RPI.icestick: a set of hardware and software components
which simplify communication between the Raspberry Pi and iCEstick.

John Emmons

Abstract:

RPI.icestick is a wrapper around the SPI protocol which provides a high-level interface for communication between the Raspberry Pi and iCEstick. Users only need to “plug” the iCEstick’s female GPIO pin header into the Raspberry Pi’s male GPIO pin header to correctly “wire” the devices to one another. The RPI.icestick library allows users to send and receive individual bytes (higher level primitives are under development). Python bindings are also provided.

Source: https://github.com/jremmons/RPI.icestick

Introduction:

Small, inexpensive development boards such as the Raspberry Pi and iCEstick FPGA have taken over the world of hobbyist computing. Their popularity has led to the formation of burgeoning communities that donate their time and expertise to making these devices easy to the use and extensible. These devices have become an integral part of many computer science curricula and are the best choice for non-experts looking to build projects which require a hardware component. However, while these devices are easier to use than most other computer hardware, support for more complex tasks is limited. In particular, software and hardware interfaces for communication between devices via standards like SPI is not provided by default. This severely limits the ability of non-experts to combine their devices.

Furthermore, even for developers who are capable of deciphering communication protocol specifications and implementing these components themselves, there is still much to be gained by contributing to an open source community of tools. Agile development relies on rapid iteration of designs and frequent testing. This is only possible if the tools and libraries available to developers (1) provide high level interfaces (i.e. abstract the frequently complicated details of hardware and software systems) and (2) can be tested quickly (i.e. requires little or no advanced preparation to implement). For the development of systems which require both hardware and software

components, agile development has historically not be possible do to failing in both these areas. RPI.icestick partially addresses these issues by providing a high level interface and ability to rapidly iterate on designs which require communication between the Raspberry Pi and iCEstick.

Related works:

RPI.icestick is not the first community developed open source tool/library for making communication between hardware and software components easier. WiringPi, PySerial, py-spide, and others all seek to improve interfaces for communication between software and hardware components. However, these libraries expect a detailed understanding of the underlying protocols to be useful for FPGA devices (i.e. users would need to implement an SPI controller and other data management logic themselves). For users with only a basic understanding of FPGAs and these protocols, this presents a nearly insurmountable obstacle.

RPI.icestick is distinct from other libraries for communication because it is specifically targeted to the iCEstick (ice40) FPGA and Raspberry Pi, the most common hobbyist devices in their respective domains. No other library which bridges the gap between these specific devices existed prior to this work.

System design and implementation:

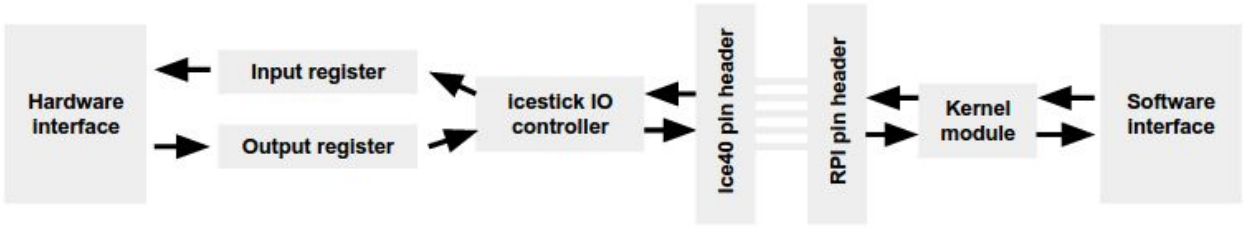

Figure 1: System diagram of the hardware and software components of RPI.icestick.

RPI.icestick exposes `data_outgoing` and `data_incoming` registers on the iCEstick FPGA to the user. A hardware queue for buffered reads and writes will also be available in future versions of the system so that the burden of managing large data transfers can be alleviated. On the software side, users make calls to `send` and `receive` functions (in C or with Python bindings) which communicate with the `spi-bcm2835` kernel module on the Raspberry Pi. The overall system architecture is detailed in figure 1.

Internally to RPI.icestick, data is transferred with the SPI protocol via the GPIO pin headers on the Raspberry Pi and iCEstick. The location of the SPI pins is chosen so the iCEstick can be “plugged” into the pin header on the Raspberry Pi, making it easier for hobbyist and non-experts to connect their devices by eliminating the need for additional material. The full capabilities of the SPI interface is not exposed to the user on either the software or hardware side; rather, simplified one-byte-at-a-time transfer semantics are imposed. This greatly simplifies the hardware on the FPGA and the software on the Raspberry Pi which provides an easier interface for those who want to build-on and extend the functionality of this library.

The RPI.icestick software interfaces are implemented in C with some glue logic in Python for the Python bindings. The hardware interface is implemented in Verilog; a port to magma/mantle is the next logical step for this project.

Evaluation and limitations:

To evaluate the quality of RPI.icestick, I measured the speed of an “echo” test and the ease with which I was able to write code in Python which interacted with the iCEstick FPGA.

For the echo test, the Raspberry Pi sent over a random 8-bit number to the FPGA; subsequently, the FPGA would return the same 8-bit number to the Raspberry Pi on the next data transfer. When running the SPI interface clock at 500KHz there were no errors in the memory transfers. Due to delays introduced by code running on the Raspberry Pi between transfers, the total throughput of the system was ~200Kbit/s. It is likely that a higher SPI interface clock frequencies could be used, but for the sake of simplicity and debugging purposes I did not run evaluations at higher frequencies. Also, since the system only sent one byte (i.e. 8-bits) per transfer, much of the time was spent starting, ending, and waiting between transfers. This limits the throughput of the system significantly.

The ease of use of the system is a subjective measure and I did not perform a user study to better quantify this metric. From my perspective, having the ability to send/receive data with a function call in software (similar to the API of a standard network socket) and have the data appear in registers in hardware is a very intuitive and easy to conceptualize abstraction. Further work will need to be done to measure the quality of this interface completely though.

Future work:

Future work on this project will be focused on addressing the limitations outlined in the previous section. The next steps will be to (1) improve the data transfer speed, (2) quantify the “goodness” of the library API, (3) add other hardware and software interfaces for bulk memory transfers (i.e. queues), and (4) port the Verilog to Magma/Mantle.