IRIDIS ALPHA THEORY

ROB HOGAN

Contents

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.

```
;--------------------------------------------------------------------
PlayTitleScreenMusic
        DEC baseNoteDuration
        BEQ MaybeStartNewTune
        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 playing
        ; 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 four
        ; 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 titleMusicNoteArray.
        LDX voice3IndexToMusicNoteArray
        LDA titleMusicNoteArray,X
```

Listing 1.1: Routine responsible for playing the title tune.

1.0.1 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':

```
        ; 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
```

Listing 1.2: 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 1.1: 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:

```
                        ;       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
```

Listing 1.3: 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.

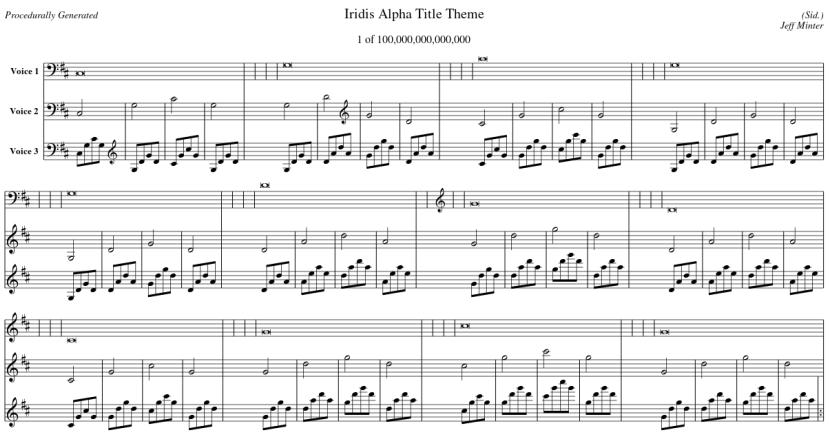
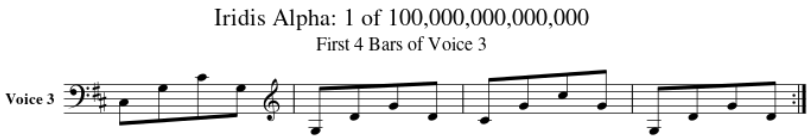


Figure 1.2: The first title tune in Iridis Alpha.

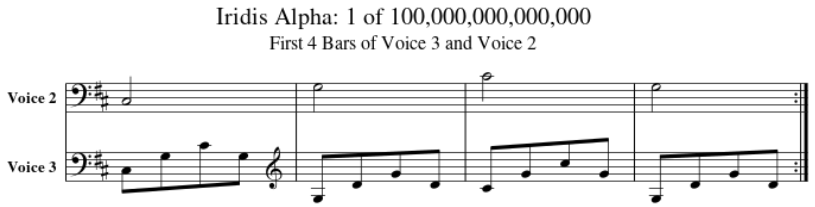
Structure

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

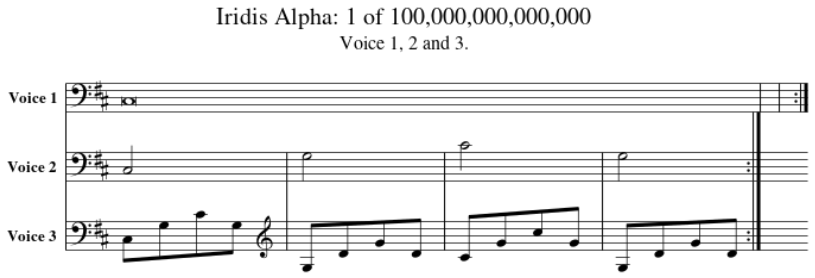
Voice 3 carries the main melody.



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:

```
        BNE  MaybePlayVoice1

        JSR  SelectNewNotesToPlay

MaybePlayVoice1
        DEC  voice1NoteDuration
        BNE  MaybePlayVoice2
```

Listing 1.4: `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).

```
        TXA
        AND #$03
        STA voice1IndexToMusicNoteArray

MaybePlayVoice2
        DEC voice2NoteDuration
```

Listing 1.5: `MaybePlayVoice2` part of `PlayTitleScreenMusic`.

```
        TAY
        JSR PlayNoteVoice2
        INX
        TXA
        AND #$03
```

Listing 1.6: `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 1.7: 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:

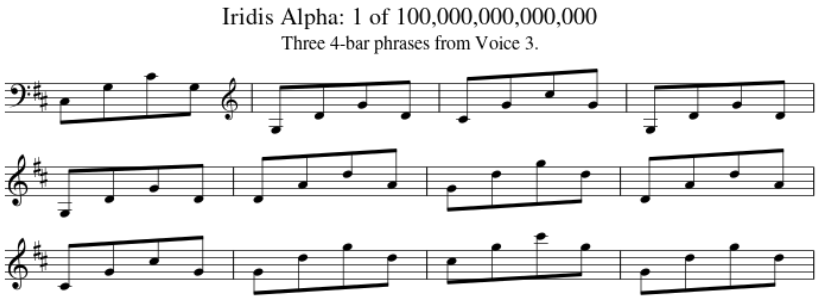


Figure 1.3: Bars 2 and 4 are always repeated

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.

```
; 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.
```

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

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

```
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 PlayNoteVoice3

        ; Move voice3IndexToMusicNoteArray to the next
        ; position in titleMusicNoteArray.
        INX
        TXA
        ; Since it's only 4 bytes long ensure we wrap
```

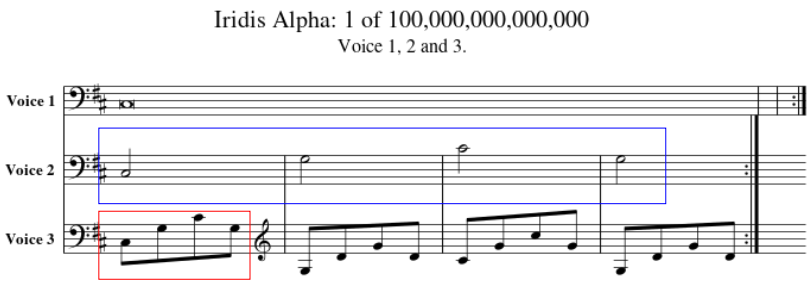
```
        STA $D408    ;Voice 2: Frequency Control - High-Byte
        RTS

;-------------------------------------------------------------
; PlayNoteVoice3
;-------------------------------------------------------------
PlayNoteVoice3
        LDA #$21
        STA $D412    ;Voice 3: Control Register
        LDA titleMusicLowBytes,Y
        STA $D40E    ;Voice 3: Frequency Control - Low-Byte
```

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.

```
        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 PlayNoteVoice2
```

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.

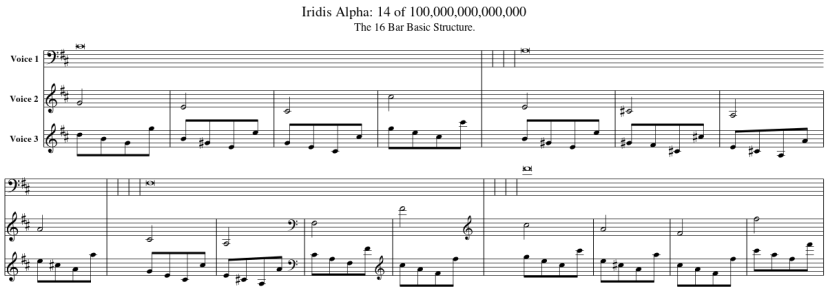


Figure 1.5: A full 16 bar passage showing the nested structure of Voices 1 and 2

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!

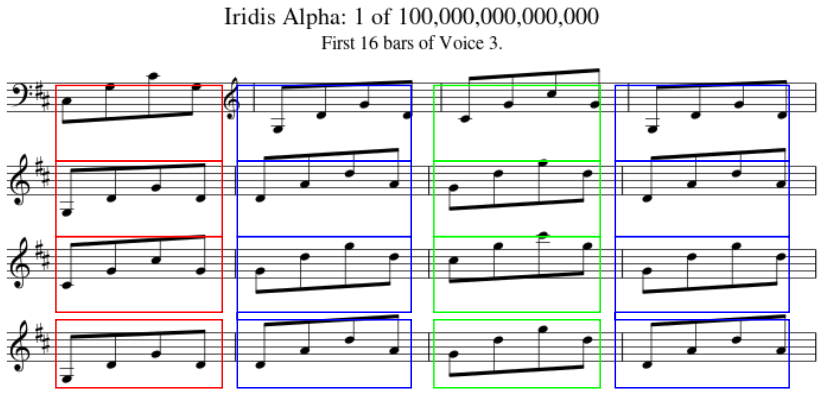


Figure 1.6: The G3-C4-G4-C4 pattern used to construct the 4 note pattern is also used to construct the key changes in each 4-bar sequence (red-blue-green-blue).

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!

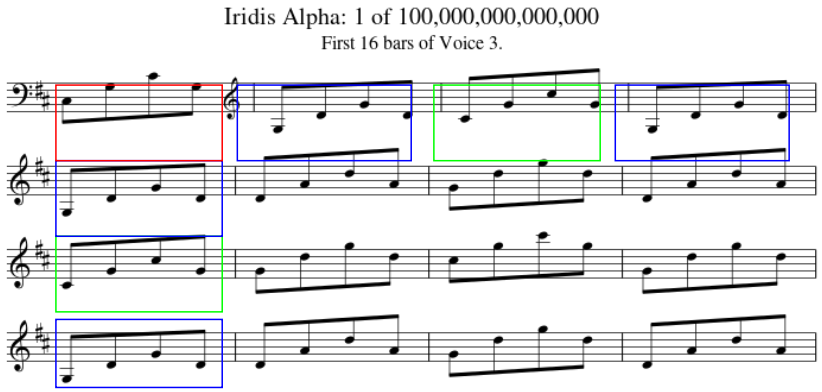
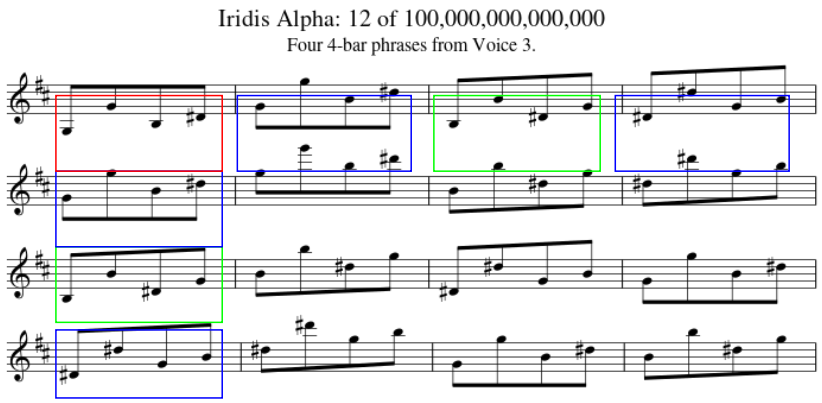


Figure 1.7: The start of each 4 bar pattern in a 16 bar cycle uses each of the 4-note patterns from the first 4 bars.

