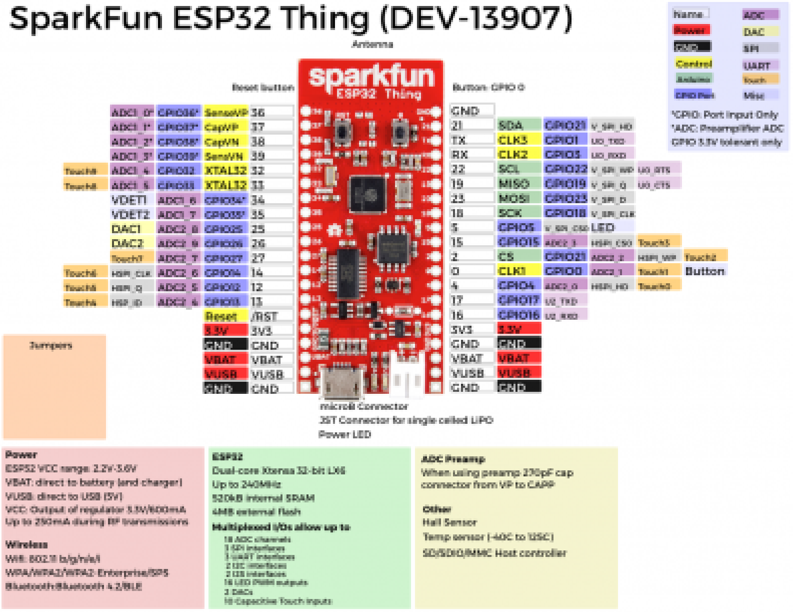
← Pharo Libclang FFI, part 5, client data and recursive visitor/callbacks

Pharo PDF Rendering, part 1, building PDFium →

An evening with Pharo and the ESP32 microcontroller

Posted on 2017/06/30 by Ben Coman

Two popular choices for controlling maker projects are the Arduino and Raspberry Pi. The Pi is a micro-”computer” that runs Linux to operate as a low powered desktop computer. The Arduino is a much lower powered micro-”controller” without display nor wireless interfaces, but it comes with analog IO the Pi lacks. But now we’ve a new cool-kid on the block – the ESP32 in the form of the Sparkfun ESP32 Thing and the WeMos LOLIN32.


SparkFun ESP32 Thing (DEV-13907)
Antenna
Reset button
Button: GPIO 0
Jumpers
microB Connector
JST Connector for single celled LiPO
Power LED
Power
ESP32 VCC range: 2.2V-3.6V
VBAT: direct to battery (and charger)
VUSB: direct to USB (5V)
VCC: Output of regulator 3.3V/600mA
Up to 250mA during RF transmissions
Wireless
WiFi: 802.11 b/g/n/e/i
WPA/WPA2/WPA2-Enterprise/WPS
Bluetooth:Bluetooth 4.2/BLE
ESP32
Dual-core Xtensa 32-bit LX6
Up to 240MHz
520kB internal SRAM
4MB external flash
Multiplexed I/Os allow up to
18 ADC channels
3 SPI interfaces
3 UART interfaces
2 I2C interfaces
2 I2S interfaces
16 LED PWM outputs
2 DACs
10 Capacitive Touch inputs
ADC Preamp
When using preamp 270pF cap
connector from VP to CAPP
Other
Hall Sensor
Temp sensor (-40C to 125C)
SD/SDIO/MMC Host controller

Fitting squarely between the Pi and Arduino, the ESP32 is a micro-controller like the Arduino nearing the speed of the Pi ZeroW. Its got even more analog IO where the Pi has none, and built-in WiFi and Bluetooth interfaces the Arduino lacks. This makes the ESP32 a great candidate platform for many applications including machine control and equipment condition monitoring. A built-in battery charger is a nice bonus.
I've tabled a spec comparison…

	RPi ZeroW	ESP32 Thing	ESP32 WeMos	Arduino Uno R3
CPU	1Ghz 64bit	240MHz 32bit	240MHz 32bit	16MHz 8bit
RAM	512MB	512kB	512kB	2kB

- A SignalR client for Pharo
- Pharo v. Cloudflare
- Pharo PDF Rendering, part 2, UFFI interfacing PDFium
- Pharo PDF Rendering, part 1, building PDFium
- An evening with Pharo and the ESP32 microcontroller

Recent Comments

- Pharo v. Signalr | openInWorld on Pharo v. Cloudflare
- Pharo PDF Rendering, part 2, PDFium UFFI interfacing | openInWorld on Pharo PDF Rendering, part 1, building PDFium
- An evening with Pharo and the ESP32 microcontroller | Weekly news about Pharo on An evening with Pharo and the ESP32 microcontroller
- Ben Coman on Contributing to Pharo By Example
- EuanM on Contributing to Pharo By Example

Archives

- February 2018
- November 2017
- June 2017
- September 2016
- May 2012
- December 2011
- April 2011
- March 2011

Flash onboard / card	0 / 64GB	4MB / 0	4MB / 0	32kB / 0
GPIO	24	28	26	14
ADC / Channels	0	2 / 18	2 / 12	1 / 6
DAC	0	2	2	0
PWM Timers / Channels	0 / 0	8 / 16	8 / 16	3 / 6
SPI / UART	2 / 1	3 / 3	3 / 3	1 / 1
I2C / I2S	1 / 0	2 / 2	2 / 2	1 / 0
Touch inputs	0	10	10	0
Graphics	HDMI	N	N	N
WiFi	Y	Y	Y	N
Bluetooth	Y	Y	Y	N
Battery charger	N	Y	Y	N
Price (AUD)	$15	$30	$20	$40

It would be cool to interface an ESP32 to my favourite programming language Pharo! Pharo is an immersive object-oriented live programming environment. What I love about it is the very tight code-compile-run-debug cycle that keeps you agile and in the flow while prototyping. This video[7min] by a Pharo newcomer gives a quick taste of the environment.

Categories

- Admin
- Debugging
- FFI
- Morphic
- Pharo
- System
- Uncategorized

Blogroll

- Learning Smalltalk
- OpenSmalltalkVM
- Pharo

Meta

- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Now so that I don’t end up out in the wilderness on my own (since this Pharo/ESP combination is new) this post seeks to inspire you to explore how Pharo and microcontrollers like the ESP32 can work together. As such, this post is targetted at newcomers to either domain.

[If you don't have a ESP32 in hand, skim through and you may see enough to inspire you to order one and give it a go. Microcontroller to PC interfacing is a growing market and you never know what opportunities may arise from expanded skills in this area.]

ESP32 first steps

Best to start with the standard tools to verify everything works.

