The Hobbes Design Document

Version 0.0

Contents

0 Introduction **8**
- 0.0 Purpose . . . 8
- 0.1 Scope . . . 8
- 0.2 Terminology References . . . 8
- 0.3 Document References . . . 8

1 Design Overview **8**
- 1.0 Background Information . . . 8
- 1.1 System Evolution Description . . . 9
- 1.2 Current Process . . . 9
- 1.3 User Characteristics . . . 10
  - 1.3.0 User Problem Statement . . . 10
  - 1.3.1 User Objectives . . . 10
- 1.4 Proposed Process . . . 10
- 1.5 Constraints . . . 10
- 1.6 Design Trade-offs . . . 11

2 Design Architecture **11**
- 2.0 HobbesDriver . . . 11
- 2.1 HobbesInterpreter . . . 11
- 2.2 ErrorWindow . . . 11
- 2.3 Device . . . 11
- 2.4 XMLSettings . . . 11
- 2.5 iCommand . . . 12
- 2.6 iLog . . . 12
- 2.7 iCmdResp . . . 12
- 2.8 Stewie . . . 12
- 2.9 XMODEM . . . 12

3 External Module interface **12**
- 3.0 Use model . . . 12
- 3.1 API/Data Structures . . . 13

4 Detailed Design **13**
- 4.0 HobbesDriver . . . 14
  - 4.0.0 Classes . . . 15
  - 4.0.1 Functions . . . 15
  - 4.0.2 Variables and miscellaneous . . . 15
- 4.1 HobbesInterpreter . . . 15

4.1.0 Classes . . . 16
4.1.0.0 HobbesInterpreter . . . 16
4.1.0.1 StartupWindow . . . 16
4.1.1 Functions . . . 16
4.1.1.0 HobbesInterpreter . . . 16
4.1.1.0.0 __init__() . . . 16
4.1.1.0.1 add_device() . . . 17
4.1.1.0.2 remove_device() . . . 18
4.1.1.0.3 load() . . . 19
4.1.1.0.4 exit() . . . 20
4.1.1.1 StartupWindow . . . 20
4.1.1.1.0 __init__() . . . 20
4.1.1.1.1 icronFileButton_Click() . . . 21
4.1.1.1.2 DeviceRefreshButton_Click() . . . 21
4.1.1.1.3 addDeviceButton_Click() . . . 21
4.1.1.1.4 StartupWindow_Closed() . . . 21
4.1.1.1.5 StopDeviceOutputButton_Click() . . . 21
4.1.1.1.6 execute_code() . . . 21
4.1.1.1.7 run() . . . 21
4.1.2 Variables and miscellaneous . . . 22
4.1.2.0 hobbes_version . . . 22
4.1.2.1 program_path . . . 22
4.1.2.2 HobbesInterpreter . . . 22
4.1.2.2.0 devices . . . 22
4.1.2.2.1 Calvin . . . 22
4.1.2.2.2 CalvinScope . . . 22
4.1.2.2.3 startupWindow . . . 22
4.1.2.3 StartupWindow . . . 23
4.2 ErrorWindow . . . 23
4.2.0 Classes . . . 23
4.2.0.0 ErrorWindow . . . 23
4.2.1 Functions . . . 23
4.2.1.0 ErrorWindow . . . 23
4.2.1.0.0 __init__() . . . 23
4.2.1.0.1 closeButton_Click() . . . 23
4.2.2 Variables and miscellaneous . . . 23
4.2.2.0 ErrorWindow . . . 23
4.3 Device . . . 24
4.3.0 Classes . . . 24
4.3.0.0 Device . . . 24
4.3.0.1 DeviceWindow . . . 24
4.3.1 Functions . . . 24
4.3.1.0 crc16() . . . 24
4.3.1.1 availablePorts() . . . 25
4.3.1.2 Device . . . 26

4.3.1.2.0 __init__() . . . 26
4.3.1.2.1 connect() . . . 26
4.3.1.2.2 DataReceivedHandler() . . . 27
4.3.1.2.3 __parseNAK() . . . 28
4.3.1.2.4 icron_untar() . . . 28
4.3.1.2.5 load_icron_file() . . . 29
4.3.1.2.6 icronFileChanged() . . . 31
4.3.1.2.7 printToDeviceWindow() . . . 31
4.3.1.2.8 update_text() . . . 31
4.3.1.3 DeviceWindow . . . 31
4.3.1.3.0 __init__() . . . 31
4.3.1.3.1 suspchk_CheckedChanged() . . . 32
4.3.1.3.2 autochk_CheckedChanged() . . . 32
4.3.1.3.3 SendButton_Click() . . . 32
4.3.1.3.4 ClearInputButton_Click() . . . 32
4.3.1.3.5 SaveButton_Click() . . . 32
4.3.1.3.6 LoadButton_Click() . . . 32
4.3.1.3.7 iCommandsButton_Click() . . . 33
4.3.1.3.8 ReconnectButton_Click() . . . 33
4.3.1.3.9 ClearOutputButton_Click() . . . 33
4.3.1.3.10 iLogsButton_Click() . . . 33
4.3.1.3.11 ReloadIcronButton_Click() . . . 33
4.3.1.3.12 BootloadButton_Click() . . . 33
4.3.1.3.13 FlashButton_Click() . . . 33
4.3.1.3.14 XMODEMSend() . . . 34
4.3.1.3.15 StewieSend() . . . 34
4.3.1.3.16 DeviceWindow_Closed() . . . 34
4.3.1.3.17 execute_code() . . . 34
4.3.1.3.18 run() . . . 34
4.3.1.3.19 aliasButton_Clicked() . . . 35
4.3.1.3.20 dev_dumpLogCache() . . . 35
4.3.2 Variables and miscellaneous . . . 35
4.3.2.0 btldr_sequence . . . 35
4.3.2.1 btldr_OK_sequence . . . 35
4.3.2.2 Device . . . 35
4.3.2.2.0 icronFile . . . 35
4.3.2.2.1 iComponents . . . 35
4.3.2.2.2 iCommands . . . 35
4.3.2.2.3 iLogs . . . 35
4.3.2.2.4 iCmdResps . . . 36
4.3.2.2.5 Discourse on the preceding variables . . . 36
4.3.2.2.6 interpreter . . . 36
4.3.2.2.7 interpreterScope . . . 36
4.3.2.2.8 More discourse, this time on multiple interpreters . . . 36
4.3.2.2.9 flashWaitingByte . . . 37

4.3.2.2.10 flashSync . . . 37
4.3.2.2.11 flashNAK . . . 37
4.3.2.2.12 serialPort . . . 37
4.3.2.2.13 devWindow . . . 37
4.3.2.2.14 deviceSettings . . . 37
4.3.2.2.15 validPortName . . . 38
4.3.2.2.16 path . . . 38
4.3.2.2.17 programPath . . . 38
4.3.2.2.18 iLogFinished . . . 38
4.3.2.2.19 numArgs . . . 38
4.3.2.2.20 headerBits . . . 38
4.3.2.2.21 messageBytes . . . 38
4.3.2.2.22 messageLength . . . 38
4.3.2.2.23 logCache . . . 38
4.3.2.2.24 customFonts . . . 38
4.3.2.2.25 btldr_index . . . 38
4.3.2.2.26 btld_OK_index . . . 39
4.3.2.3 DeviceWindow . . . 39
4.3.2.3.0 dev . . . 39
4.3.2.3.1 scroll . . . 39
4.3.2.3.2 suspend . . . 39
4.4 XMLSettings . . . 39
4.4.0 Classes . . . 39
4.4.0.0 XMLSettings . . . 39
4.4.1 Functions . . . 39
4.4.1.0 XMLSettings . . . 39
4.4.1.0.0 __init__() . . . 39
4.4.1.0.1 set() . . . 39
4.4.1.0.2 get() . . . 40
4.4.1.0.3 save() . . . 40
4.4.1.0.4 load() . . . 40
4.4.1.0.5 output() . . . 40
4.4.1.0.6 outputList() . . . 40
4.4.1.0.7 outputCategories() . . . 41
4.4.2 Variables and miscellaneous . . . 41
4.4.2.0 XMLSettings . . . 41
4.4.2.0.0 BasicXml . . . 41
4.4.2.0.1 doc . . . 41
4.5 iCommand . . . 41
4.5.0 Classes . . . 41
4.5.0.0 iCommand . . . 41
4.5.0.1 iCommandWindow . . . 41
4.5.1 Functions . . . 41
4.5.1.0 iCommand . . . 41
4.5.1.0.0 __init__() . . . 41

4.5.1.0.1 __repr__() . . . 42
4.5.1.0.2 send() . . . 42
4.5.1.1 iCommandWindow . . . 43
4.5.1.1.0 __init__() . . . 43
4.5.1.1.1 sortedCheckBox_CheckedChanged() . . . 44
4.5.1.1.2 iComponentsListBox_SelectedIndexChanged() . . . 44
4.5.1.1.3 functionsListBox_SelectedIndexChanged() . . . 44
4.5.1.1.4 argTextBox_TextChanged() . . . 44
4.5.1.1.5 saveButton_Clicked() . . . 44
4.5.1.1.6 executeButton_Clicked() . . . 44
4.5.1.1.7 execute_code() . . . 45
4.5.1.1.8 saveAliasesButton_Clicked() . . . 45
4.5.1.1.9 loadAliasesButton_Clicked() . . . 45
4.5.1.1.10 run() . . . 45
4.5.2 Variables and miscellaneous . . . 45
4.5.2.0 iCommand . . . 45
4.5.2.0.0 functionSplit . . . 45
4.5.2.0.1 helpSplit . . . 45
4.5.2.0.2 argsSplit . . . 45
4.5.2.0.3 function . . . 45
4.5.2.0.4 help . . . 45
4.5.2.0.5 argTypes . . . 45
4.5.2.0.6 compIndex . . . 46
4.5.2.0.7 commIndex . . . 46
4.5.2.0.8 port . . . 46
4.5.2.0.9 args . . . 46
4.5.2.0.10 commandBytes . . . 46
4.5.2.1 iCommandWindow . . . 46
4.5.2.1.0 dev . . . 46
4.5.2.1.1 listToUse . . . 46
4.5.2.1.2 sortedComponents . . . 46
4.6 iLog . . . 46
4.6.0 Classes . . . 46
4.6.0.0 iLog . . . 46
4.6.0.1 iLogWindow . . . 46
4.6.1 Functions . . . 46
4.6.1.0 dummyFunction() . . . 46
4.6.1.1 iLogParseByte() . . . 47
4.6.1.2 output() . . . 47
4.6.1.3 printLogCache() . . . 48
4.6.1.4 dumpLogCache() . . . 49
4.6.1.5 iLog . . . 49
4.6.1.5.0 __init__() . . . 49
4.6.1.5.1 __repr__() . . . 49
4.6.1.6 iLogWindow . . . 49

4.6.1.6.0 __init__() . . . 49
4.6.1.6.1 iComponentsListBox_SelectedIndexChanged() . . . 49
4.6.1.6.2 sortedCheckBox_CheckedChanged() . . . 50
4.6.1.6.3 iLogsListBox_SelectedIndexChanged() . . . 50
4.6.1.6.4 enabledCheckBox_CheckedChanged() . . . 50
4.6.1.6.5 fontButton_Click() . . . 50
4.6.1.6.6 run() . . . 50
4.6.2 Variables and miscellaneous . . . 50
4.6.2.0 programPath . . . 50
4.6.2.1 outputHandler . . . 50
4.6.2.2 outputLogger . . . 50
4.6.2.3 iLog . . . 51
4.6.2.3.0 enabled . . . 51
4.6.2.3.1 font . . . 51
4.6.2.3.2 fontColour . . . 51
4.6.2.3.3 args . . . 51
4.6.2.3.4 events . . . 51
4.6.2.3.5 name . . . 51
4.6.2.3.6 message . . . 51
4.6.2.4 iLogWindow . . . 51
4.6.2.4.0 dev . . . 51
4.6.2.4.1 listToUse . . . 51
4.7 iCmdResp . . . 51
4.7.0 Classes . . . 53
4.7.0.0 iCmdResp . . . 53
4.7.0.1 ThreadSet . . . 53
4.7.1 Functions . . . 53
4.7.1.0 iCmdResp . . . 53
4.7.1.0.0 __init__() . . . 53
4.7.1.0.1 respSend() . . . 53
4.7.1.0.2 startSetThread() . . . 54
4.7.1.1 ThreadSet . . . 54
4.7.1.1.0 __init__() . . . 54
4.7.1.1.1 run() . . . 54
4.7.2 Variables and miscellaneous . . . 54
4.7.2.0 responseSplit . . . 54
4.7.2.1 commandSplit . . . 54
4.7.2.2 logSplit . . . 54
4.7.2.3 argSplit . . . 54
4.7.2.4 iCmdResp . . . 54
4.7.2.4.0 response . . . 54
4.7.2.4.1 command . . . 55
4.7.2.4.2 log . . . 55
4.7.2.4.3 argIndex . . . 55
4.7.2.4.4 device . . . 55

4.7.2.4.5 lock . . . 55
4.7.2.4.6 sender . . . 55
4.7.2.4.7 iLogSetThread . . . 55
4.7.2.4.8 msg . . . 55
4.7.2.5 ThreadSet . . . 55
4.7.2.5.0 iCmdResp . . . 55
4.8 Stewie . . . 55
4.8.0 Classes . . . 55
4.8.1 Functions . . . 55
4.8.1.0 sendFileByStewie() . . . 55
4.8.2 Variables and miscellaneous . . . 56
4.9 XMODEM . . . 56
4.9.0 Classes . . . 56
4.9.1 Functions . . . 56
4.9.1.0 sendFileByXMODEM() . . . 56
4.9.1.1 send_XMODEM_raw_binary() . . . 57
4.9.2 Variables and miscellaneous . . . 57

5 Test Plan **57**

0 Introduction

Hobbes is an application that communicates with a target via a serial port, where the target is presumably an Icron board. At the most basic level, Hobbes is capable of sending commands and firmware to the target, and receiving messages from it.

