IRIDIS ALPHA THEORY

ROB HOGAN

Contents

Bumph

1.0.1 Cover

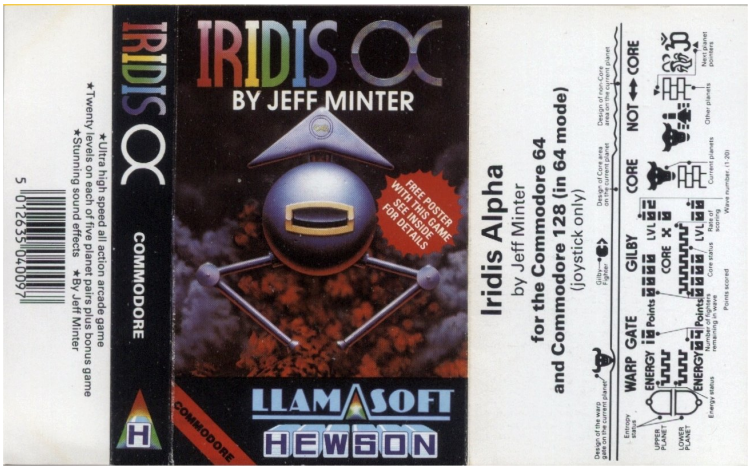
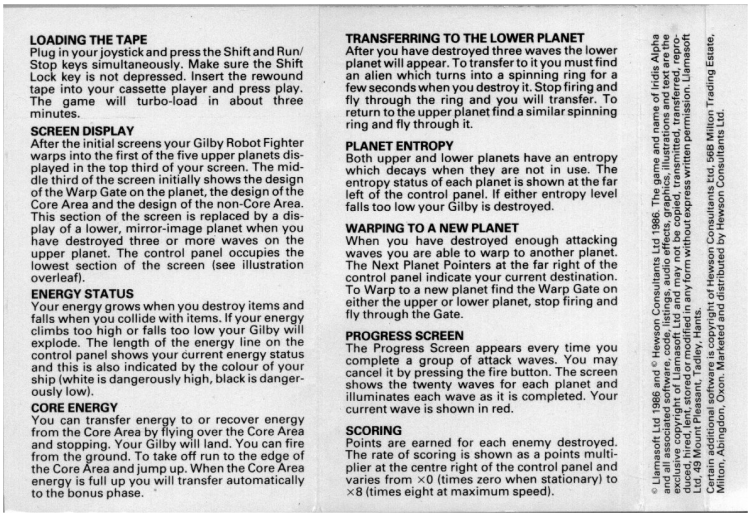


Figure 1.1: Front Cover



LOADING THE TAPE
Plug in your joystick and press the Shift and Run/Stop keys simultaneously. Make sure the Shift Lock key is not depressed. Insert the rewound tape into your cassette player and press play. The game will turbo-load in about three minutes.

SCREEN DISPLAY
After the initial screens your Gilby Robot Fighter warps into the first of the five upper planets displayed in the top third of your screen. The middle third of the screen initially shows the design of the Warp Gate on the planet, the design of the Core Area and the design of the non-Core Area. This section of the screen is replaced by a display of a lower, mirror-image planet when you have destroyed three or more waves on the upper planet. The control panel occupies the lowest section of the screen (see illustration overleaf).

ENERGY STATUS
Your energy grows when you destroy items and falls when you collide with items. If your energy climbs too high or falls too low your Gilby will explode. The length of the energy line on the control panel shows your current energy status and this is also indicated by the colour of your ship (white is dangerously high, black is dangerously low).

CORE ENERGY
You can transfer energy to or recover energy from the Core Area by flying over the Core Area and stopping. Your Gilby will land. You can fire from the ground. To take off run to the edge of the Core Area and jump up. When the Core Area energy is full up you will transfer automatically to the bonus phase.

TRANSFERRING TO THE LOWER PLANET
After you have destroyed three waves the lower planet will appear. To transfer to it you must find an alien which turns into a spinning ring for a few seconds when you destroy it. Stop firing and fly through the ring and you will transfer. To return to the upper planet find a similar spinning ring and fly through it.

PLANET ENTROPY
Both upper and lower planets have an entropy which decays when they are not in use. The entropy status of each planet is shown at the far left of the control panel. If either entropy level falls too low your Gilby is destroyed.

WARPING TO A NEW PLANET
When you have destroyed enough attacking waves you are able to warp to another planet. The Next Planet Pointers at the far right of the control panel indicate your current destination. To Warp to a new planet find the Warp Gate on either the upper or lower planet, stop firing and fly through the Gate.

PROGRESS SCREEN
The Progress Screen appears every time you complete a group of attack waves. You may cancel it by pressing the fire button. The screen shows the twenty waves for each planet and illuminates each wave as it is completed. Your current wave is shown in red.

SCORING
Points are earned for each enemy destroyed. The rate of scoring is shown as a points multiplier at the centre right of the control panel and varies from ×0 (times zero when stationary) to ×8 (times eight at maximum speed).

Figure 1.2: Back Cover

1.0.2 Manual

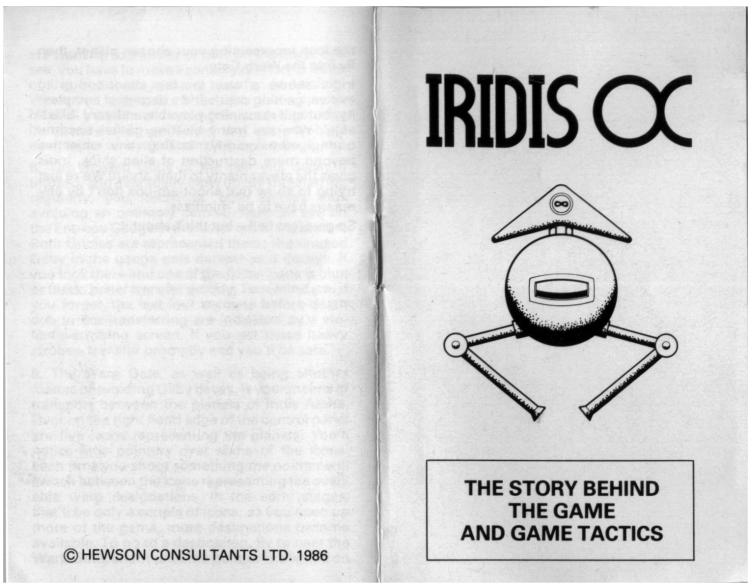
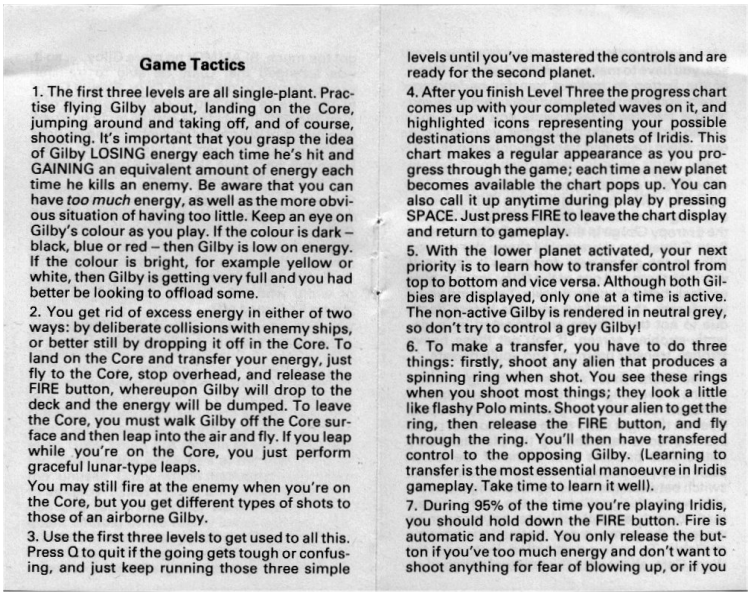


Figure 1.3: Back Cover



Game Tactics

1. The first three levels are all single-plant. Practise flying Gilby about, landing on the Core, jumping around and taking off, and of course, shooting. It's important that you grasp the idea of Gilby LOSING energy each time he's hit and GAINING an equivalent amount of energy each time he kills an enemy. Be aware that you can have *too much* energy, as well as the more obvious situation of having too little. Keep an eye on Gilby's colour as you play. If the colour is dark – black, blue or red – then Gilby is low on energy. If the colour is bright, for example yellow or white, then Gilby is getting very full and you had better be looking to offload some.

2. You get rid of excess energy in either of two ways: by deliberate collisions with enemy ships, or better still by dropping it off in the Core. To land on the Core and transfer your energy, just fly to the Core, stop overhead, and release the FIRE button, whereupon Gilby will drop to the deck and the energy will be dumped. To leave the Core, you must walk Gilby off the Core surface and then leap into the air and fly. If you leap while you're on the Core, you just perform graceful lunar-type leaps.

You may still fire at the enemy when you're on the Core, but you get different types of shots to those of an airborne Gilby.

3. Use the first three levels to get used to all this. Press Q to quit if the going gets tough or confusing, and just keep running those three simple levels until you've mastered the controls and are ready for the second planet.

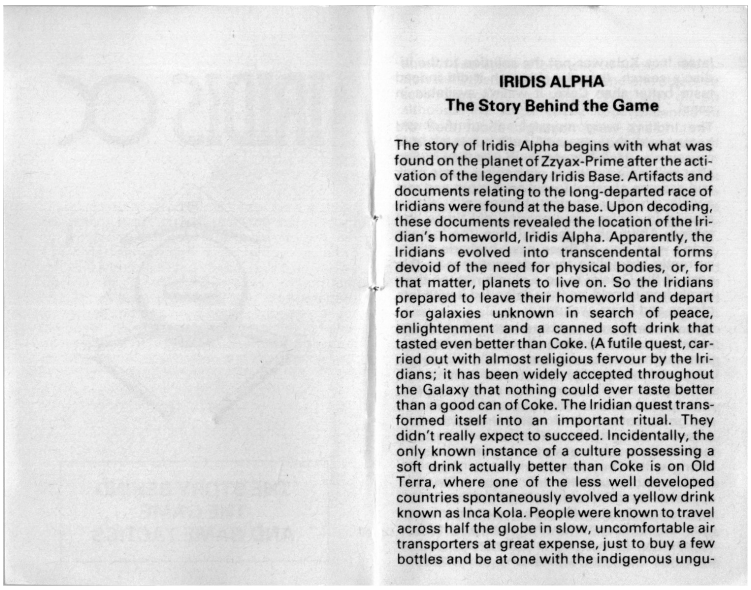
4. After you finish Level Three the progress chart comes up with your completed waves on it, and highlighted icons representing your possible destinations amongst the planets of Iridis. This chart makes a regular appearance as you progress through the game; each time a new planet becomes available the chart pops up. You can also call it up anytime during play by pressing SPACE. Just press FIRE to leave the chart display and return to gameplay.

5. With the lower planet activated, your next priority is to learn how to transfer control from top to bottom and vice versa. Although both Gilbies are displayed, only one at a time is active. The non-active Gilby is rendered in neutral grey, so don't try to control a grey Gilby!

6. To make a transfer, you have to do three things: firstly, shoot any alien that produces a spinning ring when shot. You see these rings when you shoot most things; they look a little like flashy Polo mints. Shoot your alien to get the ring, then release the FIRE button, and fly through the ring. You'll then have transfered control to the opposing Gilby. (Learning to transfer is the most essential manoeuvre in Iridis gameplay. Take time to learn it well).

7. During 95% of the time you're playing Iridis, you should hold down the FIRE button. Fire is automatic and rapid. You only release the button if you've too much energy and don't want to shoot anything for fear of blowing up, or if you

Figure 1.4: Back Cover

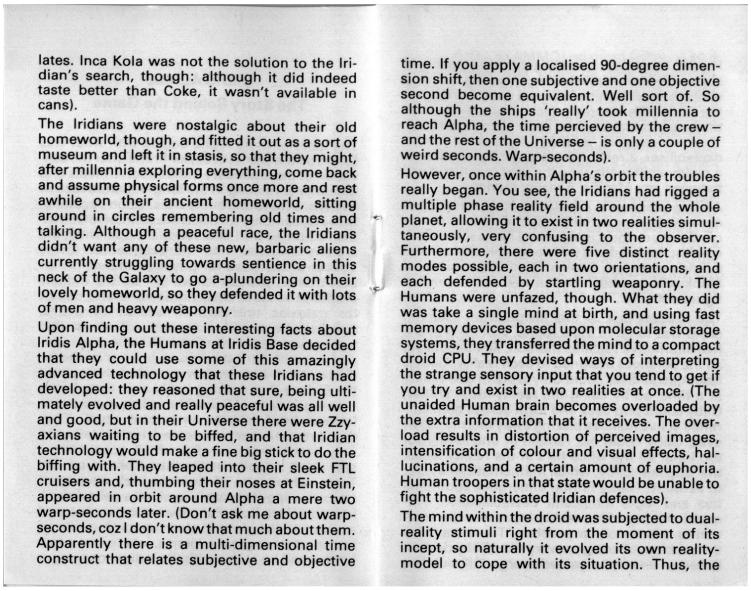


IRIDIS ALPHA

The Story Behind the Game

The story of Iridis Alpha begins with what was found on the planet of Zzyax-Prime after the activation of the legendary Iridis Base. Artifacts and documents relating to the long-departed race of Iridians were found at the base. Upon decoding, these documents revealed the location of the Iridian's homeworld, Iridis Alpha. Apparently, the Iridians evolved into transcendental forms devoid of the need for physical bodies, or, for that matter, planets to live on. So the Iridians prepared to leave their homeworld and depart for galaxies unknown in search of peace, enlightenment and a canned soft drink that tasted even better than Coke. (A futile quest, carried out with almost religious fervour by the Iridians; it has been widely accepted throughout the Galaxy that nothing could ever taste better than a good can of Coke. The Iridian quest transformed itself into an important ritual. They didn't really expect to succeed. Incidentally, the only known instance of a culture possessing a soft drink actually better than Coke is on Old Terra, where one of the less well developed countries spontaneously evolved a yellow drink known as Inca Kola. People were known to travel across half the globe in slow, uncomfortable air transporters at great expense, just to buy a few bottles and be at one with the indigenous ungu-

Figure 1.5: Back Cover



lates. Inca Kola was not the solution to the Iridian's search, though: although it did indeed taste better than Coke, it wasn't available in cans).

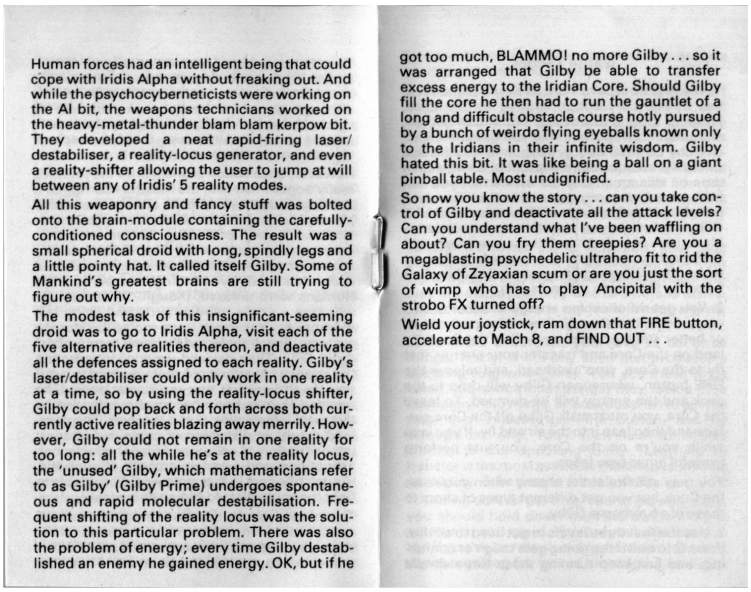
The Iridians were nostalgic about their old homeworld, though, and fitted it out as a sort of museum and left it in stasis, so that they might, after millennia exploring everything, come back and assume physical forms once more and rest awhile on their ancient homeworld, sitting around in circles remembering old times and talking. Although a peaceful race, the Iridians didn't want any of these new, barbaric aliens currently struggling towards sentience in this neck of the Galaxy to go a-plundering on their lovely homeworld, so they defended it with lots of men and heavy weaponry.

Upon finding out these interesting facts about Iridis Alpha, the Humans at Iridis Base decided that they could use some of this amazingly advanced technology that these Iridians had developed: they reasoned that sure, being ultimately evolved and really peaceful was all well and good, but in their Universe there were Zzyaxians waiting to be biffed, and that Iridian technology would make a fine big stick to do the biffing with. They leaped into their sleek FTL cruisers and, thumbing their noses at Einstein, appeared in orbit around Alpha a mere two warp-seconds later. (Don't ask me about warp-seconds, coz I don't know that much about them. Apparently there is a multi-dimensional time construct that relates subjective and objective time. If you apply a localised 90-degree dimension shift, then one subjective and one objective second become equivalent. Well sort of. So although the ships 'really' took millennia to reach Alpha, the time percieved by the crew – and the rest of the Universe – is only a couple of weird seconds. Warp-seconds).

However, once within Alpha's orbit the troubles really began. You see, the Iridians had rigged a multiple phase reality field around the whole planet, allowing it to exist in two realities simultaneously, very confusing to the observer. Furthermore, there were five distinct reality modes possible, each in two orientations, and each defended by startling weaponry. The Humans were unfazed, though. What they did was take a single mind at birth, and using fast memory devices based upon molecular storage systems, they transferred the mind to a compact droid CPU. They devised ways of interpreting the strange sensory input that you tend to get if you try and exist in two realities at once. (The unaided Human brain becomes overloaded by the extra information that it receives. The overload results in distortion of perceived images, intensification of colour and visual effects, hallucinations, and a certain amount of euphoria. Human troopers in that state would be unable to fight the sophisticated Iridian defences).

The mind within the droid was subjected to dual-reality stimuli right from the moment of its incept, so naturally it evolved its own reality-model to cope with its situation. Thus, the

Figure 1.6: Back Cover



Human forces had an intelligent being that could cope with Iridis Alpha without freaking out. And while the psychocyberneticists were working on the AI bit, the weapons technicians worked on the heavy-metal-thunder blam blam kerpow bit. They developed a neat rapid-firing laser/destabiliser, a reality-locus generator, and even a reality-shifter allowing the user to jump at will between any of Iridis' 5 reality modes.

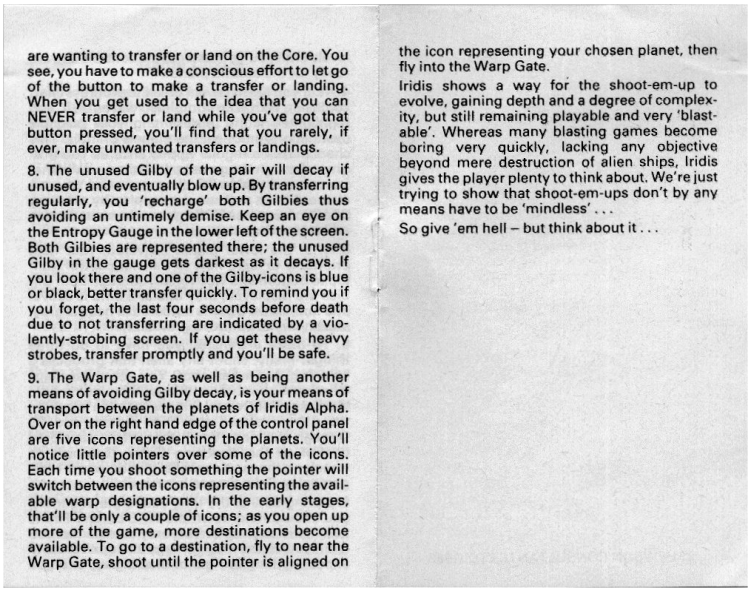
All this weaponry and fancy stuff was bolted onto the brain-module containing the carefully-conditioned consciousness. The result was a small spherical droid with long, spindly legs and a little pointy hat. It called itself Gilby. Some of Mankind's greatest brains are still trying to figure out why.

The modest task of this insignificant-seeming droid was to go to Iridis Alpha, visit each of the five alternative realities thereon, and deactivate all the defences assigned to each reality. Gilby's laser/destabiliser could only work in one reality at a time, so by using the reality-locus shifter, Gilby could pop back and forth across both currently active realities blazing away merrily. However, Gilby could not remain in one reality for too long: all the while he's at the reality locus, the 'unused' Gilby, which mathematicians refer to as Gilby' (Gilby Prime) undergoes spontaneous and rapid molecular destabilisation. Frequent shifting of the reality locus was the solution to this particular problem. There was also the problem of energy; every time Gilby destablished an enemy he gained energy. OK, but if he got too much, BLAMMO! no more Gilby . . . so it was arranged that Gilby be able to transfer excess energy to the Iridian Core. Should Gilby fill the core he then had to run the gauntlet of a long and difficult obstacle course hotly pursued by a bunch of weirdo flying eyeballs known only to the Iridians in their infinite wisdom. Gilby hated this bit. It was like being a ball on a giant pinball table. Most undignified.

So now you know the story . . . can you take control of Gilby and deactivate all the attack levels? Can you understand what I've been waffling on about? Can you fry them creepies? Are you a megablasting psychedelic ultrahero fit to rid the Galaxy of Zzyaxian scum or are you just the sort of wimp who has to play Ancipital with the strobo FX turned off?

Wield your joystick, ram down that FIRE button, accelerate to Mach 8, and FIND OUT . . .

Figure 1.7: Back Cover



are wanting to transfer or land on the Core. You see, you have to make a conscious effort to let go of the button to make a transfer or landing. When you get used to the idea that you can NEVER transfer or land while you've got that button pressed, you'll find that you rarely, if ever, make unwanted transfers or landings.

8. The unused Gilby of the pair will decay if unused, and eventually blow up. By transferring regularly, you 'recharge' both Gilbies thus avoiding an untimely demise. Keep an eye on the Entropy Gauge in the lower left of the screen. Both Gilbies are represented there; the unused Gilby in the gauge gets darkest as it decays. If you look there and one of the Gilby-icons is blue or black, better transfer quickly. To remind you if you forget, the last four seconds before death due to not transferring are indicated by a violently-strobing screen. If you get these heavy strobes, transfer promptly and you'll be safe.

9. The Warp Gate, as well as being another means of avoiding Gilby decay, is your means of transport between the planets of Iridis Alpha. Over on the right hand edge of the control panel are five icons representing the planets. You'll notice little pointers over some of the icons. Each time you shoot something the pointer will switch between the icons representing the available warp designations. In the early stages, that'll be only a couple of icons; as you open up more of the game, more destinations become available. To go to a destination, fly to near the Warp Gate, shoot until the pointer is aligned on the icon representing your chosen planet, then fly into the Warp Gate.

Iridis shows a way for the shoot-em-up to evolve, gaining depth and a degree of complexity, but still remaining playable and very 'blastable'. Whereas many blasting games become boring very quickly, lacking any objective beyond mere destruction of alien ships, Iridis gives the player plenty to think about. We're just trying to show that shoot-em-ups don't by any means have to be 'mindless' . . .

So give 'em hell – but think about it . . .

Figure 1.8: Back Cover

A Little Archaeology

Iridis Alpha was distributed on cassette tape by the publisher Hewson Consultants. Normally used to play audio, cassette tapes were the cheap and ubiquitous medium du jour of the 1980s and a natural choice for the nascent 8-bit game industry to disribute its wares.

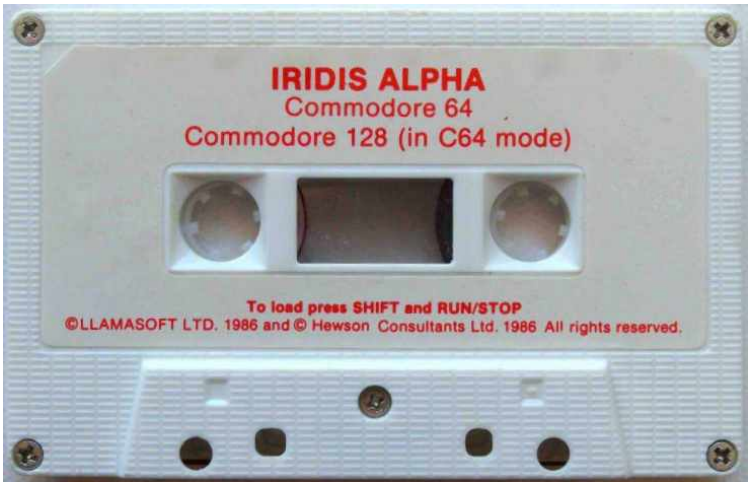


Figure 2.1: It should be simple getting bytes off this right?

Playing a cassette tape for a C64 game such as Iridis Alpha on a normal cassette tape player would be an ear-splitting mistake. Instead of music you would be subjected to a cacophony of mechanical chirruping. This is the tape attempting to convey to you its long stream of binary data in the only language available to it: lots of sound waves

of varying length.

Without knowing how it's actually done, it's tempting to imagine a variety of possible schemes that might have been used. For example, one sound wave denoting a '0' and another one denoting a '1'. The actual method used isn't very far away from such a thing but there is plenty of intricacy layered on top, particularly in a bid to spend as little time as possible loading data from the tape.

In order to get something to work with our first step is to somehow convert the contents of the cassette tape into a binary file so that we can emulate the steps the C64's tape player performed to read the sounds from the tape and load them into memory as something that could be run as a computer program.

The earliest and most durable attempt at standardizing this was from Per Hakan Sundell in 1997. He invented what is now known as the 'tap' format for representing the beeps and bloops encoded on the tape as a file of bits and bytes. The idea is that each byte in the file represents the length of a pulse. It is the length of these pulses that will ultimately tell us whether we should interpret a value of 1 or 0. When we get eights 1s and 0s we have a byte. We get enough bytes, we have a program we can run!

[http://unusedino.de/ec64/technical/formats/tap.html] https://www.c64-wiki.com/wiki/TAP
https://www.luigidifraia.com/doku/doku.php?id=commodore:tapes:loaders:mega-save

Someone, somewhere has kindly decoded the contents of the Iridis Alpha cassette tape distribution to a 'tap' file for us. So we have something to dig into. This is going to be a slightly bonkers journey into the bowels of decoding a 54kb game file from over 500kb of raw data. Every time you think you are nearly done there will be yet another convolution to wrap your head around. But at the end of it we will finally have our binary game file and will be ready to figure out how to decipher it into something approximating the original assembly langage.

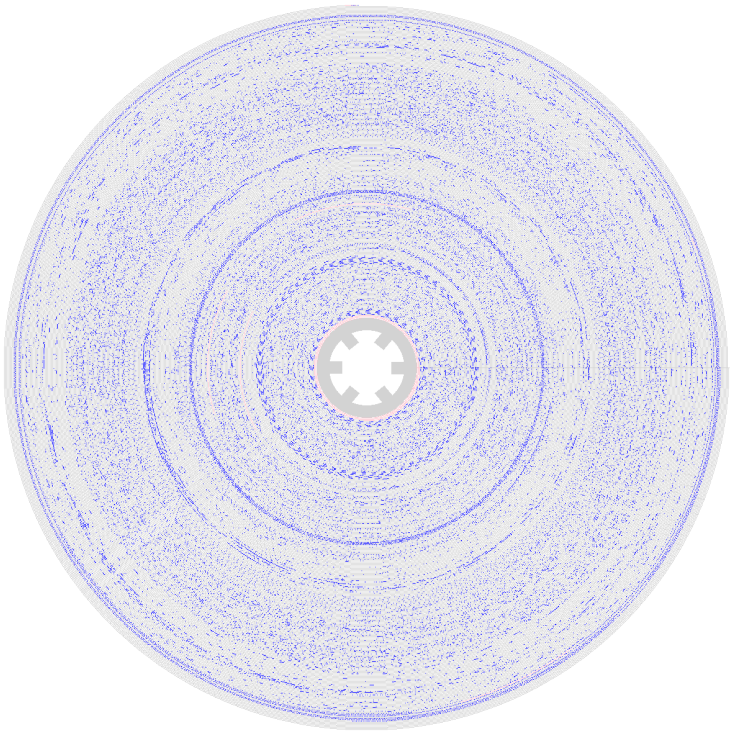


Figure 2.2: Simple, right? Here each short-pulse sound is represented by a light grey pixel, each medium-duration pulse by a blue pixel, and each long pulse by a pink pixel. Roughly speaking, gray is a shorthand for 0 bits and dark pixels for 1 bits.

2.0.1 The Madness Begins

This is what the start of the our `iridis-alpha.tap` file looks like:

43	36	34	2D	54	41	50	45	2D	52	41	57	00	00	00	00
5A	0A	08	00	00	5D	32	2F	30	2F	2F	30	30	31	30	31
30	31	31	2F	31	31	30	31	30	30	31	30	30	30	31	30
31	31	30	31	31	30	31	30	30	31	30	31	31	30	30	30
31	31	31	30	30	31	30	31	31	31	30	30	30	31	31	30
30	30	31	30	31	30	30	30	31	30	31	31	30	30	30	31
31	31	30	30	31	30	32	31	30	31	30	30	32	31	30	30

Figure 2.3: The leading material read by the Megasave loader in the run up to retrieving game data. The Pilot Bytes are in red, the 'Sync Train' in blue, and the Data Header fields from the grey cell onwards.

Field Description	Field Value	Note
TAP Format Header Description	43 36 34 2D 54 41 50 45 2D 52 41 57	'C64-TAPE-RAW' in ASCII
Version	00	Version Number 0
Reserved	00 00 00	Used by format versions ¿ 0
File Size	5A 0A 08	$080A05, in decimal: 526,853 bytes long.
Start of Data	00 00 5D 32 2F 30 2F 2F 30 30 31 30 31 30 31 31 2F 31 31 30 31 30 30 31 30 30 30 31 30 31 31 30 31 31 30 31 30 30 31 30 31 31 30 30 30 31 31 31 30 30 31 30 31 31 31 30 30 30 31 31 30 30 30 31 30 31 30 30 30 31 30 31 31 30 30 30 31 31 31 30 30 31 30 32 31 30 31 30 30 32 31 30 30	Bytes representing the invdividual pulses/sounds on the tape.

Figure 2.4: The meaning of the first batch of data we've read in.

After the header information described above, each byte in the `tap` file represents the pulse length or duration of a single sound emitted by the tape. A pair of sounds taken together represent a single bit, i.e. a `0` or a `1`. A medium length sound followed by a short one represents a `1`, a short one followed by a medium one represents a `0`.

The following table shows us whether we should consider a byte on the tape to represent short, medium, or long duration sound:

Sound Length	Lower Range	Upper Range
Short	$24	$36
Medium	$37	$49
Long	$4A	$64

Figure 2.5: Values for short, medium, and long pulses. For example, ny byte value on the `tap` file between $24 and $36 would be considered a 'Short' pulse.

Remarkably the first 27,000 or so pulses on the Iridis Alpha tape are nothing but short sounds (values between $2F and $31) so cannot be interpreted as anything. It's not until the 27,157th byte in the tape that we start to encounter real data:

```
00006a10: 3031 3031 3156 4144 3130 4231 4243 3130  01011VAD10B1BC10
00006a20: 4231 4131 4244 312f 4057 4032 4231 4231  B1A1BD1/@W@2B1B1
00006a30: 4244 312f 4231 4231 4243 3142 3055 4144  BD1/B1B1BC1B0UAD
00006a40: 3142 3143 3130 4231 4231 4232 4244 3142  1B1C10B1B1B2BD1B
00006a50: 3055 4131 4243 3142 3130 4231 4231 4231  0UA1BC1B10B1B1B1
00006a60: 4244 3130 4056 4143 3230 4143 3130 4231  BD10@VAC20AC10B1
00006a70: 4231 4232 4244 312f 4056 4232 4231 4243  B1B2BD1/@VB2B1BC
00006a80: 3230 4231 4231 4231 4243 3142 2f54 4245  20B1B1B1BC1B/TBE
00006a90: 3141 3130 4231 4231 4230 4230 4243 3030  1A10B1B1B0B0BC00
```

Listing 2.1: Data finally gets started at `56 41` in the first line above.

The first meaningful twenty bytes therefore are:

```
56 41 44 31 30 42 31 42 43 31 30 42 31 41 31 42 44 31 2F 40
```

You get a sense of how wasteful, or ahem redundant, this encoding scheme is when you learn that these twenty pulses are required to give us a single byte. The table below shows how we interpret them to construct a series of 1s and 0s.

First Byte	Second Byte	First Byte Pulse Length	Second Byte Pulse Length	Meaning
$56	$41	Long	Medium	Start of Byte Indicator
$44	$31	Medium	Short	$01
$30	$42	Short	Medium	$00
$31	$42	Short	Medium	$00
$43	$31	Medium	Short	$01
$30	$42	Short	Medium	$00
$31	$41	Short	Medium	$00
$31	$42	Short	Medium	$00
$44	$31	Medium	Short	$01
$2F	$40	Short	Medium	Parity Bit of $00

Figure 2.6: Interpretation of the first 20 meaningful bytes creating a byte. The parity bit at the end is a $00 if there are an odd number of 1s and $01 if there are an even number of 1s. `10010001` has an odd number of 1s.

We can visualize the twenty bytes as a square sound wave. When reading the tape

the C64 would interpret these sound pulses as long, short, or medium to construct a meaning for the entire sequence.

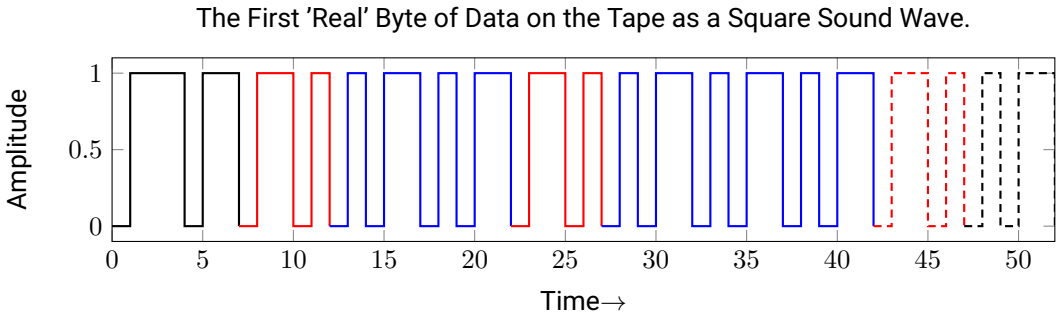


Figure 2.7: Medium-Short pairs in red represent a '1' bit, Short-Medium pairs in blue represent a '0' bit. So this gives us '10010001'. The black wave form at the beginning is the 'Start of Byte' indicator and the one at the end is a parity bit.

Once we've extracted our result of `10010001` from these twenty bytes we now must reverse it. We must do this because the bits are encoded on the tape with the 'most significant bit' first and we are used to reading binary with the 'least significant bit first'. In hexadecimal the reversed bit-string of `10001001` is `$89`.

We have our first byte! It's `89`!

2.0.2 After Our First Real Byte

With this precious commodity in hand we now continue reading off bytes in the same manner from the tape. Eventually we encounter a signal that tells us we've reached the end of the data block, a 'Long-Short' sequence. When this happens we find we've read 202 bytes in total:

89	88	87	86	85	84	83	82	81	03	A7	02	04	03	49	52
49	44	49	53	00	00	00	00	00	00	00	00	00	00	78	A9
6E	8D	06	DD	A2	01	20	D4	02	26	F7	A5	F7	C9	63	D0
F5	A0	64	20	E7	03	C9	63	F0	F9	C4	F7	D0	E8	20	E7
03	C8	D0	F6	C9	00	F0	D6	20	E7	03	99	2B	00	99	F9
00	C8	C0	0A	D0	F2	A0	00	84	90	84	02	20	E7	03	91
F9	45	02	85	02	E6	F9	D0	02	E6	FA	A5	F9	C5	2D	A5
FA	E5	2E	90	E7	20	E7	03	C8	84	C0	58	18	A9	00	8D
A0	02	20	93	FC	20	53	E4	A5	F7	45	02	05	90	F0	03
4C	E2	FC	A5	31	F0	03	4C	B9	02	A5	32	F0	03	6C	2F
00	20	33	A5	A2	03	86	C6	BD	F3	02	9D	76	02	CA	D0
F7	4C	E9	02	A9	07	85	F8	20	D4	02	26	F7	EE	20	D0
C6	F8	10	F4	A5	F7	60	00	00	E4						

Figure 2.8: The data we've read in so far. The unshaded section is machine code.

Fortunately this data has a meaning. It contains the first part of a machine code program the C64 can execute:

Field Description	Field Value	Note
Countdown	89 88 87 86 85 84 83 82 81	Data Block Header
File Type	03	03=PRG, i.e. executable machine code.
Load Address	A7 02	Address to load to: $02A7
End Address	04 03	End Address to load to: $0304
Filename	49 52 49 44 49 53 00 00 00 00 00 00 00 00 00 00	Filename: "IRIDIS"
Machine Code	78 A9 6E 8D 06 DD A2 01 20 D4 02 26 F7 A5 F7 C9 63 D0 F5 A0 64 20 E7 03 C9 63 F0 F9 C4 F7 D0 E8 20 E7 03 C8 D0 F6 C9 00 F0 D6 20 E7 03 99 2B 00 99 F9 00 C8 C0 0A D0 F2 A0 00 84 90 84 02 20 E7 03 91 F9 45 02 85 02 E6 F9 D0 02 E6 FA A5 F9 C5 2D A5 FA E5 2E 90 E7 20 E7 03 C8 84 C0 58 18 A9 00 8D A0 02 20 93 FC 20 53 E4 A5 F7 45 02 05 90 F0 03 4C E2 FC A5 31 F0 03 4C B9 02 A5 32 F0 03 6C 2F 00 20 33 A5 A2 03 86 C6 BD F3 02 9D 76 02 CA D0 F7 4C E9 02 A9 07 85 F8 20 D4 02 26 F7 EE 20 D0 C6 F8 10 F4 A5 F7 60 00 00	This is the machine code of the program to execute. We will translate this back to assembly so we can understand what it does later.
Checksum	E4	How is this calculated?

Figure 2.9: The meaning of the data we've read in so far.

This small program, once we have loaded the rest of it, is where the fun starts. But before we do that we have lots more busywork to do. How about reading in all of the

above data again from the tape? Yup, that's correct. As we keep reading the tape we will find that it contains all of the above data a second time, with the slight difference that the 'Data Block Header' will be `09 08 07 06 05 04 03 02 01` instead of `89 88 87 86 85 84 83 82 81`.

We have to read another 23,000 or so more pulses before we get to something new that we're interested in, the second and final part of the program that we can execute.

When it arrives it looks like this:

89	88	87	86	85	84	83	82	81	A9	80	05	91	4C	EF	F6
A9	A7	78	8D	28	03	A9	02	8D	29	03	58	A0	00	84	C6
84	C0	84	02	AD	11	D0	29	EF	8D	11	D0	CA	D0	FD	88
D0	FA	78	4C	51	03	AD	0D	DC	29	10	F0	F9	AD	0D	DD
8E	07	DD	4A	4A	A9	19	8D	0F	DD	60	20	8E	A6	A9	00
A8	91	7A	4C	74	A4	52	D5	0D	00	00	00	00	00	00	00
00	00	8B	E3	AE	02	53									

Figure 2.10: The data we've read in so far. The unshaded section is machine code.

Field Description	Field Value	Note
Countdown	89 88 87 86 85 84 83 82 81	Data Block Header
Machine Code	A9 80 05 91 4C EF F6 A9 A7 78 8D 28 03 A9 02 8D 29 03 58 A0 00 84 C6 84 C0 84 02 AD 11 D0 29 EF 8D 11 D0 CA D0 FD 88 D0 FA 78 4C 51 03 AD 0D DC 29 10 F0 F9 AD 0D DD 8E 07 DD 4A 4A A9 19 8D 0F DD 60 20 8E A6 A9 00 A8 91 7A 4C 74 A4 52 D5 0D 00 00 00 00 00 00 00 00 00 8B E3 AE 02	This is the rest of machine code of the program to execute.
Checksum	53	How is this calculated?

Figure 2.11: The meaning of the second batch of data we've read in.

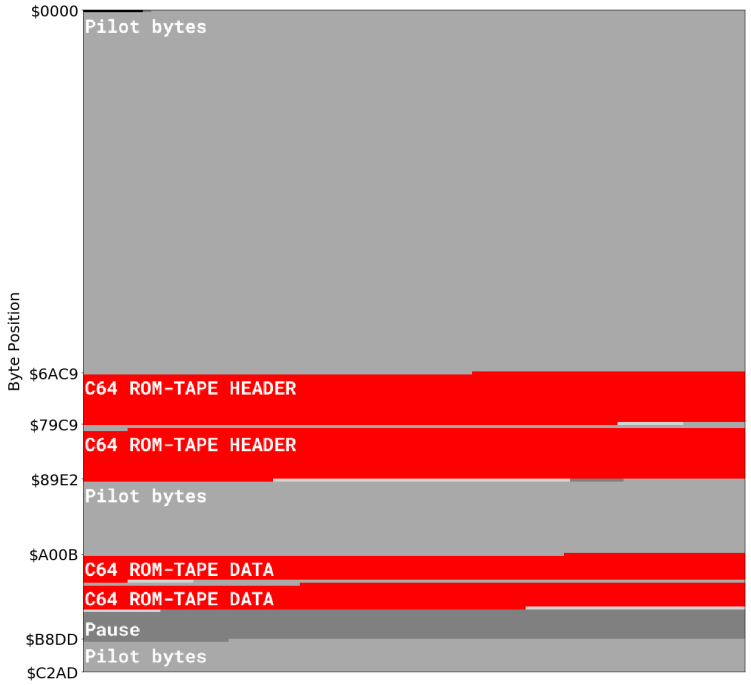


Figure 2.12: The bytes we've read so far from the tape.

Would you be surprised to learn that we have to read in this payload a second time from the tape before we're done? Let's just assume that you're not and let's move swiftly on to wondering what we're supposed to do with 100 or so bytes of data we've finally managed to read after listening to some 50,000 chirrups and clicks from cassette tape.

The answer is simple. We load the data we've received into RAM and execute it. We load the second chunk of data we received at address `$02A7` (this was given in the 'Load Address' field) and the first chunk of data we received directly after it.

What is this program? It seems a bit short to be Iridis Alpha right? What it is is a whole new program for reading the rest of the data from the tape. That's right: we've read all this data from the tape to get a program for reading data from the tape. This type of program is called a 'loader', or perhaps in an effort to justify it's existence, a 'turbo loader'.

2.0.3 A Loader for your Loader

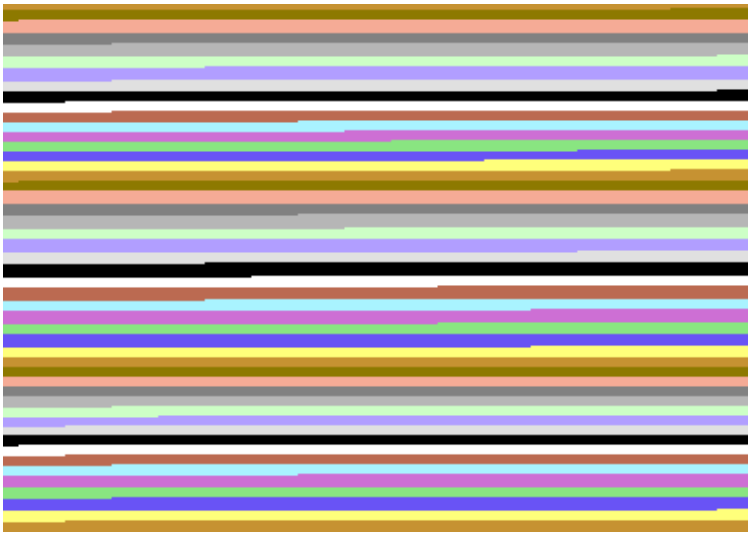


Figure 2.13: MegaSave loader pictured.

The idea is that this little program will do a better job of reading data from the tape and more quickly than the C64 can manage by itself. There is a whole menagerie of these programs that proliferated in the 1980s with exotic names such as Jetload, Easytape, Audiogenic and so on. Luigi di Fraia maintains a utility called tapclean that does a great job of identifying and emulating these loaders and thanks to him we have a disassembled version of the loader we've just found on the Iridis Alpha tape. It has the quintessentially 1980s moniker 'MegaSave' and as you can see in the listing reproduced below is relatively compact.

```
; This disassembly has been adapted from
; https://www.luigidifraia.com/doku/doku.php?id=commodore:tapes:loaders:mega-save
;
; From: Cauldron
; Note: Assemble with 64tass

; **************
; * CBM Header *
; **************

*=$033C

; Cassette I/O Buffer - Header
T033C   .byte $03,$A7,$02,$04,$03,$49,$52,$49,$44,$49,$53,$00,$00,$00,$00,$00
T034C   .byte $00,$00,$00,$00,$00

AlignAndSynchronizeLoop
        SEI

        LDA #$6E        ; Set the read bit timer/counter threshold to 0x0107
        STA $DD06
```

```
        LDX #$01

; Byte-align with the incoming stream by shifting bits in until the first pilot byte is read

_align  JSR rd_bit    ; Read a bit

        ROL $F7       ; Shift each of them into the byte receive register, MSbF
        LDA $F7
        CMP #$63      ; Until the first occurrence of 0x63 (pilot byte) is received
        BNE _align

        LDY #$64      ; Pre-load the first expected sync byte in Y

; Read the whole pilot sequence

_pilot  JSR rd_byte   ; Keep reading bytes until the pilot train is over
        CMP #$63
        BEQ _pilot

; Check that the sync sequence is as expected (Note: $F7 = byte read by means of a call to rd_byte)

_sync   CPY $F7       ; Is the currently expected sync byte following?
        BNE _align    ; Start over if not

        JSR rd_byte   ; Read byte

        INY           ; Bump the expected sync byte value
        BNE _sync     ; Read the whole sync sequence (0x64-0xff inclusive)

        CMP #$00      ; In absence of issues A is set to 0x01 here
        BEQ AlignAndSynchronizeLoop

; Read and store file header

_header JSR rd_byte   ; Read 10 header bytes
        STA $002B,Y   ; Overwrite BASIC program pointers
        STA $00F9,Y   ; Also store them in RAM where they will be changed
        INY
        CPY #$0A
        BNE _header

        LDY #$00
        STY $90       ; Zero the status flags (not set by this code)
        STY $02       ; Zero the checkbyte register

; Read and store data from tape into RAM

_data   JSR rd_byte   ; Read a byte
        STA ($F9),Y   ; Store in RAM

        EOR $02       ; Update the checkbyte value
        STA $02

        INC $F9       ; Bump the destination pointer
        BNE _check_complete
        INC $FA

_check_complete
        LDA $F9       ; And check if the file is complete
        CMP $2D       ; by comparing the destination pointer
        LDA $FA       ; to its one-past-end value
        SBC $2E
        BCC _data

        JSR rd_byte   ; Read a byte

        INY
        STY $C0       ; Control motor via software

        CLI

        CLC

        LDA #$00      ; Make sure the call to $FC93 does not restore the standard IRQ
        STA $02A0
        JSR $FC93     ; Disable interrupts, un-blank the screen, and stop cassette motor

        JSR $E453     ; Copy BASIC vectors to RAM

        LDA $F7       ; Compare saved and computed checkbytes
        EOR $02
        ORA $90       ; And check that the status flags do not indicate an error
        BEQ *+5

        JMP $FCE2     ; Soft-reset if any of the above checks fails
```

```
; Code execution after a file is completely loaded in

        LDA $31       ; Check flag #1
        BEQ *+5       ; If not set move on
        JMP J02B9     ; Otherwise re-execute the loader

        LDA $32       ; Check flag #2
        BEQ *+5       ; If not set move on in order to issue a BASIC RUN command
        JMP ($002F)   ; Otherwise execute a custom routine pointed by the vector $2f/$30

        JSR $A533     ; Relink lines of tokenized program text

        LDX #$03      ; Set the number of characters in keyboard buffer to 3
        STX $C6

        LDA T02F4-1,X ; And copy R, <shift> + U, <return> into the buffer
        STA $0276,X
        DEX
        BNE *-7

        JMP J02E9

; --------------------------------

; Read byte: read 8 bits from tape, grouping them MSbF
;
; Returns: the read byte in A

rd_byte LDA #$07      ; Prepare for 8 bits
        STA $F8       ; Using $f8 as a counter

B03EB   JSR rd_bit    ; Read a bit

        ROL $F7       ; Shift each of them into the byte receive register, MSbF

        INC $D020

        DEC $F8       ; And loop until 8 bits are received
        BPL B03EB

        LDA $F7       ; Return read byte in A

        RTS

        .cerror * > $03FC, "The CBM Header code is too long to fit in the tape buffer!"

; --------------------------------

        .align $03FC, $00 ; Padding


; ************
; * CBM Data *
; ************

*=$02A7

NMIH    LDA #$80
        ORA $91
        JMP $F6EF

J02AE   LDA #<NMIH

        SEI

        STA $0328     ; Disable <Run Stop> + <Restore>
        LDA #>NMIH
        STA $0329

J02B9   CLI

        LDY #$00
        STY $C6       ; No char in keyboard buffer
        STY $C0       ; Enable tape motor
        STY $02       ; Zero the checkbyte register (also done later on)

        LDA $D011     ; Blank the screen
        AND #$EF
        STA $D011

        DEX           ; Wait a bit
        BNE *-1
        DEY
        BNE *-4
```

```
        SEI

        JMP AlignAndSynchronizeLoop     ; Execute the main loader code

; ---------------------------------

; Read bit: loops until a falling edge is received on the read line and uses
;           the read bit timer/counter to discriminate the current bit value
;
; Returns: the read bit in the Carry flag

rd_bit  LDA $DC0D     ; Loop until a falling edge is detected
        AND #$10      ; on the read line of the tape port
        BEQ rd_bit

        LDA $DD0D
        STX $DD07
        LSR
        LSR           ; Move read bit into the Carry flag

        LDA #$19      ; Restart the bit read threshold timer/counter
        STA $DD0F

        RTS

; ---------------------------------

J02E9   JSR $A68E     ; Reset pointer to current text character to the beginning of program text

        LDA #$00
        TAY
        STA ($7A),Y

        JMP $A474     ; Print READY

; ---------------------------------

; Characters to be injected in the keyboard buffer, if required

T02F4   .byte $52,$D5,$0D ; R, <shift> + U, <return>

        .cerror * > $0300, "The CBM Data code is too long to fit in front of the vector table!"

; ---------------------------------

; Overwrite BASIC vectors in RAM

        .align $0300, $00 ; Padding

T0300   .word $E38B ; Leave IERROR unchanged
        .word J02AE ; Autostart the turbo loader, once loaded, by overwriting IMAIN
```

Listing 2.2: The data we have just loaded translated back into assembly language. This is the MegaSave loader.

