The Hobbes User Guide

Version 0.4.2

Contents

0 Important Stuff

This is the important stuff (as the section header implies), so here goes:

0.0 Software

Hobbes runs on IronPython. IronPython runs on .NET. The Hobbes git repository, shamrock:s_hobbes, includes IronPython 2.7.0.40 libraries and DLLs, so even if you don't have IronPython installed, you'll be fine.

If you don't have .NET installed though, that's a bit more serious. The .NET Framework is most definitely mandatory. The most recent version (.NET Framework 4) is strongly recommended; it *might* work on older versions, but I've developed solely on .NET Framework 4 and you know how Microsoft is with backwards-(in)compatibility.

Lastly, you need a virtual COM port driver. But if you were using Tigger before, you should have had one beforehand.

0.1 Running Hobbes

I will attempt to make this as simple as possible.

- Clone the git repository from shamrock:s_hobbes
- Run Hobbes.bat in the cloned folder (preferably from a console)
- ???
- Profit!

0.2 Why is he counting from 0 anyway?

I'm a computer scientist.

1 The Actual Introduction Now

Hobbes (I haven't thought of anything it could stand for) is a program that communicates with a device through a serial port; more specifically, with Icron boards. It is written in IronPython, and runs on top of an IronPython environment. As a result, it is rather good at Python scripting. (This short introduction will do for now.)

2 Supported boards

Hobbes is designed solely for Goldenears (GE) support. Lionsgate (LG1) is also fully supported, as Hobbes contains Stewie file support as of version 0.4-rc1. If you have an older version of Hobbes, Hobbes will still be able to send iCommands to, receive iLogs from, and flash (using XMODEM) an LG1 board; it just won't be able to bootload them. Any other board hasn't been tested on Hobbes, and support would probably be somewhat icky for them.

3 Using Hobbes

This is probably the part of the guide you're looking for.

3.0 Starting Hobbes

See §0.1. Run Hobbes.bat. You should see a console start, and a startup window pop up.

If you have Hobbes 0.4-b30 or later, you may be prompted to load the previous session. This "session" is a JSON object of serial ports and .icron files. Upon (regular) exit, the values are taken from the Devices that were open and saved to Settings\PreviousSession.hbs. If this file exists at startup time, and it's not an empty dictionary (meaning there were no Devices open from the previous session), you'll be prompted to load the previous session. If you select "Yes", Hobbes will create the necessary Device(s), connect to the serial port(s), and load the .icron file(s).

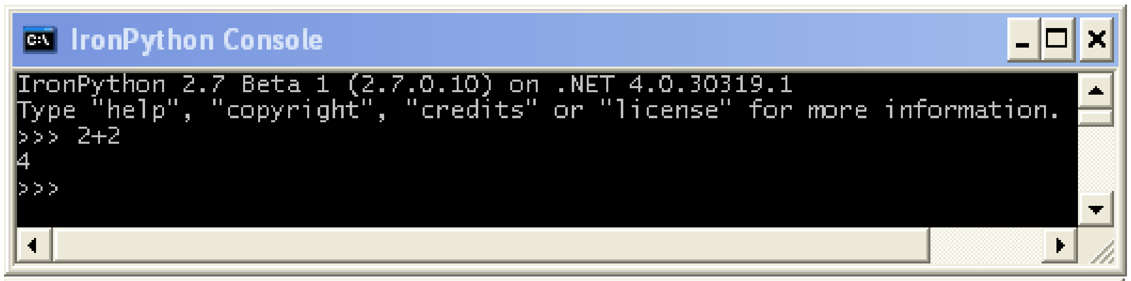
3.1 Hobbes uses a Python interpreter

Starting from Hobbes 0.4-b1, there has been a progression of GUI additions that mostly makes the following section obsolete. But this should still be of interest to you if you want to write Hobbes scripts, so I'll keep this here.