0.0 Purpose

The purpose of this document is to describe the design and functionality of Hobbes.

0.1 Scope

This document will mostly concern the design of Hobbes. A somewhat abstract view of Hobbes will be taken to allow a variety of implementations. As Hobbes is currently written in IronPython, though, many examples will be given in that language.

0.2 Terminology References

- Hobbes: the program itself
- Hobbes 0: the current implementation of Hobbes in IronPython
- Calvin: the Python interpreter running in Hobbes
- LG1: Lionsgate project
- GE: Goldenears project
- XML: Extensible Markup Language
- JSON: JavaScript Object Notation

0.3 Document References

The only real references here are the Python standard library and the .NET library.

1 Design Overview

1.0 Background Information

This document is about the design of Hobbes, hence the name of this document being "The Hobbes Design Document". One wishing only to learn how to use Hobbes should not read this document; the User's Guide is provided for that purpose. This document is better suited towards one that wishes to modify, improve, or even completely rewrite, Hobbes.

1.1 System Evolution Description

By no means is Hobbes complete. By no means is it completely stable. There are a bunch of things that can be improved:

- Ongoing GUI improvements. There will probably always be some seemingly tedious task that can be simplified by the addition of a button/checkbox/etc. There is a trade-off in speed, though.
- Other projects. At the moment, anything other than LG1 and GE is not supported well in Hobbes.
- Other file transfer protocols. Although unlikely, Hobbes may need to support things other than Stewie and XMODEM in the future.
- Saving data. Currently, Hobbes uses a mix of XML and JSON to store and load data. JSON is just as readable as XML and is much more lightweight, so XML should be phased out.
- Porting Hobbes to a different language. Hobbes is currently implemented in IronPython, a flavour of Python that allows easy integration with the .NET Framework. Although this is useful (basically having two completely difference libraries available), it's a bit awkward; it also doesn't help that I was lazy in some parts of Hobbes by copying Tigger code and correcting the syntax (e.g. when running bsdtar.exe on a .icron file). In addition, Windows is apparently terrible at serial port communication, so that may have a hand in Hobbes occasionally freezing when receiving a ton of bytes.

  If Hobbes is ported, it should be to CPython. Certainly, it would be the easiest to port to; with the pySerial module, serial port communication can be accomplished with .NET. The Python standard library should be large enough to do everything else–for example, the bz2 module can replace bsdtar.exe and the whole Process/ProcessStartInfo thing in Tigger and Hobbes. The current GUI layer of Hobbes is written in IronPython, but calls all the System.Windows.Forms stuff in .NET; with a CPython-written Hobbes one could use Python 3, PyGObject, GTK+ 3, and Glade, which is probably better suited towards Linux development.

1.2 Current Process

The program to be phased out is Tigger, itself a replacement for an even older program named Leo. Basically, Tigger does everything Hobbes does, except a) for the most part slower and b) without any scripting capabilities.

Here's Tigger's process, from its own design document:

"The new process is also a C# application which includes the capability of having multiple serial connections simultaneously allowing for simultaneous communication with multiple targets. The new solution also includes multiple protocols for programming a target, including binary flash, stewie and xmodem 1k. It also allows for different protocols for parsing data from a target, and currently parses using two protocols, the one included in the original Leo and the new ilog system."

1.3 User Characteristics

1.3.0 User Problem Statement

Basically, Hobbes was created in an (extremely successful, if I may say so myself) attempt to add a Python component to Tigger, adding a scripting capability to communication with Icron boards.

1.3.1 User Objectives

As Hobbes is a replacement for Tigger, its design will contain the objectives of Tigger:

- Easy to use
- Convenient
- Utilize a quick protocol for programming targets
- Allow for multiple protocols for programming and logging, and enable easy inclusion of new protocols
- Not crash frequently at all
- Not cause user frustration and pain

In addition, Hobbes contains additional objectives for its scripting layer:

- Provide a functional Python (or other scripting language) interpreter
- Allow scripts for said interpreter to be saved, loaded, and executed

1.4 Proposed Process

Software development since Tigger was released have resulted in many features of Tigger becoming obsolete. Some examples are XMODEM 1K, binary flash, a bunch of serial port options, and Leo-era logging. Hobbes strips these unnecessary components of Tigger away, and adds a command-line interpreter for scripting.

1.5 Constraints

Since Hobbes is currently implemented using the .NET Framework, it uses the SerialPort class for connection to serial ports, and thus presumably has the serial port disconnect issue that Tigger apparently had. Using PySerial and/or Linux may help.

GUI addons have slowed Hobbes's IronPython implementation down significantly; the biggest factor is probably an iLog setting the font and font colour of the output textbox when it's displayed, regardless of whether custom fonts are being used at all. In addition, scrolling and truncating text in the output textbox is somewhat awkward in .NET. Using a GtkTextBuffer in GTK+ (with GtkTextTag for custom fonts) would probably be easier.

In addition, the current interpreter is not quite at the level of a Python interpreter; that is, generally to see output one has to prefix a statement with "print". For example, while the input "2 + 2" would output 4 in a CPython/IronPython interpreter, Hobbes's interpreter will output nothing unless the input is "print 2 + 2".

1.6 Design Trade-offs

Generally, more GUI components means slower execution speed. Hobbes was originally designed to focus only on scripting; as development progressed, GUI components were added to satisfy the demands of the non-scripting user.

2 Design Architecture

2.0 HobbesDriver

Does nothing more than create an instance of the HobbesInterpreter class and catches any exception, displaying an error window if one is raised. This code should be able to reside in HobbesInterpreter without any trouble, so implementing this module is somewhat optional.

2.1 HobbesInterpreter

User interaction through the console should take place in this module, as the Python engine Calvin is here. There is also a StartupWindow class that contains, well, a startup window.

2.2 ErrorWindow

Pops up (usually) when something bad happens. Writes information on the exception that was raised to an error log.

2.3 Device

Provides the actual connection to the serial port, so this is probably the most important module in Hobbes. The DeviceWindow class in this module allows input to Calvin, and prints iLog output from the serial port to a textbox. Any GUI components to aid user communication to/from the serial port is in this window.

2.4 XMLSettings

Uses XML to save and load settings for other modules. However, if Python is used, a dictionary or list should suffice, and JSON can be used to save settings to disk. As a result, this module should be deprecated.

2.5 iCommand

Provides facilities to write to the serial port, signifying a command to execute. Optionally contains an iCommandWindow class that allows the user to interactively select an iCommand to send or save to an alias.

2.6 iLog

Provides facilities to display iLogs received from the serial port. Optionally contains an iLogWindow class that allows the user to modify the font and colour used to display each individual iLog when it is received.

2.7 iCmdResp

Only useful for scripting users. Amalgamates an iCommand and iLog such that executing a function corresponding to an iCmdResp executes an iCommand, waits for a specific iLog to be received, parses a specific argument from said iLog, and returns the argument.

2.8 Stewie

Sends an image file to a board using the Stewie file transfer protocol.

2.9 XMODEM

Sends an image file to a board using the XMODEM file transfer protocol.

3 External Module interface

3.0 Use model

There are sort of two layers to Hobbes:

- *The "Calvin and Hobbes" layer.* In the current implementation of Hobbes, the instance of the HobbesInterpreter class being run is named "hobbes" and the actual Python engine is named "Calvin"; hence the name. This layer consists solely of a command prompt and enough functions and variables to do anything to a Device. Note that iLogs automatically go to a window, so you won't see them if you purely work within the console.

- *The "Watterson" layer.* That is, the pretty GUI stuff. Note that Hobbes focusses on scripting, so clicking a button in a GUI component should in fact call a function in some code sent to the interpreter. GUIs are overrated. The GUI is should only be a convenience to the user; it should not be able to do anything that scripting cannot.

  Hobbes 0 uses a function to execute code from a GUI that looks similar to this:

```
def execute_code():
    '''Executes code using a Python engine.'''
    # compile code, and attempt to execute
    source = python_engine.CreateScriptSourceFromString(
             python_code, SourceCodeKind.Statements)
    try:
        source.Execute(python_scope)
    except Exception as ex:
        # show the error that's occurred
        eo = python_engine.GetService[ExceptionOperations]()
        error = eo.FormatException(ex)
        MessageBox.Show(error)
```

There are a few different implementations of this in Hobbes 0 to allow for different variables replacing "python_engine", "python_scope", and "python_code". They all point to the same Python engine, scope, and code, though. And of course, a CPython implementation of Hobbes would have a totally different exception block.

3.1 API/Data Structures

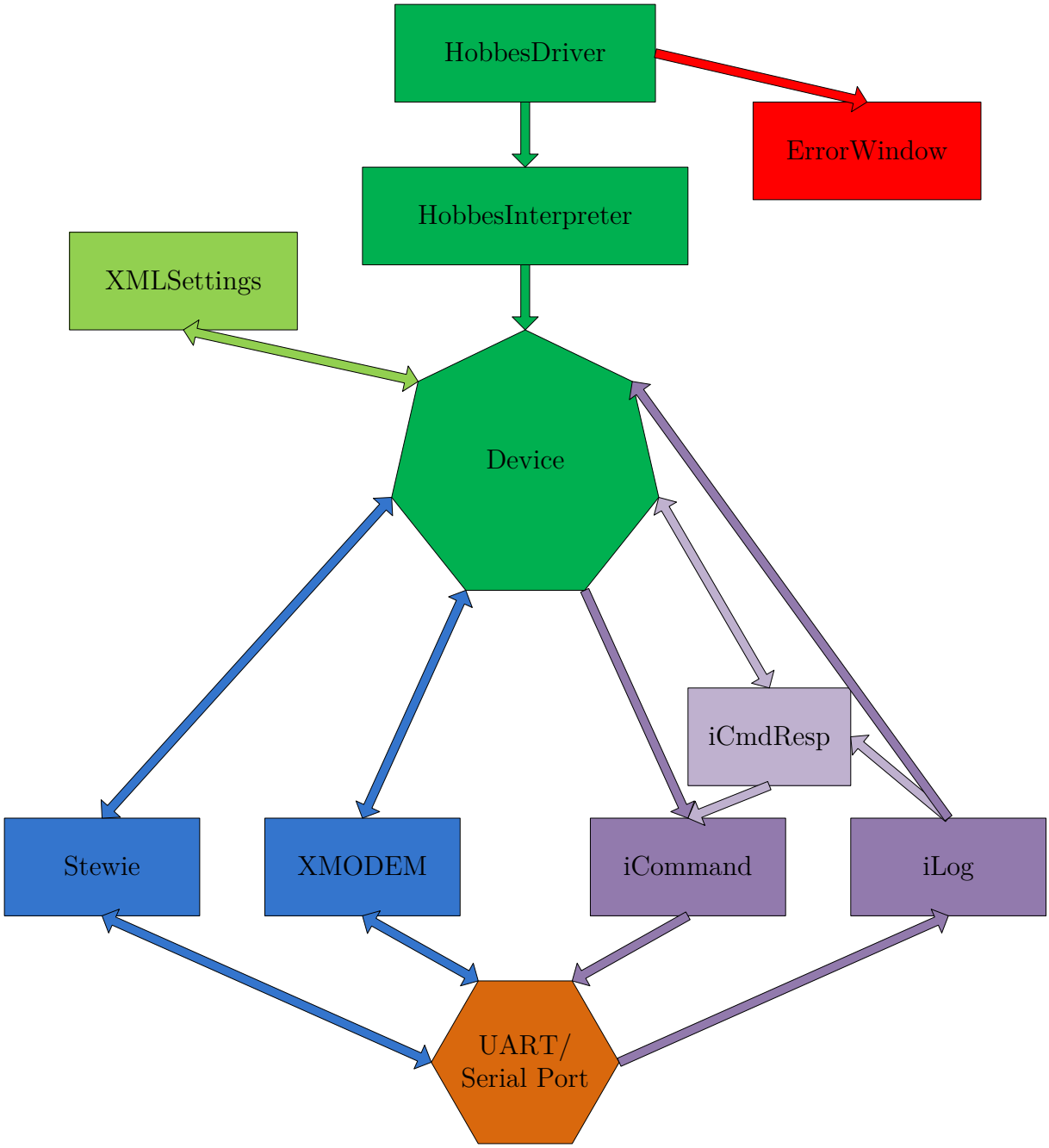
See §4 for everything regarding APIs and data structures.

4 Detailed Design

The following diagram details the interaction between all the modules in Hobbes 0.

Here's a summary of each module first:

HobbesDriver
HobbesInterpreter
ErrorWindow
Device
XMLSettings
iCommand
iLog
iCmdResp
Stewie
XMODEM



4.0 HobbesDriver

When this module is run (e.g. "ipy.exe HobbesDriver.py"), the Hobbes program starts. This should be the only method of opening the program, for simplicity.

This module should do nothing more than start the program in a try/except block. If an exception is raised, an error window (§4.2) is created and displayed.

In Hobbes 0, this module starts the program by creating an instance of the HobbesInterpreter class. Nothing is actually done with the class instance; the main loop is in the __init__

function of the instance.

If desired, this module can be phased out easily by moving all of the code in this module to the HobbesInterpreter module and renaming it HobbesMain or something similar.

4.0.0 Classes

None.

4.0.1 Functions

None.

4.0.2 Variables and miscellaneous

None.

4.1 HobbesInterpreter

The program should run in a main loop here, either from this module itself or from a driver module if it exists, as Hobbes 0 does. The interpreter should do the following:

1. Prompt the user for input.
2. If the user enters a command that requires more input (e.g. the first line of a for loop), prompt for more input until an empty line is entered.
3. Execute the input, displaying an error if one is encountered.

This is basically what the Python interpreter does. Hobbes 0's interpreter is slightly different; the User's Guide has more information on that.

