A Look Inside the KIPR Link
Braden McDorman, Joshua Southerland
KISS Institute for Practical Robotics, University of Oklahoma
bmcdorman@kipr.org, southou@gmail.com

A Look Inside the KIPR Link

1 Introduction

The KIPR Link (herein referred to as just "The Link") is the fifth generation educational robotics controller used in the Botball robotics competition. The Link is also the first robotics controller designed from the ground up for Botball. Since the Link runs a full linux kernel and is completely open source, it is a perfect platform for modification and experimentation by more advanced users. It is often difficult and intimidating, however, to approach a complex software system such as the Link and begin tinkering. As such, this paper hopes to serve as a system overview and component guide for the curious and ambitious user that wishes to modify the Link for their own purposes.

This paper also features an accompanying website (`http://bmcdorman.github.io/link`) with guides, links, and instructions. Parts of this paper will assume that the reader is familiar with this website and its instructions.

1.1 Disclaimer

Modifying the Link's firmware can lead to system instabilities and possible bricking. While the Link has been designed to be somewhat resilient to system tampering, no guarantees can be made to any given modification's safety. That said, this paper's accompanying website provides documentation for un-bricking any Link manually. Manually un-bricking a Link will, however, require opening the Link's case and **will void your warranty**.

2 Hardware Overview

While this paper is primarily aimed at understanding and modifying the Link's software, a basic understanding of the Link's hardware is essential. A basic and non-exhaustive component and communication graph for the Link is presented in Figure 1.

