Sooooo … sometimes you just need to refactor some old code. Just a little bit. These past 2 years or so, as I've studied different programming languages, I've had this feeling to, well … unscramble the code in the Pipe Dream project from 2019. You know, the "I could totally do this so much better now" feeling. So that brings you and I to today. In these next few videos I'll be documenting my process for rewriting *ALL* of the code that went into the original Roblox project! Comparing and contrasting between code when I am able, and overall just explaining my process.

Now what this project *ISN'T*:

I'm not planning to make any alterations to the 3D model itself, or add in any new features like 'sound' or 'custom midi editors'. These features would be projects in their own right, and honestly Roblox isn't made to do those kind of things.

Now regardless of all that, I'm still very fond of the Lua language itself. Back in 2000-whatever it was my first ever programming language, and still to this day it's sort of a "mother tongue" if you will. And while I'm not exactly sure \*what\* Roblox is turning into these days, 5 years down the line, I figured it wouldn't hurt to do ooone last project now that I'm no longer 16.

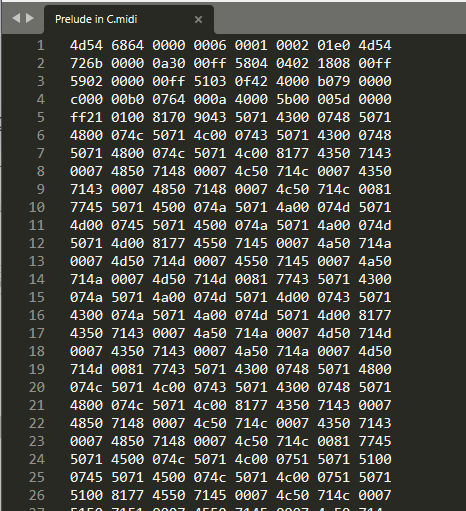
[ <https://github.com/zstolfi/Pipe-Dream-2019> ]

Here I have a sample of the original Lua code pulled up, now hosted on Github as well.

Basically a few years ago I humbly announced that the code for entire project was up and open for anyone wants a peek. A sort of "use at your own risk" kind of thing. But if anyone remembers trying to toy around with any of it, you probably remember getting a LOT of really unhelpful error messages. This was the biggest design flaw with my old code. Back then I sorta had a "just wing it" mentality to see if I could get anything up and running … Yeah that was very much a mistake. Don't skimp out on error messages kids. Everything was just sort of copy-pasted around anyways without much thought for future use. Overall there are many problems with it, which may or may not be the topic in future episodes.

For THIS episode though, I'll redo the first crucial step I took toward file input so many years ago. Coding the Base64 parser.

Essentially, how do I take any Midi file I'd like, and 'upload' it in such a way that my Lua scripts can read it. For this you'll need to know one thing: All files stored on your computer are just binary data. All of them. From the computer's point-of-view it has no notion of what even IS a Midi file.

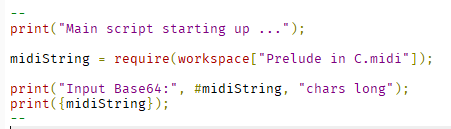


For example, here in an instance of Sublime Text, this is obvious. I drag an example midi file into it … and we see as expect, the file is nothing more than a sequence of numbers. Numbers in hexadecimal to be exact!

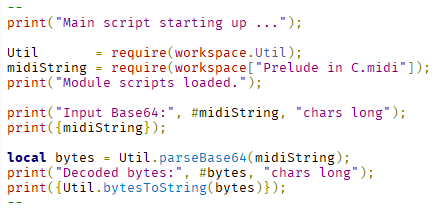
Now because, as far as I'm aware, there is no way to directly store binary data onto Roblox, we'll need a workaround. It would be nice to have our data accessible through a Lua string. String in this case meaning "sequence of bytes". Luckily, fitting arbitrary data into plain-old text is the inaugural challenge Base64 was designed solve. So the rest of this video will be dedicated to writing exactly that.

Well, to start off we're going to need some data, so I have on my computer a medium-size Midi file which I'll be using for the rest of this video. Using a tool I wrote a couple days ago I'll convert it into a Lua-ready Base64 string, and paste it directly into its own "ModuleScript".

