An open-source, extensible system for laboratory timing and control

Peter E. Gaskell,[a] Jeremy J. Thorn, Sequoia Alba, and Daniel A. Steck
*Department of Physics and Oregon Center for Optics, University of Oregon, Eugene,
Oregon 97403-1274, USA*

We describe a simple system for timing and control, which provides control of analog, digital, and radio-frequency signals. Our system differs from most common laboratory setups in that it is open source, built from off-the-shelf components, synchronized to a common and accurate clock, and connected over an Ethernet network. A simple bus architecture facilitates creating new and specialized devices with only moderate experience in circuit design. Each device operates independently, requiring only an Ethernet network connection to the controlling computer, a clock signal, and a trigger signal. This makes the system highly robust and scalable. The devices can all be connected to a single external clock, allowing synchronous operation of a large number of devices for situations requiring precise timing of many parallel control and acquisition channels. Provided an accurate enough clock, these devices are capable of triggering events separated by one day with near-microsecond precision. We have achieved precisions of ~ 0.1 ppb (parts per 10^9) over 16 s. © *2009 American Institute of Physics.* [doi:10.1063/1.3250825]

I. INTRODUCTION

In a wide range of fields, including cold-atom physics, experiments require complex sequences of timing and control signals that are both precise and repeatable. The standard approach to meeting these needs is to control the experiment with a desktop computer outfitted with a number of internally mounted (e.g., PCI/PCI-X), commercial "expansion cards," which are designed to produce the requisite timing pulses, control voltages, and real-time waveforms. Many of the available options include some data-acquisition functions as well, and may serve as fast interfaces for more complex instruments such as cameras. This hardware is then controlled from the computer via proprietary libraries, and, from the user's perspective, often through high-level programming environments (e.g., LABVIEW or MATLAB). Despite being immensely popular, this approach has a number of serious problems: Proprietary hardware and software is generally quite expensive, and traditional interfaces to external hardware (e.g., IEEE-488 and RS-232) tend to be expensive, slow, and involve limited cable distances. Hardware capabilities as enumerated in advertising literature can in some cases be misleading or even differ significantly from reality. Available hardware capabilities and software support also vary considerably, depending on the operating system running on the computer. Proprietary drivers and firmware cannot be modified to suit unusual applications not foreseen by the vendor. Even worse, driver incompatibilities that inevitably arise with upgrades to the operating system can paralyze an experiment until workarounds are implemented. In ultracold-atom experiments, it is unfortunately not uncommon for an experiment to be controlled by two interfaced computers, one new and one much older, when older proprietary hardware must be run by a sufficiently primitive computer and the cost of upgrading the hardware to support modern interfaces is prohibitive.

Other groups have worked on these problems, ranging from reprogramming an existing device to perform differently[1] to designing a new control system.[2] The general purpose interface bus (GPIB, or IEEE-488) has been under much scrutiny, since it tends to be associated with extra hardware, proprietary drivers, poor timing issues, and short, expensive cables. For example, Hall[3] discussed a scheme to improve the timing of GPIB operations, and Gao and Madison[4] discussed a scheme to convert GPIB to Ethernet, bypassing some of the proprietary drivers, the expansion cards, and the short cable limitations.

We have taken this work further, and developed hardware and software for precise timing and control of laboratory experiments. We have successfully used the architecture in our ultracold-atom experiments,[5,6] where we used no proprietary software or add-on hardware beyond the basic desktop workstation. Our design criteria for this architecture are as follows.

