```
*******************
*                 *
* STAR RAIDERS II *
*  program notes  *
*                 *
*******************
```

ATARI CONFIDENTIAL!!!

Rev. 1.2

June 16, 1983

by

Aric J. Wilmunder

INDEX

```
****************
*              *
* INTRODUCTION *
*              *
****************
```

I would like to start with a brief explanation of the code that is contained within.

All of the special effects on these disks were developed as tests of the Atari graphics capabilities and not as design documents for SRII. I had been personally interested in writing a Space game and so I began by experimenting with a variety of Space oriented special effects. Some of the techniques used to create these effects are simple, others may be difficult even for me to explain, so please bear with me. I will try to cover all of the techniques as clearly as possible and give examples when necessary.

In case you are not aware, the trench was developed by another programmer, Doug Crockford, and he has given me a copy of parts of his original Source code but he was not able to find all of it. I will try to explain parts of his code that I understand, but it may be necessary to generate new methods to duplicate this effect.

Good luck!

```
*****************
*               *
* GRAPHICS BASE *
*               *
*****************
```

This program serves as the Base Program for all of my special effects. I found that for each effect, I would always need such things as a Display List, System Equates, etc. F.SRC acted as my base program and I was able to INCLUDE any of the other files (except the Trench) and do development work on them.

This may be a big plus when trying to combine these effects because this portion of the code will be common to all.

```
     FILE NAME:    F.SRC

     PROGRAM NAME: ORIGIN

     FUNCTION:     GRAPHICS BASE
```

The code begins by setting up the color registers and then calling a routine called SETUP.

SETUP starts by adding the high byte of the frame buffer address onto the table used to point at each screen line. It then converts the text used on the Text Line into Screen format so I am able to point a DL pointer straight into a line rather than moving the line into a permanent text line.

Next, I erase both frame buffers by calling ERASESPACE with values of #$00 and #$10 in ADDHI. ADDHI determines which screen buffer we will be talking to, $00 is the buffer at $4000 and $10 is the buffer at $5000.

Finally, I set up the Display List and jump to ROTATE. ROTATE was the generic name of my test file. For example, I could simply change my Include from TIE.SRC to PLANET.SRC in order to work on the other effects.

Other routines included in F.SRC are:

DNUMBR   This routine converts a number passed into the A register and using the Y register as a pointer, it converts that number into a decimal representation.

HEXNUM   This one was used to do a real fast display of numbers but in HEX format. It is used only for debugging purposes

DELAY   Passed a value in the A register, this one will wait a non specified period of time.

DELAY2   Passed a value in the A register, this routine will wait A Jiffies and then return. It uses no other registers.

FSPOT3   This is one of my point plotting routines. I have one for each bit combination 00,01,10,11, but the 11 combination is used for greater speed. When putting a 11 combination into a byte, I do not have to clear the two bits first. Compare the code is DSPOTS.SRC of the three routines and I save a few cycles, but when plotting close to 500 points on the Planet, those cycles really add up fast.

FILE NAME: SCTABLES.SRC

PROGRAM NAME: S, C

FUNCTION: SIN AND COS TABLES

These are your basic sin/cos tables with a few twists. For starters, there are only 256 degrees. This makes life a little easier on us 8-bit programmers. Secondly, the table is centered around 128. This means that if a value is less than 128, it is negative and values greater than 128 are positive. You will see this in action in all of the code using the sin/cos function in routines called FXLOC and FYLOC. If a value is over 128 I subtract 127, but if it is under 128, I subtract it from 128.

This is rather bizarre, but you must understand this since most of the effects are based around circles and polar notation of r,theta. In my case theta is an angle from 0-255 which goes directly into an index into the tables, and r is a Fractional Multiply of the value returned. (a F.M. is a multiply that returns a % of one value based on another. Essentially n/255 of x where depending on n you can return values from 0 to x. This will be explained in more detail later.)

Finally, you will note that the sin table is 1/4 the size of the cos table. This is because the tables overlap 75% and the sin table continues into the cos table.

FILE NAME: SYSEQU.SRC

PROGRAM NAME: none

FUNCTION: PAGE 0 VARS AND EQUATES

No explanation here, simply variables use during development. These will have to be rethought for any new development since many have been changed or removed as time went by.