The interpreter must be capable of doing the following:

- Loading (optional) scripts at startup. This should be no more than running code from a specified file. In Hobbes 0, the file is StartupScript.py in the program directory.
- Maintaining Devices. This includes adding Devices, removing Devices, loading .icron files for Devices, and creating Python wrappers for the Devices.
- Saving anything that can be used in a later session before exiting (normally, i.e. when the user requests an exit without any crash occuring).

Optional GUI component: Instead of the console, there could be a window that serves as the main window of Hobbes, similar to the main window of Tigger. This window would do everything the interpreter needs to do by simply sending code to the interpreter once a certain button is clicked. This is implemented as the StartupWindow class in Hobbes 0.

4.1.0 Classes

4.1.0.0 HobbesInterpreter Contains the functions that do the tasks required by the interpreter. An instance of this class is created in HobbesDriver in Hobbes 0.

This class actually is not terribly useful in Hobbes 0. The class could be removed and the functions renamed to something else if desired.

4.1.0.1 StartupWindow Provides a GUI to call the functions in HobbesInterpreter.

4.1.1 Functions

4.1.1.0 HobbesInterpreter Functions in the HobbesInterpreter class:

4.1.1.0.0 __init__() *Arguments:* self - the class instance, as per Python conventions; nothing if the HobbesInterpreter class does not exist (note: I won't mention self again, since I'd otherwise have to mention it for nearly every other function)

Returns: None (i.e. the Python equivalent of null)

Starts the Hobbes interpreter. Since this function is in a class, this is called when a HobbesInterpreter class instance is created. If this class is not implemented, the function should have a name like hobbes_start() or something similar.

This function should do the following:

1. Print a startup message. Obviously, this is optional, but it would be nice to show the version number.

2. Execute startup code. This should include importing everything in the other modules so they're accessible in the interpreter, a startup script that can be edited by the user, and any other functions that may be of use. In Hobbes 0, a function of use that is imported is get_iCommand(), described in the following section.

3. Open a startup window. This is StartupWindow in Hobbes 0. Again, this is optional, but if there is a startup window it should be started here. Also, a new thread or process must be created so the main interpreter thread doesn't lock. Hobbes 0 requires this thread to have some sort of STA state, as apparently .NET crashes otherwise.

4. Load a previous session, if it exists. Again, this is optional, and is just a convenience to the user (albeit a major convenience). The session can contain information on the serial ports Hobbes was connected to, the .icron files loaded for the Devices connected to the ports, and the location and size of the window. This session data can be saved when Hobbes exits normally.

   Hobbes 0 does this with the JSON format, and saves the aforementioned information in Settings\PreviousSession.hbs of the program directory with the following format:

```
{
    [SERIAL PORT NAME (string)]: [
        [
            [LOCATION: X COORDINATE (integer)],
```

```
            [LOCATION: Y COORDINATE (integer)]
        ],
        [
            [SIZE: WIDTH (integer)],
            [SIZE: HEIGHT (integer)]
        ],
        [PATH OF .ICRON FILE (string)]
    ]
}
```

5. Begin an REPL (Read, Evaluate, Print Loop). A normal exit from the console would consist of the user entering "exit", "hobbes.exit()", or Control-D in the console. Note there are other possibilities, e.g. "import os; os._exit(0)" in Python, but these aren't considered "normal" because they don't save the session before exiting.

6. Lastly, save the session. As mentioned immediately above, only "normal" exits will perform this step. Saving the session consists of getting the location and size of each DeviceWindow, closing them, getting the paths of the Devices' .icron files, then writing them all to a dictionary. The dictionary is then written to Settings\PreviousSession.hbs in the program directory (of course, this can have some other path, as long as in step 4, Hobbes is looking for the session information in the right place).

4.1.1.0.1 add_device() *Arguments:* port - the name of the serial port Hobbes should create a Device to connect to

Returns: The Device that was created, assuming that it was created successfully

Does what you probably think it would do: add a Device. Note that Hobbes 0 implements this as addDevice(). Really, it's all up to whether you prefer identifiers with underscore_delimiters_with_all_lowercase or camelCase. This document will switch back and forth; Hobbes 0 also uses both because I switched from the latter to the former after developing most of Hobbes 0, so most of the newest additions use the former.

This function does the following:

1. Create a Device, using the port argument in its constructor

2. Check to see if the Device connected to a valid port name. If not, don't do anything with this Device and return.

3. Check to see if the port the new Device connects to is already taken up by another Device. In Hobbes 0, the port names are in the keys of the devices dictionary, so it checks by building a list comprehension and seeing if it's empty:

```
# check to see if there is a Device that already connects to port
# list comprehension used, of course:
# name in self.devices.keys() iterates through all port names
# if name in [port] returns True (populating the list output)
#     if there is a name match
```

```
# the leftmost name can really be anything else; as long as there is
#     something in the list, the conditional will return True
if [name for name in self.devices.keys() if name in [port]]:
    # don't mess anything up and exit the function
    print DateTime.Now.ToString('MM/dd/yy HH:mm:ss.ff ') + \
          'The device already exists!'
    return
```

4. Set the Device's interpreter and interpreter scope correctly if needed. If the Device can access the interpreter without setting references to it, this step is unnecessary. As it is, Hobbes 0 requires this step:

```
# point the interpreter and interpreter scope to the correct places
device.interpreter = self.Calvin
device.interpreterScope = self.CalvinScope
```

5. Connect the Device to the serial port. This can just consist of calling the appropriate function of the Device.

6. Save the port name and Device somewhere. In Hobbes 0, this means adding an entry to a dictionary whose keys, as port names, correspond to values of references to Devices.

7. Finally, return the newly-created Device. This will be useful if one wants to load a .icron file for the Device immediately after creating it.

4.1.1.0.2 remove_device() *Arguments:* port - the name of the serial port connected to the Device to be removed

Returns: None

Again, the function name should be self-explanatory. This function removes a Device. In Hobbes 0, this is removeDevice().

This function does the following:

1. Try to get a reference to the Device to remove; if this fails, tell the user and exit. This would probably be something like getting a reference from the Device name repository in a try-except block, and exiting if an exception is raised.

2. If the DeviceWindow of the Device hasn't been already closed, close it. Of course, one could also attempt closing the window in a try-except block, and ignoring the exception that would be raised if the window was already closed.

3. If the serial port of the Device hasn't been already closed, close it. Basically the same thing as above with the try-except option.

4. Remove the port name and Device. In Hobbes 0, this is "del self.devices[port]", utilizing the Python del function. Here, there should probably be some sort of notification to the user that the Device had been removed.

4.1.1.0.3 load() *Arguments:* self; dev - the Device to load the .icron file; path - the path of the .icron file

Returns: None

Loads a .icron file from the specified path for the specified Device. If this succeeds, Hobbes goes on to create Python wrappers for the Device's iCommands and iCmdResps. Note that load() is a rather basic function name, so if one were to remove the HobbesInterpreter class she'd probably want to rename this to something more specific like load_icron_file().

This function firstly calls the load_icron_file() function of the Device, with the .icron file path as the argument. If this function raises an exception for any reason, Python wrappers will not be created.

The first Python function created is one that prints to the output textbox of a DeviceWindow. It appends text to the textbox, then scrolls the text to the bottom if the user desires. This is the function in Hobbes 0:

```
def {0}_OUT(output):
    '''Print to the DeviceWindow's output textbox.
    Argument: output - the text to print'''
    textbox = hobbes.devices['{0}'].devWindow.OutputTextBox
    textbox.AppendText(str(output) + '\\r\\n')
    # scroll if required
    if hobbes.devices['{0}'].devWindow.scroll:
        textbox.Select(len(textbox.Text), 0)
        textbox.ScrollToCaret()
```

Note that 0 is the port argument. This code is passed to Calvin and executed, so each Device has a unique ...._OUT() function.

Next, Hobbes should figure out whether iCmdResp wrappers need to be created. iCmdResps are a recent addition, and are only present in GE .icron files with version number 2 and above. Reading information from icron_header in the .icron file will be sufficient.

Hobbes now creates Python wrappers for iCommands. The following rules are followed:

- Each iComponent that has iCommands corresponds to a Python class.
- Each iCommand corresponds to a Python function.
- Each help string of an iCommand corresponds to the doc string of the Python function.
- Each Python function does nothing more than call the send() function of an iCommand.
- Each Python function is static, i.e. a class instance does not necessarily have to be created to call functions in the class.

And each Python function should look something like what Hobbes 0 does:

```
# class [dev port]_[compStr]:
#     def [cmdStr]([args]):
#         '''[documentation]'''
#         hobbes.devices[port].iCommands["[compStr]_COMPONENT"][
```

```
#             cmdIndex].send([args])
#     [cmdStr] = staticmethod([cmdStr])
```

Depending on how iComponents and iCommands are implemented on the Device, the actual function body may vary; however, it should be clear that there is only one task to do in the function. The various [] fields are filled in using information from the Device, and iterating through each of the iComponents. The last line with "staticmethod" is used to make the function static.

This code isn't executed yet if we still have iCmdResp wrappers to create. They look virtually the same as iCommands, with the biggest exception being that they must return a value (again, this is directly copied from Hobbes 0):

```
# class [dev port]RESP_[compStr]:
#     def [cmdStr]([args]):
#         '''[documentation]'''
#         return hobbes.devices[port].iCmdResps[
#             "[compStr]_COMPONENT"][cmdIndex].respSend([args])
#     [cmdStr] = staticmethod([cmdStr])
```

Note that the function of the iCmdResp called is respSend(), but of course this depends on how iCmdResp is implemented and it could be send() as well. Also, creation of iCmdResps should only be attempted if the .icron file supports it (GE, version ¿= 2).

The iCmdResp wrapper code is added onto the iCommand wrapper code (making sure there are an appropriate amount of newlines), then the whole shebang is executed by Calvin. And now you have Python wrappers! How to use them is detailed in the User's Guide, as Python help doesn't really seem to belong in a design document.

4.1.1.0.4 exit()

Arguments: self

Returns: None

Saves the session and exits Hobbes. As with other functions, this function name should be something more specific like hobbes_exit() if there is no HobbesInterpreter class.

This function should store variable values of Devices and DeviceWindows, then write them to an external file.

The program should then close. If there is a function that exits the program immediately, it should be called. In Hobbes 0, this is "os._exit(0)". Alternatively, the function could set a boolean value that is checked periodically, and determines whether the program should continue or not.

4.1.1.1 StartupWindow

Functions in the StartupWindow class:

4.1.1.1.0 __init__()

Arguments: self

Returns: None

Initialize all the GUI components. There is nothing more to do.

This is a rather tedious function in Hobbes 0; it does what Designer code would do in a C# program. If GTK+ was used in an implementation of Hobbes, Glade could be used to make the GUI initialization much simpler.

4.1.1.1.1 icronFileButton_Click() *Arguments:* self; sender - the button sending the event; args - arguments from the event, which we don't need

Returns: None

Called when icronFileButton is clicked. It should prompt the user to select a .icron file to load. Hobbes 0 uses an OpenFileDialog; GTK+ provides a GtkFileChooserDialog.

4.1.1.1.2 DeviceRefreshButton_Click() *Arguments:* self; sender; args (note these are the same as above)

Returns: None

Called when DeviceRefreshButton is clicked, and refreshes the list of serial ports Hobbes can connect to. The easiest way to accomplish this would be to get the available ports and repopulate the serial port list.

4.1.1.1.3 addDeviceButton_Click() *Arguments:* self; sender; args

Returns: None

Called when addDeviceButton is clicked (see a pattern here?). Begins a new thread that executes code generated from the text properties of the Device and .icron file components. This code should call add_device() and load() in HobbesInterpreter.

4.1.1.1.4 StartupWindow_Closed() *Arguments:* self; sender; args

Returns: None

Called when the startup window is closed. Also executes code in a different thread; this time, the code is simply the calling of exit() in HobbesInterpreter. This allows the user to completely eschew the console if she wishes, as exiting does not necessarily have to be through the console.

4.1.1.1.5 StopDeviceOutputButton_Click() *Arguments:* self; sender; args

Returns: None

Called when StopDeviceOutputButton is clicked. Gets device argument from DeviceComboBox, this method begins a new thread that suspends output and dumps to a file the logCache of an existing device window.

4.1.1.1.6 execute_code() *Arguments:* self

Returns: None

Executes code by passing it to the interpreter. This should always execute in a different thread to prevent locking anything up. If an exception is raised, it is shown as a popup. Note there should be an appropriate facility to pass the code to execute into the thread as a parameter. In IronPython one can use ParameterizedThreadStart (.NET) or the threading module (CPython).

4.1.1.1.7 run() *Arguments:* self

Returns: None

Runs the StartupWindow. In Hobbes 0 this is just "Application.Run(self)" in a try-except block, where any exception opens an ErrorWindow class instance.

4.1.2 Variables and miscellaneous

Hobbes 0 adds the contrib/ folder to the path and adds the IronPython.dll file to the CLR at the module level.

At the module level, there are a couple variables:

4.1.2.0 hobbes_version A string containing the version of Hobbes, e.g. "0.4-rc15". This shows up in the console during startup and the startup window in Hobbes 0.

4.1.2.1 program_path A string containing the file path of the program.

This variable should be unnecessary. Usually something like os.getcwd() (in Python) would work. However, .NET has a ~~bug~~ feature where opening an OpenFileDialog or SaveFileDialog changes the current directory. Which is fine if we never need the actual path of the program...but we do for things like running an untarring program in the directory. Thus, we use some Python hackery to get the full path of the HobbesInterpreter.py file, which happens to be the program directory, and save it in a variable for convenience.

4.1.2.2 HobbesInterpreter Variables in the HobbesInterpreter class:

4.1.2.2.0 devices One of the more important variables in the program. Contains information on every Device that's currently connected to a serial port (which normally should be all of them). In Hobbes 0, this is implemented as a Python dictionary, where the keys are serial port names and the values are references to Devices connected to said port names. For example, if there is one Device currently in existence connected to serial port COM42, the devices variable in Hobbes 0 would look like "{'COM42': <Device.Device instance at 0xSOME_MEMORY_LOCATION>}". Every Device can be accessed through this variable.