(1) Performance meets the needs of cutting-edge experiments (especially with ultracold atoms), with low construction cost so that this technology may be used, for example, in teaching laboratories with modest budgets.
(2) Open-source hardware and software for longevity and easy customization.
(3) Fast, inexpensive, and long-distance connectivity, which we achieve with the transmission control protocol over internet protocol standard (TCP/IP) and user datagram protocol over internet protocol (UDP/IP), both over Ethernet networks.
(4) Decentralized, modular, and scalable hardware, where timing and control hardware is divided into relatively compact and physically separate "boxes," and each box

---
[a] Present address: Department of Electrical and Computer Engineering, McGill University, Montreal, Quebec.
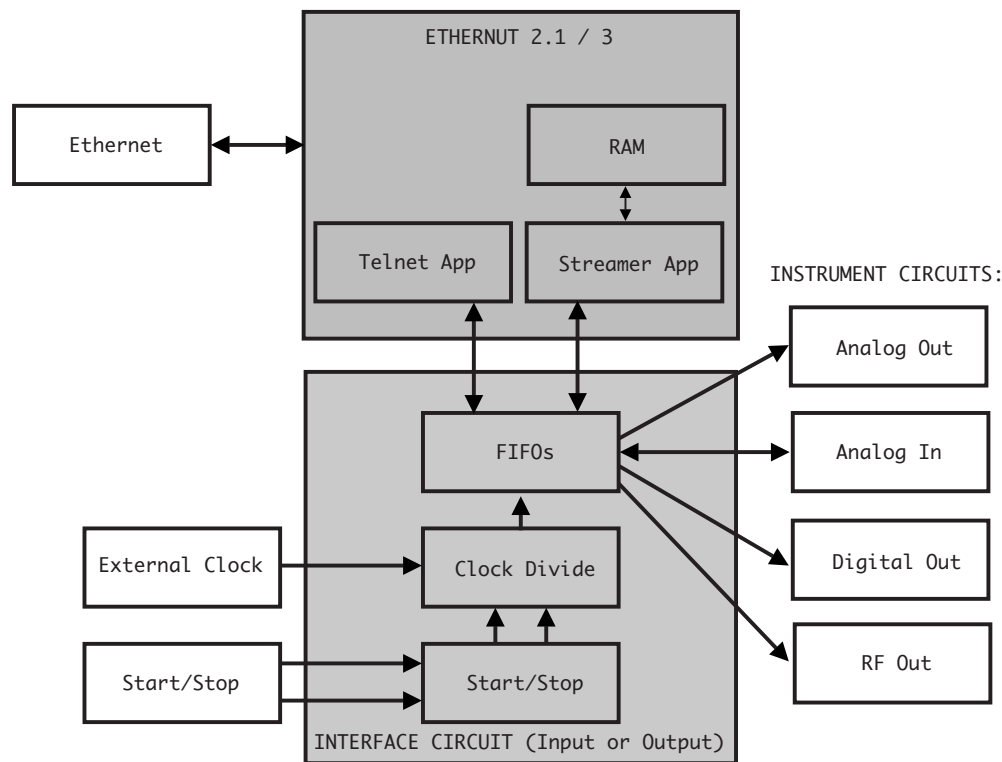
FIG. 1. Basic components of our control system. The ETHERNUT handles network communication and produces the commands for the instrument circuits. The interface circuit buffers the commands and sends them to the instrument boards with precise timing governed by an external clock, with inputs for starting and stopping the output. The instrument circuits control or read data from the experiment.

is controlled by an embedded microcontroller. By placing some "intelligence" in each box, the main computer needs no special hardware beyond a standard network interface. Furthermore, in the case of hardware problems, the modular external boxes are easily replaced by spares. Replaced boxes can be repaired and used as spares.

(5) Platform independence, mainly since the host computer only needs to support standard Ethernet communication, which is now so widely implemented that it will be supported on virtually every useful platform for many years to come. We use PERL to program the hardware, though any network-aware programming environment is suitable for this purpose.