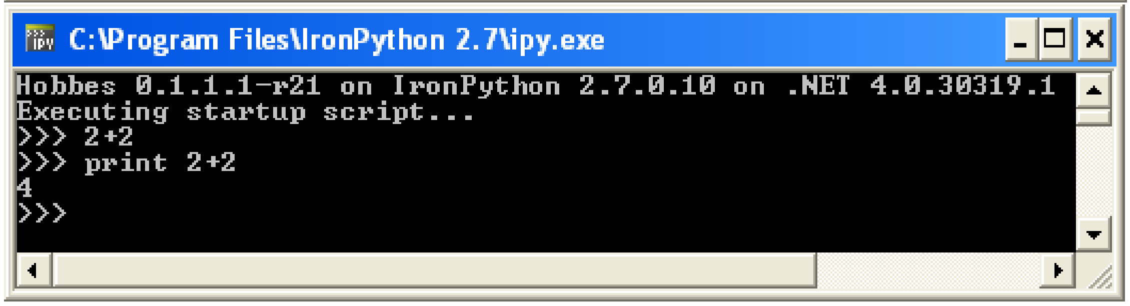
If you have no Python experience, you might be a bit lost. This is a user's guide for Hobbes, not Python–but I'll include all the useful Python interpreter stuff you can do here:

3.1.0 Create and execute Python code

Hobbes's interpreter is dynamic, so it runs Python code and such. It is, however, slightly different than a Python executable. The most important difference is that you'll probably have to prefix a command with "print" if you want to see its output, as shown below:



```
IronPython 2.7 Beta 1 (2.7.0.10) on .NET 4.0.30319.1
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>>
```



```
Hobbes 0.1.1.1-r21 on IronPython 2.7.0.10 on .NET 4.0.30319.1
Executing startup script...
>>> 2+2
>>> print 2+2
4
>>>
```

Disregarding that 2 + 2 = 5, of course (since Big Brother says so), note that the IronPython console did not require "print" to display 4 while Hobbes did. This apparently is a side effect of taking user input and executing it through an IronPython engine. But a) I don't want to spend the time figuring out when and when not to print and b) you might actually want suppressed output once in a while. In any case, if you want output and you don't see it, prefix your command with "print" and you should be fine.

(Python's code module would be very helpful here. Hobbes may use it in the future.)

3.1.1 Use the horrific Windows console "features"

They're pretty bad. You can't select anything simply by holding a button and dragging. Or get any context menu when you right-click. There *is* a bit of functionality, though; by clicking on the command prompt icon or right-clicking on the title bar, you get a little menu. And under "Edit", you'll see options for copying, marking (which you have to do before copying), and pasting.

3.1.2 The dir() function

This is a Python built-in function. If you know about this already, you can probably skip this. This function lists all the available functions and variables that can be called from a class, or an instance of a class. If you have, say, an iCommand class COM42_ICMD, and you don't remember the available functions under that class, you can use dir():

```
>>> print dir(COM42_ICMD)
['__doc__', '__module__', 'readMemory', 'readModifyWriteMemory', 'writeMemory']
```

(Note the usage of print.) The available functions are then readMemory, readModifyWriteMemory, and writeMemory. More on using these functions later.

3.1.3 The __doc__ string

(Again, users of Python will be familiar with this.) This string simply gives information about the class or function it is declared in. Hobbes uses this most commonly in its auto-generated Python code (more on that later), where the help strings of iCommands are the __doc__ strings. These are the same strings that were in the iCommand selector form in Tigger. For example, continuing on the example in the previous section:

```
>>> print COM42_ICMD.readMemory.__doc__
Read any memory address. Argument: full 32 bit address
```

(Again, print is required.) If you were to use Tigger's iCommand aliases and select component ICMD_COMPONENT and command readMemory H, you would see this.

3.1.4 Other stuff

There's other cool Python stuff (getattr(), unordered arguments, et al.), but I've found the ones above most useful for Hobbes. Go through a Python tutorial if you want to learn more.

(By popular demand, a link! Chapters 2 to 4 in `http://diveintopython.org/toc/index.html` should help. That's what I used.)

3.1.4.1.5 This is just a pi reference, you can ignore this

3.2 Using a Device, the (sort of) deprecated way

As the title of this section suggests, everything in this section has been mostly superseded by GUI windows as of the Hobbes 0.4 beta stream. However, you should still read through this section if you're interested in writing scripts for Hobbes, since you'll have to do a bunch of low-level stuff in said scripts.