1. Follow Sparkfun’s ESP32 hookup guide. I used the portable version of the Arduino IDE since on Windows I operate with a non-admin account.
2. Connect USB to the ESP32 and download your first program to verify comms. I used “File > Examples > 0.1Basics > Blink” which toggles the Onboard-LED using delays. You can play with the numbers to confirm changing blink patterns have an effect.
3. Add text output. I mixed in the code from the from the hookup guide serial comms example to produce *BlinkHello.ino*. After downloading this to the ESP32, open Tools > Serial Monitor from the Arduino IDE.

```
// BlinkHello.ino - friendly introduction to the ESP32
int counter = 0;
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
    Serial.begin(9600);
}
void loop() {
    Serial.print("Hello, world!  ");
    Serial.println(counter);
    digitalWrite(LED_BUILTIN, HIGH);
    delay(500);
    digitalWrite(LED_BUILTIN, LOW);
    delay(2000);
}
```

Pharo first steps

1. Download Pharo 6, unzip it and run `pharo` (its portable, you don't need to install it). You are presented with a "Welcome" window on an empty background called the "World". The world holds the UI objects we interact with.
2. Left-click on the World background and from the menu select Help > Pharo Tutorials. Working through the "Pharo Syntax Tutorial" should provide enough to follow the rest of this post. When you have more time, I encourage you to check out:
   * The Pharo MOOC (the audio is French, but the English subtitles are good)
   * Pharo By Example
3. Left-click the world and open a Playground.

Taking hints from How-to-use-the-serial-port-on-MacOS-X (I'm using Windows 10), in the Playground evaluate the following snippet (i.e. select and DoIt)...

```
baudRate := 9600.
comPortNumber := 6. "COM6 for me, taken from my Arduino IDE config"
```

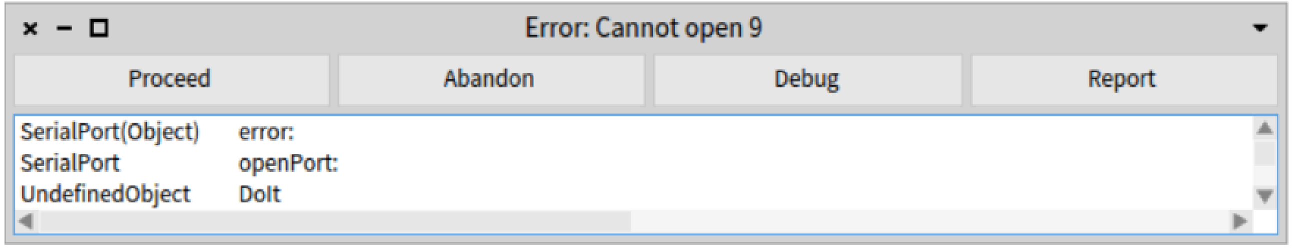
```
myPort :=
    SerialPort new
        baudRate: baudRate;
        dataBits: 8;
        stopBitsType: 1;
        parityType: 0;
        yourself.
(myPort openPort: comPortNumber) isNil ifFalse: [ myPort inspect ].
```

If a windows pops up titled "Inspector on a SerialPort…" that is great! You've successfully opened the serial port. If you get "Error: Cannot open 6″, just close the window and check your Arduino IDE Serial Monitor is closed then try again. If the error persists, confirm the port works from a terminal program (e.g. HyperTerminal or minicom) and seek help from pharo-user mail list.

If a window appears at the bottom showing "OpenComm failed…", scroll through it to read how to close it (clicking first in the World before pressing F2.)

A common exception – improper port

Lets force an error to see what that looks like! Evaluate that snippet again with a port you don't have (e.g. comPortNumber := 9 for me). I got an expected "Error: Cannot open 9″.

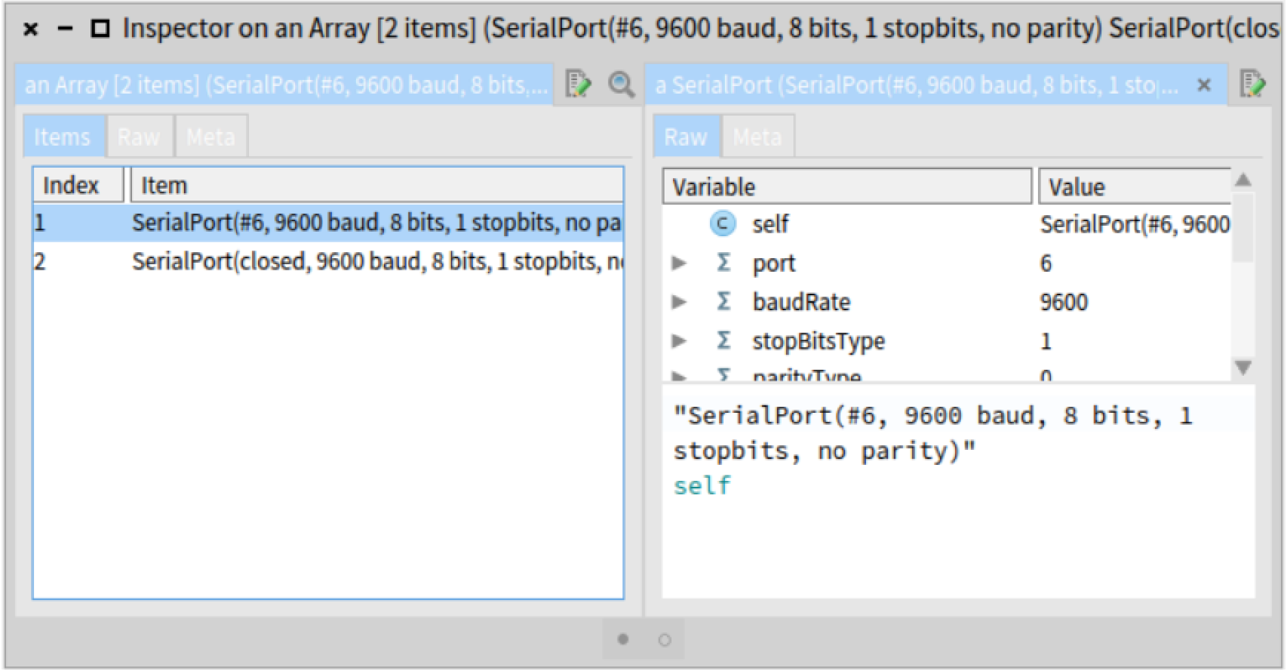


```
SerialPort(Object)    error:
SerialPort            openPort:
UndefinedObject       DoIt
```

Note that objects don't disappear when code evaluation completes. They stay live within the system for you to interact. Only when all references to them are dropped are they automatically garbage collected. But perhaps you've closed the inspector and also lost the playground's reference to that first serial port object when the "myPort" variable had a new object assigned into it. No worries, you can find it using Pharo's introspection tools by evaluating…

```
SerialPort allInstances inspect
```

which displays an inspector on an Array of two SerialPort objects – one for the first port you opened (#6 for me) and a closed one since port #9 did not exist. Clicking on a SerialPort displays its internals on the right.