In realizing the timing and control hardware here, we are leveraging existing hardware projects. We use low-cost, open-source, off-the-shelf "ETHERNUT" microcontrollers[7] to manage the main network link between the host computer and the control hardware. We also use digital, analog, and direct digital synthesis electronics from a project at the University of Texas at Austin.[8] These circuit designs are freely available, and are designed to be controlled over a digital data bus by a commercial digital output board installed in the host computer. The bulk of the work described here involves the design and implementation of interface electronics that allow the ETHERNUT microcontrollers to drive any circuit using the same data bus in real time. These electronics enable a distributed, real-time control system, free of proprietary software or hardware as outlined above. The main challenge here is that despite the many advantages of Ethernet for commu-

nication with timing and control hardware, because of the nondeterministic nature of the standard protocols, Ethernet networks are not directly suited for generating precise, real-time signals. Thus, all network communication in an experiment occurs *before* the real-time sequence. Once all control boxes are ready, they can be triggered via the interface electronics, which buffer the quasi-real-time output of the ETHERNUT microcontrollers, allowing the output to be synchronized to a high-precision external clock oscillator. Thus, all control signals can be precisely synchronized across many devices.

II. HARDWARE OVERVIEW

Our hardware setup is simple and modular. Each box shares an Ethernet network with at least one computer, used to program and run the boxes. Each box is solely an input device or an output device, and contains three parts:

(1) ETHERNUT (an embedded computer),
(2) interface circuit, and
(3) instrument circuit (possibly multiple copies).

The ETHERNUT is a simple computer that we use as an interface between the main computer (over an Ethernet network) and the control circuits. The interface circuit buffers the commands from the ETHERNUT with first-in-first-out (FIFO) buffers, and passes those commands to the instrument circuits, maintaining an even timing and synchronization with other boxes. The instrument circuit actually produces the output signals that control the experiment. These parts, shown in Fig. 1, will be described in Secs. II–X. All

schematics, board layouts, and software we designed are available on our project website.[9]

III. ETHERNUT PLATFORM AND NUT/OS

The ETHERNUT is an open-source hardware platform that runs an open-source, real-time operating system called NUT/OS. The ETHERNUT and NUT/OS were designed and are distributed by Egnite GmbH.[7]

The ETHERNUT comes in multiple hardware versions, but the firmware for this project was written for version 2.1. Our firmware includes code for version 3 of the ETHERNUTs, but has not been updated to remain compatible with current ARM compilers. Egnite GmbH provides schematics, computer-aided-design files, and a bill of materials for each version so users can manufacture the devices themselves if they cannot or do not want to purchase the ETHERNUTs assembled and tested. The ETHERNUT 2.1 is based on the Atmel ATMega128 microprocessor running at up to 16 MHz and programs are compiled with AVR-GCC. The ETHERNUT 3 uses a 70 MHz ARM7 processor, and the software is compiled with ARM-GCC.

Both the ETHERNUT 2 and ETHERNUT 3 have a 64-pin expansion bus, which is intended as an interface for external hardware. This bus contains many signal lines (ports) that may be used either as digital outputs or digital inputs for the ETHERNUT. Many of these have dual usages that prevent them from being used arbitrarily. We take advantage of the external memory interface signal lines and a few of the general-purpose signal lines for controlling the interface circuits and transferring data to the FIFOs. The external memory interface has 16 address lines and 8 data lines. However, the same interface is used to read and write to internal memory on the ETHERNUT 2 boards, so we lock 8 of the address bit lines to an address reserved for external memory. We use the remaining 8 bits of the address bus and the 8 bits of the data bus to send data to the data FIFOs, and a general-purpose 8-bit input-output ports port to send data to the address FIFO. All 24 signal lines that we use to write to the FIFOs, as well as all the other monitoring and control lines, are available through the 64-pin expansion bus. There is also a separate analog port that we do not use but could be configured for either analog-digital conversion or as another digital data port.