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



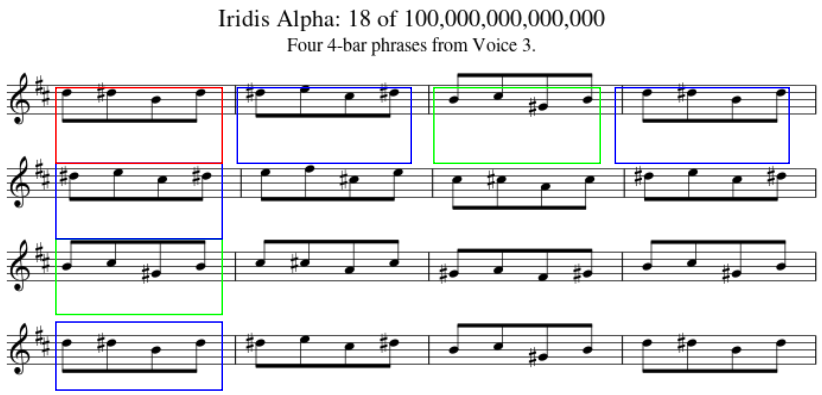


Figure 1.8: The same patterns in Tunes 12 and 18.

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`.

```
                        .BYTE $07,$08,$04,$07
                        .BYTE $00,$04,$07,$0E
                        .BYTE $00,$00,$00,$07
                        .BYTE $07,$04,$00,$0C
                        .BYTE $04,$07,$00,$0C
                        .BYTE $07,$08,$0A,$08
                        .BYTE $0C,$00,$0C,$03
                        .BYTE $0C,$03,$07,$00
;------------------------------------------------------------
; SelectNewNotesToPlay
;------------------------------------------------------------
SelectNewNotesToPlay
        ; Get a random value between 0 and 15.
        JSR PutProceduralByteInAccumulator
        AND #$0F
        ; Jump to InitializeSeedLoop if it's zero.
```

```
        BEQ InitializeSeedLoop

        ; Otherwise multiply it by 4. We do this so that
```

Listing 1.8: 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.

```
MultiplyRandomNumBy4
        CLC
        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 1.9: Use our seed value to pull 4 bytes from `titleMusicSeedArray` and store them in `titleMusicNoteArray`

```
; This is used to replenish titleMusicNoteArray with seed valuse
; for the procedurally generated title screen music.
titleMusicSeedArray .BYTE $00,$03,$06,$08
                    .BYTE $00,$0C,$04,$08
                    .BYTE $00,$07,$00,$05
                    .BYTE $05,$00,$00,$05
                    .BYTE $00,$06,$09,$05
                    .BYTE $02,$04,$03,$04
                    .BYTE $03,$07,$03,$00
                    .BYTE $04,$08,$0C,$09
                    .BYTE $07,$08,$04,$07
                    .BYTE $00,$04,$07,$0E
                    .BYTE $00,$00,$00,$07
                    .BYTE $07,$04,$00,$0C
                    .BYTE $04,$07,$00,$0C
                    .BYTE $07,$08,$0A,$08
                    .BYTE $0C,$00,$0C,$03
                    .BYTE $0C,$03,$07,$00
```

Listing 1.10: 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`.

```
         ;F7 pressed?
b168F    CMP #$03
         BNE ReturnFromCheckInput
         LDA #$04
         STA f7PressedOrTimedOutToAttractMode
         STA unusedVariable2
         RTS

txtEasy    .TEXT "EASY"
txtHard    .TEXT "UGH!"
;--------------------------------------------------
; PutProceduralByteInAccumulator
; This function is self-modifying. Every time it
; is called it increments the address that
```

Listing 1.11: 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 1.12: 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:

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

Listing 1.13: `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`:

```
        LDX notesPlayedSinceLastKeyChange
        LDA titleMusicNoteArray,X
        STA offsetForNextVoice1Note
```

Listing 1.14: `offsetForNextVoice1Note` is loaded from `titleMusicNoteArray` and is propagated down to `offsetForNextVoice3Note`.

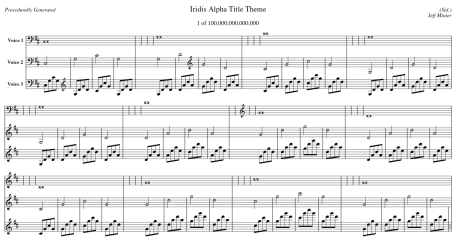
In practice there are 16 possible voice note sequences and 12 unique possible byte values to load from `titleMusicSeedArray` so there are 192 possible tunes.

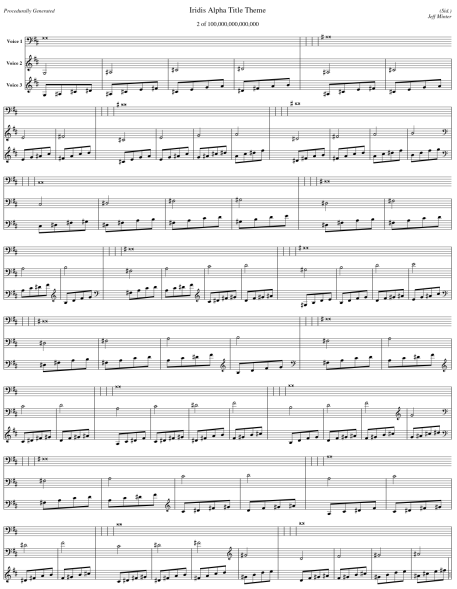
[I can only get it to generate 80 so I'm missing something here.]

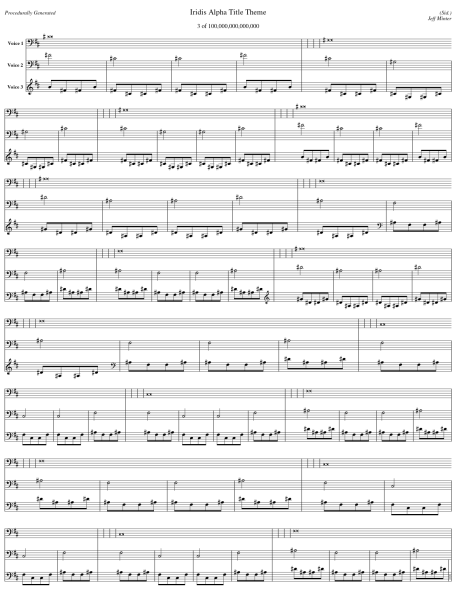
So the Hundred Thousand Billion is a lie. Iridis will indeed play a hundred thousand billion times if you leave it running long enough but ultimately even when we account for variations in key it can only ever play 192 unique title tunes.

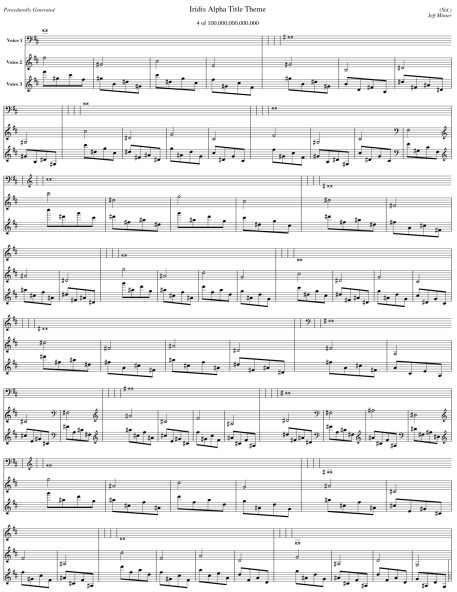
Sorry for getting your hopes up.

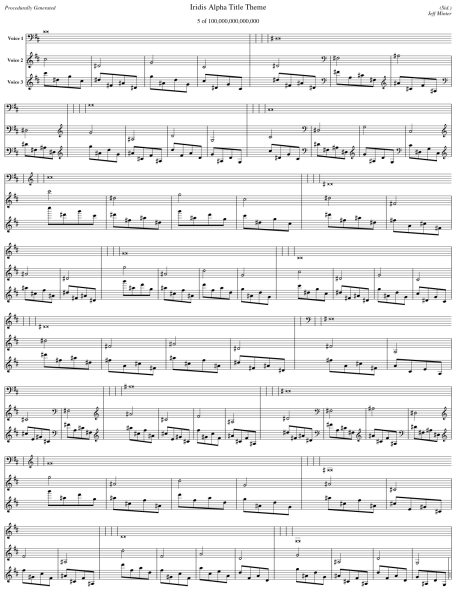
Appendix: 18/100,000,000,000,000 Theme Tunes

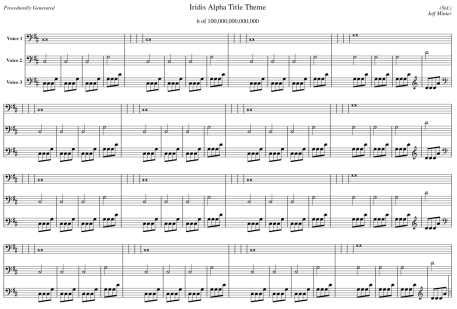


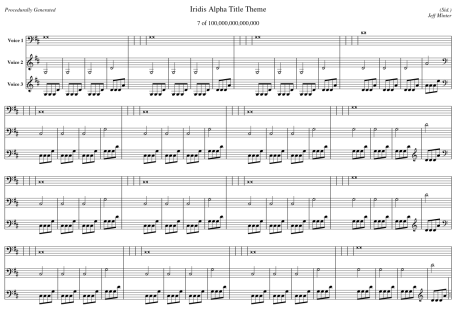


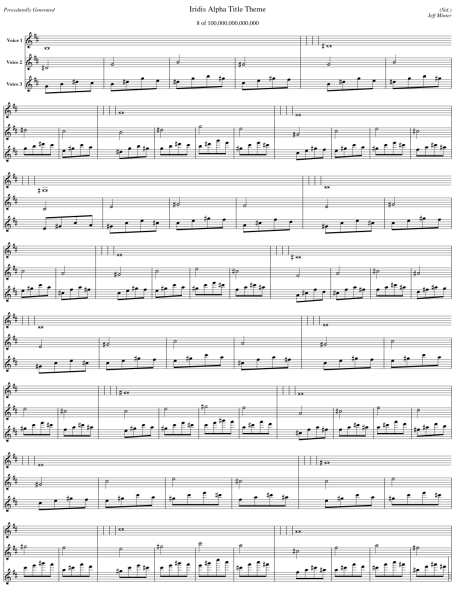

Procedurally Generated
Iridis Alpha Title Theme
3 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

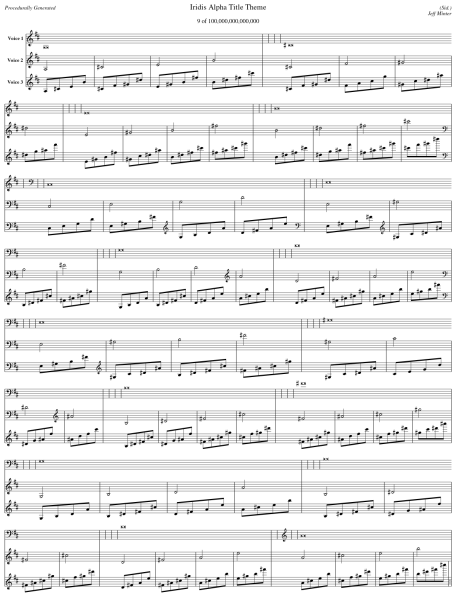

Procedurally Generated
Iridis Alpha Title Theme
4 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

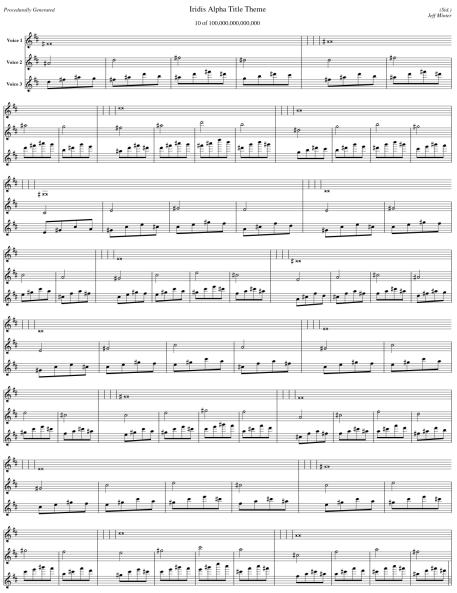

Procedurally Generated
Iridis Alpha Title Theme
5 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

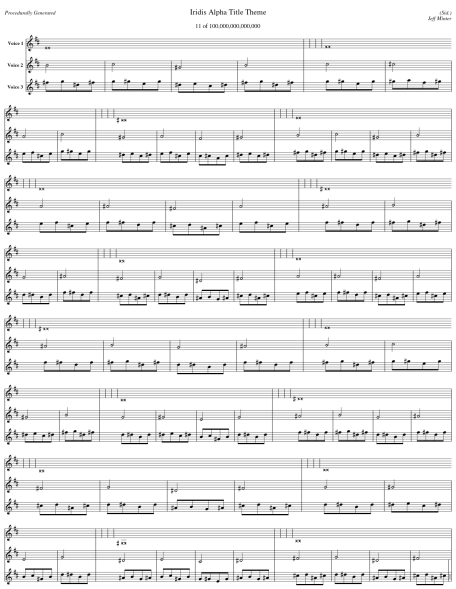

Procedurally Generated
Iridis Alpha Title Theme
6 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

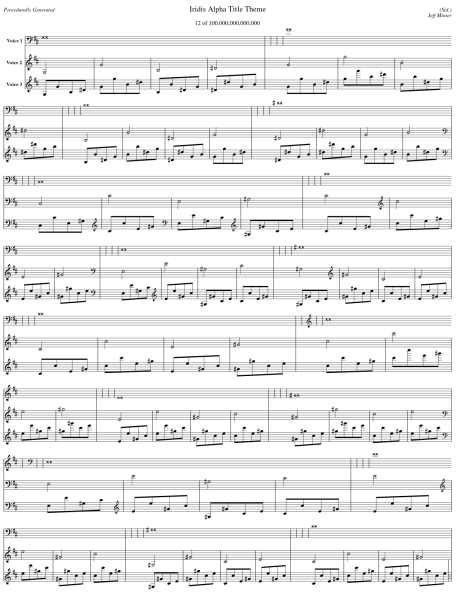

Procedurally Generated
Iridis Alpha Title Theme
7 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

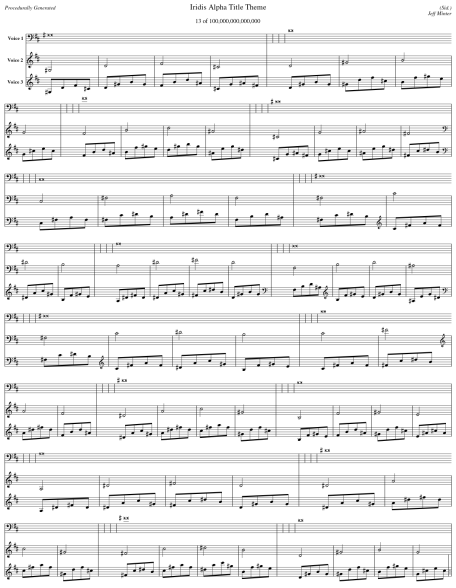

Procedurally Generated
Iridis Alpha Title Theme
8 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

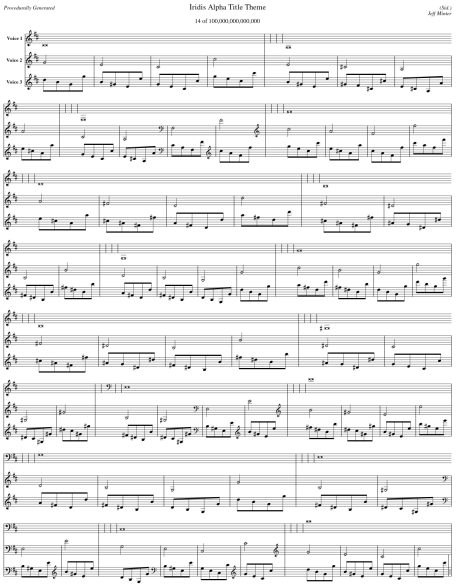

Procedurally Generated