× − □ Inspector on an Array [2 items] (SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no parity) SerialPort(clos
an Array [2 items] (SerialPort(#6, 9600 baud, 8 bits,...
a SerialPort (SerialPort(#6, 9600 baud, 8 bits, 1 sto|...
Items
Raw
Meta
Raw
Meta

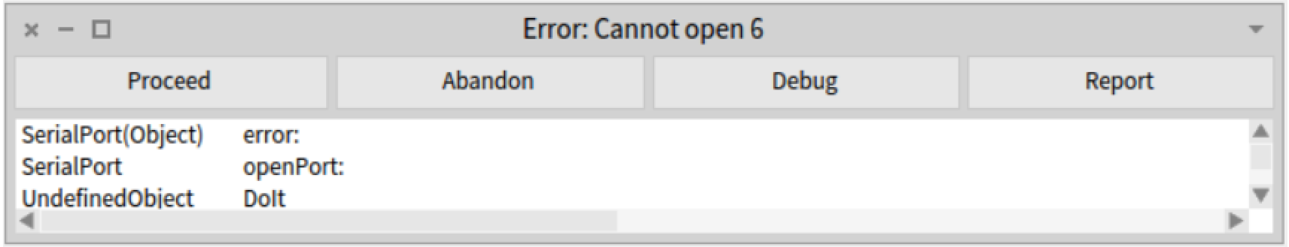
Index	Item
1	SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no pa
2	SerialPort(closed, 9600 baud, 8 bits, 1 stopbits, n

Variable	Value
self	SerialPort(#6, 9600
port	6
baudRate	9600
stopBitsType	1
parityType	0

```
"SerialPort(#6, 9600 baud, 8 bits, 1
stopbits, no parity)"
self
```

Another common exception – port already open

Restoring the code snippet to original port (e.g. for me comPortNumber := 6) and again evaluating it now gives “Error: Cannot open 6.”

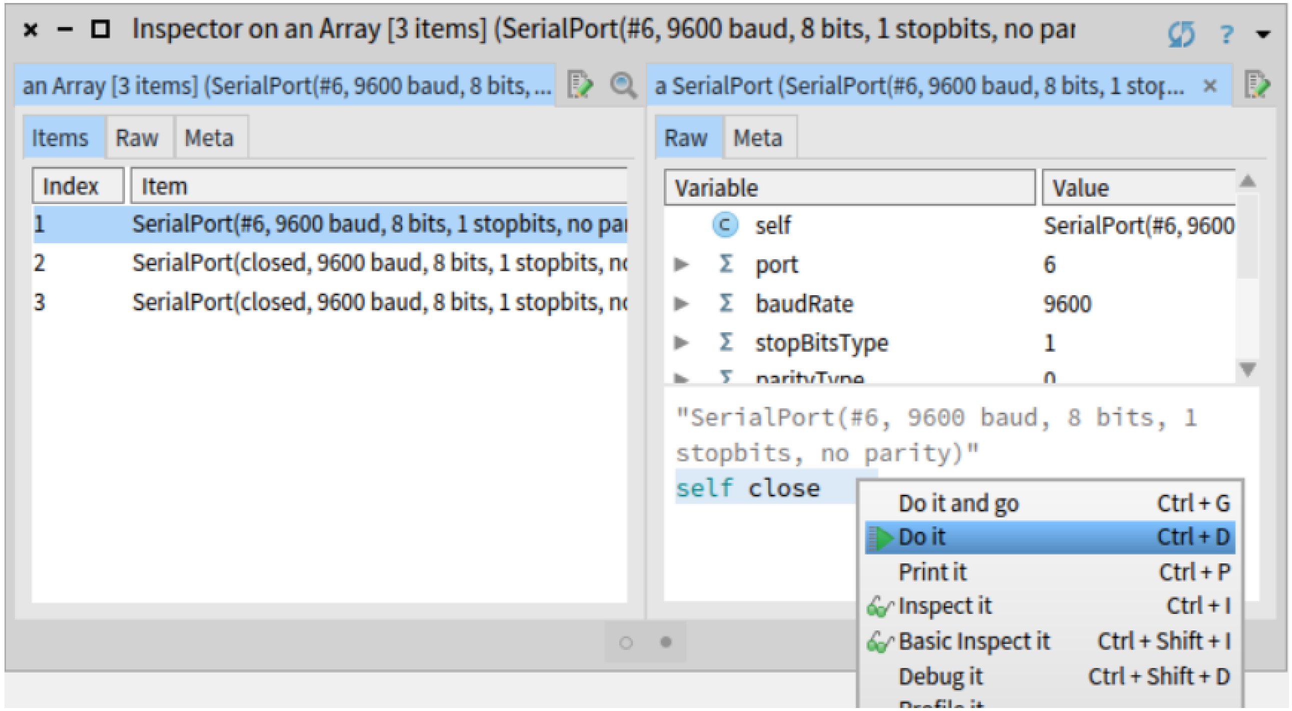


```
SerialPort(Object)    error:
SerialPort            openPort:
UndefinedObject       DoIt
```

This worked before! Why doesn't it work now? Remember that objects remain live for us to interact with. The port is still held by the first SerialPort object we created. Indeed three objects are shown when this is evaluated again…

```
SerialPort allInstances inspect
```

If you've lost all other references to an object in your program, you can still close the port from the inspector by selecting the open (e.g. #6) SerialPort, and under its [Raw] tab in the bottom pane evaluate "**self close**", and then click the Refresh icon to see the change. Now evaluating that playground snippet works to open the port.



Inspector on an Array [3 items] (SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no par

an Array [3 items] (SerialPort(#6, 9600 baud, 8 bits, ...

Index	Item
1	SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no pa
2	SerialPort(closed, 9600 baud, 8 bits, 1 stopbits, nc
3	SerialPort(closed, 9600 baud, 8 bits, 1 stopbits, nc

a SerialPort (SerialPort(#6, 9600 baud, 8 bits, 1 stop...

Variable	Value
self	SerialPort(#6, 9600
port	6
baudRate	9600
stopBitsType	1
parityType	0

```
"SerialPort(#6, 9600 baud, 8 bits, 1
stopbits, no parity)"
self close
```

Cleaning up

Garbage collection happens automatically in the background. Mostly you don’t need to think about it, but sometimes an object hangs around longer than you expect due to a bug in your program, or you lost track of some inspector or playground with a reference to it. To force all SerialPorts to close and be garbage collected, evaluate the following...