4.1.2.2.1 Calvin The actual interpreter. It could interpret any suitable scripting language, but in Hobbes 0 it interprets Python, and scripting in Python seems to be the one of the easiest options, so I would recommend it's a Python interpreter. Of course, this should have a function for executing code passed in to it.

4.1.2.2.2 CalvinScope The scope of Calvin. Includes all the variables that are set in Calvin. This may or may not be necessary, depending on the implementation of Hobbes; in Hobbes 0, this is mandatory for executing code, as a function in Calvin is used to merely compile the code–it is actually executed in a function under CalvinScope.

4.1.2.2.3 startupWindow A simple variable containing a reference to a StartupWindow class instance.

4.1.2.3 StartupWindow There are *a lot* of variables here that are all related to the GUI. I won't be specific as to how the GUI should be implemented in terms of what variables there should be, and how it should look, as the objectives of the GUI outlined above should be sufficient. If you're actually very interested in what all the variables are, check out the __init__() function of the StartupWindow class in Hobbes 0.

4.2 ErrorWindow

If you're lucky, this module will never be used. But crashes happen, so the ErrorWindow class in this module will be used occasionally.

Note that the class actually isn't necessary. Instead of creating an ErrorWindow class instance when an exception is raised, one could simply "import ErrorWindow" and modify the code such that everything gets called at the module level. Hobbes 0 uses a class though.

4.2.0 Classes

4.2.0.0 ErrorWindow Contains a window that is shown when something bad happens. In Hobbes 0, this doesn't necessarily mean the entire program's crashed. If a DeviceWindow encounters an error, the thread running it will exit, while the main thread will soldier on.

Also writes the exception information/stack trace to an error log. This is ErrorLog.txt in the program directory in Hobbes 0.

4.2.1 Functions

4.2.1.0 ErrorWindow Functions in the ErrorWindow class:

4.2.1.0.0 __init__() Creates all the GUI components, then displays the window. This freezes the thread that the error occurred on.

The only real mandatory GUI component is a button to close the error window.

4.2.1.0.1 closeButton_Click() Called when the close button is clicked, and closes the window (if it wasn't already obvious enough).

4.2.2 Variables and miscellaneous

The variable program_path is not available here, so Hobbes 0 just repeats the Python stuff to create it here. Hobbes 0 does this to add an icon to the error window:

```
self.Icon = Icon(os.path.dirname(os.path.realpath(__file__)) +
            "\\hobbesicon.ico")
```

4.2.2.0 ErrorWindow Again, peruse Hobbes 0 code if you are interested in the detailed design of the GUI. If you do implement it a completely different way...at least keep the comic. Or put in another Calvin and Hobbes one. But I thought the one that's there seemed to fit the moment.

4.3 Device

All communication with the serial port should involve this module. Although sending of bytes to the serial port is handled by the iCommand module and parsing of bytes received from the port is handled by the iLog module, the iCommands and iLogs contained in a .icron file are stored in a Device class instance.

There should also be an additional class that contains a separate window, which at the minimum allows iLog output to be printed to it. Technically, this is optional, but having all iLog output go to the console would quickly get very annoying.

4.3.0 Classes

4.3.0.0 Device Connects to a serial port. Contains iCommands and iLogs to send/parse bytes to/from the serial port. Also parses bytes when bootloading/flashing through the port by Stewie or XMODEM.

4.3.0.1 DeviceWindow Contains a textbox where iLog output is printed. Additionally provides another textbox for code input to Calvin, and optionally methods for opening other GUI components. There should also be options for controlling the display of the text in the output textbox in the window (e.g. clearing all the text).

4.3.1 Functions

In addition to the functions in the Device and DeviceWindow classes, there are some additional module-level functions. These are optional, but may be very useful depending on the type of user.

4.3.1.0 crc16() *Arguments:* data - an 8-bit integer to calculate the CRC-16 on; last - the starting CRC-16 to use (i.e. the last one calculated)

Returns: The new CRC-16 calculated on the data input

Surprisingly enough, calculates the CRC-16 of an 8-bit input using a previous CRC-16. The calculation of the CRC-16 is done as follows:

1. Take the bitwise NOT of the last CRC-16, and if necessary, mask it with 0xFFFF to ensure the final result is 16 bits. Store this result in a variable; in Hobbes 0 this variable is called "next", and will henceforth be referred to as such.

2. Check to see if the value of the least significant bit in the 8-bit input matches the value of the least significant bit in "next". If so, shift the bits of "next" to the right by one. If they don't match, do the aforementioned right bitwise shift by one, but also XOR "next" by 0xA001.

3. Do the same check on the remaining seven bits, from least significant to most significant. That is, after the first check, check the second least significant bit, then the third last significant bit, and so on.

4. Take the bitwise NOT of "next", and mask with 0xFFFF if necessary. This is the final CRC-16. Return this value.

Note that if the data input is, say, 64 bits, the CRC-16 will likely be incorrect. If the data to calculate the CRC-16 on is greater than 8 bits, it should not be too difficult to create a loop where the CRC-16 is calculated on each 8-bit segment of the data.

In case the CRC-16 calculation procedure above was too confusing, here's Hobbes 0's implementation (which is probably even more confusing):

```
# the only peculiar thing is masking with 0xFFFF; this limits Python's ~
#     operator to 16 bits
next = ~last & 0xFFFF
for i in range(8):
    if ((data >> i) & 1 == next & 1):
        next >>= 1
    else:
        next >>= 1
        next ^= 0xA001

return ~next & 0xFFFF
```

4.3.1.1 availablePorts()

Arguments: None. (That is, there are no arguments. Not the Python keyword None.)

Returns: A collection of the available serial ports on the host system.

This function can be implemented in many different ways. It's not even used in Hobbes 0 as it were, since .NET provides SerialPort.GetPortNames(). However, in an alternate implementation of Hobbes, .NET won't be available, so there needs to be a way of finding all the serial ports that are connected.

One way of implementing this function is checking the OS, i.e. the value of sys.platform (assuming the Hobbes implementation is in Python) and enumerating a list of serial ports depending on the OS:

- In Windows, sys.platform would return "win32" or "cli", depending on the implementation (in IronPython it would be the latter), and the _winreg module (assuming Python 2 is used; it's winreg in Python 3) would be used to read from the Windows registry. In Hobbes 0:

```
ports = []
# we can only read from the Windows registry if (surprise) we're on Windows
if sys.platform == "win32" or sys.platform == "cli":
    # get access to the serial port registry entry
    key = winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE,
                         r"HARDWARE\DEVICEMAP\SERIALCOMM")
    # find the number of values in the entry, which corresponds to the
    #     number of serial ports connected to the system
    for i in range(winreg.QueryInfoKey(key)[1]):
```

```
            # append each port name to the ports list
            ports.append(str(winreg.EnumValue(key, i)[1]))

        # close the handler
        key.Close()
```

Note that "import _winreg as winreg" is performed at the module level in Hobbes 0 to avoid the annoying underscore.

- In Linux, sys.platform would return "linux2", and all serial ports would show up in /dev/ttyS*. Searching there would be sufficient.
- In OS X, sys.platform would return "darwin"...refer to what Hobbes 0 does. (Obfuscation is fun. I didn't want to make it *too* obvious what I was doing there.)

Lastly, return the collection of ports. Sorting the collection, if necessary, would be optional but perhaps more convenient to the user. In Python this would just be "return sorted(ports)".

4.3.1.2 Device

Functions in the Device class:

4.3.1.2.0 __init__()

Arguments: self; programPath - the path of Hobbes program (see program_path); port - the name of the serial port to connect to; settings - the path of the XMLSettings file to load for Device settings

Returns: None

This function should do, at minimum, the following:

- Initialize any variables that need initializing, including any lists, dictionaries, class instances, etc. Hobbes 0 also includes all the variables that are used in the class instance here regardless of whether they need to be initialized, simply to have a convenient list of all the variables used (since it's Python, variables don't need to be declared if they have something immediately assigned to them).
- Set the port name of the serial port assuming it is valid, i.e. if it is in the list of available ports. If it isn't, be helpful and print a list of the available serial ports.
- Load Device settings. This uses the settings argument, which defaults to a blank string. If this default is used, default Device settings should be loaded. The default settings should be in an external file; it's in Settings\DefaultDeviceSettings.xml in Hobbes 0.

4.3.1.2.1 connect()

Arguments: self

Returns: None

Opens the serial port of the Device class instance, or reopens it if it's already open. Load the settings for the serial port: the baud rate, parity, data bits, stop bits, handshake, write timeout, and read timeout.

Following this, create a DeviceWindow class instance and open it in a new thread. There should also be some notification that tells the user the Device has connected properly with the given settings.

The Device settings should be deprecated. It's not necessary, and was only included in Hobbes 0 as Tigger legacy. The only parameters that need to be specifically set are the serial port name, and possibly the baud rate if 115200 isn't the default.

4.3.1.2.2 DataReceivedHandler() *Arguments:* self; sender - the serial port calling this function; args

Returns: None

Called when bytes are received from the serial port of the Device class instance.

In .NET, this handler is called once when data is received, not every time a byte is received. For the most part, this seems to work fine. In a non-.NET implementation of Hobbes, there could be a separate thread that runs and parses bytes if it is determined that there are bytes to read from the serial port, and sleeps if there is nothing to read. This might look something like this, with PySerial and the Python threading and time modules:

```
import serial, threading, time

ser=serial.Serial([SERIAL PORT], 115200)

def read_thread():
    while True:
        try:
            if ser.inWaiting():
                [do stuff]
            else:
                time.sleep(0.01)
        except AttributeError:
        # raised when we try to read from a closed port
        # may occur during exit
            pass

thread=threading.Thread(target=read_thread)
thread.daemon = True # ensure the program can actually exit
thread.start()
```

This function should first read a byte from the serial port. One could read more, but everything parsing received bytes would have to iterate through the block of received bytes and it would generally be rather messy.

Next, something should be done with the byte, depending on what's happening:

- If we're currently processing an iLog, or the byte is greater than or equal to 240, parse the byte using the iLog module. The latter case means that the byte is a header for an iLog.

- Otherwise, if the byte matches a byte we're waiting for during flash, and it's not equal to 0, set an event that notifies another function that the appropriate byte has been received. Note the non-zero check; in Hobbes 0, the waiting flash byte is 0 when it's not being used. (It just occurred to me that it could be -1, which would be simpler, but you never know what could happen.)

- Otherwise, if the byte is 21, it's an ASCII NAK. In this case, further assess the situation. If we're waiting for a flash byte, we're flashing, and as a NAK was received, we need to resend the packet. This could involve setting the event that is mentioned in the second item above, while also setting a variable that is only True when a NAK is received to True. In Hobbes 0, there is a variable of the sort named "self.flashNAK". If we're not waiting for anything, then output the timestamp and NAK message to a DeviceWindow.

- Otherwise, if the byte received is part of a bootload message when flashing using Stewie, increment a counter that iterates across the length of set lists containing the bytes of each message. In other words, check if the entire bootload message has been received. If it has, print it.

- Finally, if none of the above conditions were satisfied, print a message saying an unknown byte was received.

Note that this entire segment of code should be wrapped in a try-except block. A SystemError (in Hobbes 0) could occur if Hobbes attempts to read from a closed serial port, which could happen if the program is closing and the port is automatically closed. If any other error is encountered, appropriate ErrorWindow handling should occur.

4.3.1.2.3 __parseNAK()

Arguments: self

Returns: None

Optional function. This can handle the case where the byte read from the serial port is a NAK, and an iLog isn't being processed. The rationale for this function would be making the DataReceivedHandler() function a little shorter and probably a bit easier to read.

Also, it of course doesn't have to be a Python "private" function; "parse_NAK()" would be fine too.

4.3.1.2.4 icron_untar()

Arguments: self; name - the name of the .icron file to extract data from

Returns: the data extracted, as a string

Untars a file from a .icron file. The .icron file is in fact a .tar.bz2 file, so in CPython the tarfile module could be used to read the .icron file. In Hobbes 0, however, bz2 is not supported in IronPython, so an external program must be used to extract files from the .icron file. This is exactly what Tigger does.

The function needs to extract the specific file from the .icron file, and write the standard output to a string. Once this is done, the standard error stream should be checked to ensure it's empty, and the standard output is not empty. If one or both of these conditions are not

met, something went wrong and an exception should be raised. In the case that everything is fine and dandy, return the string that standard output was written to.

This is actually quite tedious in Tigger and Hobbes 0; the .NET Process and ProcessStartInfo classes are used, and many inane variables must be set to extract a file correctly. As noted, CPython's tarfile module would be quite a bit easier to use and understand.

4.3.1.2.5 load_icron_file()

Arguments: self; path - the path of the .icron file to load
Returns: None

Untars a .icron file to read the icomponent, icmd, ilog, and possibly icmdresp files into corresponding data structures of the Device.

Firstly, there should be a check to ensure the path points to a file, and bsdtar.exe in the program directory exists. If one of these don't exist, an exception should be raised.

The path argument should be saved in an instance variable, as it will be used later.

Next, the "icron_header" file is extracted from the .icron file. The icron_untar() function is used here, if it was implemented. This file contains an assortment of keys and values, which must be stored somewhere. In Hobbes 0, they are stored in an XMLSettings class instance.

Now, the list of iComponents is retrieved by untarring the file listed under the key "icomponent" in icron_header. This file is formatted as follows:

```
components:
C:[NAME]_COMPONENT
C:[NAME]_COMPONENT
C:[NAME]_COMPONENT
[...]
```

For each line, the "C:" is removed and the remaining string is stored in a list.