NUT/OS supports simple, non-pre-emptible threads, network stacks, and many other useful features. It also provides a convenient, well-documented application programming interface for the ETHERNUT. All of the firmware we wrote for ETHERNUTs uses NUT/OS. Instructions on setting up NUT/OS and our application to compile for either ETHERNUT board are available in our website.[9]

We use ETHERNUTs as Ethernet interfaces and memory-storage to simplify interfacing control circuitry with the computer system. The controlling computer can connect to the ETHERNUTs and send single commands to be executed immediately, or large streams of commands with timing information to be executed later. These commands are specific to the particular instrument circuit used in the device.

IV. INTERFACE CIRCUIT

Every device uses the same specialized interface circuit, which buffers commands to be sent to the instrument circuit. These circuits allow us to control the timing of the commands sent to the instrument boards with high precision.

The ETHERNUT reads commands from its own internal memory, and generates commands for the instrument circuits. Using the internal memory solves the problem of sending precisely timed packets over a network, but it is difficult to precisely time multiple, fast ETHERNUT outputs. As described in Sec. VII B, the data that we store on the ETHERNUTs is compressed in a very simple fashion, and decompressing causes the output rate to vary. In order to get an even output rate among multiple data sets, we would have to slow down the output to match the slowest decompressing rate. Also, the memory on the ETHERNUT 2 is paged, and switching memory pages takes a few clock cycles as well. Thus, synchronizing multiple ETHERNUTs would require either slowing the maximum output rate or some complicated method to ensure that every operation on each ETHERNUT took the same amount of time. This would be in addition to making sure the clocks of the ETHERNUTs were synchronized, which would be particularly difficult if some ETHERNUTs had different clock rates (which is true between ETHERNUT 2 and ETHERNUT 3).

We solved these problems by introducing the interface circuit boards. Each board has FIFOs that buffer the output of an attached ETHERNUT. The ETHERNUTs only try to keep the FIFOs full. We then have a single clock control the output (or unloading) of all FIFOs across multiple boxes. With this system, commands are sent in a timely fashion and are synchronized across multiple boxes. Furthermore, each box can be started and stopped independently, without losing synchronization, by having the interface circuit block the unloading clock signal from the FIFOs. We also have the option of having different boards use different clock dividers, which allows each board to have a different output rate but still remain synchronized.

The interface circuit is connected to instrument circuits by a standard 50-pin, flat ribbon cable. Each output board has headers for two cables, but more devices can be connected to the interface by using ribbon cables with multiple connectors. This bus system was designed at the University of Texas at Austin for ultracold-atom experiments.[8] It is a simple parallel bus with 8 address lines, 16 data lines, and 1 strobe line. Each device is assigned with a unique address. When the strobe is pulsed, the addressed device reads the data from the bus and performs the specified function. The specifics of the bus are available online.[8]

The interface circuits have three external inputs: START, STOP, and CLOCK. The CLOCK input has an adjustable Schmitt trigger so that either sine or square waves can be converted into regular transistor-to-transistor logic (TTL) pulses within the circuit. This TTL clock signal is then divided down to a slower rate [controlled by a dual inline package (DIP) switch on the circuit], gated (either passed or blocked) based on the state of the interface circuit, and then sent to the unload clock of the output FIFOs. The START

and STOP inputs turn the gate on and off, respectively, controlling whether or not the FIFOs send new commands to the instrument circuits.

The interface circuits also have a 64-pin expansion port that connects directly to the 64-pin expansion port on the ETHERNUT. This bus is used to transfer data between the FIFOs and ETHERNUTs, and includes lines to control and monitor the FIFOs, as well as possible overrides to the START and STOP inputs. We also use it to supply power to the ETHERNUT, reducing the number of power lines we require. When an ETHERNUT enters streaming mode at the start of an experiment, it clears the FIFOs, turns the interface circuit clock off, and then enters a loop where it sends instrument circuit commands to the FIFOs whenever they are not full. Initially, the FIFOs just fill up because no output is being sent. Once a START pulse is received, the FIFOs begin to send out commands and the ETHERNUT keeps them full. Once the ETHERNUT has sent all the commands that are needed, it exits streaming mode, and the FIFOs drain. Once the FIFOs are empty, the clock is again disabled.