What does MegaSave do that makes it so much faster than the default C64 tape loader? The simple answer is that it cuts corners and strips away all of the cautious redundancy we observed when loading the loader itself. Instead of reading 20 bytes (or pulses) from the tape in order to construct a single byte it just needs 8. It does what we naively thought at the beginning might be the way to read data from the tape: a long pulse is a 0, a short pulse is a 1, you read 8 of them you have the 8 bits for your byte.

This simplicity is risky. With no repetition of data blocks, no parity check on each individual byte, and no delimiters between bytes, the loader is vulnerable to corruption of the tape itself or any hardware flakiness in the cassette reader. The fact that it generally works is simply due to a lack of conservatism paying off in practice. In addition, the MegaSave loader isn't totally bereft of precautions. There is a slightly elaborate dance it goes through to gain some assurance that the tape medium is going to yield

a reliable string of bytes.

The first thing it does is look for a sentinel value of $63.

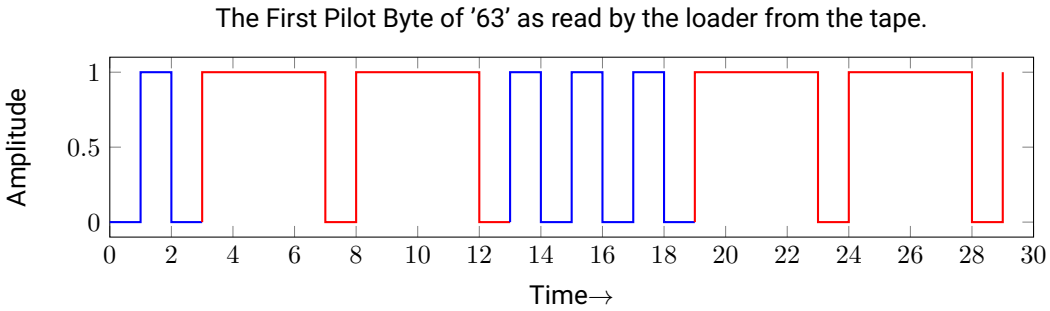


Figure 2.14: Long square waves in red represent pulses giving a '1' bit, Short square waves in blue represent a '0' bit. So this gives us '01100011' i.e. $63. Unlike the default tape loader, MegaSave expects the 'most significant bit first' - which is the natural way of representing bits on paper so we don't need to reverse the bits to 'make sense' of them.

.

There is an inordinately long string of these, followed by an ascending sequence of byte values from $63 to $FF. This sequence has the catchy name of a 'Sync Train':

63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63	63
63	63	63	63	63	64	64	65	66	67	68	69	6A	6B	6C	6D	6E
6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	90
91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	A0	A1
A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0	B1	B2
B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2	C3
C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	D0	D1	D2	D3	D4
D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	E0	E1	E2	E3	E4	E5
E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	F0	F1	F2	F3	F4	F5	F6
F7	F8	F9	FA	FB	FC	FD	FE	FF	01	00	08	FF	BF	00	00	01
02	CA	00														

Figure 2.15: The leading material read by the Megasave loader in the run up to retrieving game data. The Pilot Bytes are in red, the 'Sync Train' in blue, and the Data Header fields from the grey cell onwards.

The bytes after the blue cells above are our first bit of raw meat in a while. Here's what they mean:

Field Description	Field Value	Note
Header Sentinel	01	Indicates the start of the header, expected to be non zero.
Load Address	00 08	Address to load to: $0800
End Address	FF BF	End Address to load to: $BFFF
Execution Address	00 00	Filename: ”IRIDIS”
Next Action Indicator	01	Resume loading data when done or execute the loaded data.
Execution Type	02	How to execute the code.

Figure 2.16: The meaning of the Data Header values read in by MegaSave.

What the loader can garner from this is that the data that follows should be read in and stored at $0800 and that rather than execute it straight away it should then resume loading more data from the tape.

The entire game is stored across four separate chunks on the tape. Once it has loaded this first one, the loader reads in the next three chunks.

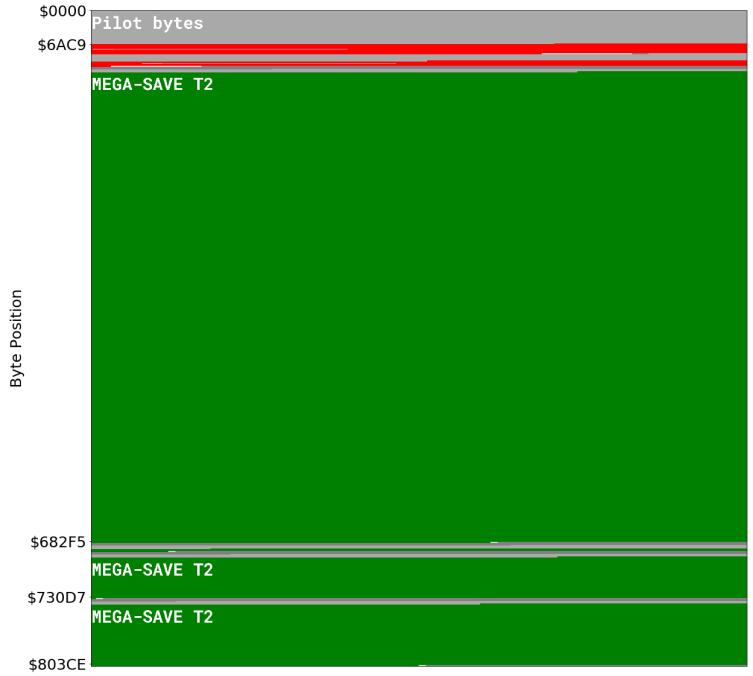


Figure 2.17: All the data that has been read from the tape. The four chunks of game data are in green the third is only a sliver. The relative sizes of the red data (the MegaSave loader which is only actually 200 or so bytes long) and the green data (representing over 50,000 bytes of game data) illustrates how efficient the MegaSave loader's storage is by comparison with the default.

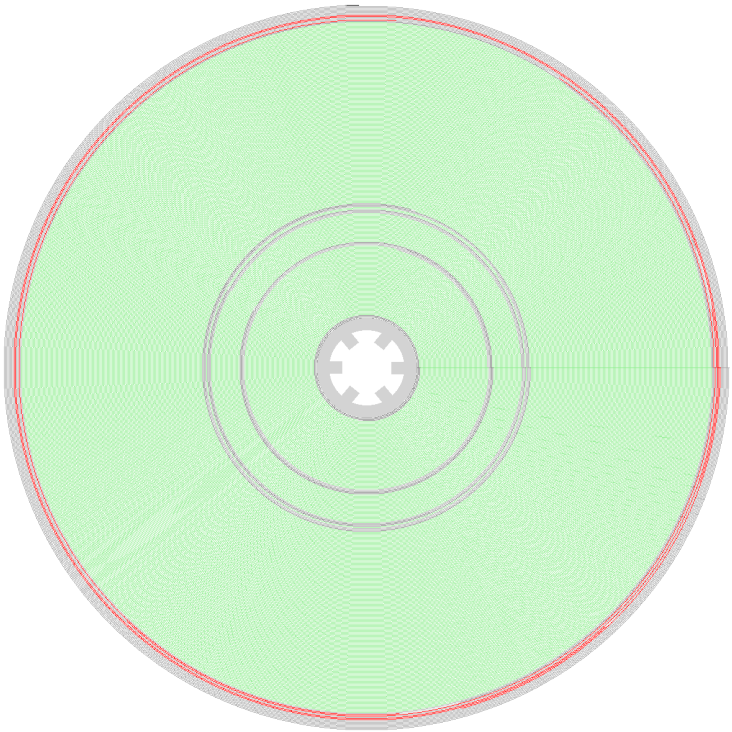


Figure 2.18: Our image of the spool from the start of this chapter but this time with each section colored in as described in the previous image.

When it has completed it has loaded the following to RAM:

Start Address	End Address	Note
0800	BFFE	.
BF00	BFFF	.
C000	CFFE	.
E000	F7FF	.

Figure 2.19: The four chunks of game data.

Figure 2.20: Where the different parts of the game end up in memory.

2.0.4 Putting an End to the Madness

With all the data read in you might wonder how the loader knows what to do next (i.e. to run the game) and how it will know where to start running it from. The answer is given in the `Header Data` for the final chunk of data:

Field Description	Field Value	Note
Header Sentinel	01	Indicates the start of the header, expected to be non zero.
Load Address	00 E0	Address to load to: $E000
End Address	00 F8	End Address to load to: $F800
Execution Address	10 08	Address to start execution at: $0810.
Next Action Indicator	00	00 means start executing code, don't read any more data.
Execution Type	02	How to execute the code: 01 means used the address given in 'Execution Address'.

Figure 2.21: The meaning of the Data Header values in the final chunk of data read in by MegaSave.

So according to the header data, the loader should stop reading data now and start executing what has already been loaded, and it should start doing this at address $0810.

```
*=$0810
StartExecution
        SEI
        ; Tell the C64 to execute the code at MainControlLoop
        ; the next time an interrupt happens.
        LDA #>MainControlLoop
        STA $0319    ;NMI
        LDA #<MainControlLoop
        STA $0318    ;NMI

        ; Turn off the tape deck.
        LDA #$10
        STA $DD04    ;CIA2: Timer A: Low-Byte
        LDA #$00
        STA $DD05    ;CIA2: Timer A: High-Byte
        LDA #$7F
        STA $DD0D    ;CIA2: CIA Interrupt Control Register
        LDA #$81
        STA $DD0D    ;CIA2: CIA Interrupt Control Register
        LDA #$19
        STA $DD0E    ;CIA2: CIA Control Register A
        CLI
LoopUntilExexcutes
        JMP LoopUntilExexcutes
```

Listing 2.3: The first piece of code that is executed in Iridis Alpha.

This routine does two things: it turns off the tape deck and tells the C64 to execute the code at a different location (`MainControlLoop`) the next time it wakes up and wonders what to do. This will be in a few microseconds time. When it starts executing `MainControlLoop`, Iridis Alpha will get underway.

Figure 2.22: A `prg` is born.

Some Disassembly Required

We've reached the point where the game has started to execute. We just saw a snippet of code that turned off the tape recorder and prompted the C64 to run a routine a called `MainControlLoop`. Though perhaps a little cryptic, and you shouldn't expect to understand it yet, this code is not exactly what the machine 'saw'. Instead it read and executed something far more puzzling looking:

```
78A9 408D 1903 A900 8D18 03A9 108D
04DD A900 8D05 DDA9 7F8D 0DDD A981
8D0D DDA9 198D 0EDD 584C 3508
```

Listing 3.1: The first piece of machine code that is executed in Iridis Alpha.

That's right it executed a stream of bytes. A stream of bytes commonly referred to as 'machine code'. Each of these bytes is meaningful to the C64 whether individually, or taken in pairs, or groups of three. It can comprehend them as instructions to carry out that will shuffle data around in its memory and ultimately result in a game that can be played.

Before we can dig into the internals of how Iridis Alpha works we have to convert all of the machine code we've loaded into memory in the previous chapter into something we have a chance of reading and understanding. This process is called disassembly and here we're going to explain how it is done and along the way gain a little basic understanding of the human-readable language, called 6502 Assembly Language, that we convert the machine code back into.

The process is called disassembly simply because it is the exact reverse of the process that was originally followed to generate the data on the tape from the assembly language written by Jeff Minter in the first place. Programs that do this are referred to as 'assemblers'. They assemble the instructions written by the programmer into ma-

chine code that the C64 can execute. As self-appointed disassemblers we are going to turn it back into assembly language.

Along the way we will have to invent names for things that are meaningful to us: the names Jeff Minter gave to his functions, routines, and variables are long since lost to us. They were thrown away by the assembler as unnecessary for execution of the machine code. As we proceed, we will see why but it hopefully makes sense to say that the 6502 CPU in the C64 doesn't care what things are called it just cares where in the 65,632 bytes of its RAM things are located. That is to say it only cares about their 'address'.

So let's start by stepping back for a second. Let's put the machine code and the snippet of code we disassembled it back into and see what we can learn:

```
78
A9 40
8D 19 03
A9 00
8D 18 03
A9 10
8D 04 DD
A9 00
8D 05 DD
A9 7F
8D 0D DD
A9 81
8D 0D DD
A9 19
8D 0E DD
58
4C 35 08
```

Listing 3.2: Machine Code

```
SEI
LDA #$40
STA $0319
LDA #$00
STA $0318
LDA #$10
STA $DD04
LDA #$00
STA $DD05
LDA #$7F
STA $DD0D
LDA #$81
STA $DD0D
LDA #$19
STA $DD0E
CLI
JMP $0835
```

Listing 3.3: Assembly Language

Straightaway we can infer meanings to some of the bytes:

Byte	Instruction
78	SEI
A9	LDA
8D	STA
58	CLI
4C	JMP

Figure 3.1: Machine code bytes and their assembly language counterparts.

Let's quickly explain what two of these mean as they're extremely common and also fundamental to machine code programming in general.

`LDA` loads the byte you give it into a small one-byte slot called the 'Accumulator' ('A' for short so `LDA` is an abbreviation for 'Load to Accumulator'). Think of this slot as a pocket - somewhere for the CPU to store a value for use later on. The technical name for this kind of slot/pocket is a 'register'. The 'Accumulator' is so-called because a lot of the operations performed on bytes stored in this particular pocket have very precise behaviour when addition operations are executed on it. But we never really have to worry about this too often and in the general case it is simply used, as here, as a place to put a value so we can do something else with it.

```
LDA #$40
```

Listing 3.4: Loading the byte $40 to the `Accumulator`.

What we do with it for the most part in the code above is go on to store it somewhere else. This is where the `STA` instruction comes in. This stores the value in the 'Accumulator' pocket at the address in memory that you give to it. As we can see such addresses are not one byte long, but two bytes. Are they ever more than two bytes long? No, and for a very simple reason. The C64 can only understand addresses that are at most two bytes long and this is what ultimately limits it to 64KB of memory. The largest address it can understand is therefore `$FFFF` - the largest value that can be expressed by 2 bytes. Which translates to 65,532. 65,532 bytes is 64KB of RAM.

When we look at the disassembly of the STA instructions we see something quite puzzling:

```
8D 19 03
```

```
STA $0319
```

Shouldn't we have expected `8D 19 03` to translate to `STA $1903` rather than `STA $0319`? Why are the numbers back to front like that? The reason is due to something

you may have heard described with a word before that you've never fully understood and maybe never dared to question. The machine code stores the address `0319` as `1903` because the 6502 CPU in the C64 expects to read addresses with the second half of the number first. When we read numbers we expect it to start with the most significant digits first, i.e. `$0319`. But most computer architectures of the 1980s had the opposite expectation - reading the least significant digits first, or the least significant bits first since this is a computer we're talking about, i.e. `$1903`. This approach possesses a couple of advantages, the main one being that the computer can start performing an operation on the number (e.g. addition) before it has read all of it. For example, if you're adding 5312 and 2043 you already have enought to get started with once you've read 12 and 43. The carrying etc. can all happen once you've read the rest of the two numbers. This approach is known as 'little-endian' byte ordering. The opposite, which is closer to what we are familiar with when reading numbers ourselves, is called 'big-endian'. If you had never heard of either of those terms before, then you have now.

The next incremental step in understanding our disassembly is to apply some meaning to these two-byte values that we've decoded. As it happens, all the ones given in this piece of code represent addresses in the C64's memory of `$FFFF` bytes, or in the preferred parlance of we humans, 65,532 bytes. Each address here has a particular function so storing a value in it does something. Let's add the meanings, which are still a little cryptic, to our listing:

```
SEI
LDA #$40
STA $0319
LDA #$00
STA $0318
LDA #$10
STA $DD04
LDA #$00
STA $DD05
LDA #$7F
STA $DD0D
LDA #$81
STA $DD0D
LDA #$19
STA $DD0E
CLI
JMP $0835
```

Listing 3.5: Assembly

```
SEI
LDA #$40
STA $0319    ;Non-Maskable Interrupt
LDA #$00
STA $0318    ;Non-Maskable Interrupt
LDA #$10
STA $DD04    ;CIA2: Timer A: Low-Byte
LDA #$00
STA $DD05    ;CIA2: Timer A: High-Byte
LDA #$7F
STA $DD0D    ;CIA2: CIA Interrupt Register
LDA #$81
STA $DD0D    ;CIA2: CIA Interrupt Register
LDA #$19
STA $DD0E    ;CIA2: CIA Control Register A
CLI
JMP $0835    ;Jump to address $0835
```

Listing 3.6: Assembly Language with some comments

As you may remember we said in the previous chapter that this little routine is doing two things: the first is telling the C64 to jump to and execute the routine that starts the actual game the next time it 'wakes up'; the other thing it's doing is turning off the cassette tape reader so that no more data is read from the tape.

The bit that's turning off the tape reader is this:

```
LDA #$10
STA $DD04     ;CIA2: Timer A: Low-Byte
LDA #$00
STA $DD05     ;CIA2: Timer A: High-Byte
LDA #$7F
STA $DD0D     ;CIA2: CIA Interrupt Control Register
LDA #$81
STA $DD0D     ;CIA2: CIA Interrupt Control Register
LDA #$19
STA $DD0E     ;CIA2: CIA Control Register A
```

We are better off treating this series of instructions as a magic formula. The operation of the tape reader is managed by the values stored in a series of bytes between `$DD04` and `$DD0F`. The fact that we have to write a variety of values to 5 of them to just stop reading from the tape is just a testament to the power of boring overhead - we'll never interact with the tape reader again so studying the entrails here is not going to benefit us in any way.

The other thing this routine is doing - preparing the C64 to execute the game proper - is more compact and introduces two concepts that are going to be important throughout our tour of the Iridis Alpha code so it is worth spending some time on them here to get them clear.

```
LDA #$40
STA $0319     ;Non-Maskable Interrupt
LDA #$00
STA $0318     ;Non-Maskable Interrupt
```

Listing 3.7: Containing two important concepts.

Important Concept Number One: High Bytes and Low Bytes

On the face of it these four instructions are doing something very simple. They are storing the value $40 at address $0319 and the value $00 at $0318. What they are actually doing is storing the address $4000 in a place where the C64's 6502 CPU will be expected to look in a moment's time and take that as a command to start executing

whatever is at address $4000.

43	36	34	00	40	41	50	45
0315	0316	0317	0318	0319	$031A$	$031B$	$031C$

Figure 3.2: The slice of C64 memory at addresses $0315 and $031C and the bytes that live there after we've written `$40` to `$0319` and $00 to $0318.

The reversed order we spoke about earlier, the 'little-endian' order, in which the 6502 CPU stores and reads pairs of bytes is observed again here. When the CPU reads the contents of `$0318` and `$0319` it interprets them not as `$0040`, which is the order in which they appear to us, but as `$4000`.

When discussing this storage arrangement we refer to the contents of `$0318` as the 'Low Byte' and the contents of `$0319` as the 'High Byte'. 'High' is just another way of saying 'first', and 'Low' another way of saying 'second. When taking about a values like `$4000` stored in `$0318`–`$0319` such as in this case `$40` is the 'High Byte' and `$00` is the 'Low Byte'.

Writing values to a pair of adjacent addresses in memory like this so that they can be subsequently interpreted as yet another address to get something from or do something with is a very common pattern in programming 6502 CPUs such as the C64's and we will encounter it a *lot* in this book.

It's a strange sort of indirection when you first attempt to understand it. Instead of storing actual values at an address we're storing an address in the address. If you are familiar with other programming languages you may already recognize this concept and understand how powerful it can be. If you are not it will probably seem strange and maybe even wasteful. Seeing the many uses it is put to in the Iridis Alpha code may persuade you otherwise, but hopefully when we look at the use it is put to here you may begin to get a flavor of its utility. Let's do that by looking at our second important concept.

Important Concept Number Two: Interrupts

```
LDA #$40
STA $0319    ;Non-Maskable Interrupt
LDA #$00
STA $0318    ;Non-Maskable Interrupt
```

Listing 3.8: It's an interrupt. And it's non-maskable.

We previously waved away what's happening in this code by saying that the address `$4000` will be interpreted as an address to jump to and start executing the next time the 6502 CPU 'wakes up'. That's a lot of hand-waving.

The technical term for this 'waking up' is an 'interrupt'. This waking up happens incredibly frequently, 60 times every second. 60 times a second the C64 will stop what it's doing and execute whatever is given as an address by the bytes at `$0318-$0319`.

The number 60 may ring a bell for you in this context. A common aspiration for graphics-based games, and minimum table stakes today, is that a game runs at 60 'frames per second'. In other words, that at least 60 times a second the display is updated and whatever is on the screen moves a little bit.

While a C64 programmer would never achieve 60 frames per second in practice, 'interrupts' are the C64's mechanism for at least getting some of the way there. They allow the game developer to at least do something to the display many times per second. Whatever it is, it has to be something short and sweet and at the same time effective enough to actually progress the gameplay. This is why this concept is important to us: most of the important things that happen in Iridis Alpha will be effected during an interrupt. When we look at moving, shooting, and blowing things up, all of them are going to happen in routines that are called multiple times per second by the 6502 CPU executing the code that has been stored at the address `$0318-$0319` at that point in time.

And this is our first taste of the power of storing an address in an address. Depending on where we are in the game - be it the title screen, the main game, the bonus phase, or in a pause mode sub-game, the code that we want to run during these interrupts will be different.

We can see this in action if we look at what happens when the code at `$4000` is executed. We've given this address `$4000` a 'label' name of `MainControlLoop` in our disassembled code so `MainControlLoop` always refers to whatever lives at `$4000`.

```
*=$2000
.include "graphics/charset.asm"
.include "graphics/sprites.asm"

difficultySelected    =*+$01
;
    ----------------------------------------------------------------
```

```
; MainControlLoop
```

Listing 3.9: The code at $4000.

As you can see the first thing it does is change the address of the code that should be executed at every 'interrupt'. It's now an address referred to by the label MainControlLoopInterruptHandler. This happens to be address $6B3E but the beauty of using labels in our code is that we no longer need to worry about the number of the addresses anymore. The label will do the job for us. It does mean we have to know what the syntax #<MainControlLoopInterruptHandler means though. What it means is: if MainControlLoopInterruptHandler lives at $6B3E the #< decorators refer to the $3E part of the address, so we're actually saying LDA #$3E, i.e. load the value $3E into the 'Accumulator'. Similarly the syntax #>MainControlLoopInterruptHandler refers to the $6B part of the address.

43	36	34	$3E$	$6B$	41	50	45
0315	0316	0317	0318	0319	$031A$	$031B$	$031C$

Figure 3.3: The values in $0318 and $0319 after we've updated them in MainControlLoop.

We've almost squeezed as much as we can out of this short and boring snippet of code. There's just one last thing to point out that will be useful to us later. The last instruction in the routine:

```
JMP $0835
```

Listing 3.10: Jump.

This tells the CPU to jump to the address $0835 and execute whatever is there. What is at $0835? This:

```
JMP $0835
```

Listing 3.11: Hello again.

That's right - it's jumping back to itself and will execute in an infinite loop repeatedly executing the instruction over and over again. The reason it does this is because it won't be doing it very long. The interrupt will take over and steal execution away so that the C64 can find better things to do that run in circles.

Before we move on let's take one last look at our disassembly accomplishment.

```
StartExecution
        SEI
        ; Tell the C64 to execute the code at MainControlLoop
        ; the next time an interrupt happens.
        LDA #>MainControlLoop
        STA $0319    ;NMI
        LDA #<MainControlLoop
        STA $0318    ;NMI

        ; Turn off the tape deck.
        LDA #$10
        STA $DD04    ;CIA2: Timer A: Low-Byte
        LDA #$00
        STA $DD05    ;CIA2: Timer A: High-Byte
        LDA #$7F
        STA $DD0D    ;CIA2: CIA Interrupt Control Register
        LDA #$81
        STA $DD0D    ;CIA2: CIA Interrupt Control Register
        LDA #$19
        STA $DD0E    ;CIA2: CIA Control Register A
        CLI
LoopUntilExexcutes
        JMP LoopUntilExexcutes
```

Listing 3.12: Fully disassembled

Hopefully with these basics under our belt we can begin to understand how the interesting parts of Iridis Alpha work.

The First 16 Milliseconds

The race is now on to get the title sequence up on the screen. After a little more setup in `MainControlLoop` we call a routine to set up the main title screen:

```
        ; Display the title screen. We'll stay in here until the
        ; player presses fire or we time out and go into attract
    mode.
        JSR EnterMainTitleScreen
```

Listing 4.1: In `MainControlLoop`

This brings us to a routine called `InitializeSpritesAndInterruptsForTitleScreen` in which we do some vital setup for the next 16 milliseconds and how we go about getting everything on the screen that we need to:

```
        ; Set up the our interrupt handler for the title
        ; screen. This will do all the animation and title
        ; music work.
        LDA #<TitleScreenInterruptHandler
        STA $0314     ;IRQ
        LDA #>TitleScreenInterruptHandler
        STA $0315     ;IRQ

        ; Acknowledge the interrupt, so the CPU knows that
        ; we have handled it.
        LDA #$01
        STA $D019     ;VIC Interrupt Request Register (IRR)
        STA $D01A     ;VIC Interrupt Mask Register (IMR)

        ; Set up the raster interrupt to happen when the
        ; raster reaches the position we specify in D012.
```

```
        LDA $D011    ;VIC Control Register 1
        AND #$7F
        STA $D011    ;VIC Control Register 1

        ; Set the position for triggering our interrupt.
        LDA #$10
        STA $D012    ;Raster Position
```

Listing 4.2: In `InitializeSpritesAndInterruptsForTitleScreen`

You'll notice we've changed our interrupt handler again, this time to a routine called `TitleScreenInterruptHandler`. What we also do is make this interrupt something called a 'Raster Interrupt'. A 'raster' can be thought of as a beam of light that scans across the screen from top to bottom and left to right painting each pixel on the screen one at a time. It travels so quickly down and across the screen painting pixels that it can do so up to 60 times a second. As it makes this journey our 'Raster Interrupt' gives us the opportunity to tell it to stop once it reaches a certain position on the screen and allow us to run some code before it resumes again. We can do this as many times as we want along the journey, but each time we interrupt it we have to be quick. If our code takes too long the display will flicker and the content of the screen become inconsistent.

In this routine we set our first interrupt to line 16 (`$10` on the screen:

```
        ; Set the position for triggering our interrupt.
        LDA #$10
        STA $D012    ;Raster Position
```

Listing 4.3: In `InitializeSpritesAndInterruptsForTitleScreen`

This facility is the key that will allow us to do all sorts of magic in the 16 milliseconds it takes to traverse the screen. Every time we get the opportunity to run some code thanks to this interrupt we'll change the location of the screen it should stop at the next time so that we get to stop dozens of times in each single 16 millisecond traversal.

Before we look at how we fit it all in, let's first appreciate just how much we plan to do each time the screen is painted.

4.0.1 Sprites

The C64 makes 8 sprites available to us. A sprite is a special purpose graphical object that can be up to 24 pixels wide by 20 pixels high. We can place them wherever we want on the screen. They are the core of graphics programming and Iridis Alpha has dozens of them. But if the C64 only has 8 sprites, are we limited to displaying just 8

sprites at once on the screen? The simple answer is that thanks to Raster Interrupts we are not: when we run some code after receiving an interrupt we can place new sprites wherever we like in any position that the raster hasn't reached yet. This means our only effective limitation is the number of sprites we can place on a single line, which is eight.

If you look carefully at the title screen of Iridis Alpha you'll notice that it is actually split in two. The top half has the title in large letters and the bottom half has a rainbow of jumping gilbies. Each half uses seven sprites to display these assets.

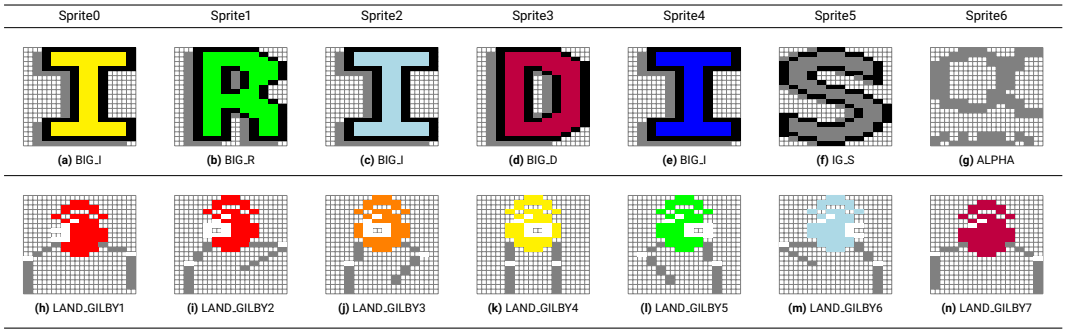
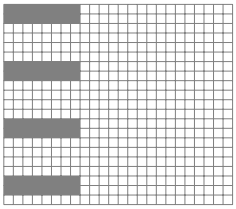


Figure 4.1: The sprites used by the top half of the screen and the bottom half of the screen.

The eighth sprite (Sprite 7) is used on both halves of the screen to display the starfield. This sprite pushes right up against the line limitation. It's painted at intervals throughout the screen but we're careful to avoid it ever being painted twice on the same line. We'll see how this is achieved very soon.



(a) STARFIELD_SPRITE

Figure 4.2: The sprite used for painting the starfield. Only a part of the sprite is ever painted!

4.0.2 Waiting for the Beam

With our 'Raster Interrupt' handler set up as `TitleScreenInterruptHandler` we're ready to react when the raster reaches line 16 on the screen. Since the screen is made up of 512 lines in total this will be along soon.

Before it comes in we just have time to prepare the relatively light amount of text we want displayed on the screen in memory. We only need to do this once. Throughout the code we refer to this area we write to as `SCREEN_RAM`. It's an address range between `$0400` and `$07E8` This is a very simple bitmap representation of the entire screen that is 40 characters wide and 25 characters wide, giving a total of 1000 bytes ($3E8 bytes in hex). If we wanted to think of it as pixels it is 320 pixels wide (40 * 8) and 200 pixels high (25 * 8). The important thing to remember about this `SCREEN_RAM` is that it is solely for storing what we call character data. You can think of character data as 'text'. This text gets painted first and then sprites get painted on top of it.

In `EnterTitleScreenLoop` we call two routines that will prepare the character data for the raster to paint.

The first, `DrawStripesBehindTitle` writes the rainbow stripes to five lines in the top half of the screen. The second, `DrawTitleScreenText` writes some text to the bottom half of the screen. Before we look at these in detail we need to understand how this thing `SCREEN_RAM` works and how we store characters for display in it.

Our starting point for displaying text on screen is to define what our characters look like. We define the appearance of a character using 8 bytes. This is what the definition of the stripe character looks like:

```
        .BYTE $FF,$00,$FF,$00,$00,$FF,$00,$FF   ;.BYTE $FF,$00,$FF,$00,$00,$FF,$00,$FF
                                                ; CHARACTER $00
                                                ; 11111111   ********
                                                ; 00000000
                                                ; 11111111   ********
                                                ; 00000000
                                                ; 00000000
                                                ; 11111111   ********
                                                ; 00000000
                                                ; 11111111   ********
```

Listing 4.4: The 'stripe' character.

As you can see each byte translates to a row of 0s and 1s. Each 1 defines a dot and each 0 a blank space. We end up with a character that is 8 pixels wide and 8 pixels high:

1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

The stripe chracter

We create this definition for every character we want to display and store it at the address starting at `$2000` in RAM. The order in which we store them determines the reference we use for them later. So for example the stripe character is referred to as `$00`, the 'A' character we've defined as `$01` and so on:

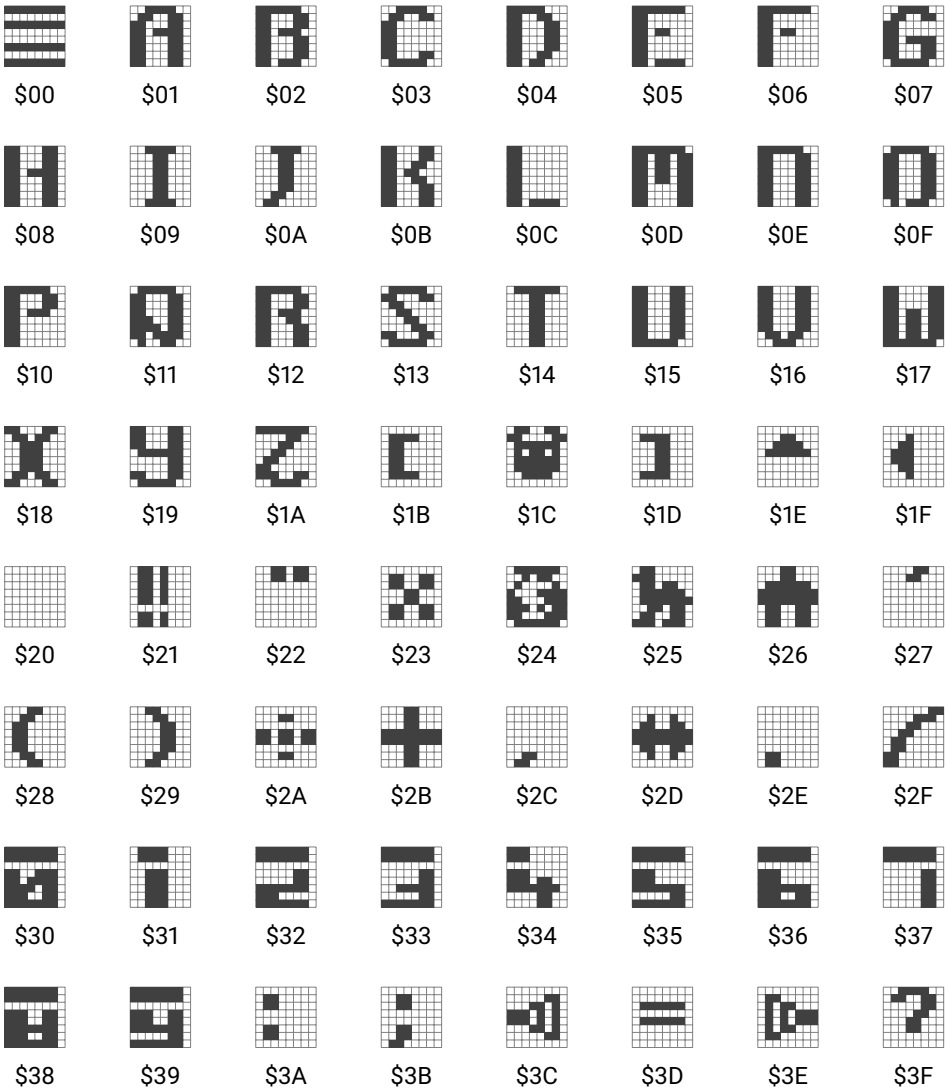


Figure 4.3: Tilesheet: Font Character Set stored at $2000

With our character set defined we can now write some text to the screen ram. Note that when we write it `SCREEN_RAM` we're not yet writing it to the actual screen. This is just a place in memory that the raster (our beam of light) will refer to later when it is actually writing dots to the screen. If we write a stripe character to a particulas position in this

SCREEN_RAM memory it will know to write it the corresponding position on the screen.

Drawing the Stripes

So let's write some stripes to RAM!

```
        STA titleScreenStarfieldMSBXPosArray,X
b0A6D   DEX
        BNE b0A50
        RTS

;
    ------------------------------------------------------------------

; DrawStripesBehindTitle
;
    ------------------------------------------------------------------

DrawStripesBehindTitle
        LDX #$28
        LDA #$00
        STA shouldUpdateTitleScreenColors
DrawStripesLoop
        LDA #RED
        STA COLOR_RAM + LINE2_COL39,X
        LDA #ORANGE
        STA COLOR_RAM + LINE3_COL39,X
        LDA #YELLOW
        STA COLOR_RAM + LINE4_COL39,X
        LDA #GREEN
        STA COLOR_RAM + LINE5_COL39,X
        LDA #LTBLUE
        STA COLOR_RAM + LINE6_COL39,X
        LDA #PURPLE
        STA COLOR_RAM + LINE7_COL39,X
        LDA #BLUE
        STA COLOR_RAM + LINE8_COL39,X
        LDA #$00 ; Stripe character
        STA SCREEN_RAM + LINE2_COL39,X
```

Listing 4.5: The 'stripe' character.

As you can hopefully see, what we're dealing with here is a loop. We load X with the value $28 (40 in decimal) and perform everything inside DrawStripesLoop until DEX has reduced the value of X to zero.

The magic number 40 gives us a clue that what we are doing in each loop is drawing a character in each column of the screen: remember that our screen is 40 columns wide and 25 rows high. The bit actually writing the stripe character to RAM is:

```
        LDA #$00 ; Stripe character
        STA SCREEN_RAM + LINE2_COL39,X
        STA SCREEN_RAM + LINE3_COL39,X
        STA SCREEN_RAM + LINE4_COL39,X
        STA SCREEN_RAM + LINE5_COL39,X
        STA SCREEN_RAM + LINE6_COL39,X
        STA SCREEN_RAM + LINE7_COL39,X
        STA SCREEN_RAM + LINE8_COL39,X
```

Listing 4.6: In `DrawStripesBehindTitle`

For the current column, this writes the stripe character (reference by `$00` as we mentioned above) to each of lines 2 to 8. The use of the `X` in the `STA` statement is an offset. So where `X` is 14, for example, it will write to the position referred to by `SCREEN_RAM + LINE2_COL39` plus 14.

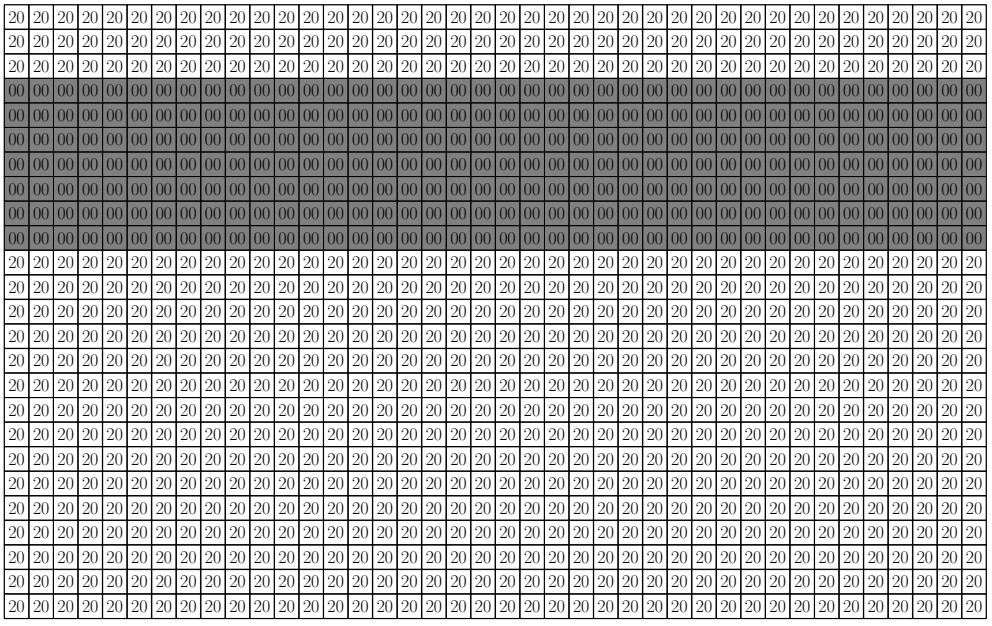


Figure 4.4: The shaded areas of `SCREEN_RAM` after they have been written to by `DrawStripesBehindTitle`.

The other thing we do in DrawStripesLoop is set the colors of the stripes. This is achieved using a region of memory similar in concept to SCREEN_RAM, that we call COLOR_RAM. This lives at $D800 - $DBFE. Another region of 1000 bytes, each one controlling the color of the character placed at a position in the 40 * 25 character rectangle of our screen.

```
        LDA #RED
        STA COLOR_RAM + LINE2_COL39,X
        LDA #ORANGE
        STA COLOR_RAM + LINE3_COL39,X
        LDA #YELLOW
        STA COLOR_RAM + LINE4_COL39,X
        LDA #GREEN
        STA COLOR_RAM + LINE5_COL39,X
        LDA #LTBLUE
        STA COLOR_RAM + LINE6_COL39,X
        LDA #PURPLE
        STA COLOR_RAM + LINE7_COL39,X
        LDA #BLUE
        STA COLOR_RAM + LINE8_COL39,X
```

Listing 4.7: In DrawStripesBehindTitle

We've used a meaningful alias for each of the color values that we write, these are defined as:

```
RED           = $02
PURPLE        = $04
GREEN         = $05
BLUE          = $06
YELLOW        = $07
ORANGE        = $08
BROWN         = $09
LTBLUE        = $0E
```

Listing 4.8: In DrawStripesBehindTitle

So by writing a value to the corresponding place in COLOR_RAM, we're defining the color of the character in that position.

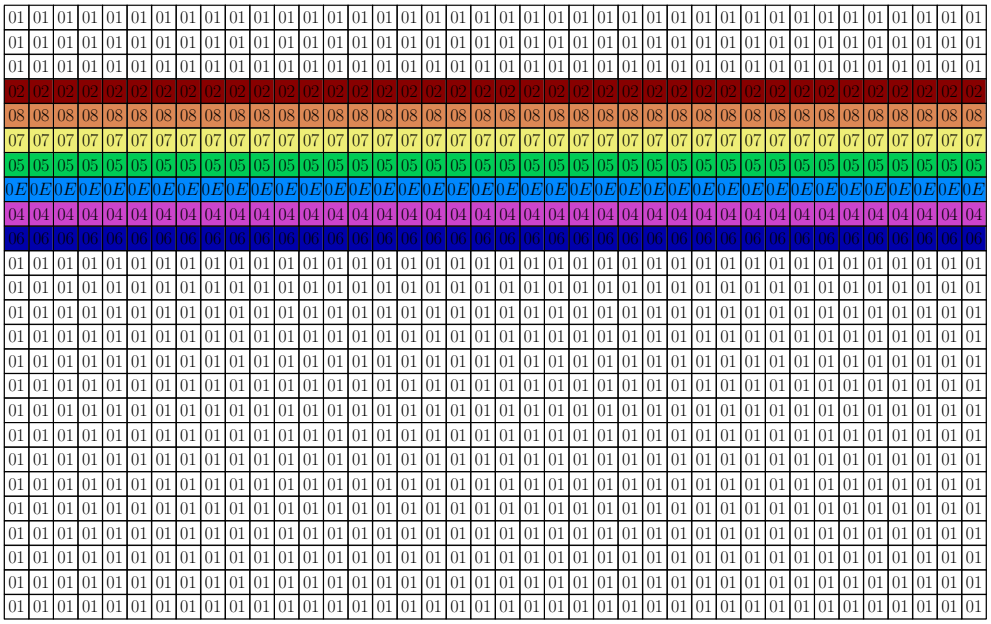


Figure 4.5: The shaded areas of `COLOR_RAM` after they have been written to by `DrawStripesBehindTitle`.

Drawing the Text

Next up is to write out the title screen's text to `SCREEN_RAM`. This we do in `DrawTitleScreenText` using a similar loop to `DrawStripesBehindTitle`.

```
DrawTitleTextLoop
        LDA titleScreenTextLine1 - $01,X
        AND #ASCII_BITMASK
        STA SCREEN_RAM + LINE11_COL39,X
        LDA titleScreenTextLine2 - $01,X
        AND #ASCII_BITMASK
        STA SCREEN_RAM + LINE13_COL39,X
        LDA titleScreenTextLine3 - $01,X
        AND #ASCII_BITMASK
        STA SCREEN_RAM + LINE15_COL39,X
        LDA titleScreenTextLine4 - $01,X
        AND #ASCII_BITMASK
        STA SCREEN_RAM + LINE17_COL39,X
        LDA titleScreenTextLine5 - $01,X
        AND #ASCII_BITMASK
```

```
        STA SCREEN_RAM + LINE19_COL39,X

        LDA #GRAY2
        STA COLOR_RAM + LINE11_COL39,X
        STA COLOR_RAM + LINE13_COL39,X
        STA COLOR_RAM + LINE15_COL39,X
        STA COLOR_RAM + LINE17_COL39,X
        STA COLOR_RAM + LINE19_COL39,X
        DEX
        BNE DrawTitleTextLoop
```

Listing 4.9: In `DrawTitleScreenText`

In this case we're not writing a single character over and over, rather we're writing text we've defined elsewhere in variables `titleScreenTextLine[1-5]`:

```
titleScreenTextLine1        .TEXT "IRIDIS ALPHA.....  HARD AND FAST ZAPPING"
titleScreenTextLine2        .TEXT "PRESS FIRE TO BEGIN PLAY.. ONCE STARTED,"
titleScreenTextLine3        .TEXT "F1 FOR PAUSE MODE     Q TO QUIT THE GAME"
titleScreenTextLine4        .TEXT "CREATED BY JEFF MINTER...SPACE EASY/HARD"
titleScreenTextLine5        .TEXT "LAST GILBY HIT 0000000; MODE IS NOW EASY"
```

Listing 4.10: In `DrawTitleScreenText`

In each iteration of the loop we write a character to all five columns, plucking it from the position in `titleScreenTextLine[1-5]` given by `X`.

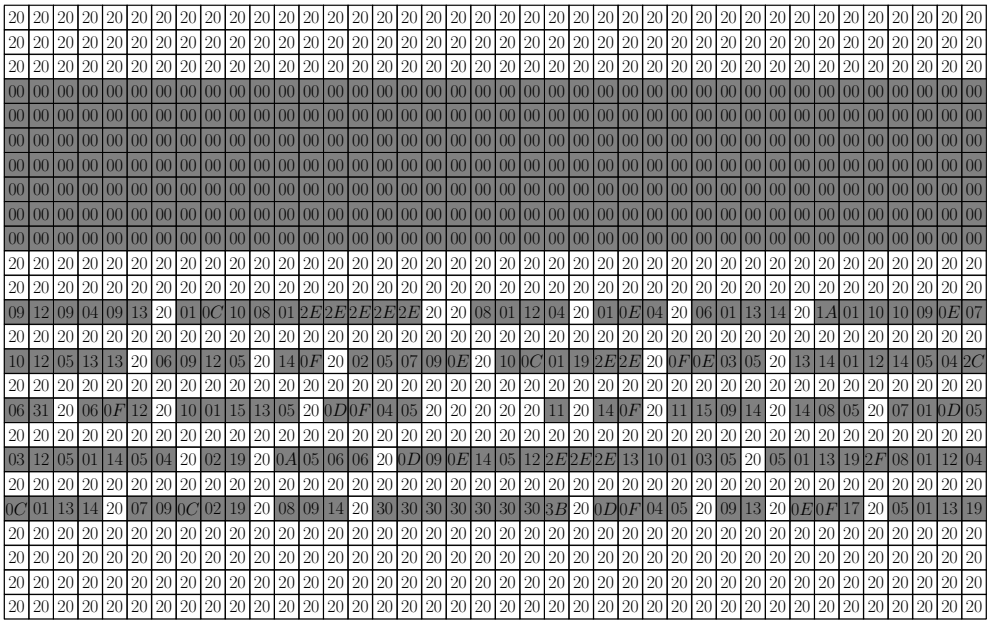


Figure 4.6: The shaded areas of `SCREEN_RAM` after they have been written to by `DrawStripesBehindTitle` and `DrawTitleScreenText`.

While writing text for the column we also set the color for each of the text lines to grey:

```
        LDA #GRAY2
        STA COLOR_RAM + LINE11_COL39,X
        STA COLOR_RAM + LINE13_COL39,X
        STA COLOR_RAM + LINE15_COL39,X
        STA COLOR_RAM + LINE17_COL39,X
        STA COLOR_RAM + LINE19_COL39,X
```

Listing 4.11: In `DrawTitleScreenText`

Once it is done the `COLOR_RAM` has the appropriate lines set to grey:

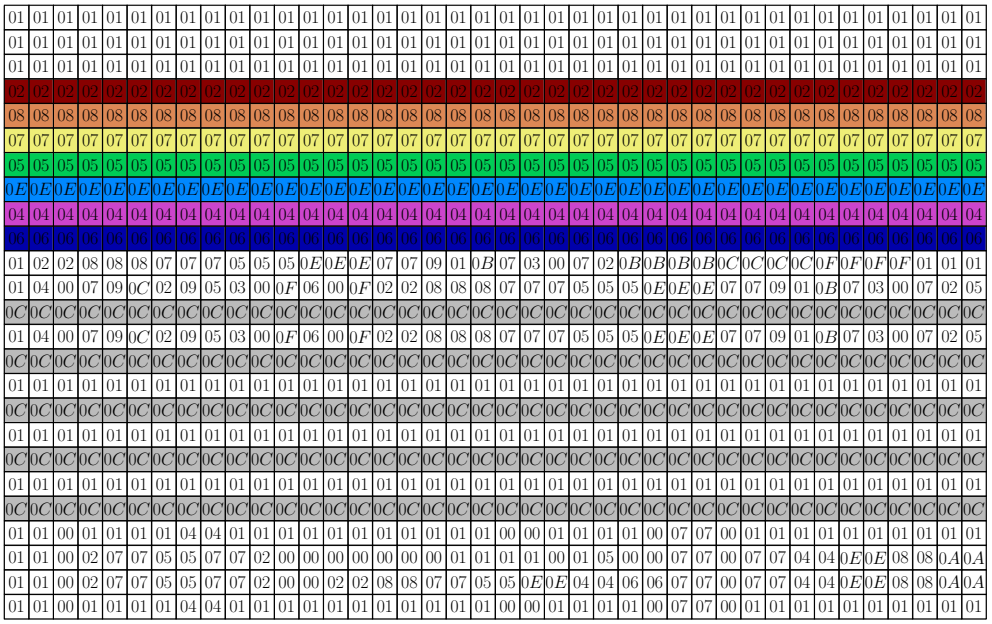


Figure 4.7: The shaded areas of `SCREEN_RAM` after they have been written to by `DrawStripesBehindTitle` and `DrawTitleScreenText`.