Next, iCommands are retrieved by untarring the file listed under the key "icmd" in icron_header. This file is formatted as follows:

```
component:[NAME]_COMPONENT
F:[function] H:"[help string]" A:[argument(s); void if there are no arguments]
F:[function] H:"[help string]" A:[argument(s); void if there are no arguments]
[...]
component:[NAME]_COMPONENT
F:[function] H:"[help string]" A:[argument(s); void if there are no arguments]
F:[function] H:"[help string]" A:[argument(s); void if there are no arguments]
[...]
```

These iCommands are stored in a dictionary. Hobbes reads this file line by line. If the line begins with "component", Hobbes creates a new dictionary entry, with the component name as the key. The values are then added as iCommand class instances. Note that the parsing of the line is done in the iCommand __init__() function.

Next, iLogs are retrieved by untarring the file listed under the key "ilog" in icron_header. This file is formatted as follows, for LG1 .icron files that are version 4 or later and GE .icron files that are version 2 or later:

```
component:[NAME]_COMPONENT
L:[ILOG NAME] S:[iLog string]
L:[ILOG NAME] S:[iLog string]
[...]
component:[NAME]_COMPONENT
L:[ILOG NAME] S:[iLog string]
L:[ILOG NAME] S:[iLog string]
[...]
```

And for older .icron files:

```
component:[NAME]_COMPONENT
S:[iLog string]
S:[iLog string]
[...]
component:[NAME]_COMPONENT
S:[iLog string]
S:[iLog string]
[...]
```

Note that older .icron files do not contain an iLog name identifier. These were added in recent .icron files to identify iLogs more easily, which in turn makes iCmdResps easier to implement.

The iLogs are stored in a dictionary, similar to how iCommands are stored. Again, parsing is done in an __init__() function, and there should be an extra parameter specifying if "L:[ILOG NAME]" needs to be parsed or not.

Next, iCmdResps are retrieved by untarring the file listed under the key "icmdresp" in icron_header. First though, there should be a check to see if the .icron file is GE and version 2 or above, as icmdresp files only exist in those .icron files. Even if it is GE version 2+, there may not be an icmdresp file, so untarring should be done in a try-except block where any exception raised can be safely ignored. The file is formatted as follows:

```
C:[NAME]_COMPONENT
R:[response name] C:[iCommand name] L:[iLog name] A:[argument of iLog to return]
```

If there is an icmdresp file, and it is extracted to a string successfully, it is stored in an iCmdResps dictionary.

Untarring/extraction is now complete. There is one more very important thing to do, and that is setting up a handler to parse received bytes. In Hobbes 0 this is simple: a function is attached to the SerialPort's DataReceived handler.

```
# and *now* start parsing received bytes
# this avoids weird stuff happening when an .icron file hasn't been
#     loaded yet, but a bunch of bytes have been received and parsed
# set a function that will be called when data is received from the
#     serial port
self.serialPort.DataReceived += self.DataReceivedHandler
```

In CPython, .NET is not available, so the convenient (albeit possibly unreliable) event is not available. Thus, a thread running an infinite loop checking if bytes are in the serial port's buffer could be created and started; see DataReceivedHandler() for example code.

Optional GUI components: If there is a DeviceWindow, the title of it should be changed to include the file path of the new .icron file. If desired, the directory containing the .icron file could be watched for deletions, and a function set up to be called if deleting occur. This function is icronFileChanged() in Hobbes 0. And if there is a "Reload .icron File" button in the DeviceWindow, ensure its font colour is black.

4.3.1.2.6 icronFileChanged() *Arguments:* self; sender; args

Returns: None

A simple function that serves as a notification to the user that the .icron file may have to be reloaded (if she's paying attention). This could be anything from changing a colour somewhere to popping up an animation of a screaming face. As it is, Hobbes 0 does the relatively tame notification of changing the font colour of the "Reload .icron File" button from black to red.

4.3.1.2.7 printToDeviceWindow() *Arguments:* self; output - the text string to be printed in the DeviceWindow

Returns: None

Takes a string as input to be displayed in the DeviceWindow. First checks to see if ouput has been suspended, if not the messages will begin printing. Also checks to see whether or not auto scroll is enabled.

4.3.1.2.8 update_text() *Arguments:* self; newText - the text string to update the window title to.

Returns: None

Takes a string as input to be displayed as the title of the window. Called when programming a device and invoked onto GUI thread.

4.3.1.3 DeviceWindow Functions in the DeviceWindow class follow. Note most of these are handlers for button clicking events; as a result, if some buttons aren't implemented, there isn't any need for the functions connected to them.

4.3.1.3.0 __init__() *Arguments:* self; dev - the Device to create a DeviceWindow for

Returns: None

As with all GUI initialization functions, there is not a lot to specify here. The dev argument is used to save a reference to the Device for future use, and there are a couple of default values set: auto scrolling occurs and iLog suspension does not occur by default.

There is a little bit of code that isn't extremely tedious though; it creates a context menu (the pop-up menu when one right clicks in a textbox) and connects it to the input and output textboxes in Hobbes 0. However, this is only necessary for .NET because, well, it's .NET; GTK+ does context menus fine.

4.3.1.3.1 suspchk_CheckedChanged() *Arguments:* self; sender; args

Returns: None

Called when the "Suspend iLogs" checkbox is clicked. Modifies whether iLogs are suspended or not. Also, if the "Suspend iLogs" checkbox becomes unchecked, the log cache is printed.

4.3.1.3.2 autochk_CheckedChanged() *Arguments:* self; sender; args

Returns: None

Called when the "Auto Scroll" checkbox is clicked. Modifies whether the output textbox scrolls automatically or not.

4.3.1.3.3 SendButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Send" button is clicked. Executes, then clears, the code in the input textbox. If desired, the executed code can be printed to the output textbox with Python prompts (">>>", "..."). If auto scroll exists and is enabled, the output textbox should scroll. Finally, a new thread is started to execute the code.

4.3.1.3.4 ClearInputButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Clear" button (located under the input textbox) is clicked. Clears the input textbox of all text.

Side note: this is a good example of a GUI component doing a task that could alternatively be accomplished by entering something in the console; in this case something like "hobbes.devices['COM42'].InputTextBox.Clear()".

4.3.1.3.5 SaveButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Save..." button is clicked. Prompts the user for a location to save the text in the input textbox. In Hobbes 0, a SaveFileDialog is used. Once the path is selected, all of the text in the input textbox is written to the file at the path.

4.3.1.3.6 LoadButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Load..." button is clicked. Prompts the user for a file to load into the input textbox. Hobbes 0 uses an OpenFile Dialog. Once the path is selected, the text of the input textbox is entirely replaced with everything read from the file.

Note that Unix newlines (LFs) don't play nicely with Windows textboxes, which use Windows newlines (CR LFs). As a result, in Hobbes 0, there is an extra couple of commands performed on the text read from the file. First, any Windows newlines are converted to Unix newlines by replacing all occurrences of "\r\n" with "\n". Then, the reverse is done; that is, all Unix newlines are then converted to Windows newlines. Note that Unix newlines in the file are thus converted to Windows newlines, while existing Windows newlines are kept as such. And of course, no one cares about silly Macs and their CR newlines.

4.3.1.3.7 iCommandsButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "iCommands" button is clicked. Shows an iCommandWindow.

4.3.1.3.8 ReconnectButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Reconnect" button is clicked. First, this function makes sure that the serial port of the Device is in the list of available ports; if not, it loops, so it is entirely possible that a thread freezes here as it is. If there is a match (and there should be), the Device reconnects to the serial port. This has actually been useful on occasion when bytes are inexplicably not being read from the serial port.

4.3.1.3.9 ClearOutputButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Clear Output" button is clicked. Clears the output textbox.

4.3.1.3.10 iLogsButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "iLogs" button is clicked. Shows an iLogWindow.

4.3.1.3.11 ReloadIcronButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Reload .icron File" button is clicked. Reloads the .icron file of the Device by repopulating the iComponents, iCommands, etc. data collections. In Hobbes 0, this function creates code to use load() on the appropriate Device, and executes the code in a new thread.

The user should call this function whenever a change is made in firmware. If there is any change to the iComponents, iCommands, etc., there will be an obvious incompatibility between the .icron file loaded in the Device and the .icron file containing the firmware that had been flashed to the board. This is why Hobbes 0 watches the directory containing the .icron file for deletions, and changes the colour of "Reload .icron File" if there is a deletion.

4.3.1.3.12 BootloadButton_Click() *Arguments:* self; sender; args

Returns: None

Called when the "Bootload" button is clicked. Bootloads the board in a new thread.

The Stewie protocol is used if the "project" key in the .icron file starts with "lg1", signifying that the .icron file is for LG1 firmware or an LG1 test harness. Otherwise, the .icron file is for GE (note Hobbes won't support anything other than LG1 or GE as it is) and the XMODEM protocol is used.

4.3.1.3.13 FlashButton_Click() *Arguments:* self; sender; args

Returns: None

Called with the "Flash" button is clicked. Flashes the board in a new thread.

This function is somewhat similar to BootloadButton_Click(). However, an iCommand is sent first. This iCommand is the first one in "TOPLEVEL_COMPONENT", which should be "xmodem_new_image". If not, flashing should not go ahead. If it is, it should be sent, which results in the board beginning to send NAKs. A new thread is then started to flash the board using the XMODEM protocol.

4.3.1.3.14 XMODEMSend() *Arguments:* self

Returns: None

Whatever happens here, the function sendFileByXMODEM() is called at least once. Everything sent to the board in the function uses XMODEM.

- If the .icron file is for a test harness, the "test_image" file in the .icron file is sent.
- If the .icron file is for GE, the "flash_writer" file is sent, followed by "main_firmware".
- If the .icron file is for LG1, the "flash_writer2" file is sent, followed by "main_firmware".

4.3.1.3.15 StewieSend() *Arguments:* self

Returns: None

Whatever happens here, the function sendFileByStewie() is called at least once.

If the .icron file is for a test harness, the "test_image" file is sent by Stewie.

Otherwise, if the .icron file is for LG1, the "flash_writer" file is sent by Stewie, followed by "main_firmware" sent by XMODEM.

4.3.1.3.16 DeviceWindow_Closed() *Arguments:* self; sender; args

Returns: None

Called when the DeviceWindow is closed. Executes code that calls remove_device() to remove the Device containing the DeviceWindow being closed (note in Hobbes 0, removeDevice() is called instead). The execution of the code should be in a new thread.

4.3.1.3.17 execute_code() *Arguments:* self; code - the code to execute

Returns: None

Very similar to the execute_code() function in the StartupWindow class, but here the interpreter and interpreterScope variables are used for code execution.

In addition, the input textbox is cleared regardless of whether the code executes without any problems or not. If this function was called from SendButton_Click(), the desired effect is achieved. If this function was called from DeviceWindow_Closed(), nothing happens. But nothing bad happens either.

4.3.1.3.18 run() *Arguments:* self

Returns: None

Runs the DeviceWindow. In Hobbes 0 this is just "Application.Run(self)" in a try-except block, where any exception opens an ErrorWindow class instance.

4.3.1.3.19 aliasButton_Clicked() *Arguments:* self; sender; args
Returns: None
Called when one of the nine alias buttons is clicked in either the iCommand window or Device window. Executes the code saved in that alias button in a new thread.

4.3.1.3.20 dev_dumpLogCache() *Arguments:* self
Returns: None
Calls the dumpLogCache method in the iLog module with the device as the argument.

4.3.2 Variables and miscellaneous

If you're still here after reading that last section...

Hobbes 0 needs Microsoft.Scripting.dll in the project directory to get functions and classes to use when executing code.

There are two module-level variables:

4.3.2.0 btldr_sequence A collection of the bytes 66, 116, 76, 100, 114, 58, 32, 118, 49, 46, 51, 13, and 10. This is the ASCII equivalent of the string "BtLdr: v1.3\r\n". When an LG1 board is started up into bootloading mode, this string is transferred through the serial port. Instead of outputting "Unknown bytes received" messages in Hobbes, these bytes should be checked to see they are received in this order. Then Hobbes can output something like "Device COM42 is in bootloading mode".

4.3.2.1 btldr_OK_sequence A collection of the bytes 13, 10, 79, 75, 13, and 10. This is the ASCII equivalent of the string "\r\nOK\r\n". This occurs after bootloading of an LG1 board using the Stewie protocol is successful. Again, this is here simply so Hobbes is able to output a nice "Bootload OK" message instead of dumping unknown bytes.

4.3.2.2 Device Note that everything in this section uses Hobbes 0 variable names. Of course, whether variables use camelCase or an_underscore_delimiter is up to you.

Variables in the Device class:

4.3.2.2.0 icronFile An XMLSettings class instance. Stores everything that was in the .icron file for this Device.

I've probably said it many times, but this should be sufficient as a Python dictionary.

4.3.2.2.1 iComponents An ordered list of iComponents.

4.3.2.2.2 iCommands An unordered dictionary of iCommands. The keys are iComponents, and the values are ordered lists of iCommands for each iComponent.

4.3.2.2.3 iLogs An unordered dictionary of iLogs. The keys are iComponents, and the values are ordered lists of iLogs for each iComponent.

4.3.2.2.4 iCmdResps An unordered dictionary of iCmdResps. The keys are iComponents, and the values are ordered lists of iCmdResps for each iComponent.

4.3.2.2.5 Discourse on the preceding variables Why this structure of lists and dictionaries and lists within dictionaries? Python dictionaries are unordered, so the order of items in the three dictionaries of iCommand, iLogs, and iCmdResps must be irrelevant. And I found it simpler to navigate separate dictionaries. However, alternate implementations of .icron file data storage may of course be considered. Newer versions of CPython (2.7, 3.1) now contain an OrderedDict type that could certainly be used.

Also, the three dictionaries iCommands, iLogs, and iCmdResps could be modified so the values are instead dictionaries, where the keys are iCommand/iLog/iCmdResp unique names, and the values are actual iCommand/iLog/iCmdResp objects.

Whichever way the data storage is implemented, one must always remember that order must be saved: either through an ordered list or dictionary, or values that are stored in each iCommand and iLog. This is because sent iCommands must contain component and command indices, and received iLogs always contain component and log indices.

4.3.2.2.6 interpreter The Python interpreter. This points to Calvin in Hobbes 0.

