Ex Machinis

Player's Manual

The Player's Manual provides a detailed description of available game-play mechanics and strategies. This manual will grow with the game. As more features are added, they will be introduced below. Currently, the manual focuses primarily on coding spacecraft with the fully developed FORTH interpreter. However, as we develop the more physical components of Ex Machinis game play more details will be provided on navigation, resource extraction, refining, and production. Thank you for your support and please stay tuned for future developments!

Table of Contents

(Click the header to jump to the content.)

Chapter 1: Overview

Your fortune awaits!

Ex Machinis is a multiplayer space game, which takes place in a parallel universe. Within this altered reality, space technologies have rapidly advanced since the 1950's to the point where extensive interplanetary industries exist to produce materials that drive the human economy back on Earth. The vast majority of these spacecraft and are unmanned vehicles, which are assembled in space and are controlled terrestrial barons.

As part of the international agreement that drove the colonization of space at the end of the second world war, citizens of the signitor nations are guaranteed access to basic spacecraft which can be used to build profitable businesses. This democratization of space has been the biggest driver behind the rapid growth of extraterrestrial industries and technologies.

It is your birthright to take control of your own remotely piloted spacecraft and build a thriving stellar industry. But this can only be accomplished by programming your spacecraft to efficiently mine, manufacture, and trade materials on the interplanetary economy.

We still have light years to go

Ex Machinis is in the very early stages of development but we want to make it available to potential players so they can kick the tires and provide us with valuable feedback and suggestions.

We currently have a functional email server that allows players to create an account and code a fleet of three spacecraft using a FORTH compiler. The spacecraft are able to process simple FORTH commands but can't execute any in game activities yet.

Right now we're developing the Event Engine that will govern the physical actions of spacecraft. When the Event Engine comes online, your drones will be able manufacture materials move from one location to another.

This player's guide is a work in progress. It will be updated as new features become available in the game. Please follow us on http://patreon.com/exmachinis to receive updates and find the latest version of this guide.

Chapter 2: Getting started

Requesting your spacecraft

As an immersive multiplayer space simulation, there's no logging into or out of the game. Your assets are always in play. To communicate with your spacecraft, simply send them emails and they will respond in real time. Keep in mind that it will take longer for transmissions to reach spacecraft on the outer edge of the solar system.

To start playing, you need to email the Registrar (registrar@exmachinis.com) with the word "register" in the subject line. You'll receive a response from the registrar listing the email addresses of three spacecraft that are under your control. These spacecraft will only respond to communications from the email address you used to contact the registrar.

Also, please be assured that we'll only use your email to send in-game communications from your spacecraft or an occasional newsletter summarizing updates to the game. Furthermore, you'll only receive emails from your spacecraft in response to your own communications and you can unsubscribe from the newsletter at any time.

Contacting your spacecraft

Most of the game is played via direct email communications with your spacecraft. When one of your ships receives an email from you it will interpret everything between the <run> and </run> brackets as code and respond to you accordingly via email.

For example, try sending the following text to one of your spacecraft and see what happens. Just make sure you retain the spaces between the numbers, the plus sign, and the period, which instructs the spacecraft to report the result.

```
<run> 2 3 + . </run>
```

Choosing better names!

You'll notice that each of your spacecraft as a rather generic and unremarkable email address like SN0230854@advolition.com. You'll probably want to give it a more memorable name like

"hero" since you'll be interacting with it on a regular basis. To rename your spacecraft, simply send the following text in an email to your drone:

```
<rename>hero</rename>
```

You'll actually receive an email from hero@advolition.com confirming the name change. You can now send your instructions to this email address instead of the original!

There is no limit to the number of times you change your drone's email address. The only requirements are that you pick a name that doesn't contain any illegal characters and isn't already in use by someone else. If your chosen name is not allowed, the drone will let you know via its original email address.

Commanding your fleet

Unlike most space games, players succeed in Ex Machinis by avoiding the grind. Although it's possible to send individual commands to your drones, it's far more efficient to develop advanced routines for coordinating the actions of your multiple ships in distant space. Remember, the game doesn't stop when you're AFK and you don't want to wait for your communications to bounce back and forth across the void.