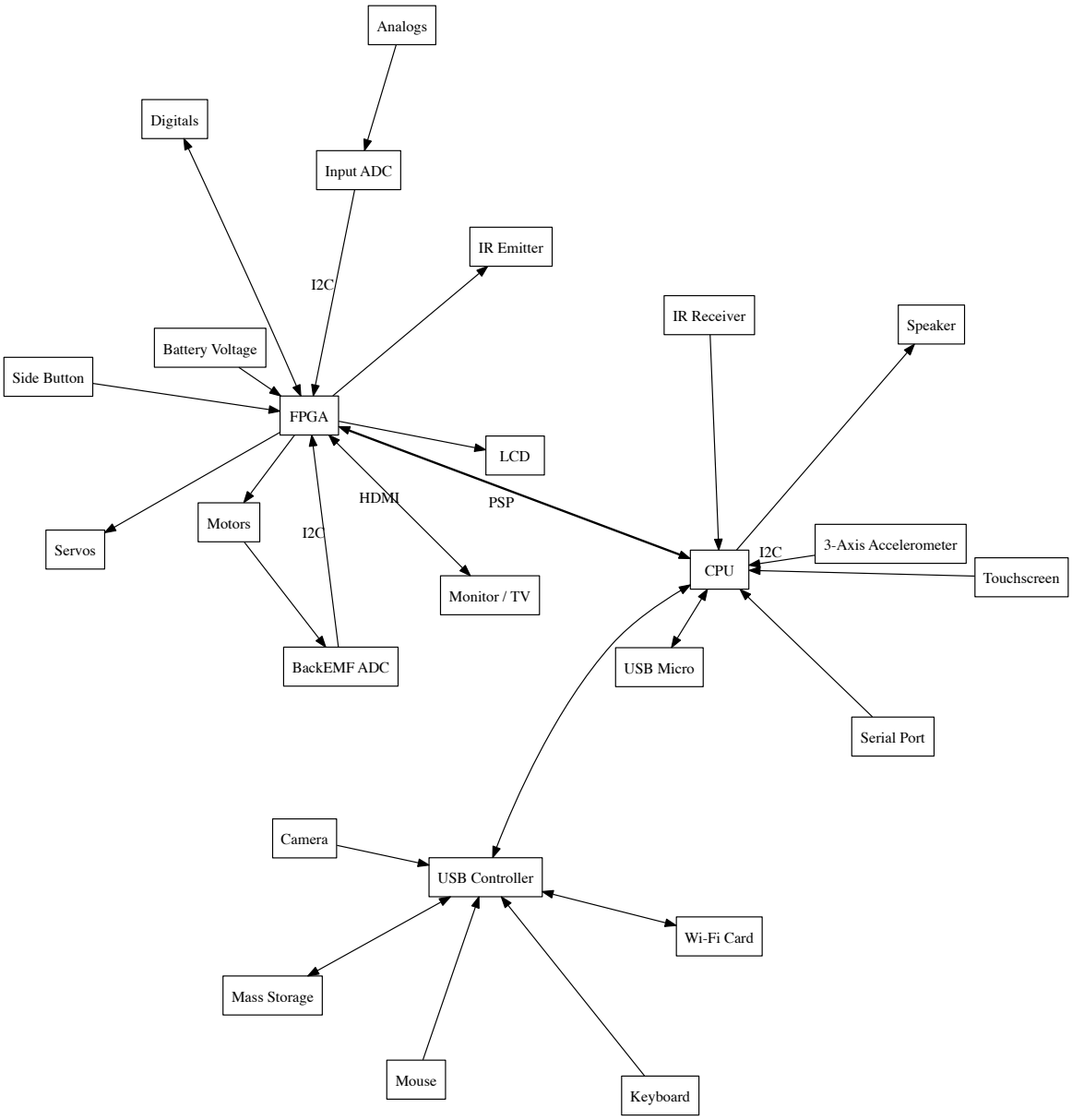

Figure 1: Hardware Overview of the KIPR Link

The Link features an ARMv5te CPU and a Spartan 6 FPGA. Both of these chips work in tandem to enable all of the functionality of the Link. The Link also features two USB host ports for accessories and external mass storage devices. A TTL serial port was added for connections to robot platforms like the iRobot Create.

2.1 The Spartan 6 FPGA

A FPGA, or Field-Programmable Gate Array, is an integrated circuit that can be programmed to perform highly parallel and fast custom logic. FPGAs can be orders of magnitude faster than CPUs for certain types of operations, but are not designed to handle the sequential logic that a CPU would normally perform.

The Spartan 6 FPGA communicates with the CPU via a PSP (programmable serial protocol) bus. PSP is a more general form of SPI, and the parameters chosen differ from SPI in that the clock signal (26MHz) is continuous. This was a requirement because the PSP clock is also used by the FPGA to drive its many clocks (up to 208MHz). The FPGA stores 46 different registers allowing the user to retrieve information such as analogue sensor voltages, or control peripherals such as the motors.

The FPGA uses many modules[1] to perform its duties. Two modules provide I2C and PSP communication capability. PWM modules exist for modulating servo and motor control signals. A "quad_motor" module is used in order to control the states of the motors (forward, reverse, brake, idle). Modules also exist which automatically update the ADC and BEMF sensed values. Finally, there is a module which provides HDMI output. These modules are mostly running in parallel, since a FPGA is capable of executing many operations at the same time.

Unfortunately the FPGA is more of a hindrance than a benefit in the current version of the Link. The FPGA was originally added to mediate fast vision processing like that of the XBC's. Using a USB camera, however, means that the cost of transporting the image data to the FPGA is more expensive than just doing the computation on the CPU. To realize the performance the XBC enjoyed, the camera would need a direct (and preferably non-USB) connection to the FPGA.

3 The Software Stack

The Link's full software stack is composed of hundreds of libraries, executables, and configuration files. Fortunately, many of these are not entirely relevant to the discussion of the Link's software components. In this section I will enumerate the important libraries and executables to understand, provide notes on their implementation, provide examples of their usage. This paper's website also goes over "modification checklists" that can be used as guides for certain types of system modifications. Figure 2 shows connections between various components of the Link.