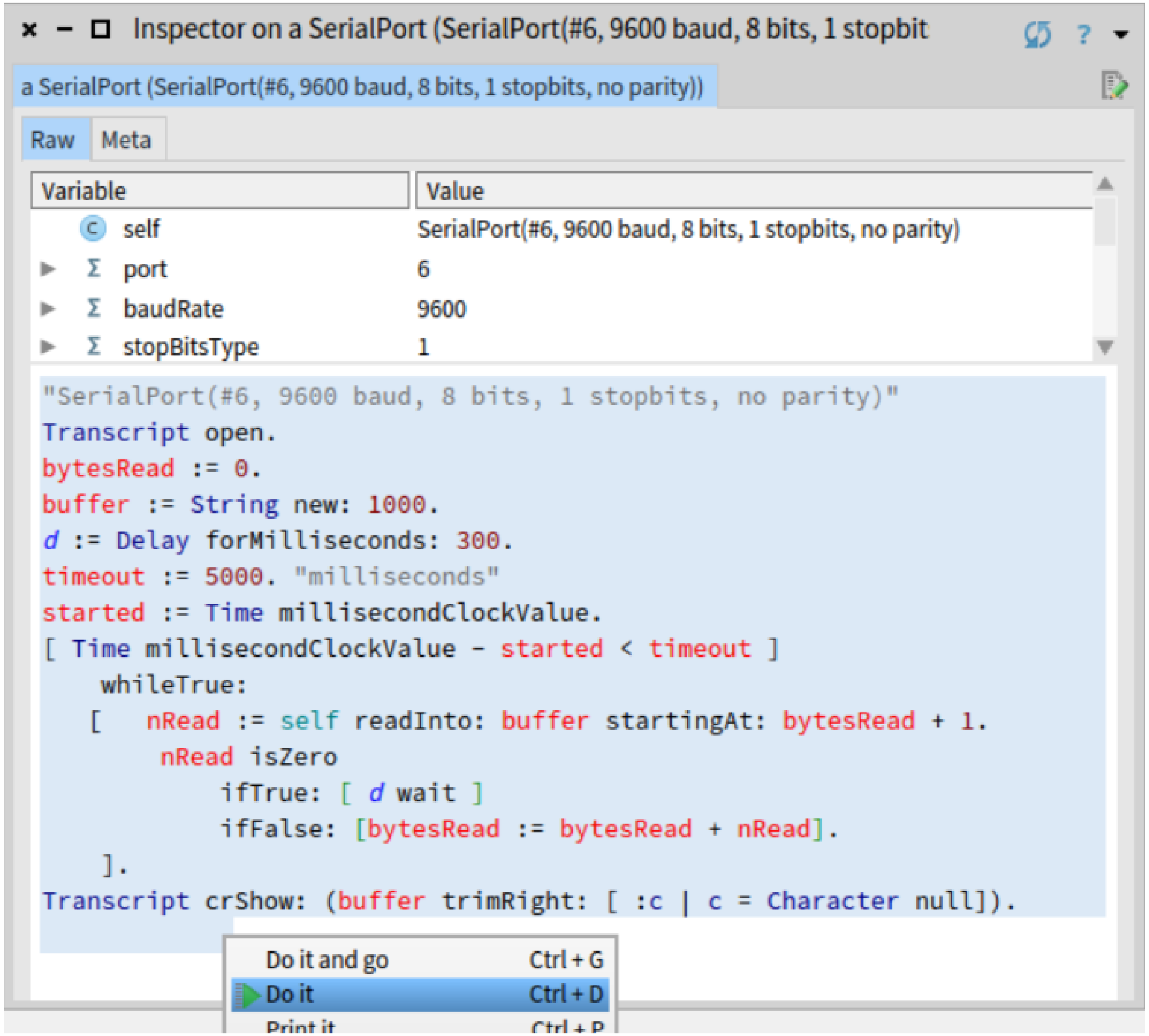
```
Smalltalk inform: 'Before=' , SerialPort allInstances size printString.
SerialPort allInstancesDo: [:port| port close].
GTPlayground allInstancesDo: [ :playground | playground resetBindings ].
SystemWindow allSubInstances
    select: [ :window | window title beginsWith: 'Inspector' ]
    thenDo: [ :window | window delete ].
Smalltalk garbageCollect.
Smalltalk inform: 'After=' , SerialPort allInstances size printString.
```

As final backup, if it the system reports there are zero SerialPort instances and the port still won’t open, close and reopen Pharo (Save-and-Quit, then reopen it) since stopping the VM drops all resources.

Reading the serial port

Find the inspector that has the serial open, then under its [Raw] tab in the bottom pane, evaluate this snippet...

```
Transcript open.
bytesRead := 0.
buffer := String new: 1000.
d := Delay forMilliseconds: 300.
timeout := 5000. "milliseconds"
started := Time millisecondClockValue.
[ Time millisecondClockValue - started < timeout ]
    whileTrue:
        [   nRead := self readInto: buffer startingAt: bytesRead + 1.
        nRead isZero
            ifTrue: [ d wait ]
            ifFalse: [bytesRead := bytesRead + nRead].
    ].
Transcript crShow: (buffer trimRight: [ :c | c = Character null]).
```

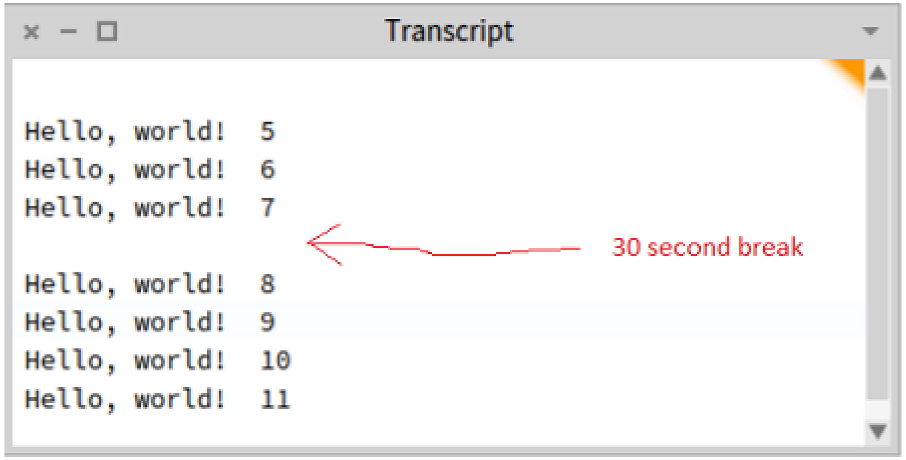


Inspector on a SerialPort (SerialPort(#6, 9600 baud, 8 bits, 1 stopbit
a SerialPort (SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no parity))
Raw
Meta
Do it and go Ctrl + G
Do it Ctrl + D
Print it Ctrl + P

Variable	Value
self	SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no parity)
port	6
baudRate	9600
stopBitsType	1

```
"SerialPort(#6, 9600 baud, 8 bits, 1 stopbits, no parity)"
Transcript open.
bytesRead := 0.
buffer := String new: 1000.
d := Delay forMilliseconds: 300.
timeout := 5000. "milliseconds"
started := Time millisecondClockValue.
[ Time millisecondClockValue - started < timeout ]
    whileTrue:
   [    nRead := self readInto: buffer startingAt: bytesRead + 1.
         nRead isZero
             ifTrue: [ d wait ]
             ifFalse: [bytesRead := bytesRead + nRead].
    ].
Transcript crShow: (buffer trimRight: [ :c | c = Character null]).
```

After five seconds a friendly window pops up to greet us. The reason for the delay is that the code is evaluated within the thread of the UI event loop. We will improve that shortly. For now, just wait a short time, then evaluate that snippet again. Notice that the counter is contiguous. A buffer somewhere between Pharo and the board accumlates transmitted data until we asked for it.



```
Hello, world!  5
Hello, world!  6
Hello, world!  7

Hello, world!  8
Hello, world!  9
Hello, world!  10
Hello, world!  11
```

30 second break

Now we don't want the IDE or our application to pause five seconds every time we read the serial port. We need to move its execution away from the Playground. The Playground is great for experimenting, but then the time comes to graduate to application coding so that programs can be more interactive. So we are going to create a class to model interaction with the ESP32. From the world menu open the System Browser and in the bottom pane enter and accept the following class definition...

```
Model subclass: #ESP32
      instanceVariableNames: 'pollTime'
      classVariableNames: ''
      package: 'ESP32Stuff'
```

There you go, you've just created your first class in Pharo!