Iridis Alpha Title Theme
9 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

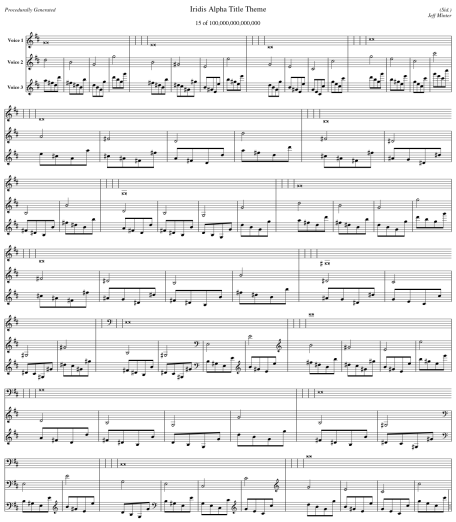

Procedurally Generated
Iridis Alpha Title Theme
10 of 100,000,000,000,000
(Std.)
Jeff Mosier
Voice 1
Voice 2
Voice 3

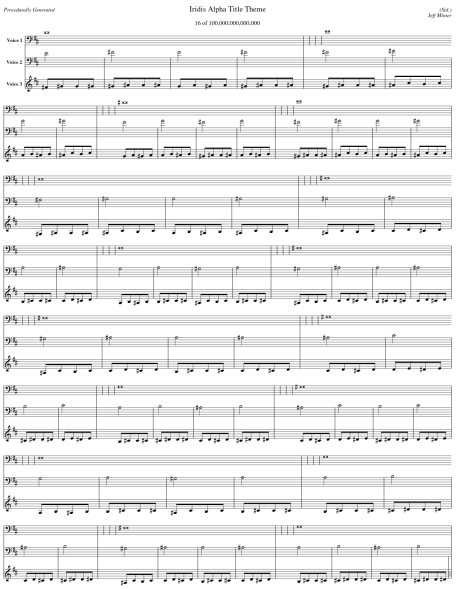

Procedurally Generated
Iridis Alpha Title Theme
11 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

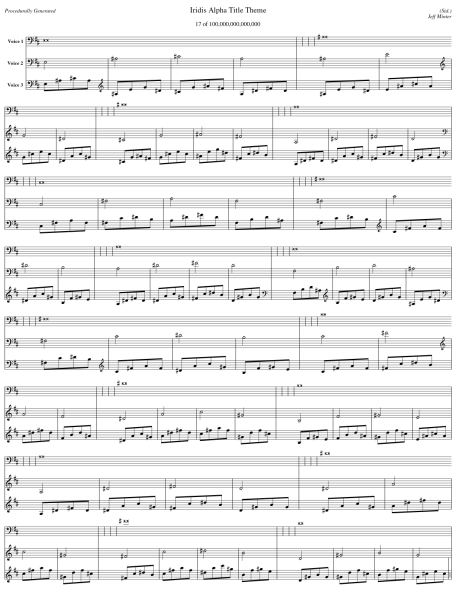

Procedurally Generated
Iridis Alpha Title Theme
12 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

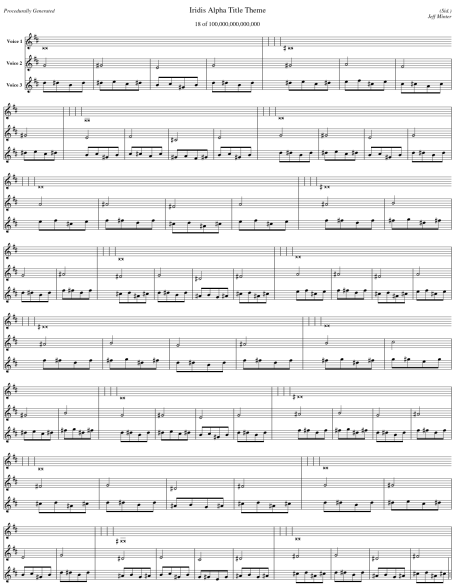

Procedurally Generated
Iridis Alpha Title Theme
13 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

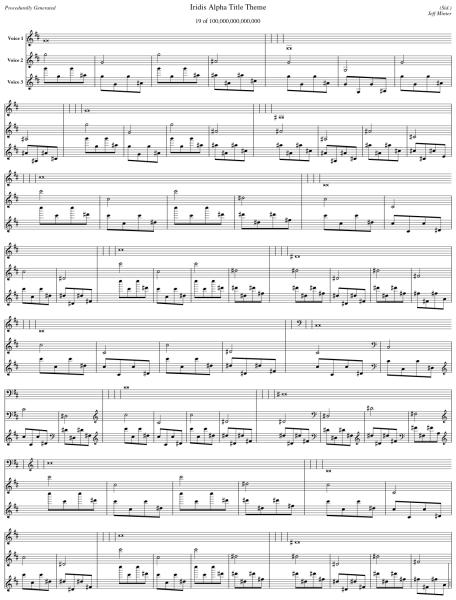

Procedurally Generated
Iridis Alpha Title Theme
14 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

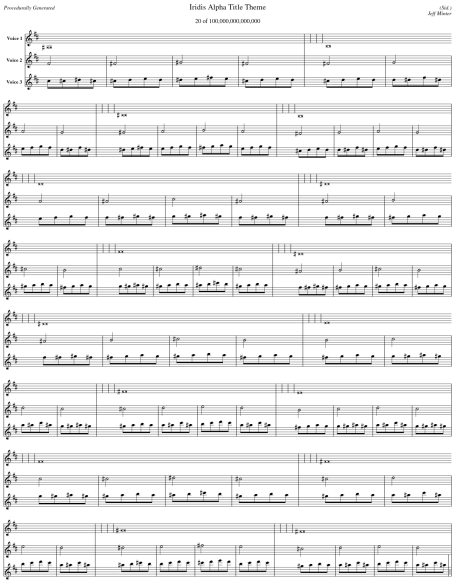

Procedurally Generated
Iridis Alpha Title Theme
15 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

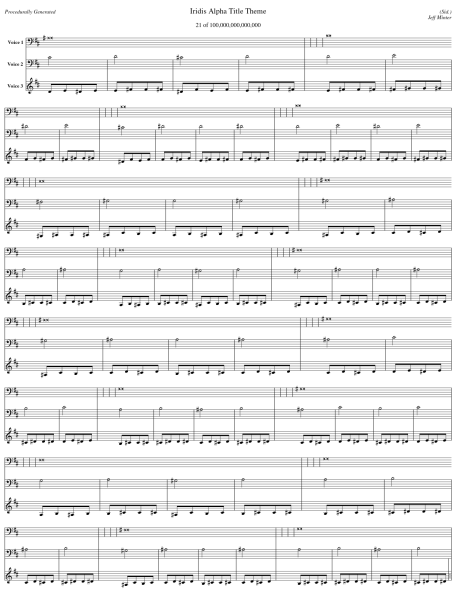

Procedurally Generated
Iridis Alpha Title Theme
16 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

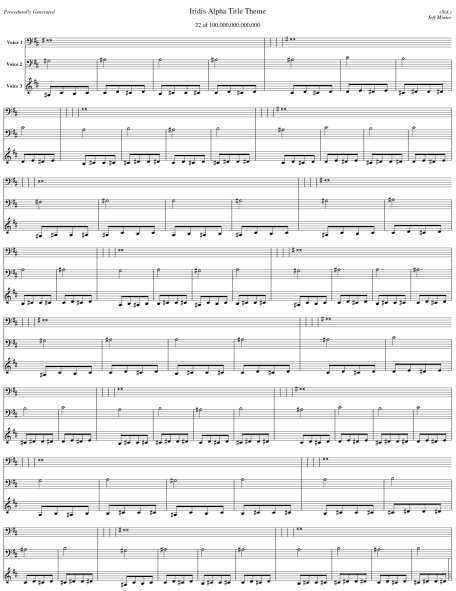

Procedurally Generated
Iridis Alpha Title Theme
17 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

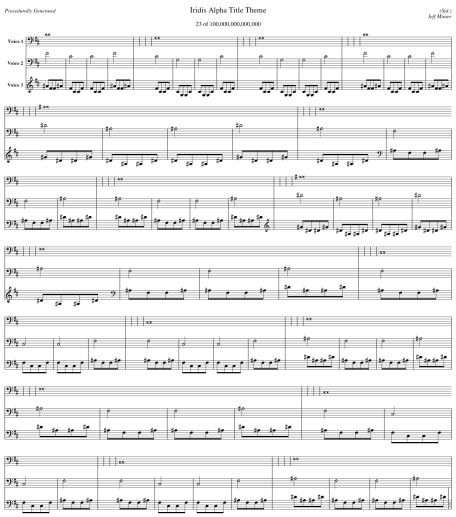

Procedurally Generated
Iridis Alpha Title Theme
18 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

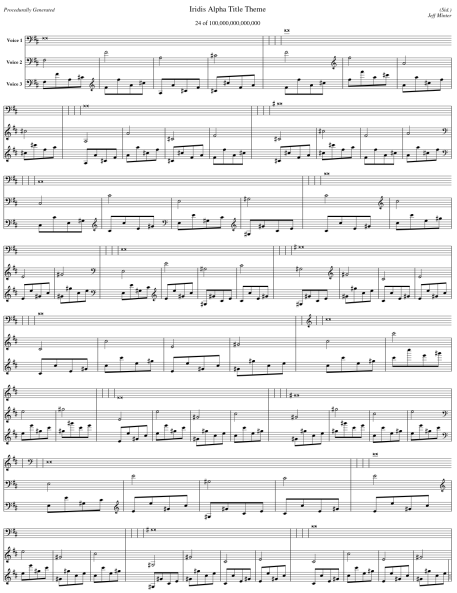

Procedurally Generated
Iridis Alpha Title Theme
19 of 100,000,000,000,000
(Std.)
Jeff Munce
Voice 1
Voice 2
Voice 3

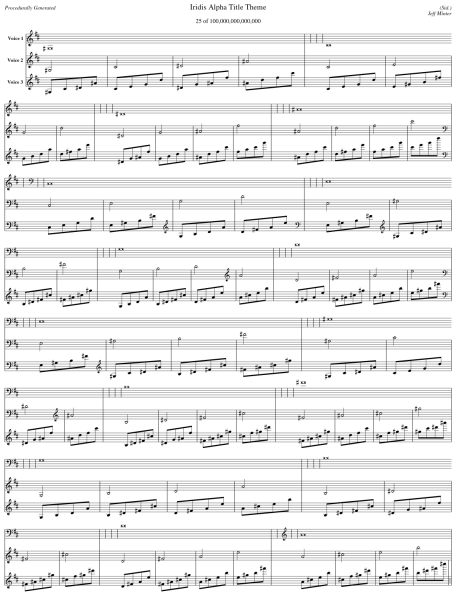

Procedurally Generated
Iridis Alpha Title Theme
20 of 100,000,000,000,000
(Std.)
Jeff Munce
Voice 1
Voice 2
Voice 3

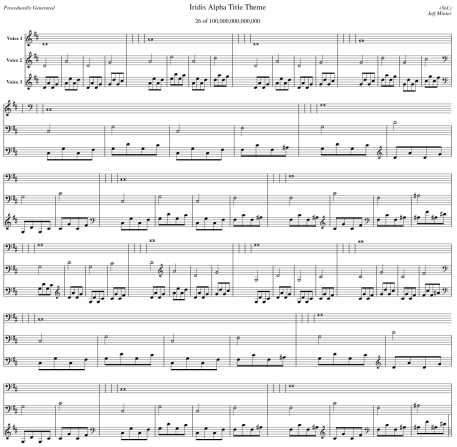

Procedurally Generated
Iridis Alpha Title Theme
21 of 100,000,000,000,000
(Std.)
Jeff Munce
Voice 1
Voice 2
Voice 3

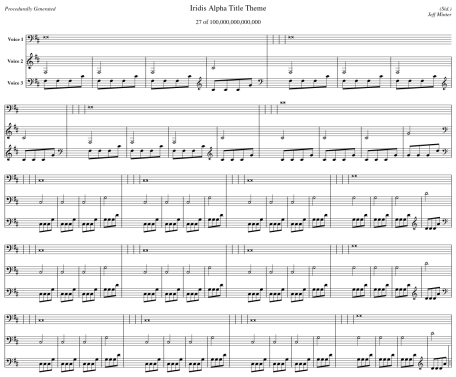

Procedurally Generated
Iridis Alpha Title Theme
22 of 100,000,000,000,000
(Std.)
Jeff Munce
