchek" (Previous-Erase-Check-A); if "Pscra" is less than "Scora," it is incremented and control advances to "Plota," which calls 'numpl' and plots the Score. If "Pscra" equals "Scora," then there ordinarily would not be a need to plot the Score; however, the need for "Pchek" (and likewise, "Pchkb" for Plane B) is to check to see if any Explosions were erased. If so, then both the Scores need to be replotted.

It was mentioned above what would happen if "Scora" is less than "Maxsc"; however, a slight variation takes place for the case where they're equal. In this case "Scorb" is loaded and the subroutine 'maxck' (Maximum-Score-Check) is called. This subroutine subtracts the value ("Scorb" for Plane A) from "Maxsc," and then compares it to "2." Since we already know that "Scora" equals "Maxsc," then if "Scorb" differs from it by more than two, this means that Plane A has won the Game; control passes to "Endit" (described elsewhere). If, however, "Scorb" differs from it by less than two, then Plane A has not won by two, and the "Maxsc" is automatically incremented; control jumps back to the place from which 'maxck' was called. Note that if control jumps out of 'maxck,' the top of the stack is popped in lieu of a "Ret."

5.9.1 "Retrn" - Return To Main-Program -

As the last step completed before control passes again to the beginning of the program, this sequence resets RFA and RFB so that the Ground and Scores will not be replotted on subsequent passes unless there is an Explosion. There is a jump to "Start."

5.9.2 Description Of 'Numpl' -

This subroutine either plots or "blanks" numbers and digits to form the Score. The value passed to it in the A register is the value of Score to be plotted; however, it is in hexadecimal, and must be converted to two decimal digits. This is done by successive subtractions of 10D, and comparisons until the value remaining in A is less than 10D. For each subtraction the E register is incremented by one; thus, the "ones" digit is left in the A register, and the "tens" digit is left in the E register. These are each four

bit quantities given in BCD, and thus they are combined into the A register. Each BCD is then multiplied by 12 (the lower is saved in "LSD" (Least-Significant-Digit); the upper is passed through). This is because 12 bytes are stored in the memory RAM for each of the numbers: 0 through 9, and all zeroes for the purposes of "blanking out" the Score (used in "Endit.") Thus, each BCD is used to calculate the starting address of that number in permanent storage. The value in the D register is next used to determine an offset (if necessary) to Score B. Finally, 'digit' is called twice: once for each digit, to do a block move (LDIR), into the color page. Thus, 'numpl' is basically a "set-up" routine for the block move of 'digit.'

5.9.3 Description Of 'Digit' -

As mentioned above, 'digit' is a block move which uses the address of one Score or the other in the color page, the address of the number in memory storage, and a 12-byte limit to plot a number.

5.9.4 Description Of 'Blank' -

This is an integral part of 'numpl' and its function has already been explained; however, the difference between this and a regular number plot is that 'blank' skips the calculation of two BCD digits. Instead, when it is called, it is given the offset from "Datad" (beginning of the Data-Digits) to the 12 bytes of zeroes. Thus, it will plot blank spaces where there would be a digit.

5.10 'Gplot' - Ground Plotting

'Gplot' uses a block move of two segments to move a string of bytes with the value: AAH (green) to the bottom of the screen. It uses "Oplus" to tell it to plot on the color page.

5.11 Name Plotting

The terms "DOGFIGHT!" and "GAME-OVER" are plotted in an exactly similar manner by a block move. Refer to the diagram showing the four quadrants which Dazzler plots; the two terms are plotted in the very lower portions of each of the upper two quadrants. As can be seen from the figure, the bytes in these two portions are numbered consecutively.

This makes it very easy to do a block move and use the LDIR command. The block move is called once for each of the two quadrants. The beginning of the memory area in ROM in which is stored the letters spelling "DOGFIGHT!" is labeled "Namlt" (meaning "Name letters"), and the beginning of the ROM area in which is stored the letters spelling "GAME-OVER" is labeled "Camlt" (meaning "Game-over letters"). The block move subroutine is called 'block.'

5.11.1 'Block' -

This is a block move of 112 bytes from a RAM area previously (i.e., before calling 'block') determined, to the memory area being examined by Dazzler.

5.12 'Delay'

Both the terms "DOGFIGHT!" and "GAME-OVER" must be displayed for a finite time so that we may see them. This "delay" in viewing time is provided by the subroutine called 'delay.' 'delay' counts down from a value of FFFFH (65,535D) to 0 before returning to the routine which called it. This provides about a 1-second delay when using the 2 MHz. clock, and about a 0.5-second delay when using the 4 MHz. clock.

5.13 Details Following Name Plotting

Following the plotting of "DOGFIGHT!," 'delay' is called twice; then if Bit 7 of the sense-switches is a "1" (i.e., it is up), the program loops and waits for it to be a zero. Upon its becoming a "0," the program continues with 'gamst,' which initializes all necessary parameters and stores the maximum score (which has been input from the sense-switches), and then erases the words "DOGFIGHT!." It then continues with 'init,' 'input,' and plotting of the planes.

The plotting of "GAME-OVER" occurs only when one player has reached the maximum score (which was set when the game began) and when that score has been a win by two. Then the program goes into the routine called 'endit.' First, 'endit' erases the color page, then it plots "GAME-OVER" as previously described. Next it goes into the loop called 'flash.' This loop simply displays the maximum score of the winning player, and "flashes" it on and off by a combination of calling 'numpl' (explained elsewhere) and 'blank' with a suitable delay between each call by using the routine 'delay.' It will looping until sense-switch Bit 7 again

becomes a "1," whence the program proceeds again with the name-plot.

6.0 DAZZLER AND ITS RELATED SUBROUTINES 'DAZLD' AND 'DAZZL'

'Dazld' (Dazzler-Address-Load) sets up the initial addresses, which are output to the Dazzler to tell it which pages of memory to access. 'Dazzl' (or Dazzler-Call) is the subroutine which alternates between the B/W and color pages. Both of these subroutines use the "primed" registers: C', DE', and HL'. In 'Dazld' C' is loaded with the number of output port 14D; E' is loaded with the value which tells the Dazzler to plot black and white in 128x128 mode; and L' is loaded with the value which tells it to plot color in 64x64 mode. The value of "Offst" is used to calculate the "turn-on" addresses for these two pages, which are output to the Dazzler. The address for the B/W page is just "Offst" right-rotated once, and with Bit 7 set (this is important because it tells the Dazzler to turn on). Then 4 (i.e., the value 8H, right-shifted once) is added to this value to generate the "turn-on" address of the color page. Next, 'dazzl' is called, which "exchanges" the primed and unprimed registers, then first outputs the H register to port 14D, and then ouputs the L register to port 15D, and finally restores register C and the non-primed registers. Prior to doing this, however, it exchanges the DE' and HL' registers, thus causing the values that were stored in DE to be output to the Dazzler upon the subsequent time that 'Dazzl' is called.

It will noted from the assembly listing that 'Dazld' is called only at the beginning of a new game, just prior to plotting "DOGFIGHT!". Thus, the program is running in color mode until the next 'Dazzl' call, which occurs just after the first 'updat' call. Note that from here to the next 'dazzl' call is just a very short "time" from the standpoint of the listing; however, it is about equal to the time of the entire rest of the program in terms of actual minutes!

INDEX