You'll be controlling your spacecraft via a simple but highly efficient programming language known as FORTH.

Chapter 3: Learning FORTH

FORTH is an amazingly simple but highly flexible programming language that grows out of a few core interpreter functions. In fact, the version of FORTH, which is currently installed in each of your spacecraft, is described in around 200 lines of game-side C code.

This simple architecture has encouraged many FORTH loyalists to develop their own interpreters and led others to muse that a true FORTH master is one who builds his own interpreter from scratch.

Consequently, FORTH will give advanced Ex Machinis players a tremendous amount of flexibility in developing a programming environment that works for them. However, it also means that beginners can start the game with a more standard dictionary of intuitive FORTH routines, which facilitate their entry into the game and it's dynamic programming environment.

This chapter is intended to give new users a basic understanding of FORTH so that they can use it to remotely pilot their spacecraft.

Doing Simple Math

You may have already noticed that FORTH has a strange way of doing math. In FORTH, the numbers come first followed by the operator (2 3 +). This is because FORTH relies on a data stack to transfer values to and from functions.

For example, when you enter "2" and "3" seperatred by a space, FORTH places a 2 and then a 3 on the data stack. You can visualize this as a stack of dishes in which the number 2 plate is laid down first and the number 3 plate is placed on top of it. Consequently, when objects are removed from the stack the topmost value is removed first (3) and then the next value (2).

When you call a FORTH function, like addition (+), it removes the top two values from the stack (2 and 3), adds them together, and returns the result (5) to the stack.

Similarly, calling the print function (.), pulls a value off the stack (5) and adds it the output stream, which is returned to you in an email as "5".

All FORTH functions behave the same way, by removing and adding values to the stack. This is why FORTH is a stack-based language. It's also why FORTH is a very efficient and intuitive programming language.

Understanding stacks is the key to learning FORTH and programming kick-ass spacecraft in Ex Machinis.

Managing the stack

In FORTH, different program elements share data by adding and removing values from this stack in a last-on-first-off manner. For example, when I email the following code one of my spacecraft, it adds the space-delimited values to the stack so that 1 is on the bottom of the stack and 4 is on top.

```
<script>
  1 2 3 4
</script>
```

If I were to follow this with a call to the addition function (+), it would remove the 4 and 3 from the top of the stack and place a 7 on the stack. The stack from bottom to top would end up looking like this: `1 2 7`

You may not always want to deal with values in the order they were placed on the stack. Therefore, a number of commands exist that allow you to directly manipulate the order and contents of the stack.

Try applying each of the following commands to manipulate the contents of the data stack in one of your spacecraft:

Word	**Action**	**Example ***
drop	Discards the top most item on the stack.	`0 1 2 3 drop . . .`
nip	Discards the second item on the stack.	`0 1 2 3 nip . . .`
dup	Copies the top most item on the stack.	`0 1 2 3 dup . . .`
over	Copies the second item on the stack the stack.	`0 1 2 3 over . . .`
swap	Switches to order of the top two items on the stack.	`0 1 2 3 swap . . .`
* The three dots in each of these examples instruct the interpreter to print the first three items on the stack. It's important that you place a space between each of the periods and the word that precedes them.		

When trying the above examples, remember that FORTH words are case sensitive and it is important that you include spaces between the numbers, words, and periods. It's also important that you include all FORTH script between <run> and </run>. Otherwise the interpreter won't processes the code.

Building a dictionary

FORTH is a very simple language. Most of the heavy lifting is done using the data stack and a dictionary of user-defined functions.

Every time you send your spacecraft instructions enclosed between the script brackets, the spacecraft's on-board interpreter will try to parse your text into words or numbers.

The interpreter will first look for a string of space delimited characters in the onboard dictionary. If the characters represent a previously defined word, the interpreter will execute the corresponding function. If not, it will attempt to interpret the string as a value and place it on the stack.