FILE NAME: MULTIPLY.SRC

PROGRAM NAME: MULTIPLY

FUNCTION: FRACTIONAL MULTIPLY

This is an extremely fast multiply routine used for scaling by most of my programs. If you can make this faster (faster than +-80 cycles) you can speed up many of the special effects.

MULTIPLY uses a value in MAG to determine a % of the value in the A register and return it in A. As an example, if you had 0 in Mag, you would receive 0/255 of the value in the A register. If you had 128 in MAG, MULTIPLY would return 128/255 or 1/2 of the value in A.

This routine is used to Scale values up to but not exceeding their original values. It is technically equal to the High Byte of a two byte multiply where the Low byte is thrown away.

I understand that there are even faster ways to perform multiplies through the use of sin/cos tables, but I have not come across any that are faster than this one, but it may be an area worth study.

FILE NAME: MULT48.SRC

PROGRAM NAME: HITAB, LOTAB

FUNCTION: POINTERS TO SCREEN

The tables, HITAB and LOTAB are used to point at the address of the start of each screen line. By using an index into the tables, we will return the address of the start of each line. These tables are used only in the FSPOT and DSPOT routines.

You will notice that HITAB starts and ends with the values 15 or $0F. These are used in the vertical clipping of shapes. They create a buffer at the top and bottom of the screen that point into a garbage collector page (New thought: Why waste ram, point them at a ROM or something???). What occurs is that if an image is larger than the screen, I do not need to do a clip at the immediate edge of the screen, I can have a large overlap. This is also why I used a 48 byte wide screen. I can draw lines past the edge and garbage collect on one line. This is why a divide by 4 table is used in DSPOTS to do my X clipping.

HITAB starts with values between 0 and 15 but during SETUP, I add a #$40 to each of these so they will point directly at one of the screen buffers.

With regards to screen buffers, they are both 4K each and they must be on a memory boundary of 4K where the first is an address whose address is even. I use $4000 and $5000. This is in order that I can use the value ADDHI to tell the plot routines which screen I want to draw on. It also tells the erase routines which screen to erase.

```
     FILE NAME:    DSPOTS.SRC

     PROGRAM NAME: DSPOT1, DSPOT2, DSPOT3, ESPOT

     FUNCTION:     DISPLAY POINTS ON SCREEN
```

All four routines DSPOT1, DSPOT2, DSPOT3, and ESPOT are called exactly the same but they each place a different color on the screen.

The routines are called by placing the x location of the screen in the A register, and the y location of the screen in the Y register. these are calculated similar to the PLOT function in the OS from the upper left corner being 0,0 except that since we have garbage buffers, the real upper left corner is closer to 32,90 where the values from 0-32 and 0-90 are off screen.

Internally, DSPOT creates a 2 byte pointer to the left edge of the row desired. It then places the distance across divided by 4 into the Y register. This points to the specific byte to change. The X register is then set to which 2 bit combination to change and then the points are plotted.

The AND ETAB,X is used on 10 and 01 bit combinations so that the combination is not OR'd with anything other than a 00.

DIV4 is used to find the x location of the desired pixel and it also garbage collects all off-screen data onto the 47 byte line. Anything too big will be plotted but not on the screen. This may be rethought if a better method is found.

```
     FILE NAME:    FLINE.SRC

     PROGRAM NAME: DRAW

     FUNCTION:     DRAW A 17 POINT LINE
```

The reason I can attain such speed is through my fast line algorithm. This allows all lines to be drawn in roughly 1200 cycles vs. 5000+ for other routines.

The theory of the system is to take two points on a plane and find their mid-point. This is found by adding their x's and dividing by 2, and adding their y's and dividing by 2 as well. Adding and dividing by 2 is one of the fastest functions that micros can handle. By loading, clc, adc, and ror, we can find the mid x and y locations of any two points. By doing this repeatedly, we can find mid-points of midpoints and eventually create enough points to call a line. Another advantage is that all lines will have 17 points and take the exact same time to draw every time. This is advantageous when smaller shapes tended to rotate faster since line draws took less time.

The routine DRAW requires that the two endpoints have their coordinates placed in locations XPNT/YPNT and XPNT+16/YPNT+16 respectively. The routine then finds their midpoint and stores it in XPNT+8/YPNT+8, plots the point, and then finds the next set of midpoints. The last set of midpoints do not need to be stored since they are not needed to calculate further. This way, I am able to overlap the Page 0 labels since each only uses the even location from 0-16 and the other is offset by 1.