We use a 10 MHz sine wave generated by an atomic clock as the CLOCK input for all our boxes, which gets divided down to either 250 or 500 kHz, depending on the instrument circuit that is attached. These speeds were chosen because they are sufficient for our needs, and because they are near the maximum speed at which the ETHERNUT 2.1 boards (without increasing the clock rate) can keep up with the data output. There are several options for higher speeds. One option is to use a faster ETHERNUT. We timed an ETHERNUT 3, which has a 70 MHz ARM processor, as capable of producing output at 1.67 MHz (but it has substantially less memory to store programs than an ETHERNUT 2.1). A second option, if fast output is only needed for very short time periods (shorter than the time it takes for all the data on a FIFO to be clocked out), is that a given board could be run for only those short periods of time. The board could be stopped and started using the START and STOP inputs, with the FIFOs unloading only for short periods of times so that the ETHERNUT could refill the FIFOs during the stopped times. The interface circuit provides a line to tell the ETHERNUT whether the FIFOs ever went empty during a given experiment so the ETHERNUT could be programed to check if the FIFOs ever went empty while the ETHERNUT was trying to fill it. Should the FIFOs ever empty, the output clock will shut off, so for this usage, the ETHERNUT should be programed to hold a START pulse while in streaming mode. The full schematics of revision 4 of the output and input interface circuits are included in Appendix.

V. INSTRUMENT CIRCUITS

Our system, consisting of an ETHERNUT and an interface circuit, was designed to be a general enough control interface to be able to control a wide range of instrument circuits. However, it was specifically designed and tested to work with three existing instrument circuits.[8] These circuits have broad capabilities that should be useful and perhaps completely sufficient in many laboratory control settings.

A. Digital output

Many of the controllable devices in our laboratory have TTL-compatible inputs, typically for enabling or disabling some function of the device. We use a digital output instrument circuit to control these.

Each digital output circuit provides 16 TTL-compatible outputs, each buffered by a transistor in order to drive 50 Ω loads. The digital output circuits have 8-bit addresses (selected by a DIP switch), allowing us to connect multiple digital output circuits to a single interface circuit. Each command consists of 8 address bits that select a circuit and 16 data bits that control the values of the 16 outputs on that circuit.

We drive two digital output circuits off of one interface circuit, with a clock divided down to 500 kHz, and send commands to one circuit per clock cycle, alternating between the two circuits. This gives us 32 TTL channels, with the ability to update each output every 4 μs (a 250 kHz output rate). The interface circuit is capable of controlling more digital output circuits, but only has two connectors. To add more digital output circuits to an interface circuit, one needs only add the extra circuits in parallel to the existing circuits. The firmware we use currently requires exactly two instrument circuits.

B. Analog output

Many of the controllable devices in our laboratory also have analog control inputs. These inputs respond to an input voltage, which might set a current in a power supply, control the strength of a magnetic field, or set the frequency of a voltage-controlled oscillator. We control many of these analog inputs with analog output instrument circuits.

Each analog output circuit consists of eight channels, each with a 16-bit digital-to-analog converter. The circuits are again addressable, with three of the 8 address bits used for selecting one of the eight output channels within the circuit, and the remaining 5 bits available for selecting a particular analog output circuit. Each analog output circuit connected to a particular interface circuit should have a unique 5-bit address, determined by an onboard DIP switch. A single command for an analog output circuit consists of those 8 address bits and 16 data bits, which determine the output voltage (from ± 10 V).

We drive two analog output circuits off a single interface circuit, with the clock divided down to 250 kHz. This way, our analog output circuit output rate matches our digital output circuit output rate, except that the digital output circuits update every output at a rate of 250 kHz, whereas only one output of the analog output circuits can be updated each cycle. The firmware we use should support from 1 to 32 analog output circuits, but has only been tested with 2. Since the interface circuit only has two connectors for instrument circuits, extras would need to be added in parallel with those connections.