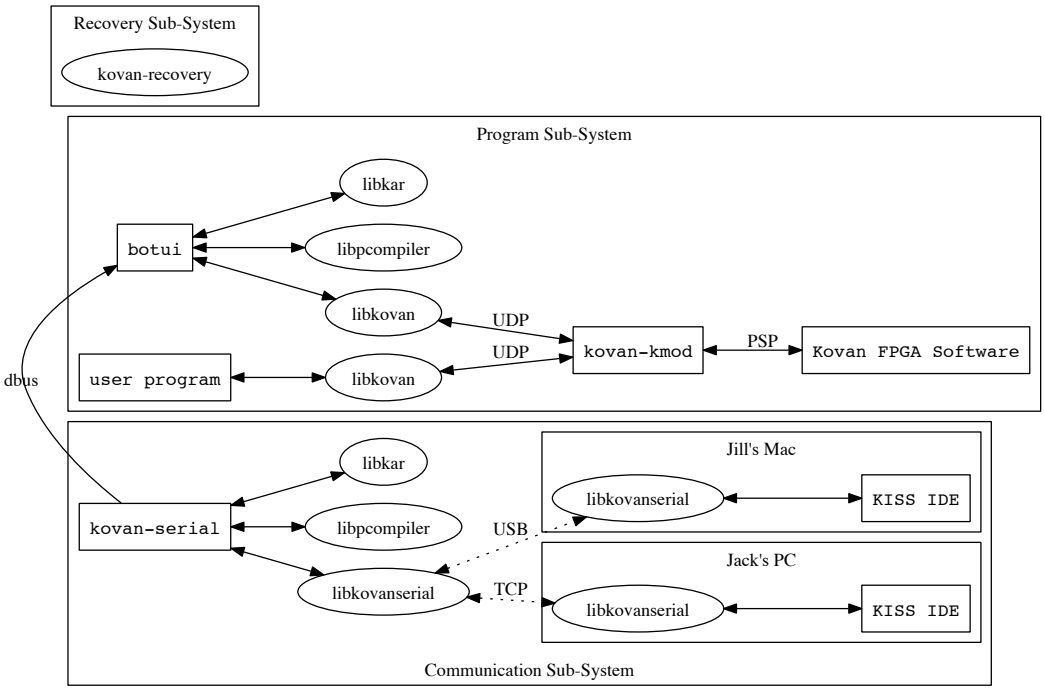

Figure 2: Software Overview of the KIPR Link

The components listed below are collectively referred to as the *KISS Platform*. Almost every library in the KISS Platform is used seamlessly across the Link, KISS IDE, and the computer targets for KISS IDE. As such, keep in mind that the Link is a piece of a much larger software architecture spanning several platforms, protocols, and running machines.

3.1 `libkovanserial` [2]

`libkovanserial` is the unified network and USB communication protocol for the Link and KISS IDE. `libkovanserial` is divided into three layers:

1. "Transmitters" are back-ends that implement a specific communication mechanism, such as TCP/IP sockets or USB comm ports. Security is not handled on this layer.
2. The "Transport Layer" handles packet creation, checksumming, and a basic ACK/resend mechanism for non-reliable protocols such as serial communication ports. Session-level security is defined on this layer.
3. The "Protocol Layer" helps facilitate protocol-level communication with the KIPR Link. This is intentionally left as a somewhat leaky abstraction. User password security is implemented on this layer.

3.1.1 Authentication Handshake

`libkovanserial` uses XOR encryption with a mutual shared session key that is negotiated during handshake. XOR encryption is notoriously insecure with plain text, but the use of random data paired with cryptographically secure data makes normal XOR encryption attack vectors useless. Since the session key is a pseudo-randomly generated 512 bits and sessions are short lived, cracking the session key with brute force is unlikely. `libkovanserial` also goes a step further and fills empty space in packets with pseudo-random bytes. This prevents sniffers from detecting the key using zeroed-out sections of packets. This handshake method guards against man-in-the-middle attacks and other sniffers by XOR encrypting the session key during transmission using a private but mutual piece of information: `sha1(password)`. Since the session key is 512 pseudo-random bits and the SHA1 key is a cryptographically strong hash, decoding either piece of information is unlikely. A new session key is generated with every high-level command to the Link, so a session key does not remain valid for more than a few seconds.

3.1.2 Handshake Example

A typical handshake looks like this:

1. Ask the server if it requires authentication. If no, `finish`. If yes, `goto 2`.
2. Send the server our password's MD5 hash.
3. Check if authentication was successful. If no, prompt user for new password and `goto 2`. If yes, decrypt the session key using our password's SHA1 hash and `finish`.

3.2 The `kovan-serial` Daemon [3]

The `kovan-serial` daemon mediates all incoming connections to the Link over USB and Wi-Fi. `kovan-serial` is simply a multi-threaded front-end for the `libkovanserial`, `pcompiler`, and `libkar` libraries. Since `kovan-serial` is best understood in terms of the libraries it uses, I will not delve any deeper into its description.

3.2.1 Notes on Using the USB Micro Port

The kernel driver used for the USB Micro port on the Link (`otg_serial`) is unfortunately very buggy with the Link's hardware. If the USB cable is ever physically disconnected and then subsequently reconnected to the Link, the kernel driver enters an error state that *can not be directly detected*. Attempting to read or write data while in this error state will eventually lead to the kernel driver locking up, which can not be fixed without power cycling the device. After much experimentation, it was discovered that this error state can be indirectly detected by attempting to `write` an array of size zero to the open USB file descriptor at semi-frequent intervals (`kovan-serial` checks every two seconds). If `write` fails and sets `errno` to `EIO`, close the file descriptor and re-open it. It should be noted that `read` *will not return any error* other than setting `errno` to `EAGAIN` (An error that means there was no data ready to read, but that the connection is still good), even though attempting to read once the error state is entered could eventually result in the kernel driver locking up.