Diversion – under the covers

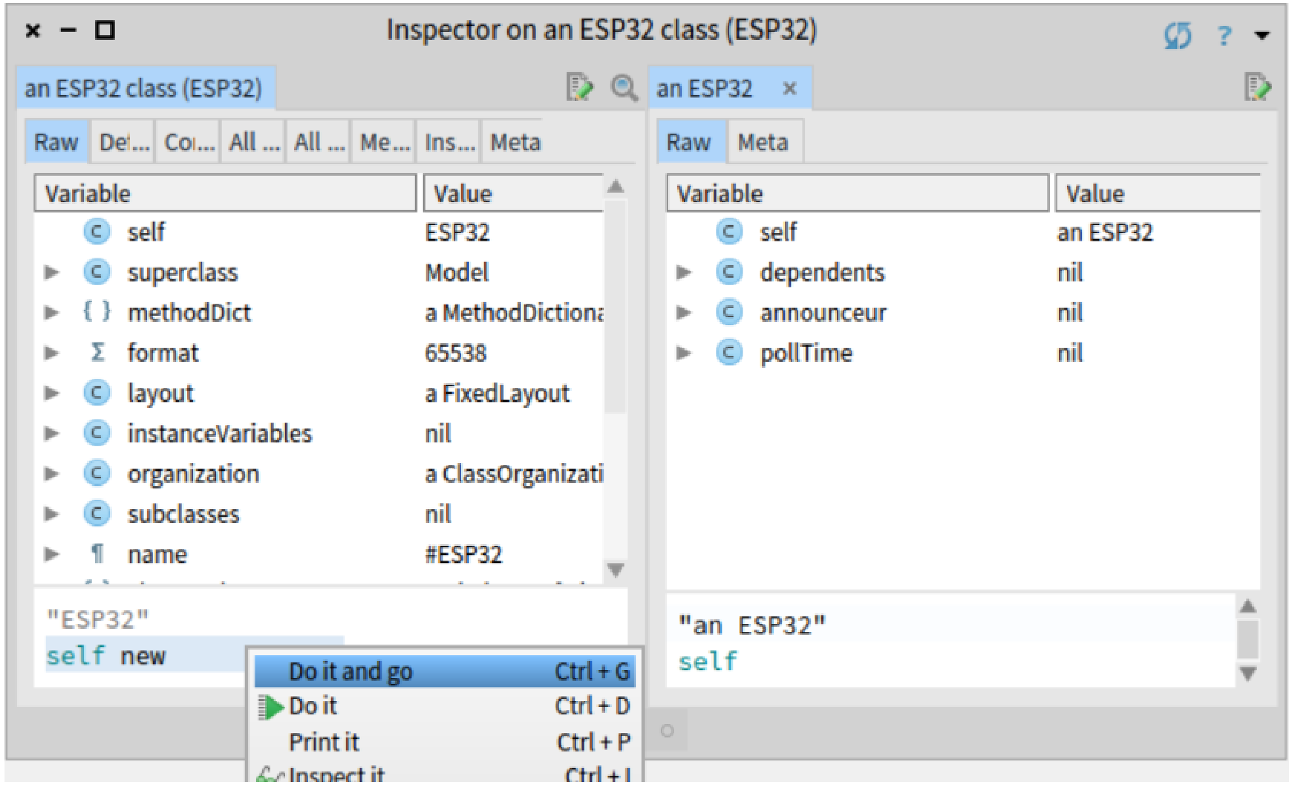
Now be aware that in a significant deviation from most languages where classes are defined by dead text, in Pharo this new class in already alive.   Every class definition instantiates a factory object for producing instances of the class. The factory object is globally identified by the name of the class and you directly interact with it like any other object. Normally you wouldn't do the following, but just by way of demonstration, try…

```
ESP32 inspect. "Inspects the ESP32 factory object."
```

What your looking at is the internals of how the Pharo system works.   You can see this factory knows its class name and its superclass. There are currently no subclasses and no methods. I didn't expect instanceVariables to be nil, but this is probably due to an implementation change in Pharo 5 to use Slots. Don't worry, you'll never need to look at this again. It just helps get your head around the Smalltalk paradigm that Pharo is based.

New instances of ESP32 are created by sending the message **#new** to the factory object. You can do this from the bottom pane of the inspector's [Raw] tab like this…

```
self new inspect.  "Inspects an ESP32 instance"
```



Variable	Value
self	ESP32
superclass	Model
methodDict	a MethodDictiona
format	65538
layout	a FixedLayout
instanceVariables	nil
organization	a ClassOrganizati
subclasses	nil
name	#ESP32

```
"ESP32"
self new
```

Variable	Value
self	an ESP32
dependents	nil
announceur	nil
pollTime	nil

```
"an ESP32"
self
```

Notice the tab on the right is “an ESP32”, which is a Smalltalk semantic for discussing an instance object, and on the left the tab is “an ESP32 class” which is how the factory object is referred to. The “pollTime” instance variable is visible on the right. From the playground, you can also create new instances like this…

```
ESP32 new inspect.  "Inspects an ESP32 instance"
```

And just for fun you can try these…

```
ESP32 superclass inspect.
```

```
ESP32 superclass methods inspect.
```

```
ESP32 superclass subclasses inspect.
```

Except for the VM providing a bytecode execution engine with a few highly optimised primitives and a few platform independent primitives, everything about how Pharo works is coded in Pharo. Indeed the code for Pharo's IDE lives the image where you are working. For example, the Playground, Transcript and System Browser windows are composed of widgets inside a SystemWindow. Lets just open a blank one...

```
SystemWindow new openInWorld.
```

And do you want to see the code behind that? Select that text again and choose [Debug it] rather than [Do it] and step through how it works. Or to review its class inheritance chain try this...

```
SystemWindow browseHierarchy.
```

I find this accessessibility to the guts of the system amazing and fun to explore. Your application is coded as a sibling to the IDE, which makes it extremely easy in Pharo to extend the IDE with domain specific tools. (You can strip out the IDE when you deploy your application).

Now close all those “diversion” windows, and lets move on.