C. Direct digital synthesizer

The protocols for controlling the direct digital synthesizer (DDS) instrument circuits are still under development.
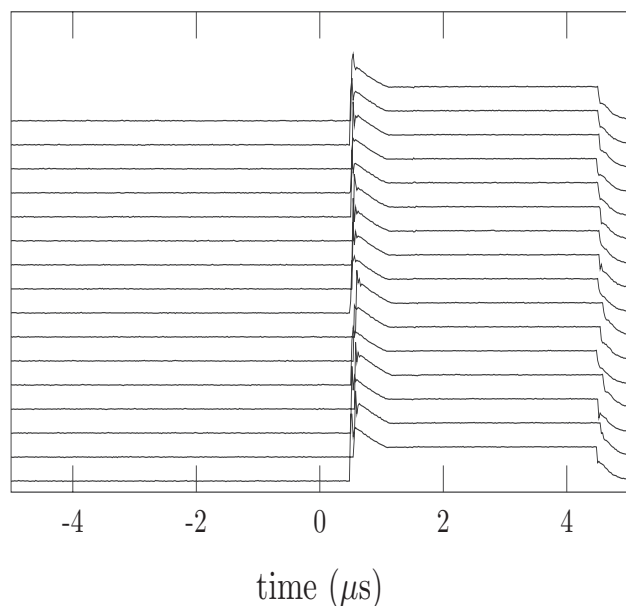
FIG. 2. This plot shows 16 repetitions of the same program, measuring the output (in volts) of a digital output instrument circuit programmed to produce 16 pulses (4 μs followed by 4 μs off) after waiting for 16 s. Each data set is shifted vertically for clarity. The data were taken with an oscilloscope locked to an independent time reference, which triggered off the same trigger pulse that started the box. The horizontal axis measures the time in microseconds, starting from 16 s after the trigger pulse, and captures only the first pulse. The timing variations are barely noticeable at this scale.

The DDS circuits utilize a programmable sine-wave synthesizer that produces and mixes up to two sine waves of programmable frequency, amplitude, and phase. The synthesizer compares to an external frequency reference, which we will run off an atomic clock for extremely accurate frequency control.

VI. PERFORMANCE

Here we show results from a test of the repeatability and accuracy of our system. We programed a digital output box to wait for 16 s, and then produce 16 short (4 μs on, 4 μs off) pulses. We measured these pulses with an oscilloscope that was locked to an independent time reference. The oscilloscope was triggered by the same signal used as a START input to the digital output box. Figure 2 shows the beginning of the pulse train, demonstrating how repeatable and accurate the pulses are. The pulses do not start right at 16 s (0 in Fig. 2) due to a 0.05 ppm uncertainty between the oscilloscope clock and the clock used by the digital output box.

The pulse times in Fig. 2 vary by 0.1 μs. This tenth-microsecond variation is due to our triggering mechanism. The trigger was not synchronized to the clock for the digital boxes. Thus, while the oscilloscope triggered on the pulse, the digital boxes waited until the start of the next clock cycle of the 10 MHz clock. The tenth-microsecond spread is because the oscilloscope could trigger anywhere from zero to a full period (a tenth-microsecond) before the digital output box actually started.

In an actual experiment where the a tenth-microsecond drift is unacceptable, the trigger signal could be gated to the clock, eliminating the tenth-microsecond drift in start times.

Alternatively, a digital box could be used to trigger all other boxes at precise (and possibly different) times. When we tested the timing between pulses over 16 s (from the same box), we found no variations to within the resolution of our oscilloscope (down to 0.02 μs). Thus, with this system, we can fairly easily achieve the same accuracy and precision of our clock. With even a modest (by today's standards) atomic clock, we can achieve both an accuracy and a precision of $\sim 10^{-10}$.