For example, the arithmetic operators (+, -, *, /) represented words, which the interpreter can find in the dictionary and execute to perform the proscribed computation.

However, FORTH makes it very easy to define your own words on the fly for use in scripts and higher level definitions. New words are defined by simply enclosing the word and the corresponding script between a colon (:) and a semicolon (;).

For example, if you find that you are frequently adding five to a number on the stack you can create the word, AddFive, to handle this operation in the future. You would define AddFive by sending the following script to your spacecraft:

```
<run>
  : AddFive 5 + ;
</run>
```

Now, anytime you want your spacecraft to add five to the number on the stack you can use your new word instead of "5 +":

```
<run>
  4 AddFive .
</run>
```

This script will cause your spacecraft to return the same result ("9") as the following script:

```
<run>
  4 5 + .
</run>
```

In this example, the word AddFive really doesn't offer much advantage over simply typing "5 +". However, you can easily imagine defining words that perform more complex functions that you wouldn't want to retype every time.

Give this a try with your own spacecraft! I'm sure you can come up with more useful definitions than the one provided here.

Commenting on your code

As with most programming languages, FORTH offers for some convenient mechanisms for commenting on your code. This techniques can be very handy in reminding yourself and others of the intentent and function of your definitions. There are two primary ways to add comments to your script that will be ignored by the interpreter:

- When you start a line with a backslash (\) the interpreter will ignore everything else on that line.
- The interpreter will also ignore any text enclosed between parentheses, regardless of where they occur in the line or whether they enclose multiple lines.

Note: Remember that the interpreter uses white space to deliminate words. Therefore, backslashes and parentheses will only be recognized as such by the interpreter if there are spaces between them and other characters.

Sending greetings

Having your spacecraft output text in an email to you is actually quite easy to do and obviously very useful. There are a couple ground rules though: First, this technique for printing quotes can only be called from within a word definition (ie. function). Second, it's critical that spaces separate your quotes from the text that is contained between them.

With that in mind, the following definition creates a word that, when called, instructs the spacecraft to print a specific string in its email response to you.

Start by emailed the following text to one of your spacecraft:

```
<run>
  : helloworld ." Greetings Earthlings! " ;
```

```
</run>
```

The spacecraft will respond to this email with a simple "ok" to acknowledge that it has received the new definition and incorporated it into its on-board dictionary. Remember, the spaces in the above example are very important since the allow the interpreter to delineate words and numbers.

Next, envoke your new word by sending the following email to your spacecraft:

```
<run> helloworld </run>
```

Sending this script to your drone will result in a heartfelt greeting from your favorite spacecraft! I guess there's no surprises there. But this will obviously be a useful tool when having your ships provide intelligible status reports.

The important components of this command are the dot-quote (."), which instructs the craft to print the text that follows and the end-quote ("), which ends the printed string. Just remember that because the FORTH interpreter views the dot and end quotes as distinct words, they must be separated from everything around them by a white-space.

Finding the truth in numbers

FORTH is very loose with numbers and doesn't have specially defined boolean values. Rather, logical operators interpret all non-zero numbers as true. Only zero is used to represent the boolean equivalent of false.

A good way to demonstrate this and introduce you to simple branching structures is by defining a new word, which uses an if-else-then statement to make a value judgement about any number that precedes it.

Try emailing the following text to one of your spacecraft:

```
<run>

  ( define true? )
  : true? if ." true " else ." false " then ;

  ( test true? )
  -1 true?
   0 true?
   1 true?
```

```
</run>
```

Remember that spaces are very important, so start by copying and pasting the above script directly into your email, including the <run> </run> brackets.

The first part of the emailed script defines a new word, true?, which will return a "true" if the word is preceded by a number other than zero and a "false" if it follows a zero. In the FORTH syntax, if begins a branched statement, which implements one of two else-delimited options based on the value on the top of the stack. The word, then, closes the statement.

The second part of this statement invokes true? after placing each of three different values (-1, 0, and 1) on the stack. This will cause your spacecraft to evaluate each of the numbers as a boolean value and respond accordingly.