Hobbes has a Device class that includes everything you need to communicate with an Icron board. Most importantly, it includes a .NET SerialPort class instance which connects to the external board.

3.2.0 Finding serial port names

To create a Device that's actually useful, you'll need to find your serial port names. You could do this in Hobbes, where there are (at least) three methods:

- Using the .NET SerialPort class

  The class has a nice static method that returns an array of all the available serial ports. I'll just use an example:

  ```
  >>> print SerialPort.GetPortNames()
  Array[str](('COM13', 'COM37', 'COM42'))
  ```

  Your serial ports are probably going to be different (mine definitely are), but you'll see the serial ports there.

- Using the builtin function in Hobbes

  Quite similar to the above method. The main difference is that a Python module (_winreg in Python 2, winreg in Python 3) is used to find the serial ports. It's a bit slower than SerialPort.GetPortNames(), but it'd be helpful if you want to port Hobbes to something that doesn't use .NET.

  ```
  >>> print availablePorts()
  ['COM13', 'COM37', 'COM42']
  ```

  Note that a list is returned, instead of a .NET array of strings. By the way, don't do this on a Mac. Or enter "sys.platform = 'darwin'; availablePorts()". You may be surprised at the output.

- Being lazy and letting Hobbes do the work for you

  Alternatively, you could create a Device with an errant port name:

  ```
  >>> dev = Device("This is definitely not a valid port name")
  Invalid serial port.
  Available serial ports:
  COM13
  COM37
  COM42
  ```

In any case, you should have a list of serial port names now.

3.2.1 Creating the Device, and doing useful things with it

There are two ways of going about this: the hard way, and the easy way.

3.2.1.0 The hard way Obviously, not the recommended way. You should skip this section and go on to reading about the easy way. But here it is.

1. Create a Device instance:
   ```
   >>> dev = Device("COM42")
   Port set to COM42
   Loading default settings
   ```

2. Set the Device's interpreter and interpreter scope:
   ```
   >>> dev.interpreter = hobbes.Calvin
   >>> dev.interpreterScope = hobbes.CalvinScope
   ```

3. Connect to the serial port:
   ```
   >>> dev.connect()
   02/16/11 12:50:57 Connected to port COM42: 115200 Bd, 8-N-1; handshake: None
   ```
   (Of course, you're going to see a different timestamp.)

4. Add the Device to deviceList:
   >>> hobbes.devices[“COM42”] = dev

5. Load a .icron file for the Device:
   >>> hobbes.load(“COM42”, “Z:\\designs\\goldenears\\working\\angusl\\p_goldenears_sw\\goldenears_fpga_bin\\goldenears_fpga.icron”)
   Loaded .icron file successfully
   Python code for serial port COM42 created

3.2.1.1 The easy way

1. Do everything that was done in the previous section in *one step*:
   >>> hobbes.load(hobbes.addDevice(“COM42”), “Z:\\designs\\goldenears\\working\\angusl\\p_goldenears_sw\\goldenears_fpga_bin\\goldenears_fpga.icron”)
   Port set to COM42
   02/16/11 12:50:57 Connected to port COM42: 115200 Bd, 8-N-1; handshake: None
   Loaded .icron file successfully
   Python code for serial port COM42 created

As you may have figured out, hobbes.addDevice() is *really* useful. It does a lot of the steps you would be doing manually in the previous section. In addition, it returns the argument you just passed in, so you can use it in hobbes.load(). Speaking of hobbes...

3.2.2 What’s “hobbes”, anyway?

Note this is “hobbes”, not “Hobbes”, the actual program. It’s the instance of the HobbesInterpreter class that you work with in the console. If you’re interested, in HobbesInterpreter.py there’s the following code:

self.CalvinScope.SetVariable(“hobbes”, self)

This allows the interpreter to call its own functions. These are addDevice(), load(), removeDevice(), and the constructor. They all have doc strings, so figuring out what happens in each of the functions is left as an exercise. (Side note: yes, you could call the constructor! But do you really want to? The only thing you’d get out of it is a good “yo dawg” joke and nothing more.)

Anyway, I’ll get back to stuff you can do with your shiny new Device.