VII. COMMUNICATION PROTOCOL

The ETHERNUT in every box is programed with firmware specific to the instrument circuits it controls. The devices are different enough to warrant different data structures, although the overall protocols are the same between devices. Each ETHERNUT is programed with two methods for connecting to it and controlling the instrument circuits it interfaces. We call these two methods the *telnet thread* and the *data streamer*.

A. Telnet thread

The telnet thread is intended for human interface and initial experimental setup. A client connects to the telnet thread in order to control the box using a simple menu-based system. This system is intended to be fairly simple for either a person to use or a script to parse. The client (typically a person or script on the controlling computer) sends one-line ASCII commands terminated by ASCII linefeeds (byte-value 0A in hexadecimal). Carriage returns (byte-value 0D in hexadecimal) in addition to linefeeds also work. The ETHERNUT responds to commands with some data and/or queries for more information.

When the connection is first established, the ETHERNUT sends a menu header. In any menu, the l or list command (or any other unspecified command) will result in a list of available commands in that menu. The q or quit command will back up a menu, or terminate the connection from the top menu. Each menu includes a menu header, which is always a three-digit code followed by a brief description. These numbers are intended to facilitate automated navigation of the menus. Some commands enter other menus, while others actually change settings on the ETHERNUT or cause commands to be sent to the instrument circuits. The ETHERNUTs can force a START pulse to the interface circuit to immediately send the commands to the instrument circuits.

The connection to the telnet thread is via TCP/IP, port 23 (the standard telnet port). Since our software currently just sends ASCII text back and forth and ignores non-ASCII characters, it should work with most telnet clients.

B. Data streamer

The data streamer is for precisely timed operations and is designed to be able to run the ETHERNUTs at maximum speed to stream data out to the FIFOs. For this purpose, a program describing what to do at what times is sent to the ETHERNUT before performing an experiment, and then the ETHERNUT can stream the data out during the experiment. We separated the tasks of actual streaming and sending the data
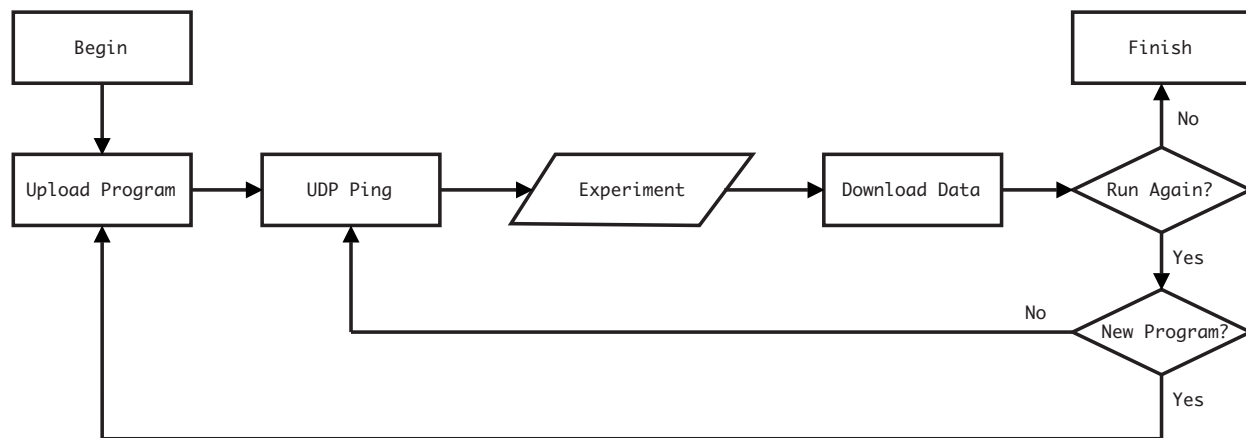
FIG. 3. Flow chart showing the actions the host computer takes in a typical experiment. The upload program stage includes any initialization of the boxes that are necessary.

for streaming. This allows a user to send one data set to the ETHERNUT, change her mind, and overwrite it with a second data set, and then have the ETHERNUT stream that multiple times without having to send more data.