The routine also calls FSPOT3 since maximum speed is required and placing a 11 bit pattern on the screen is a little faster.

There may be faster ways to do this but anyone is welcome to try. There may also be ways to make it smaller.

```
*****************
*               *
* ROTATE PLANET *
*               *
*****************
```

The original idea for the rotating planet was a program on an apple done by a friend where 2 frames of a rotating planet were placed one on each frame buffer. By swapping buffers, he was able to create the effect except that with only 2 frames, the planet tended to change directions when observed.

After discussion, I tried to emulate the effect on an Atari but using 3 buffers instead of 2. This would create only one direction of motion. After initial success, I then added my Fractional Multiply to allow for different sizes, and I found that a nice planet approach could be developed by increasing planet size on the screen.

This routine still has some bugs, such as the additional planets that appear at extreme magnification, as well as the zooming from the upper left hand corner that has been since changed. Still the effect is worthwhile using but many changes may be necessary.

One idea that occurred to me was that since plotting every point took so much time, first mirror the top onto the bottom. This has been done, not by changing your Display List but by calculating and using the same x value while reflecting the y value through the x axis of the planet. We may also want to speed up time by using my midpoint technique and only calculate every other row and use midpoints for the rows in between.

```
     FILE NAME:    P.SRC

     PROGRAM NAME: ROTATE

     FUNCTION:     ROTATE PLANET
```

The code starts by setting up some one time pointers. The important ones are MAG which is the planet size, DX,DY which are used for momentum and TMPAD0,TMPAD1 which are the center points of the planet.

FRAME is a loop that repeats ad infinitum. It starts by storing a value into PTCTR which holds the number of points used in each frame. These are different on the three separate frames. We may want to change the point tables in order to save some room and gain some speed.

XLOC and YLOC are page zero pointers into the X and Y tables of points to be plotted for each frame. This way I can use a generic program for displaying the planet DFRAME.

DHOT1 and DHOT2 are used to display the hot spots on the planet and these will change if the tables are changed.

DFRAME starts by clearing the count up timer for the purpose of counting the # of Jiffies. Next, it EOR's ADDHI in order to change which frame you are drawing to and calls ERASEFRAME to clear that frame before being drawn.

EACHFRAME displays important numbers on the text line as well as calls the routine to test the keyboard for key presses. Hotspot colors are tied to the timer to give them some shimmer. MAG is used to determine the brightness of the planet. MAG is changed depending on whether the CONSOL keys are pressed, and the planet position is affected by joystick control.

After EACHFRAME, we finally get around to drawing the planet. MAG is used to change coordinates in order for the planet to zoom from its center. BLOOP1 loops down until all points in PTCTR reach zero.

The points are taken from the tables in (XLOC) and (YLOC) and MULTIPLY is used to affect the planet size. The X position is saved in ARG4 since points are mirrored around #216 in order to find the lower portion of the planet.

NODRAW is a delay used to prevent the screens from flipping before the previous planet has been displayed. When the delay is over, the address of the current buffer is stored into the display list causing the frames to change.

Finally, the cross hairs are drawn on the screen, and the # of Jiffies used are displayed at the bottom of the screen.

The KEYTEST routines are of little importance since we will use completely different ones in the future, however, the routine that sets the textline to a message, waits and then changes it back are interesting.

PCOLOR is simply a table used for planet colors at different magnifications

HOTFR1, HOTFR2, HADD1, HADD2 are tables that are currently used for the Hotspots but should be changed for a new system.

FILE NAME:	PLAN.SRC
PROGRAM NAME:	X1,Y1,X2,Y2,X3,Y3
FUNCTION:	PLANET POINT COORDS.

These tables are of the planets x and y coordinates for frames 1, 2, and 3. These are given as cartesian coords but they should be recalculated in order to save space and gain some speed. As it turns out, most of the Y coords are not necessary to duplicate since lines of points exist. We may wish to reorganize the sets of points in order to save space. Data compression of some sort could be used.

FILE NAME: DHOT.SRC

PROGRAM NAME: DHOT1, DHOT2

FUNCTION: DISPLAY HOT SPOTS

Since this code will be completely changed, I will only give a brief description. DHOT1 and DHOT2 first check that the COMPUTER system is on. If not, they simply return.