Now that we've looped through all 40 columns we have both `SCREEN_RAM` and `COLOR_RAM` fully prepared for painting by the raster. As we watch the screen getting painted in the next section we'll see the following picture we've prepared gradually appear - with the sprites painted on top of course. The magic of adding the sprites to this picture, and animating them while we're at it, is what we will unpick as follow the raster on its journey to the bottom of the screen in the next few milliseconds.

Figure 4.8: The screen as it would appear after `DrawStripesBehindTitle` and `DrawTitleScreenText` have run. The added grid helps compare with our previous figures for `SCREEN_RAM` and `COLOR_RAM`.

4.0.3 Racing the Beam

Now we're ready to receive our ifirst beam. You may remember we set this to happen when the raster reached line 16:

```
        ; Set the position for triggering our interrupt.
        LDA #$10
        STA $D012    ;Raster Position
```

Listing 4.12: In `InitializeSpritesAndInterruptsForTitleScreen`

And that the routine we'll run when that happens is `TitleScreenInterruptHandler` (which itself will pass the work onto `TitleScreenAnimation`:

```
        ; Set up the our interrupt handler for the title
        ; screen. This will do all the animation and title
        ; music work.
        LDA #<TitleScreenInterruptHandler
```

```
        STA $0314     ;IRQ
        LDA #>TitleScreenInterruptHandler
        STA $0315     ;IRQ
```

Listing 4.13: In `InitializeSpritesAndInterruptsForTitleScreen`

The painting of sprites and playing of music as the screen gets painted is all handled by `TitleScreenAnimation`. This routine works by calling one of three different subroutines each time its called. It picks the one to run depending on some internal state it maintains, all with a view to ensuring that the sprites spelling out the game's title and the sprites depicting the animated gilbies are updated and in place before the raster reaches them.

To ensure it gets called by the interrupt when its needed it will repeatedly update the line that the next interrupt should happen. We'll trace this as it actually happens, interrupt by interrupt, and sift through what the routine does at each step during the raster's first pass at painting the entire screen.

The first time the raster is called, this is what the screen looks like:

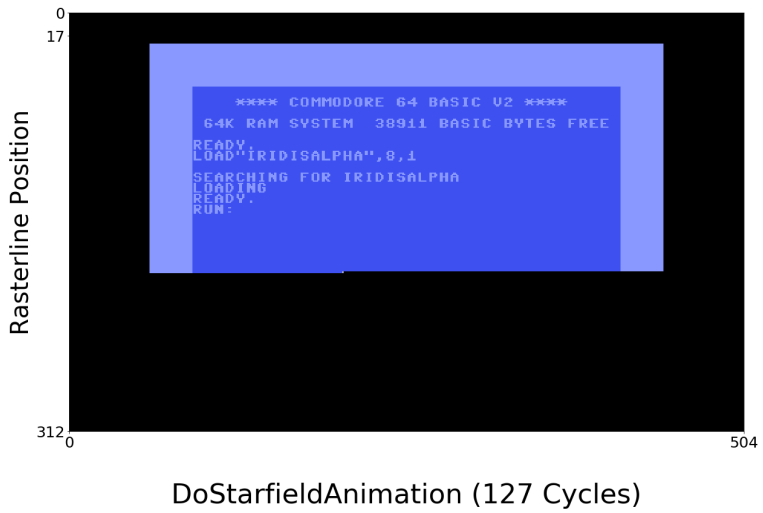


Figure 4.9: The state of the screen the first time the raster interrupt is received at line 16.

Of course we never actually see the screen in this state because it only appears for a microsecond or two, much too fast for us to observe. But as you can see the painting has aleady started. Everything above line 16 indicated in the figure has been painted black, as per our preparation of `SCREEN_RAM` a little earlier.

What our diagram above also tells us is that in this visit from the beam `TitleScreenAnimation` chose to execute the sub-routine `DoStarfieldAnimation` and that it took 127 CPU cycles to complete it. Since it takes the raster 63 cycles to do an entire line this means that by the time we've finished this piece of work, the raster will have moved on to the next line - which is why the diagram shows 17 rather than 16. Every time we get an interrupt, the raster doesn't wait for us. We have to work quickly, especially if we're preparing graphics on lines that its likely to reach soon. This is why each of these subroutines does as little as it can get away with to get the job done.

The three sub-routines the work at each raster interrupt can get divvied out to are `UpdateJumpingGilbyPositionsAndColors`, `DoStarfieldAnimation`, and a cluster of routines starting with `UpdateTitleScreenSpriteColors`. The one that's called the most often is `DoStarfieldAnimation` as it is responsible for sprinkling the screen with animated stars traversing it left to right.

```
;
    ------------------------------------------------------------------

; TitleScreenAnimation
; This handles all the activity in the title screen and is
    called
; roughly 60 times a second by the Raster Interrupt.
;
    ------------------------------------------------------------------

TitleScreenAnimation
        LDY titleScreenStarFieldAnimationCounter
        CPY #$0C
        BNE MaybeDoStarFieldOrTitleText

        JSR UpdateJumpingGilbyPositionsAndColors
        LDY #$10
        STY titleScreenStarFieldAnimationCounter

MaybeDoStarFieldOrTitleText
        LDA titleScreenStarFieldYPosArray,Y
        BNE DoStarfieldAnimation

PaintTitleTextSprites
        JSR TitleScreenMutateStarfieldAnimationData

        LDA #$00
        STA titleScreenStarFieldAnimationCounter

        LDA #$10
        STA $D012    ;Raster Position
```

```
        ; Acknowledge the interrupt, so the CPU knows that
        ; we have handled it.
        LDA #$01
        STA $D019    ;VIC Interrupt Request Register (IRR)
        STA $D01A    ;VIC Interrupt Mask Register (IMR)

        JSR UpdateTitleTextSprites
        JSR MaybeUpdateSpriteColors
        JSR RecalculateJumpingGilbyPositions
        JSR PlayTitleScreenMusic
        JMP ReEnterInterrupt
        ; We're done, returns from function.
```

Listing 4.14: `TitleScreenAnimation` responsible for choosing what to do at each interrupt.

The internal accounting responsible for choosing the routine to run is tricky to decipher by just looking at the code. So instead let's follow what actually happens in practice. If we roll ahead to the next interrupt we can already see something happening:

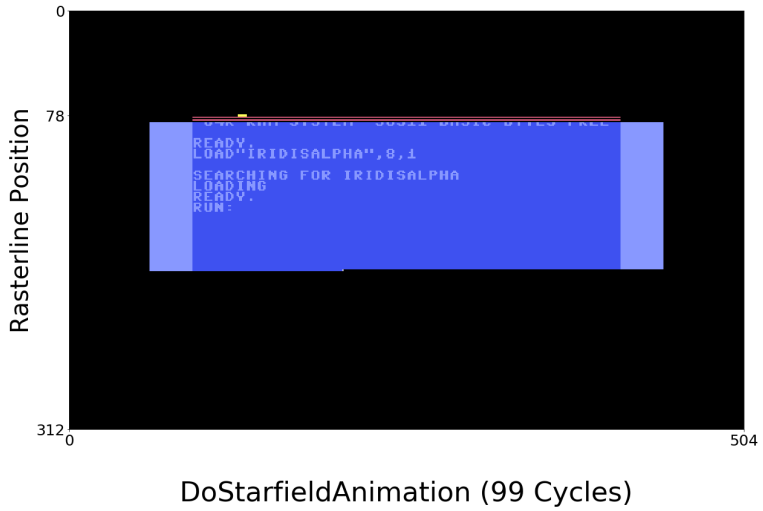


Figure 4.10: The start of the stripes and a star.

If you look closely, you can see a yellow star painted over the first band of red stripes. This is our first sprite. You may be wondering: what about the title sprites? Shouldn't they be there by now? The answer is no: we will paint them when the raster reaches the end of the screen. When it goes to paint the screen a second time (the second 16

milliseconds) they will be ready for painting. We'll see this in action a little later.

So let's see what `DoStarfieldAnimation` did to get this sprite ready for painting.

When `TitleScreenAnimation` chose `DoStarfieldAnimation` as the sub-routine to run it loaded in a value from `titleScreenStarFieldYPosArray` to the `A` register:

```
MaybeDoStarFieldOrTitleText
        LDA titleScreenStarFieldYPosArray,Y
        BNE DoStarfieldAnimation
```

Since `Y` is zero at this point this means it referenced the first value in the array, which is `$48`:

```
titleScreenStarFieldYPosArray .BYTE $48,$4E,$54,$5A,$60,$66,$6C,$72
                              .BYTE $78,$7E,$84,$8A,$90,$96,$9C,$A2
                              .BYTE $A8,$AE,$B4,$BA,$C0,$C6,$CC,$D2
                              .BYTE $D8,$DE,$E4,$EA,$F0,$F6
titleScreenStarFieldXPosArray .BYTE $00,$3A,$1A,$C4,$1B,$94,$7B,$96
                              .BYTE $5D,$4F,$B5,$18,$C7,$E1,$EB,$4A
                              .BYTE $8F,$DA,$83,$6A,$B0,$FC,$68,$04
                              .BYTE $10,$06,$A7,$B8,$19,$BB
```

So when `DoStarfieldAnimation` is called the first thing it does is set the y position of the star to paint to `$48` (72 in decimal):

```
DoStarfieldAnimation
        ; A was loaded from titleScreenStarFieldYPosArray
        ; by the caller.
        STA $D00F     ;Sprite 7 Y Pos

        ; Set the X position of the star.
        LDA titleScreenStarFieldXPosArray + $01,Y
        STA $D00E     ;Sprite 7 X Pos
```

Listing 4.15: The start of `DoStarfieldAnimation` responsible for painting stars.

You can also see it then sets the X position of the star using values plucked from `titleScreenStarFieldXPosArray`. So we're leaning heavily on these two arrays to decide where to place stars. But so far, so simple. We've placed the star on the screen more or less and when the raster reaches line 72 it will paint it. There's an additional complication to specifying the X coordinate of the star though and we can't really gloss over it here. We'll also encounter this wrinkle elsewhere too so it's worth pausing on for a moment.

The next few lines of the routine do quite a bit of convoluted work to handle something called the `spriteMSBXPosOffset` of the star. This is our complication.

A Complication

```
        ; Set the rest of the X position of the star
        ; if it's greater than 255.
        LDA titleScreenStarfieldMSBXPosArray + $01,Y
        AND #$01
        STA spriteMSBXPosOffset

        BEQ StarFieldSkipMSB

        LDA #$80
        STA spriteMSBXPosOffset
StarFieldSkipMSB
        LDA $D010     ;Sprites 0-7 MSB of X coordinate
        AND #$7F
        ORA spriteMSBXPosOffset
        STA $D010     ;Sprites 0-7 MSB of X coordinate
```

Listing 4.16: MSBXPos.. some'it.

If you look at the diagram again you may recall we said the screen we're painting is 504 pixels wide. Fortunately the only part we can paint is the section in the center that is 320 pixels wide and 200 pixels high.

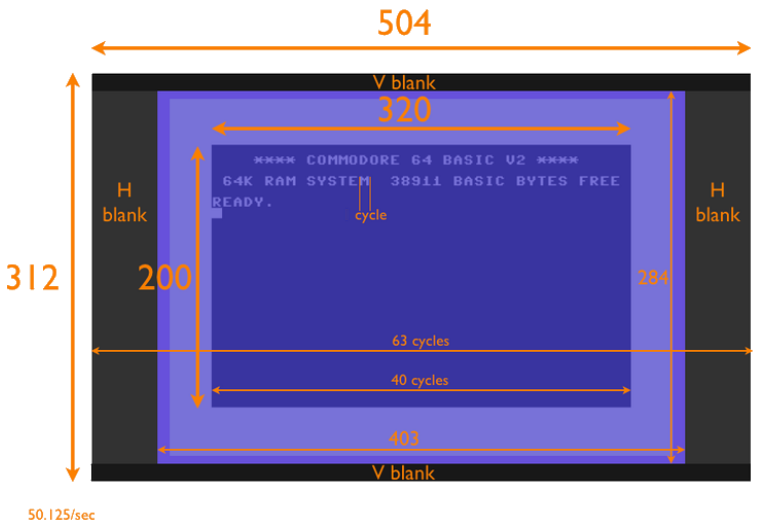


Figure 4.11: The different parts of the screen, we can only paint the bit in the middle. (Source: dustlayer.com)

200 is a value that can be expressed with a single byte. However, 320 is not. A byte can only store a number up to 255 so if we want to specify an X co-ordinate greater than 255 a single byte will not do. The way the C64 works around this is by making a single extra bit for our sprite's X co-ordinate available that brings the available values up from 256 (0 - 255) to 512. Since there are 8 sprites in total we need 8 extra bits to cover this requirement for all of them. For this purpose we use a single byte at address `$D010` that contains the extra bit for all 8 sprites. We refer to this bit as the `MSB` for the X co-ordinate because it is the 'Most Significant Bit', i.e. the left most bit, in the 9-bit number that we store the X co-ordinate in. That's to say our x co-ordinate is given by combining the value between 0 and 255 we store in our 8-bit byte for 'Sprite 7' in `$D00E` and the extra bit we store in `$D010`.

Sprite 7	Sprite 6	Sprite 5	Sprite 4	Sprite 3	Sprite 2	Sprite 1	Sprite 0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	0

The most significant bits in `$D010` for each sprite.

Since we are using 'Sprite 7' for painting the starfield the bit we are interested in is bit 7. The way we're going to manage this value for the starfield is by keeping array `titleScreenStarfieldMSBXPosArray` that indicates whether the x co-ordinate for the current index is greater than 255. If the value in there indicates that it is we'll set our bit in `$D010` to 1.

So when we determine from looking in `titleScreenStarfieldMSBXPosArray` that the x co-ordinate of the star is greater than 256..

```
        LDA titleScreenStarfieldMSBXPosArray + $01,Y
        AND #$01
        STA spriteMSBXPosOffset
```

.. we set `spriteMSBXPosOffset` to indicate that that's the case. That's all this step, with the help of the `AND #$01` statement is doing. If 'Bit 1' is set in the value we pluck from `titleScreenStarfieldMSBXPosArray` it is just an indicator that the x co-ordinate for this star is greater than 256. So if we see a value of `$02` in there our operation `AND #$01` will give us a zero result, meaning the intended value of the x co-ordinate is not greater than 256, othwerise it will give us a non-zero result indicating that it is:

	Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	$02	0	0	0	0	0	0	1	0
	$01	0	0	0	0	0	0	0	1
Result:	$00	0	0	0	0	0	0	0	0

AND'ing $02 and $01 gives $00 (0). For `AND` to give a 1 both bits must 1 or both must be 0.

This zero result allows `BEQ StarFieldSkipMSB` to evaluate as `True` so we skip ahead to `StarFieldSkipMSB` to set `$D010`. It means for us that the value of the x co-ordinate is not going to be greater than 255. If this is not the case, we instead load a value of `$80` to `spriteMSBXPosOffset` to overwrite the `00` there. This will indicate that the value of the x co-ordinate is greater than 255.

```
        BEQ StarFieldSkipMSB

        LDA #$80
        STA spriteMSBXPosOffset
StarFieldSkipMSB
        LDA $D010     ;Sprites 0-7 MSB of X coordinate
        AND #$7F
        ORA spriteMSBXPosOffset
        STA $D010     ;Sprites 0-7 MSB of X coordinate
```

The remaining step above is to load the value we've arrived at in `spriteMSBXPosOffset` to `$D010`. Since we want to do this without affecting any of the other bits in there that have been set for the other sprites we can't just do a `LDA/STA` as that will overwrite what's already there. The combination of the `AND/OR` operations here accomplishes something quite nifty - it allows us to update just the bit (Bit 7) that interests us in `$D010`.

If we suppose the current value in `$D010` is `$F3`, our `AND #$7F` operation clears 'Bit 7' so that it is always set to zero:

	Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	$F3	1	1	1	1	0	0	1	1
	$7F	0	1	1	1	1	1	1	1
Result:	$73	0	1	1	1	0	0	1	1

AND'ing $F3 and $00 gives $73. It clears 'Bit 7' for us of whatever value was there originally.

Now when we perform an 'or' operation on the result with `ORA spriteMSBXPosOffset` it will have the effect of just setting 'Bit 7' with the value we've stored in `spriteMSBXPosOffset`. In this case it remains at zero because that's what we have in `spriteMSBXPosOffset` but if we have `$80` in there it would set it to 1:

	Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	$73	0	1	1	1	0	0	1	1
	$00	0	0	0	0	0	0	0	0
Result:	$73	0	1	1	1	0	0	1	1

OR'ing $73 and $00 gives $73.

Back to the Beam

Now that we've fixed the star's co-ordinates there's just two main things left to do before we're done with handling this raster interrupt. One is to set the color of the star. We do this using a look-up array where we get the color for the star per our current index and set it:

```
        LDA titleScreenStarFieldColorsArrayLookUp,Y
        TAX
        LDA titleScreenColorsArray - $01,X
        STA $D02E    ;Sprite 7 Color
```

The second, and most important, is that we update the position on the screen that we want the next interrupt to happen. Remember for this visit we were interrupted at line 17. We want to place stars on other lines so we keep a list of the lines we want to write stars on in `titleScreenStarFieldYPosArray`, i.e. an array that stores the y co-ordinates of our stars. What we do is simply update the Raster Interrupt with the next position from this array so that we get called when the raster reaches it:

```
        ; Update the raster position for the next interrupt
        ; to the current line - 1. This will allow us to
        ; draw the sprite multiple times on different lines.
        LDA titleScreenStarFieldYPosArray + $01,Y
        SEC
        SBC #$01
        STA $D012    ;Raster Position
```

In this instance we're setting the value to `$48` (72). So when the next interrupt happens

it will reveal the star we just prepared. THis is the one we took a peek at in Figure 1.10.

The next ten interrupts will continue to revisit `DoStarfieldAnimation`. Let's look at the screen as it unfolds through each of these interrupts:

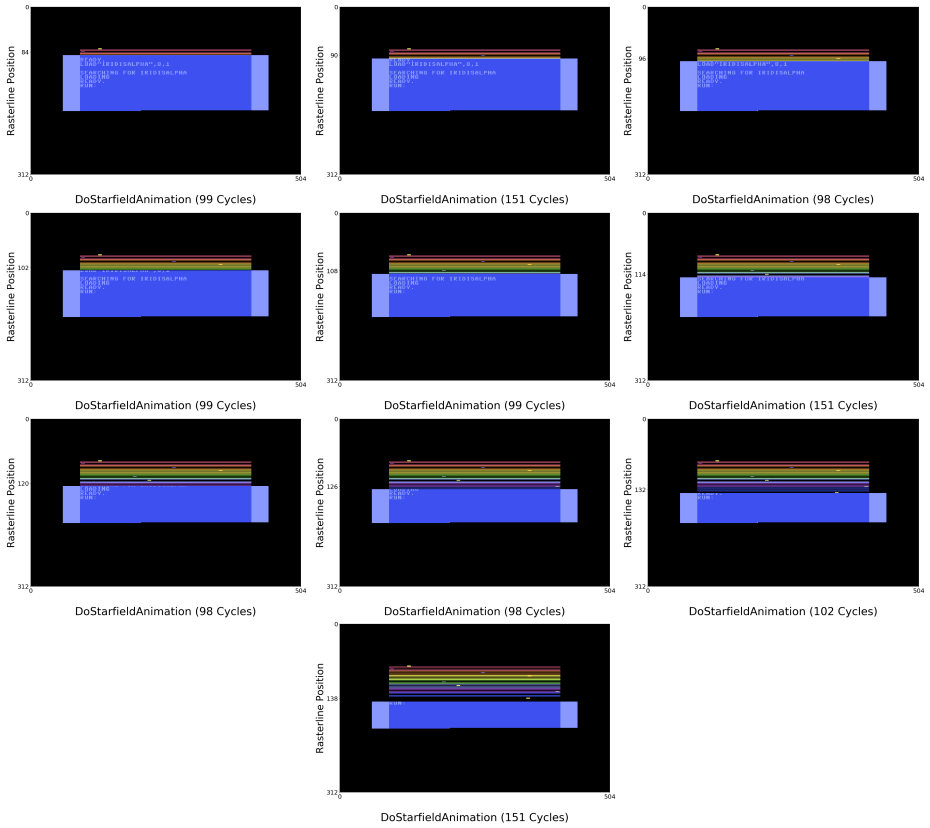


Figure 4.12: The next ten interrupts paint the starfield on the screen until we reach the point at which we want to prepare the gilby sprites.

4.0.4 Enter The Gilbies

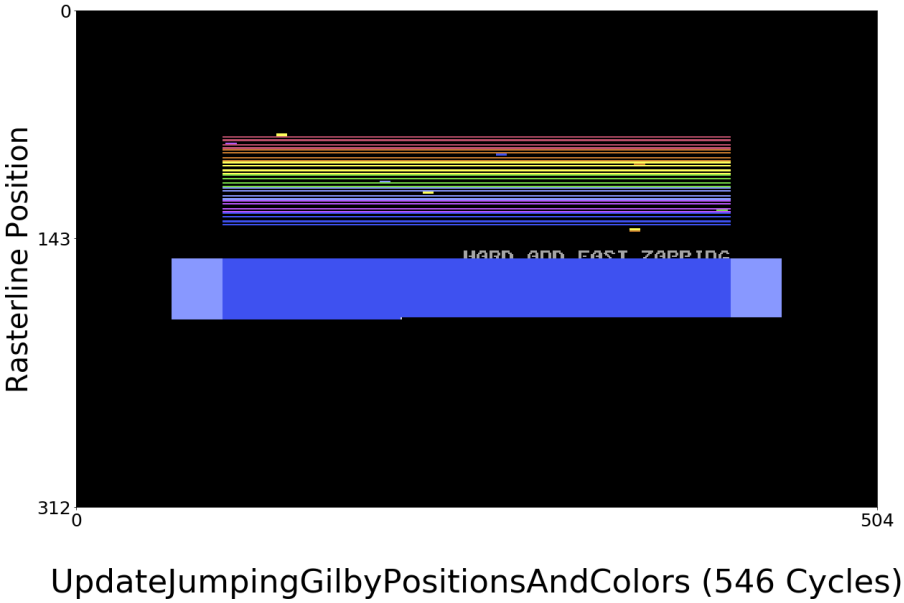


Figure 4.13: The point we reach in the screen paint when we decide to prepare the gilby sprites.

Finally we've reached a point in the screen where we're not just going to add another star to the background. We've been keeping a count of the number of interrupts we've handled in `titleScreenStarFieldAnimationCounter`. When it reaches `$0C` (12) we've handled the raster interrupt twelve times and painted nothing but the text background we prepared earlier and the stars we've added along the way. Now's the time to do something else:

```
TitleScreenAnimation
        LDY titleScreenStarFieldAnimationCounter
        CPY #$0C
        BNE MaybeDoStarFieldOrTitleText

        JSR UpdateJumpingGilbyPositionsAndColors
        LDY #$10
        STY titleScreenStarFieldAnimationCounter
```

The routine we call into here by the name of `UpdateJumpingGilbyPositionsAndColors`

will prepare the sequence of jumping rainbow gilbies somewhere in the lower half of the screen before our current raster line. That's why we call it at this point in the raster's journey - because the raster hasn't reached that position in the screen yet (but will reach it shortly) so now is our opportunity to position the gilbies where we want them.

Since this is an animation sequence we're managing the approach of `UpdateJumpingGilbyPositionsAndCol` is simply to update their position on the screen. Calculating this new position is something that happens a little later in the routine `RecalculateJumpingGilbyPositions` when we are nearer the bottom of the screen. This means the position values we're picking up here are the initial ones set in the game's code:

```
titleScreenGilbiesYPosArray         .BYTE $B2,$B6,$BB,$C1,$D0,$C8,
    $C1
titleScreenGilbiesXPosArray         .BYTE $54,$58,$5C,$60,$64,$68,
    $6C
```

The next time around we will pick up the positions as re-calculated by `RecalculateJumpingGilbyPositions`. So the positioning of the gilbies and the calculation of the updated positions happen separately. The reason for that approach is simple: there isn't enough time right now to do anything but simply update the positions the gilbies are displayed at. Later on, when the raster has passed line 320 we will have a lot more time available to perform complex calculations because we don't need to worry about the raster painting anything on the screen for a while.

Since there are 7 of them, setting the x and y co-ordinates of the seven gilby sprites is handled by a loop:

```
        ; Loop through the gilby sprites in the title screen and
        ; update their position and color
        LDX #$00
UpdateJumpingGilbiesLoop
        TXA
        ASL
        TAY
        LDA titleScreenGilbiesXPosArray,X
        ASL
        STA $D000,Y  ;Sprite 0 X Pos
        BCC SkipGilbyMSBXPos
        LDA $D010    ;Sprites 0-7 MSB of X coordinate
        ORA titleScreenGilbiesMSBXPosArray,X
        STA $D010    ;Sprites 0-7 MSB of X coordinate
        JMP UpdateYPosJumpingGilbies

SkipGilbyMSBXPos
```

```
        LDA $D010     ;Sprites 0-7 MSB of X coordinate
        AND titleScreenGilbiesMSBXPosOffset,X
        STA $D010     ;Sprites 0-7 MSB of X coordinate

UpdateYPosJumpingGilbies
        LDA titleScreenGilbiesYPosARray,X
        STA $D001,Y   ;Sprite 0 Y Pos

        LDA currentTitleScreenGilbySpriteValue
        STA Sprite0Ptr,X

        ; Update Gilby color.
        LDA titleScreenColorsArray,X
        STA $D027,X   ;Sprite 0 Color

        INX
        CPX #$07
        BNE UpdateJumpingGilbiesLoop
```

Listing 4.17: The loop in `UpdateJumpingGilbyPositionsAndColors` updating the x and y position on screen and color of each of the gilby sprites.

We can see in here the verbosity required to handle the most significant bit of the sprite's x co-ordinate. Just as with the starfield we need a separate array (`titleScreenGilbiesMSBXPosArray`) to handle this in addition to arrays to manage the basic x/y positions themselves.

With the gilbies prepared we fall through and update the starfield again. Once that's done we're finished handling the current raster interrupt. This is followed by another dozen or so interrupts where we again just prepare stars for display and as the raster progress our gilbies are revealed.

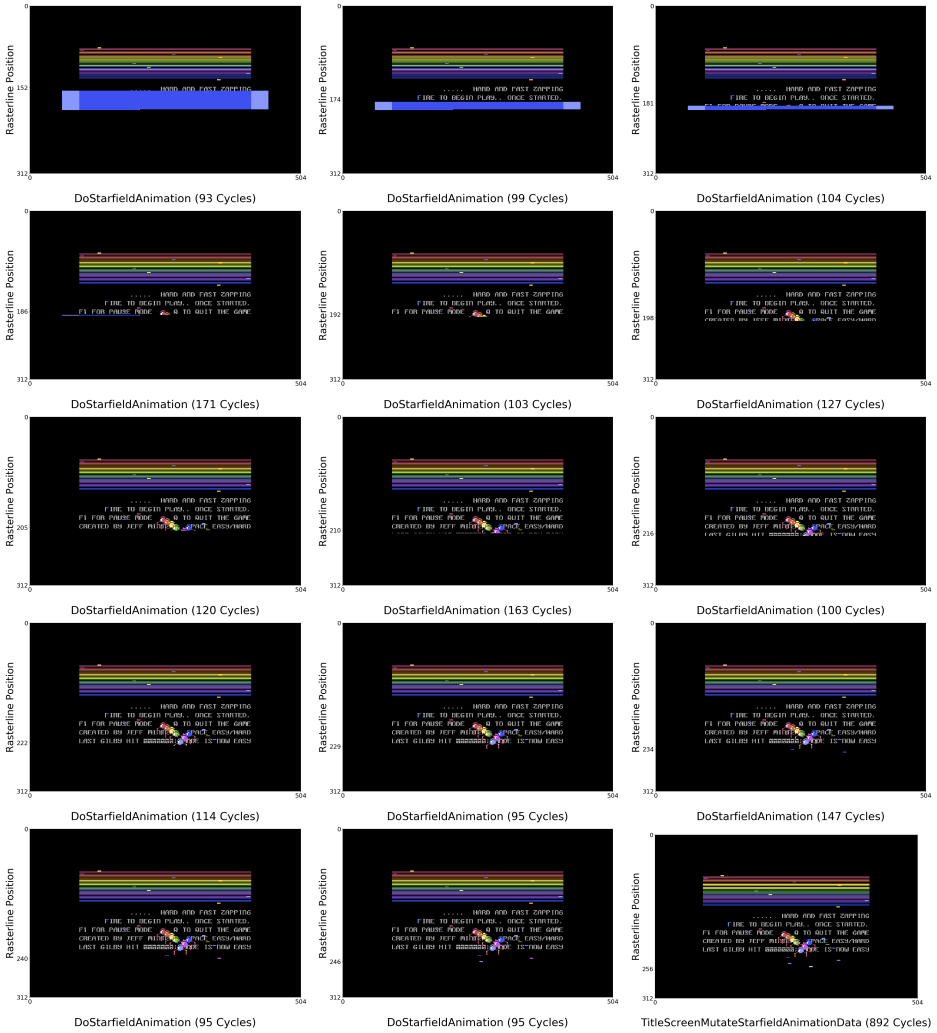


Figure 4.14: Behold the gilbies.

4.0.5 Title Text

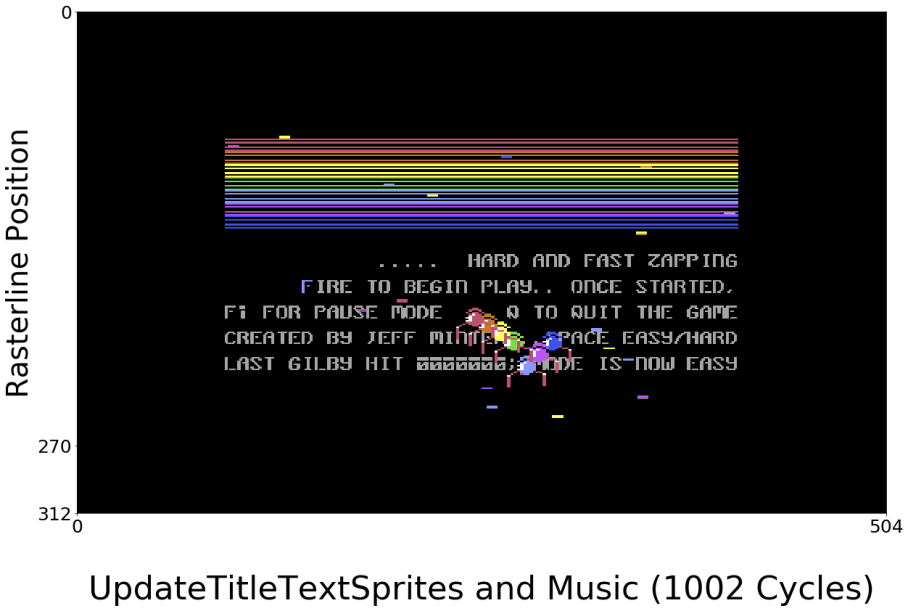


Figure 4.15: We've finally reached the bottom of the screen, with gilbies and stars painted, but still no title.

We've finally reached the bottom of the screen in our first raster paint, at least the bottom of the portion of the screen that we can paint. When the raster hits line 270 we're beyond the point that we can place anything on the screen and into the border area. This gives us time to do some more complicated and time consuming stuff.

There's a relatively full agenda:

```
        LDA #$10
        STA $D012     ;Raster Position

        ; Acknowledge the interrupt, so the CPU knows that
        ; we have handled it.
        LDA #$01
        STA $D019     ;VIC Interrupt Request Register (IRR)
        STA $D01A     ;VIC Interrupt Mask Register (IMR)

        ; All of this stuff can be done before the raster
        ; reaches the top of the screen again.
```

```
        JSR UpdateTitleTextSprites
        JSR MaybeUpdateSpriteColors
        JSR RecalculateJumpingGilbyPositions
        JSR PlayTitleScreenMusic
        JMP ReEnterInterrupt
```

First of all we set the raster interrupt to line 16 at the top of the screen again, then we acknowledge the interrupt. The raster will continue its journey but because we're going to do this while it works its way the next 42 lines at the bottom of the screen we have more time than at any point previously to get things done.

Adding the title sprites is relatively light work. Just as with the gilbies we use a tight loop to paint each of them on the screen. THere's no animation to handle here.

```
titleTextSpriteArray          .BYTE $20,BIG_I,BIG_R,BIG_I,BIG_D,
    BIG_I,BIG_S
...

PaintSpriteLettersLoop
        ; Assign the sprite.
        LDA titleTextSpriteArray,X
        STA Sprite0Ptr - $01,X

        ; Shift the value in X left 1 bit and assign to Y.
        ; So e.g. 6 becomes 12, 5 becomes 10, 4 becomes 8,
        ; 3 becomes 6 and so on. This allows us to use Y
        ; as an offset to the appropriate item in $D000-
        ; $D012 for updating the sprite's position.
        TXA
        ASL
        TAY

        ; Update the X Position of the sprite
        LDA titleTextXPosArray - $01,X
        STA $D000 - $02,Y

        LDA $D010     ;Sprites 0-7 MSB of X coordinate
        ORA titleTextMSBXPosArray,X
        STA $D010     ;Sprites 0-7 MSB of X coordinate

        ; Update the Y position of the sprite
        LDA #$40
        STA $D000 - $01,Y
        DEX
        BNE PaintSpriteLettersLoop
```

The other complex thing we do is calculate the next step in the jumping gilby animations. `RecalculateJumpingGilbyPositions` updates the x and y positions of the gilby sprites in `titleScreenGilbiesYPosArray` and `titleScreenGilbiesXPosArray`.

Finally we play a single note from the title music, we cover this in detail in a later chapter.

The raster continues on its journey and progresses through its second paint journey of the screen. The title sprites are finally revealed.

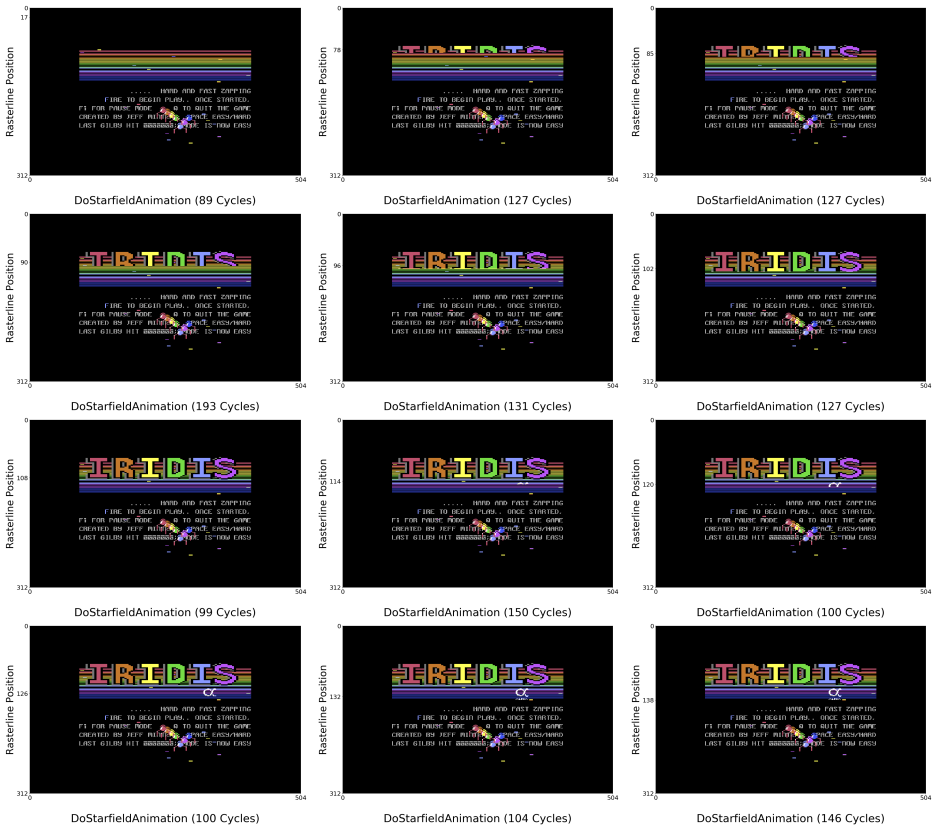


Figure 4.16: The title text is finally revealed

And with that the title sequence is finally up and running after 20 milliseconds or so.

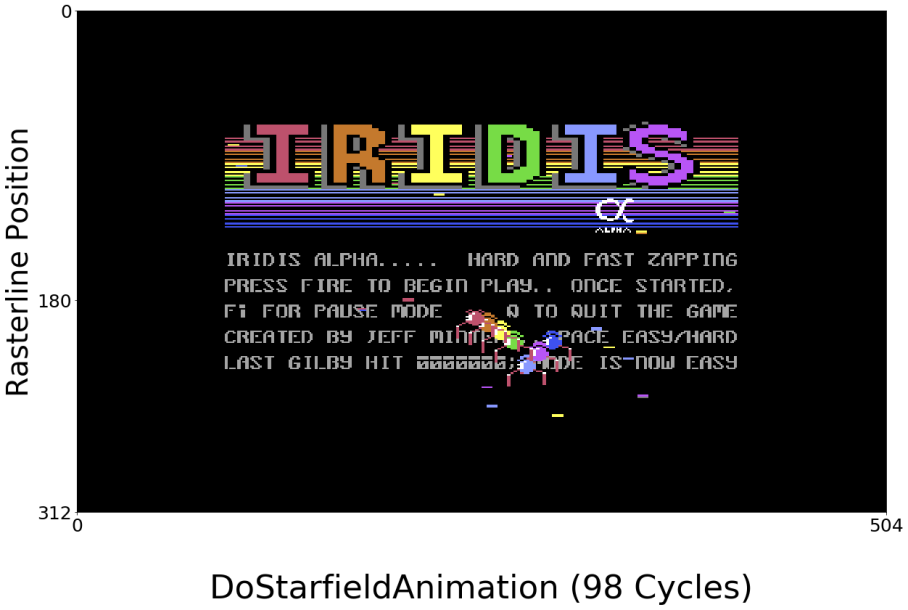


Figure 4.17: We're done here.

Making Planets for Nigel

> 17 February 1986
>
> Redid the graphics completely, came up with some really nice looking metallic planet structures that I'll probably stick with. Started to write the `GenPlan` routine that'll generate random planets at will. Good to have a C64 that can generate planets in its spare time. Wrote pulsation routines for the colours; looks well good with some of the planet structures. The metallic look seems to be 'in' at the moment so this first planet will go down well. There will be five planet surface types in all, I reckon, probably do one with grass and sea a bit like 'Sheep in Space', cos I did like that one. It'll be nice to have completely different planet surfaces in top and bottom of the screen. The neat thing is that all the surfaces have the same basic structures, all I do is fit different graphics around each one.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

Making planets is easy.

When making a planet, ensure you perform each of the following simple steps in the order given below.

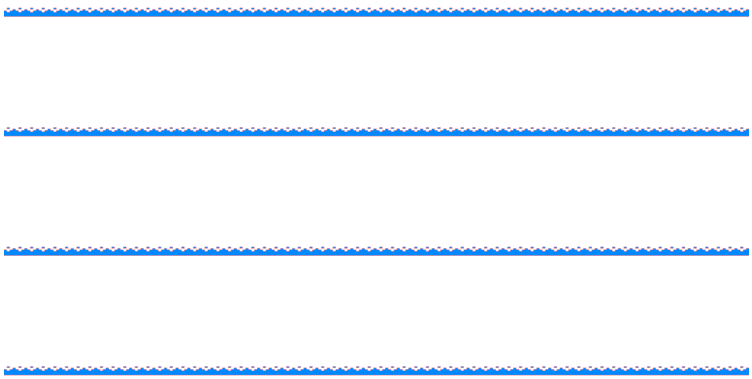


Figure 5.1: Step One: Add the sea across the entire surface of the planet.

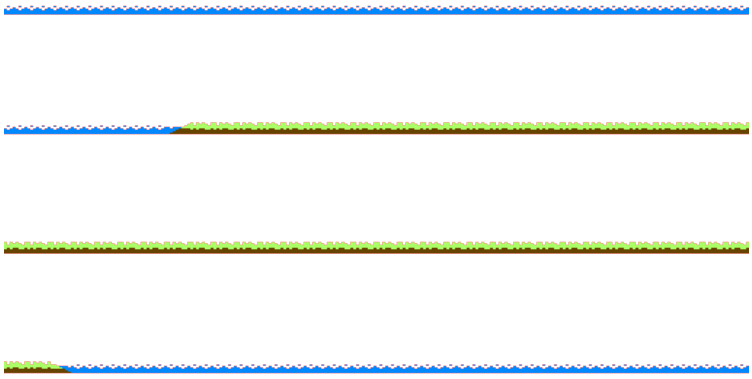


Figure 5.2: Step Two: Insert a land mass at least 32 bytes and at most 128 bytes long.

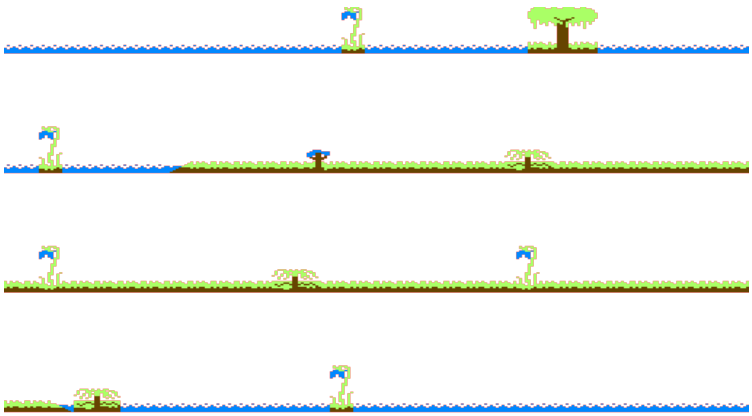


Figure 5.3: Step Three: Add a random structure every 13 to 29 bytes.

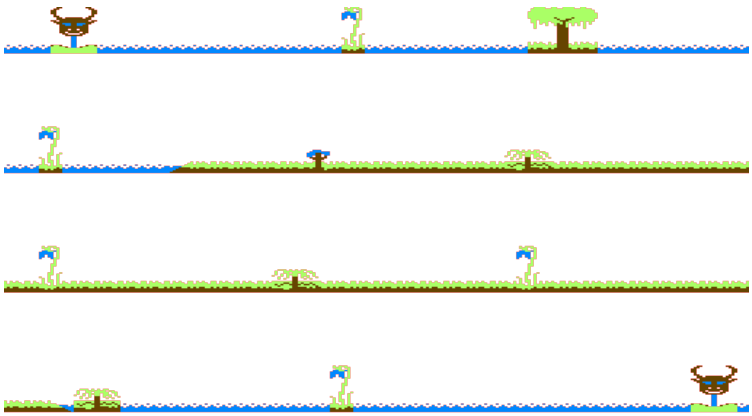


Figure 5.4: Step Four: Add warp gates at the beginning and end of the planet surface.

Now you have not just a layout for one planet, but a layout for all five.

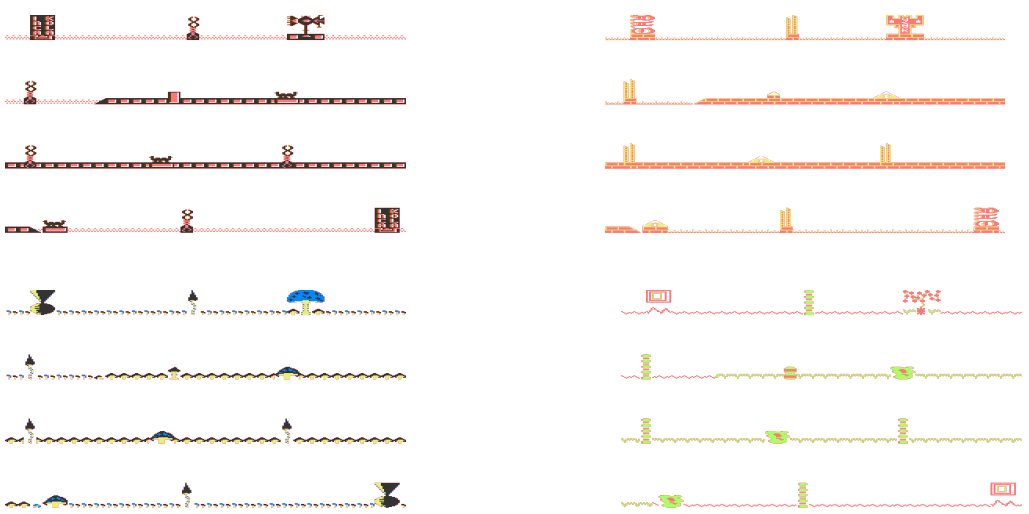
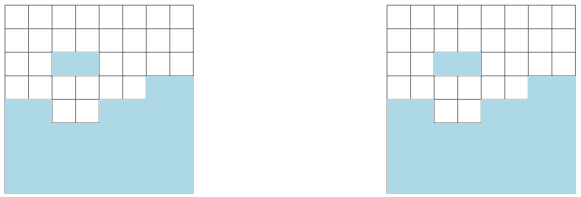


Figure 5.5: A layout that will suit all the planets in your life.

But making planets isn't all simple steps and big picture decisions. There are also trifling details for the little people to wrestle with.

5.0.1 Step One: Creating the Sea

Making a sea is very easy. You come up with a character than can be repeated 1024 times to fill the surface of the planet.



((1)) planet1Charset $40 **((2))** planet1Charset $42

Figure 5.6: There are two characters used for creating the sea and they're both the same! This will make more sense when we look at the land, where they are different.

Figure 5.7: planet1Charset Sea

The bit that needs explaining is how you define the character. If it was a simple bitmap then we could imagine the character as 8 rows of 8 bits and where a bit is set to 1 you color that pixel in. That is not the case. You can see how the bits are actually set below:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

Figure 5.8: planet1Charset $40 representing a tile of sea.

Look closely at the picture above and you should see how it works. What is happening is that we fill two adjacent cells with blue when together they form the value `10`. So we create graphic characters not with a simple bit-map but with a map of bit pairs. Each pair of bits is treated as a unit giving us four units on each row. Maybe it's intuitively obvious that `00` means 'blank' or 'background' but I've pointed that out to you now just in case.

```
planet1Charset
        .BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA   ;.BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA
                                                ; CHARACTER $40
                                                ; 00000000
                                                ; 00000000
                                                ; 00100000      *
                                                ; 00000010          *
                                                ; 10001010    *     * *
                                                ; 10101010    * * * *
                                                ; 10101010    * * * *
                                                ; 10101010    * * * *
```

Listing 5.1: Character `$40` representing the sea as it is defined in the source code. A full eight bytes are required to define each character so not cheap.

Is that all there is to it? No. Before we look at how me might color things other than blue, let's look at how we color them with the big blue brush we have so far. The first thing we do is clear down the entire surface of the planet:

```
```

```
;
    ------------------------------------------------------------------

; ScrollPlanets
;
    ------------------------------------------------------------------

ScrollPlanets
        INC planetScrollFrameRate
        LDA planetScrollFrameRate
        AND #$0F
        TAX
        LDA colorSequenceArray2,X
        STA unusedByte1
        LDA colorSequenceArray,X
        STA unusedByte2
        LDA currentGilbySpeed
        BMI b7234
        JMP ScrollPlanetLeft

b7234   LDA planetScrollSpeed
        CLC
        ADC currentGilbySpeed
        STA planetScrollSpeed
        AND #$F8
        BNE b7243
```

Listing 5.2: The surface data is stored from $8000 to $8FFF. This code overwrites it all with the value $60 which is an empty bitmap.

```
        .BYTE $00,$00,$00,$00,$00,$00,$00,$00   ;.BYTE $00,$00,$00,$00,$00,$00,$00,$00
                                                ; CHARACTER $60
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
                                                ; 00000000
```

Listing 5.3: The empty character bit map (all zeroes) used to overwrite the surface before populating it.