Back to our regularly scheduled programme…

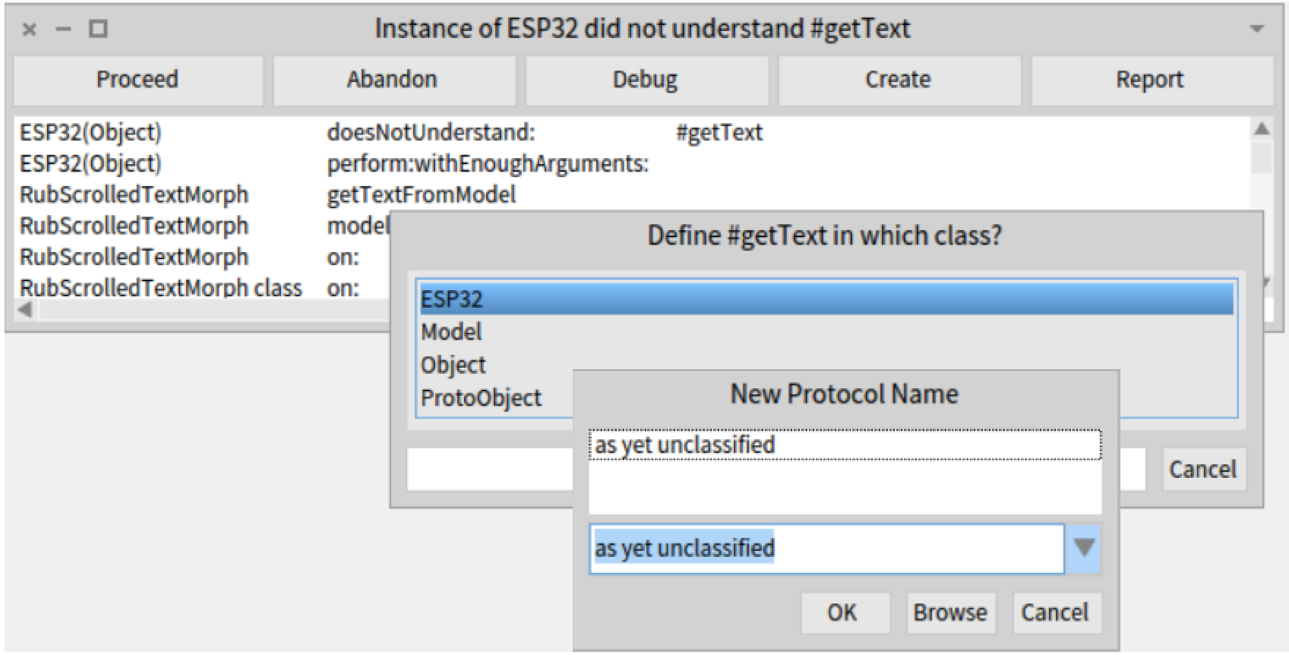
Before we do anything else, as we quickly iterate, I know you’ll get tired of manually dealing with serial-port-already-open errors (I did). So lete get ahead to shortcut that pain. But to shortcut keep this tutorial short, I used the following ugly hack to close all open serial ports every time a ESP32 instance is created. In the System Browser add method **ESP32>>#initialize**.

```
initialize
    SerialPort allInstances do: [:port| port close].
```

Now we can start making use our new class.   In the playground, evaluate…

```
model  := ESP32 new.
window := (SystemWindow labelled: 'ESP32 Serial port').
pane := RubScrolledTextMorph on: model.
window addMorph: pane frame: (0@0 corner: 1@1).
window openInWorld.
```

and you'll get an expected error "*Instance of ESP32 did not understand #getText*" since we've not defined that method yet. Click [Create] and from the next list choose "ESP32" as the class where we want to create the method, and for now just leave it "as yet unclassified".


Instance of ESP32 did not understand #getText

```
ESP32(Object)              doesNotUnderstand:        #getText
ESP32(Object)              perform:withEnoughArguments:
RubScrolledTextMorph       getTextFromModel
RubScrolledTextMorph       model
RubScrolledTextMorph       on:
RubScrolledTextMorph class on:
```

Define #getText in which class?

- ESP32
- Model
- Object
- ProtoObject

New Protocol Name

as yet unclassified

as yet unclassified

Enter your name as author if required. Then what you are then looking at is the debugger open on the newly created method. The top pane shows the call stack with the right-column the message received. The left-column shows the receiving class (of the object that received the message) with brackets indicating the superclass where an implementation method was found, being the method code shown in the middle pane. The bottom pane shows the variables scoped for the current execution context.

You'll see the default method template has been used for **ESP32>>getText**…

```
getText
    self shouldBeImplemented.
```

So now we are going to do something reasonably unique to Smalltalk. We are going to code our application from within the debugger. Rather than think too hard up front about our class structure and api, it will it flow naturally from dealing with requirements just-in-time, as they arise from a running program. This facilitates rapid prototyping of new domains as you explore the problem space.

Enter and the following code into the debugger [Source] tab. (And just for demonstration purposes, keep the ESP32 class definition in a System Browser visible. Upon accepting the code, you'll be advised that you have "Unknown variable: text…".   Click "**Declare new instance variable**" and you'll see the System Browser show an updated ESP32 class definition.

```
getText
    | buffer nRead |
    buffer := String new: 1000.
    nRead := self port readInto: buffer startingAt: 1.
    nRead isZero ifFalse: [self appendText: (buffer copyFrom: 1 to: nRead)].
    ^text
```

As you can see from “self port” it was natural here to assume that an ESP32 would know which serial port its connected to, without being distracted by how this was implemented. But its time now for that. Continue debugger execution by clicking [Over] until you get an expected “doesNotUnderstand: #port” at the top of the call stack. Click [Create], in ESP32, “as yet unclassified” and we’ll adapt our previous playground snippet.
Enter the **ESP32>>#port** method as follows…

```
port
    | baudRate comPortNumber |    SerialPort allInstances do: [:port| port clos
    baudRate := 9600.
    comPortNumber := 6. "COM6 for me, taken from the Arduino IDE config"
    (myPort isNil or: [myPort isClosed]) ifTrue:
    [   myPort :=
            SerialPort new
                baudRate: baudRate;
                dataBits: 8;
                stopBitsType: 1;
                parityType: 0;
                yourself.
        (myPort openPort: comPortNumber) isNil
            ifFalse: [ Transcript crShow: 'Opened ' , myPort printString ].
    ].
    ^myPort. "the return value"
```

After accepting this code, for “Unknow variable: myPort” select “declare new instance variable”. Continue the debugger execution using [Over], and if it fails to open the port, close any open ports by evaluating the comment then click. Now depending on exactly everything you do in parallel, you'll get a doesNotUnderStand for either **ESP32>>#appendText** or **SerialPort>>#isClosed**.

For the former, click [Create] in ESP32, "as yet unclassified", then enter this...

```
appendText: aString
    text ifNil: [text := ''].
    text := text , aString.
```

And for the latter? Well!... I had presumed SerialPort would understand something like **#isClosed,** but it turns out there is nothing like it. (Hey! This is my first time using the Pharo serial interface). So lets extend SerialPort for our needs. Click [Create] in SerialPort, but this time the new protocol name will prefix our package name with an asterix, i.e.... *ESP32Stuff , which is the system convention to organise extension methods into their relevant package. Define method **SerialPort>>#isClosed** as follows...

```
isClosed
    ^ port isNil
```

Continue stepping until you return to the last line of **ESP32>>#getText**, then click [Proceed] to exit debugging. A window should open, however depending on how quickly you stepped through the debugger (i.e. how soon after opening the serial port it was read) it may or may not show any text. We need to prompt the text to update itself. Evaluate the following in the playground...

```
model announcer announce: RubTextUpdatedInModel.
```

and you should see some friendly text appear. Now we need that to occur periodically. Now in the playground we've been using **RubScrolledTextMorph**. We want the same thing but polling periodically, so we can subclass like this...

```
RubScrolledTextMorph subclass: #ESP32Morph
    instanceVariableNames: 'error'
    classVariableNames: ''
    package: 'ESP32'
```

and modify behaviour by defining the following methods on **ESP32Morph**...

```
initialize    super initialize.
    self model: ESP32 new.
```

```
stepTime
    ^5000
```

```
step
    self model announcer announce: RubTextUpdatedInModel
```

```
openInWorld
    | window |
    window := SystemWindow labelled: 'ESP32 Serial Port Monitor'.
    window addMorph: self frame: (0@0 corner: 1@1).
    window openInWorld.
```

ATTENTION: As a precaution before proceeding, save your image so you might experient with the problem I bumped in to.