Voice 1
Voice 2
Voice 3

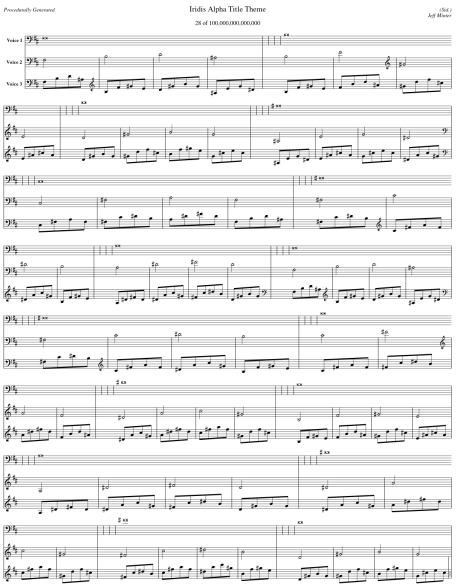

Procedurally Generated
Iridis Alpha Title Theme
23 of 100,000,000,000,000
(Sol.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

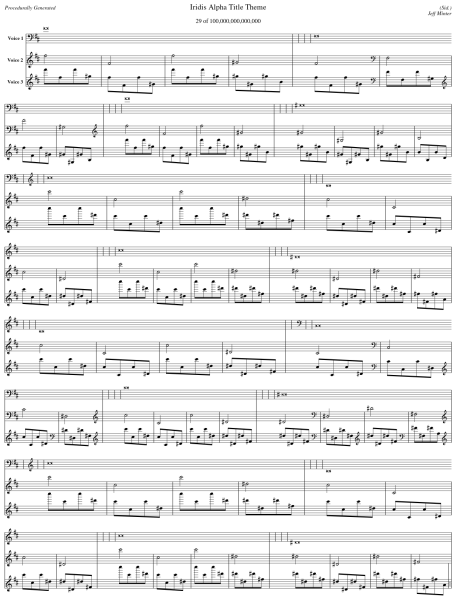

Procedurally Generated
Iridis Alpha Title Theme
24 of 100,000,000,000,000
(Sol.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

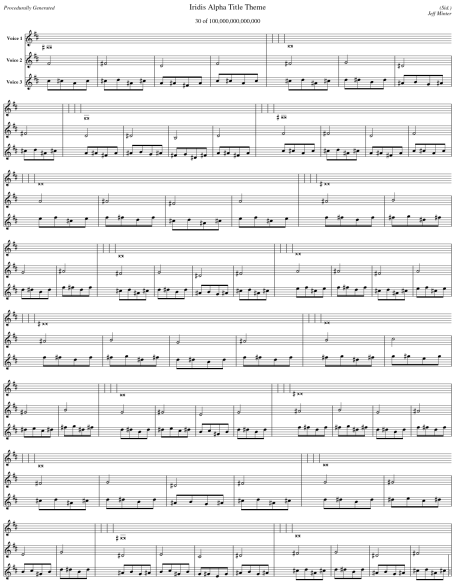

Procedurally Generated
Iridis Alpha Title Theme
25 of 100,000,000,000,000
(Sol.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

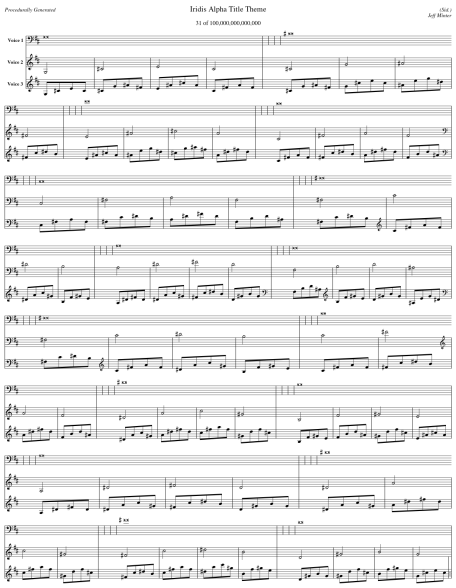

Procedurally Generated
Iridis Alpha Title Theme
26 of 100,000,000,000,000
(Sol.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

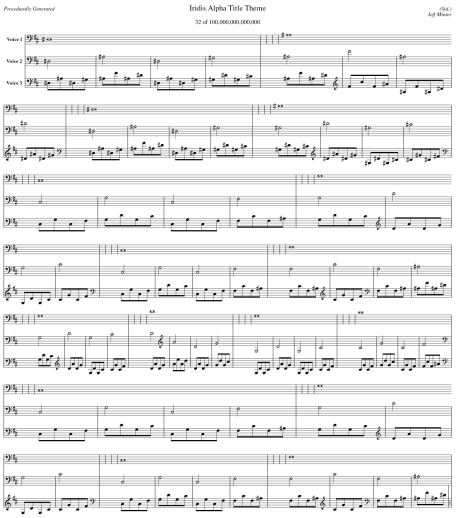

Procedurally Generated
Iridis Alpha Title Theme
27 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3

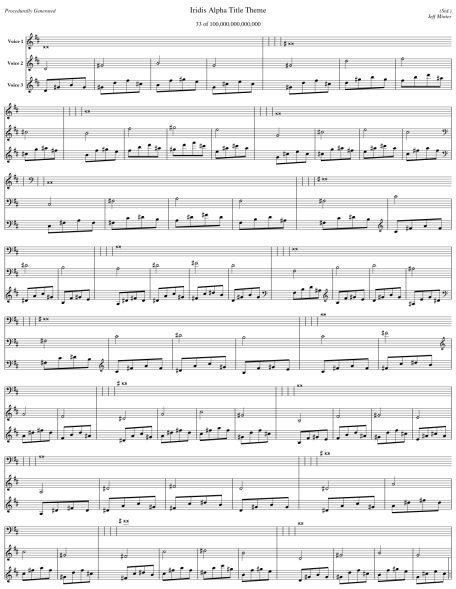

Procedurally Generated
Iridis Alpha Title Theme
28 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3

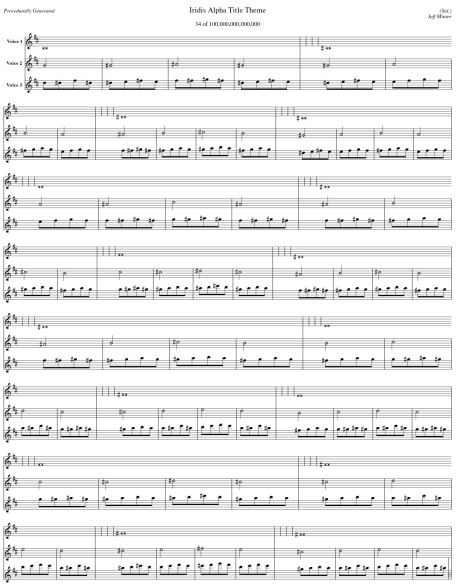

Procedurally Generated
Iridis Alpha Title Theme
29 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3

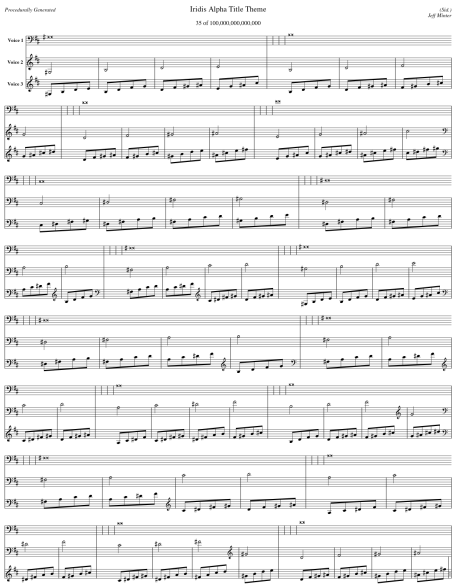

Procedurally Generated
Iridis Alpha Title Theme
30 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3

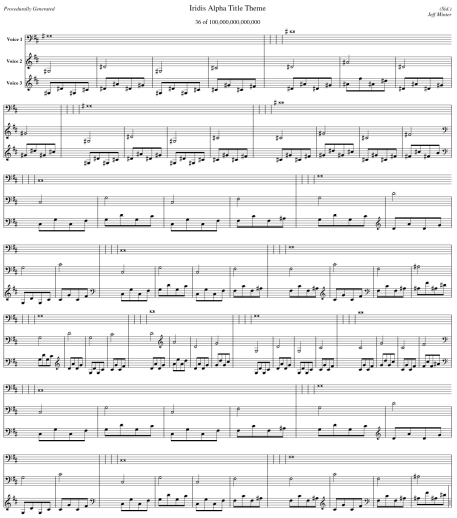

Procedurally Generated
Iridis Alpha Title Theme
31 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

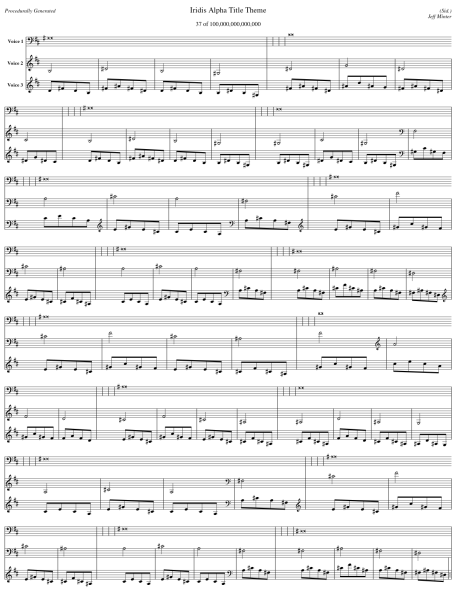

Procedurally Generated
Iridis Alpha Title Theme
32 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

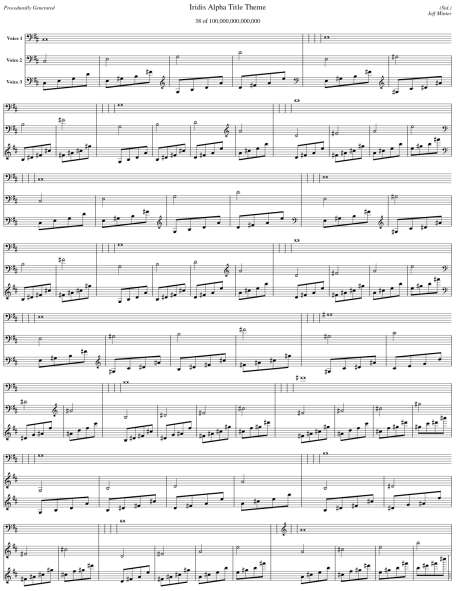

Procedurally Generated
Iridis Alpha Title Theme
33 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

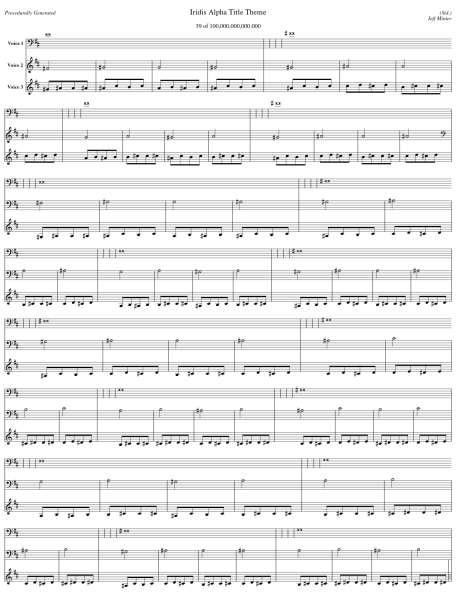

Procedurally Generated
Iridis Alpha Title Theme
34 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

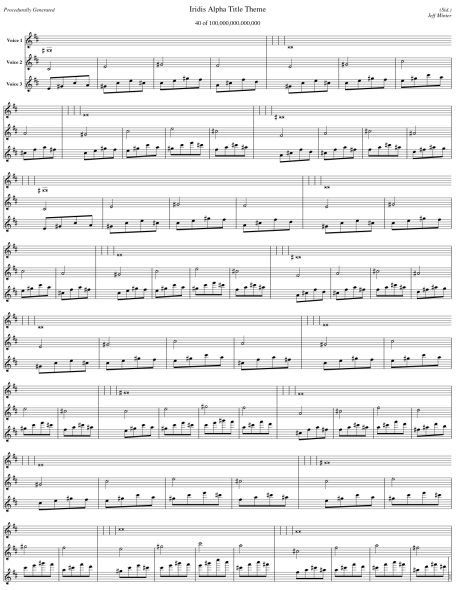

Procedurally Generated
Iridis Alpha Title Theme