With the planet surface cleared out (overwritten with all $60s) we can now.. overwrite it all again with sequences of $40,$42. No, that's not right. We're only overwriting the bottom layer - the surface layer - this time. This is the layer that contains the land and/or sea and it lives between $8C00 and $8FFF which if your hexadecimal arithmetic is better than mine you will realize is 1024 bytes ($400 in hex).

```
        EOR #$FF
```

```
        CLC
        AND #$F8
        ROR
        ROR
        ROR
        STA tempHiPtr1
        INC tempHiPtr1
        LDA planetTextureTopLayerPtr
        CLC
        ADC tempHiPtr1
        STA planetTextureTopLayerPtr
        STA planetTextureSecondFromTopLayerPtr
        STA planetTextureSecondFromBottomLayerPtr
        STA planetTextureBottomLayerPtr
        LDA planetTextureTopLayerPtrHi
        ADC #$00
;
    ----------------------------------------------------------------

; DrawPlanetScroll
;
    ----------------------------------------------------------------

DrawPlanetScroll
        ; Adjust the layer pointers to the approriate
```

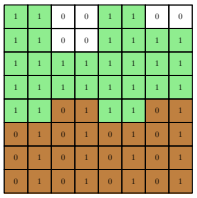
Listing 5.4: Filling the entire bottom surface of the planet with `$40,$42` which gives us the sea. Our next step is to overwrite some of this with land.

Figure 5.9: That sea again. Our work so far.

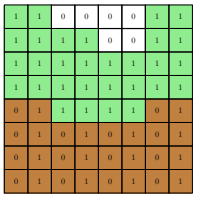
5.0.2 Step Two: Creating the Land

Is that all there is to it? Painting things with blue? No.

There are other possible values aside from `10` and `00` that we could use to paint colors. We could also have `11` and `01`. This is useful since we want to color things in with more than one color. We have blue assigned to `10` on Planet 1, while for the land we can use two other colors: `11` which we will assign 'green' and `01` which we will assign 'brown'. We can assign whatever colors we like but we can only choose three, not counting the background. This is the kind of limitation you run into when you only allow two bits for assigning possible colors.



((1)) planet1Charset $41



((2)) planet1Charset $43

Figure 5.10: Planet 1 Land uses two different characters that alternate to generate the land surface.

Figure 5.11: planet1Charset Land

The location and length of the landmass is randomly generated with a couple of constraints: it must be at least 128 bytes and not more than 256 bytes from the start of surface and it must be at least 32 bytes and not more than 150 bytes long. The result is that the planet surface will be mostly sea since the entire surface is 1024 bytes long.

Picking a random number between 128 and 256 is slightly convoluted in assembly:

```
        STA planetTextureSecondFromTopLayerPtrHi
        CLC
        ADC #$04
        STA planetTextureSecondFromBottomLayerPtrHi
        CLC
        ADC #$04
        STA planetTextureBottomLayerPtrHi

        LDA planetScrollSpeed
```

Listing 5.5: Convoluted

Neat Little Trick?

```
          JMP AnimateGilbyMovement

gilbyIsLanding    .BYTE $00
```

Listing 5.6: Neat

This little snippet's job is to return a quasi-random byte for use in the planet generation routines. To achieve this, it does something quite fiendish that is more or less unhead of in modern programming: it mutates itself.

When called for the first time it loads a value from the address at `randomPlanetData` to the accumulator. On first run `randomPlanetData` points to the address `$9ABB` which contains the value `$42`:

```
randomPlanetData
          .BYTE $42,$E4,$3F,$94,$4E,$29,$B0,$59
          .BYTE $2C,$FE,$7F,$B2,$40,$9B,$63,$2B
```

Listing 5.7: Not Quite Random Bytes

Before returning this value as its result it alters itself by changing `randomPlanetData` to reference `$9ABC` (`INC randomIntToIncrement`). In other words, it increments the pointer. In the assembly listing we make `randomIntToIncrement` reference the position that holds `randomPlanetData` by positioning it one byte before and adding a 1 to shift is reference beyond the byte holding `LDA` to `randomPlanetData`.

Every time the routine is called it increments the reference again so that the next time it will pick up whatever lies in the bytes beyond `9ABB`. The results it returns are never truly random, but random enough to permit the procedural generation of planets that they're used for.

— A

With a random start position selected, a similar convolution is performed to choose the length of the land mass:

```
          SBC #$28
          STA planetTextureTopLayerPtr
          STA planetTextureSecondFromTopLayerPtr
          STA planetTextureSecondFromBottomLayerPtr
```

```
        STA planetTextureBottomLayerPtr
        LDA #$83
        JMP DrawPlanetScroll
```

Listing 5.8: A convolution

Since the random number we get can be anything between `$00` - `$FF` (i.e. 0 and 255) and we want a number that's between 0 and 128 we need to do a bitwise `AND` to mask out Bit 7 which by itself is 128.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$FC	1	1	1	1	1	1	0	0
$7F	0	1	1	1	1	1	1	1
Result; $7C	0	1	1	1	1	1	0	0

AND'ing $FC and $7F gives $7C (124).

With the position and length selected we can start laying turf. We don't just plop down our basic land tiles. Posh and proper means giving the shore of the land its own look and feel. This we have in the characters `$5C` and `$5E` in our character set:

Figure 5.12: Character tiles for the left shore ($5C,$5E) and the right shore ($5D,$5F).

Now we can put the rest of the land down:

```
        CMP #$DA
```

Listing 5.9: Write pairs of `$41,$43` for the main land mass.

And finally the right shore:

```
;
    ------------------------------------------------------------

; ScrollPlanetLeft
```

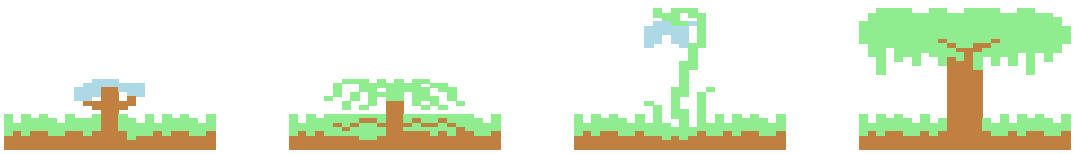
```
;
    ------------------------------------------------------------------------

ScrollPlanetLeft
        LDA planetScrollSpeed
        CLC
        ADC currentGilbySpeed
```

Listing 5.10: Drawing the right shore.

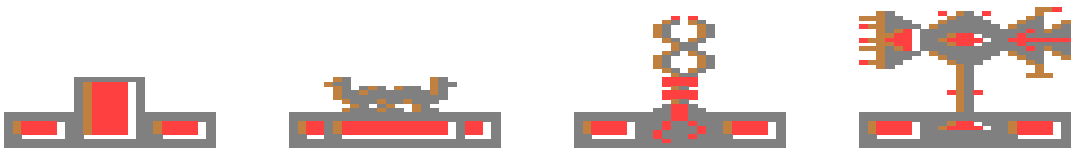
5.0.3 Step Three: Structures Structures Structures

The routines for adding structures to the planet are the opportunity to observe some assembly language cleverness. For each structure we draw we have to decide two things: where to drop it on the surface and what type of structure to draw. Apart from the Warp Gates, there are four structure types available.



((1)) planet1Charset littleStructureData | **((2))** planet1Charset mediumStructureData | **((3))** planet1Charset nextLargestStructure | **((4))** planet1Charset largestStructureData

Figure 5.13: The four possible structure types for Planet 1.



((1)) planet2Charset littleStructureData | **((2))** planet2Charset mediumStructureData | **((3))** planet2Charset nextLargestStructure | **((4))** planet2Charset largestStructureData

Figure 5.14: The four possible structure types for Planet 2.

You may be getting the sense that there is a sort of economy at work here. The structures are effectively the same for each planet, but with the textures swapped out. Your

intuition is correct, the structures are only defined once and the same definition does regardless of which planet we're painting:

```
b72CF   CLC
        ROR
        ROR
        ROR
        STA tempHiPtr1

        LDA planetTextureTopLayerPtr
        SEC
        SBC tempHiPtr1
```

Listing 5.11: The definitions of three of the structures above each of which serves all five planets.

The $FF at the end of each line serves as a sentinel for the drawing routine to know that the subsequent bytes are for the next layer 'up'. The $FE is a terminator, indicating there is no more data for the structure.

Drawing a structure is relatively straightforward so we'll cover that briefly first. Drawing the littlest structure provides the most compact example of the technique:

```
;
    ------------------------------------------------------------------

; StoreRandomPositionInPlanetInPlanetPtr
;
    ------------------------------------------------------------------

StoreRandomPositionInPlanetInPlanetPtr
        LDA #<planetOneBottomLayer
        STA planetPtrLo

        LDA #>planetOneBottomLayer
        STA planetPtrHi

        LDA charSetDataPtrLo
        BEQ LoPtrAlreadyZero

        INC planetPtrHi
        INC planetPtrHi

LoPtrAlreadyZero
        LDA planetPtrLo
```

```
        CLC
        ADC charSetDataPtrHi
        STA planetPtrLo

        LDA planetPtrHi
        ADC #$00
        STA planetPtrHi

        LDA planetPtrLo
        CLC
        ADC charSetDataPtrHi
        STA planetPtrLo

        LDA planetPtrHi
        ADC #$00
        STA planetPtrHi
```

Listing 5.12: The littlest structure has only two layers.

Given that we're only writing 4 bytes this is a lot of code. As we will see there are separate routines for each of the structures and unfortunately for our search for evidence of coding genius they're all identical. So this is a pretty open-and-shut case of code duplication. It would have been more compact to rationalize them down to a single function and use a pointer to the structure data instead of repeating almost verbatim the same assembly code for each structure.

```
;
    ------------------------------------------------------------------

; UpdateTopPlanetSurfaceColor
;
    ------------------------------------------------------------------

UpdateTopPlanetSurfaceColor
        LDX #$28
b731F   STA COLOR_RAM + LINE6_COL39,X
        STA COLOR_RAM + LINE7_COL39,X
        STA COLOR_RAM + LINE8_COL39,X
        STA COLOR_RAM + LINE9_COL39,X
        DEX
        BNE b731F
        RTS

;
    ------------------------------------------------------------------
```

```
; UpdateBottomPlanetSurfaceColor
;
    ------------------------------------------------------------------

UpdateBottomPlanetSurfaceColor
        LDX #$28
b7331   STA COLOR_RAM + LINE11_COL39,X
        STA COLOR_RAM + LINE12_COL39,X
        STA COLOR_RAM + LINE13_COL39,X
        STA COLOR_RAM + LINE14_COL39,X
        DEX
        BNE b7331
        RTS

;
    ------------------------------------------------------------------

; AnimateEntryLevelSequence
;
    ------------------------------------------------------------------

AnimateEntryLevelSequence
        LDA currentBottomPlanetDataLoPtr
        STA planetSurfaceDataPtrLo
        LDA currentBottomPlanetDataHiPtr
        STA planetSurfaceDataPtrHi
        JSR MutateSomeMoreOfThePlanetCharsetForEntrySequence
        LDA mutatedCharToDraw
        STA planetTextureCharset3,X
        INC planetSurfaceDataPtrHi
        JSR MutateSomeMoreOfThePlanetCharsetForEntrySequence
```

Listing 5.13: `DrawMediumStructure` and `DrawLargestStructure` are identical to each other and to `DrawLittleStructure` and `DrawNextLargestStructure`.

The cleverness comes a little earlier so let's console ourselves with that. When we've chosen a position to draw our structure we need to pick a type of structure at random. The secret to this is to store the addresses to our regrettably repetitive draw routines in a pair of arrays.

```
        CMP #$90
        BNE WriteSeaLoop

        ; Pick a random point between $8C00 and $8FFF for
```

Listing 5.14: A 'jump table' containing the addresses to our draw routines. The address for `DrawLittleStructure` happens to be $7486 so we store $74 in the first byte of `structureSubRoutineArrayHiPtr` and $86 in the first byte of `structureSubRoutineArrayLoPtr`.

With this in place our routine consists of getting a random number between 0 and 3, then using that as in index to pick out a value at the same position from `structureSubRoutineArrayLoPtr` and `structureSubRoutineArrayHiPtr`. We then store those values in `structureRoutineLoPtr` and `structureRoutineHiPtr` respectively. We now have a pointer to one of our draw routines at `structureRoutineLoPtr` which we can jump to with the simple command: `JMP (structureRoutineLoPtr)`.

```
ClearPlanetLoPtrs
        STA (planetSurfaceDataPtrLo),Y
        DEY
        BNE ClearPlanetLoPtrs
        INC planetSurfaceDataPtrHi
        LDA planetSurfaceDataPtrHi
        CMP (#>planetSurfaceData) + $10
        BNE ClearPlanetHiPtrs

        ; Fill $8C00 to $8FFF with a $40,$42 pattern. These are
    the
        ; character values that represent 'sea' on the planet.
        LDA #$8C
        STA planetSurfaceDataPtrHi
WriteSeaLoop
        LDA #$40
        STA (planetSurfaceDataPtrLo),Y
        LDA #$42
        INY
        STA (planetSurfaceDataPtrLo),Y
        DEY
        ; Move the pointers forward by 2 bytes
        LDA planetSurfaceDataPtrLo
        CLC
        ADC #$02
        STA planetSurfaceDataPtrLo
        LDA planetSurfaceDataPtrHi
        ADC #$00
```

Listing 5.15: `DrawRandomlyChosenStructure` picks a random position and a random draw routine to use at that position.

Rinse and repeat this for the length of the map and we get a surface with sea and land

that is dotted with structures of different types.

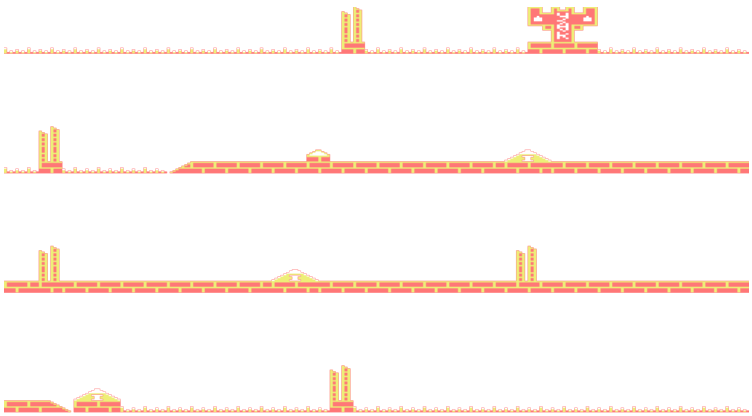


Figure 5.15: Planet 3 once `DrawRandomlyChosenStructure` has finished its business.

5.0.4 Step Four: Add the warp gate

Our final step is to add the warp gate.

Figure 5.16: The warp gates on each planet.

There's something funny here I haven't figured out yet. The routine for drawing the warp gate draws it twice. Yet each level has only one warp gate. Each one gets an initial position of $F1 and $05 respectively. This is used by `StoreRandomPositionInPlanetInPlanetPtr` to point to a position on the surface where the warp gate is drawn.

```
; planet surface types in all, I reckon, probably do one with
    grass
```

```
; and sea a bit like 'Sheep in Space', cos I did like that one.
    It'll
; be nice to have completely different planet surfaces in top
    and bottom
; of the screen. The neat thing is that all the surfaces have
    the same
; basic structures, all I do is fit different graphics around
    each one."
;
    ------------------------------------------------------------------


GeneratePlanetSurface
        LDA #<planetSurfaceData
        STA planetSurfaceDataPtrLo
        LDA #>planetSurfaceData
        STA planetSurfaceDataPtrHi

        ; Clear down the planet surface data from $8000 to
    $8FFF.
        ; There are 4 layers:
        ; Top Layer:    $8000 to $83FF - 256 bytes
        ; Second Layer: $8400 to $87FF - 256 bytes
        ; Third Layer:  $8800 to $8BFF - 256 bytes
        ; Bottom Layer: $8C00 to $8FFF - 256 bytes
        LDY #$00
```

Listing 5.16: Why does it draw 2 warp gates when there's only 1? Haven't figured this out yet..

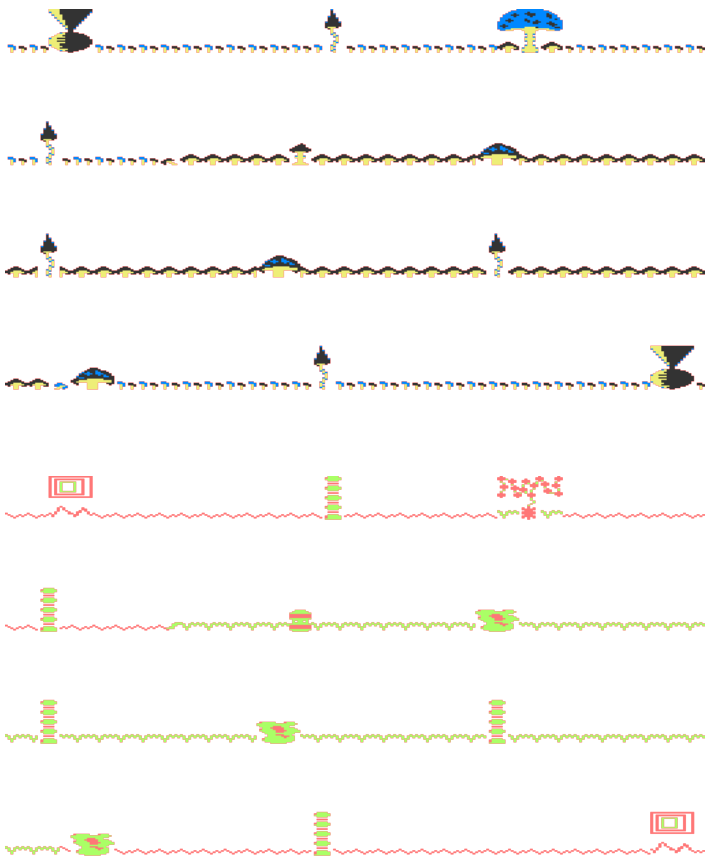


Figure 5.17: The final surfaces for Planets 4 and 5.

5.0.5 Inactive Lower Planet

When the lower planet is inactive a surface with land, sea, and a warp gate is displayed. This doesn't reuse any of the logic described above. Instead it is generated from some customized data in the routine `DrawLowerPlanetWhileInactive`.

```
b5E5D   LDA setToZeroIfOnUpperPlanet
        BEQ UpperPlanetInitializePlanetEntropyStatus

        ; Initialize the entropy on the lower planet if that's
        ; where we are.
        LDA #$08
```

```
        STA lowerPlanetEntropyStatus
        BNE SkipToUpdateEntropy

UpperPlanetInitializePlanetEntropyStatus
        LDA #$08
        STA upperPlanetEntropyStatus

SkipToUpdateEntropy
        DEC entropyUpdateRate
        BEQ MaybeUpdateDisplayedEntropy
        BNE UpdateDisplayedEntropyStatus

entropyUpdateRate         .BYTE $A3
upperPlanetEntropyStatus  .BYTE $08
lowerPlanetEntropyStatus  .BYTE $08
entropyDisplayUpdateRate   .BYTE $23

;-----------------------------------------------------------------
; MaybeUpdateDisplayedEntropy
;-----------------------------------------------------------------
MaybeUpdateDisplayedEntropy
        DEC entropyDisplayUpdateRate
        BNE UpdateDisplayedEntropyStatus
        LDA #$10
        STA entropyDisplayUpdateRate
        LDA #$00
        STA currentEntropy

        ; We update the entropy of the planet we're not on. So if
        ; we're on the upper planet we update the entropy of the lower.
        LDA setToZeroIfOnUpperPlanet
        BEQ UpdateLowerPlanetEntropy

        DEC upperPlanetEntropyStatus
        BNE b5E95
        INC currentEntropy
b5E95   LDA upperPlanetEntropyStatus
        CMP #$FF ; Decremented past zero?
        BNE UpdateDisplayedEntropyStatus

EntropyKillsGilby
        LDA #$02
        STA reasonGilbyDied ; Entropy
        JMP GilbyDied
        ; Returns

        ; Not reached.
        BNE UpdateDisplayedEntropyStatus
```

Listing 5.17: Draw the inactive lower planet.

Enemies and their Discontents

> ACONT
>
> This is the bit that I knew would take me ages to write and get glitch free, and the bit that is absolutely necessary to the functioning of the game. The module ACONT is essentially an interpreter for my own 'wave language', allowing me to describe, exactly, an attack wave in about 50 bytes of data. The waves for the first part of IRIDIS are in good rollicking shoot-'em-up style, and there have to be plenty of them. There are five planets and each planet is to have twenty levels associated with it. It's impractical to write separate bits of code for each wave; even with 64K you can run outta memory pretty fast that way, and it's not really necessary coz a lot of stuff would be duplicated. Hence ACONT.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

The bits and bytes that define the behaviour and appearance of wave after wave of Iridis Alpha's enemy formations - twenty across each of the five planets giving on hundred in all - takes up relatively little space given the sheer variety of adversaries the player faces.

6.0.1 You're a Waste of Space

Each 'wave' of enemies is defined by a 40 byte data structure, not 50 bytes as Minter initially suggested in his development diary. There's a little bit of waste going on in here too, bytes 10 to 14 are unused, while `Byte 15` is only ever set (to `$01`) by the wave data structure that describes the default explosion behaviour for enemy ships.

As we can see here, the sole purpose of `Byte 15` is to determine whether a new set of wave data needs to be loaded. This makes sense, once the animation is finished we'll need to load a new enemy ship. Still, you can't help thinking there might have been a way that didn't waste 99 bytes!

```
BulletHitShipOnBottomPlanet
        ; Hitting the warp gate increments the planet for warp.
        LDA levelEntrySequenceActive
        BNE AddPointsForHittingEnemyLowerPlanet
        LDA attractModeCountdown
        BNE AddPointsForHittingEnemyLowerPlanet
        INC bottomPlanetPointerIndex
        LDA bottomPlanetPointerIndex
        CMP currentBottomPlanet
        BNE AddPointsForHittingEnemyLowerPlanet

        LDA #$00
        STA bottomPlanetPointerIndex
        ; Get the points multiplier for hitting enemies in this level
        ; from the wave data.
AddPointsForHittingEnemyLowerPlanet
        LDY #$22
        LDA (currentShipWaveDataLoPtr),Y
        JSR CalculatePointsForByte2
        CLC
```

Listing 6.1: "Routine for Animating Enemy Sprites"

And actually, it's more than that because as we shall see the `ACONT` 40-byte data structure is defined more than once per wave. Separate instances are defined for later phases of the enemy ship, such as when it is first hit. Early examples of this in the game are the 'spinning rings' you get when you hit an enemy in the first level.

In all there are 200 instances of the `ACONT` data structure: 100 defining each of the enemy waves and another 100 defining the subsequent behaviour of the ships when hit. There isn't a one-to-one mapping here either - many of the effects are reused across levels and as we shall see there can be multiple stages in an enemy's lifecycle.

So there's already 1000 bytes or 1 kilobyte of wasted space in the level data due to bytes that are never or rarely used. That's out of a total of 8 kilobytes actually used.

Shocking stuff. Awful.

6.0.2 And You're a Waste of Space

`Bytes 33 - 34` seem to be left in an unfinished state. Wave 12 on Planet 5 has both populated, `flowchartArrowAsExplosion` is the only other wave that has anything in either byte, in this case `$60` in `Byte 33`.

This another 200 wasted bytes it seems but it seems these bytes have some game logic attached and when you look at what that logic is doing it seems broken.

```
        LDA #$00
```

```
        STA topPlanetPointerIndex
        ; Get the points multiplier for hitting enemies in this level
        ; from the wave data.
AddPointsForHittingEnemy
        LDY #$22
        LDA (currentShipWaveDataLoPtr),Y
        JSR CalculatePointsForByte2
        CLC
        ADC pointsEarnedTopPlanetByte1
        STA pointsEarnedTopPlanetByte1
        LDA pointsEarnedTopPlanetByte2
        ADC pointsToAddToPointsEarnedByte2
        STA pointsEarnedTopPlanetByte2
```

Listing 6.2: "Routine for Animating Enemy Sprites"

When Byte 34 ($21) is populated (and fire has not been pressed) the game will attempt to load a set of wave date from Bytes 33 and 34:

```
        STA bottomPlanetPointerIndex
        ; Get the points multiplier for hitting enemies in this level
        ; from the wave data.
AddPointsForHittingEnemyLowerPlanet
        LDY #$22
        LDA (currentShipWaveDataLoPtr),Y
        JSR CalculatePointsForByte2
        CLC
        ADC pointsEarnedBottomPlanetByte1
        STA pointsEarnedBottomPlanetByte1
        LDA pointsEarnedBottomPlanetByte2
        ADC pointsToAddToPointsEarnedByte2
        STA pointsEarnedBottomPlanetByte2

ContinueCalculatingScoreFromHit
        JSR DrawPlanetProgressPointers
        PLA
        TAX
        ; Get the points multiplier for hitting enemies in this level
```

Listing 6.3: "Routine for Animating Enemy Sprites"

In the case of the data for Planet 5 Level 12 this translates to whatever is at $1488. As it happens this is the address of the frequency data used in the title screen's music. So effectively pretty random data:

```
        ; Cycle the title screen gilby sprite between $C1 and $C8.
        ; For some reason, we use the letter 'I' from the 'IRIDIS
        ; ALPHA' title screen text to double as the storage for
        ; this value.
        ; The effect of cycling the sprite this way is to create an
        ; animated effect as it bounces along the screen.
        INC currentTitleScreenGilbySpriteValue
        LDA currentTitleScreenGilbySpriteValue
        CMP #LAND_GILBY8
        BNE ReturnFromTitleScreenUpdateSprites
        LDA #LAND_GILBY1
        STA currentTitleScreenGilbySpriteValue

ReturnFromTitleScreenUpdateSprites
        RTS
```

Listing 6.4: "Routine for Animating Enemy Sprites"

Clearly, no one has ever reached level 12 in planet 5!

6.0.3 Clever Business

> You pass the interpreter data, that describes exactly stuff like: what each alien looks like, how many frames of animation it uses, speed of that animation, colour, velocities in X— and Y— directions, accelerations in X and Y, whether the alien should 'home in' on a target, and if so, what to home in on; whether an alien is subject to gravity, and if so, how strong is the gravity; what the alien should do if it hits top of screen, the ground, one of your bullets, or you; whether the alien can fire bullets, and if so, how frequently, and what types; how many points you get if you shoot it, and how much damage it does if it hits you; and a whole bunch more stuff like that. As you can imagine it was a fairly heavy routine to write and get debugged, but that's done now; took me about three weeks in all I'd say.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

The level data does actually define some of this stuff. It does so by making heavy use of a simple but clever trick that in its way is very specific to 8-bit assembly programming: storing references to other data structures as a pair of bytes. We've discussed the way this works previously but we'll try again briefly here as it won't do any harm.

The 40-byte data structure that defines the default explosion animation (and behaviour, so far as it goes) is stored at position `$18C8` while the game is running. To use this explosion data when an enemy is killed, bytes 31 and 32 of the data structure contain the values `$C8,$18`.

When an enemy is hit, the game routine responsible for figuring out what to do next with it looks at bytes 31 and 32 and loads in the data structure at the address given by combining `$18` and `$C8` as the 'new' wave data that defines how that enemy ship will now behave. Since the data structure at `$18C8` basically says: animate an explosion sprite and don't move anywhere that is exactly what the enemy ship now does.

Here's the explosion data structure, which we've labelled `defaultExplosion` in our disassembly, in the first twenty bytes or so of it's gory detail:

```
defaultExplosion = $18C8
        ; Byte 1 (Index $00): An index into colorsForAttackShips that applies a
        ; color value for the ship sprite.
        .BYTE $07
        ; Byte 2 (Index $01): Sprite value for the attack ship for the upper planet.
        ; Byte 3 (Index $02): The 'end' sprite value for the attack ship's animation
        ; for the upper planet.
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 4 (Index $03): The animation frame rate for the attack ship.
        .BYTE $03
        ; Byte 5 (Index $04): Sprite value for the attack ship for the lower planet.
        ; Byte 6 (Index $05): The 'end' sprite value for the ship's lower planet animation.
```

```
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 7 (Index $06): Whether a specific attack behaviour is used.
        .BYTE $00
        ; Byte 8 (Index $07): Lo Ptr for an unused attack behaviour
        ; Byte 9 (Index $08): Hi Ptr for an unused attack behaviour
        .BYTE <nullPtr,>nullPtr
        ; Byte 10 (Index $09): Lo Ptr for an animation effect? (Doesn't seem to be used?)
        ; Byte 11 (Index $0A): Hi Ptr for an animation effect (Doesn't seem to be used?)?
        .BYTE <nullPtr,>nullPtr
        ; Byte 12 (Index $0B): some kind of rate limiting for attack wave
        .BYTE $00
        ; Byte 13 (Index $0C): Lo Ptr for a stage in wave data (never used).
        ; Byte 14 (Index $0D): Hi Ptr for a stage in wave data (never used).
        .BYTE <nullPtr,>nullPtr
        ; Byte 15 (Index $0E): Controls the rate at which new enemies are added?
        .BYTE $01
        ; Byte 16 (Index $0F): Update rate for attack wave
        .BYTE $0D
        ; Byte 17 (Index $10): Lo Ptr to the wave data we switch to when first hit.
        ; Byte 18 (Index $11): Hi Ptr to the wave data we switch to when first hit.
        .BYTE <nullPtr,>nullPtr
        ; Byte 19 (Index $12): X Pos movement for attack ship.
        .BYTE $80
        ; Byte 20 (Index $13): Y Pos movement pattern for attack ship.
```

Listing 6.5: "Routine for Animating Enemy Sprites"

We can see the first 7 bytes are concerned with the appearance and basic behaviour of the enemy. Bytes 2 and 3 define the sprite used for display on the upper planet, Bytes 5 and 6 for the lower planet. The reason there's two in each case is because they are describing the start and end point of the sprite's animation. The game will display `EXPLOSION_START $ED`) first, then cycle through the next two sprites until it reaches `EXPLOSION_START` + 3 (`$F0`).

6.0.4 Sprites Behaving Badly

We can see this in action in `AnimateAttackShipSprites`. When this routine runs Byte 4 has been loaded to `upperPlanetAttackShipInitialFrameRate` for the upper planet and `lowerPlanetAttackShipInitialFrameRate` for the lower planet. This routine is cycling through the sprites given by Byte 2 as the lower limit and Byte 3 as the upper limit. This is what the animation consists of: an animation effect achieved by changing the sprite from one to another to create a classic animation effect.

```
        RTS

;----------------------------------------------------------------------
; PlaySecondSoundEffect
;----------------------------------------------------------------------
PlaySecondSoundEffect
        LDA soundTmpLoPtr
        CLC
        ADC #$05
        STA soundTmpLoPtr
        LDX indexToCurrentOrSecondarySoundEffectPtr
        STA currentSoundEffectLoPtr,X
        LDA soundTmpHiPtr
        ADC #$00
        STA soundTmpHiPtr
        STA currentSoundEffectHiPtr,X
        RTS

TrySequenceByteValueWithHighBitSet
        AND #$80
        BEQ TrySequenceByteValueOf1
```

```
        JMP TrySequenceByteValueOf80

TrySequenceByteValueOf1
        LDA sequenceControlByte
        CMP #$01
        BNE TrySequenceByteValueOf2
        LDX offsetIntoSoundEffectBuffer
        LDA soundEffectBuffer,X
        CLC
        ADC dataforSoundEffectBuffer
        LDX nextEffectLoPtr
        STA soundEffectBuffer,X
        STA $D400,X  ;Voice 1: Frequency Control - Low-Byte
        JMP PlayNextSound
```

Listing 6.6: "Routine for Animating Enemy Sprites"

Byte 2 (loaded to `upperPlanetAttackShipAnimationFrameRate` comes into play here. It's decremented and as long as it's not zero yet the animation is skipped, execution jumps forward to `AnimateLowerPlanetAttackShips`:

```
        LDA soundTmpHiPtr
        ADC #$00
        STA soundTmpHiPtr
```

Listing 6.7: "Routine for Animating Enemy Sprites"

If it is zero, it instead gets reset to the initial value from Byte 2 (stored in `upperPlanetAttackShipInitialFram` and the current sprite for the enemy ship is incremented to point to the next 'frame' of the ship's animation:

```
        RTS

TrySequenceByteValueWithHighBitSet
        AND #$80
```

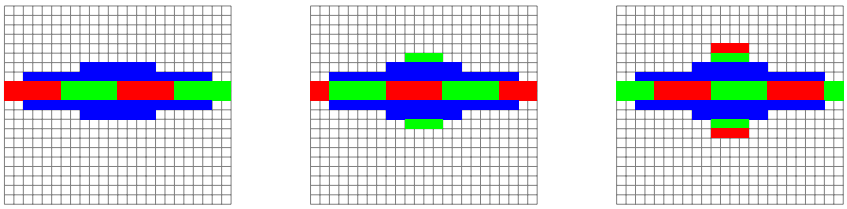
Listing 6.8: "Routine for Animating Enemy Sprites"

Next we check if we've reached the end of the animation by checking the value of Byte 3 (loaded to `upperPlanetAttackShipSpriteAnimationEnd`). If so, we reset `upperPlanetAttackShip2SpriteValue` to the value initially loaded from Byte 2 - and that is what will be used to display the ship the next time we pass through to animate the ship:

```
        JMP TrySequenceByteValueOf80

TrySequenceByteValueOf1
        LDA sequenceControlByte
        CMP #$01
        BNE TrySequenceByteValueOf2
        LDX offsetIntoSoundEffectBuffer
        LDA soundEffectBuffer,X
```

Listing 6.9: "Routine for Animating Enemy Sprites"



(a) FLYING_SAUCER1 **(b)** FLYING_SAUCER2 **(c)** FLYING_SAUCER3

Figure 6.1: The sprites used to animate the 'UFO' in the first level.

6.0.5 Enemy Movement

Enemy movement is controlled by two parameters in each direction: the number of pixels to move in one go and the number of cycles to wait between each movement. So for movement in the horizontal (or X direction) `Byte 19` controls the number of pixels to move at once, while `Byte 21` controls the number of cycles to wait between each movement. The same applies to `Byte 20` and `Byte 22` for the vertical (or Y direction).

If we look at Byte 19 and Byte 21 for Level 1 we can see that the the fast lateral movement of the 'UFO's is implemented by a relatively high value of $06 for the number of pixels it moves at each step while the interval between steps is relatively low ($01). Meanwhile the more gradual up and down movement is implemented by a value of $01 in Byte 20 and Byte 22.

For the second level ('bouncing rings') the horizontal movement is more constrained (Byte 19 is $00) while the vertical movement is more extreme (Byte 20 is $24) - achieving the bouncing effect.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$06	$01	$01	$01
2	$00	$00	$24	$02	$01
3	$00	$FA	$01	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Movement data for the first three levels.

The horizontal movement for level three is $FA, which would make you think the enemies must be moving horizontally extremely quickly. In fact, when the high bit is set a special behaviour is invoked:

```
;
    ----------------------------------------------------------------

PlayFirstSoundEffect
```

Listing 6.10: "From `UpdateAttackShipsXAndYPositions.`"

When the upper bit is set (e.g. $FC,$80) on the value loaded to the accumulator by `LDA` then `BMI` will return true and jump to `UpperBitSetOnXPosMovement`.

```
        BNE TrySequenceByteValueWithHighBitSet
        LDA dataforSoundEffectBuffer
        LDX nextEffectLoPtr
        STA soundEffectBuffer,X
        STA $D400,X   ;Voice 1: Frequency Control - Low-Byte

PlayNextSound
        JSR PlaySecondSoundEffect
        LDA nextEffectHiPtr
        BEQ PlayFirstSoundEffect
        CMP #$01
        BNE StorePointersAndReturn
        RTS

StorePointersAndReturn
```

Listing 6.11: "From `UpdateAttackShipsXAndYPositions`. "

This first line `EOR #$FF` performs an exclusive-or between Byte 19 in the `Accumulator` ($FA) and the value $FF. An exclusive-or, remember, is a bit by bit comparison of two bytes which will set a bit in the result if an only if the bit in one of the values is set but the other is not:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$FF	1	1	1	1	1	1	1	1
$FA	1	1	1	1	1	0	1	0
Result	0	0	0	0	0	1	0	1

X-OR'ing $FF and $FA gives $05.

This result is stored in `attackShipOffsetRate`:

Listing 6.12: "From `UpdateAttackShipsXAndYPositions`. "

Incremented:

```
PlayNextSound
```

Listing 6.13: "From `UpdateAttackShipsXAndYPositions`. "

And then subtracted from the enemy's X position:

```
        JSR PlaySecondSoundEffect
        LDA nextEffectHiPtr
        BEQ PlayFirstSoundEffect
        CMP #$01
```

Listing 6.14: "From `UpdateAttackShipsXAndYPositions`. "

The net result is a deceleration effect. This is observed in the way the licker ship wave will accelerate out to the center before dialling back again.

What is going on with Byte 7?

Byte 7 comes into play when setting the initial Y position of a new enemy. This initial vertical position is random, but subject to some adjustment:

```
        DEC gameSequenceCounter
        BEQ GetNewWaveDataForAnyDeadShips
mainGameProcReturnEarly
        RTS

gameSequenceCounter    .BYTE $14

;
    ----------------------------------------------------------------

; GetNewWaveDataForAnyDeadShips
;
    ----------------------------------------------------------------

GetNewWaveDataForAnyDeadShips
        LDA #$20
        STA gameSequenceCounter
        LDX #$02
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ b4A1F ; Didn't find any dead ships to replace.

        LDA topPlanetStepsBetweenAttackWaveUpdates
        CMP currentStepsBetweenTopPlanetAttackWaves
        BEQ b4A1F ; Skips updating waves for top planet.

        LDA topPlanetLevelDataLoPtr
        STA activeShipsWaveDataLoPtrArray,X
        LDA topPlanetLevelDataHiPtr
        STA activeShipsWaveDataHiPtrArray,X
        TXA
        TAY
        JSR UpdateCurrentShipWaveDataPtrs
        INC currentStepsBetweenTopPlanetAttackWaves

b4A1F   LDX #$08
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ReturnEarly ; Skips updating waves for bottom
    planet.

        LDA bottomPlanetStepsBetweenAttackWaveUpdates
        CMP currentStepsBetweenBottomPlanetAttackWaves
```

```
        BEQ ReturnEarly ; Skips updating waves for bottom
    planet.

        LDA bottomPlanetLevelDataLoPtr
        STA activeShipsWaveDataLoPtrArray,X
        LDA bottomPlanetLevelDataHiPtr
        STA activeShipsWaveDataHiPtrArray,X
        TXA
        CLC
        ADC #$02
        TAY
        INC currentStepsBetweenBottomPlanetAttackWaves
        ; Falls through

;
    ----------------------------------------------------------------

; UpdateCurrentShipWaveDataPtrs
;
    ----------------------------------------------------------------

UpdateCurrentShipWaveDataPtrs
        LDA activeShipsWaveDataLoPtrArray,X
        STA currentShipWaveDataLoPtr
        LDA activeShipsWaveDataHiPtrArray,X
        STA currentShipWaveDataHiPtr
        LDA #$00
        STA updatingWaveData
        STA shipHasAlreadyBeenHitByGilby,X
        ; Y contains the index of the previous enemy ship.
        STY previousAttackShipIndex
        ; Falls through

;
    ----------------------------------------------------------------

; GetWaveDataForNewShip
; Loads the wave data for the current wave from level_data.asm
    and level_data2.asm.
; currentShipWaveDataLoPtr is a reference to one of the data
    chunks in those
```

Listing 6.15: "The sub-routine `SetInitialRandomPositionUpperPlanet` in `GetWaveDateForNewShip`. "

The first order of business is to call `PutRandomByteInAccumulatorRegister` which gets a random value and stores it in the accumulator.

```
        CMP currentStepsBetweenTopPlanetAttackWaves
```

Since A can now contain anything from 0 to 255 ($00 to $FF) this needs to be adjusted to a meaningful Y-position value for the upper planet. So if we imagine PutRandomByteInAccumulatorRegister returned $85, we now do the following operations on it:

```
        BEQ b4A1F ; Skips updating waves for top planet.

        LDA topPlanetLevelDataLoPtr
```

First we do an AND #$3F with the value of $85 in A:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$85	1	0	0	0	0	1	0	1
$3F	0	0	1	1	1	1	1	1
Result	0	0	0	0	0	1	0	1

AND'ing $3F and $85 gives $05.

Our result is $05. The effect of the AND'ing here is to ensure that the random number we get back is between 0 and 63 rather than 0 and 255. Next we add $40 (decimal 64) to this result:

```

        LDA topPlanetLevelDataLoPtr
        STA activeShipsWaveDataLoPtrArray,X
```

This gives $45 and this is what we store as the initial Y position for the enemy.

You'll notice that the steps for SetInitialRandomPositionLowerPlanet are identical but with only the constant of the add value of $98 instead of $40. This is simply an additional offset to ensure that the Y position is lower on the screen for the initial position of the enemy on the lower planet.

We still haven't got into what Byte 7 is doing though. With an initial Y position determined, it looks like the intention was for Byte 7 to specify some adjustment to this value. But this looks like another bit of non-functioning game logic. If Byte 7 contains a value, the function will return early without any further adjustments. If it's zero it will then try Byte 9. If that's zero, it will return early. So the logic needs Byte 7 to be zero

and Byte 9 to contain something for anything to happen. That's never the case, so the the adjustment never happens:

```
        TXA
        TAY
        JSR UpdateCurrentShipWaveDataPtrs
        INC currentStepsBetweenTopPlanetAttackWaves

b4A1F   LDX #$08
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ReturnEarly ; Skips updating waves for bottom
    planet.

        LDA bottomPlanetStepsBetweenAttackWaveUpdates
        CMP currentStepsBetweenBottomPlanetAttackWaves
        BEQ ReturnEarly ; Skips updating waves for bottom
    planet.
```

Listing 6.16: "An adjustment that never happens. Byte 7 and Byte 9 are never set in this way"

This is definitely some forgotten code. Byte 7 is elsewhere used in combination with Byte 8 and Byte 9 to define an alternate enemy mode for some levels where the ship will supplement any dead ships with alternate enemy types and attack patterns periodically.

6.0.6 Pointer Data

This happens in `MaybeSwitchToAlternateEnemyPattern` in `UpdateAttackShipDataForNewShip`.

```
No3rdWaveData
        LDA joystickInput
        AND #$10
        BNE Byte34IsZero
        ; Check if we should load extra stage data for this
    enemy.
        ; FIXME: When this is set it would incorrectly expect
    there
        ; to be a hi/lo ptr in $20 and $21, when there isn't.
        LDY #$21
        LDA (currentShipWaveDataLoPtr),Y
        BEQ Byte34IsZero
        DEY
```

```
        JMP UpdateWaveDataPointersForCurrentEnemy
        ; Returns

Byte34IsZero
        LDA updateRateForAttackShips,X
        BEQ UpdateAttackShipDataForNewShip
        DEC updateRateForAttackShips,X
        BNE UpdateAttackShipDataForNewShip
        ; Controls the rate at which new enemies are added.
        ; This is only set when the current ship data is
    defaultExplosion
        LDY #$0E
        LDA (currentShipWaveDataLoPtr),Y
        BEQ UpdatePointersToWaveDataWhenFirstHit

        ; Are we on the top or bottom planet?
        TXA
        AND #$08
        BNE NewShipDataForBottomPlanet

        ; We're on the top planet.
        DEC currentStepsBetweenTopPlanetAttackWaves
        JMP UpdatePointersToWaveDataWhenFirstHit

NewShipDataForBottomPlanet
        DEC currentStepsBetweenBottomPlanetAttackWaves

UpdatePointersToWaveDataWhenFirstHit
        LDY #$10
        ; Y has been set to $10 above, so we're pulling in the
    pointer
        ; to the second tranche of wave data for this level.
        ; Or Y has been set by the caller.

;
    ----------------------------------------------------------------

; UpdateWaveDataPointersForCurrentEnemy
;
    ----------------------------------------------------------------

UpdateWaveDataPointersForCurrentEnemy
        LDA (currentShipWaveDataLoPtr),Y
```

Listing 6.17: "Byte 7 is used to periodically switch to an enemy mode defined by Bytes 8-9"

Byte 7 is used to drive the rate at which this routine switches over to the enemy data/-mode defined by Byte 8 and Byte 9.

```
        ; FIXME: When this is set it would incorrectly expect
    there
        ; to be a hi/lo ptr in $20 and $21, when there isn't.
        LDY #$21
        LDA (currentShipWaveDataLoPtr),Y
        BEQ Byte34IsZero
```

Listing 6.18: "`rateForSwitchingToAlternateEnemy` (Byte 7) is decremented and reloaded each time it reaches zero."

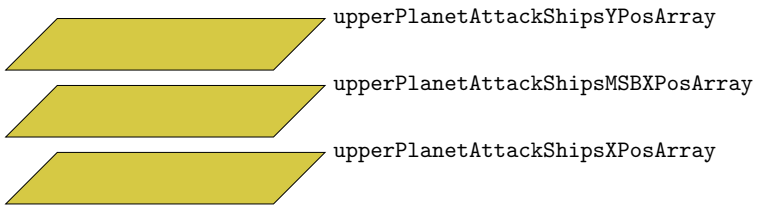
What this routine is going to do is replace the first dead ship it finds in the current wave with the wave data pointed to by Byte 8-9 and create a new enemy with the current ship's position with it.

First, we store the current ship's position. The way to do this is get the index (`Y`) for the current ship `X` and store each of the X and Y Position information into the accumulator first `A` and then push it onto the 'stack' (`PHA` which means 'push `A` onto the stack').

```
        JMP UpdateWaveDataPointersForCurrentEnemy
        ; Returns

Byte34IsZero
        LDA updateRateForAttackShips,X
        BEQ UpdateAttackShipDataForNewShip
        DEC updateRateForAttackShips,X
        BNE UpdateAttackShipDataForNewShip
        ; Controls the rate at which new enemies are added.
        ; This is only set when the current ship data is
    defaultExplosion
```

When this has run the stack of accumulator values now looks like this:



The stack after the code above has run with `upperPlanetAttackShipsXPosArray` at the top.

With our position data safely stashed away on the stack we now decide which planet we're on:

```
        LDA (currentShipWaveDataLoPtr),Y
        BEQ UpdatePointersToWaveDataWhenFirstHit

        ; Are we on the top or bottom planet?
```

If we're on the upper planet we use `SetXToIndexOfShipThatNeedsReplacing` look in the `activeShipsWaveDataHiPtrArray` for any ships that need replacing between positions `$02` and `$06`. If we don't find one, we return early:

```
        AND #$08
        BNE NewShipDataForBottomPlanet

        ; We're on the top planet.
        DEC currentStepsBetweenTopPlanetAttackWaves
```

If we do find one we can now pull (or 'pop') the positional data we stored away in the stack and assign that to the once-dead ship. First we use the index we retrieved to `X` to get the ship's index (`Y`) into the positional arrays:

```

NewShipDataForBottomPlanet
        DEC currentStepsBetweenBottomPlanetAttackWaves
```

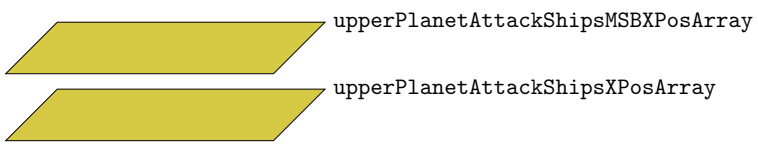
Then we pop the first positional item `upperPlanetAttackShipsYPosArray` from the top of the stack and store in the new ship's position in the array:

```

UpdatePointersToWaveDataWhenFirstHit
```

Listing 6.19: "`PLA` remove the top item from the stack and stores it in `A`

The stack now looks like this, popping from the stack has the effect of removing the first item:



Then we pop the rest of the items one by one and assign them to the new ship. We

ignore the sprite's MXB offset if it is zero:

```
        LDY #$10
        ; Y has been set to $10 above, so we're pulling in the
    pointer
        ; to the second tranche of wave data for this level.
        ; Or Y has been set by the caller.

;
    ----------------------------------------------------------------

; UpdateWaveDataPointersForCurrentEnemy
;
    ----------------------------------------------------------------

UpdateWaveDataPointersForCurrentEnemy
```

Listing 6.20: "`PLA` remove the top item from the stack and stores it in `A`

Now that we have set up the positional data for the new enemy we load all its other features from the data pointed to by Byte 8-9:

```
        PHA
        INY
        LDA (currentShipWaveDataLoPtr),Y
```

Let's take a closer look at this routine `UpdateWaveDataPointersForCurrentEnemy`. What it does in this instance is take the address pointed to by Bytes 8 and 9 and load the data there using the routine `GetWaveDataForNewShip`. To be used in this way the values in Bytes 8 and 9 are combined and treated as an address in memory. For example if Byte 8 contains `$70` and Byte 9 contains `$13` they are treated as providing the address `$1370`. This is the location where the enemy data for `planet1Level8Data` is kept so that is what is loaded.

Planet	Level	Byte 7	Bytes 8-9
1	11	$03	smallDotWaveData
1	14	$03	planet1Level8Data
2	19	$0C	landGilbyAsEnemy
3	4	$04	gilbyLookingLeft
3	6	$04	planet3Level6Additional
4	19	$01	planet4Level19Additional
5	3	$01	planet5Level3Additional
5	5	$05	planet5Level5Additional
5	14	$06	llamaWaveData

Byte 7 : Whether a specific attack behaviour is used.
Bytes 8-9 : Lo and Hi Ptr for alternate enemy mode

Table 6.1: Actual use of Bytes 7, 8, and 9. Note that the value in Byte 7 doesn't matter, as long as it's non-zero.

6.0.7 Enemy Behaviour

6.0.8 Level Movement Data

A Hundred Thousand Billion Theme Tunes

The theme music in Iridis Alpha is procedurally generated. There isn't a chunk of music data that the game plays every time you visit the title screen. Instead a new tune is generated for every visit. There's a distinction to be made here between procedural and random. The music isn't random: the first time you launch Iridis Alpha, and every subsequent time you launch it, you will hear the same piece of music. But as you let the game's attract mode cycle through and return to the title screen you will hear a new, different piece of music. Iridis Alpha has an infinite number of these tunes and it plays them in the same order every time you launch it and as it loops through the title sequence waiting for you to play.