This is left open as a possibility for multiple interpreters to be used.

4.3.2.2.7 interpreterScope The scope of the Python interpreter. This points to Calvin in Hobbes 0.

Again, this allows multiple interpreters to be used if desired.

4.3.2.2.8 More discourse, this time on multiple interpreters Hobbes 0 does not use multiple Python interpreters. But future implementations could.

- *Advantages:* Each interpreter can only be got at from one source. This means that if there are two DeviceWindows open, "print some_variable" in one DeviceWindow can produce a different result than the other one. And speaking of "print", each DeviceWindow's interpreter could redirect sys.stdout (and probably sys.stderr) to their respective output textboxes, so each DeviceWindow could be used without any interaction with the console.

- *Disadvantages:* Probably a noticeable speed decrease. Sharing variables between interpreters would quickly become annoying. And perhaps most importantly, one interpreter is the simplest. In terms of implementing Hobbes, and in terms of "did I declare this variable in *this* interpreter or *that* one?"

Really, it is up to the person implementing Hobbes.

Also, Hobbes 0 compensates for the lack of multiple interpreters by providing a couple of handy items for the script-writing user:

- **currentDevice** - A variable set by a DeviceWindow in DeviceWindow.execute_code(), just before code is executed. This allows each DeviceWindow to have a unique identifier, and "print currentDevice" would output different results depending on which

DeviceWindow the command was executed in. Note that running code using currentDevice concurrently in different threads may have unexpected results.

- **get_iCommand()** - A function defined in startup code executed in HobbesInterpreter.__init__(). I'll just be lazy and copy the definition here:

```
def get_iCommand(dev_name, comp, func):
    """Returns an iCommand corresponding to the appropriate Device,
    iComponent, and function name, assuming the iCommand exists.

    Arguments: dev_name - the Device name
               comp - the iComponent; omit "_COMPONENT" (e.g. use "ICMD")
               func - the function name"""

    return getattr(globals()[dev_name + "_" + comp], func)
```

  The function is simply a wrapper for Python's awesome getattr() function. Note the currentDevice variable could be used as the dev_name argument, which would allow something like "get_iCommand(currentDevice, 'ICMD', 'readMemory')(0)" to be used. If that string is executed in a COM37 DeviceWindow, it would send an iCommand to read 0x00000000 to port COM37. If it's executed in a COM42 DeviceWindow, it would send the same iCommand to COM42, and so on.

4.3.2.2.9 flashWaitingByte Used when Hobbes is flashing. Set to some non-zero integer value when we are waiting for a byte to be received (e.g. an ACK); set to 0 otherwise.

4.3.2.2.10 flashSync An asynchronous event that begins when Hobbes sends something to the board during flashing, and ends when flashWaitingByte is received. If this event does not end in time, a timeout should be raised; if a NAK was received instead, the thing that Hobbes sent to the board must be sent again.

In Hobbes 0, this is an AutoResetEvent, with an unset initial state.

4.3.2.2.11 flashNAK True if Hobbes sent something to the board and got a NAK in response; False otherwise. If this variable is True, Hobbes must send the same thing it had just sent. Note this happens a lot during flashing of LG1 firmware; there is an average of about two NAKs for each packet sent.

4.3.2.2.12 serialPort The serial port that this Device connects to.

4.3.2.2.13 devWindow The DeviceWindow of this Device.

4.3.2.2.14 deviceSettings An XMLSettings class instance that contains settings for the serial port.

This is unnecessary and should be removed in future implementations of Hobbes.

4.3.2.2.15 validPortName Its default value is False. Only set to True if the serial port name of the Device is valid.

Depending on how the addition of a Device is implemented, this may be unnecessary.

4.3.2.2.16 path The file path of the Device's .icron file.

4.3.2.2.17 programPath The path of the Hobbes program itself.

4.3.2.2.18 iLogFinished Default value is True. This is False if an iLog is currently being processed.

4.3.2.2.19 numArgs The number of arguments in an iLog. This is set when the header byte is parsed.

4.3.2.2.20 headerBits A string containing the binary representation of the header byte. This contains information on the endianness of the bytes received, whether the previous iLog was printed or not, and the number of arguments in the iLog.

4.3.2.2.21 messageBytes The bytes comprising an iLog being received.

4.3.2.2.22 messageLength The length of the iLog being received. This is set when the header byte is parsed. When messageBytes reaches this length, the iLog being received has finished being transmitted.

4.3.2.2.23 logCache In Hobbes, iLogs are not directly printed to a DeviceWindow's output textbox as soon as they are parsed. Instead, they are pushed to a cache (this variable), and only printed if there are no bytes to read from the serial port. This variable is a Python list, and contains entries of one-item dictionaries; the key is a reference to the iLog object, allowing custom font values stored in the iLog object to be read, and the value is the string to print. If there is no iLog object (e.g. when printing a NAK message), the key is None.

4.3.2.2.24 customFonts True if custom fonts should be used when printing iLogs to an output textbox, and False otherwise. Usually custom fonts are pretty cool and desired for making messages stand out, but if a lot of messages are being received, applying custom fonts slows Hobbes down immensely, and Hobbes 0 may crash if too many iLogs are received at once. Thus, if the user is not utilizing custom fonts, she should disable it to make Hobbes go faster. This value is modified by checking/unchecking the "Custom Fonts" checkbox.

4.3.2.2.25 btldr_index The index of the btldr_sequence list Hobbes needs to match next. Begins at 0. If the value at btldr_sequence[btldr_index] matches a byte being read, and the byte would otherwise be dismissed as an unknown byte, this variable is incremented. Once this variable has reached the end of the btldr_sequence list, a message is printed stating that a Device is in bootloading mode.

This is unnecessary if you don't care about seeing a bunch of unknown bytes every time an LG1 board starts up into bootloading mode.

4.3.2.2.26 btld_OK_index See btldr_index above. The only difference is that this variable is applied to the btldr_OK_sequence list instead of btldr_sequence.

4.3.2.3 DeviceWindow Variables in the DeviceWindow class:

4.3.2.3.0 dev A reference to the Device this DeviceWindow is a part of.

4.3.2.3.1 scroll True if auto scrolling should occur; False otherwise.

4.3.2.3.2 suspend True if iLogs should be suspended; False otherwise.

4.4 XMLSettings

Used by other modules to save and load settings via XML.

This module should not be implemented. There is no use for this module in Python; settings can be stored in a dictionary, and the json module can be used to save and load the dictionary to/from a disk drive.

4.4.0 Classes

4.4.0.0 XMLSettings Stores settings using XML. Can also save settings to and load settings from disk.

4.4.1 Functions

4.4.1.0 XMLSettings Functions in the XMLSettings class:

4.4.1.0.0 __init__() *Arguments:* self
Returns: None
Creates an XML file. Hobbes 0 uses an XmlDocument to manipulate the XML string. Uses this string as its base:

```
<?xml version="1.0"?><Settings></Settings>
```

4.4.1.0.1 set() *Arguments:* self; name - name of tags to be created; value - content within the tags; category - optional category of the tags
Returns: None
Creates tags in the XML string with the specified value and optional category.
This function should do the following:

1. Determine if there are existing tags with the same name as the name of the tags to be created. If they do exist, they should be removed, unless the API being used can overwrite them without any problems.

2. Create the new tags with "name", and set their content with the value argument.

3. If the category argument was specified, create an attribute of the newly-created tags, and set its value to the category argument.

4. Add the new tags to the XML string.

4.4.1.0.2 get() *Arguments:* self; name - the name of the tags containing the value to retrieve and return

Returns: The content of the specified tags, or None if they don't exist

Gets (and returns) the content of the tags identified by "name". If the tags don't exist, an exception may be raised. Therefore, the retrieval of the content should be in a try-except block, where exceptions result in an error message being printed and nothing being returned.

4.4.1.0.3 save() *Arguments:* self; filePath - the path to save the XML string to

Returns: None

Simply write to the specified file path. This should be done within a try-except block though, as an exception will be raised if the file path cannot be navigated to (e.g. if there is a non-existent directory in the file path).

4.4.1.0.4 load() *Arguments:* self; filePath - the path to load an XML string from

Returns: None

Load an XML string from the specified file path. Again, a try-except block should be used. An exception would be raised if filePath doesn't lead to a file.

Hobbes 0 also saves the original XML string due to what most definitely is a feature of .NET's XML namespace. If the file path leads to something that's not an XML string, and loading it into an XmlDocument is attempted, an error is raised (which makes sense), and the original XML string is wiped out (which makes slightly less sense). Thus, if this scenario occurs, Hobbes loads the original XML string back into the XmlDocument.

4.4.1.0.5 output() *Arguments:* self

Returns: The XML string being stored

Returns the XML string. In Hobbes 0, this can also be accessed by [XMLSettings object].doc.OuterXml, but output() is easier to type than doc.OuterXml.

4.4.1.0.6 outputList() *Arguments:* self

Returns: The XML string parsed into dictionaries

Returns the XML string parsed into a list of dictionaries. Each dictionary contains information on each tag, specifically its name, value, and category.

4.4.1.0.7 outputCategories() *Arguments:* self

Returns: All the categories in the XML string

Returns a list of categories. Iterates through all tags within the "Settings" tags, and adds to the list if a "Category" attribute was found.

4.4.2 Variables and miscellaneous

The XML functions used in this module are all in the System.Xml namespace of .NET.

4.4.2.0 XMLSettings Variables in the XMLSettings class:

4.4.2.0.0 BasicXml The base XML string used:

```
<?xml version="1.0"?><Settings></Settings>
```

This is used across all XMLSettings instances, so this could be a module-level variable.

4.4.2.0.1 doc A structure containing the XML string. In Hobbes 0 this is an XmlDocument, which provides a bunch of functions for easily manipulating the XML string.

4.5 iCommand

Sends iCommands to a serial port via instances of the iCommand class.

In addition, there could be a GUI window implemented which also sends iCommands but features much more mouse clicking, for users who want it.

4.5.0 Classes

4.5.0.0 iCommand Contains the function to send an iCommand to the board through the serial port. Objects of this class are stored in the "iCommands" dictionary of a Device.

4.5.0.1 iCommandWindow Contains a window that makes the non-scripting user of Hobbes much happier. This window includes alias buttons, so some features of Tigger are carried over here. Except here it's better, of course.

4.5.1 Functions

4.5.1.0 iCommand Functions in the iCommand class:

4.5.1.0.0 __init__() *Arguments:* self; line - a raw string to parse from the "icommand" file; compIndex - the index of the iComponent this iCommand is under in the list of iComponents; commIndex - the index of the iCommand in the list of iCommands for the iComponent; port - a reference to the Device's serial port

Returns: None

Initializes a few values in the iCommand class instance.

The line argument will look like 'F:[function] H:"[help]" A:[args]'. These are parsed into separate variables: one for the function, one for the help string, and one as a list for the argument types.

In addition, the latter three arguments are stored into three instance variables.

4.5.1.0.1 __repr__() *Arguments:* self

Returns: The function string of the iCommand

Returns self.function.

This will only make sense in a Python implementation. This allows actual iCommand objects to be added to GUI components, and the user would see the function names of iCommands instead of "<iCommand.iCommand instance at 0xSOME_MEMORY_LOCATION>".

4.5.1.0.2 send() *Arguments:* self; *args - a tuple of arguments for the iCommand

Returns: None

Sends an iCommand to the board by writing to the serial port.

This function does the following:

1. Iterate through all arguments, i.e. entries in args. For each argument, determine if it is a valid argument for its type. For each type, there are specific limitations for acceptable arguments:

   - boolT - boolean type; either "TRUE" or "FALSE".
   - sint8 - signed 8-bit integer; between -128 and 127 inclusive.
   - uint8 - unsigned 8-bit integer; between 0 and 255 inclusive.
   - sint16 - signed 16-bit integer; between -32768 and 32767 inclusive.
   - uint16 - unsigned 16-bit integer; between 0 and 65535 inclusive.
   - sint32 - signed 32-bit integer; between -2147483648 and 2147483647 inclusive.
   - uint32 - unsigned 32-bit integer; between 0 and 4294967295 inclusive.
   - component_t - component type. This is the index of an iComponent in a list of iComponents; although it is probably unnecessary, the argument can be between 0 and 4294967295 inclusive, the same range of a uint32 type.

   Note that if the argument is not a string, and not an integer using 32 bits or less, it's invalid regardless of type. The specific condition for the argument type may also be invalid. In either case, an exception should be thrown stating that an invalid argument was received.

   In Tigger, the checking of conditions being satisfied was done by parsing the argument into a specific type (e.g. Byte, SByte, Int16, UInt16...), and throwing an exception if something bad happened. Python has just int and long, so Hobbes instead checks if the argument is between the two specified values:

```
if argType == "sint8":
    if not -128 <= args[i] <= 127:
        raise TypeError("Invalid signed 8-bit integer")
elif argType == "uint8":
    if not 0 <= args[i] <= 255:
        raise TypeError("Invalid unsigned 8-bit integer")
[...]
```

2. Convert the argument to a list of four bytes. This consists of splitting a 32-bit integer into four 8-bit integers. The four bytes must be in big endian order, as the board (assuming it runs a LEON processor) uses big endianness. How this conversion is done is up to the person implementing Hobbes; Hobbes 0 uses the struct module and a list comprehension to convert Unicode characters into integers. Add these bytes to a list of numbers for the argument bytes of the iCommand to be sent. Once all arguments are done, the actual list of bytes to send to the serial port can then be constructed.

3. Create a new list, and add to it the following:

   - The header byte. This is 0x98 plus the number of arguments, so this byte could range from 152 (no arguments) to 158 (six arguments).
   - The component number. This is the index of the iComponent with the iCommand to be sent in the list of iComponents.
   - The command number. This is the index of the iCommand in the list of iCommands under the iComponent.
   - The argument bytes. This is what all the conversion to four byte lists was for.

4. Once this list is populated, write it to the serial port, preferably in a try-except block so nothing explodes if an exception is raised.