THOTS is the Total # of Hot spots on the planet. THOTS is used as a counter/pointer into the VERTPOS and HOTFR1/HADD1 tables that give coordinates of Hotspots as well as not displaying them when they are behind the planet. MIDHOT displays the points on the planet if they are visible. It also tests the MAG register and makes the Hotspots larger when the planet is closer.

FILE NAME: DHAIR.SRC

PROGRAM NAME: DHAIRS

FUNCTION: DISPLAY CROSS HAIRS

DHAIRS simply tests whether the computer system is off or on and if it is on, it plots the cross hairs onto the main display.

```
***************
*             *
* TIE FIGHTER *
*             *
***************
```

This is going to be more difficult to explain then most of the other features, but I'll give it a try. The Fighter requires a number of the Graphic Base features in order to run. We will need the Sin/Cos tables, Multiply, and Fast line drawer to name a few.

The Fighter is calculated using what shall be called Cylindrical Coordinates. The best way to imagine this is having points that lie one the radius of a number of circles of varying sizes. Each of these circles lies on a different vertical plane, like stacking plates of varying sizes on top of each other or with space in between. We are able to rotate the stack of plates and they all rotate at the same speed. This is the cylinders THETA. We are also able to tilt all of the plates at an angle towards us. This is referred to as TWIST. Finally we are able to MAGnify the whole works. This gives us depth.

Of the ways I used to save calculations, I only make one calculation for the ALTitudes of each plate even if many points lie on the same plane. Also many points have the same Radius and therefore the same TWIST. These calculations are done only once saving many calculations.

We also store the calculated endpoints into tables and later use a table of XORDR and YORDR telling us which points to connect lines between as well as how many lines there are.

The method found for movement was to calculate the location of a point out in front of the Fighter and this later becomes the Fighter's new center. This creates a Fighter that moves faster when it gets larger.

```
     FILE NAME:    TIE.SRC

     PROGRAM NAME: ROTATE

     FUNCTION:     DRAW AND FLY FIGHTER
```

VARIABLES FIRST:

EPNTS	# OF END POINTS
R	POINTER INTO RTAB1/2
RTAB1	ACTUAL RADIUS BEFORE CALC
RTAB2	RADIUS AFTER CALC
TTAB1	AMOUNT OF TWIST
THETA	ANGLE OF POINTS
ALT	POINTER INTO ATAB1/2

LINES	# OF LINES BETWEEN ENDPOINTS
XORDR	START POINT
YORDR	END POINT
ZOOM	DEPTH MOVEMENT OF SHIP

Start by setting up the center of the ship XPOS and YPOS. Next call SPIRINIT which is used to initialize the starfield. Colors are then set up as well as the initial size of the ship.

SPLOOP is the endless loop that flies the ship around forever. SPLOOP starts by clearing the Jiffy timer so that time measures can be made.

SCALE is used to determine the color of a ship. If it is further away, it should be darker.

Next, you calculate all of the changes to Radii due to SCALE. In the Fighter, there were only 6 different radii that were calculated. TWIST is also calculated which is basically taking a circle and squashing its Y coordinates turning it into an ellipse.

Then, you must find how the ALTitude is affected by SCALE and put the values into a table, ATAB2.

Now that we have done scrunching to save some calculations, we must save the Endpoints of each of the parts of the Fighter.

SCOUNT is a counter used to index into tables holding all of the data on each point. Find the Altitude of this point and place it into TMPALT, find the Radius and put it into MAG, and find the tilt and put it into TWIST. Finally, find the THETA of the point and put it into the Y register.

FXLOC finds the x location of the point. This is done by finding the Cosine of the angle (in Y) and using the Fractional Multiply using the value in MAG, then adding or subtracting this value from TXPOS or the origin of the Fighter. This value is then placed is XLOC

FYLOC is essentially the same as FXLOC except that TWIST is used in MAG in order to turn the circle into an ellipse on the Y axis. TWISTUP vs. TWISTDOWN allows the Fighter to move both up and down. The values is added or subtracted from the center Y coord and this value is placed in YLOC.

Next, the values of XLOC and YLOC are stored into a table holding all endpoint coordinates and the loop continues until all endpoints are calculated.

To draw the shape, we first erase the buffer screen with ERASESPACE and the draw the stars with SPIRAL. Next, we use the tables XORDR and YORDR to find which points connect to others. By grabbing these points, we can do some clipping by comparing them with known outside areas. Then we place these values into the endpoint locations XPNT, YPNT, XPNT+16, and YPNT+16 and call the line draw routine. Repeat this for all lines to be drawn.