After sending the above code to your spacecraft, you should receive an email from your drone, which contains the following response:

```
true false true ok.
```

By the way, once you define a word like **true?**, your spacecraft will remember it, allowing you to use it in future emails. So, true evaluating some other integers with the word.

We'll be using **true?** in the next section to explore logical (**and**, **or**, **xor**) and comparison (**<**, **>**, **=**) operators.

Experimenting with logical operators

In the previous section, we used an if else then statement to demonstrate how FORTH, which is used to program player-controlled spacecraft in Ex Machinis, loosely interprets 0 as false and any other number as true. In the process, we defined the word true?, which returns "false" if the value on the stack is zero and "true" if it is any other number.

If you haven't already done so, you can define the word true? by emailing the following script to your spacecraft.

```
<run>
  : true? if ." true " else ." false " then ;
</run>
```

It turns out that FORTH has three logical operators (and, or, xor) which allow you to combine Boolean values in different ways. Therefore, if we use 0 for false and 1 for true, you can test these operators by sending the following script to your drone:

```
<run>
  0 1 and true?
  0 1 or true?
  0 1 xor true?
</run>
```

You may not recall that FORTH uses a Reverse Polish Notation (RPN) in which you enter the numbers first (i.e. "0 1") before the function or mathematical operator ("and"), which processes them. Finally, we need to use our previously defined word, true?, to report back the value of the number. Since "0 and 1" is false, "0 or 1" is true, and "0 xor 1" is also true, your drone will respond to the above script with:

```
true true true ok.
```

Using the above example, instruct you spacecraft to process other combinations of 0's and 1's with and, or, and xor to make sure these logical operators behave as expected.

You may have noticed that there is one very important operator that is missing from the version of FORTH that is currently running on your spacecraft. There is no not operator. However, the beauty of FORTH is that you can easily define your own not operator as follows:

```
<run>
  : not 1 xor ; ( define not )

  0 not true? ( see how not works on 0 )
  1 not true? ( see how not works on 1 )
</run>
```

Try sending the above code to your spacecraft to see how it work.

Using variables when programming your drones

In FORTH, most values are stored and passed between program elements using the stack. However, the language also allows users to define simple variables using the variable command.

When a variable is declared, the FORTH interpreter creates an entry in the dictionary using the variable's name. This new definition serves as a placeholder for any values stored in that variable.

Values can then be stored in and retrieved from a variable using the store (**!**) and fetch (**@**) commands. In the following example, we define a variable called "box" and then store, fetch, and print the number 2 using this variable.

```
<script>
    variable box
    2 box !
    box @ .
</script>
```

That's all there is to it! Try running this script and variations of it on one of your drones.

Of course, like most things in FORTH, there are some interesting things going on behind the scenes when you declare a variable and store and retrieve values from it. We'll get into some of this in the next section when we address the creation of arrays.

What's really happening with variables?

So let's talk a little bit about what's happening behind the scenes with variables. When you declare a variable with the statement, "variable box", you're essentially telling the FORTH interpreter to define a new word for you called **box**. This definition tells the interpreter that whenever **box** is called, it should place an address on the stack, which points to a four byte block at the end of the definition. This location is where your data is stored.

You can confirm this is happening by calling **box** and then printing the contents of the stack. When you do so, your drone will return the address of the place in memory where your value is being stored. The variable, **box**, will always return the same value no matter what's actually being stored in the variable.

```
<script>
    box .
</script>
```

With this in mind, consider that the true functions of fetch (**@**) and store (**!**) are to read and write values to a specified location in memory. Fetch works by pulling an address off the stack and

returning the four-byte value that is stored in that location. Store pulls an address and then a value from the stack and writes the value to the designated memory slot.

Fetch and store are very powerful (and dangerous!) functions because they allow you to read from and write to ANY location in your drone's memory! However, until you're sure of what you are doing, I suggest that you only use these functions in conjunction with a previously defined variable like **box**.

So why am I telling you about these functions now? Because understanding how FORTH uses variables is the key to defining and using more powerful strings and arrays!