3.3 `libkovan` [4]

`libkovan` is the most important library on the Link. It is designed to implement and/or expose every piece of functionality a user would expect from a robotics controller. This includes, but is not limited to:

- Motor actuation
- Servo actuation
- Camera perception
- AR.Drone communication
- iRobot Create communication
- High-level threading routines
- Simple key/value configuration files
- Data collection and export
- Utility functions

`libkovan` is written in C++ and exposes a C++ API and a C API. Since the C API is intended for simple use cases, it does not expose all of the functionality of the library. For example, it is possible to open two cameras simultaneously using the C++ API. Exposing similar behavior in the C API would greatly complicate function signatures and documentation. As such, it is recommended advanced users wishing to get the most from their device use the C++ API. Under the hood, `libkovan` works by queueing commands to be written out to `kovan-kmod`. Once commands are written, `libkovan` waits for a state response from `kovan-kmod` containing updated sensor and system information. By default, commands are flushed as a result of every pertinent `libkovan` function call. This behavior can be modified and controlled, however, using the `set_auto_publish(int)` and `publish()` library functions.

`libkovan` is also written to be bindable to other languages, such as Java, Lisp, and Python. As such, the C API attempts to avoid complex data types, memory allocation routines, and pointers (which may not be available in the bound language).

Some of the functionality of `libkovan` can be easily used on certain non-embedded computers. For example, it is possible to use `libkovan` for AR.Drone control, data logging, threading, and camera perception with only a POSIX-compliant operating system.

3.4 `libkar` [5]

`libkar` is an extremely simple Qt-based archive format used to store and transport data in KISS IDE and its targets (including, but not limited to, the Link.) `libkar` is implemented as a flat key/value dictionary with methods to treat the keys as a two-dimensional tree structure when useful. This dictionary is then serialized to a byte stream for transfer over other mediums.

Serialization is a fancy term for taking a data structure like a color, time, or image and converting it into an array of bytes that can be either stored on a disk or transferred to another computer via a serial connection. Serialization is reversed using *deserialization*.

3.5 pcompiler [6]

pcompiler (short for precedence compiler) is a library that automatically attempts to produce executables from a set of input files. pcompiler uses file suffix precedence to create a weak build ordering, and then applies *transforms* to mutate the input into an output. For example, given the files: main.c, functions.h, and functions.c, pcompiler will generate the following transforms (herein denoted by a $\rightarrow$):

1. {functions.h} $\rightarrow \emptyset$ (passthrough transform)
2. {functions.c, main.c} $\rightarrow$ {functions.c.o, main.c.o} (c transform)
3. {functions.c.o, main.c.o} $\rightarrow$ {a.out} (o transform)

The C language goes through three build phases: preprocess, compile, and link. The preprocess step takes macros like #include "file.h" and replaces them with the specified file before passing the .c files to the compiler. The compiler generates .o (*object files)* from the .c files. Object files are linked together to create an executable (the default executable name on linux is a.out)

The passthrough transform is used to retain files in the build directory but remove them from the compilation set. Since .h files are handled by the .c compiler, there is no need for pcompiler to consider them further. pcompiler expects to only produce one output file (or *terminal*), which means keeping the .h around would result in a failed compilation.

While most advanced programmers would prefer to use a Makefile or other build system for their code, the syntax of these files are often esoteric and distract from mastering the basics of programming. Since the Link's target audience is middle and high school students, the decision was made to avoid manual build systems altogether. Enabling support for custom Makefiles would be an interesting project to attempt for the Link.

3.6 botui [7]

botui is the graphical user interface the user is presented upon starting a Link. botui focuses on four primary tasks: Sensor visualization, simple motor/servo control, device configuration, and, perhaps most importantly, managing user programs. botui is itself a libkovan instance that communicates with the system using standard libkovan functions. This means that any behavior exhibited by botui can be replicated in a user program or alternate user interface painlessly.