Because the music is generated procedurally, and not randomly, you will hear the same sequence of tunes every time you launch the game so it appears to you as if the music was composed in advance and stored in the game waiting its turn. This is not the case.

Each piece of music is generated dynamically using the same algorithm but because the logic is chaotic enough, the smallest difference in the initial values fed into it will result in a completely different tune being generated.

The routine responsible for creating this music is remarkably short so I've reproduced it here in full before we start to dive in and try to understand what's going on.

```
;   |    66    |    D-4    |    4817    |    18    |    209    |
;   |    67    |    D#-4   |    5103    |    19    |    239    |
;   |    68    |    E-4    |    5407    |    21    |     31    |
;   |    69    |    F-4    |    5728    |    22    |     96    |
;   |    70    |    F#-4   |    6069    |    23    |    181    |
;   |    71    |    G-4    |    6430    |    25    |     30    |
;   |    72    |    G#-4   |    6812    |    26    |    156    |
;   |    73    |    A-4    |    7217    |    28    |     49    |
;   |    74    |    A#-4   |    7647    |    29    |    223    |
;   |    75    |    B-4    |    8101    |    31    |    165    |
```

```
;   |       80      |       C-5       |      8583     |       33      |      135      |
;   |       81      |       C#-5      |      9094     |       35      |      134      |
;   |       82      |       D-5       |      9634     |       37      |      162      |
;   |       83      |       D#-5      |     10207     |       39      |      223      |
;   |       84      |       E-5       |     10814     |       42      |       62      |
;   |       85      |       F-5       |     11457     |       44      |      193      |
;   |       86      |       F#-5      |     12139     |       47      |      107      |
;   |       87      |       G-5       |     12860     |       50      |       60      |
;   |       88      |       G#-5      |     13625     |       53      |       57      |
;   |       89      |       A-5       |     14435     |       56      |       99      |
;   |       90      |       A#-5      |     15294     |       59      |      190      |
;   |       91      |       B-5       |     16203     |       63      |       75      |
;   |       96      |       C-6       |     17167     |       67      |       15      |
;   |       97      |       C#-6      |     18188     |       71      |       12      |
;   |       98      |       D-6       |     19269     |       75      |       69      |
;   |       99      |       D#-6      |     20415     |       79      |      191      |
;   |      100      |       E-6       |     21629     |       84      |      125      |
;   |      101      |       F-6       |     22915     |       89      |      131      |
;   |      102      |       F#-6      |     24278     |       94      |      214      |
;   |      103      |       G-6       |     25721     |      100      |      121      |
;   |      104      |       G#-6      |     27251     |      106      |      115      |
;   |      105      |       A-6       |     28871     |      112      |      199      |
;   |      106      |       A#-6      |     30588     |      119      |      124      |
;   |      107      |       B-6       |     32407     |      126      |      151      |
;   |      112      |       C-7       |     34334     |      134      |       30      |
;   |      113      |       C#-7      |     36376     |      142      |       24      |
;   |      114      |       D-7       |     38539     |      150      |      139      |
;   |      115      |       D#-7      |     40830     |      159      |      126      |
;   |      116      |       E-7       |     43258     |      168      |      250      |
;   |      117      |       F-7       |     45830     |      179      |        6      |
;   |      118      |       F#-7      |     48556     |      189      |      172      |
;   |      119      |       G-7       |     51443     |      200      |      243      |
;   |      120      |       G#-7      |     54502     |      212      |      230      |
;   |      121      |       A-7       |     57743     |      225      |      143      |
;   |      122      |       A#-7      |     61176     |      238      |      248      |
;   |      123      |       B-7       |     64814     |      253      |       46      |
;   +---------------+-----------------+---------------+---------------+---------------+

                    ;       C   C#  D   D#  E   F   F#  G   G#  A   A#  B
titleMusicHiBytes   .BYTE $08,$08,$09,$09,$0A,$0B,$0B,$0C,$0D,$0E,$0E,$0F  ; 4
                    .BYTE $10,$11,$12,$13,$15,$16,$17,$19,$1A,$1C,$1D,$1F  ; 5
                    .BYTE $21,$23,$25,$27,$2A,$2C,$2F,$32,$35,$38,$3B,$3F  ; 6
                    .BYTE $43,$47,$4B,$4F,$54,$59,$5E,$64,$6A,$70,$77,$7E  ; 7
                    .BYTE $86,$8E,$96,$9F,$A8,$B3,$BD,$C8,$D4,$E1,$EE,$FD  ; 8

                    ;       C   C#  D   D#  E   F   F#  G   G#  A   A#  B
titleMusicLowBytes  .BYTE $61,$E1,$68,$F7,$8F,$30,$DA,$8F,$4E,$18,$EF,$D2  ; 4
                    .BYTE $C3,$C3,$D1,$EF,$1F,$60,$B5,$1E,$9C,$31,$DF,$A5  ; 5
                    .BYTE $87,$86,$A2,$DF,$3E,$C1,$6B,$3C,$39,$63,$BE,$4B  ; 6
                    .BYTE $0F,$0C,$45,$BF,$7D,$83,$D6,$79,$73,$C7,$7C,$97  ; 7
                    .BYTE $1E,$18,$8B,$7E,$FA,$06,$AC,$F3,$E6,$8F,$F8,$2E  ; 8

; This seeds the title music. Playing around with these first
; four bytes alters the first few seconds of the title music.
; The routine for the title music uses these 4 bytes to determine
; the notes to play.
; This array is periodically replenished from titleMusicSeedArray by
; SelectNewNotesToPlay.
titleMusicNoteArray .BYTE $00,$07,$0C,$07

; These variables are used to choose a value from titleMusicNoteArray,
; mutate it, and then use that as an index into titleMusicHiBytes/titleMusicLowBytes
; which gives PlayNoteVoice1/2/3 a note to play.
voice3NoteDuration              .BYTE $01
voice2NoteDuration              .BYTE $01
voice1NoteDuration              .BYTE $25
numberOfNotesToPlayInTune       .BYTE $85
voice3IndexToMusicNoteArray     .BYTE $00
voice2IndexToMusicNoteArray     .BYTE $01
voice1IndexToMusicNoteArray     .BYTE $02
notesPlayedSinceLastKeyChange   .BYTE $02
offsetForNextVoice2Note         .BYTE $0E
offsetForNextVoice1Note         .BYTE $07
offsetForNextVoice3Note         .BYTE $0E
;------------------------------------------------------
; PlayTitleScreenMusic
;------------------------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
```

Listing 7.1: Routine responsible for playing the title tune.

7.0.1 Some Basics

The rudiments of playing music on the Commodore 64 are simple. It has a powerful-for-its-time sound chip that has 3 tracks or 'voices'. You can play any note across 8 octaves on each of these voices together or separately. There are a whole bunch of settings you can apply to each voice to determine the way the note sounds. We'll cover a couple of these settings here but when it comes to playing music these extra settings aren't so important. They're much more useful when generating sound effects.

Playing a note on one of the voices consists of loading a two-byte value into the location (or 'register') associated with that voice. Here's the routine in Iridis used to play a note for the theme tune on Voice '1':

```
        STA baseNoteDuration

        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1

        ; Set up a new tune.
        LDA #$C0 ; 193
        STA numberOfNotesToPlayInTune
```

Listing 7.2: Plays a note on Voice 1. The routine is supplied with a value in Y that indexes into two arrays containing the first (Hi) and second (Lo) byte respectively associated with the selected note.

Once the selected bytes have been loaded into `$D400` and `$D401` the new note will start playing. It's as blunt an instrument as that. (Well not quite, we'll cover some other gory details soon).

The full list of available notes is given in the C64 Progammer's Reference Manual. I've adapted and reproduced it below.

Octave	Note	High Byte	Low Byte	Octave	Note	High Byte	Low Byte	Octave	Note	High Byte	Low Byte
0	C	$01	$0C	2	G#	$06	$A7	5	E	$2A	$3E
0	C#	$01	$1C	2	A	$07	$0C	5	F	$2C	$C1
0	D	$01	$2D	2	A#	$07	$77	5	F#	$2F	$6B
0	D#	$01	$3E	2	B	$07	$E9	5	G	$32	$3C
0	E	$01	$51	3	C	$08	$61	5	G#	$35	$39
0	F	$01	$66	3	C#	$08	$E1	5	A	$38	$63
0	F#	$01	$7B	3	D	$09	$68	5	A#	$3B	$BE
0	G	$01	$91	3	D#	$09	$F7	5	B	$3F	$4B
0	G#	$01	$A9	3	E	$0A	$8F	6	C	$43	$0F
0	A	$01	$C3	3	F	$0B	$30	6	C#	$47	$0C
0	A#	$01	$DD	3	F#	$0B	$DA	6	D	$4B	$45
0	B	$01	$FA	3	G	$0C	$8F	6	D#	$4F	$BF
1	C	$02	$18	3	G#	$0D	$4E	6	E	$54	$7D
1	C#	$02	$38	3	A	$0E	$18	6	F	$59	$83
1	D	$02	$5A	3	A#	$0E	$EF	6	F#	$5E	$D6
1	D#	$02	$7D	3	B	$0F	$D2	6	G	$64	$79
1	E	$02	$A3	4	C	$10	$C3	6	G#	$6A	$73
1	F	$02	$CC	4	C#	$11	$C3	6	A	$70	$C7
1	F#	$02	$F6	4	D	$12	$D1	6	A#	$77	$7C
1	G	$03	$23	4	D#	$13	$EF	6	B	$7E	$97
1	G#	$03	$53	4	E	$15	$1F	7	C	$86	$1E
1	A	$03	$86	4	F	$16	$60	7	C#	$8E	$18
1	A#	$03	$BB	4	F#	$17	$B5	7	D	$96	$8B
1	B	$03	$F4	4	G	$19	$1E	7	D#	$9F	$7E
2	C	$04	$30	4	G#	$1A	$9C	7	E	$A8	$FA
2	C#	$04	$70	4	A	$1C	$31	7	F	$B3	$06
2	D	$04	$B4	4	A#	$1D	$DF	7	F#	$BD	$AC
2	D#	$04	$FB	4	B	$1F	$A5	7	G	$C8	$F3
2	E	$05	$47	5	C	$21	$87	7	G#	$D4	$E6
2	F	$05	$98	5	C#	$23	$86	7	A	$E1	$8F
2	F#	$05	$ED	5	D	$25	$A2	7	A#	$EE	$F8
2	G	$06	$47	5	D#	$27	$DF	7	B	$FD	$2E

Figure 7.1: All available notes on the C64 and their corresponding hi/lo byte values. Note that Iridis Alpha only uses octaves 3 to 7. The available notes in octaves 1 to 2 are never used.

With 96 notes in total available, Iridis only uses 72 of them, omitting the 2 lowest octaves. We can see this when we look at the note table in the game. This pair of arrays are where the title music logic plucks the note to be played once it has dynamically selected one:

```
;-----------------------------------------------------------------
; The DNA pause mode mini game. Accessed by pressing *
; from within the Made in France pause mode mini game.
;-----------------------------------------------------------------
.include "dna.asm"


; This is the frequency table containing all the 'notes' from
; octaves 4 to 8. It's very similar to:
;  http://codebase.c64.org/doku.php?id=base:ntsc_frequency_table
; The 16 bit value you get from feeding the lo and hi bytes into
; the SID registers (see PlayNoteVoice1 and PlayNoteVoice2) plays
```

Listing 7.3: The lookup table for all of the notes used in the theme music. The two lowest available octaves are not used by the game. To see this for yourself compare the first entry in `titleMusicHiBytes`/`titleMusicLowBytes` ($08 and $61 giving $0861) with the entry highlighted in red in the previous table.

So now that we know where the notes are and how to make them go beep we just have to figure out the order that `PlayTitleScreenMusic` contrives to play them.

It would certainly help if we could see what the music looks like, so lets do that. Here is the opening title tune as sheet music in Western notation.

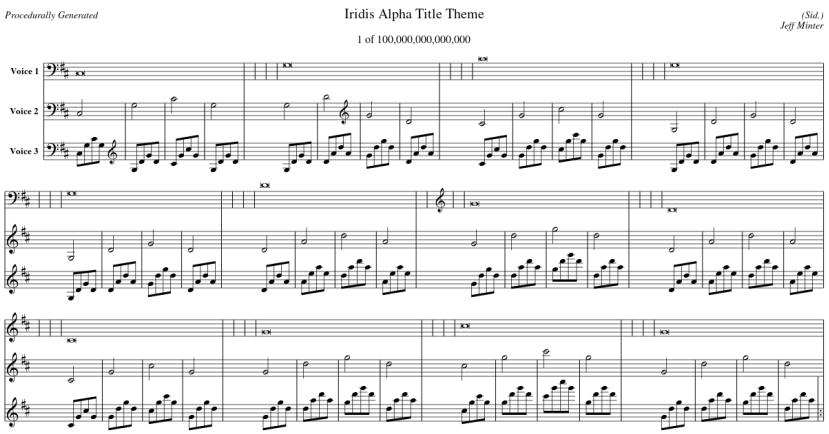
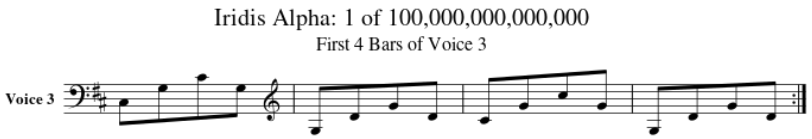


Figure 7.2: The first title tune in Iridis Alpha.

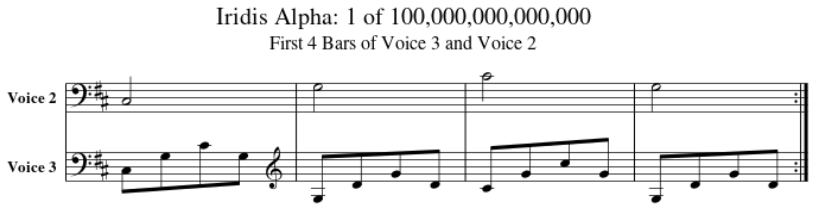
Structure

Even if you can't read sheet music notation some structure should be evident.

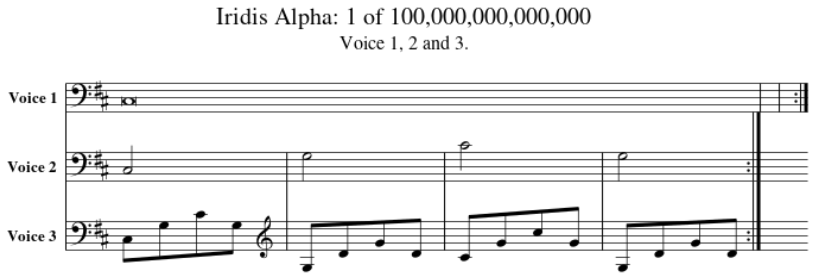
Voice 3 carries the main melody.



For every 4 notes Voice 3 plays, Voice 2 chimes in with a new note that it sustains until the next one.



Voice 1 does the same for every 16 notes that Voice 3 plays and every 4 notes of Voice 2..



Armed with this insight we can see it reflected in the logic in `PlayTitleScreenMusic`. This routine is called regularly by a system interrupt, a periodic wake-up call performed by the C64 CPU. So multiple times every second it is run and must figure out what new notes, if any, to play on each of the three voices.

Here it is deciding whether or not to play new note on Voice 1:

```
;   |   105   |   A-6   |   28871   |   112   |   199   |
;   |   106   |   A#-6  |   30588   |   119   |   124   |
;   |   107   |   B-6   |   32407   |   126   |   151   |
;   |   112   |   C-7   |   34334   |   134   |    30   |
;   |   113   |   C#-7  |   36376   |   142   |    24   |
;   |   114   |   D-7   |   38539   |   150   |   139   |
;   |   115   |   D#-7  |   40830   |   159   |   126   |
```

Listing 7.4: `MaybePlayVoice1` part of `PlayTitleScreenMusic`.

`voice1NoteDuration` is used to count the interval between notes on Voice 1. It's decremented on each visit and when it reaches zero it gets reset to 48 ($30) and a note is played. What's being counted here isn't seconds, it's cycles or 'interrupts'. So this translates to only a few seconds between notes being played.

The same is done for both Voice 2 and Voice 3 but the intervals are shorter: 12 ($0C)

and 3 ($03). This matches the relationship we see in the sheet music, one note in Voice 1 for every sixteen in Voice 3 (48/3=16) and one note in Voice 2 for every four in Voice 3 (12/3=4).

```
                    .BYTE $86,$8E,$96,$9F,$A8,$B3,$BD,$C8,$D4,$E1,$EE,$FD  ; 8

                    ;      C   C#  D   D#  E   F   F#  G   G#  A   A#  B
titleMusicLowBytes  .BYTE $61,$E1,$68,$F7,$8F,$30,$DA,$8F,$4E,$18,$EF,$D2  ; 4
                    .BYTE $C3,$C3,$D1,$EF,$1F,$60,$B5,$1E,$9C,$31,$DF,$A5  ; 5
                    .BYTE $87,$86,$A2,$DF,$3E,$C1,$6B,$3C,$39,$63,$BE,$4B  ; 6
```

Listing 7.5: `MaybePlayVoice2` part of `PlayTitleScreenMusic`.

```
; mutate it, and then use that as an index into titleMusicHiBytes/titleMusicLowBytes
; which gives PlayNoteVoice1/2/3 a note to play.
voice3NoteDuration          .BYTE $01
voice2NoteDuration          .BYTE $01
voice1NoteDuration          .BYTE $25
numberOfNotesToPlayInTune   .BYTE $85
```

Listing 7.6: `MaybePlayVoice3` part of `PlayTitleScreenMusic`.

Extracting the Title Music

Since each tune is dynamically generated there's nowhere for us to pull them from. We could record the tunes as audio files and maybe extract something useful that way. A feature of Vice, the C64 emulator, allows us to do something much simpler. We can log every note that's played to a text file and use that trace to reconstruct the tunes.

We launch Iridis Alpha with `x64` using the following command:

```
x64 -moncommands moncommands.txt orig/iridisalpha.prg
```

The `moncommands.txt` file contains a series of debugger directives that tells `x64` to log every value stored to the music registers at `$D400-$D415`. This will capture all notes played on all three voices as well as any updates made to the other sound parameters and write them to `IridisAlphaTitleMusicAll.txt`:

```
log on
logname "IridisAlphaTitleMusicAll.txt"
tr store D400 D415
```

We end up with `IridisAlphaTitleMusicAll.txt` full of lines like:

```
TRACE: 1  C:$d400-$d415  (Trace store)
#1 (Trace store d400)  279 052
.C:1598  8D 00 D4    STA $D400      - A:61 X:00 Y:00 SP:e8 ..-..I..   96135469
#1 (Trace store d401)  279 060
.C:159e  8D 01 D4    STA $D401      - A:08 X:00 Y:00 SP:e8 ..-..I..   96135477
#1 (Trace store d40b)  280 059
```

This examples gives us the value in `A` written to each register for Voice 1. For example, `$61` has been written to `$D400` and `$08` has been written to `$D401`.

We can now write a short Python notebook that parses this file and for each tune constructs three arrays, each representing a voice, with the sequence of notes played to each. For example, in the extract above we can extract `$0861` as the note 'C' in octave 3 played on Voice 1 (`$D400`–`$D401`). (Refer to the tables above to see why `$0861` translates to 'C-3'.

With the sequence of notes in three arrays, each representing one of the 3 voices, it is a simple matter to transform this into ABC format, a music notation frequently used for traditional music.

```
%abc-2.2
%%pagewidth 35cm
%%header "Example  page: $P"
%%footer " $T"
%%gutter .5cm
%%barsperstaff 16
%%titleformat R-P-Q-T C1 O1, T+T N1
%%composerspace 0
X: 2 % start of header
T: Iridis Alpha Title Theme
T:1 of 100,000,000,000,000
C: (Sid.)
O: Jeff Minter
R: Procedurally Generated
L: 1/8
K: D % scale: C major
V:1 name="Voice 1"
C,16     |      |      |      | G,16     |      |      |      | C16      |      |      |      | G,16     |      |      |      |
        G,16     |      |      |      | D16      |      |      |      | G16      |      |      |      | D16      |      |      |      |
          | C16      |      |      |      | G16      |      |      |      | c16      |      |      |      | G16      |      |      |
            | :|
V:2 name="Voice 2"
C,4      | G,4      | C4       | G,4      | G,4      | D4       | G4       | D4       | C4       | G4       | c4       | G4       | G,4      | D4
           | G4       | D4       | G,4      | D4       | G4       | D4       | D4       | A4       | d4       | A4       | G4       | d4       | g4
           | d4       | D4       | A4       | d4       | A4       | C4       | G4       | c4       | G4       | G4       | d4       | g4       | d4
         | c4       | g4       | c'4      | g4       | G4       | d4       | g4       | d4       | :|
V:3 name="Voice 3"
C,1G,1C1G,1|G,1D1G1D1|C1G1c1G1|G,1D1G1D1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|C1G1c1G1|G1d1g1d1|c1g1c'1g1|
        G1d1g1d1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|G,1D1G1D1|D1A1d1A1|G1d1g1d1|D1A1d1A1|D1A1d1A1|A1e1a1e1|
        d1a1d'1a1|A1e1a1e1|G1d1g1d1|d1a1d'1a1|g1d'1g'1d'1|d1a1d'1a1|D1A1d1A1|A1e1a1e1|d1a1d'1a1|A1e1a1e1|C1G1c1G1
        |G1d1g1d1|c1g1c'1g1|G1d1g1d1|G1d1g1d1|d1a1d'1a1|g1d'1g'1d'1|d1a1d'1a1|c1g1c'1g1|g1d'1g'1d'1|c'1g'1c''1g'
        1|g1d'1g'1d'1|G1d1g1d1|d1a1d'1a1|g1d'1g'1d'1|d1a1d'1a1|:|
```

Listing 7.7: Title Tune No 1 in ABC format

We can then use the tool 'abcm2ps' to transform this into an SVG image file giving the music in standard Western notation.

Phrasing

Now that we've identified the underlying 4-bar structure of the arrangement. We can take a closer look at the phrasing of the invidividual parts. Voice 3 has a simple repetitive structure for each 4-bar phrase:

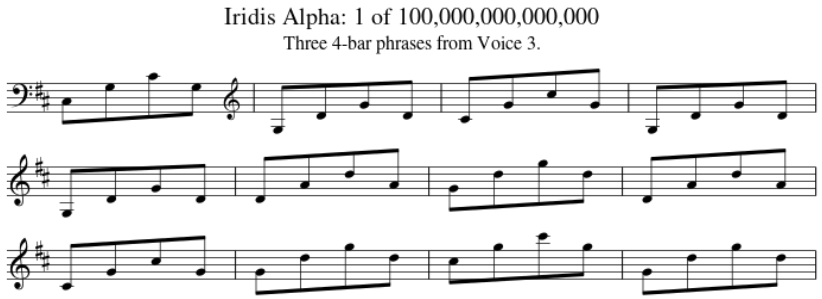


Figure 7.3: Bars 2 and 4 are always repeated

Bars 2 and 4 are repeated. Each bar consists of the same tonic formula: three notes rising two notes at a time, falling back on the final note. The difference between bars 1 and 3 is a simple key change.

This 4 note basis is driven by the 4 bytes in `titleMusicNoteArray`. Generating the music for Voice 3 consists of calculating and loading 4 values into this array and using them as an index into `titleMusicLowBytes/titleMusicHiBytes` to play the actual note.

```
; based frequency. This is usually 440hz, but not here.
;
; In fact the values here are the standard ones given in the
; Commodor 64 Programmer's Reference Guide. The decimal values
    are
; below in the table in Appendix D there:
;
    +------------------------------+---------------------------------------+
```

Notice how the values populated in `titleMusicNoteArray` at start-up match the structure of our basic tonic formula, e.g. C3-G3-C4-G3.

titleMusicNoteArray	titleMusicHiBytes	titleMusicLowBytes	Note
$00	$08	$61	C-3
$07	$8F	$0C	G-3
$0C	$C3	$10	G-3
$07	$8F	$0C	G-3

Figure 7.4: The value in `titleMusicNoteArray` is an index into `titleMusicHiBytes/titleMusicLowBytes`.

Playing the 4 note phrase we've stored in this array is done here:

```
voice2IndexToMusicNoteArray    .BYTE $01
voice1IndexToMusicNoteArray    .BYTE $02
notesPlayedSinceLastKeyChange  .BYTE $02
offsetForNextVoice2Note        .BYTE $0E
offsetForNextVoice1Note        .BYTE $07
offsetForNextVoice3Note        .BYTE $0E
;------------------------------------------------------
; PlayTitleScreenMusic
;------------------------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
        RTS

MaybeStartNewTune
        LDA previousBaseNoteDuration
        STA baseNoteDuration

        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1

        ; Set up a new tune.
```

```
        LDX voice1IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice1Note
        TAY
        STY offsetForNextVoice2Note

        JSR PlayNoteVoice1

        INX
        TXA
```

The variable that's doing a bit of extra work here is `offsetForNextVoice3Note`. This is what's shifting the notes for subsequent bars from the base position of C3-G3-C4-G3 to G3-D4-G4-D4. This value has to get updated after every four notes, otherwise we just keep playing the same four notes over and over again.

The obvious place to do this is when play a note on Voice 2, which is something we're already doing every 4 notes in Voice 3.

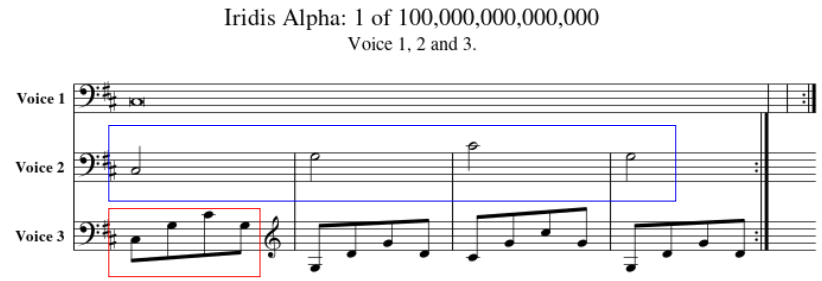
```
                        .BYTE $86,$8E,$96,$9F,$A8,$B3,$BD,$C8,$D4,
    $E1,$EE,$FD   ; 8

                        ;       C    C#   D    D#   E    F    F#   G    G#   A
       A#   B
titleMusicLowBytes   .BYTE $61,$E1,$68,$F7,$8F,$30,$DA,$8F,$4E,
    $18,$EF,$D2   ; 4
                        .BYTE $C3,$C3,$D1,$EF,$1F,$60,$B5,$1E,$9C,
    $31,$DF,$A5   ; 5
                        .BYTE $87,$86,$A2,$DF,$3E,$C1,$6B,$3C,$39,
    $63,$BE,$4B   ; 6
                        .BYTE $0F,$0C,$45,$BF,$7D,$83,$D6,$79,$73,
    $C7,$7C,$97   ; 7
                        .BYTE $1E,$18,$8B,$7E,$FA,$06,$AC,$F3,$E6,
    $8F,$F8,$2E   ; 8

; This seeds the title music. Playing around with these first
; four bytes alters the first few seconds of the title music.
; The routine for the title music uses these 4 bytes to
    determine
; the notes to play.
; This array is periodically replenished from
    titleMusicSeedArray by
; SelectNewNotesToPlay.
titleMusicNoteArray .BYTE $00,$07,$0C,$07

; These variables are used to choose a value from
    titleMusicNoteArray,
; mutate it, and then use that as an index into
    titleMusicHiBytes/titleMusicLowBytes
; which gives PlayNoteVoice1/2/3 a note to play.
voice3NoteDuration            .BYTE $01
```

As we can see the mechanics of playing a note for Voice 2 are otherwise the same as Voice 3. We're playing the same phrase encoded in `titleMusicNoteArray` that is played by Voice 3 but just over a longer period of time. And if you look closely again at the first four bars of the first title tune you can see that Voice 2 is in fact playing the exact same 4 notes of the first bar of Voice 3.



The same thing happens for Voice 1: it is playing the same notes as the first bar of Voice 3 but over 16 bars (1 every 4 bars).

So ultimately what we have underlying every tune generated by Iridis Alpha is a 16-bar structure where the same 4 notes are played by Voice 3 in its first bar, Voice 2 in its first 4 bars, and Voice 1 over the full 16 bars. This structure recurs every 16 bars, each time using the 4 initial notes from Voice 3.

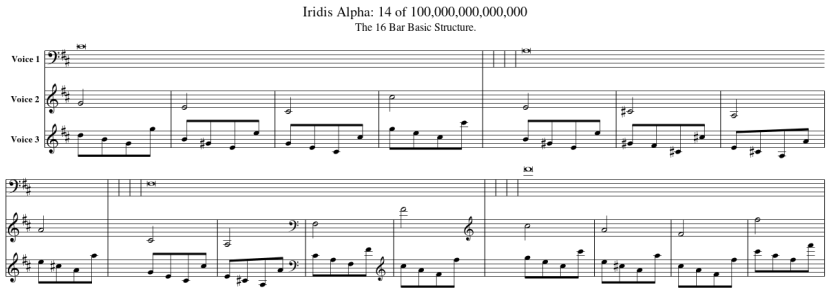


Figure 7.5: A full 16 bar passage showing the nested structure of Voices 1 and 2

This is a nested structure with the initial musical phrase that occurs every 4 bars in Voice 3 being picked up by Voice 2 and the one that occurs at every 16th bar being picked up by Voice 1.

The second, finer-grained structure of each tune lies in Voice 3 and consists of selecting a fundamental 4 note pattern (as we discussed above) and applying that same pattern to the key change between each 4 note phrase!

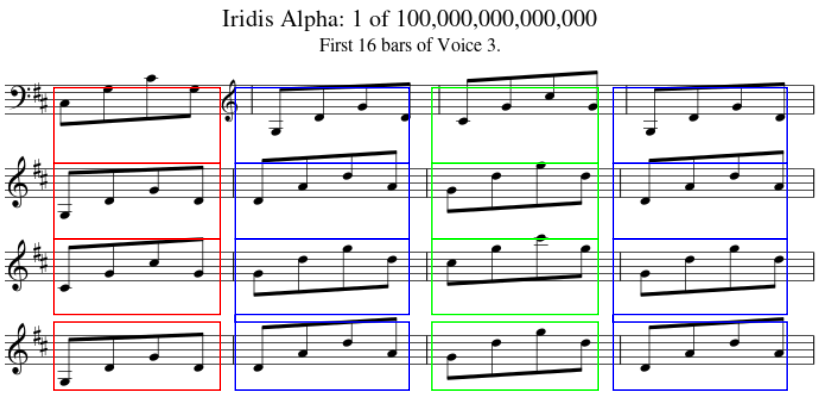


Figure 7.6: The G3-C4-G4-C4 pattern used to construct the 4 note pattern is also used to construct the key changes in each 4-bar sequence (red-blue-green-blue).

This is why we observed the repeating structure of Bars 2 and 4 earlier! It's the same pattern used to construct the 4 note formula.

But how do we choose the key for the start of each 4-bar pattern? By applying the same pattern to the start of each 4-bar section!

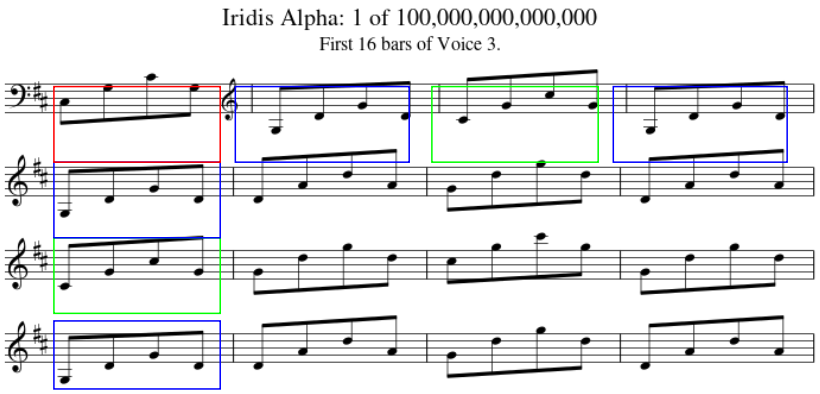


Figure 7.7: The start of each 4 bar pattern in a 16 bar cycle uses each of the 4-note patterns from the first 4 bars.

If we look at two other procedurally generated tunes we can see the same pattern:

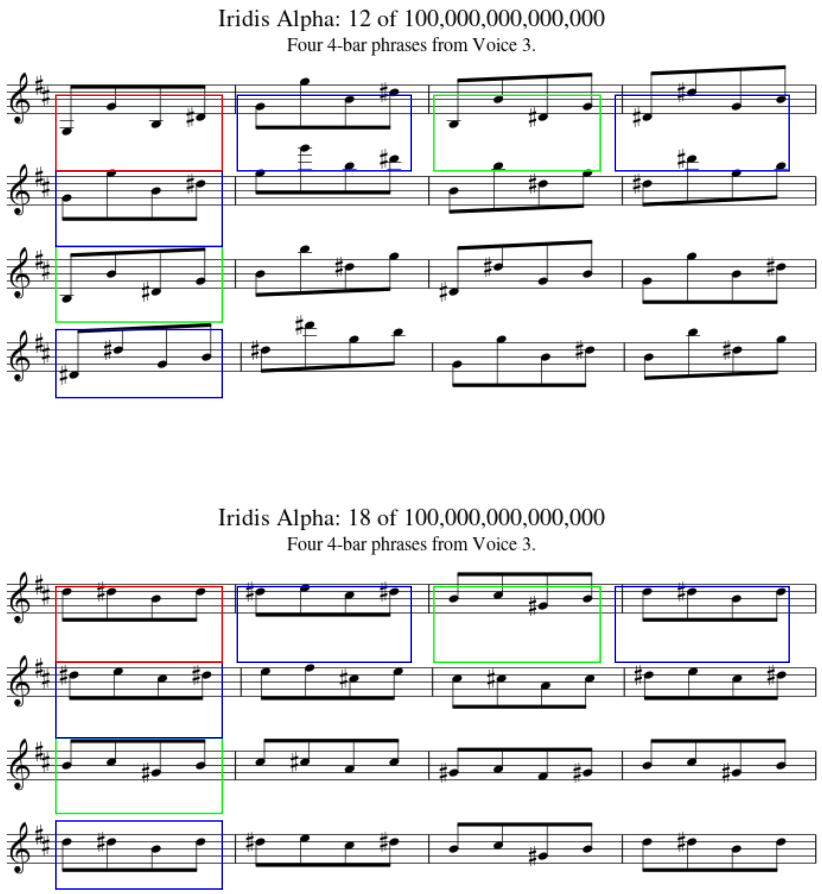


Figure 7.8: The same patterns in Tunes 12 and 18.

Seeding the Random

We've established how each tune is built entirely off the same 4-byte sequence, all the way from selecting notes to play to filling out the larger structure of the tune at almost every level. What remains is to see how this 4-byte sequence is selected. We know it's not entirely random since if it was, none of us would ever hear the same tune.

The selection of our 4-byte structure for each tune happens in `SelectNewNotesToPlay`. Once a seed value has been plucked, this is used as an index into `titleMusicSeedArray` and the next four values are populated into our magic 4-byte sequence that determines everything `titleMusicNoteArray`.

```
        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in titleMusicNoteArray.
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice3Note
        TAY
        JSR PlayNoteVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
        ; Since it's only 4 bytes long ensure we wrap
        ; back to 0 if it's greater than 3.
        AND #$03
```

Listing 7.8: Put a seed byte in the accumulator and multiply this by 4 if it's not zero. This gives us what we need for the next step.

```
;
    ------------------------------------------------------------------

; PlayNoteVoice1
;
    ------------------------------------------------------------------

PlayNoteVoice1
        LDA #$21
        STA $D404     ;Voice 1: Control Register
        LDA titleMusicLowBytes,Y
        STA $D400     ;Voice 1: Frequency Control - Low-Byte
        LDA titleMusicHiBytes,Y
        STA $D401     ;Voice 1: Frequency Control - High-Byte
        RTS

;
    ------------------------------------------------------------------

; PlayNoteVoice2
;
    ------------------------------------------------------------------
```

Listing 7.9: Use our seed value to pull 4 bytes from `titleMusicSeedArray` and store them in `titleMusicNoteArray`

```
        JSR PlayNoteVoice2
        INX
        TXA
        AND #$03
        STA voice2IndexToMusicNoteArray

MaybePlayVoice3
        DEC voice3NoteDuration
        BNE ReturnFromTitleScreenMusic

        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in titleMusicNoteArray.
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
```

Listing 7.10: Our seed bank for 4-byte sequences. It's 64 bytes long giving 16 possibles sequences in all.

The real source of variety here is this 'seed value' that we pluck at the very start of the process. This is done by `PutProceduralByteInAccumulator` at the very start of `SelectNewNotesToPlay`.

```
        ADC #$04
        DEX
        BNE MultiplyRandomNumBy4

        ; Fill titleMusicNoteArray with the next four bytes from
        ; titleMusicSeedArray.

InitializeSeedLoop
        ; Put our random number in Y and use it as index into
        ; the seed array.
        TAY
        ; Initialize X to 0, we will use this to iterate up to
        ; 4 bytes for pulling from titleMusicSeedArray.
        LDX #$00
```

Listing 7.11: Our seed value ultimately comes from `sourceOfSeedBytes`.

And `sourceOfSeedBytes` turns out to be a string of 256 random-looking data at `$9A00`:

```
*=$9A00
sourceOfSeedBytes
        .BYTE $E0,$D3,$33,$1F,$BF,$EC,$EF,$3E
        .BYTE $FA,$70,$DA,$26,$87,$C2,$C9,$9C
        .BYTE $F7,$FB,$C8,$85,$C1,$A9,$64,$AD
        .BYTE $6B,$DE,$8B,$8F,$05,$5E,$54,$51
        .BYTE $78,$0A,$6E,$6F,$FD,$0C,$A5,$32
        .BYTE $F5,$56,$44,$75,$38,$D6,$23,$98
        .BYTE $61,$D5,$49,$C6,$F2,$95,$BA,$08
        .BYTE $C3,$3D,$F4,$F0,$21,$48,$84,$02
        .BYTE $7E,$5B,$68,$55,$04,$92,$AE,$34
        .BYTE $72,$F6,$71,$A1,$39,$4F,$74,$E5
        .BYTE $E8,$31,$9A,$C7,$E3,$86,$6D,$14
        .BYTE $60,$CD,$50,$FF,$82,$52,$66,$9E
        .BYTE $E9,$53,$25,$93,$07,$77,$2E,$D7
        .BYTE $1A,$62,$80,$B7,$0D,$1B,$15,$46
        .BYTE $CE,$AA,$47,$24,$8D,$E1,$18,$67
        .BYTE $6A,$4A,$F1,$B9,$D0,$91,$BC,$EE
        .BYTE $B5,$D1,$7B,$A0,$DB,$36,$45,$E7
        .BYTE $11,$22,$81,$FC,$58,$30,$28,$CB
        .BYTE $8C,$B1,$0B,$A7,$DC,$B4,$9D,$57
        .BYTE $B3,$ED,$3C,$43,$16,$8A,$EA,$D8
        .BYTE $0E,$89,$1D,$1E,$DF,$9F,$BD,$BB
        .BYTE $F9,$D9,$01,$3B,$7A,$BE,$69,$B8
        .BYTE $5A,$A6,$E2,$96,$F8,$AC,$6C,$12
        .BYTE $2D,$19,$2A
randomPlanetData
        .BYTE $42,$E4,$3F,$94,$4E,$29,$B0,$59
        .BYTE $2C,$FE,$7F,$B2,$40,$9B,$63,$2B
        .BYTE $90,$73,$97,$EB,$F3,$C0,$79,$8E
        .BYTE $27,$06,$0F,$7C,$A4,$C5,$41,$99
        .BYTE $CA,$17,$A8,$D4,$AB,$5F,$5D,$B6
        .BYTE $4C,$7D,$E6,$CC,$37,$09,$35,$65
        .BYTE $D2,$83,$2F,$4B,$A3,$76,$DD,$3A
        .BYTE $00,$20,$4D,$C4,$03,$1C,$A2,$AF
        .BYTE $88,$CF,$5C,$13,$10
```

Listing 7.12: Our seed value ultimately comes from `sourceOfSeedBytes`.

As you might have guessed by now, given that there are only 16 possible sequences to choose from the seed bank there must actually be a lot less than a hundred thousand billion possible them tunes. With only 16 sequences there may even be only 16!

Well yes, there are certainly a lot closer to just 16 than a hundred thousand billion. The variety of values we get from `sourceOfSeedBytes` is not really of any account in the number of tunes we can generate. We're just using it to get pseudo-random but relatively predicatble values between 0 and 15 and using that to choose one of the 16 4-byte 'tune seeds'.

There's an additional bit of variability that gives us more than just 16 tunes though. This is the value we add to the note's index before we play it:

```
;------------------------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
        RTS

MaybeStartNewTune
        LDA previousBaseNoteDuration
```

Listing 7.13: `offsetForNextVoice3Note` introduces additional tune permutations.

When we select a new tune the value `offsetForNextVoice3Note` may be carrying over a value from the previous tune so rather than be a consistent value every time the 'tune seed' is selected, it will vary in value. The result is that the logic will select a different note-group even though it used the same 4-byte 'tune seed'.

Since the value in `offsetForNextVoice3Note` is ultimately loaded from `titleMusicSeedArray`:

```
;   |       91      |       B-5       |     16203     |       63      |
          75      |
;   |       96      |       C-6       |     17167     |       67      |
          15      |
;   |       97      |       C#-6      |     18188     |       71      |
          12      |
```

Listing 7.14: `offsetForNextVoice1Note` is loaded from `titleMusicNoteArray` and is propagated down to `offsetForNextVoice3Note`.

In practice there are 16 possible voice note sequences and 12 unique possible byte values to load from `titleMusicSeedArray` so there are 192 possible tunes.

[I can only get it to generate 80 so I'm missing something here.]

So the Hundred Thousand Billion is a lie. Iridis will indeed play a hundred thousand billion times if you leave it running long enough but ultimately even when we account for variations in key it can only ever play 192 unique title tunes.

Sorry for getting your hopes up.

Another 16^4 Tunes

It's possible to dig into the making of the title music and how Jeff Minter arrived the music configuration he did thanks to a number of tiny demo programs that survive from the period when he was developing Iridis Alpha.

Jeff distributed these on Compunet in the summer of 1986.

It turns out he was inspired by an article in 'Byte' magazine from June 1986 that described how to make 'Fractal Music'. This article outline a version of the algorithm that Jeff ultimately adopted. The 'self-similarity' we encountered in the way the Iridis Alpha theme tunes are constructed, a four-note structure repeated across different time intervals on each of the three voices, finds its roots in this article.

MUSICAL FRACTALS

of the musical structure.

In listing 4, we have limited the number of layers to three, but given the hardware, you could add layers to the musical structure until you achieve a point of aural saturation. You reach saturation when the input exceeds your aural resolution (when an additional layer makes the music move so fast that its tones become blurs) or surpasses the limitations of the synthesis system (when the tones in a new layer are too high for the synthesizer to play).

You can make a musical fractal that sounds less mechanical by incorporating a certain degree of randomness into the previous, nonrandom example. The idea is to produce random offsets of the original, nonrandom values. The program shown in listing 4 gives you the option of calling for Brownian random offsets in the selection of pitch intervals on the second and third layers. For the second layer the program will impart a change in the range of 0 to 6 semitones above or below the pitch value specified. The same range and distribution are applied for the third layer to those tones produced for the second layer.

The technique for creating this musical fractal is quite closely related to the technique in computer graphics for making fractal, Brownian mountain ranges (see reference 7). This complex computer graphic is made by nesting triangles, where the sides of the triangles are offset by some random amount proportional to the length of that side. In the musical fractals produced by the Brownian variation that is given in listing 4, an analogous technique is applied.

The final musical fractal uses a technique similar to the one described above: You generate an original layer and then add faster layers to it. You generate all layers through a 1/*f*-noise algorithm producing self-similar patterns. Because it is a random process, its similarity is statistically, rather than literally, the same. This fact lends a varied yet consistent surface to the resulting music. (The code is provided in listing 5.)

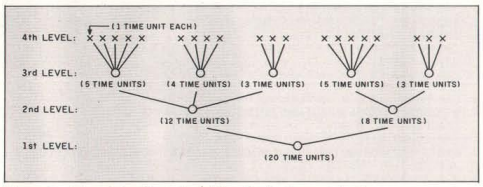


Figure 5a: *The relationship in time of four levels of generated notes.*

A Layered Structure

To understand how the layers are made, you need to understand the concept of *pitch class*. A pitch class is a C or an F#, for example, without regard for its register. In other words, the highest and lowest Cs on the piano keyboard share the same pitch class, C; they differ only in octave.

You don't specify the number of notes in the first layer directly. Instead, you begin by specifying the number of different pitch classes you want to generate in the first layer. You may have noticed that the 1/*f* algorithm commonly produces repeated notes. Therefore, the contents of the layer are related to pitch-class diversity, not simply to the number of pitches in the layer. The number of pitches in the layer does determine the perceived

(continued)

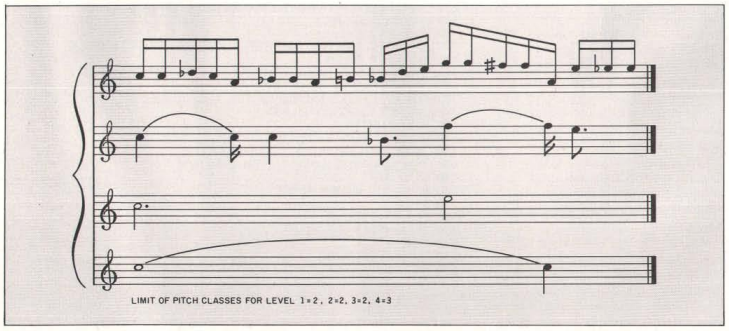


Figure 5b: A *four-layer musical structure generated using the 1/f-noise algorithm.*

Figure 8.1

8.0.1 Taurus:Torus

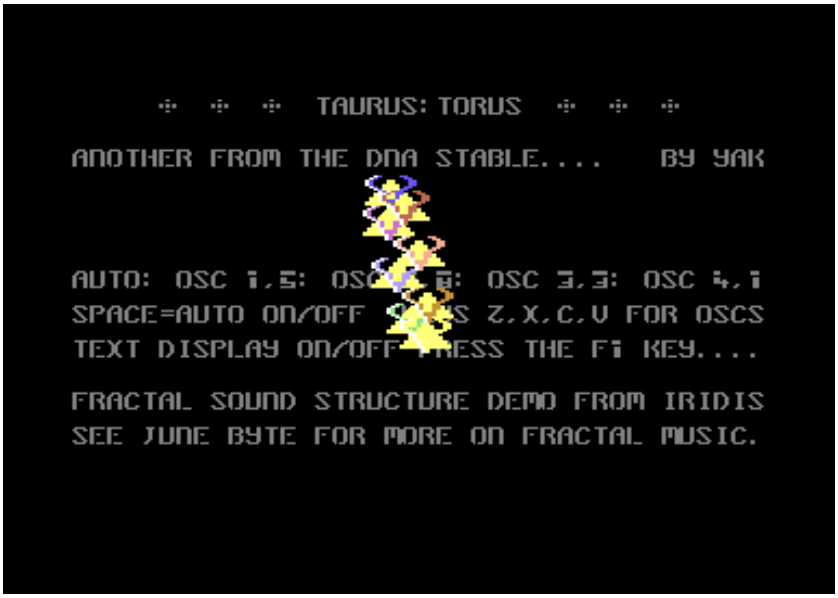


Figure 8.2

This first demo, released in July 1986(?), has a version of Iridis' music-generating algorithm that is nearly fully formed. However, the music it produces is quite different. In fact, it is nearer to a tool for listening to and selecting music than anything else.