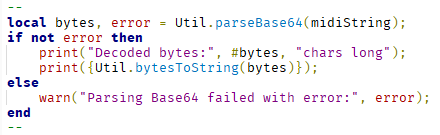
"ModuleScript"s are pretty much Roblox's way of keeping data/tables in Lua code. They're pretty analogous to "header files" in the C language. For this project I'll be using one regular "Script" object as the start of execution. The "main function" if you will.



Here the "require" keyword just tells the interpreter "This is some data I would like to retrieve".



Our main script should be nothing more than a simplified overview of 'what' we are accomplishing in our program. It's perfectly valid to sketch out the look your code before you even write out all the functions by the way.



Now, because it is possible for our program to encounter invalid input– in other words there are strings which aren't Base64– we account for a possible 'failed' parse. It's very easy for someone to, say, mistakenly delete a single character from the input string. It's up to our code to detect and report that error as early as possible.

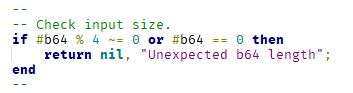
The comma before the "=" sign is syntax for a multi-return statement. Lua, unlike most programming languages, allows functions which return more than 1 value. It's a nifty little feature I'll be taking full advantage of next episode as well.

So, let's reorganize our files a bit before creating a new "utilities" ModuleScript to store our Base64 parser.



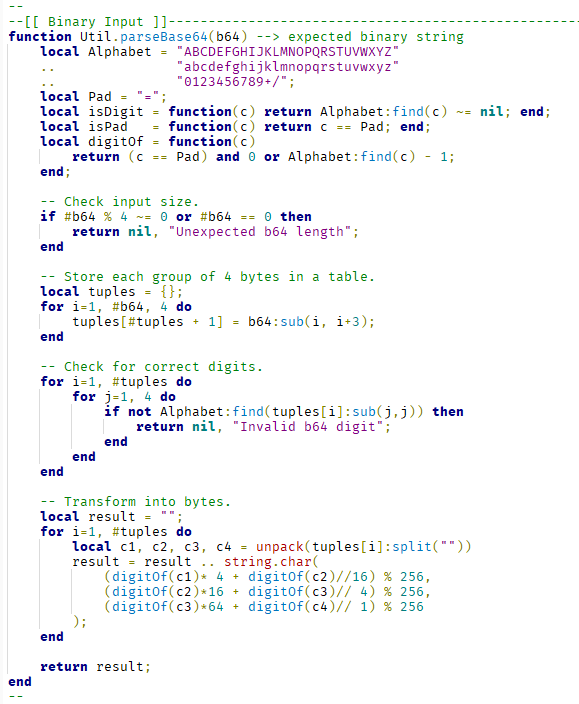
[Code auto-fills]

This is the real meat-and-potatoes of this lesson. At the top here I'm defining a couple constants. Even if our Base64 format is likely to never change, it's still good practice to separate 'data' from 'code' in our programs. Right after, to aid in legibility, there are a few helper-functions. Yes functions inside functions are valid Lua syntax, in fact, in this language functions are just variables like any other. In this case, even though the tasks they perform are very simple, it greatly improves code legibility when we name 'what' we're doing instead of only 'how' we're doing it.

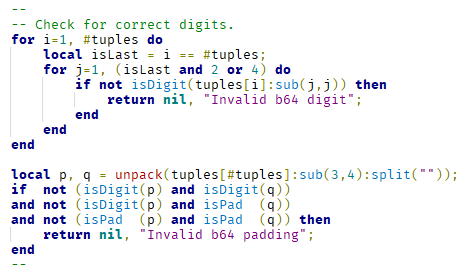


On line 17 we see our first example of the multi-return syntax in action. Remember how our main-script expected two values from our "parseBase64" function? This is exactly that. A single-value return is also allowed, as seen in the bottom of our code. Lua will automatically fill in the second return value with "nil", the Lua equivalent of "null".

Here's a simplified diagram from the Wikipedia article on "Base64". On the top we have 6 binary digits (bits for short) per Base64 character. For each tuple of 4 characters in our input, we want to reinterpret as 3 bytes, and store them all together in a string.