   Note that Hobbes 0 first needs to convert the list to a System.Array[Byte] collection, as the SerialPort.Write() function in .NET does not support a Python list.

4.5.1.1 iCommandWindow

Functions in the iCommandWindow class:

4.5.1.1.0 __init__()

Arguments: self; dev - the Device to send iCommands to
Returns: None

Initialize everything, as usual. Save the Device reference, which is the dev argument. Select the list of iComponents to use as the iComponents list of the Device. And create all the GUI components.

There are six argument collections (consisting of a textbox and two labels) and nine alias buttons. A massive list of constructors would have been boring, so in Hobbes 0 a bit of dynamic programming and introspection is done. If you're curious, peruse the code there; as it is, the final effect is nothing different than what would have been accomplished by, say, C# Designer code.

4.5.1.1.1 sortedCheckBox_CheckedChanged() *Arguments:* self; sender; args

Returns: None

Called when the "Sort alphabetically" checkbox changes its state. Sorts the iComponents depending on what state the checkbox is now in. If it is checked, then a list of sorted iComponents is used in the list of iComponents. If it is not checked, the original list of iComponents is used. The latter is useful for determining component indices for component_t argument types in iCommands, although [Device].iComponents.index("[NAME]_COMPONENT") could be used instead.

4.5.1.1.2 iComponentsListBox_SelectedIndexChanged()

Arguments: self; sender; args

Returns: None

Called when an iComponent is selected in the listbox of iComponents. Populates the list of functions based on the iComponent selected. The list of functions is cleared, then each iCommand under the selected iComponent is added to the list. (This is why the __repr__() function was defined earlier.) If in fact there are no iCommands, then don't do anything and leave the list blank.

4.5.1.1.3 functionsListBox_SelectedIndexChanged()

Arguments: self; sender; args

Returns: None

Called when an iCommand is selected in the listbox of functions. First disables all textboxes for iCommand arguments, then re-enables some or all of them based on how many arguments are required for the iCommand. Also sets the text of the help and code textboxes: the former is set with the iCommand's help string, and the code textbox is set with the code that would be required to call the function using Python wrappers.

4.5.1.1.4 argTextBox_TextChanged() *Arguments:* self; sender; args

Returns: None

Called when text changes in one of the argument textboxes. Modifies the code in the code textbox to show the change in an argument.

4.5.1.1.5 saveButton_Clicked() *Arguments:* self; sender; args

Returns: None

Called when the "Save" button is clicked. Saves the code to one of the nine alias buttons in the iCommandWindow; which one specifically is set by the user. Also sets the text and tooltip of the button.

4.5.1.1.6 executeButton_Clicked() *Arguments:* self; sender; args

Returns: None

Called when the "Execute" button is clicked. Executes the code in the code textbox.

4.5.1.1.7 execute_code()

Arguments: self; sender; args
Returns: None
Executes code by sending it to the Python interpreter. With the exception of a finally block, this is exactly the same as DeviceWindow.execute_code().

4.5.1.1.8 saveAliasesButton_Clicked()

Arguments: self; sender; args
Returns: None
Called when the "Save aliases..." button is clicked. Saves information (button names and Python code) for each alias button to disk using JSON.
In Hobbes 0, an OrderedDict is used, as the order of the alias buttons must be preserved.

4.5.1.1.9 loadAliasesButton_Clicked()

Arguments: self; sender; args
Returns: None
Called when the "Load aliases..." button is clicked. Loads information from the specified file to the alias buttons.
In Hobbes 0, an OrderedDict is again used.

4.5.1.1.10 run()

Arguments: self
Returns: None
Runs the window. First though, this function should populate the listbox of iComponents. Also, the list of sorted iComponents should be created; in Hobbes 0, this is as simple as using the sorted() function.

4.5.2 Variables and miscellaneous

The file Microsoft.Scripting.dll is required to be added to the CLR in Hobbes 0.

4.5.2.0 iCommand

Variables in the iCommand class:

4.5.2.0.0 functionSplit

The string 'F:'. Used to parse "line" in __init__().

4.5.2.0.1 helpSplit

The string ' H:"'. Used to parse "line" in __init__().

4.5.2.0.2 argsSplit

The string '" A:'. Used to parse "line" in __init__().

4.5.2.0.3 function

A string containing the function name of the iCommand.

4.5.2.0.4 help

A string containing help (i.e. information) on the iCommand.

4.5.2.0.5 argTypes

A list containing the argument types of the iCommand.
This list is empty if the argument type section of "line" in __init__() is "void".

4.5.2.0.6 compIndex The index of the iComponent the iCommand is under in the list of iComponents. If this iCommand is under, say, TOPLEVEL_COMPONENT, compIndex would be 0.

4.5.2.0.7 commIndex The index of the iCommand in the list of iCommands for the iComponent. If this iCommand is the first one listed under a computer, commIndex is 0.

4.5.2.0.8 port A reference to the Device's serial port the iCommand will be sent to.

4.5.2.0.9 args A list of bytes; more specifically, the bytes comprising the arguments section of the iCommand to send.

4.5.2.0.10 commandBytes A list of bytes; more specifically, the full collection of bytes to send as the iCommand.

4.5.2.1 iCommandWindow Variables in the iCommandWindow class:
(Note none of the GUI component variables are listed here.)

4.5.2.1.0 dev A reference to the Device this should send iCommands to.

4.5.2.1.1 listToUse A list to use in the listbox of iComponents. This will be either the original list of iComponents, or a sorted list of them.

4.5.2.1.2 sortedComponents A sorted list of iComponents.

4.6 iLog

Displays iLogs received from the serial port by parsing the bytes that comprise them.
An optional component is a window that sets custom fonts for iLogs as they are displayed.
Another optional component is a logger, where all iLogs are written to this log.

4.6.0 Classes

4.6.0.0 iLog Contains an iLog message, its name, and custom font information.

4.6.0.1 iLogWindow Modifies the display of iLogs.

4.6.1 Functions

4.6.1.0 dummyFunction() *Arguments:* dummy1, dummy2 - dummy variables
Returns: None
Provides nothing more than an initial function for an iLog's event handler. If you call this function yourself, it doesn't do anything.

4.6.1.1 iLogParseByte()

Arguments: byte - the byte to parse; device - the Device whose serial port the byte was read from.

Returns: None

If the byte received is 240 or higher, and Hobbes is not processing an iLog:

- Ensure everyone knows an iLog is now being processed. In Hobbes 0, this is changing the value of device.iLogFinished to False.
- Set the messageBytes array to contain only the header byte.
- Convert the header byte to a bit array, and save it as a string. Or, do a bunch of bit shifting later when we need to access individual bits of the header byte.
- Save the number of arguments as the value of the two rightmost bits in the byte.
- Set the messageLength variable to the length of the iLog in bytes, which we now know. It's 4 (header byte, component index, command index, log level) plus 4 times the number of arguments (4 bytes for each argument).

And if an iLog is being processed:

- Add the byte to messageBytes. Now, if the length of messageBytes is equal to messageLength, the iLog is now finished. Thus, Hobbes should print it now.

4.6.1.2 output()

Arguments: device - the Device to print output to

Returns: None

Prints an iLog with parameters based on device.messageBytes.

The header byte is examined first.

If the fifth bit from the left is 1, big endianness is used for the parameters; if it is 0, little endianness is used.

If the previous iLog was sent from the board successfully, the sixth bit from the left is 1. Otherwise it is 0.

Now, the second, third, and fourth bytes in device.messageBytes correspond to the component index, log index, and log level, respectively.

The message format of the specific iLog can be retrieved now, with the component index and log index now known. This message format may include format strings like "%d" and "%x". The latter format (hexadecimal) can be standardized if desired; Hobbes 0 does this by replacing all occurrences of "%x" with "%#010x" in the message format.

The args list of the specified iLog can be modified now. For each argument, get the slice of four bytes constituting it, unpack them into one integer, and append it to args. The unpacking is dependant on endianness; thus, the fifth bit from the left mentioned before should be read here.

If event handling is present in iLog, any event handlers connected should be fired now. This is important in iCmdResp.

Here is also a good time to get the log level of the iLog as a string.

Finally, the message can be completely parsed. A timestamp can be included. If the previous log was not printed, there should be a notification in the message that it wasn't.

The component name should be included, although "_COMPONENT" can be removed. The message format is formatted with the arguments, thereby replacing all the "%d" and "%x" and "%#010x" instances with actual numbers.

Now, if an error occurred during parsing, the subsequent exception should be caught and an error message printed. The error message should contain the serial port where the error was encountered, and all the bytes comprising the iLog, separated into the sections of header byte, component index, message index, log level, and argument bytes. Well, it doesn't need to be, but it would be useful for the user.

If an error didn't occur, the function continues and appends a dictionary of one entry to device.logCache. The printing of this cache may occur now, but two conditions have to be satisfied: first, there must be a window to access in the first place (i.e. device.devWindow cannot have been destroyed), and there cannot be any bytes to read in the serial port. The second condition is present to reduce the strain on the system having to print and parse bytes at the same time.

Note: starting from the paragraph detailing retrieval of the message format, an exception could have been raised anywhere. Thus, starting from (and including) that message format retrieval, there should be a try-except block to the possible printing of device.logCache. If any exception is raised, the type of exception should be printed, along with the serial port, and information on all the bytes comprising the iLog, as before.

4.6.1.3 printLogCache()

Arguments: device - the Device with the output textbox to print to

Returns: None

Prints everything in device.logCache, assuming that device.devWindow.suspend is False. However, if it is True, then this function should return immediately, preserving whatever's in the cache while not printing anything in it.

For each one-entry dictionary in device.logCache, the function does this:

1. Check to see if the iLog is "None". This means that the message is not associated with any iLog, such as a message with "NAK". If the iLog is something other than None, first check to see if the iLog was disabled. If so, don't do anything more with the iLog and move on to the next one. If not, get the custom font and colour saved in the iLog and apply it to the textbox.

2. Get the message to print, which should be the value of the dictionary.

3. Move focus to the input textbox (this is a holdover from Tigger, so this might not actually have any effect).

4. Print the message at the end of the textbox.

5. Scroll the textbox to the bottom, if device.devWindow.scroll is True.

6. Finally, log the output to an external file, if supported.

Note that the one-entry dictionary is not mandatory; if you have a different way of matching custom fonts to iLog strings, go ahead and use that instead.

After this has been completed, the text in the textbox should be truncated if necessary, and if there is a way for the user to specify a maximum textbox size. Note in Hobbes 0, truncating the text results in the textbox scrolling all the way up to the top, so it needs to be scrolled back down to the bottom.

Finally, clear the logCache variable by setting it to an empty list.

4.6.1.4 dumpLogCache()

Arguments: device - the Device whose logCache will be dumped.

Returns: None

Dumps everything in the iLog cache to the log file. Useful when the device window is being completely spammed with iLog messages and is unresponsive.

dumpLogCache first suspends output of log messages to the device window and then dumps all of the messages in the logCache to the log file, temporarily clearing the logCache.

Output is left suspended since this function is mainly used when the device is being flooded with log messages, if output is enabled you would just have to keep dumping the cache.

4.6.1.5 iLog

Functions in the iLog class:

4.6.1.5.0 __init__()

Arguments: self; line - the line from "ilog" to parse information from; moreParsing - whether the name of the iLog needs to be parsed

Returns: None

Initialize everything. Parse line: if moreParsing is True, do a little more work as there is a "L:" in the string as well as a "S:".

4.6.1.5.1 __repr__()

Arguments: self

Returns: The message of the iLog

Returns the message of the iLog. The rationale behind this is the same as the similar function in iCommand.

4.6.1.6 iLogWindow

Functions in the iLogWindow class:

4.6.1.6.0 __init__()

Arguments: self; dev - a reference to the Device with the iLogs to customize

Returns: None

Saves the reference to the Device. Sets dev.iComponents as the default list of iComponents to use. And does all the GUI stuff necessary.

4.6.1.6.1 iComponentsListBox_SelectedIndexChanged()

Arguments: self; sender; args

Returns: None

Fills the iLog textbox based on the iComponent selected. Clears any items already inside, then adds all the iLogs from the appropriate iComponent to the list.

4.6.1.6.2 sortedCheckBox_CheckedChanged() *Arguments:* self; sender; args
Returns: None

Sorts, or unsorts, the list of iComponents. Basically the same thing as the iCommandWindow equivalent.

4.6.1.6.3 iLogsListBox_SelectedIndexChanged() *Arguments:* self; sender; args
Returns: None

Changes the states of GUI components using variables in the iLog.

The text of the preview textbox (in Hobbes 0) is set to the message format of the iLog, with its font and colour applied. And the enabled status of the iLog is set depending on the enabled flag of the iLog.

4.6.1.6.4 enabledCheckBox_CheckedChanged() *Arguments:* self; sender; args
Returns: None

Changes the enabled flag of the selected iLog(s). Note the possible plural; the flags of numerous selected iLogs should be able to be changed simultaneously.

4.6.1.6.5 fontButton_Click() *Arguments:* self; sender; args
Returns: None

Opens a font dialog, with default font and colour set to the currently selected iLog's font and colour. Once the user exits the dialog with an "OK" result, update the preview textbox's font and colour, then modify the font and colour of every iLog that was selected.

The modification of multiple iLogs' fonts and colours should be supported.

4.6.1.6.6 run() *Arguments:* self
Returns: None

Similar to iCommand.run(). Creates a sorted list of iComponents and populates the listbox of iComponents, then shows the window.

4.6.2 Variables and miscellaneous

In Hobbes 0, the Lib\directory in the program directory is added to the system path so IronPython modules can be imported.

Included here are a few module-level variables:

4.6.2.0 programPath See HobbesInterpreter.program_path. Created the same way.

4.6.2.1 outputHandler Used to set up logging for Hobbes. The type of handler is selected here; Hobbes 0 uses a regular file handler. This means that every iLog is written to one log file; for better precision, it is more suitable to do something like what Tigger does and have rotating file logs in multiple folders.