Ship movement is done by taking the values of the point in front of the ship and placing them into TXPOS and TYPOS giving us new origin coordinates.

SHIPSPEED simply changes the radius of the point in front of the ship thereby changing the distance that a ship will move each turn.

SSOUND takes the size of the ship, SCALE, and uses it to adjust the volume of the ship when it flies by.

FILE NAME: STARS.SRC

PROGRAM NAME: SPIRINIT, SPIRAL

FUNCTION: MOVE STARS OUT FROM CENTER

In a nutshell, this program initializes STARS # of stars giving them a speed, a radius and an angle. after being initialized, SPIRAL is called to update the stars and when old ones reach a specified radius, they are re-generated as new stars and placed near the center.

Since we will most likely want to use the Star routine from Star Raiders, this code will probably be unused.

```
***********************
*                     *
* ROTATE SOLAR SYSTEM *
*                     *
***********************
```

This is a simple but effective routine using most of the same code used in STARS and TIE. By selecting specific RADII and speeds, we can create the effect of planets rotating around a sun.

Things to note include different orbital speeds for the planets, the faster orbits are closer in, as well as the ability to twist the systems through the x axis.

We may want to add an additional multiply outside of all of the RADII allowing us to Zoom into the Solar System for the planet lock on used when exiting hyperwarp. Another possibility is to use players as some of the planets so we can have ringed planets as well as planets with moons. The Sun should be a Player and its aura should shimmer.

```
     FILE NAME:    SOLAR.SRC

     PROGRAM NAME: ROTATE

     FUNCTION:     ROTATE SOLAR SYSTEMS
```

If you have looked through any of the other rotation code, none of this should be any surprise to you. This code has not been optimized so there may be much of it than can be shared with other routines.

Solar starts by choosing a random # of stars between 2 and 10. This number is stored into the var TURN. ERASESPACE is called to clear the screen and color registers are setup. Next, we find random radii for the stars between 8 and 93 and these are stored into the R table. A speed value is generated from the orbital radius and this value is placed in SSPEED. Finally a random THETA is found. This is repeated for each planet.

In SPLOOP, I start by testing the Trigger for a restart and then I use the center coords of the screen for drawing the Sun. PADDLE3 is used to determine the amount of delay time for slowing rotation speed and then we finally get to plot the planets.

We loop once for each planet first using the Radius and using it in the MAG variable of the Fractional Multiply. The planet's THETA is added to its rotation speed and then replaced into its THETA.

FXLOC and FYLOC are used to find the x and y location of each point, and PADDL2 is used to change the amount of twist on the y axis. When the x and y locations were found, the original code first erased the old location using the old location values that were stored in XOFF and YOFF. This can be changed by simply double buffering the screens and not saving the old locations. This old code also used different colors for the planets depending on radius but this should not be necessary.

```
************************
*                      *
* FIRST PERSON TRENCH *
*                      *
************************
```

I should start by saying that I only have a vague idea as to how Doug created the Trench effect and I have not taken the time to try to figure out his code. I will therefore try to cover some of the important facts that I do understand.

One of the techniques Doug used was to create curved lines giving the effect of flying over a curved planet. This is accomplished by mixing mode B and C graphics lines. Since they both are the same width but of different # of scan lines, by mixing the two together he could make a line bow while using a standard line draw routine. This is not necessary but further study may be warranted.

Another effect was that the Trench took approx. 4 Jiffies to draw. By clocking the Players every Jiffy, we are given the feeling of much greater speed. The players also grow in width and we may want them to get brighter as they get closer.

Doug used a two color Bitmap mode with a floating DLI to change the color of sky to the color of the floor of the trench. We could use more than the one interrupt to create horizon effects and to make colors dimmer in the distance.

My understanding as to the manner used to draw the Trench are somewhat vague. What I believe was done was to find the four corners and plot lines from the corners to the center. Rather than placing these points onto the screen, their coordinates were placed into tables corresponding to each line. For each scan line, there was a value telling you where across the screen to change from one color to another. A second table would tell you when to change back. After these tables were made, Doug used a fill to start from the top and fill the screen to the bottom until the frame was drawn. When this was done, he would switch buffers and start to draw the next frame.

There were a few extra features such as JOYNOISE that would cause the midpoint to move but other than that, those are the basics, and we may want to find another way to do this.

###