35 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

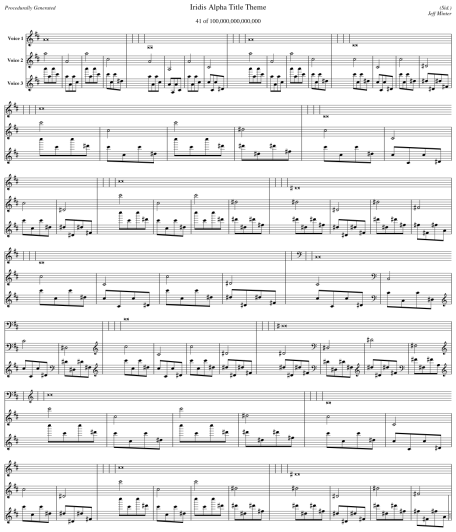

Procedurally Generated
Iridis Alpha Title Theme
36 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

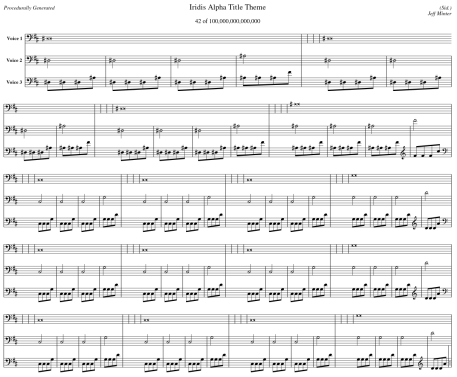

Procedurally Generated
Iridis Alpha Title Theme
37 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

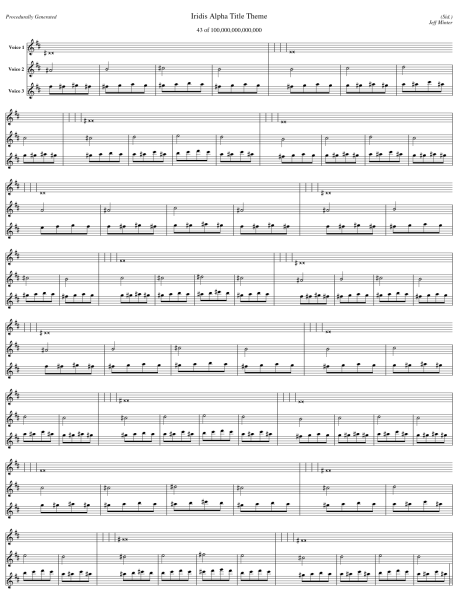

Procedurally Generated
Iridis Alpha Title Theme
38 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

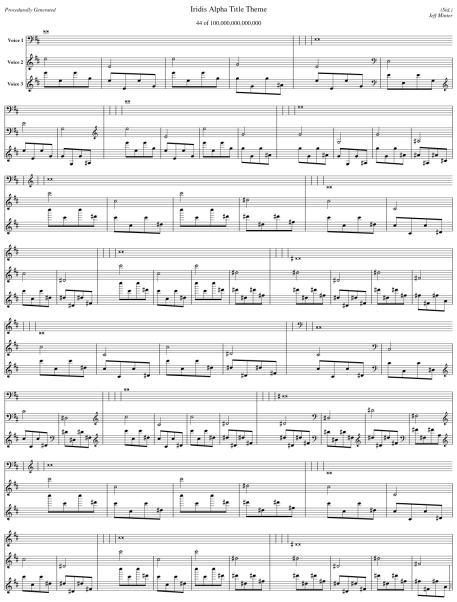

Procedurally Generated
Iridis Alpha Title Theme
39 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

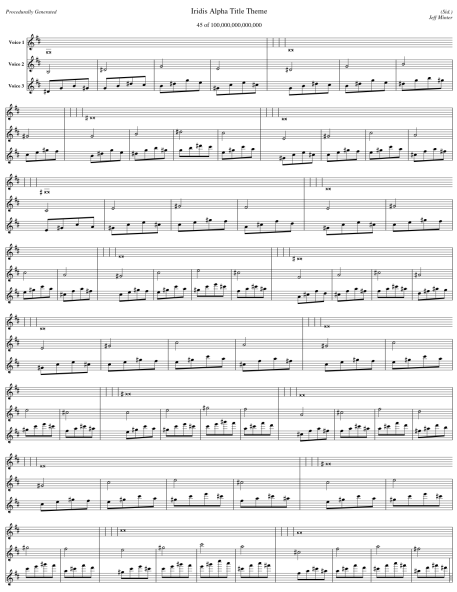

Procedurally Generated
Iridis Alpha Title Theme
40 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

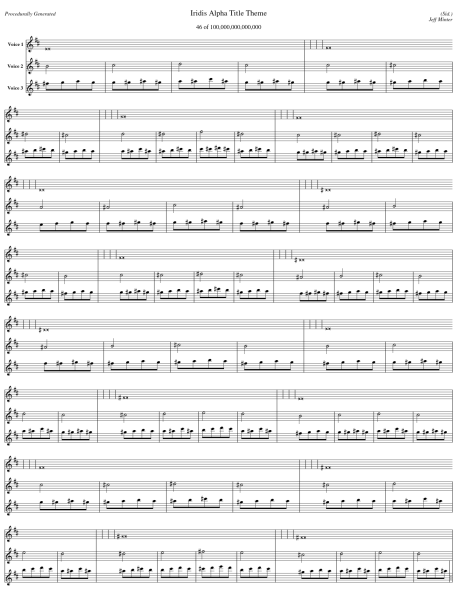

Procedurally Generated
Iridis Alpha Title Theme
41 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

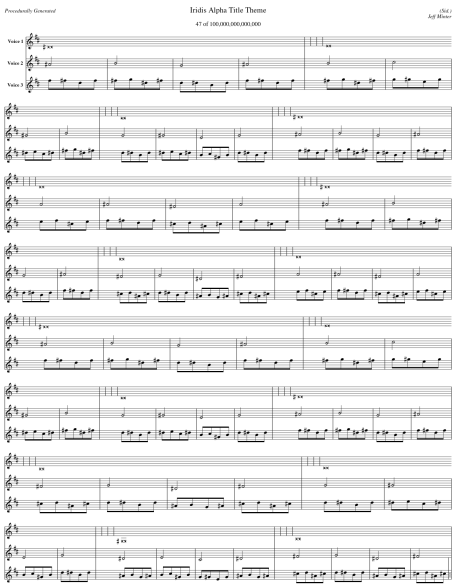

Procedurally Generated
Iridis Alpha Title Theme
42 of 100,000,000,000,000
(Std.)
Jeff Muntz
Voice 1
Voice 2
Voice 3

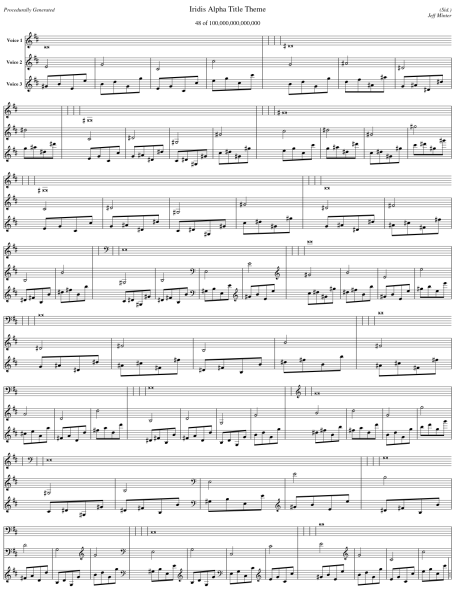

Procedurally Generated
Iridis Alpha Title Theme
43 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

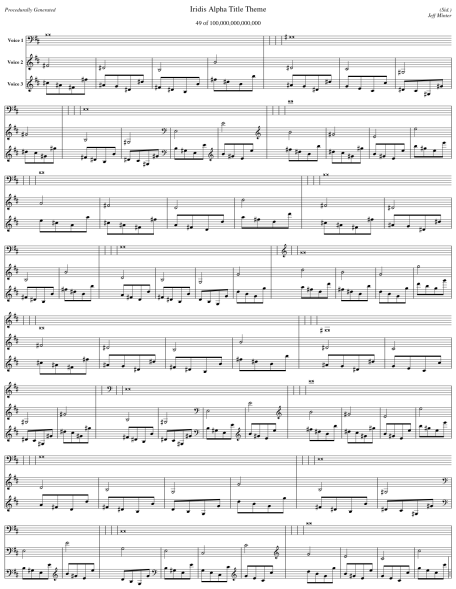

Procedurally Generated
Iridis Alpha Title Theme
44 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

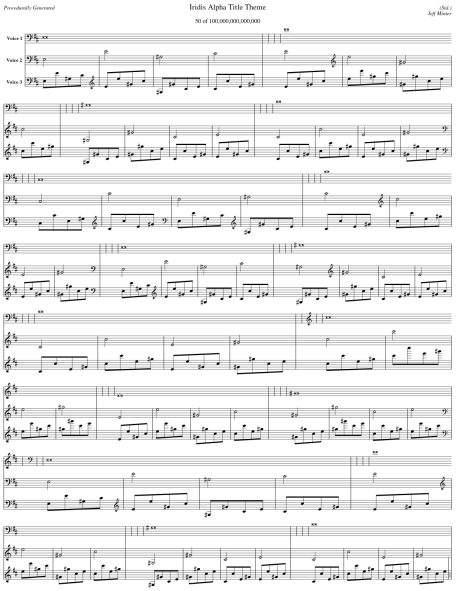

Procedurally Generated
Iridis Alpha Title Theme
45 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

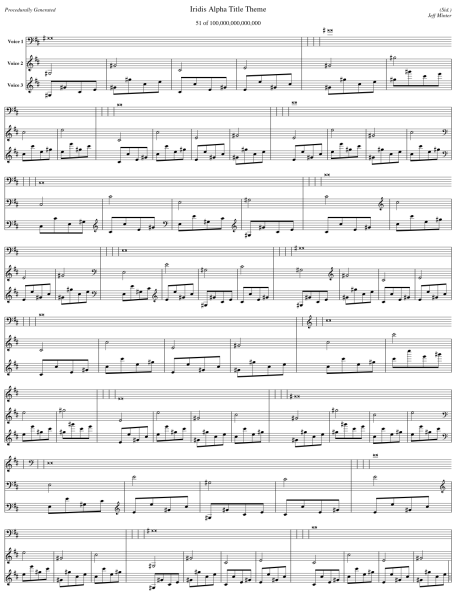

Procedurally Generated
Iridis Alpha Title Theme
46 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

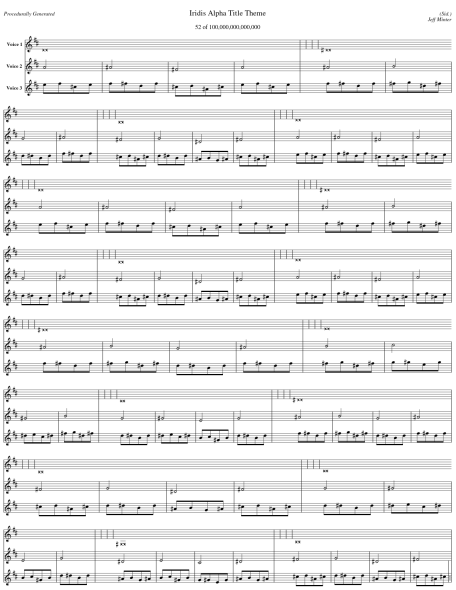

Procedurally Generated
Iridis Alpha Title Theme
47 of 100,000,000,000,000
(Std.)
Jeff Manor
Voice 1
Voice 2
Voice 3

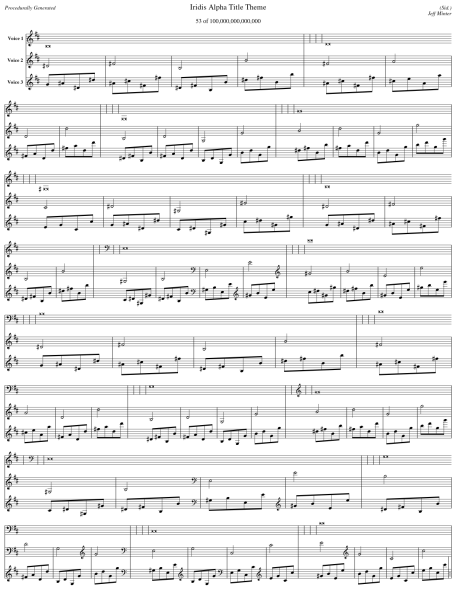

Procedurally Generated
Iridis Alpha Title Theme
48 of 100,000,000,000,000
(Std.)
Jeff Manor
Voice 1
Voice 2
Voice 3

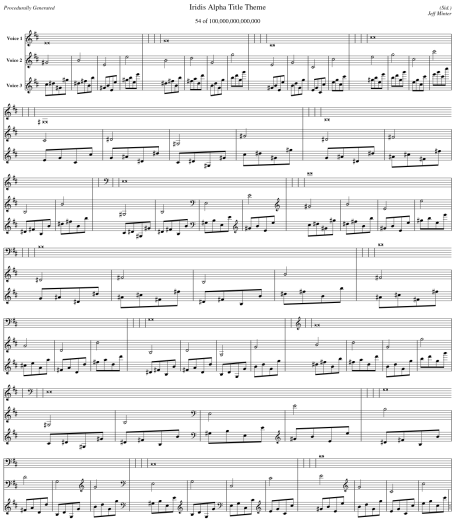