3.7 kovan-kmod [8]

kovan-kmod is a kernel module that mediates communication between libkovan instances and the FPGA. The chosen IPC mechanism was UDP, as UDP allows full duplex communication and avoids the synchronization headaches of other IPC mechanisms. This implementation could also be used to theoretically send commands to kovan-kmod from a remote libkovan instance. The motor PID controllers currently reside inside of kovan-kmod and is continuously called using a kernel timer. The PID controllers were originally intended to

reside on the FPGA, but the Link's FPGA is too small to hold the relatively complex sequential logic required for four PID controllers. This means that motor PID control is intensive on the CPU and very "rough" compared to FPGA or co-processor based PID controllers. This issue will hopefully be addressed in a future version of the controller.

3.8 `kovan-recovery` [9]

Recovery mode is a special boot mode that is used to perform firmware upgrades and recover semi-bricked devices. To understand `kovan-recovery`, it is first necessary to understand the boot process of the Link and the initialization process of linux. The Link uses a custom Master Boot Record (MBR) initially designed by Chumby Industries for the Chumby. [1] This custom MBR contains two linux kernels: One regular kernel and one special recovery kernel. It is important to reiterate that these kernels *are not* stored on the device's filesystem, but instead directly in the MBR. The process of booting a linux kernel goes something like this:

1. Load the kernel into RAM.
2. The kernel then builds what is called an `initramfs` and executes a process called `init`. `initramfs` contains an extremely minimal filesystem that is used by the `init` process. The idea behind an `initramfs` is that certain operations that don't belong in the kernel can be executed before the main filesystem is loaded. For example, imagine the case in which a filesystem is encrypted. Linux doesn't know how to use an encrypted filesystem, so a developer could design an `initramfs` that loads the correct keys, decrypts the filesystem, and then continues the boot process. It should be noted that the `initramfs` is actually compiled *directly into* the kernel itself.
3. The real root filesystem is loaded and initialization of the system continues.

`kovan-recovery` takes advantage of `initramfs` to "hijack" the boot process before a root filesystem is mounted. This detail is important, as `kovan-recovery` works by overwriting the entire internal SD card. Since the SD card is being overwritten, it can not be used as a root filesystem. Since `kovan-recovery` and linux exist only in RAM, this is not an issue.

As mentioned earlier, the `initramfs` is compiled directly into the kernel. This means that the Link needs two kernels: One special recovery kernel and one regular kernel. Before the linux kernel is loaded, the side button is checked to see if it is pressed down. If it is, the special kernel is loaded instead of the regular one.

`kovan-recovery` itself is an extremely simple C program that uses `zlib` and basic file i/o to inflate and subsequently write to the internal drive. Since including libraries in the `initramfs` is burdensome, `kovan-recovery` performs double-buffered drawing directly to the `/dev/fb` LCD frame-buffer, using an extremely simple bitmapped font to perform font rendering.

Unix represents almost every installed device as a special type of file in the `/dev` directory. This means that a developer can open a device and write to it like it was a file. As noted above, `kovan-recovery` opens the LCD as a file and writes pixels at certain offsets that correspond to onscreen (x, y) locations. This concept is applied in several places throughout the system, and is important to understand.

[1] `http://en.wikipedia.org/wiki/Chumby`

3.9 KISS IDE [10]

An elementary understanding of KISS IDE 4's architecture is important for modifying and extending the Link.