The four seed values in `titleMusicNoteArray` that are used to seed all subequently generated tunes (`00 07 0C 07` in Iridis Alpha) can be selected and changed by the user. They're called 'Oscillators' and each can be any value between 0 and 16, i.e. any of `0 1 2 3 4 5 6 7 8 9 A B C D E F`.

```
;-----------------------------------------
; PlayTitleScreenMusic
; (TORUS:TAURUS)
;-----------------------------------------
PlayTitleScreenMusic




        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1

        JSR SelectNewNotesToPlay

        ; Set up a new tune.
```

```
;-----------------------------------------
; PlayTitleScreenMusic
; (IRIDIS ALPHA)
;-----------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
        RTS

MaybeStartNewTune
        LDA previousBaseNoteDuration
        STA baseNoteDuration

        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1


        ; Set up a new tune.
```

```
        LDA #$C0
        STA numberOfNotesToPlayInTune

        LDX notesPlayedSinceLastKeyChange
        LDA titleMusicNoteArray,X
        STA offsetForNextVoice1Note

        ; We'll only select a new tune when we've reached the
        ; beginning of a new 16 bar structure.
        INX
        TXA
        AND #$03
        STA notesPlayedSinceLastKeyChange

MaybePlayVoice1
        DEC voice1NoteDuration
        BNE MaybePlayVoice2

        LDA #$30
        STA voice1NoteDuration

        LDX voice1IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice1Note
        TAY
        STY offsetForNextVoice2Note

        JSR PlayVoice1

        INX
        TXA
        AND #$03
        STA voice1IndexToMusicNoteArray

MaybePlayVoice2
        DEC voice2NoteDuration
        BNE MaybePlayVoice3

        LDA #$0C
        STA voice2NoteDuration
        LDX voice2IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice2Note

        ; Use this new value to change the key of the next four
        ; notes played by voice 3.
        STA offsetForNextVoice3Note

        TAY
        JSR PlayVoice2
        INX
        TXA
        AND #$03
        STA voice2IndexToMusicNoteArray

MaybePlayVoice3
        DEC voice3NoteDuration
        BNE ReturnFromTitleScreenMusic

        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in titleMusicNoteArray.
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice3Note
        TAY
        JSR PlayVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
```

```
        LDA #$C0
        STA numberOfNotesToPlayInTune

        ; This is what will eventually time us out of
        ; the title music and enter attract mode.
        INC f7PressedOrTimedOutToAttractMode

        LDX notesPlayedSinceLastKeyChange
        LDA titleMusicNoteArray,X
        STA offsetForNextVoice1Note

        ; We'll only select a new tune when we've reached
        ; the beginning of a new 16 bar structure.
        INX
        TXA
        AND #$03
        STA notesPlayedSinceLastKeyChange
        BNE MaybePlayVoice1

        JSR SelectNewNotesToPlay

MaybePlayVoice1
        DEC voice1NoteDuration
        BNE MaybePlayVoice2

        LDA #$30
        STA voice1NoteDuration

        LDX voice1IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice1Note
        TAY
        STY offsetForNextVoice2Note

        JSR PlayNoteVoice1

        INX
        TXA
        AND #$03
        STA voice1IndexToMusicNoteArray

MaybePlayVoice2
        DEC voice2NoteDuration
        BNE MaybePlayVoice3

        LDA #$0C
        STA voice2NoteDuration
        LDX voice2IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice2Note

        ; Use this new value to change the key of the next
        ; notes played by voice 3.
        STA offsetForNextVoice3Note

        TAY
        JSR PlayNoteVoice2
        INX
        TXA
        AND #$03
        STA voice2IndexToMusicNoteArray

MaybePlayVoice3
        DEC voice3NoteDuration
        BNE ReturnFromTitleScreenMusic

        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice3Note
        TAY
        JSR PlayNoteVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
```

```
        ; Since it's only 4 bytes long ensure we wrap
        ; back to 0 if it's greater than 3.
        AND #$03
        STA voice3IndexToMusicNoteArray

ReturnFromTitleScreenMusic
        RTS
```

```
        ; Since it's only 4 bytes long ensure we wrap
        ; back to 0 if it's greater than 3.
        AND #$03
        STA voice3IndexToMusicNoteArray

ReturnFromTitleScreenMusic
        RTS
```

Listing 8.1: The music routine in Torus:Taurus side-by-side with Iridis Alpha.

When it runs the demo cycles through procedural configurations of `titleMusicNoteArray` of 64 notes each. In other words, exactly the kind of fractal structure we observed in Iridis Alpha proper. The examples below give a flavour of the music it generats:

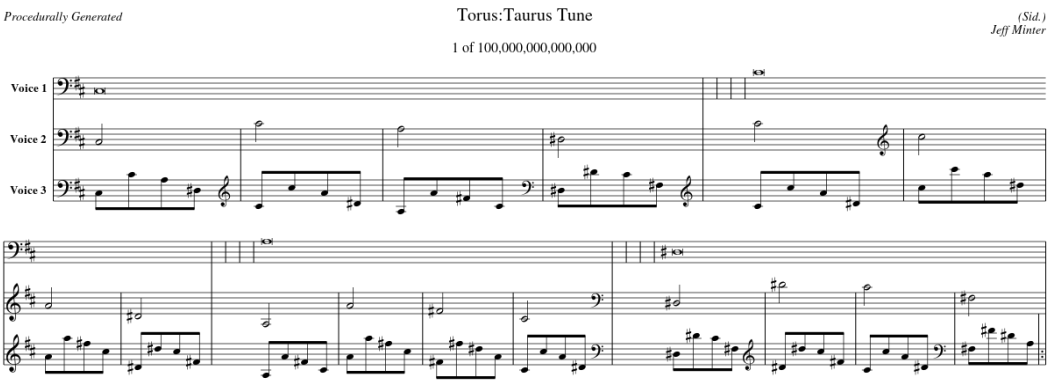


Figure 8.3: Bars 2 and 4 are always repeated

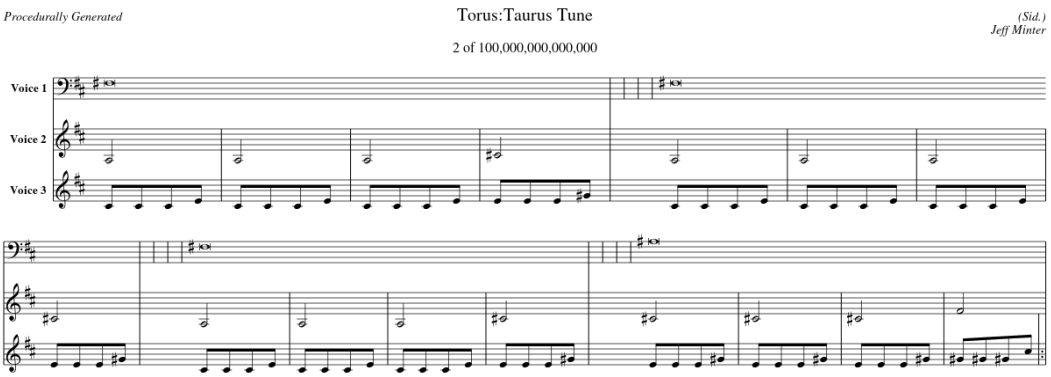


Figure 8.4: Bars 2 and 4 are always repeated

Not all of the tunes are 64-note based. It does generate some that are truncated.

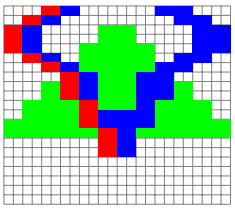


Figure 8.5: BULLHEAD

Figure 8.6: The 'Torus' sprite.

8.0.2 Taurus/Torus Two

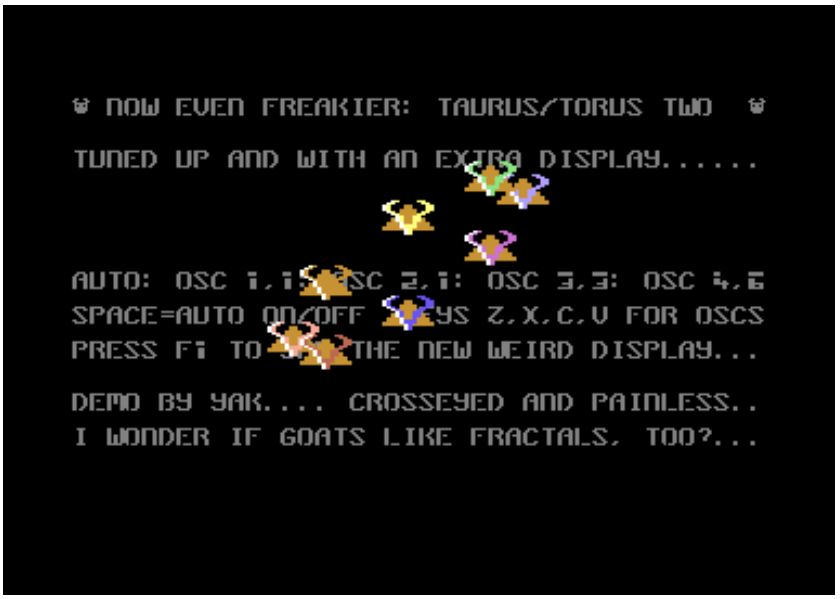


Figure 8.7

```
;-------------------------------------------
; PlayTitleScreenMusic
; (TORUS:TAURUS II)
;-------------------------------------------
PlayTitleScreenMusic
        LDA UnusedValue1
        STA UnusedValue2
```

```
;-------------------------------------------
; PlayTitleScreenMusic
; (IRIDIS ALPHA)
;-------------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
```

```
MaybeStartNewTune
        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1

        ; Set up a new tune.
        LDA #$C0
        STA numberOfNotesToPlayInTune


       playing
        LDX notesPlayedSinceLastKeyChange
        LDA titleMusicNoteArray,X
        STA offsetForNextVoice2Note

        ; We'll only select a new tune when we've reached the
       reached the
        ; beginning of a new 16 bar structure.
        INX
        TXA
        AND #$03
        STA notesPlayedSinceLastKeyChange
        BNE MaybePlayVoice1

        JSR SelectNewNotesToPlay

MaybePlayVoice1
        DEC voice1NoteDuration
        BNE MaybePlayVoice2

        LDA #$30
        STA voice1NoteDuration

        LDX voice1IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice2Note
        TAY
        STY offsetForNextVoice3Note

        JSR PlayNoteVoice1

        INX
        TXA
        AND #$03
        STA voice1IndexToMusicNoteArray

MaybePlayVoice2
        DEC voice2NoteDuration
        BNE MaybePlayVoice3

        LDA #$0C
        STA voice2NoteDuration
        LDX voice2IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice3Note

        ; Use this new value to change the key of the next four
       next four
        ; notes played by voice 1.
        STA offsetForNextVoice1Note

        TAY
        JSR PlayNoteVoice2
        INX
        TXA
        AND #$03
        STA voice2IndexToMusicNoteArray


        TAY
        JSR PlayNoteVoice2
        INX
        TXA
        AND #$03
        STA voice2IndexToMusicNoteArray

MaybePlayVoice3
        DEC voice3NoteDuration
        RTS
MaybeStartNewTune
        LDA previousBaseNoteDuration
        STA baseNoteDuration

        DEC numberOfNotesToPlayInTune
        BNE MaybePlayVoice1

        ; Set up a new tune.
        LDA #$C0 ; 193
        STA numberOfNotesToPlayInTune

        ; This is what will eventually time us out of
        ; the title music and enter attract mode.
        INC f7PressedOrTimedOutToAttractMode

        LDX notesPlayedSinceLastKeyChange
        LDA titleMusicNoteArray,X
        STA offsetForNextVoice1Note

        ; We'll only select a new tune when we've

        ; beginning of a new 16 bar structure.
        INX
        TXA
        AND #$03
        STA notesPlayedSinceLastKeyChange
        BNE MaybePlayVoice1

        JSR SelectNewNotesToPlay

MaybePlayVoice1
        DEC voice1NoteDuration
        BNE MaybePlayVoice2

        LDA #$30
        STA voice1NoteDuration

        LDX voice1IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice1Note
        TAY
        STY offsetForNextVoice2Note

        JSR PlayNoteVoice1

        INX
        TXA
        AND #$03
        STA voice1IndexToMusicNoteArray

MaybePlayVoice2
        DEC voice2NoteDuration
        BNE MaybePlayVoice3

        LDA #$0C
        STA voice2NoteDuration
        LDX voice2IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice2Note

        ; Use this new value to change the key of the
        ; notes played by voice 3.
        STA offsetForNextVoice3Note

MaybePlayVoice3
        DEC voice3NoteDuration
```

```
        BNE ReturnFromTitleMusic

        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in titleMusicNoteArray.
        titleMusicNoteArray.
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice1Note
        TAY
        JSR PlayNoteVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
        ; Since it's only 4 bytes long ensure we wrap
        ; back to 0 if it's greater than 3.
        AND #$03
        STA voice3IndexToMusicNoteArray

ReturnFromTitleMusic
        RTS
```

```
        BNE ReturnFromTitleScreenMusic

        LDA #$03
        STA voice3NoteDuration

        ; Play the note currently pointed to by
        ; voice3IndexToMusicNoteArray in

        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
        CLC
        ADC offsetForNextVoice3Note
        TAY
        JSR PlayNoteVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
        ; Since it's only 4 bytes long ensure we wrap
        ; back to 0 if it's greater than 3.
        AND #$03
        STA voice3IndexToMusicNoteArray

ReturnFromTitleScreenMusic
        RTS
```

Listing 8.2: The music routine in Taurus:Torus II side-by-side with Iridis Alpha.

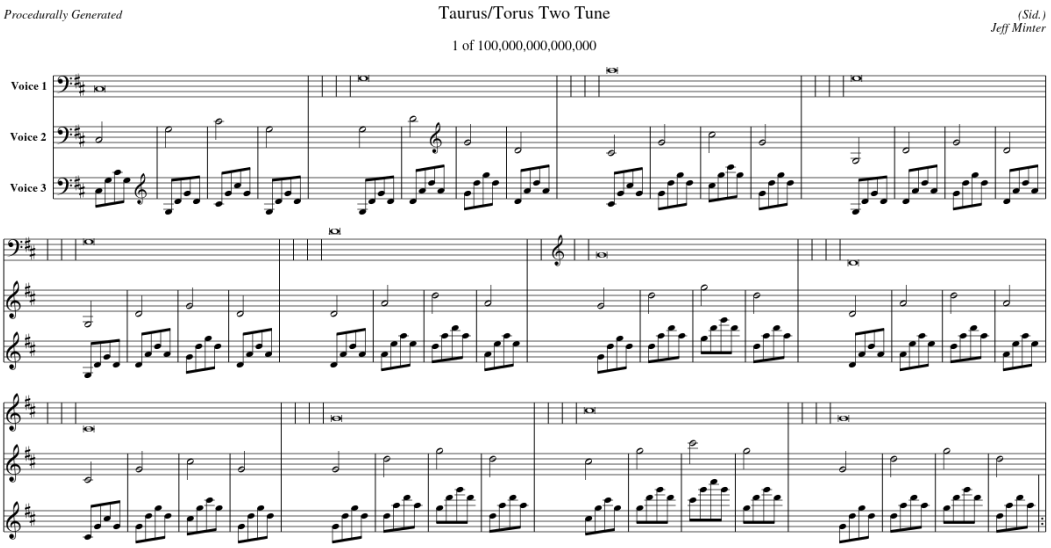


Figure 8.8

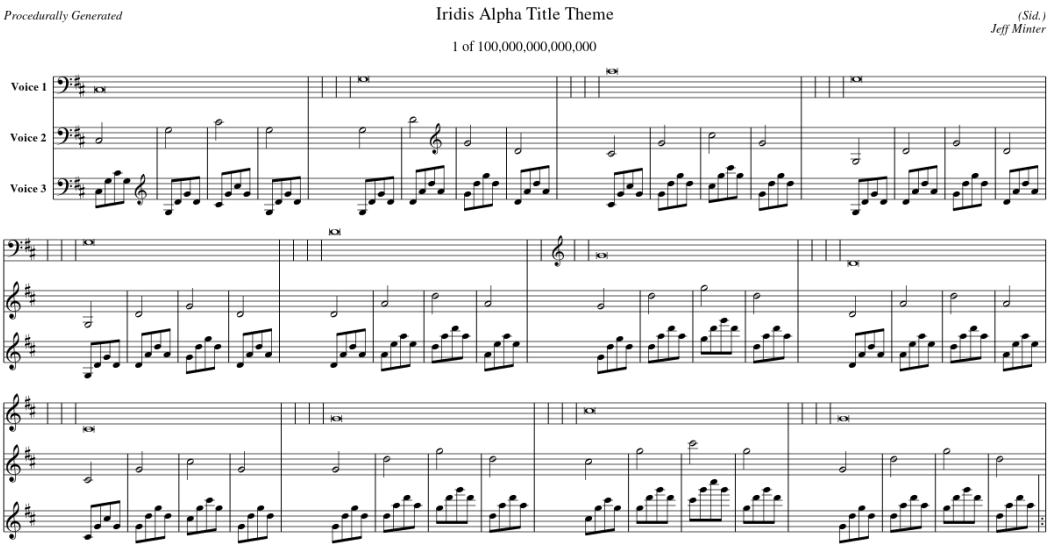

Procedurally Generated
Iridis Alpha Title Theme
1 of 100,000,000,000,000
(Sid.)
Jeff Minter
Voice 1
Voice 2
Voice 3

Figure 8.9

An Oscillator in 4 Parts

Figure 9.1: The Torus oscillator animation and Iridis' bonus animation.

The Torus demo is also the laboratory where the elegant animation used when awarding a bonus was developed. The code handling each is identical and was only very lightly modified for the final game.

```
RunMainInterruptHandler
        LDY #$00
        LDA #$F0
        STA $D012     ;Raster Position
        DEC counterBetweeXPosUpdates
        BNE MaybeUpdateYPos

UpdateXPos
        LDA initialCounterBetweenXPosUpdates
        STA counterBetweeXPosUpdates

        LDA incrementForXPos
        CLC
        ADC indexForXPosInSpritePositionArray
        STA indexForXPosInSpritePositionArray

MaybeUpdateYPos
        DEC counterBetweenYPosUpdates
        BNE MaybeUpdateXPosOffset

        LDA initialCounterBetweenYPosUpdates
        STA counterBetweenYPosUpdates

        LDA indexForYPosInSpritePositionArray
        CLC
        ADC incrementForYPos
        STA indexForYPosInSpritePositionArray

MaybeUpdateXPosOffset
        DEC cyclesBetweenXPosOffsetUpdates
        BNE MaybeUpdateYPosOffset

        LDA oscillator3Value
        STA cyclesBetweenXPosOffsetUpdates
        INC indexForXPosOffetsetInSpritePositionArray

MaybeUpdateYPosOffset
        DEC cyclesBetweenYPosOffsetUpdates
        BNE StoreInitialIndexValues

        LDA oscillator4Value
        STA cyclesBetweenYPosOffsetUpdates
        INC indexForYPosOffsetInSpritePositionArray

StoreInitialIndexValues
        ; Store the initial values for our indices
        ; on the stack.
        LDA indexForXPosInSpritePositionArray
        PHA
        LDA indexForYPosInSpritePositionArray
        PHA
        LDA indexForXPosOffetsetInSpritePositionArray
```

Listing 9.1: Animation in Torus Demo

```
DisplayNewBonus
        SEI
        INC bonusBountiesEarned
        LDA bonusBountiesEarned
        AND #$07
        STA bonusBountiesEarned
        LDA #$00
        STA bonusAwarded
        STA $D010     ;Sprites 0-7 MSB of X coordinate
        LDA #<NewBonusGilbyAnimation
        STA $0314     ;IRQ
        LDA #>NewBonusGilbyAnimation
        STA $0315     ;IRQ
        JSR ClearScreen3
        LDA $D011     ;VIC Control Register 1
        AND #$7F
        STA $D011     ;VIC Control Register 1
        LDA #$F0
        STA $D012     ;Raster Position
        LDA #BLACK
        STA $D020     ;Border Color
        STA $D021     ;Background Color 0
        CLI

        LDA #$FF
        STA $D015     ;Sprite display Enable
        STA $D01C     ;Sprites Multi-Color Mode Select
        LDX #$07

        ; Display the animated gilbies
b6595   LDA #$C1
        STA Sprite0Ptr,X
        LDA newBonusGilbyColors,X
        STA $D027,X   ;Sprite 0 Color
        DEX
        BPL b6595

        LDX #$0A
b65A5   LDA txtBonus10000,X
        AND #$3F
        STA SCREEN_RAM + LINE10_COL15,X
        LDA #YELLOW
        STA COLOR_RAM + LINE10_COL15,X
        DEX
        BPL b65A5

        ; Increment the total bonus bounty by 10000
        LDX #$03
b65B7   INC currentBonusBounty,X
        LDA currentBonusBounty,X
        CMP #$3A
        BNE b65C9
```

Listing 9.2: ... and Iridis Alpha

A Testing Hack

In the CheckKeyboardInGame Routine, we find the following:

```
;------------------------------------------------------------
; ResetUpperPlanetBullet
;------------------------------------------------------------
ResetUpperPlanetBullet
        LDA #$F0
        STA upperPlanetGilbyBulletSpriteValue,X
        LDA #$FF
        STA upperPlanetGilbyBulletXPos,X
        LDA #$00
        STA upperPlanetGilbyBulletYPos,X
ReturnEarlyFromBullet
        RTS
```

In the above the ‘canAwardBonus‘ byte is the first letter in the name of the player

with the top score in the Hi-Score table. By default this is 'YAK':

```
;--------------------------------------------------------------
DetectGameOrAttractMode
        LDA attractModeSelected
        BNE b7EB8
```

But if we change 'Y' to `$1C` like so, we can activate the hack:

```
hiScoreTablePtr             .TEXT "0068000"
canAwardBonus               .TEXT $1C,"AK "
```

Note that `$1C` is charset code for a bull's head symbol in Iridis Alpha, so it is also possible to enter this as the initial of a high scorer name if we get a score that puts us to the top of the table:

```
        .BYTE $66,$C3,$7E,$5A,$7E,$7E,$3C,$00   ;.BYTE $66,$C3,$7E,$5A,$7E,$7E,$3C,$00
                                                ; CHARACTER $1c
                                                ; 01100110    **  **
                                                ; 11000011   **    **
                                                ; 01111110    ******
                                                ; 01011010    * ** *
                                                ; 01111110    ******
                                                ; 01111110    ******
                                                ; 00111100     ****
                                                ; 00000000
```

I'm guessing this was used for testing the animation routine and left in as an Easter egg.

To start getting a handle on how the oscillation animation works, lets plot the first 24 animations that the Torus demo uses when left to its own devices. We get a variety of different trajectories, some relatively simple, some quite convoluted.

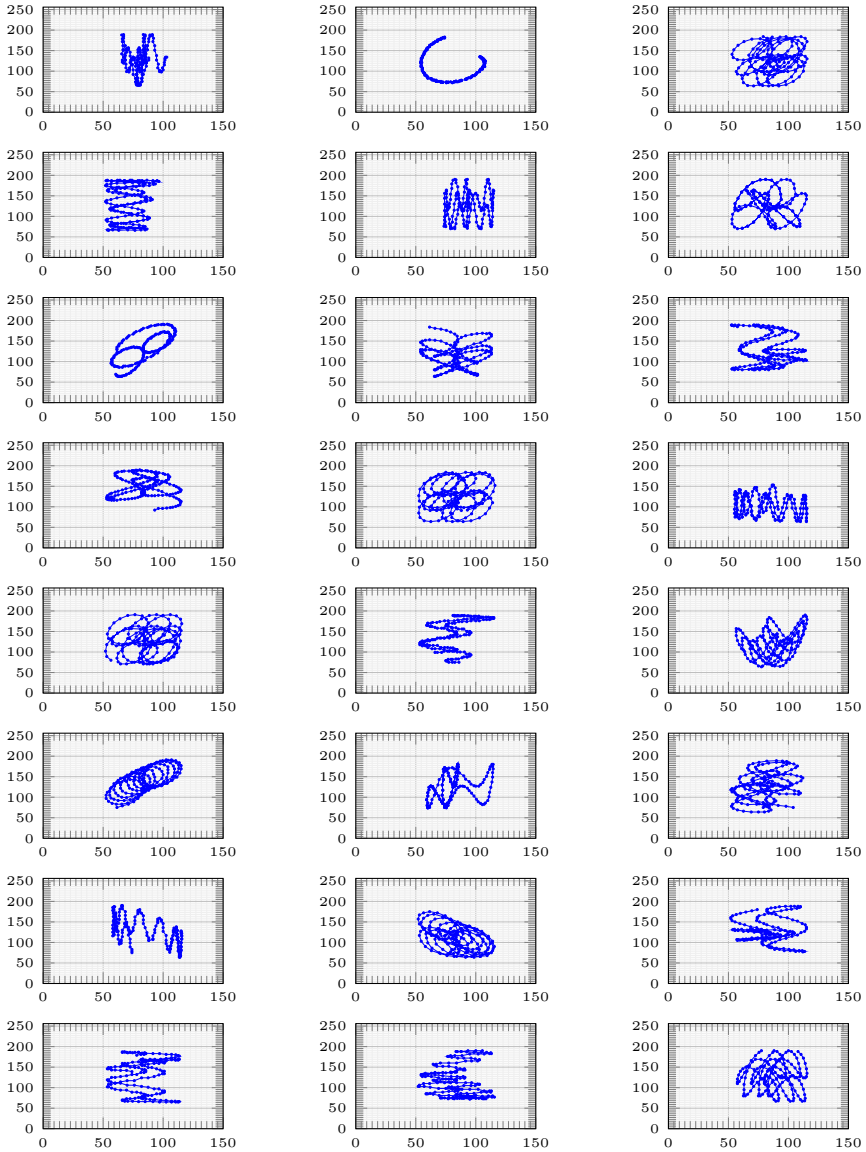


Figure 9.2: The first 24 oscillation patterns generated by the Torus demo.

When we look into the code we find this petting zoo of animations is principally driven by a simple sequence of bytes stored in `spritePositionArray`.

```
spritePositionArray   .BYTE $40,$46,$4C,$52,$58,$5E,$63,$68
                      .BYTE $6D,$71,$75,$78,$7B,$7D,$7E,$7F
                      .BYTE $80,$7F,$7E,$7D,$7B,$78,$75,$71
                      .BYTE $6D,$68,$63,$5E,$58,$52,$4C,$46
                      .BYTE $40,$39,$33,$2D,$27,$21,$1C,$17
                      .BYTE $12,$0E,$0A,$07,$04,$02,$01,$00
                      .BYTE $00,$00,$01,$02,$04,$07,$0A,$0E
                      .BYTE $12,$17,$1C,$21,$27,$2D,$33,$39
                      .BYTE $FF
```

We can get a sense of how this rising and falling sequence of values can be used to plot a course across the screen if we treat each as an x and y value on a graph of cartesian co-ordinates. In the twenty four instances below we start by treating the value as providing both the x and y position. In each subsequent one we skip an increasing number of positions ahead in the sequence to get the y value, producing a variety of elliptical orbits around the screen.

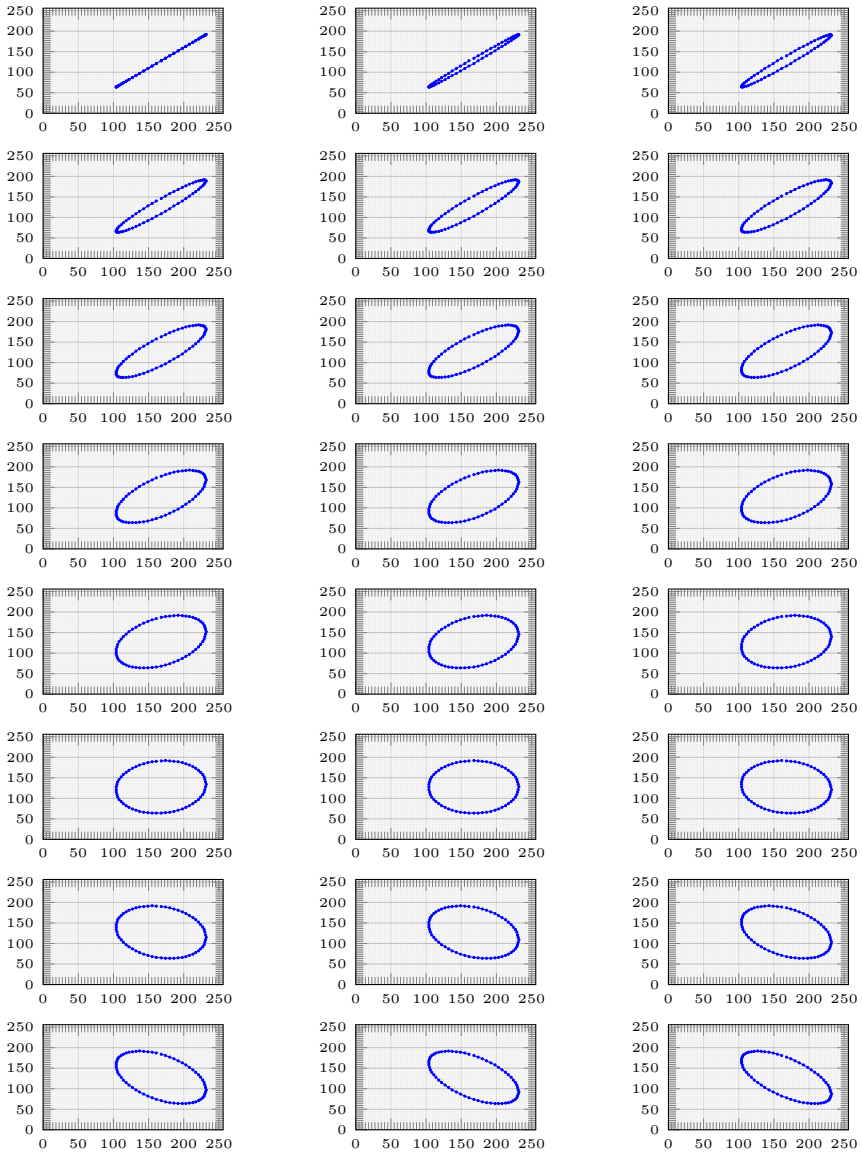


Figure 9.3: Using the x/y offset in `spritePositionArray` where y is the value after x in the array.

To get beyond simple ellipsoids we need to do more than pick a different value in the array for our x and y offsets. Here we experiment with something a little more involved. We update the x and y positions at different intervals and when skipping ahead in `spritePositionArray` for a new value for x and y we use a pre-selected, random number of bytes to skip past.

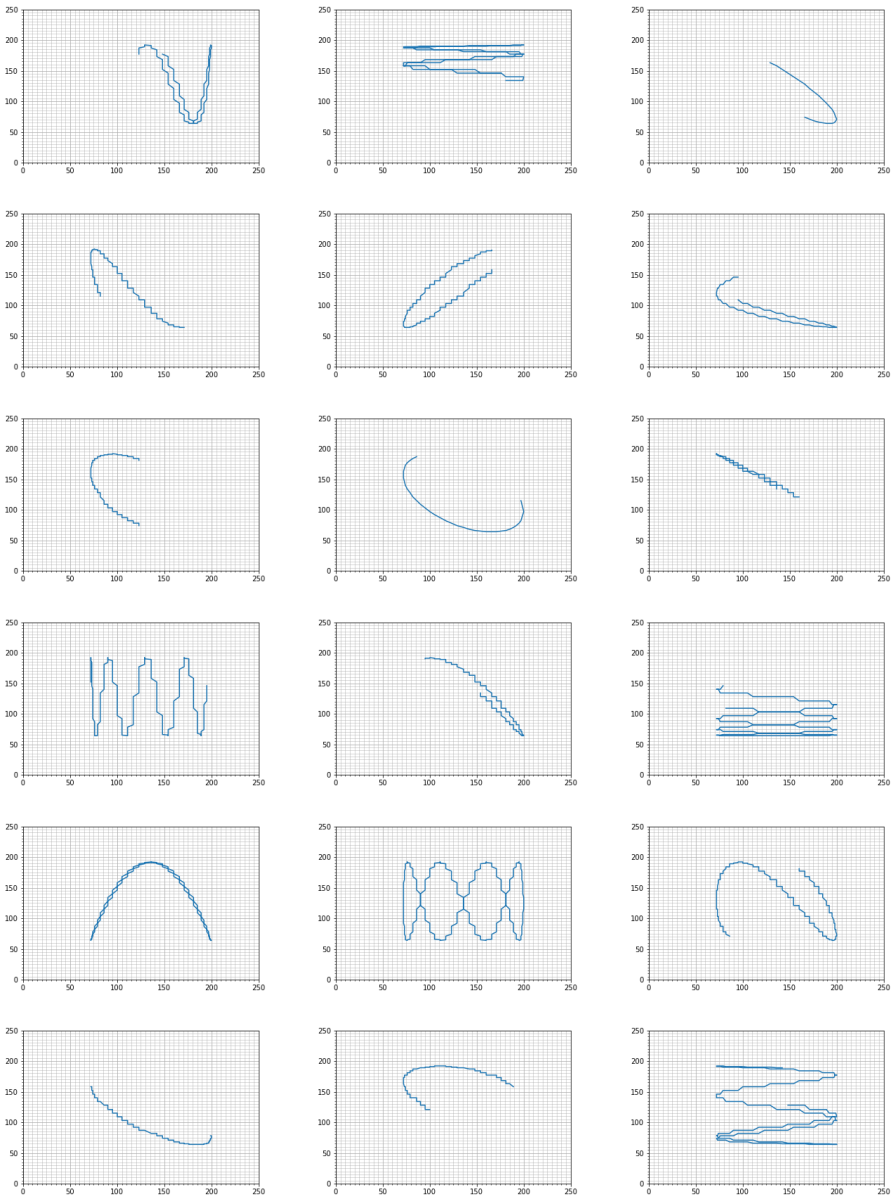


Figure 9.4: Testing different values of x and y

This is starting to look more like the actual results we observed and it is where the 4 values selectable by the player using keys z, x, c, and v in the Torus demo come in. In addition to controlling the music generation procedure, as we've already seen, they also determine the way the values in `spritePositionArray` are selected for position the sprite in each new frame. This is based on letting them determine the frequency with which the position of the x and y values of each sprite is changed and how far to skip ahead in `spritePositionArray` when selecting a new value from it for the x and y position.

Key	Name	Purpose	Code

Z	Oscillator 1	- Intervals between updating X position.
		- The amount to increment the index into `spritePositionArray`
		when getting the next X position.
			`MaybeZKeyPressed`
			`CMP #$0C`
			`BNE MaybeXKeyPressed`
			
			`; Update Oscillator 1`
			`LDA oscillator1Value`
			`CLC`
			`ADC #$01`
			`AND #$0F`
			`STA oscillator1Value`
			
			`TAX`
			`LDA intervalBetweenPosUpdatesArray,X`
			`STA initialCounterBetweenXPosUpdates`
			`LDA positionIncrementArray,X`
			`STA incrementForXPos`
			`JMP ContinueCheckingForKeyPress`

X	Oscillator 2	- Intervals between updating Y position.
		- The amount to increment the index into `spritePositionArray`
		when getting the next Y position.
			`MaybeXKeyPressed`
			`CMP #$17`
			`BNE MaybeCKeyPressed`
			
			`; Update Oscillator 2`
			`LDA oscillator2Value`
			`CLC`
			`ADC #$01`
			`AND #$0F`
			`STA oscillator2Value`
			`TAX`
			`LDA intervalBetweenPosUpdatesArray,X`
			`STA initialCounterBetweenYPosUpdates`
			`LDA positionIncrementArray,X`
			`STA incrementForYPos`
			`JMP ContinueCheckingForKeyPress`

C	Oscillator 3	- How often to increase the index that seeks
		ahead to get a value
		from `spritePositionArray`for adding
		to the next X position.
			`MaybeCKeyPressed`
			`CMP #$14`
			`BNE MaybeVKeyPressed`
			
			`; Update Oscillator 3`
			`LDA oscillator3Value`
			`CLC`
			`ADC #$01`
			`AND #$0F`
			`STA oscillator3Value`
			`JMP ContinueCheckingForKeyPress`

V	Oscillator 4	- How often to increase the index that seeks
		ahead to get a value
		from `spritePositionArray`for adding
		to the next Y position.
			`MaybeVKeyPressed`
			`CMP #$1F`
			`BNE MaybeF1Pressed`
			
			`; Update Oscillator 4`
			`LDA oscillator4Value`
			`CLC`
			`ADC #$01`
			`AND #$0F`
			`STA oscillator4Value`

Figure 9.5: The purpose of each of the oscillator values.

In `RunMainInterruptHandler`) we can see how each of these values set by the player is used to maintain an accounting of the different sprite positions for each of the 8 sprites:

```
RunMainInterruptHandler
        LDY #$00
        LDA #$F0
        STA $D012     ;Raster Position
        DEC counterBetweeXPosUpdates
        BNE MaybeUpdateYPos

UpdateXPos
        LDA initialCounterBetweenXPosUpdates
        STA counterBetweeXPosUpdates

        LDA incrementForXPos
        CLC
        ADC indexForXPosInSpritePositionArray
        STA indexForXPosInSpritePositionArray

MaybeUpdateYPos
        DEC counterBetweenYPosUpdates
        BNE MaybeUpdateXPosOffset

        LDA initialCounterBetweenYPosUpdates
        STA counterBetweenYPosUpdates

        LDA indexForYPosInSpritePositionArray
        CLC
        ADC incrementForYPos
        STA indexForYPosInSpritePositionArray

MaybeUpdateXPosOffset
        DEC cyclesBetweenXPosOffsetUpdates
        BNE MaybeUpdateYPosOffset

        LDA oscillator3Value
        STA cyclesBetweenXPosOffsetUpdates
        INC indexForXPosOffetsetInSpritePositionArray

MaybeUpdateYPosOffset
        DEC cyclesBetweenYPosOffsetUpdates
        BNE StoreInitialIndexValues

        LDA oscillator4Value
        STA cyclesBetweenYPosOffsetUpdates
        INC indexForYPosOffsetInSpritePositionArray
```

Before animating each of the 8 sprites we use the values set by the player to prepare the variables that will be applied to positioning each sprite. For example the value selected with the Z key has been used to set `initialCounterBetweenXPosUpdates` and `incrementForXPos`. In the first lines above in `UpdateXPos` we use them to set up `indexForXPosInSpritePositionArray`. THis is then used in `SpriteAnimationLoop` to selec the X position of the current sprite:

```
SpriteAnimationLoop
        LDA indexForXPosInSpritePositionArray
        AND #$3F
        TAX
        LDA spritePositionArray,X
        STA currSpriteXPos
```

You can follow the same lineage between the setting of each value in our table above with the rest of the `SpriteAnimationLoop` routine.

Appendix: Enemy Data

Sprite Data for Each Level

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$06	FLYING_SAUCER1	$03	FLYING_SAUCER1
2	$06	BOUNCY_RING	$01	BOUNCY_RING
3	$05	FLYING_DOT1	$04	FLYING_DOT1
4	$11	FLYING_TRIANGLE1	$03	FLYING_TRIANGLE1
5	$11	BALLOON	$00	BIRD1
6	$0A	BIRD1	$03	BIRD1
7	$09	FLAG_BAR	$00	FLAG_BAR
8	$11	TEARDROP_EXPLOSION1	$03	TEARDROP_EXPLOSION1
9	$06	WINGBALL	$03	MONEY_BAG
10	$08	CAMEL	$00	INV_MAGIC_MUSHROOM
11	$0E	GILBY_AIRBORNE_LEFT	$06	GILBY_AIRBORNE_LOWERPLANET_RIGHT
12	$09	CAMEL	$02	LICKERSHIP_INV1
13	$0B	BUBBLE	$04	BUBBLE
14	$06	TEARDROP_EXPLOSION1	$05	TEARDROP_EXPLOSION1
15	$08	LLAMA	$00	LLAMA
16	$05	QBERT_SQUARES	$00	QBERT_SQUARES
17	$0A	BOUNCY_RING	$02	BOUNCY_RING
18	$05	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
19	$04	STARSHIP	$00	STARSHIP
20	$07	COPTIC_CROSS	$00	COPTIC_CROSS

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 1 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$55	LITTLE_DART	$01	LITTLE_DART
2	$04	FLYING_COCK1	$05	FLYING_COCK1
3	$06	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
4	$05	TEARDROP_EXPLOSION1	$01	TEARDROP_EXPLOSION1
5	$05	LICKER_SHIP1	$00	LICKERSHIP_INV1
6	$04	SPINNING_RING1	$00	SPINNING_RING1
7	$0F	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
8	$0C	BUBBLE	$04	BUBBLE
9	$04	LAND_GILBY1	$03	LAND_GILBY_LOWERPLANET1
10	$11	FLYING_TRIANGLE1	$00	FLYING_TRIANGLE1
11	$00	FLYING_SAUCER1	$01	FLYING_SAUCER1
12	$0C	BUBBLE	$01	BUBBLE
13	$08	LLAMA	$00	LLAMA
14	$04	FLYING_COCK1	$05	FLYING_COCK1
15	$08	FLAG_BAR	$00	FLAG_BAR
16	$10	WINGBALL	$04	MONEY_BAG
17	$06	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
18	$10	BOLAS1	$02	BOLAS1
19	$0E	LAND_GILBY1	$04	LAND_GILBY_LOWERPLANET1
20	$06	EYE_OF_HORUS	$00	EYE_OF_HORUS

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 2 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$10	$FC	$02	$FC
2	$0D	LITTLE_DART	$00	LITTLE_DART
3	$02	BOUNCY_RING	$04	BOUNCY_RING
4	$06	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
5	$0B	SMALL_BALL1	$02	SMALL_BALL1
6	$00	LAND_GILBY1	$01	LAND_GILBY_LOWERPLANET1
7	$07	LICKER_SHIP1	$07	LICKERSHIP_INV1
8	$0C	BUBBLE	$03	BUBBLE
9	$06	FLYING_DART1	$05	FLYING_DART1
10	$06	FLYING_SAUCER1	$03	FLYING_SAUCER1
11	$04	LICKER_SHIP1	$05	LICKERSHIP_INV1
12	$00	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
13	$06	LICKER_SHIP1	$05	LICKERSHIP_INV1
14	$08	CAMEL	$00	CAMEL
15	$10	BOUNCY_RING	$01	BOUNCY_RING
16	$10	STRANGE_SYMBOL	$00	STRANGE_SYMBOL
17	$08	LLAMA	$00	LLAMA
18	$06	FLYING_SAUCER1	$01	FLYING_SAUCER1
19	$0E	FLYING_COMMA1	$04	FLYING_COMMA1
20	$11	PSI	$00	PSI

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 3 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$04	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
2	$0E	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
3	$02	LITTLE_DART	$03	LITTLE_DART
4	$03	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
5	$09	LOZENGE	$00	LOZENGE
6	$06	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
7	$05	TEARDROP_EXPLOSION1	$06	TEARDROP_EXPLOSION1
8	$00	LLAMA	$00	LLAMA
9	$11	BUBBLE	$04	BUBBLE
10	$11	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
11	$0E	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
12	$00	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
13	$0D	FLYING_DOT1	$03	FLYING_DOT1
14	$11	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
15	$10	BOLAS1	$02	BOLAS1
16	$07	CAMEL	$00	CAMEL
17	$00	CUMMING_COCK1	$06	BOLAS1
18	$10	CUMMING_COCK1	$05	LICKERSHIP_INV1
19	$06	QBERT_SQUARES	$00	QBERT
20	$10	BULLHEAD	$00	BULLHEAD

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 4 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$11	STARSHIP	$00	STARSHIP
2	$11	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
3	$02	LAND_GILBY1	$04	LAND_GILBY_LOWERPLANET1
4	$0E	TEARDROP_EXPLOSION1	$04	TEARDROP_EXPLOSION1
5	$04	FLYING_COMMA1	$05	FLYING_COMMA1
6	$0B	STARSHIP	$00	STARSHIP
7	$10	FLYING_FLOWCHART1	$01	FLYING_FLOWCHART1
8	$10	BALLOON	$01	BOUNCY_RING
9	$00	BOUNCY_RING	$03	BOUNCY_RING
10	$08	CAMEL	$00	CAMEL
11	$06	BIRD1	$04	BIRD1
12	$07	BALLOON	$03	LAND_GILBY_LOWERPLANET8
13	$11	BUBBLE	$01	BUBBLE
14	$08	CAMEL	$00	CAMEL
15	$10	BOUNCY_RING	$04	BOUNCY_RING
17	$10	BUBBLE	$02	BUBBLE
18	$10	LITTLE_OTHER_EYEBALL	$01	SMALL_BALL1
20	$02	ATARI_ST	$00	ATARI_ST

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 5 - Sprite Data.

10.0.1 Enemy Pointer Data

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet1Level1Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	planet1Level2Data	spinningRings	planet1Level2Data
3	nullPtr	planet1Level3Data2ndStage	nullPtr	nullPtr	lickerShipWaveData	lickerShipWaveData
4	nullPtr	planet1Level4Data2ndStage	nullPtr	nullPtr	planet1Level4Data2ndStage	planet1Level4Data2ndStage
5	nullPtr	nullPtr	nullPtr	planet1Level5Data2ndStage	planet1Level5Data3rdStage	defaultExplosion
6	nullPtr	planet1Level6Data2ndStage	nullPtr	nullPtr	spinningRings2ndType	defaultExplosion
7	nullPtr	planet1Level7Data2ndStage	nullPtr	nullPtr	planet1Level7Data2ndStage	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level8Data2ndStage	planet1Level8Data2ndStage
9	nullPtr	planet1Level9DataSecondStage	planet1Level9DataSecondStage	nullPtr	defaultExplosion	defaultExplosion
10	nullPtr	planet1Level10Data2ndStage	nullPtr	planet1Level10Data	planet1Level10Data2ndStage	defaultExplosion
11	smallDotWaveData	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
12	nullPtr	nullPtr	nullPtr	planet1Level12Data	planet1Level2Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	planet1Level13Data	planet1Level13Data2ndStage	planet1Level13Data2ndStage
14	planet1Level8Data	nullPtr	nullPtr	nullPtr	planet1Level8Data	planet1Level8Data
15	nullPtr	planet1Level15Data	nullPtr	nullPtr	teardropExplosion	lickerShipWaveData
16	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level19Data	defaultExplosion
17	nullPtr	planet1Level17Data2ndStage	nullPtr	nullPtr	gilbyLookingLeft	defaultExplosion
18	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level18Data2ndStage	defaultExplosion
19	nullPtr	planet1Level19Data	nullPtr	nullPtr	planet5Level6Data	planet5Level6Data
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet1Level20Data	planet1Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 1 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet2Level1Data	nullPtr	nullPtr	pinAsExplosion	defaultExplosion
2	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	lickerShipWaveData
3	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	lickerShipWaveData
4	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
5	nullPtr	nullPtr	nullPtr	planet2Level5Data2ndStage	planet2Level5Data3rdStage	planet2Level5Data2ndStage
6	nullPtr	nullPtr	planet2Level6Data2ndStage	nullPtr	secondExplosionAnimation	lickerShipWaveData
7	nullPtr	planet2Level7Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
8	nullPtr	planet2Level8Data2ndStage	nullPtr	nullPtr	planet2Level8Data2ndStage	defaultExplosion
9	nullPtr	nullPtr	nullPtr	planet2Level9Data	gilbyTakingOffAsExplosion	defaultExplosion
10	nullPtr	nullPtr	nullPtr	nullPtr	flowchartArrowAsExplosion	defaultExplosion
11	nullPtr	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level11Data2ndStage
12	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level1Data	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level13Data2ndStage	defaultExplosion
14	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level14Data2ndStage	lickerShipWaveData
15	nullPtr	nullPtr	nullPtr	planet2Level15Data	planet2Level15Data2ndStage	defaultExplosion
16	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level9Data	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level17Data2ndStage	lickerShipWaveData
18	nullPtr	planet2Level18Data2ndStage	nullPtr	nullPtr	defaultExplosion	defaultExplosion
19	landGilbyAsEnemy	nullPtr	nullPtr	planet2Level19Data	planet2Level19Data2ndStage	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet2Level20Data	planet2Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 2 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet3Level1Data	nullPtr	nullPtr	$50	defaultExplosion
2	nullPtr	planet3Level2Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
3	nullPtr	nullPtr	nullPtr	planet3Level3Data2ndStage	secondExplosionAnimation	defaultExplosion
4	gilbyLookingLeft	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
5	nullPtr	nullPtr	nullPtr	nullPtr	stickyGlobeExplosion	planet3Level5Data
6	planet3Level6Additional	nullPtr	nullPtr	planet3Level6Data	planet2Level9Data	defaultExplosion
7	nullPtr	nullPtr	planet3Level7Data2ndStage	nullPtr	spinningRings	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	bubbleExplosion	defaultExplosion
9	nullPtr	planet3Level9Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
10	nullPtr	planet3Level10Data2ndStage	nullPtr	nullPtr	spinningRings	planet3Level10Data
11	nullPtr	nullPtr	nullPtr	planet3Level11Data	planet3Level11Data2ndStage	defaultExplosion
12	nullPtr	nullPtr	nullPtr	planet3Level12Data	planet3Level12Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	lickerShipAsExplosion	defaultExplosion
14	nullPtr	planet1Level12Data	nullPtr	nullPtr	planet3Level14Data2ndStage	defaultExplosion
15	nullPtr	nullPtr	nullPtr	nullPtr	planet3Level15Data2ndStage	defaultExplosion
16	nullPtr	planet2Level5Data	nullPtr	nullPtr	planet3Level16Data	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level14Data	defaultExplosion
18	nullPtr	nullPtr	nullPtr	nullPtr	planet3Level18Data2ndStage	planet3Level18Data2ndStage
19	nullPtr	planet3Level19Data2ndStage	nullPtr	nullPtr	planet4Level17Data	planet4Level17Data
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet3Level20Data	planet3Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 3 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	nullPtr	nullPtr	planet4Level1Data2ndStage	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	nullPtr	planet4Leve2Data2ndStage	defaultExplosion
3	nullPtr	planet4Level2Data2ndStage	nullPtr	nullPtr	planet1Level5Data3rdStage	defaultExplosion
4	nullPtr	planet4Level4Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
5	nullPtr	nullPtr	nullPtr	planet4Level5Data2ndStage	secondExplosionAnimation	defaultExplosion
6	nullPtr	planet4Level6Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
7	nullPtr	planet1Level14Data	nullPtr	nullPtr	defaultExplosion	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level8Data	planet4Level8Data2ndStage
9	nullPtr	nullPtr	nullPtr	planet4Level9Data2ndStage	spinningRings	lickerShipWaveData
10	nullPtr	planet4Level10Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
11	nullPtr	planet4Level11Data2ndStage	nullPtr	nullPtr	planet4Level11Data2ndStage	planet4Level11Data2ndStage
12	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level12Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level5Data	planet5Level5Data
14	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level14Data2ndStage	defaultExplosion
15	nullPtr	nullPtr	nullPtr	nullPtr	spinnerAsExplosion	defaultExplosion
16	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level16Data2ndStage	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	cummingCock	defaultExplosion
18	nullPtr	nullPtr	nullPtr	planet4Level18Data	secondExplosionAnimation	defaultExplosion
19	planet4Level19Additional	nullPtr	nullPtr	planet4Level19Data	spinningRings	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet4Level20Data	planet4Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 4 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet5Level1Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	planet5Level2Data	planet5Level2Explosion	defaultExplosion
3	planet5Level3Additional	nullPtr	nullPtr	planet5Level3Data	planet5Level3Data2ndStage	lickerShipWaveData
4	nullPtr	planet5Level5Data2ndStage	nullPtr	nullPtr	spinningRings	lickerShipWaveData
5	planet5Level5Additional	planet5Level5Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
6	nullPtr	nullPtr	nullPtr	nullPtr	fighterShipAsExplosion	defaultExplosion
7	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level7Data2ndStage	defaultExplosion
8	nullPtr	nullPtr	nullPtr	planet5Level8Data	planet1Level5Data	defaultExplosion
9	nullPtr	planet5Level9Data2ndStage	nullPtr	nullPtr	planet5Level9Data2ndStage	defaultExplosion
10	nullPtr	nullPtr	defaultExplosion	nullPtr	lickerShipWaveData	lickerShipWaveData
11	nullPtr	planet5Level11Data	nullPtr	nullPtr	planet5Level11Data2ndStage	defaultExplosion
12	nullPtr	nullPtr	planet1Level5Data	planet5Level12Data	nullPtr	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level13Data2ndStage	defaultExplosion
14	llamaWaveData	nullPtr	nullPtr	nullPtr	spinningRings	lickerShipWaveData
15	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level15Data2ndStage	defaultExplosion
17	nullPtr	nullPtr	nullPtr	planet5Level17Data	planet3Level8Data	defaultExplosion
18	nullPtr	planet5Level18Data	nullPtr	nullPtr	planet1Level5Data3rdStage	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet5Level20Data	planet5Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 5 - Pointer Data.