3.2.3 Auto-generated Python code

You might have noticed this whenever you load a .icron file to the Device:

“Python code for serial port COM42 created”

What happened there? Well, before this, the load() function of the Device was called. This called bsdtar.exe a bunch of times to extract icron_header, icomponent, icmd, etc. After this, Python code is created, compiled, and executed, assuming the .icron file loaded.

I will spare you the gory details of what happens, and instead go on to what you could use the auto-generated Python code for:

3.2.4 Sending iCommands

As with creating a Device, there are some different ways of doing this. In all examples, I will use the iCommand of the Icron board connected to port COM42 to read the memory address at 0x00000000.

3.2.4.0 Self-torture Don't do it this way. Just don't.

```
>>> print dir(COM42_ICMD)
['__doc__', '__module__', 'readMemory', 'readModifyWriteMemory', 'writeMemory']
>>> print hobbes.devices["COM42"].iComponents
[All of the iComponents are printed here]
>>> for cmd in hobbes.devices["COM42"].iCommands["ICMD_COMPONENT"]:
...     print cmd
# This is done to find the index of the readMemory iCommand within the list of iCommands
# for ICMD_COMPONENT.
readMemory
writeMemory
readModifyWriteMemory
>>> hobbes.devices["COM42"].iCommands["ICMD_COMPONENT"][0].send(0)
# In DeviceWindow:
# "02/16/11 14:42:49 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

This is why I made Python code. I'm not going to bother explaining what's done here (except for the fact I made some comments). Still, the iComponents list may be useful if you need to see all of them at the same time.

3.2.4.1 Utilizing the Python code, Method I This involves creating an instance of one of the classes in the auto-generated Python code.

```
>>> icmd = COM42_ICMD()
>>> icmd.readMemory(0)
# In DeviceWindow:
# "02/16/11 14:42:49 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

3.2.4.2 Utilizing the Python code, Method II This is similar to Method I in the immediately preceding section, but no class instance is required this time:

```
>>> COM42_ICMD.readMemory(0)
# In DeviceWindow:
# "02/16/11 14:42:49 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

As you can tell, either method using Python code is much easier to remember and enter than the first one by far.

I should probably say *where* to enter these commands. Either the main Hobbes console or the input textbox of the DeviceWindow that pops up after a Device connects is acceptable– they both send Python code to the same interpreter (which, of course, I had to name Calvin). Currently, the only real good reasons that you should use the DeviceWindow instead of the console is that you can load Python scripts into the textbox with the "Load..." button, and copy/paste/cut is supported. With the main console, you have some functional history available, i.e. you can access previously-entered commands with the arrow keys. But of course, in either case, it's much easier to execute scripts than Tigger.

And I should say how to enter these commands, too. The class name (COM42_ICMD in the previous examples) is derived from a) the serial port name the Device is connected to and b) the iComponent name. So you have, in order:

1. the serial port name ("COM42" in the previous example);
2. an underscore; and
3. the iComponent name, without "_COMPONENT" ("ICMD" in the previous example).

In each of these classes, there are functions corresponding to each iCommand. Continuing with COM42_ICMD, one of the available iCommands is readMemory, and so you would call it like a Python function of a class (which it is!): COM42_ICMD.readMemory().

(Caveat: iComponents without any iCommands don't have auto-generated Python code for them. Frankly, there's no use for them.)

With COM42_ICMD.readMemory(), you can read the __doc__ string (see §3.1.3). And you could do Python stuff with it, as it is a Python function. You could even use __call__, although it'd be completely inane as you can just call readMemory() with its argument! But anyway, you probably don't want to bother with all that (I didn't either). The __doc__ string should help you figure out how to send the iCommand: just enter COM42_ICMD.readMemory(0) to read 0x00000000. And COM42_ICMD.readMemory(0x30000000) to read 0x30000000. And so on. Hexadecimal numbers are supported as well as decimal numbers; don't use strings though, unless there's a boolT argument that only accepts "TRUE" or "FALSE" (actually, the quotation marks are optional for boolT arguments). If you avoid strings, you should be able to avoid raising any exceptions. Of course, mind the argument type; you won't get away with entering -42 for a uint32 argument type, and you'll definitely raise an exception by doing that.

If you don't have Hobbes 0.4-b5 or later, the following won't apply to you. The variable "currentDevice" contains the name of the serial port that sends code to Calvin. So if you have two DeviceWindows, say one connected to COM13 and the other to COM37, entering "print currentDevice" from the latter will print "COM13" to the console, while "COM37" will be printed for the latter.

Also, there is a function get_iCommand() (as of Hobbes 0.4-b6) that takes a serial port name, iComponent name, and function name as arguments, and returns the iCommand corresponding to the arguments. You can combine this with currentDevice to dynamically retrieve iCommands without dependency on port names:

```
>>> get_iCommand(currentDevice, "ICMD", "readMemory")(0)
# In DeviceWindow:
# "02/16/11 14:42:49 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

3.2.5 Receiving iLogs

Bytes sent from the serial port are received. Assuming you aren't bootloading/flashing the board, the bytes are sent to a function that parses them into an iLog message. Once a message is complete, it is output to the DeviceWindow with a nice timestamp. For more information on modifying output and scrolling the output textbox, see §3.3.2.

Note that if an unknown byte is received, the console will output a message saying an unknown byte was received, along with the actual byte. I've noticed this happening when I cut power to a board, and somehow 0's are received. Sometimes there are 1's and 2's as well. Sometimes an unknown iLog is received as well. This might happen during startup of a board, when two 252 bytes are sent for some reason. Hobbes will figure out that there sure aren't 253 iComponents, and output a message to the console saying that an unknown iLog was received, and the bytes comprising that iLog in order. Hobbes is also very helpful, labelling each of the bytes according to their meaning in the iLog, i.e. labels like "Header byte", "Component index", "Message index", and so on.

3.2.6 Sending iCmdResps

This is only supported with GE version 2 .icron files and up. If you're not using those, this section will be useless to you.

iCmdResps are pretty neat structures that intertwine an iCommand and an iLog together. Its function is quite simple:

1. Send an iCommand.
2. Wait for a specific iLog; if it doesn't come in time, timeout.
3. Parse the iLog as usual.
4. Get a specific argument from the iLog.
5. Return the argument.

Sending one is similar to an iCommand.

Don't do this:

```
>>> print hobbes.devices["COM42"].iCmdResps["ICMD_COMPONENT"][0].respSend(0)
5452888
# In DeviceWindow:
# "02/16/11 16:36:04 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

Do this:

```
>>> print COM42RESP_ICMD.read32(0)
5452888
```

```
# In DeviceWindow:
# "02/16/11 16:36:04 ICMD: UsrLog: Read address 0x00000000: value 0x033fffc0"
```

I won't bore you with everything I said before regarding sending iCommands, because sending an iCmdResp is basically the same thing. The only difference is that the port name is immediately followed with "RESP" before the underscore. And the iCmdResp is named read32, not readMemory (although the iCmdResp *calls* the readMemory iCommand).

That "5452888" is the returned value of read32(). If you check, the hexadecimal value of that is 0x033fffc0, which happens to be the value we read! Currently, calculating checksums might not be too exciting (and probably won't work, if you manage to cause a fatal error while reading memory). But iCmdResps to come should be very exciting.

3.2.7 Bootloading and flashing

Click the "Bootload" button. Or the "Flash" button. That's all there is to it.

You might want to bootload/flash in a script, so I should mention that the function being called for XMODEM sending is sendFileByXMODEM() in the XMODEM module. The function takes two arguments: the first being the Device to flash to, and the second being the name of the file to send, as specified in the .icron file. (For example, to send flash_writer.bin you would enter "flash_writer" or "flash_writer2" as the second argument, depending on whether the board is LG1 or GE.) If for any reason the flashing of the file fails, the function will return False; if the flashing succeeds, it will return True. Note that the difference between clicking "Bootload" and clicking "Flash" is that clicking the latter will first send the first iCommand listed under TOPLEVEL_COMPONENT (assumed to be xmodem_new_image) before beginning flashing of the flash writer, then the firmware. So if you have a test harness loaded, and you want to get back to main firmware, you'll want to manually reset the board to bootload mode, as there probably wouldn't be an iCommand available to start sending NAKs.

As of Hobbes 0.4-rc1, Stewie files are now supported; ergo, LG1 bootloading is now possible. The function called to send a Stewie file is sendFileByStewie() in the Stewie module; just like its XMODEM counterpart, it takes a Device and file name as parameters. The file name should always be "flash_writer", as this corresponds to flash_writer.icr in all versions of LG1 .icron files.

There are a couple of interesting things that may happen when you're bootloading an LG1 board:

- When an LG1 board is powered on into bootload mode, it will send a bunch of bytes through the serial port signifying that the bootloader is running. Hobbes will print a message when these certain bytes are received; however, on occasion the board decides to throw in a high-value byte before the bootloader bytes. As a result, Hobbes may show messages saying an exception was raised, but in reality the board is in bootloading mode. You can either reset power until you see the nice message stating a Device is in bootloading mode, or just go ahead with clicking "Bootload". It doesn't really matter.

- When the Stewie file is transferred successfully, the Device will spit out a bunch of bytes again, consisting of a CR LF, the letters "OK", and another CR LF. Hobbes will also print out a neater message here, saying that bootloading was OK, but if the weird-exception-raised-thingy described above had happened, you'll see two unknown byte received messages: one for 13 and one for 10. No big deal, I hope.

3.2.8 Changing Device settings

Currently, the only Device settings are serial port settings. I don't see any reason why you would need to change the default settings, so don't.

3.2.9 Printing

When you use the print statement, regardless of whether the command was entered in the console or a DeviceWindow, the output will always go to the console. This is by design. I'm too lazy to redirect sys.stdout based on where a command was entered, so there is one function to print to console, one function to print to an arbitrary DeviceWindow, a different function to print to a different DeviceWindow, and so on. The functions to print to DeviceWindows are named after the serial ports they are connected to. The function name is the serial port name followed by "_OUT". So for a Device connected to COM42, the function to print "hello world" to its output textbox would be "COM42_OUT('hello world')".

Note: before Hobbes 0.4-rc9, the function would be "COM42_out()", with lowercase "out".

3.3 Using a Device, the mouse-intensive way

This section will pretty much describe the exact same things as §3.2 does, with the exception that all the GUI components are used instead. So if you don't want to write any scripts for Hobbes, then you'll just want to read this section.

3.3.0 Creating the Device, and loading a .icron file for it

As of Hobbes 0.4-b1, there is no longer any need to find out which serial ports you have manually. Hobbes does it for you!

Currently, the startup window (the one with an actual picture of Hobbes) does nothing more than create a Device and load a .icron file for it. There are two parameters:

- The serial port to connect to. This is why you don't need to figure out which ports you have! Also, Hobbes 0.4-b7 and later have a refresh button, so if you add or remove a serial port, you only have to hit the button to see a repopulated list.
- The .icron file to load. Clicking the open button will open a dialog where you can select your .icron file. (Note you can resize this dialog, so you can actually read all of those ridiculously long paths.) You could also edit the path in the textbox. In any case, whatever's in the textbox is the path used when loading the .icron file.

Upon clicking "Add Device", a DeviceWindow will pop up after a while, and its title will change to include the path of the .icron file that was loaded. This assumes that you entered a valid serial port and .icron file path, of course.