Procedurally Generated
Iridis Alpha Title Theme
49 of 100,000,000,000,000
(Std.)
Jeff Manor
Voice 1
Voice 2
Voice 3

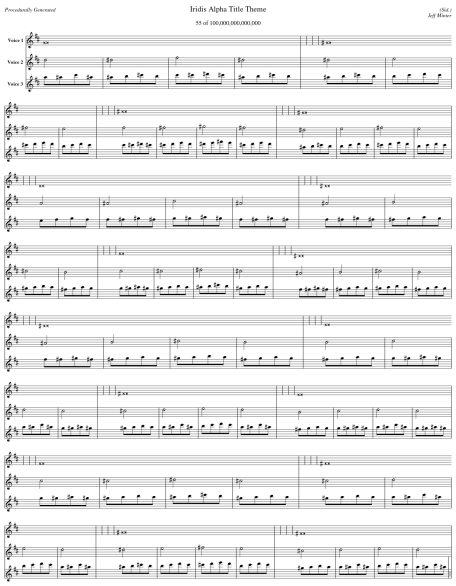

Procedurally Generated
Iridis Alpha Title Theme
50 of 100,000,000,000,000
(Std.)
Jeff Manor
Voice 1
Voice 2
Voice 3

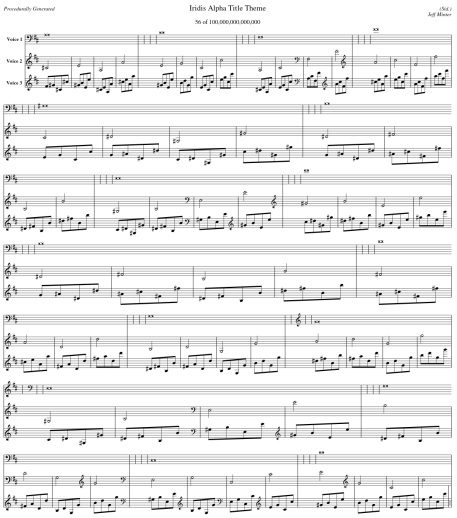

Procedurally Generated
Iridis Alpha Title Theme
51 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

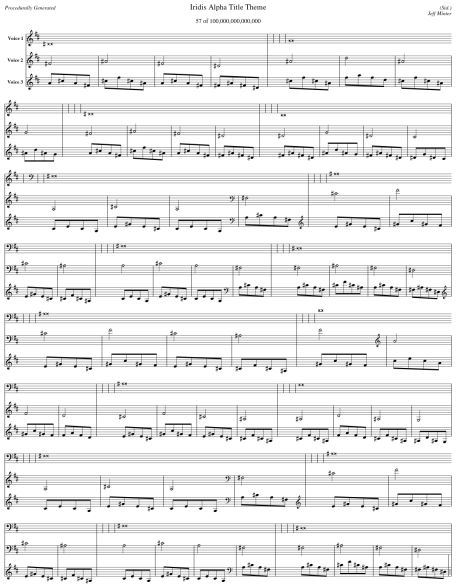

Procedurally Generated
Iridis Alpha Title Theme
52 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

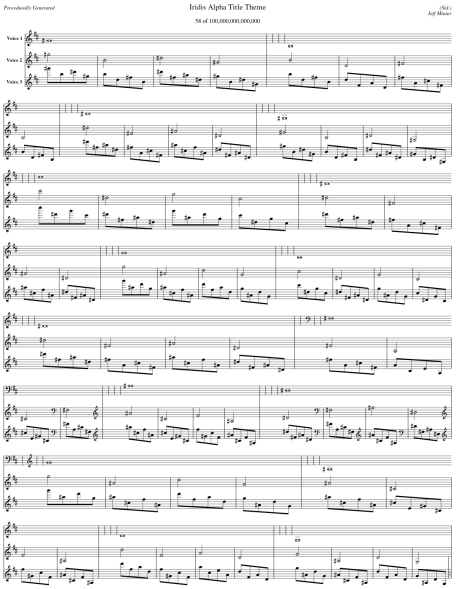

Procedurally Generated
Iridis Alpha Title Theme
53 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

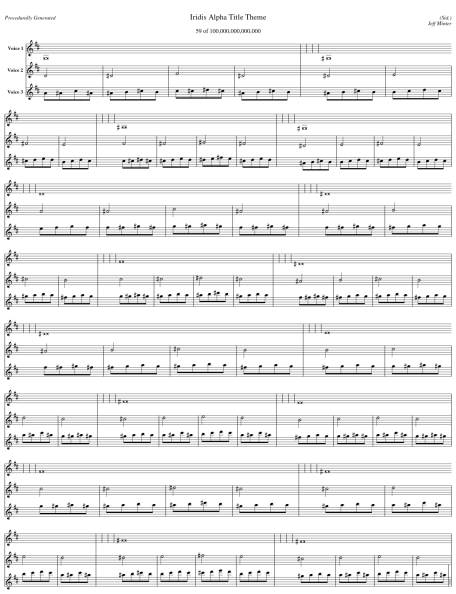

Procedurally Generated
Iridis Alpha Title Theme
54 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

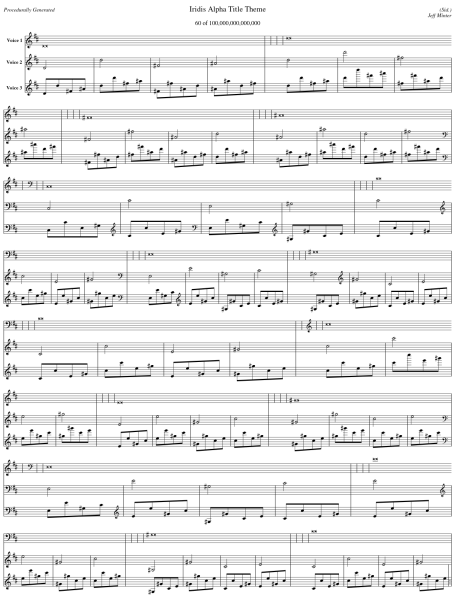

Procedurally Generated
Iridis Alpha Title Theme
55 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

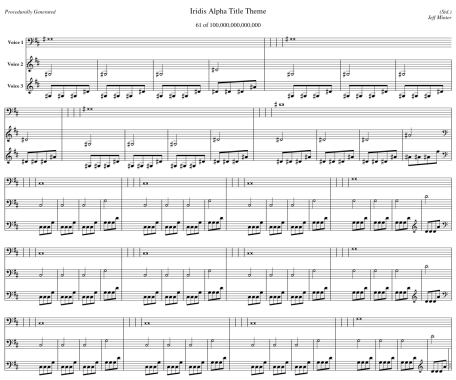

Procedurally Generated
Iridis Alpha Title Theme
56 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

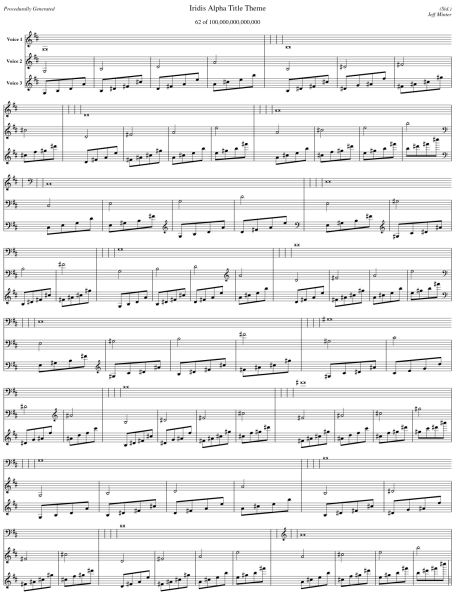

Procedurally Generated
Iridis Alpha Title Theme
57 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

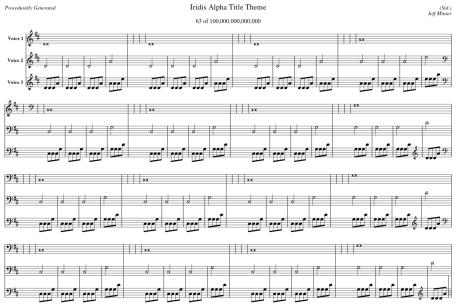

Procedurally Generated
Iridis Alpha Title Theme
58 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

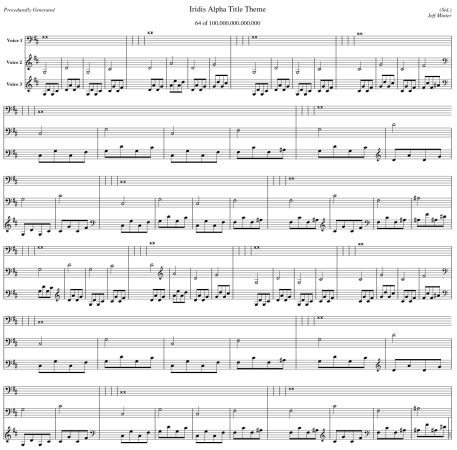

Procedurally Generated
Iridis Alpha Title Theme
59 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

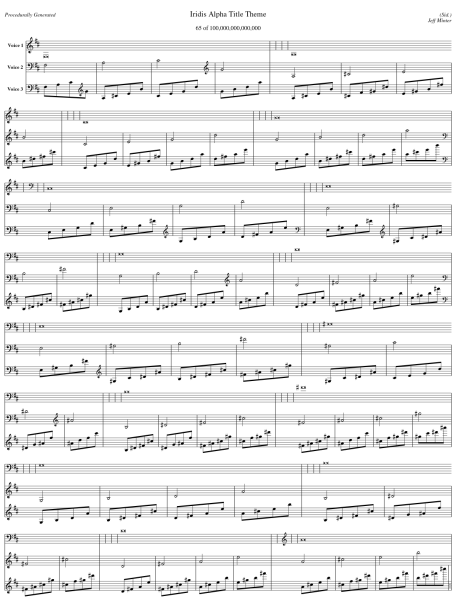

Procedurally Generated
Iridis Alpha Title Theme
60 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

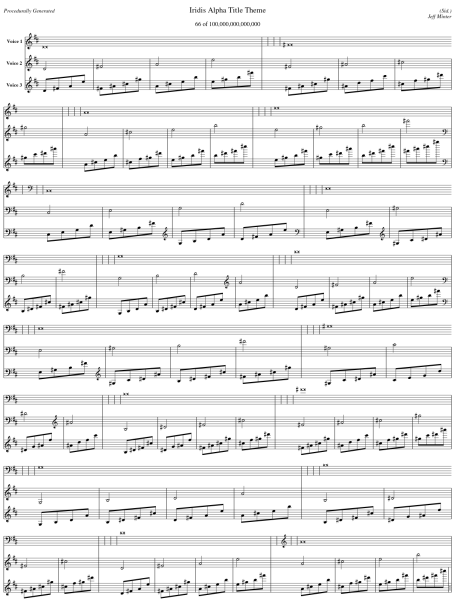

Procedurally Generated
Iridis Alpha Title Theme
61 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

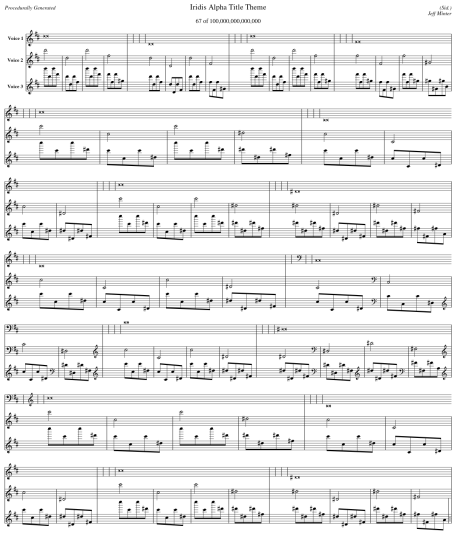

Procedurally Generated
Iridis Alpha Title Theme
62 of 100,000,000,000,000
(Sol.)
Jeff Munro