10.0.2 Enemy Behaviour

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$40	$00	$00	$02	$02	$00	$04	$18
2	$00	$01	$23	$01	$01	$00	$04	$20
3	$30	$00	$00	$02	$01	$00	$04	$20
4	$60	$00	$01	$04	$02	$00	$04	$20
5	$00	$00	$23	$00	$05	$00	$04	$20
6	$03	$01	$01	$01	$04	$00	$04	$10
7	$50	$00	$01	$00	$03	$00	$04	$28
8	$00	$00	$01	$02	$02	$00	$04	$20
9	$0C	$00	$00	$00	$08	$00	$04	$20
10	$80	$00	$23	$00	$06	$00	$04	$18
11	$00	$00	$01	$04	$05	$00	$04	$10
12	$00	$00	$23	$03	$02	$00	$04	$20
13	$00	$01	$23	$04	$02	$00	$04	$20
14	$00	$00	$01	$00	$08	$00	$04	$10
15	$10	$01	$01	$03	$03	$00	$04	$20
16	$00	$01	$00	$00	$06	$00	$04	$18
17	$40	$00	$00	$05	$0C	$00	$04	$20
18	$00	$00	$01	$00	$03	$00	$04	$20
19	$20	$80	$01	$00	$04	$00	$04	$20
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 1 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$08	$01	$10	$01	$02	$00	$04	$18
2	$00	$00	$01	$02	$02	$00	$04	$18
3	$00	$00	$01	$02	$02	$00	$04	$18
4	$00	$00	$23	$06	$03	$00	$04	$18
5	$00	$00	$23	$02	$01	$00	$04	$30
6	$00	$00	$00	$01	$02	$00	$04	$20
7	$40	$00	$01	$03	$02	$00	$04	$10
8	$60	$00	$23	$00	$20	$00	$04	$10
9	$00	$00	$23	$00	$04	$00	$04	$10
10	$00	$00	$00	$00	$06	$00	$04	$18
11	$00	$10	$01	$00	$00	$00	$04	$10
12	$00	$00	$00	$00	CAMEL	$00	$04	$30
13	$00	$01	$01	$02	$01	$00	$04	$40
14	$00	$00	$01	$02	$02	$00	$04	$18
15	$00	$00	$23	$03	$02	$00	$04	$20
16	$00	$00	$00	$00	$10	$00	$04	$30
17	$00	$00	$01	$02	$02	$00	$04	$18
18	$08	$01	$01	$00	$06	$00	$04	$20
19	$00	$00	$23	$04	$02	$00	$04	$38
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 2 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$20	$01	$01	$01	$01	$53	$41	$56
2	$50	$00	$00	$02	$05	$00	$04	$18
3	$00	$00	$23	$02	$04	$00	$04	$10
4	$00	$00	$01	$04	$08	$00	$04	$18
5	$00	$00	$00	$02	$03	$00	$04	$20
6	$00	$00	$23	$03	$04	$00	$04	$10
7	$00	$00	$00	$01	$02	$00	$04	$20
8	$00	$00	$00	$00	$0C	$00	$04	$10
9	$0C	$01	$01	$03	$05	$00	$04	$18
10	$0A	$01	$01	$03	$03	$00	$04	$20
11	$00	$00	$23	$00	$08	$00	$04	$20
12	$00	$00	$23	$01	$02	$00	$04	$28
13	$00	$00	$00	$00	$05	$00	$04	$18
14	$F0	$00	$00	$00	$08	$00	$04	$20
15	$00	$00	$00	$03	$02	$00	$04	$28
16	$C0	$00	$00	$00	$10	$00	$04	$10
17	$00	$00	$01	$00	$0C	$00	$04	$30
18	$00	$01	$01	$04	$02	$00	$04	$20
19	$40	$00	$01	$00	$04	$00	$04	$20
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 3 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$00	$00	$23	$02	$02	$00	$04	$20
2	$00	$00	$01	$04	$01	$00	$04	$10
3	$60	$00	$02	$00	$03	$00	$04	$20
4	$40	$00	$23	$02	$04	$00	$04	$18
5	$00	$00	$23	$02	$04	$00	$04	$18
6	$20	$00	$00	$01	$04	$00	$04	$20
7	$E0	$00	$00	$00	$08	$00	$04	$08
8	$00	$00	$00	$00	$00	$00	$04	$20
9	$00	$00	$23	$04	$01	$00	$04	$20
10	$10	$00	$01	$00	$08	$00	$04	$20
11	$E0	$00	$23	$02	$02	$00	$04	$20
12	$00	$00	$00	$00	$04	$00	$04	$30
13	$00	$00	$01	$00	$0C	$00	$04	$40
14	$00	$00	$00	$00	$10	$00	$04	$18
15	$00	$01	$01	$03	$03	$00	$04	$28
16	$00	$00	$01	$00	$0C	$00	$04	$30
17	$00	$00	$00	$00	$0C	$00	$04	$20
18	$00	$01	$23	$03	$02	$00	$04	$20
19	$00	$00	$23	$02	$08	$00	$04	$0C
20	$00	$00	$23	$01	$01	$05	$05	$05

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 4 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$60	$00	$01	$01	$01	$00	$04	$18
2	$00	$00	$23	$00	$08	$00	$04	$18
3	$00	$00	$23	$02	$01	$00	$04	$30
4	$10	$01	$00	$02	$02	$00	$04	$20
5	$30	$00	$00	$02	$02	$00	$04	$20
6	$00	$00	$01	$00	$05	$00	$04	$20
7	$00	$00	$01	$01	$03	$00	$04	$20
8	$00	$00	$23	$00	$10	$00	$04	$30
9	$E0	$00	$00	$02	$01	$00	$04	$08
10	$00	$00	$00	$03	$03	$00	$04	$18
11	$0C	$10	$01	$03	$02	$00	$04	$18
12	$00	$18	$23	$00	$04	$00	$04	$08
13	$00	$00	$00	$00	$20	$00	$04	$C0
14	$00	$00	$01	$03	$01	$00	$04	$60
15	$00	$00	$00	$06	$10	$00	$04	$10
17	$00	$00	$23	$00	$0C	$00	$04	$30
18	$30	$01	$01	$00	$0C	$00	$04	$40
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby’s energy?.
Byte 36: Does colliding with this enemy decrease the gilby’s energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 5 - Enemy Behaviour Data.

10.0.3 Level Movement Data

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$06	$01	$01	$01
2	$00	$00	$24	$02	$01
3	$00	$FA	$01	$01	$02
4	$00	$07	$00	$01	$02
5	$00	$FC	$23	$02	$02
6	$00	$00	$00	$01	$01
7	$00	$07	$00	$01	$02
8	$00	$05	$00	$01	$02
9	$00	$FC	$23	$01	$03
10	$00	$00	$25	$00	$02
11	$03	$02	$00	$01	$02
12	$00	$FC	$21	$01	$01
13	$00	$00	$24	$02	$02
14	$03	$FA	$00	$01	$01
15	$00	$00	$00	$01	$01
16	$00	$00	$00	$02	$00
17	$00	$80	$80	$01	$01
18	$00	$06	$00	$01	$02
19	$00	$00	$00	$02	$02
20	$00	$04	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 1 - Movement Data.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$00	$00	$01	$02
2	$00	$E9	$00	$01	$02
3	$00	$17	$00	$01	$03
4	$00	$FC	$00	$02	$02
5	$00	$06	$24	$01	$02
6	$00	$07	$24	$01	$01
7	$00	$04	$00	$01	$01
8	$00	$00	$00	$00	$01
9	$00	$04	$24	$01	$02
10	$00	$06	$00	$01	$00
11	$00	$00	$00	$01	$02
12	$00	$03	$00	$01	$00
13	$00	$00	$00	$02	$02
14	$00	$E9	$00	$01	$02
15	$00	$03	$22	$01	$01
16	$00	$FC	$00	$01	$00
17	$00	$17	$00	$01	$03
18	$00	$00	$00	$01	$01
19	$0C	$05	$24	$01	$02
20	$00	$FC	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$00	$00	$02	$02
2	$00	$F8	$01	$01	$0C
3	$00	$03	$23	$01	$01
4	$04	$08	$00	$01	$03
5	$00	$00	$00	$00	$00
6	$04	$F9	$23	$01	$07
7	$00	$03	$23	$01	$02
8	$00	$00	$00	$00	$00
9	$00	$00	$00	$01	$02
10	$00	$00	$00	$01	$02
11	$00	$FD	$21	$02	$01
12	$00	$00	$23	$00	$01
13	$00	$00	$00	$00	$00
14	$00	$00	$00	$00	$00
15	$00	$00	$00	$00	$00
16	$00	$00	$00	$00	$00
17	$00	$03	$00	$01	$01
18	$00	$00	$00	$02	$02
19	$00	$05	$00	$01	$02
20	$00	$06	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$04	$23	$01	$02
2	$00	$0C	$00	$01	$02
3	$00	$03	$00	$01	$03
4	$00	$00	$00	$00	$01
5	$00	$04	$23	$01	$02
6	$00	$00	$00	$00	$00
7	$00	$00	$00	$00	$00
8	$00	$00	$00	$00	$00
9	$00	$80	$25	$80	$02
10	$00	$0A	$00	$01	$02
11	$00	$00	$00	$00	$01
12	$00	$00	$00	$00	$00
13	$00	$F9	$00	$01	$01
14	$00	$80	$80	$80	$80
15	$00	$00	$00	$02	$03
16	$00	$F8	$00	$01	$04
17	$00	$04	$00	$01	$00
18	$00	$00	$24	$02	$03
19	$01	$01	$00	$01	$01
20	$00	$FA	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$FC	$00	$01	$02
2	$00	$00	$25	$00	$01
3	$01	$FD	$24	$01	$02
4	$00	$00	$00	$01	$00
5	$05	$07	$03	$01	$01
6	$00	$F4	$00	$01	$02
7	$00	$FE	$00	$01	$01
8	$00	$04	$24	$01	$02
9	$00	$00	$00	$00	$00
10	$00	$00	$20	$00	$01
11	$00	$00	$00	$01	$02
12	$00	$00	$23	$02	$02
13	$00	$00	$00	$00	$00
14	$06	$FC	$00	$01	$02
15	$00	$00	$00	$00	$00
17	$00	$02	$22	$01	$01
18	$00	$00	$00	$02	$02
20	$00	$0C	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 5 - Movement Data.

Appendix: Planet Data

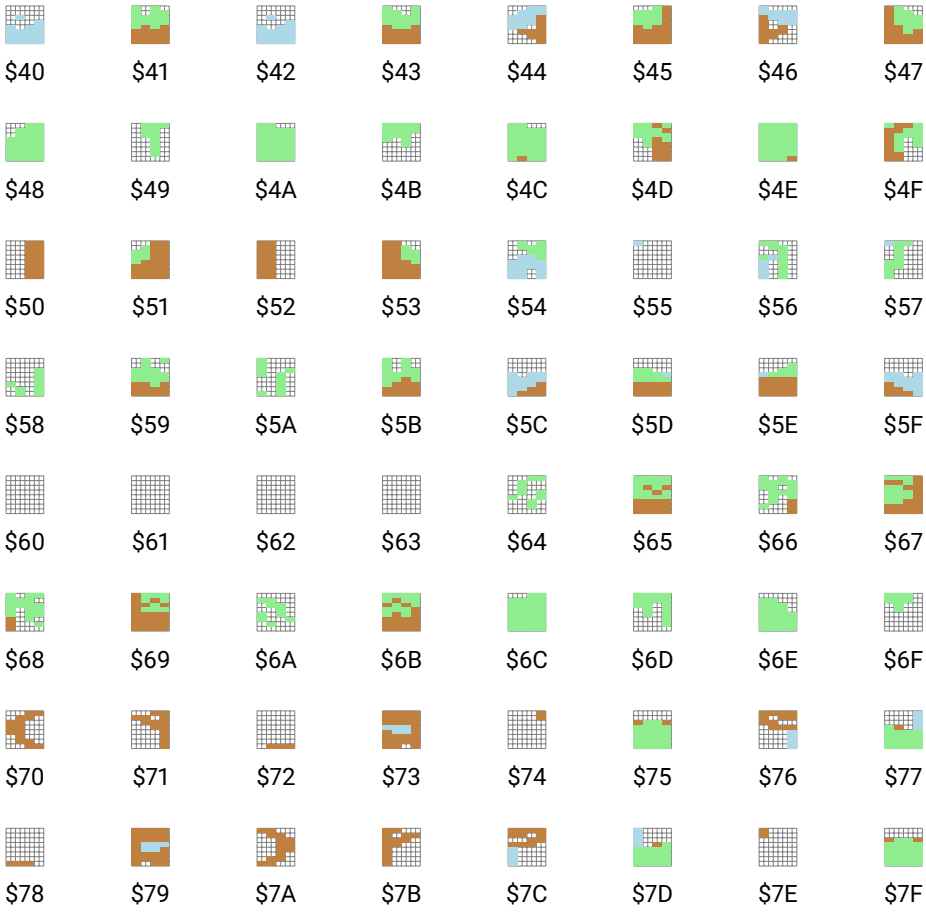


Figure 11.1: Tilesheet: planet1Charset

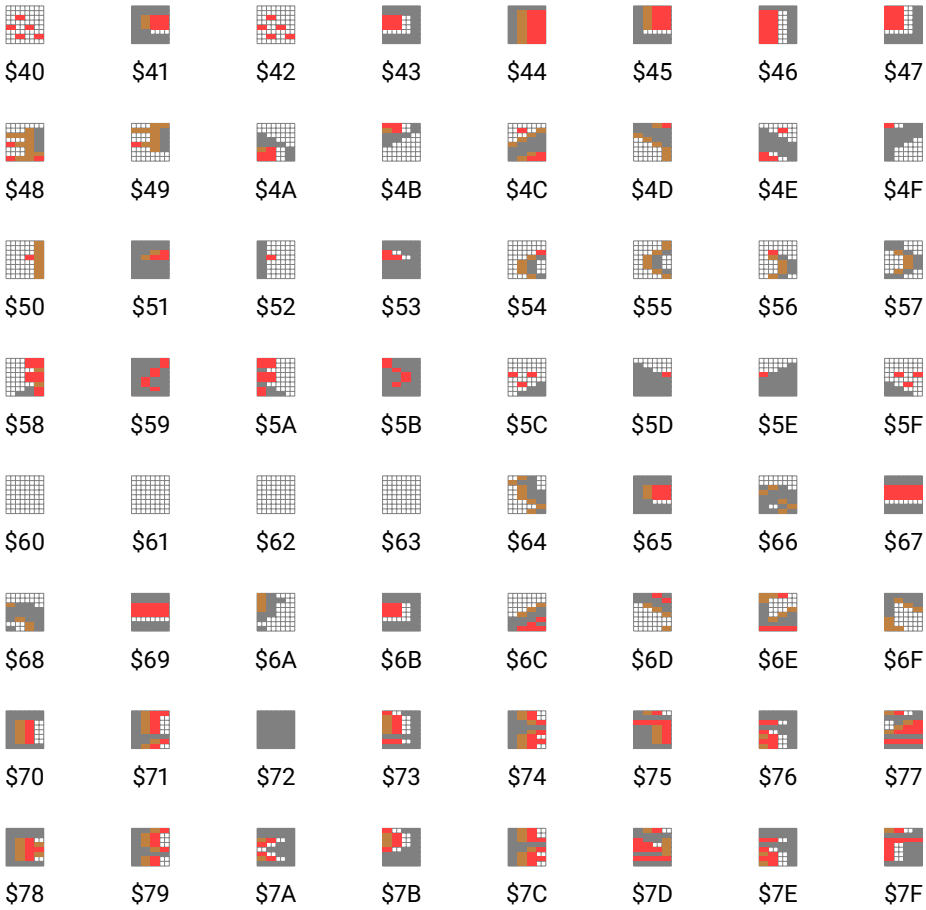


Figure 11.2: Tilesheet: planet2Charset

Figure 11.3: Tilesheet: planet3Charset

Figure 11.4: Tilesheet: planet4Charset

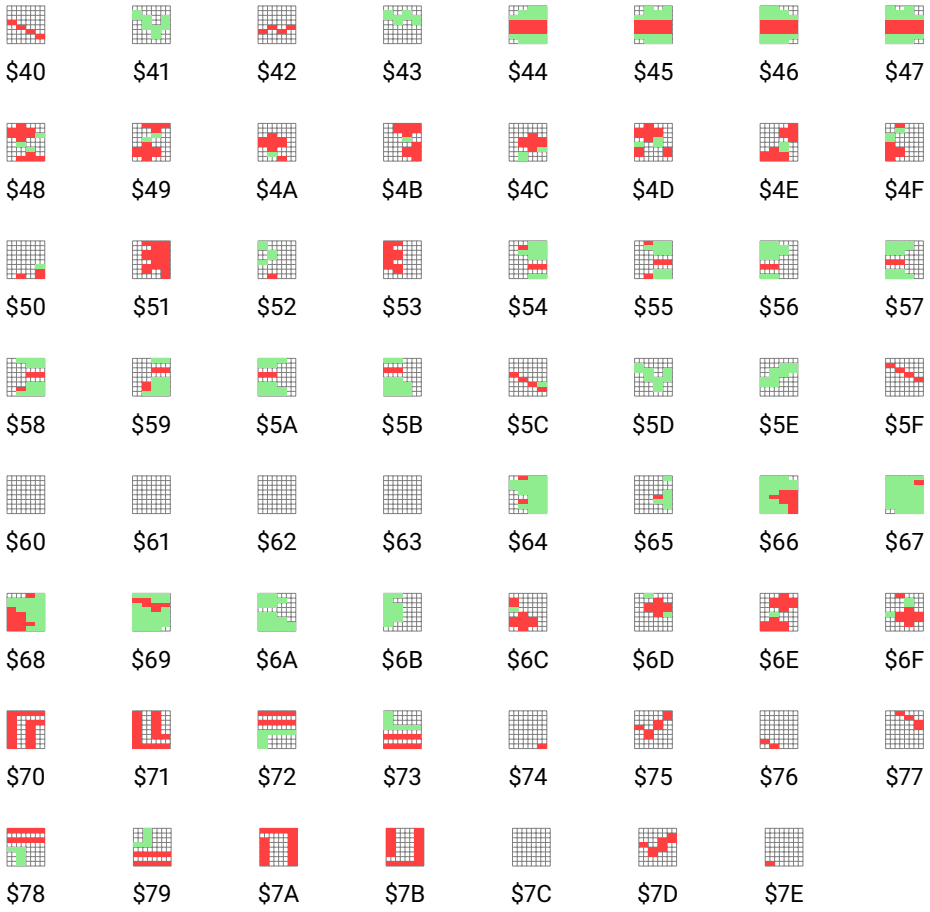
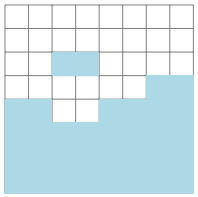
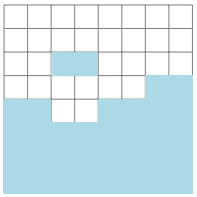


Figure 11.5: Tilesheet: planet5Charset



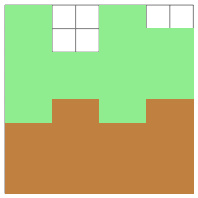
(a) planet1Charset $40



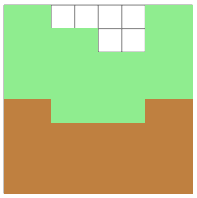
(b) planet1Charset $42

Figure 11.6: Tilesheet: Planet 1 Sea.

Figure 11.7: planet1Charset Sea



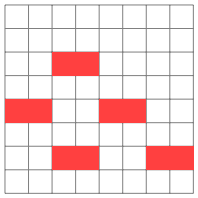
(a) planet1Charset $41



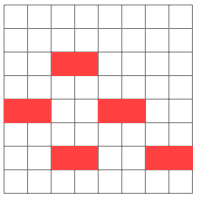
(b) planet1Charset $43

Figure 11.8: Tilesheet: Planet 1 Land.

Figure 11.9: planet1Charset Land



(a) planet2Charset $40



(b) planet2Charset $42

Figure 11.10: Tilesheet: Planet 2 Sea.

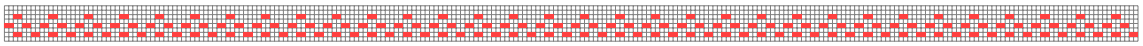
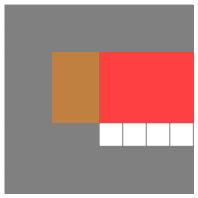
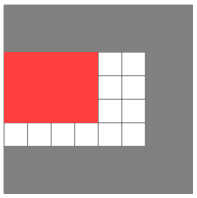


Figure 11.11: planet2Charset Sea



(a) planet2Charset $41



(b) planet2Charset $43

Figure 11.12: Tilesheet: Planet 2 Land.

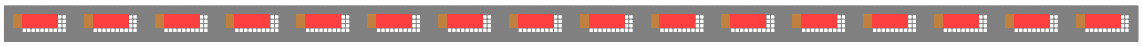
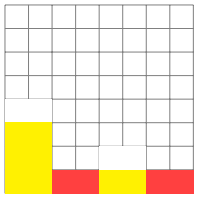
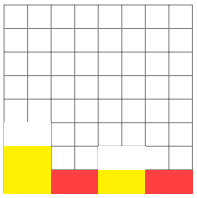


Figure 11.13: planet2Charset Land



(a) planet3Charset $40



(b) planet3Charset $42

Figure 11.14: Tilesheet: Planet 3 Sea.

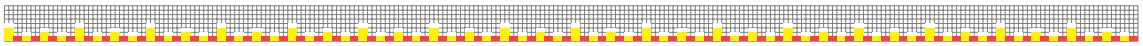


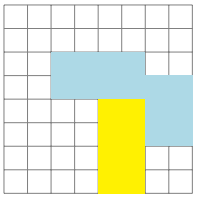
Figure 11.15: planet3Charset Sea

(a) planet3Charset $41

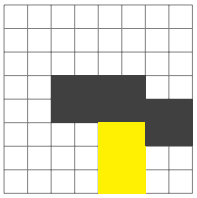
(b) planet3Charset $43

Figure 11.16: Tilesheet: Planet 3 Land.

Figure 11.17: planet3Charset Land



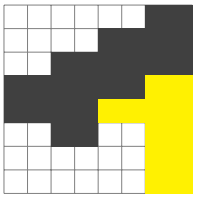
(a) planet4Charset $40



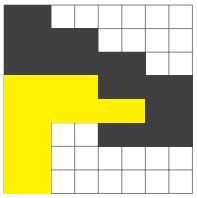
(b) planet4Charset $42

Figure 11.18: Tilesheet: Planet 4 Sea.

Figure 11.19: planet4Charset Sea



(a) planet4Charset $41



(b) planet4Charset $43

Figure 11.20: Tilesheet: Planet 4 Land.

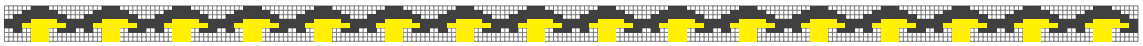
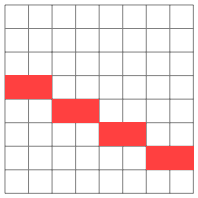
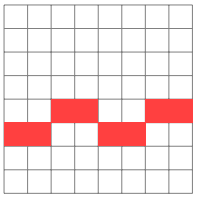


Figure 11.21: planet4Charset Land



(a) planet5Charset $40



(b) planet5Charset $42

Figure 11.22: Tilesheet: Planet 5 Sea.

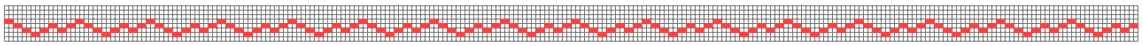
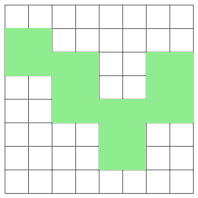
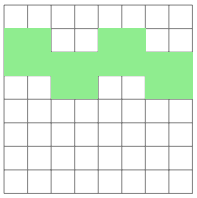


Figure 11.23: planet5Charset Sea



(a) planet5Charset $41

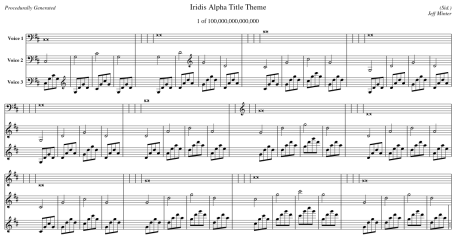


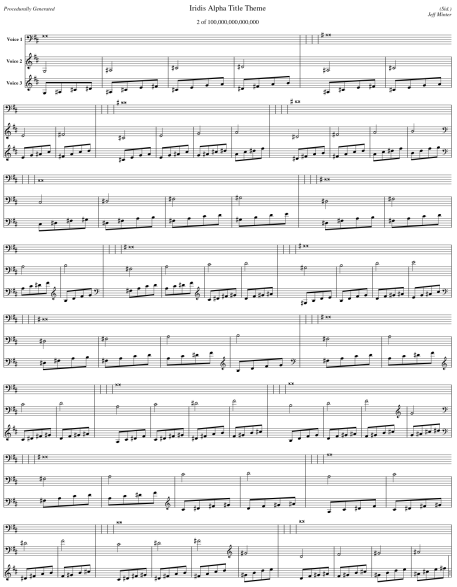
(b) planet5Charset $43

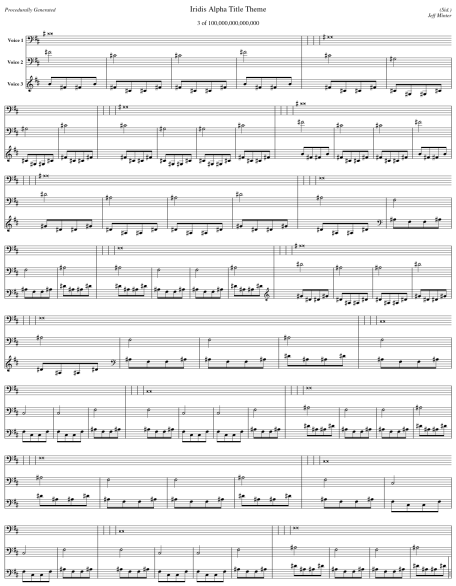
Figure 11.24: Tilesheet: Planet 5 Land.

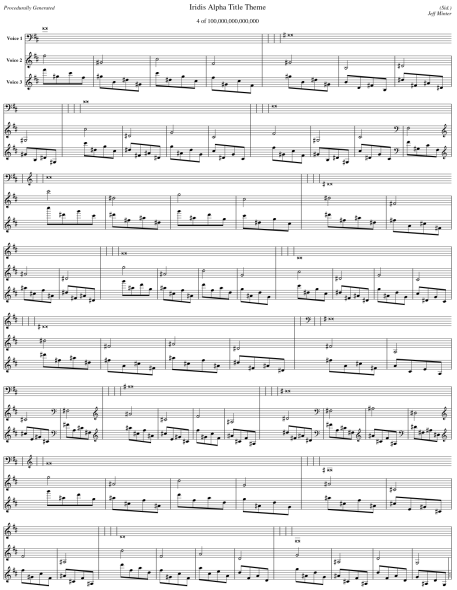
Figure 11.25: planet5Charset Land

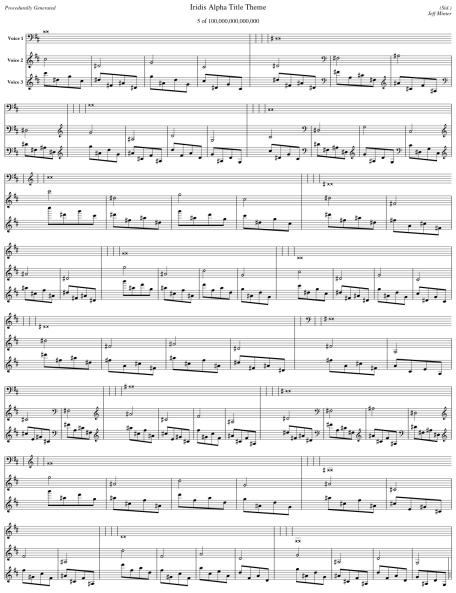
Appendix: 18/100,000,000,000,000 Theme Tunes

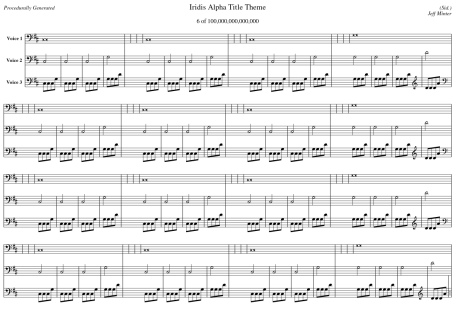


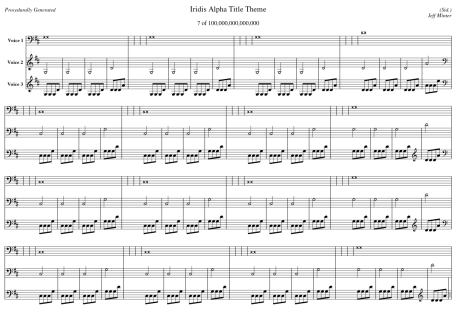


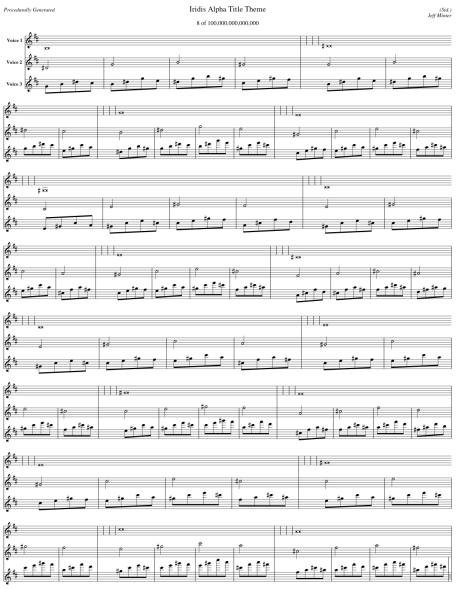

Procedurally Generated
Iridis Alpha Title Theme
3 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

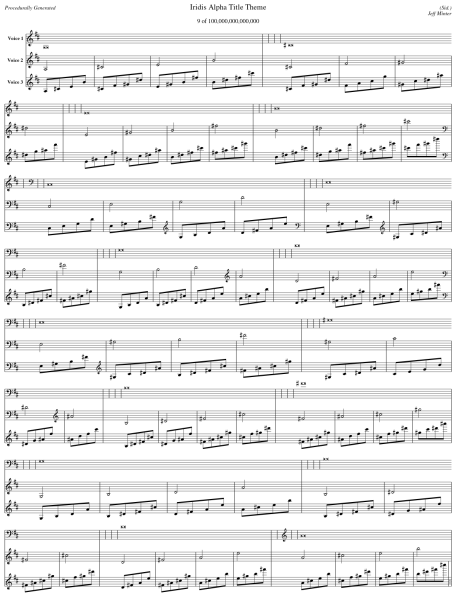

Procedurally Generated
Iridis Alpha Title Theme
4 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

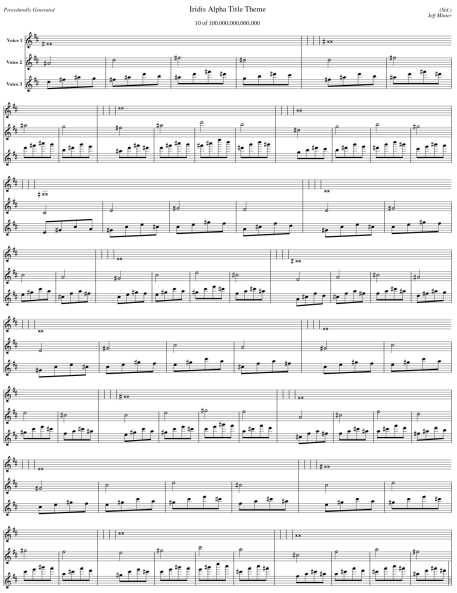

Procedurally Generated
Iridis Alpha Title Theme
5 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

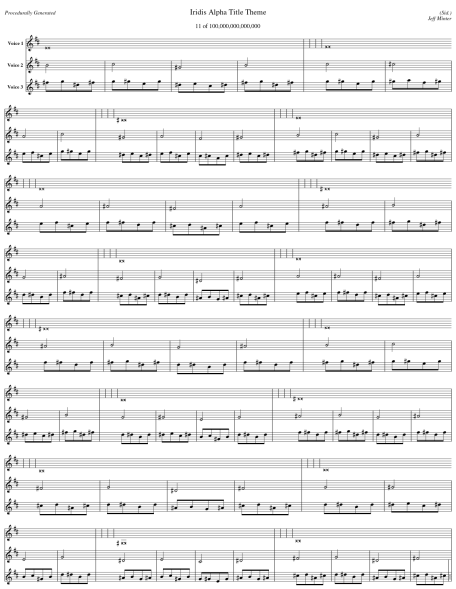

Procedurally Generated
Iridis Alpha Title Theme
6 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

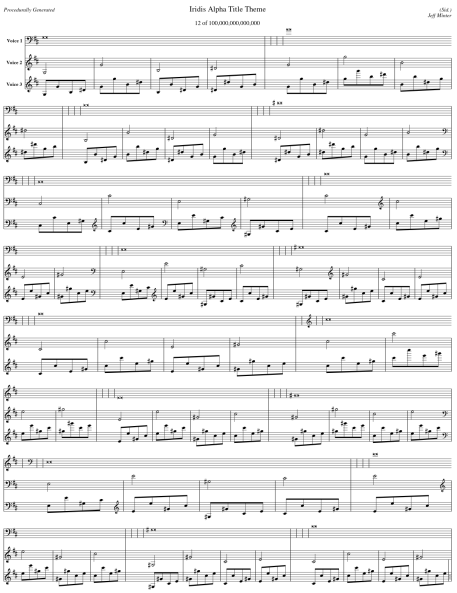

Procedurally Generated
Iridis Alpha Title Theme
7 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

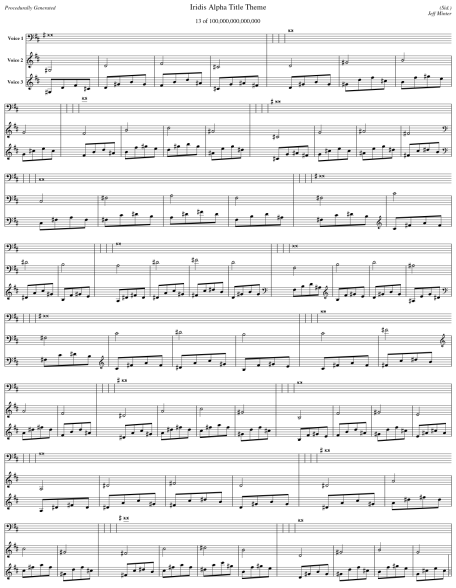

Procedurally Generated
Iridis Alpha Title Theme
8 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

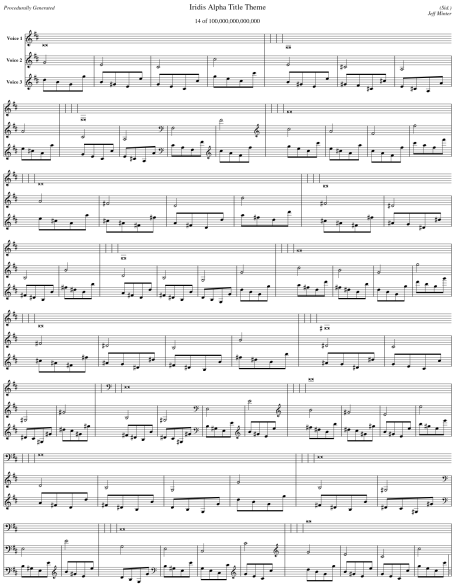

Procedurally Generated
Iridis Alpha Title Theme
9 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

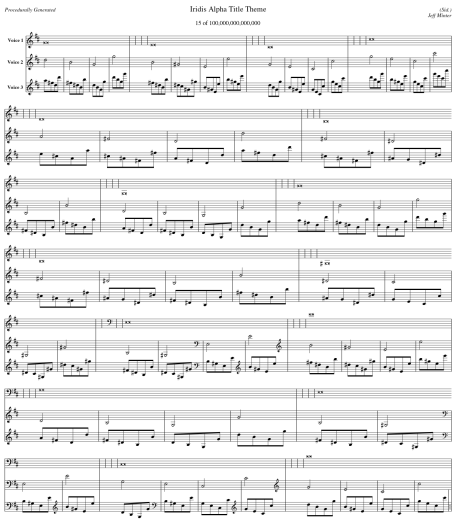

Procedurally Generated
Iridis Alpha Title Theme
10 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

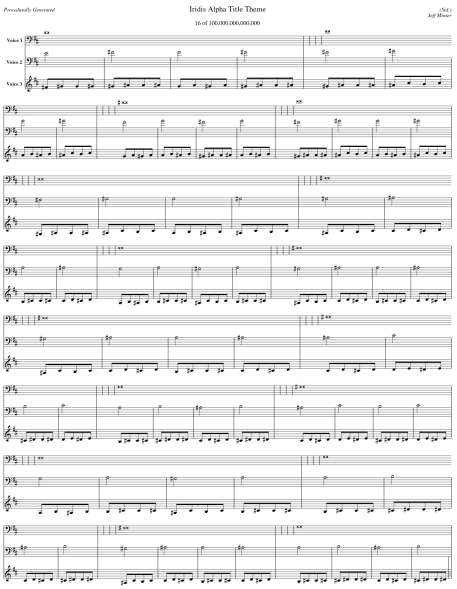

Procedurally Generated
Iridis Alpha Title Theme
11 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

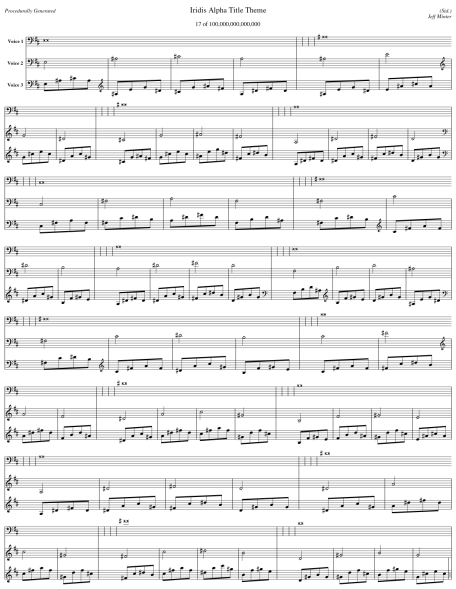

Procedurally Generated
Iridis Alpha Title Theme
12 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

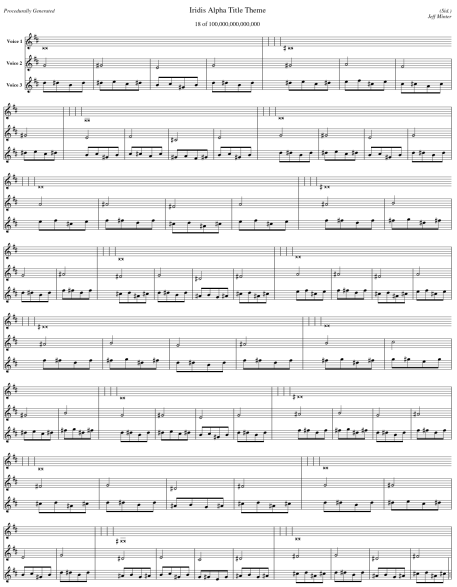

Procedurally Generated
Iridis Alpha Title Theme
13 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

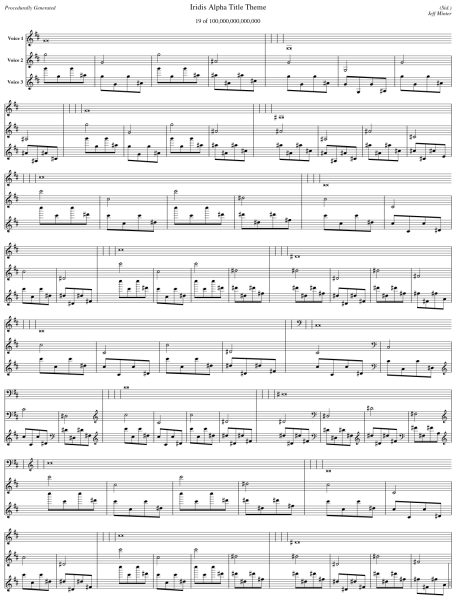

Procedurally Generated
Iridis Alpha Title Theme
14 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

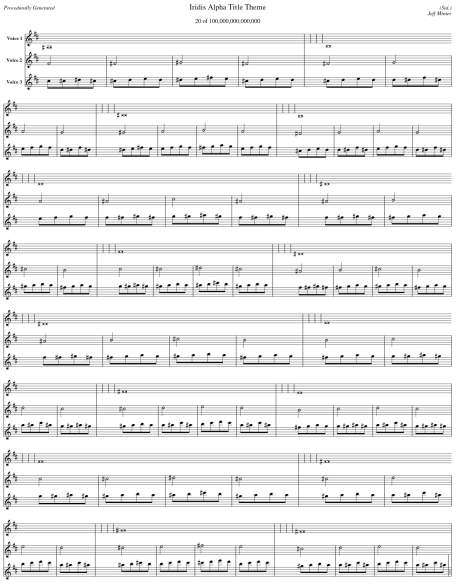

Procedurally Generated
Iridis Alpha Title Theme
15 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

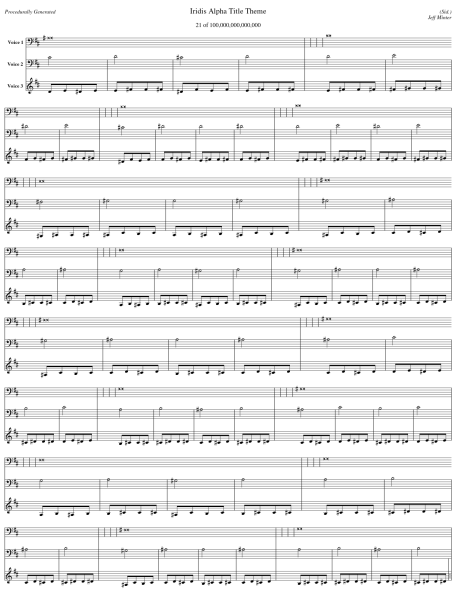

Procedurally Generated
Iridis Alpha Title Theme
16 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

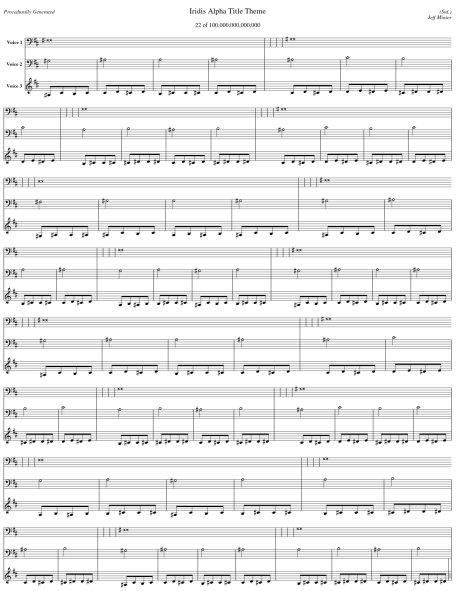

Procedurally Generated
Iridis Alpha Title Theme
17 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

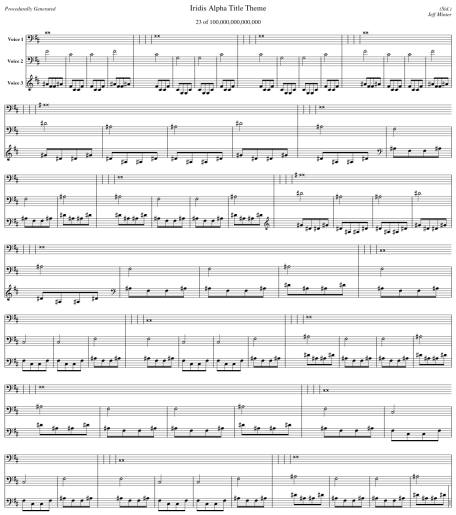

Procedurally Generated
Iridis Alpha Title Theme
18 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

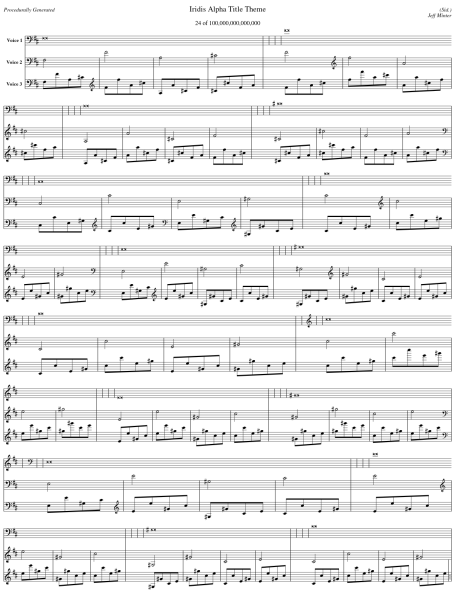

Procedurally Generated
Iridis Alpha Title Theme
19 of 100,000,000,000,000
(Std.)
Jeff Minter
Voice 1
Voice 2
Voice 3

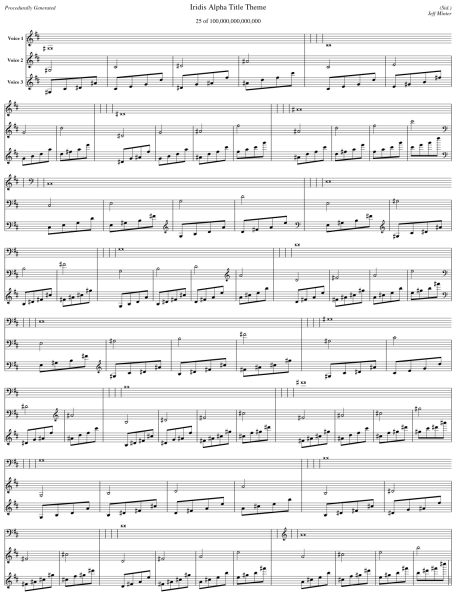

Procedurally Generated
Iridis Alpha Title Theme
20 of 100,000,000,000,000
(Std.)
Jeff Minter
Voice 1
Voice 2
Voice 3

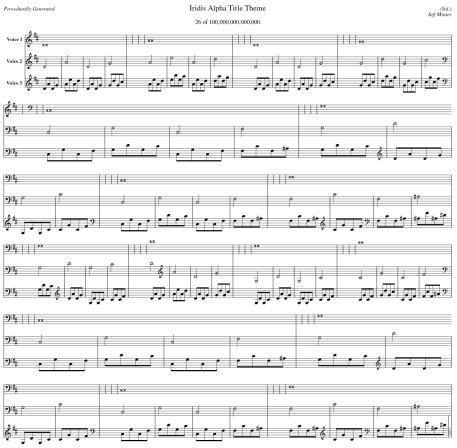

Procedurally Generated
Iridis Alpha Title Theme
21 of 100,000,000,000,000
(Std.)
Jeff Minter
Voice 1
Voice 2
Voice 3

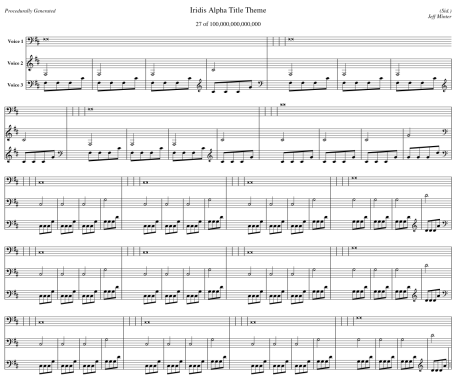

Procedurally Generated
Iridis Alpha Title Theme
22 of 100,000,000,000,000
(Std.)
Jeff Minter
Voice 1
Voice 2
Voice 3