In the playground, we can now we can start our serial monitor in one line…

```
ESP32Morph new openInWorld
```

and you should see greetings ticking by. But whoops! Now its updating periodically I bumped into one prickly issue. While watching the greetings tick over, I disconnected the ESP32 board from the USB. Bam! Multiple debuggers swarmed the screen. I needed to quit and restart Pharo, or if your stepTime allows, close the monitor window and in playground do the following…

```
SerialPort allInstances do: [ :port | port close ].
```

(btw, You can right-click the Pharo task bar to find “Close all > close all debuggers”.) Digging into one of those debuggers, the guilty code line in **ESP32>>getText** is…

```
nRead := self port readInto: buffer startingAt: 1.
```

so lets wrap some #on:do: exception handling around it. We’ll have a guard variable “error” which will be nil in a newly created instance and will be set to true when there is an errort. Like this…

```
getText
    | buffer nRead |
    buffer := String new: 1000.
    error ifNotNil: [^text].
    [   nRead := self port readInto: buffer startingAt: 1.
        nRead isZero
                    ifFalse: [self appendText: (buffer copyFrom: 1 to: nRead)].
    ] on: PrimitiveFailed
      do: [ error := true.
            self port close.
            self appendText: 'PRIMITIVE ERROR READING PORT !!'.
          ].
^text
```

Accept that and same as before, when it reports “Unknown variable: error”, select “Declare new instance variable” as before, and thats all thats needed here.

Minor polish

Nothing fancy, but just a couple of final points of polish. First when we close the window it would be good to close the port so the Arduino IDE can download new programs. So modify **ESP32Morph** as follows...

```
openInWorld
    | window |
    window := SystemWindow labelled: 'Pharo ESP32 Serial Port Monitor'.
    window addMorph: self frame: (0@0 corner: 1@1).
    window announcer
        when: WindowClosed
        send: #closePort
        to: self.
    window openInWorld.
```

```
closePort    self model close.
```

Second, shift to the class-side (i.e the factory) **by clicking the [Class] button** below the second pane of the System Browser. Select ESP32Morph, then in the third-pane click on the bold "no messages" to show the method template in the bottom pane. Replace that with…

```
open
    self new openInWorld
```

so now from the playground you can evaluate…

```
ESP32Morph open.
```

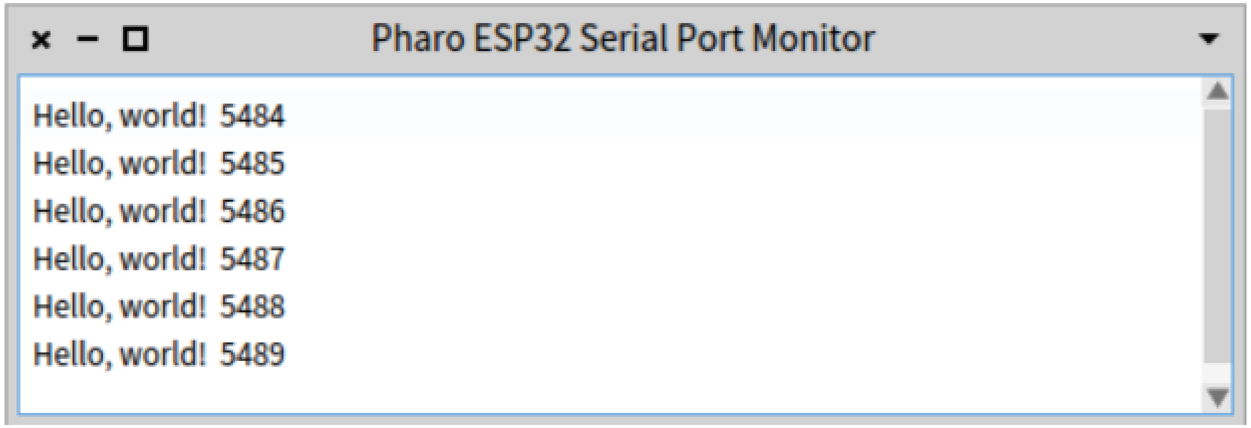
And finally, when you close the window we get the expected complaint "Instance of ESP32 did not understand #close", so click [Create], "ESP32″, "as yet unclassified". Reviewing the live state on the object in the variable list at the bottom you can see variable "myPort" is holding a SerialPort object, so its clear we want to act on that, so make the method look like this…

```
close
    myPort close.
```

…and [Accept] then click [Proceed]. Now just to cement understanding, in the System Browser find where that method was created. You may need to **click the [Class] button** again to toggle back to viewing instance-side of the ESP32 class.

And the monitor is done...

Actually it turned out there wasn't there ESP32 specific. It would work similar with any microcontroller.



```
Hello, world! 5484
Hello, world! 5485
Hello, world! 5486
Hello, world! 5487
Hello, world! 5488
Hello, world! 5489
```

Lets impact the real world

The ultimate aim of the coming adventure is for Pharo to interact with the real world. As a quick demo of this, lets mix this example into our existing *BlinkHello.ino*. Download the following into the ESP32 to blink the onboard LED at a speed we send it from Pharo.

```
// BlinkEcho.ino - minor control of the real world
int blinkPeriod = 1000;
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  Serial.print("Blink period = ");
  Serial.println(blinkPeriod, DEC);
}
void loop() {
  if (Serial.available() > 0) {
    blinkPeriod = Serial.parseInt();
    Serial.print("Blink period = ");
    Serial.println(blinkPeriod, DEC);
  };
  digitalWrite(LED_BUILTIN, HIGH);
  delay(blinkPeriod);
  digitalWrite(LED_BUILTIN, LOW);
  delay(blinkPeriod);
}
```

First test it from the Arduino IDE > Tools > Serial Monitor. Type 500 and [Send], then type 100 and [Send] to watch the LED change its rate of blinking. The monitor should show...

```
Blink period = 1000
Blink period = 500
Blink period = 100
```

Back in Pharo (after closing the Arduino IDE Serial Monitor), evaluate this...

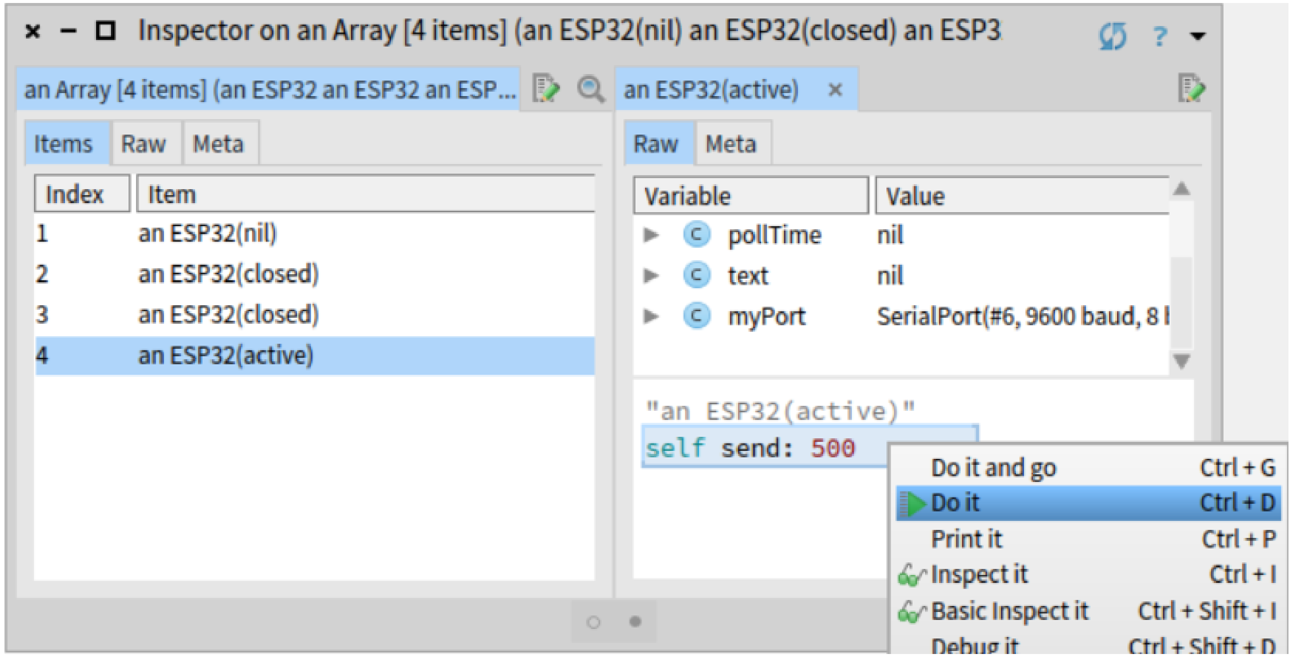
```
ESP32Morph open.
ESP32 allInstances inspect
```