But because the Base64 standard is a little bit more complex than this, we'll have to modify our code just a little a bit to account for possible padded input at the end of the last tuple.



Now there's a curious looking syntax on this line right here. In programming, whenever there's a frequently occurring pattern in a language, we call it an "idiom". Here's an example of a Lua idiom that simply means "if 'isLast' is true, pretend we wrote a 2, otherwise pretend we wrote a 4". It's an almost equivalent emulation of C's "ternary operator", really just a fancy way of doing an if-statement. While it's always possible to write code with the same behavior without the "and-or" In my mind it's often valuable to allow brevity in our code.

Our complete function reads as follows:

"parseBase64" is a utility function which takes in base64 data and returns an expected string.

If our input is empty, or not a multiple of 4, we report an error to the user.

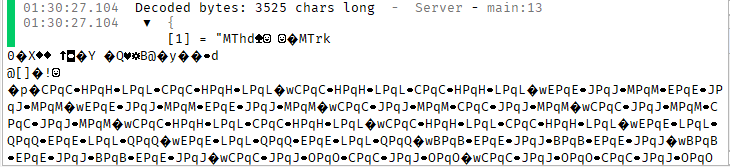
Otherwise, store every 4 Base64 digits into a table called "tuples".

For each digit in each tuple, if it's not a valid character where we expect it, return an error.

Otherwise, there are 3 valid cases for the end of a Base64 string. It's it's not the first one AND it's not the second, AND it's not the third, it's an error.

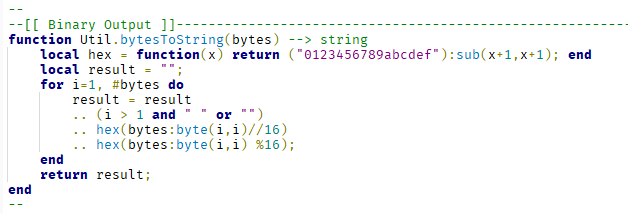
If all other checks have passed then we're ready to transform our data, and push each triplet of bytes onto our result string.

Let's see if our function does as we expect, shall we? So I'll hit F8 to run …

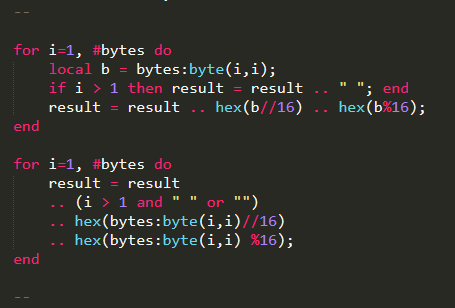


Hmm … well it seems to be correct. I do see a Midi header and track header in there. Everything else is less than readable this. This is because we gave Lua no specification on \*how\* to print our data, so it defaulted to displaying each byte as its equivalent ASCII character, more-or-less.

Let's help it out a little bit with another utility function.

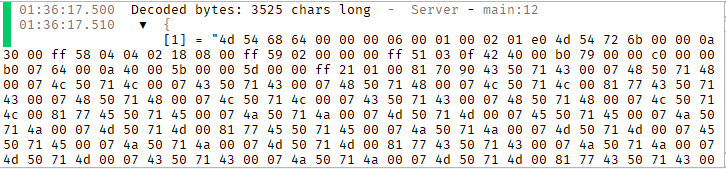


As a side note, when preparing this video I considered using an alternate version of THIS for-loop. Here you can see both side-by-side.



As is often in programming there will be multiple ways of writing code that does the same thing. These two for-loops have literally the same outcome, but it's only down to preference for how you would like to spell it out.

Anyways, let's try it out once more.



Ah cool! Looks just like our Sublime Text view from earlier.

Well, this concludes episode 1 of our Pipe Dream rewrite series. We now have a fully implemented "File-to-Roblox" system in our code, which allows arbitrary binary input to be read by any Lua script that we desire. Stay tuned for the next episode, where I take a look at the Midi file format, and interpret our prepared binary data as Midi file input.

Until next time. Y'all have a wonderful rest of your day, bye!