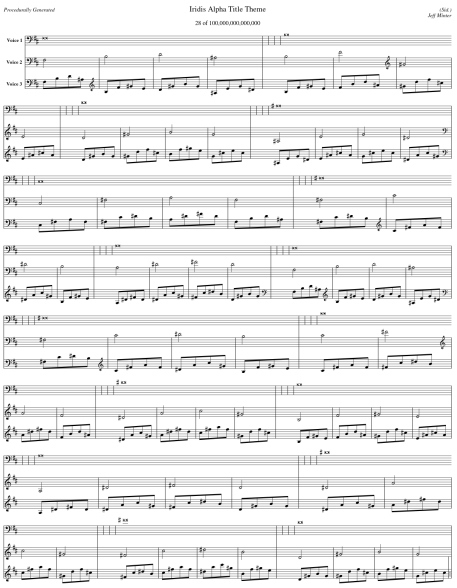

Procedurally Generated
Iridis Alpha Title Theme
23 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

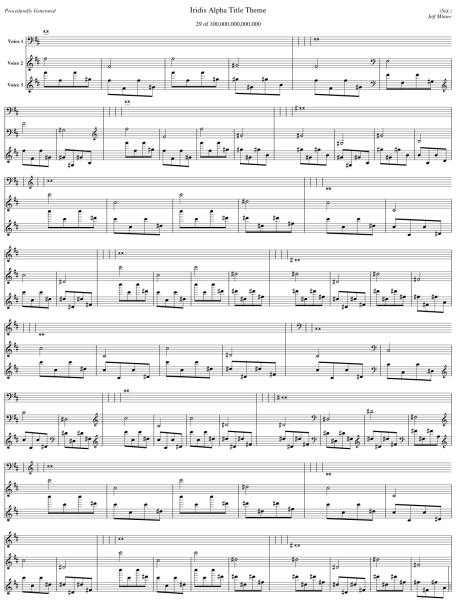

Procedurally Generated
Iridis Alpha Title Theme
24 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

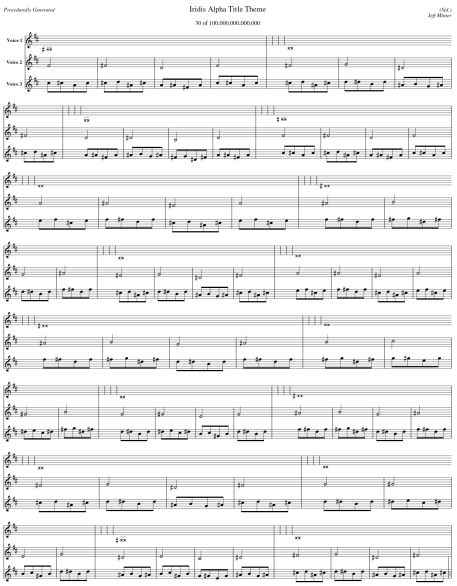

Procedurally Generated
Iridis Alpha Title Theme
25 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

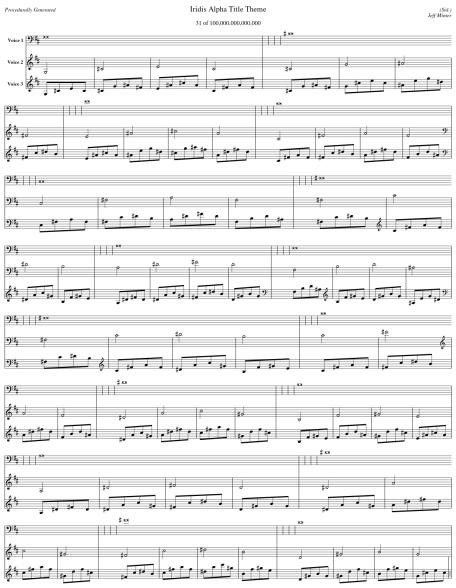

Procedurally Generated
Iridis Alpha Title Theme
26 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

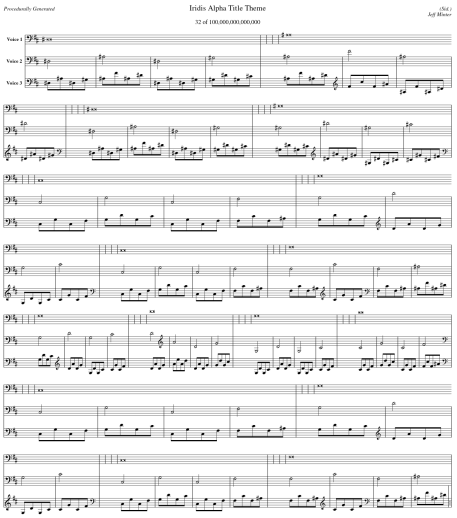

Procedurally Generated
Iridis Alpha Title Theme
27 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

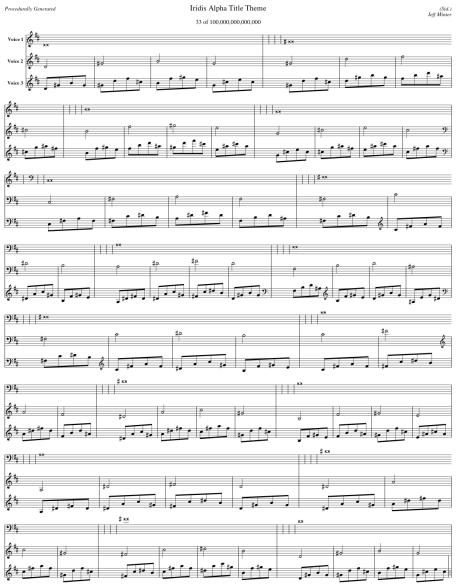

Procedurally Generated
Iridis Alpha Title Theme
28 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

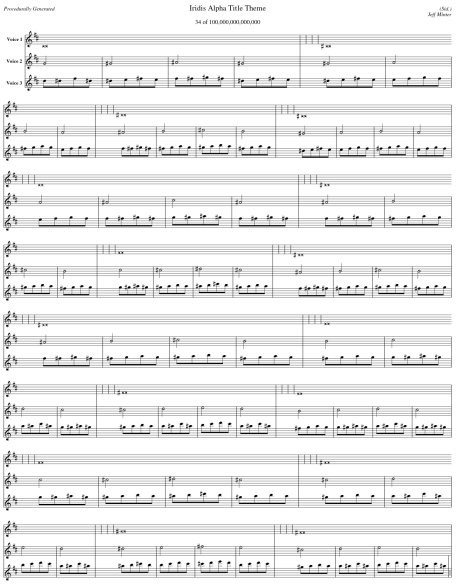

Procedurally Generated
Iridis Alpha Title Theme
29 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

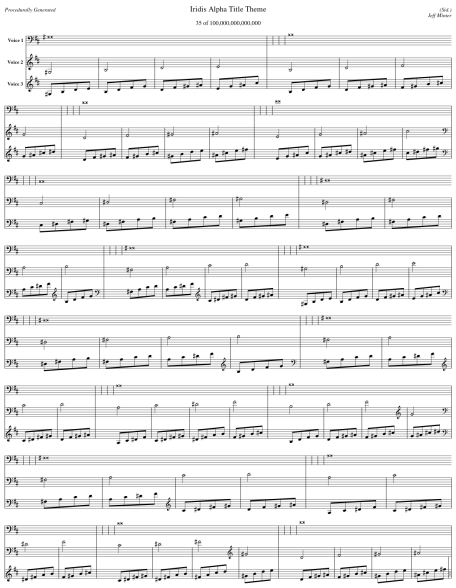

Procedurally Generated
Iridis Alpha Title Theme
30 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

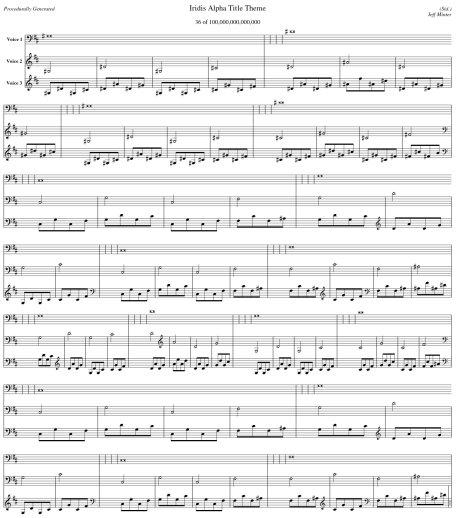

Procedurally Generated
Iridis Alpha Title Theme
31 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

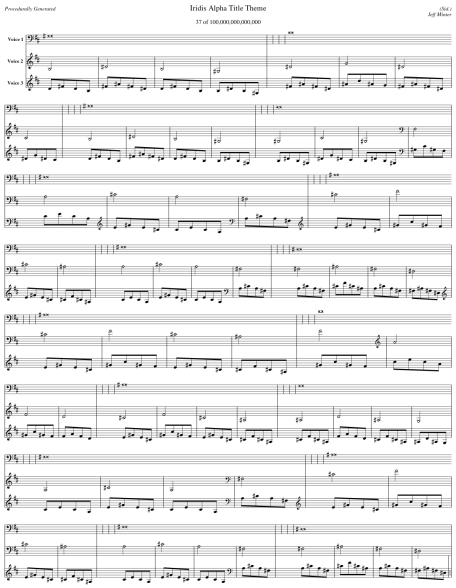

Procedurally Generated
Iridis Alpha Title Theme
32 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

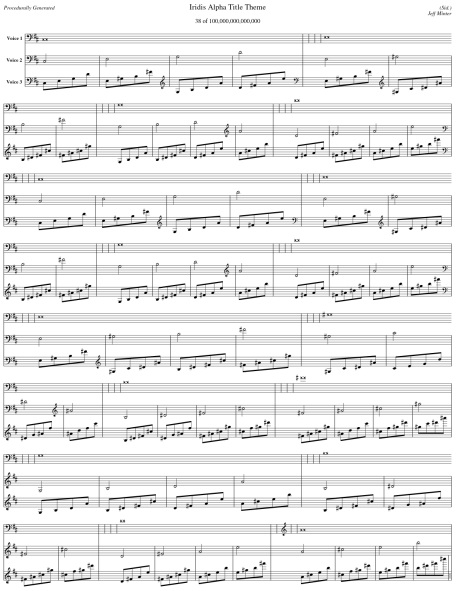

Procedurally Generated
Iridis Alpha Title Theme
33 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

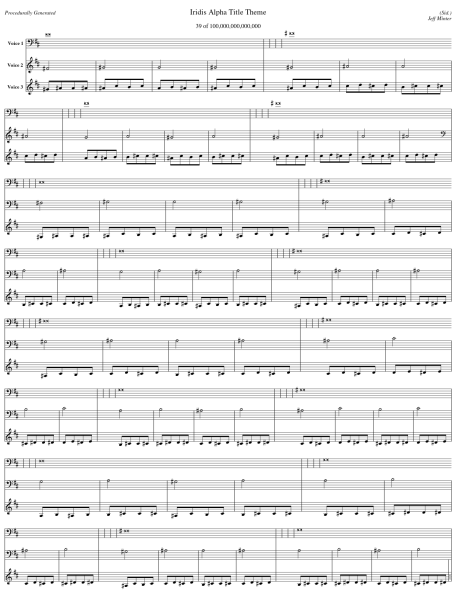

Procedurally Generated
Iridis Alpha Title Theme
34 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

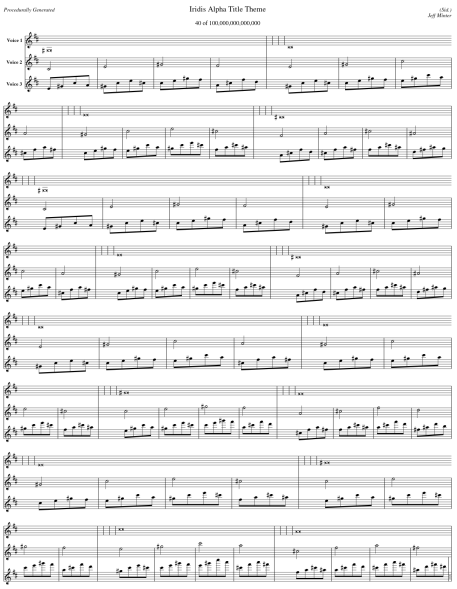

Procedurally Generated
Iridis Alpha Title Theme
35 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

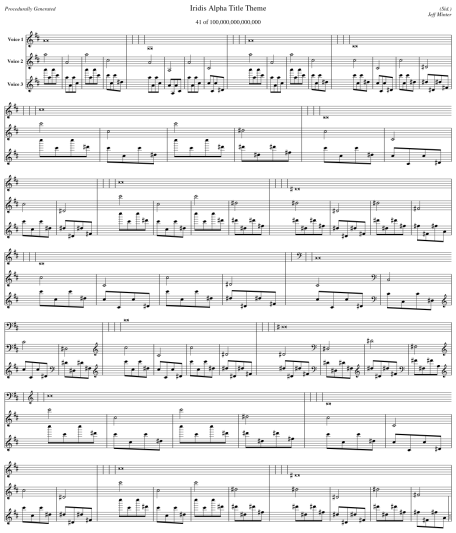

Procedurally Generated
Iridis Alpha Title Theme
36 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

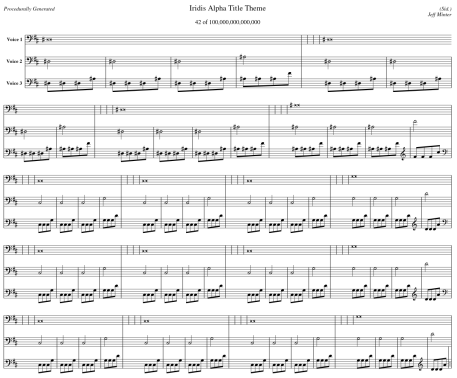

Procedurally Generated
Iridis Alpha Title Theme
37 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

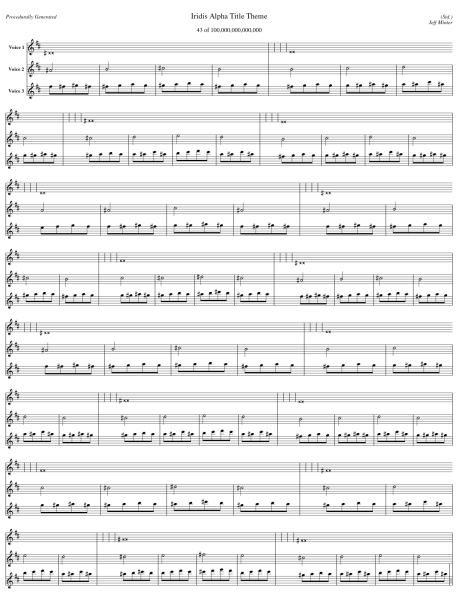

Procedurally Generated
Iridis Alpha Title Theme
38 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

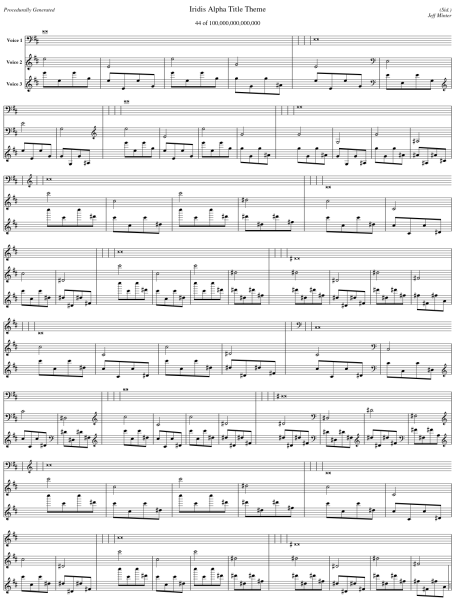

Procedurally Generated
Iridis Alpha Title Theme
39 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

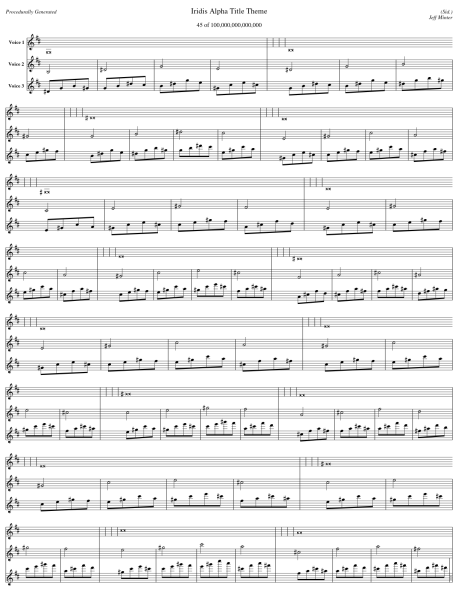

Procedurally Generated
Iridis Alpha Title Theme
40 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

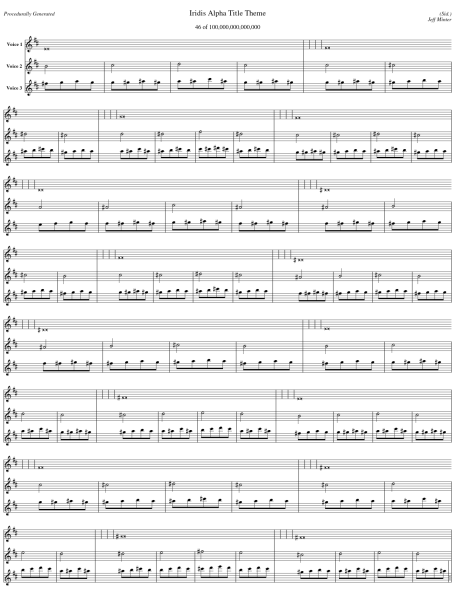

Procedurally Generated
Iridis Alpha Title Theme
41 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

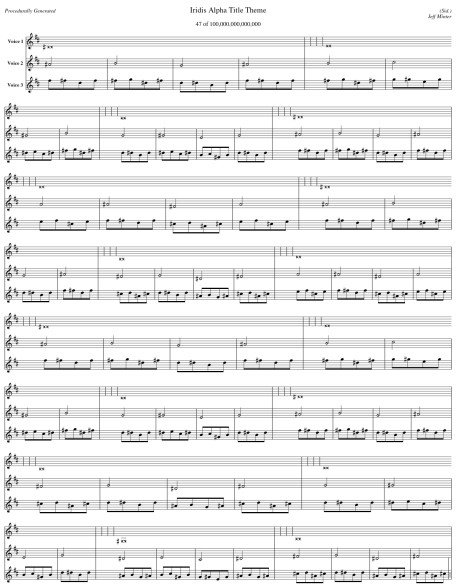

Procedurally Generated
Iridis Alpha Title Theme
42 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

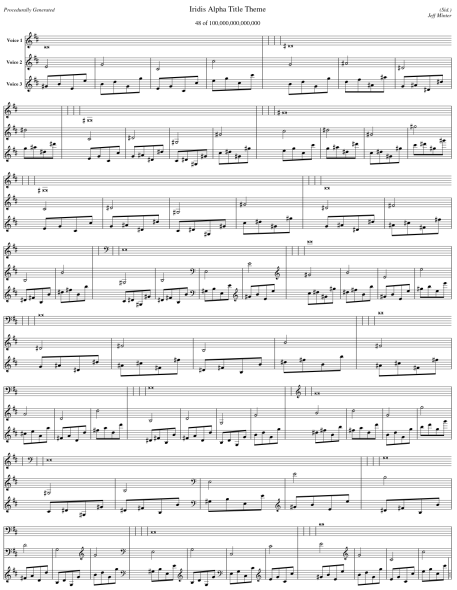

Procedurally Generated
Iridis Alpha Title Theme
43 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

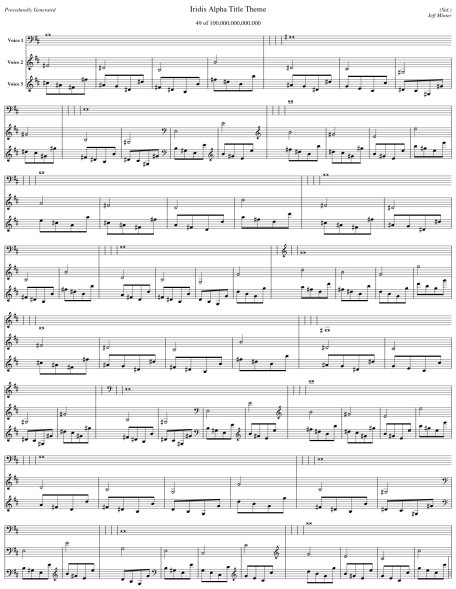

Procedurally Generated
Iridis Alpha Title Theme
44 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

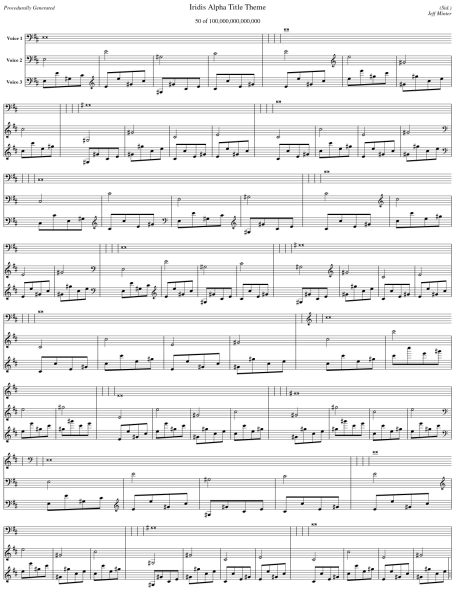

Procedurally Generated
Iridis Alpha Title Theme
45 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

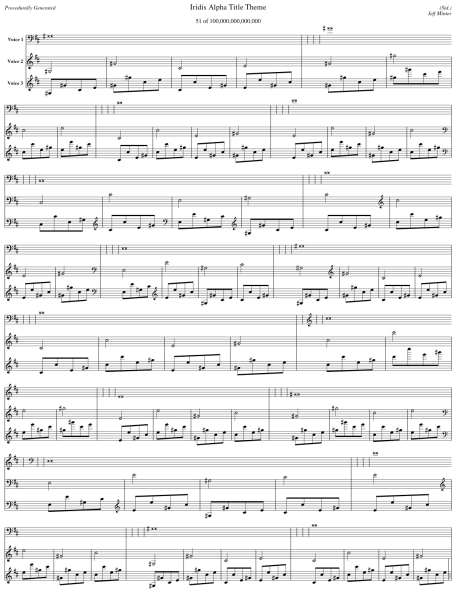

Procedurally Generated
Iridis Alpha Title Theme
46 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

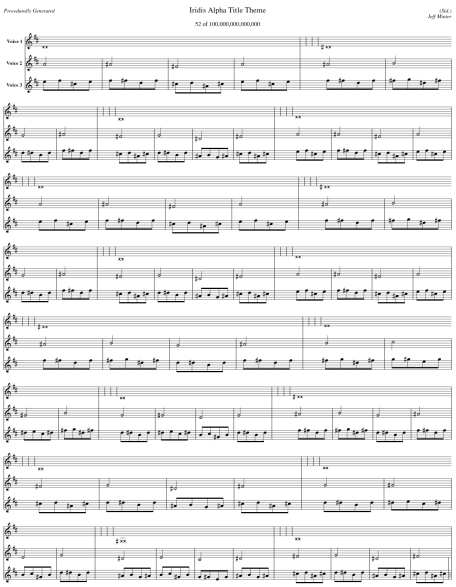

Procedurally Generated
Iridis Alpha Title Theme
47 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

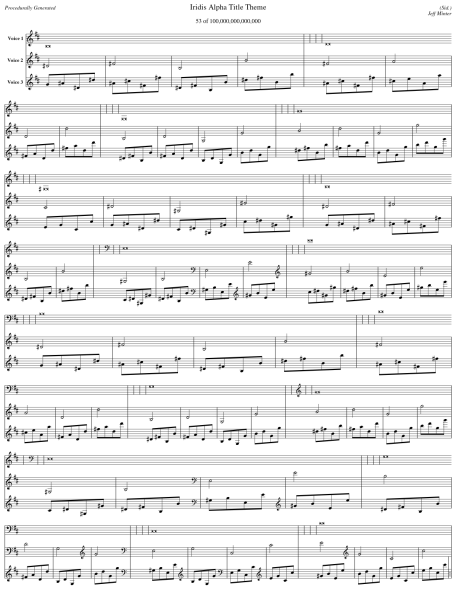

Procedurally Generated
Iridis Alpha Title Theme
48 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

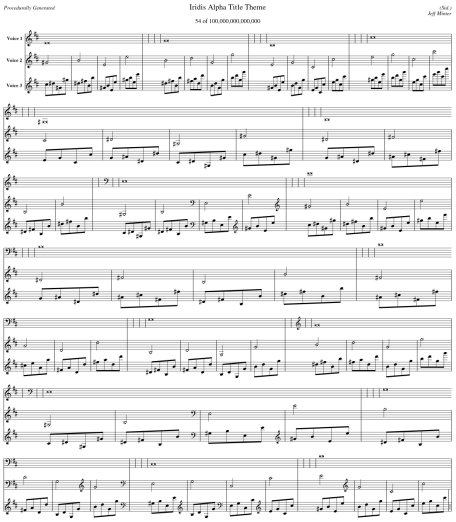

Procedurally Generated
Iridis Alpha Title Theme
49 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

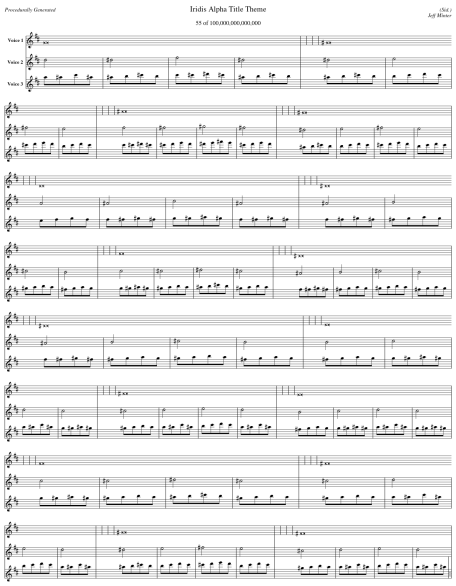

Procedurally Generated
Iridis Alpha Title Theme
50 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

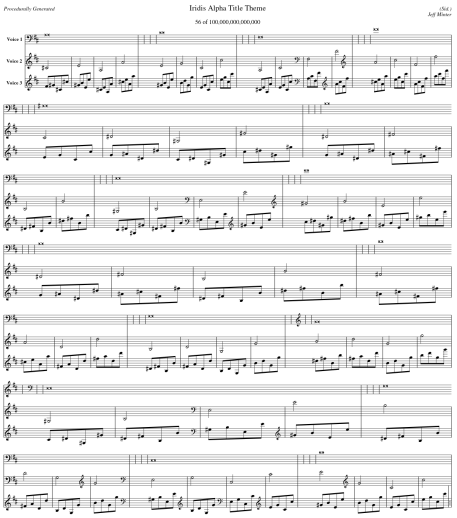

Procedurally Generated
Iridis Alpha Title Theme
51 of 100,000,000,000,000
(Sid.)
Jeff Minter
Voice 1
Voice 2
Voice 3

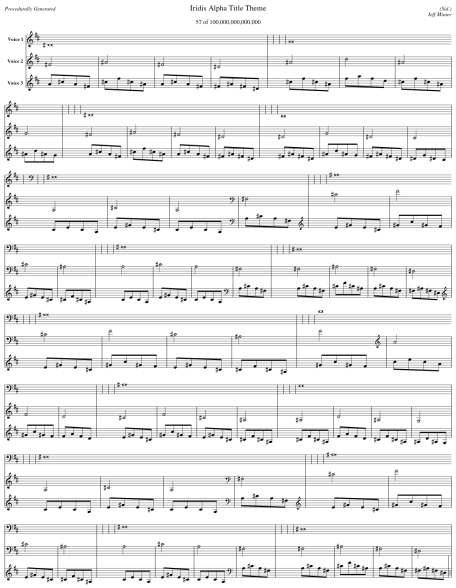

Procedurally Generated
Iridis Alpha Title Theme
52 of 100,000,000,000,000
(Sid.)
Jeff Minter
Voice 1
Voice 2
Voice 3

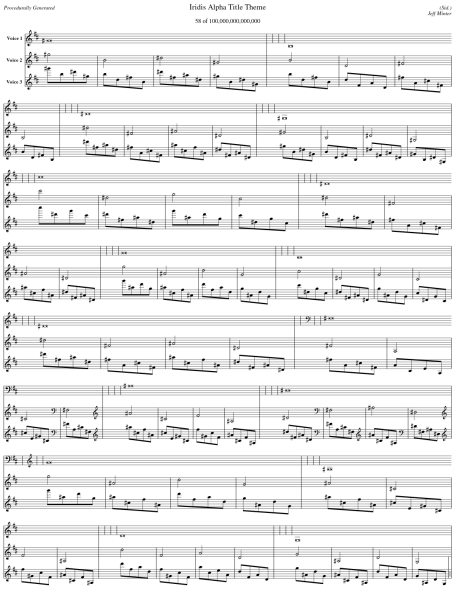

Procedurally Generated
Iridis Alpha Title Theme
53 of 100,000,000,000,000
(Sid.)
Jeff Minter
Voice 1
Voice 2
Voice 3

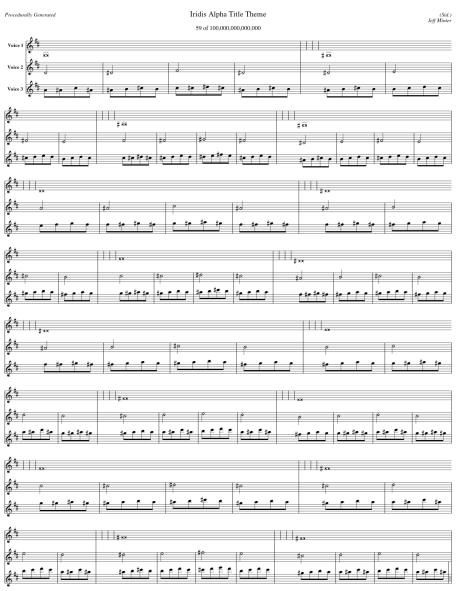

Procedurally Generated
Iridis Alpha Title Theme
54 of 100,000,000,000,000
(Sid.)
Jeff Minter
Voice 1
Voice 2
Voice 3

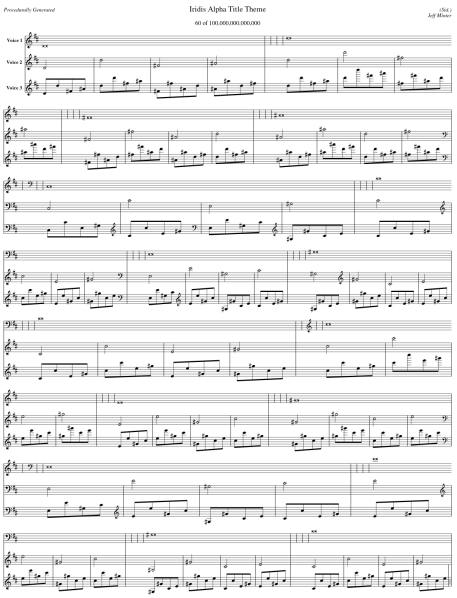

Procedurally Generated
Iridis Alpha Title Theme
55 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

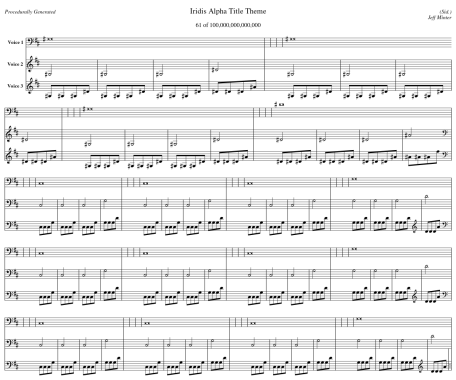

Procedurally Generated
Iridis Alpha Title Theme
56 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

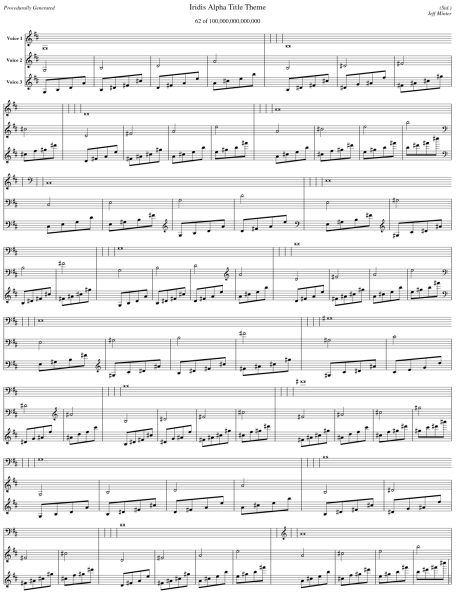

Procedurally Generated
Iridis Alpha Title Theme
57 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

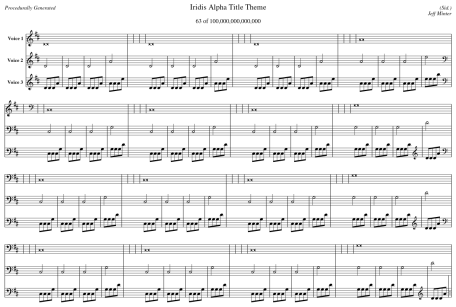

Procedurally Generated
Iridis Alpha Title Theme
58 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

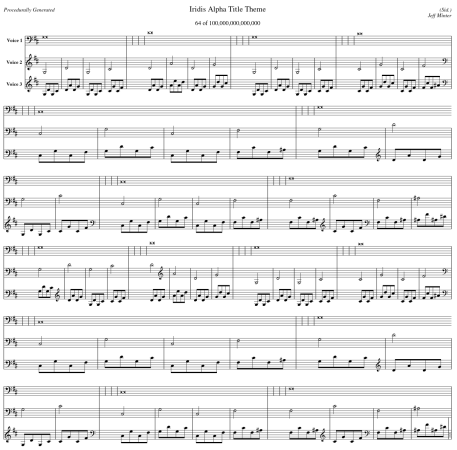

Procedurally Generated
Iridis Alpha Title Theme
59 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

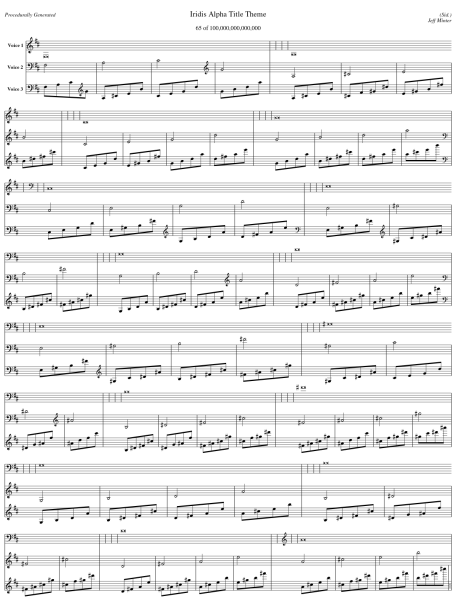

Procedurally Generated
Iridis Alpha Title Theme
60 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

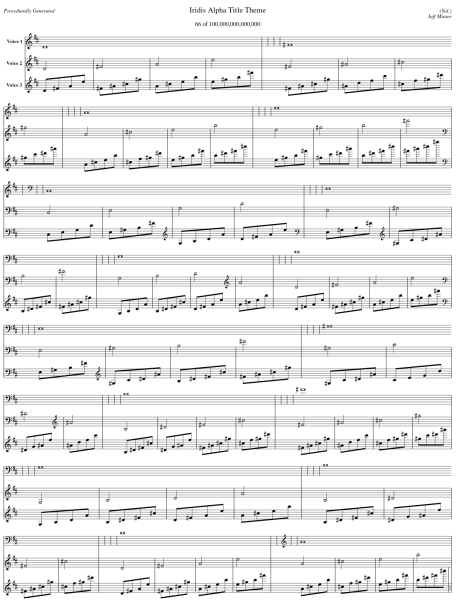

Procedurally Generated
Iridis Alpha Title Theme
61 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

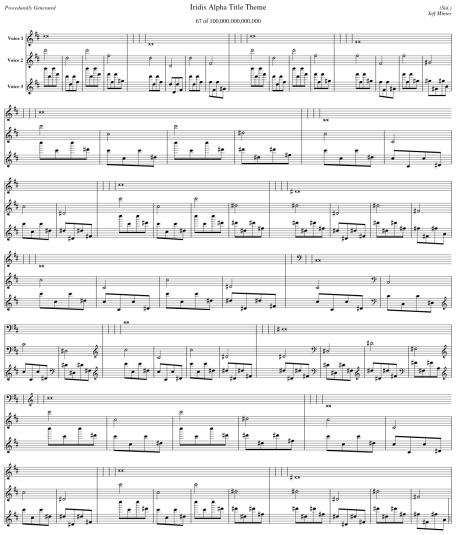

Procedurally Generated
Iridis Alpha Title Theme
62 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

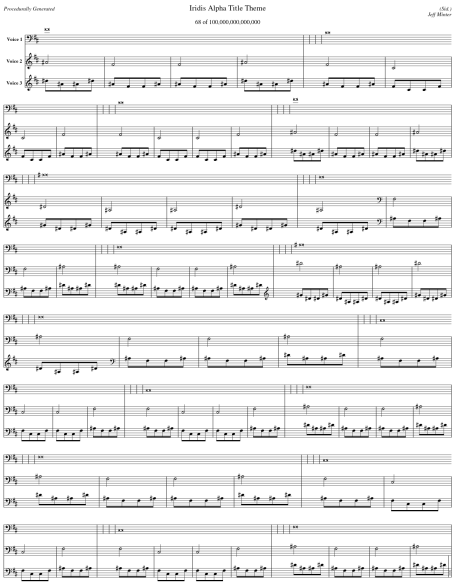

Procedurally Generated
Iridis Alpha Title Theme
63 of 100,000,000,000,000
(Sic.)
Jeff Moser
Voice 1
Voice 2
Voice 3

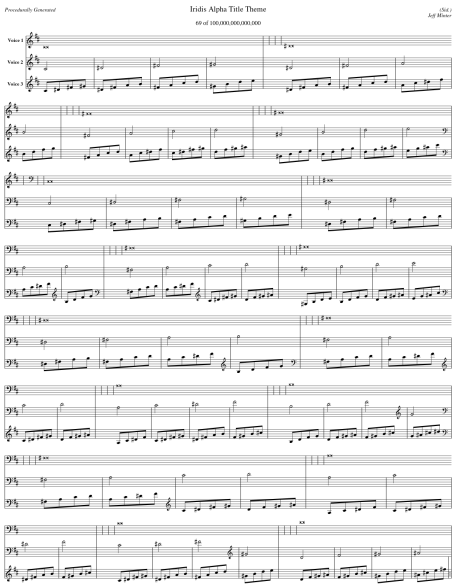

Procedurally Generated
Iridis Alpha Title Theme
64 of 100,000,000,000,000
(Sic.)
Jeff Moser
Voice 1
Voice 2
Voice 3

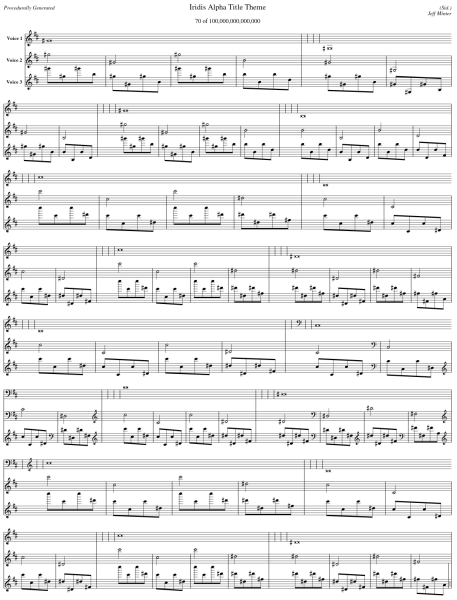

Procedurally Generated
Iridis Alpha Title Theme
65 of 100,000,000,000,000
(Sic.)
Jeff Moser
Voice 1
Voice 2
Voice 3

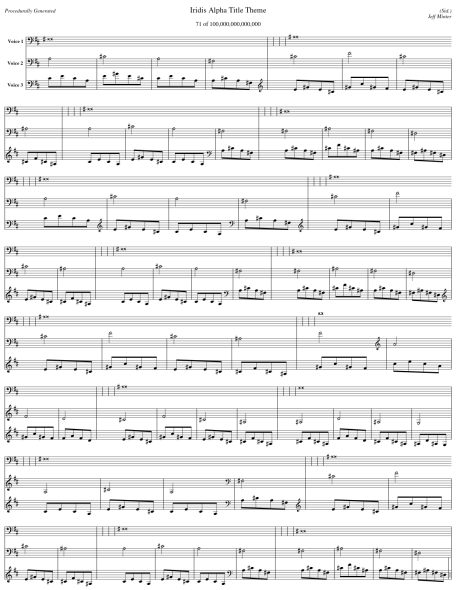

Procedurally Generated
Iridis Alpha Title Theme
66 of 100,000,000,000,000
(Sic.)
Jeff Moser
Voice 1
Voice 2
Voice 3

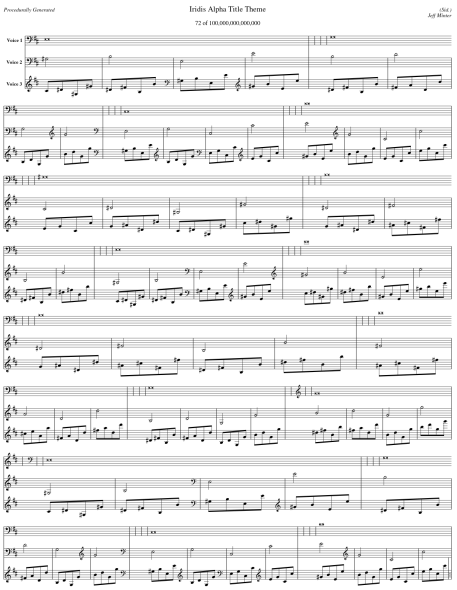

Procedurally Generated
Iridis Alpha Title Theme
67 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

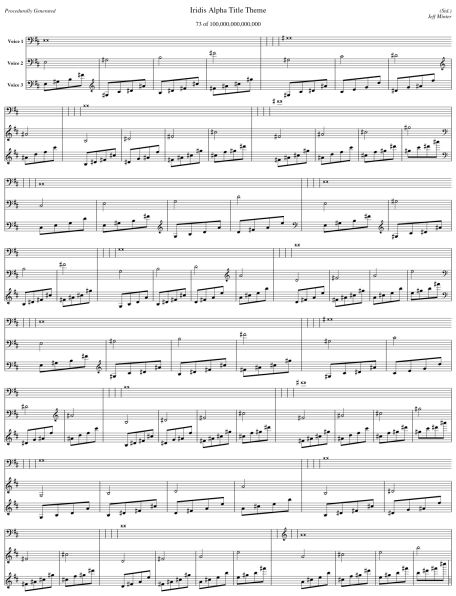

Procedurally Generated
Iridis Alpha Title Theme
68 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

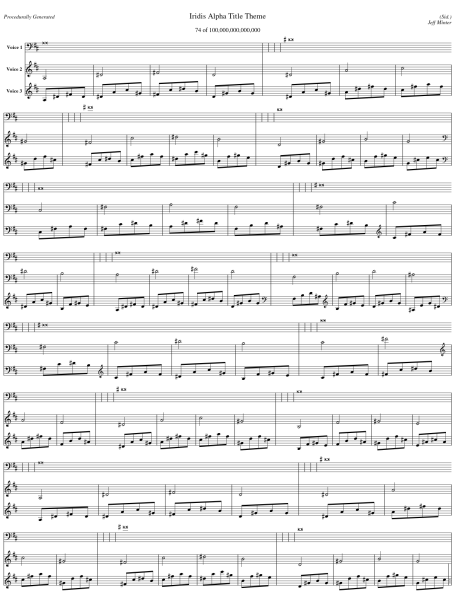

Procedurally Generated
Iridis Alpha Title Theme
69 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

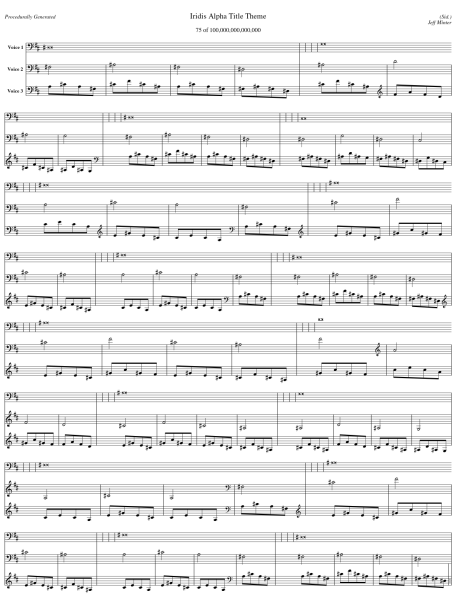

Procedurally Generated
Iridis Alpha Title Theme
70 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

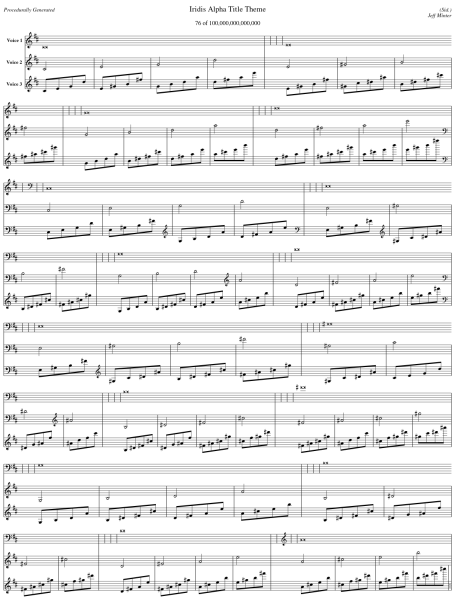

Procedurally Generated
Iridis Alpha Title Theme
71 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

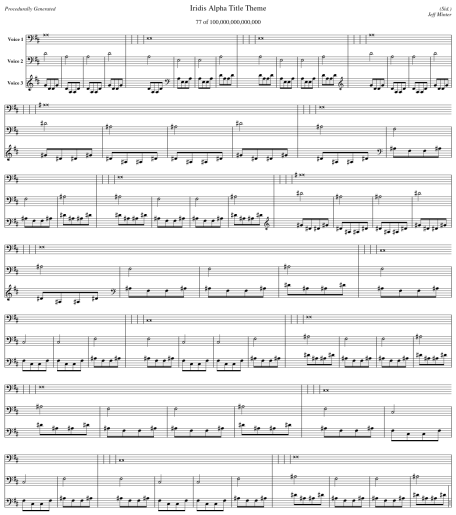

Procedurally Generated
Iridis Alpha Title Theme
72 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

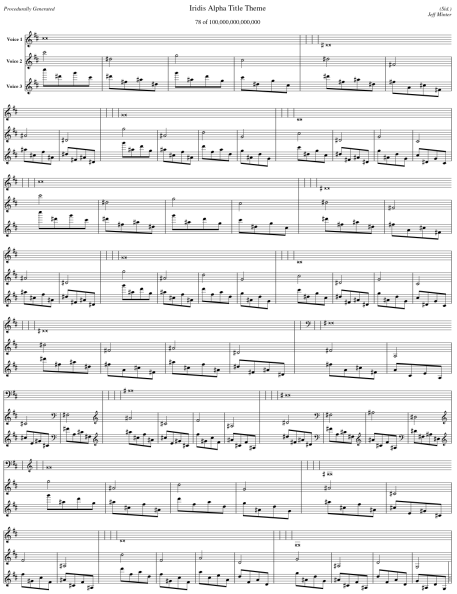

Procedurally Generated
Iridis Alpha Title Theme
73 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

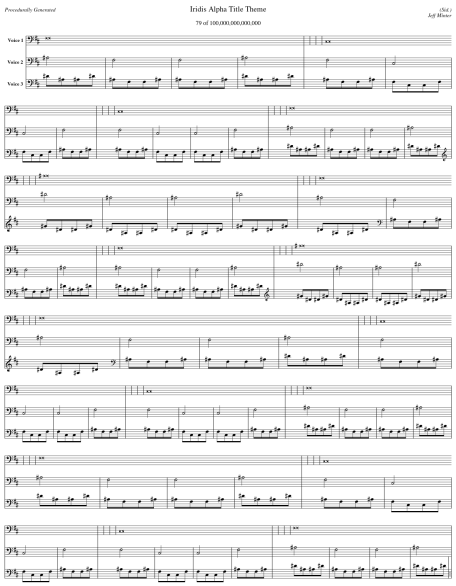

Procedurally Generated
Iridis Alpha Title Theme
74 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

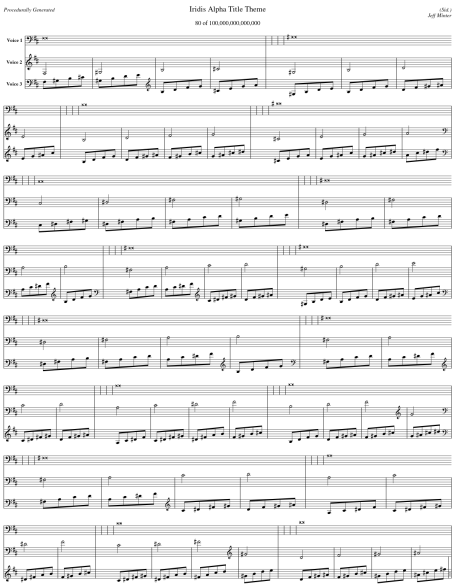

Procedurally Generated
Iridis Alpha Title Theme
75 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3


Procedurally Generated
Iridis Alpha Title Theme
76 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3


Procedurally Generated
Iridis Alpha Title Theme
77 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3


Procedurally Generated
Iridis Alpha Title Theme
78 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3


Procedurally Generated
Iridis Alpha Title Theme
79 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3


Procedurally Generated
Iridis Alpha Title Theme
80 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3