You are probably looking at a list of several ESP32 objects. Hmm… its a bit hard to tell which is active one. Lets improve that! Add the following method to the **instance-side** of the ESP32 class (i.e. next to #initialize, #getText, #close, etc.)

Tip: After opening a new System Browser, enter "ESP" into the package filter text box.

```
printOn: aStream
        super printOn: aStream.
        aStream nextPutAll: '('.
        myPort isNil
                ifTrue: [ aStream nextPutAll: 'nil' ]
                ifFalse: [  myPort isClosed
                        ifTrue:  [ aStream nextPutAll: 'closed' ]
                        ifFalse: [ aStream nextPutAll: 'active' ]
                        ].
        aStream nextPutAll: ')'.
```

And [Refresh] the Inspector to see the impact of this. Select the active ESP32 and in the code pane of its [Raw] tab, evaluate "**self send: 500**"


Inspector on an Array [4 items] (an ESP32(nil) an ESP32(closed) an ESP3
an Array [4 items] (an ESP32 an ESP32 an ESP...
an ESP32(active)
Items
Raw
Meta
Raw
Meta

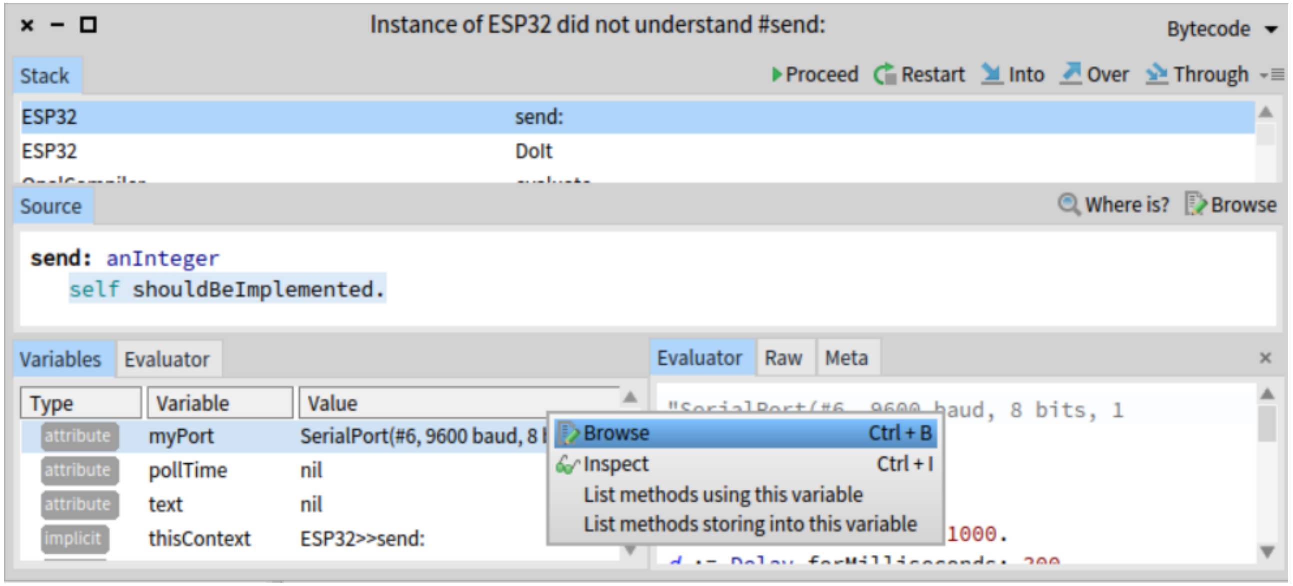
Index	Item
1	an ESP32(nil)
2	an ESP32(closed)
3	an ESP32(closed)
4	an ESP32(active)

Variable	Value
pollTime	nil
text	nil
myPort	SerialPort(#6, 9600 baud, 8 I

```
"an ESP32(active)"
self send: 500
```

Do it and go Ctrl + G
Do it Ctrl + D
Print it Ctrl + P
Inspect it Ctrl + I
Basic Inspect it Ctrl + Shift + I
Debug it Ctrl + Shift + D

Of course we haven't defined #send for the ESP32 yet, so we do our usual dance, click [Create], "ESP32", "as yet unclassified". Now the debugger is waiting at our newly created method. What should we write here? I'm not immediately sure, but its nice to have the method activation state available to us help find out. Scroll down to the "*myPort*" instance variable, right-click on it and choose [Browse].



Instance of ESP32 did not understand #send:

ESP32	send:
ESP32	DoIt

```
send: anInteger
    self shouldBeImplemented.
```

Type	Variable	Value
attribute	myPort	SerialPort(#6, 9600 baud, 8 b
attribute	pollTime	nil
attribute	text	nil
implicit	thisContext	ESP32>>send:

```
"SerialPort(#6, 9600 baud, 8 bits, 1
1000.
```

Up pops a System Browser on the SerialPort class and at a guess the “input/output” protocol is probably what we’re looking for. Yes, clicking on that provides a short list. The *#nextPutAll:* method looks promising. Back in the debugger lets try accepting…

```
send: anInteger
        myPort nextPutAll: anInteger printString.
```

and click [Proceed]. Then in the inspector, do “**self send: 100**” and check back in our Pharo ESP32 Serial Monitor where you should see…

```
Blink period = 500
Blink period = 100
```

and observe the change in the LED blink rate.

Well thats all for now. Hope you enjoyed. What else would you liked to have seen? Feel free to comment on the pharo-users mail list, if you prefer that over the comment section below.

This entry was posted in Pharo, Uncategorized. Bookmark the permalink.

← Pharo Libclang FFI, part 5, client data and recursive visitor/callbacks

Pharo PDF Rendering, part 1, building PDFium →

One Response to *An evening with Pharo and the ESP32 microcontroller*

Pingback: *An evening with Pharo and the ESP32 microcontroller | Weekly news about Pharo*

Leave a Reply

You must be logged in to post a comment.