Voice 1
Voice 2
Voice 3

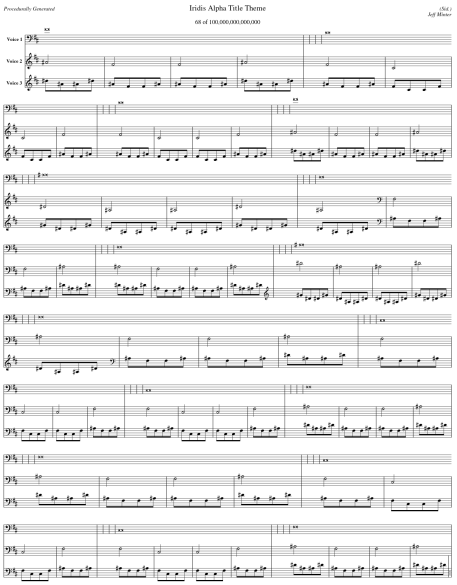

Procedurally Generated
Iridis Alpha Title Theme
63 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

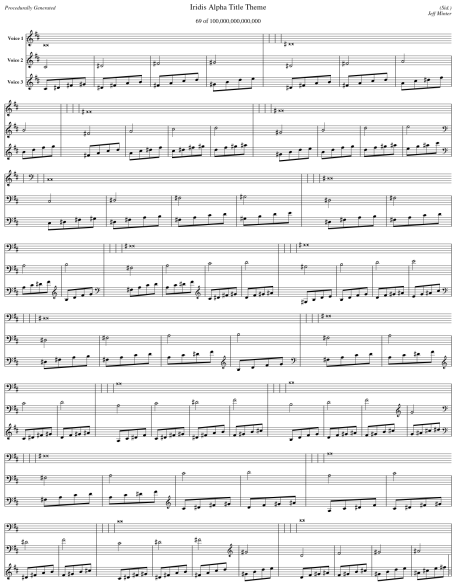

Procedurally Generated
Iridis Alpha Title Theme
64 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

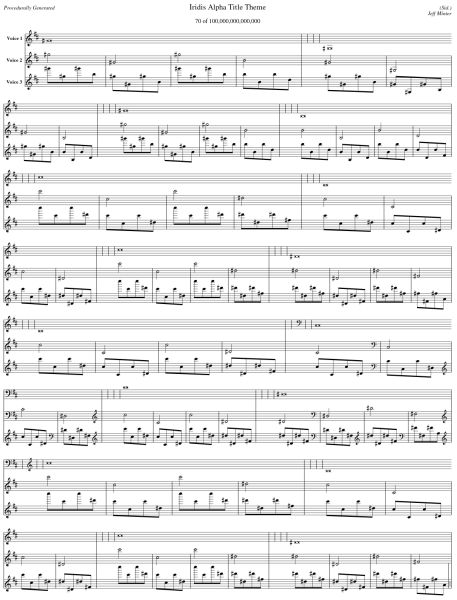

Procedurally Generated
Iridis Alpha Title Theme
65 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

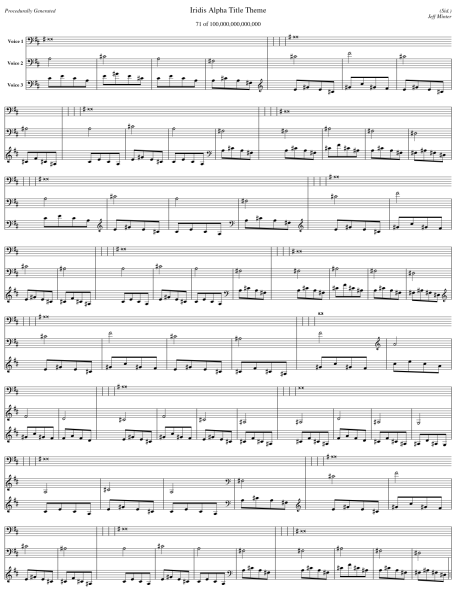

Procedurally Generated
Iridis Alpha Title Theme
66 of 100,000,000,000,000
(Sol.)
Jeff Moser
Voice 1
Voice 2
Voice 3

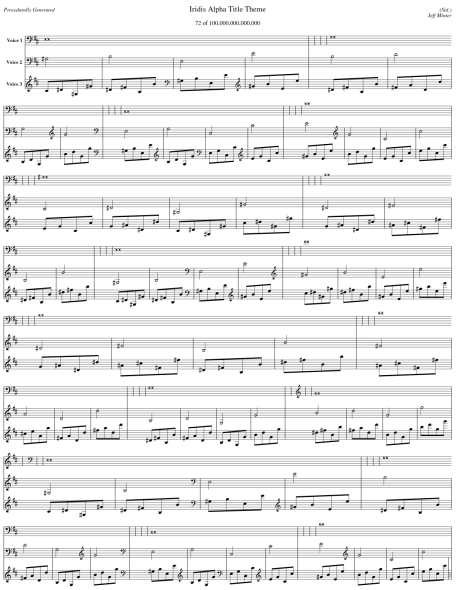

Procedurally Generated
Iridis Alpha Title Theme
67 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

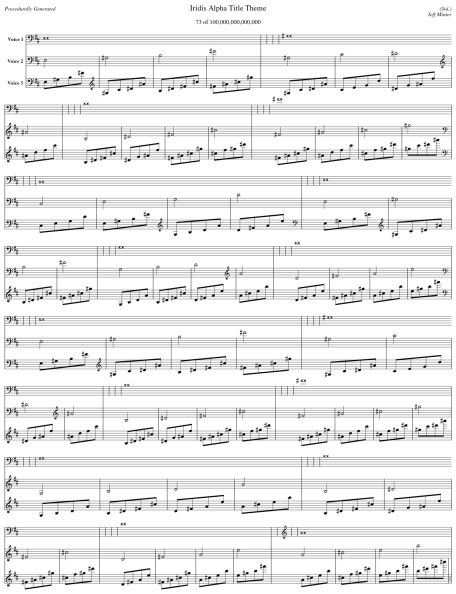

Procedurally Generated
Iridis Alpha Title Theme
68 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

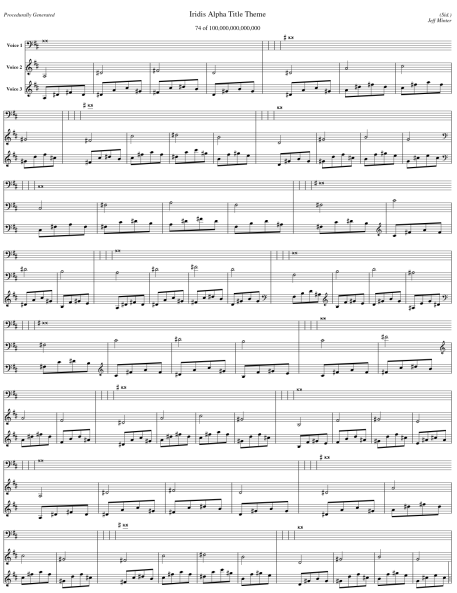

Procedurally Generated
Iridis Alpha Title Theme
69 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

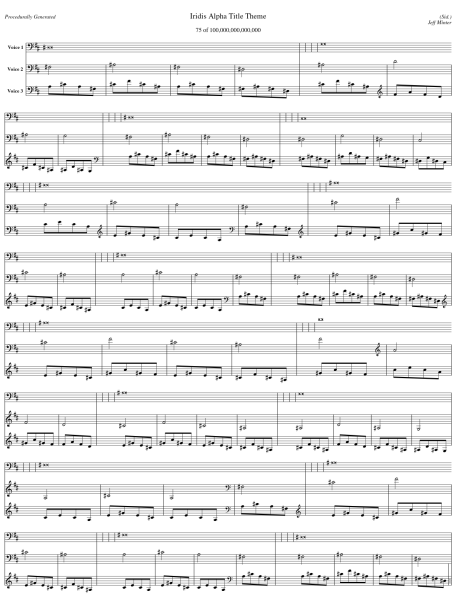

Procedurally Generated
Iridis Alpha Title Theme
70 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

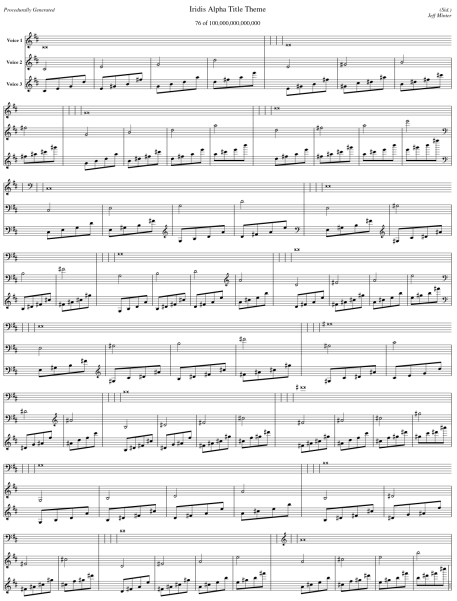

Procedurally Generated
Iridis Alpha Title Theme
71 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

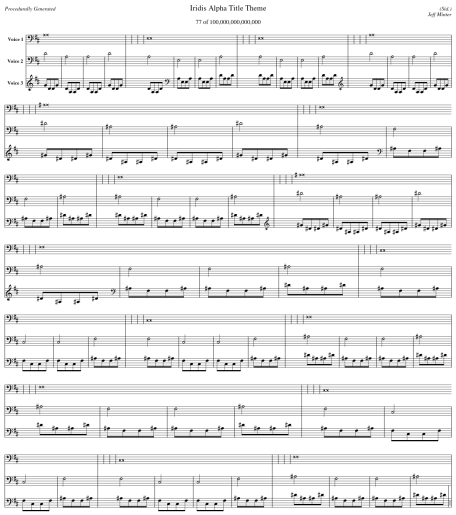

Procedurally Generated
Iridis Alpha Title Theme
72 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

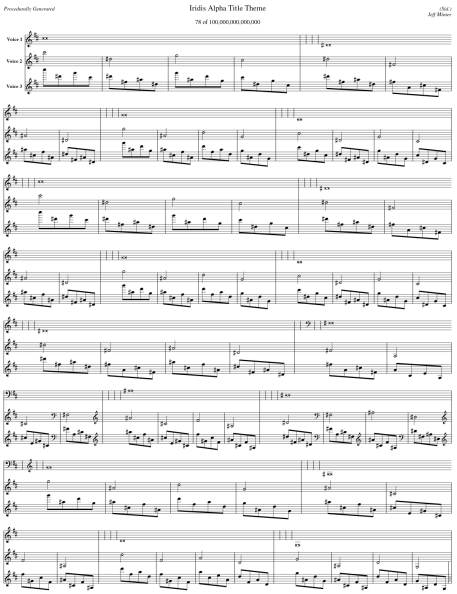

Procedurally Generated
Iridis Alpha Title Theme
73 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

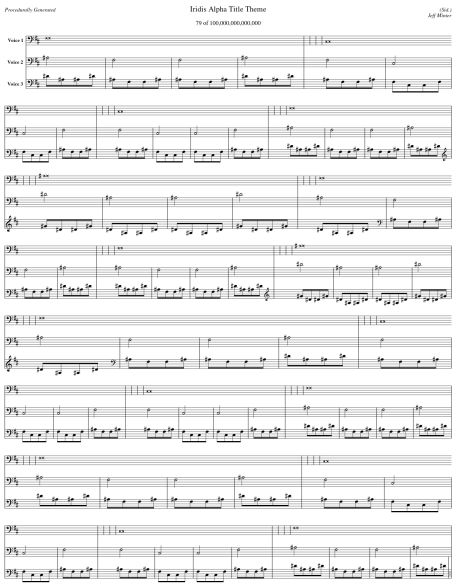

Procedurally Generated
Iridis Alpha Title Theme
74 of 100,000,000,000,000
(Std.)
Jeff Moser
Voice 1
Voice 2
Voice 3

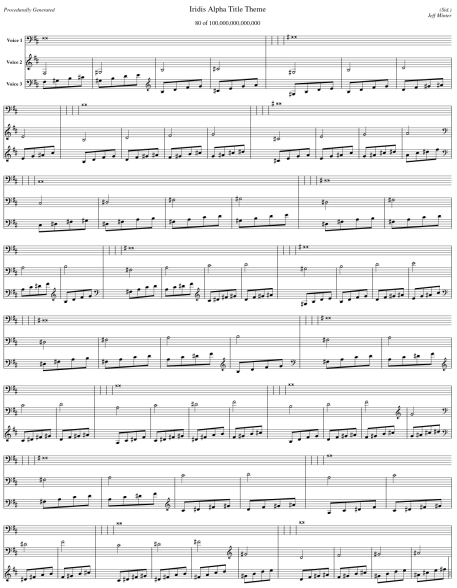

Procedurally Generated
Iridis Alpha Title Theme
75 of 100,000,000,000,000
(Std.)
Jeff Munro
Voice 1
Voice 2
Voice 3


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