Hobbes 0.4-rc22 added a "Load .icron file" button. This allows the user to change .icron files of Devices that already exist, so there is no need to close and reopen the DeviceWindow. Just make sure the "Device:" entry is the serial port whose .icron file you want changed, select the new .icron file to load, and hit "Load .icron file".

3.3.1 Sending iCommands

If you have an open Device, and Hobbes 0.4-b11 or later, click "iCommands" on the bottom toolbar. A window will open.

This window, which I will henceforth refer to as the "iCommandWindow", has quite a few useful features. Of course, they all pertain to iCommands, hence the title of this section.

If you've used Tigger before, you should be somewhat familiar with this procedure. First select the iComponent on the far left (but note that unlike Tigger, you can see all the components at once, and you can sort them alphabetically!). If there are iCommands for the selected iComponent, they will be shown in the box labelled "Functions:" to the immediate right of the iComponent box. Once you select a function (no plan for alphabetical sorting here), a bunch of stuff on the right will be populated, or re-populated.

- If there are any arguments required by the iCommand, the argument boxes will be enabled. How many are enabled corresponds to the number of arguments required. Also, the type of each argument is displayed below each enabled box. Note that everything will be a number, except arguments of type boolT. They must be either "TRUE" or "FALSE".
- The help box will contain the help string of the iCommand.
- The code box will contain the Python code used to send the iCommand...but if you're not interested in scripting, you can ignore this.

Once you've entered all your arguments, you can do one of two things:

- Execute the code. This does some fancy stuff that ultimately results in bytes being sent to the Device's serial port. But you're just clicking one button, you lazy dog.
- Save the code to one of the buttons below. This is similar to Tigger's alias buttons. You'll probably want to set the "Save to button" and "Text" fields–the former sets which button to save to, with a range from 1 to 9 (counting left to right, top to bottom), and the latter sets the text on the button that the code will be saved to.

  A warning: if you save some things to all nine buttons, and then accidentally close the iCommandWindow, your aliases will be gone! Once you reopen the window, none of them will be there anymore. I may fix this eventually, but for now just make sure you don't accidentally close windows. And make sure you use the save/load aliases functions. Clicking "Save aliases..." will open a dialog where you can save your aliases (comprising of the text and code for each button) to a file. Clicking "Load aliases..." will open a dialog where you can load your aliases.

3.3.2 Receiving iLogs

Fully parsed iLogs are printed to the upper textbox (henceforth referred to as the output textbox). As of Hobbes 0.4-b13, there are two checkboxes on the toolbar immediately below the output textbox that control output:

- The “Suspend iLogs” checkbox. This prevents iLogs from being printed, and saves them to a cache (it’s a bit more complicated than that, but you won’t need to know here). As soon as you uncheck this, the suspended iLogs are all displayed at once, unless you’ve disabled some of them–more on that later.
- The “Auto Scroll” checkbox. This scrolls the textbox to the bottom automatically, as you may expect. Note that .NET is quirky and if you have focus on the output textbox (i.e. you’ve clicked inside of it), it’s going to scroll to the bottom automatically regardless of whether “Auto Scroll” is clicked or not.

Hobbes 0.2.0.0-r33 added the iLogWindow, a window that is opened from a DeviceWindow and allows manipulation of the display of iLogs. This is somewhat similar to the iCommandWindow: to access individual iLogs, select an iComponent from the leftmost list (which can be sorted alphabetically). Assuming there are iLogs for your chosen iComponent, they will show up in the rightmost list. You can select as many iLogs as you want in this list to modify them all at once. There are two settings for an iLog:

- The font used to display the iLog. By clicking “Font...”, you can modify the font itself, and its style, size, and colour. There is a preview textbox on the bottom of the iLogWindow, where you can see what the iLog will look like.

  Note: the default font used is Consolas. You may not have it. If you see a default font displayed that is definitely not monospaced, then you don’t have it.
- Whether the iLog is displayed at all. This is controlled by the “Enable iLog” checkbox. If unchecked, the iLog(s) selected will not be displayed at all in the output textbox.

3.3.3 Sending iCmdResps

See §3.2.6 for iCmdResps. Why aren’t they here? If you’re using Hobbes only for Tigger-like functionality, parsing an iLog to return a value to a function really isn’t that useful. Creating a GUI for iCmdResps would be completely unnecessary. And you’ll only be using iCmdResps if you’re scripting, and this is the non-scripting section.

3.3.4 Bootloading and flashing

Click the “Bootload” button if you see the Device sending NAKs. Or the “Flash” button if you know for sure that the first iCommand under TOPLEVEL_COMPONENT is named xmodem_new_image.

But if you’re connected to an LG1 device, and you have Hobbes 0.4-rc1 or later, click “Bootload” if you know it’s in bootloading mode and “Flash” if it’s in regular operation. See §3.2.7 if you want to know more (but you probably don’t).

3.4 Exiting Hobbes

You have a few options:

- Hitting the X button in the console
- Entering “exit” in the console
- Entering Ctrl-D in the console (the real EOF key combination, of course)

4 A conclusion of sorts

This is a temporary paragraph. Its only purpose is to serve as a reminder that this conclusion is far from complete, only the really important information needed to conclude the document is included, and ideally it will progressively become more and more complete.

And if you think that this sounds a tiny bit familiar, good for you.