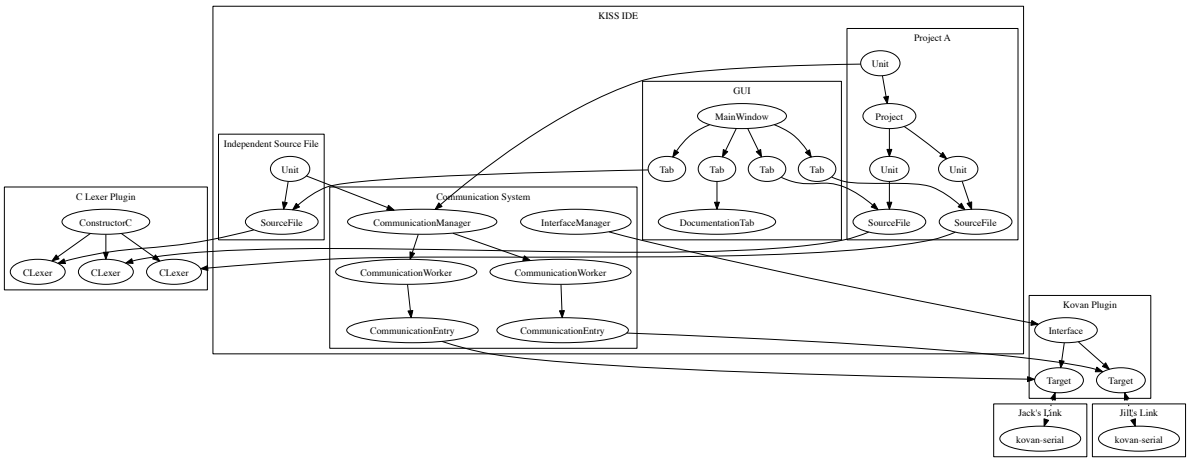

Figure 3: A Hypothetical Runtime Snapshot of KISS IDE

Two of the core concepts in KISS IDE are that of the `Interface` and `Target`. `Interfaces` encapsulate the the discovery, enumeration, and creation of `Targets` for a specific device. A `Target` is used to *download*, *compile*, *run*, and perform other operations on a connected device. Another core concept in KISS IDE is that of the `Unit`. A `Unit` is an abstract class that uses the visitor pattern to create in-memory project and source file archives. For example, imagine the existence of a project called "Task1" that contains three source files: `foo.c`, `bar.h`, and `baz.c`. Now imagine that the user invokes a *download* command on the source file `foo.c`. `foo.c` will ask that its `Unit` invoke a *download* command on the top-level `Unit`, which is Task1. Task1's unit will then create an empty archive, add project files to it, and *visit* its children `Units` recursively. A child `Unit` may then add its required files to the archive. Once this recursive process finishes, the completed archive is sent wrapped in a `CommunicationEntry` and queued in the `CommunicationManager`. The `CommunicationManager` then handles non-blocking execution of the task on the given target.

The most important thing to note here is that KISS IDE *does not* compile source files locally when communicating with a Link. A *compile* command results in the following chain of actions:

1. Download the source archive to the Link
2. Request a remote compilation on the Link.
3. The Link extracts the archive to a temporary directory
4. The Link invokes pcompiler on the directory and blocks until the compilation is completed.

5. KISS IDE then receives these results from the Link.
6. Results are formatted, highlighted, and then presented to the user.

While compile time is negatively influenced by this methodology, there are several notable advantages:

- A firmware exposing new APIs on the Link doesn't require a new version of KISS IDE. This was a large problem with the CBCv2 robotics controller.
- KISS IDE can work without a local compiler.
- The archive format is language neutral, which means languages can be added or removed without changing KISS IDE (although syntax highlighting might not work).

3.10 `ks2` [11]

`ks2` is KIPR's 2D simulator for the Link. Internally, `ks2` implements the same protocol and UDP server used by `kovan-kmod`, meaning local instances of `libkovan` will be able to seamlessly communicate with the simulator rather than an actual Link. `ks2` also implements sections of the `kovan-serial` network server to communicate with KISS IDE in exactly the same way a Link would.

4 Conclusion

While this document covers several pieces of the Link's firmware and attempts to give context to certain features, it is by no means comprehensive. If you have any questions or concerns, please feel free to contact the authors for more information. Once again, for detailed information please visit this paper's website: `http://bmcdorman.github.io/link`. This paper's website is also an open source and public domain git repository, so feel free to fork it and add insight!

4.1 Acknowledgements

- Thank you to Clemens Koza for his efforts in creating reliable instructions for setting up a build server from scratch.
- Thank you to the users that have coped with the early instabilities of the new system.

References

[1] J. Southerland. kovan-fpga. https://github.com/kipr/kovan-fpga, 2013.

[2] B. McDorman. libkovanserial. http://github.com/kipr/libkovanserial, 2013.

[3] B. McDorman. kovan-serial. http://github.com/kipr/kovan-serial, 2013.

[4] B. McDorman and J. Southerland. libkovan. http://github.com/kipr/libkovan, 2013.

[5] B. McDorman. libkar. http://github.com/kipr/libkar, 2013.

[6] B. McDorman and N. Zaman. pcompiler. http://github.com/kipr/pcompiler, 2013.

[7] B. McDorman, J. Southerland, and N. Zaman. botui. http://github.com/kipr/botui, 2013.

[8] J. Southerland and B. McDorman. kovan-kmod. http://github.com/kipr/kovan-kmod, 2013.

[9] B. McDorman. kovan-recovery. http://github.com/kipr/kovan-recovery, 2013.

[10] B. McDorman, N. Zaman, and J. Villatoro. Kiss ide. http://github.com/kipr/kiss, 2013.

[11] B. McDorman and J. Southerland. ks2. http://github.com/kipr/ks2, 2013.

This document was compiled on June 19, 2013.