4.6.2.2 outputLogger Does the logging to the file. The log should contain the fully formatted iLog message, with perhaps a serial port name if everything is written to one log.

4.6.2.3 iLog

Variables in the iLog class:

4.6.2.3.0 enabled

Flag that determines whether the iLog should be printed to a DeviceWindow if it is received from the serial port.

4.6.2.3.1 font

The default font of iLogs. Could be anything, really; Hobbes 0 uses Consolas 10 pt.

4.6.2.3.2 fontColour

The default colour of iLogs. Again, could be anything. Hobbes 0 uses black.

4.6.2.3.3 args

A list of numerical arguments used when formatting iLogs. These are created by combining four-byte arguments from the serial port to form one 32-bit integer.

4.6.2.3.4 events

An event that fires whenever an iLog is completely parsed. Any functions that are connected to it are executed when this is fired. As a default to satisfy .NET, Hobbes 0 uses dummyFunction() as a default connected function.

4.6.2.3.5 name

The name of the iLog. Only set with recent .icron files, i.e. LG1 version 4 or later and GE version 2 or later.

4.6.2.3.6 message

The message format of the iLog. Probably a C-style format string, with possible instances of "%d" and "%x".

4.6.2.4 iLogWindow

Variables in the iLogWindow class, omitting GUI components:

4.6.2.4.0 dev

A reference to the Device this iLogWindow is a part of.

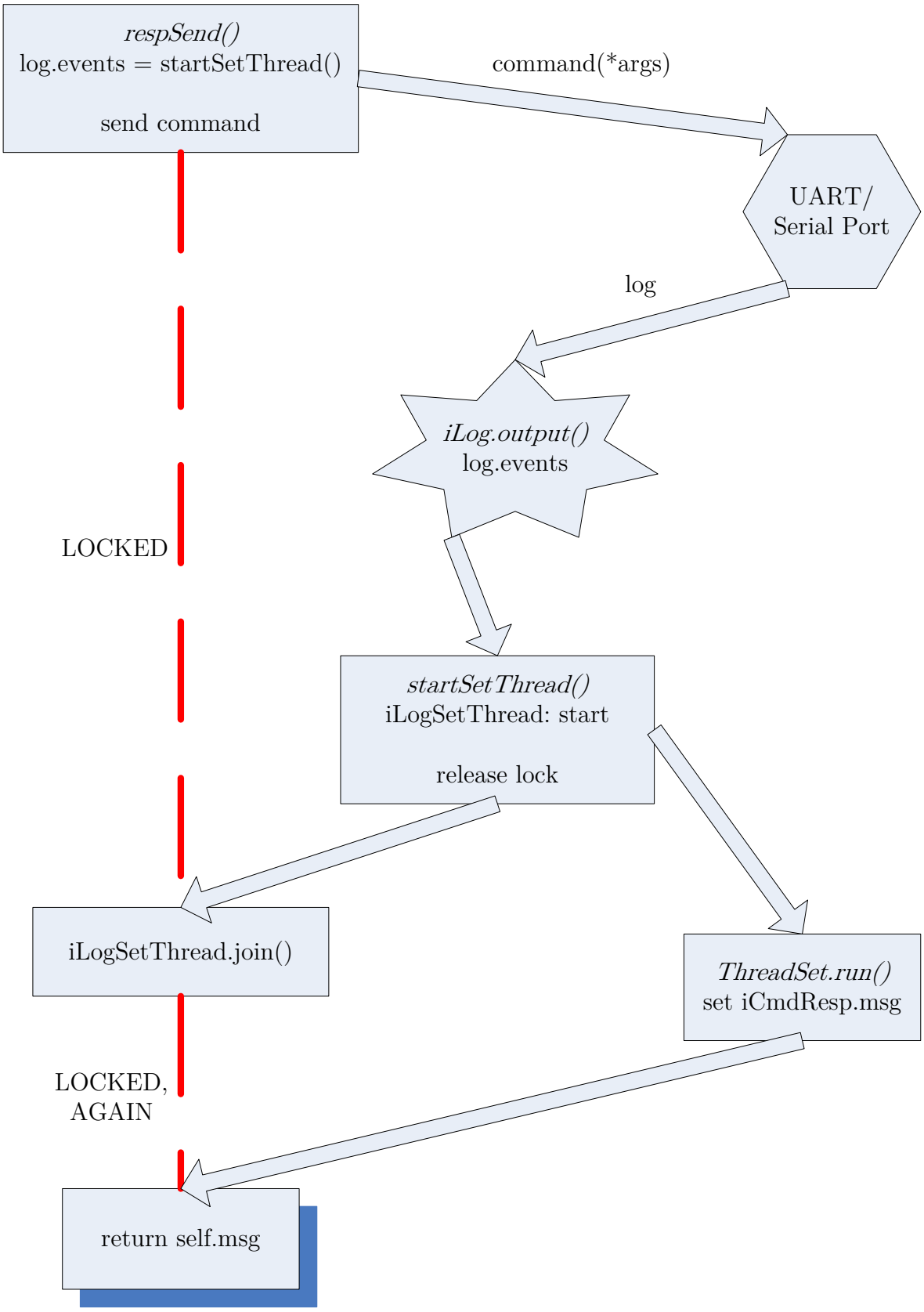
4.6.2.4.1 listToUse

Exactly the same as iCommandWindow.listToUse. Contains a list of iComponents to use in the iComponent listbox.

4.7 iCmdResp

A structure (rather, a collection of structures) that combine an iCommand and iLog into a function. When this function is called, an iCommand is sent, the function locks until a specific iLog is received, the value of a specific iLog argument is retrieved, and the function returns the value.

Following is a diagram of how the iCmdResp module works.


respSend()
log.events = startSetThread()
send command
command(*args)
UART/
Serial Port
log
iLog.output()
log.events
LOCKED
startSetThread()
iLogSetThread: start
release lock
iLogSetThread.join()
ThreadSet.run()
set iCmdResp.msg
LOCKED,
AGAIN
return self.msg

4.7.0 Classes

4.7.0.0 iCmdResp Contains the iCommand and iLog to work with, and a function to send the iCommand and wait for an appropriate iLog.

4.7.0.1 ThreadSet Extends the Python Thread class to set the message of an iCmdResp, which is eventually returned at the end of respSend().

4.7.1 Functions

4.7.1.0 iCmdResp Functions in the iCmdResp class:

4.7.1.0.0 __init__() *Arguments:* self; line - the line from an "icmdresp" file to parse; componentName - the name of the iComponent containing the iCommand and iLog to be set; device - the Device containing the iCmdResp object
Returns: None
This function does the following:

1. Parses line, which looks like "R:[response] C:[command] L:[log] A:[arg]", to get the response name, iCommand name, iLog name, and iLog argument index.
2. Find and save references to the iCommand and iLog of the iCmdResp.
3. Saves a reference to device, as it will be used later.
4. Creates a Python Lock instance and calls acquire() once. This means that the next thread that calls acquire() will be locked until release() is called in a different thread.

4.7.1.0.1 respSend() *Arguments:* self; *args - a tuple of iCommand arguments
Returns: None
Looking at the preceding diagram may help.

1. Saves the current events of the iLog. They will be restored at the end.
2. Set the new event handler of the iLog as startSetThread().
3. Send the iCommand with *args as the arguments.
4. Acquire the lock, locking this thread. respSend will attempt to acquire the lock for 3 seconds before giving up and returning None. This timeout should be sufficiently long enough to only occur if the device has been disconnected.
5. Join iLogSetThread, locking this thread again.
6. Restore the original event handler(s) of the iLog.
7. Return the self.msg variable.

4.7.1.0.2 startSetThread() *Arguments:* self; sender - the iLog causing this function to be called; args

Returns: None

Creates a thread that will set the iCmdResp's message.

Is this really necessary? Maybe. The whole module would be much easier to comprehend if a thread wasn't created here, but .NET and its event handling is iffy. If it ever turns out that code execution and iLog parsing occur on the same thread, you have an instant deadlock.

1. Saves the sender argument, i.e. the iLog.
2. Creates a ThreadSet instance. Note that in order to save the iCmdResp in this instance, self is passed in as an argument.
3. Starts the newly-created ThreadSet instance.
4. Release the lock so respSend() can then unlock (only to lock again, of course).

4.7.1.1 ThreadSet Functions in the ThreadSet class:

4.7.1.1.0 __init__() *Arguments:* self; iCmdResp - the iCmdResp calling this function

Returns: None

Saves the iCmdResp reference, then calls __init__() of the parent Thread class to initialize everything else (following proper Python conventions).

4.7.1.1.1 run() *Arguments:* self

Returns: None

Called when the start() function of the underlying Thread is called. Sets the iCmdResp's message by retrieving the appropriate argument from the arguments of the appropriate iLog.

4.7.2 Variables and miscellaneous

Some module-level variables are included here.

4.7.2.0 responseSplit The string "R:". Used to parse the line argument in __init__().

4.7.2.1 commandSplit The string " C:". Used to parse the line argument in __init__().

4.7.2.2 logSplit The string " L:". Used to parse the line argument in __init__().

4.7.2.3 argSplit The string " A:". Used to parse the line argument in __init__().

4.7.2.4 iCmdResp Variables in the iCmdResp class:

4.7.2.4.0 response The name of the iCmdResp. This is only used when creating Python wrappers for iCmdResps.

4.7.2.4.1 command The iCommand used in the iCmdResp.

4.7.2.4.2 log The iLog used in the iCmdResp.

4.7.2.4.3 argIndex The argument index of the iLog's argument to set as the iCmdResp's message.

4.7.2.4.4 device A reference to the Device this iCmdResp is a part of.

4.7.2.4.5 lock A Python Lock used for locking a thread.

4.7.2.4.6 sender The iLog firing the event once it is completely parsed.

4.7.2.4.7 iLogSetThread An instance of the ThreadSet class.

4.7.2.4.8 msg The message of the iCmdResp to return.

4.7.2.5 ThreadSet Variables in the ThreadSet class:

4.7.2.5.0 iCmdResp The iCmdResp whose message should be set.

4.8 Stewie

Sends an image file to a board using the Stewie file transfer protocol.

4.8.0 Classes

None.

4.8.1 Functions

4.8.1.0 sendFileByStewie() *Arguments:* dev - the Device to send the file; name - the name of the file as specified in icron_header

Returns: True if transfer was successful, False otherwise

First, uses "name" to extract the actual binary to send. Hobbes 0 uses bsdtar.exe.

Then, checks the Stewie file to make sure the file is formatted correctly (note all bytes are represented in decimal):

- The header must be four bytes: 83, 48, 48, and 51.
- The footer must be two bytes: 83 and 56.
- Each record in the binary should contain, in order, 83, the address length, the record length, the address, the actual data, and the checksum.

- The address length must be 49, 50, or 51. These bytes correspond to an address length of 2, 3, and 4. This length is included in the record length.
- The record length must be at least the same value as the address length (of course), and cannot result in bytes being read past the end of the file.
- The checksum is calculated by adding together the bytes from the record length byte to the end of the actual data (including the ends), taking the complement of the sum, then masking the result by 0xFF. This checksum must match the checksum byte located at the index directly after the data.

If any of these conditions are not satisfied, an error message should be printed and the function would return False.

If all of the conditions are satisfied, sending can begin:

- Send 63 ("?") to the board. If 64 ("@") is not received in 2 seconds, retry. If there have been 3 retries, quit the function.
- Send the header of 83, 48, 48, and 51. If 123 ("{") is not received in 2 seconds, quit.
- Send each record. If 126 ("~") is not received in 2 seconds after sending, quit.
- Send the footer of 83 and 56. If 125 ("}") is not received in 2 seconds, quit.

Once 125 is received, bootloading is finished and the function can return True.

4.8.2 Variables and miscellaneous

None. Well, there are variables, but they can easily change.

4.9 XMODEM

Sends an image file to a board using the XMODEM file transfer protocol.

4.9.0 Classes

None.

4.9.1 Functions

4.9.1.0 sendFileByXMODEM() *Arguments:* dev - the Device to send the file to; name - the name of the file as specified in the .icron file

Returns: True if the file was successfully sent, False otherwise

Wait for a NAK to be received. If it isn't in 5 seconds, quit. Note that 5 seconds is required, as this function is called directly after flashing a flash writer to the board. The board must first finish erasing the flash and verifying it before becoming ready for XMODEM transfer and sending a NAK.

Untars the binary to be sent, and calls send_XMODEM_raw_binary() with the result.

4.9.1.1 send_XMODEM_raw_binary()

Arguments: dev - the Device to send the binary to; binary_bytes - the bytes of the binary; file_name - the name of the file being sent

Returns: None

Sends the binary_bytes variable (as some sort of collection) to the board using the XMODEM protocol:

1. Send a packet of 132 bytes to the board. This is composed of all bytes, so values will wrap if they go below 0 or go above 255:
   - SOH (the byte 1)
   - The packet number, i.e. n where the packet is the nth one being sent
   - The inverse packet number: 255 - (packet number)
   - The data: 128 bytes of the binary, or the rest of the binary padded by 0xFFs to 128 bytes
   - The checksum: the sum of the data mod 256
2. After the packet is sent, wait for a response. If there is no response in 2 seconds, time out. If there is a response of an ACK (byte 6), send the next packet. If there is a response of a NAK (byte 21), try sending the packet again. Allow a maximum of 10 retries; if this maximum is reached, quit the function.
3. Finally, send an EOT (byte 4). Do the same procedure of waiting and possibly resending as if a packet had been sent.

Lastly, notify the user transfer was successful and return True.

4.9.2 Variables and miscellaneous

None. Again, depends on the implementation.

The splitting of flashing a board via XMODEM into two functions means one can call send_XMODEM_raw_binary() to send a binary that is not in a .icron file.

5 Test Plan

There shouldn't be a need for a detailed test plan. If it works, great. If it crashes, maybe there's a bug or .NET is just plain bad in the case of Hobbes 0.