The data are sent directly via a TCP/IP connection on port 24. Upon opening a connection, the ETHERNUT sends a single line, containing a decimal number (in ASCII) followed by a new line (character value 0A in hexadecimal). This is the number of bytes it is ready to accept. When that many bytes have been sent, it will send back another request for data. This will continue until either the sending computer shuts down the connection (indicating end of data), or the ETHERNUT runs out of memory, in which case it will request 0 bytes and shut down the connection itself. Every new connection clears the data buffers on the ETHERNUT.

Once the data for the ETHERNUTs have been sent to them via TCP/IP port 24, the data can either be replaced (by starting another connection) or streamed out to the FIFOs. Our intent is that the computer has full control over the experiment using only the Ethernet connection, including signaling the ETHERNUTs to start streaming to the FIFOs. Ideally, the signal to start streaming would be a normal internet control message protocol echo request (ICMP ping) or some other standard ping. However, NUT/OS intercepts and responds to pings and does not pass the ping to the application. Instead we use an equally simple method that we call a *UDP ping*, and is simply a UDP/IP connection to port 11235. The client computer sends a short string to that port and the ETHERNUT responds with a fixed response. Since UDP is stateless, there are no sockets to close on the ETHERNUT and no closing handshake as in TCP. Thus, immediately after sending the response, the ETHERNUT can begin streaming. This is important because we stream data from the ETHERNUTs to the FIFOs near the maximum speed of the ETHERNUTs. Therefore, if the ETHERNUT processor spends some time dealing with network traffic, the FIFOs may run empty, ruining the timing of the output. Thus, during this streaming mode, the ETHERNUTs ignore all network traffic, and only start responding again after the streaming has finished. The controlling computer needs to know that the ETHERNUTs are ready to stream nearly immediately after the UDP ping response since that is the last it hears from them. We would like to empha-

size that the UDP ping only starts the ETHERNUT streaming data to the FIFOs; the FIFOs do not actually output data until a hardware trigger pulse is received at the START input. The exception to this is the master box, which as described in Sec. VII C, sends its own START pulse. We use the master box to send hardware trigger pulses to all the other boxes, which synchronizes trigger pulses and all subsequent actions of all the boxes to the main clock. This way, the only timing that is dependent on the network is exactly when the master box starts, which does not matter in our experiments.

After an experiment has finished, the controlling computer briefly communicates with the ETHERNUTs to make sure that every box has finished streaming. It can then repeat the same sequence by sending another UDP ping or restart the process by opening another connection. Figure 3 shows this process schematically.

The data that get sent over TCP/IP port 24 are a concatenation of fixed-length data structures. The particular structures are device dependent but all start with a 4-byte integer for repetition. This is one less than the number of times that data structure will be used (written to the FIFOs) before moving on to the next structure. The repetition value could be thought of as the number of times to repeat the data *after* writing it out once. All data in the structures are little endian, chosen because it is the native byte order of the processors on ETHERNUT 2.1 and ETHERNUT 3. For example, the data structure for the digital output instrument circuits is 8 bytes. The first 4 bytes constitute a little-endian integer for the number of times to repeat the data, minus one. The next 2 bytes are the 16 bits for the first digital output circuit (each ETHERNUT controls two), and the final 2 bytes are the 16 bits for the second digital output circuit. We have the ETHERNUTs programmed to write the data for each circuit sequentially. In the digital output boxes, we run the FIFO output clocks at 500 kHz. The FIFOs actually write to one digital output circuit at a time so each of the two output boards in the box are being updated at 250 kHz. Thus, a data structure with a repetition value of 0 will last 1 cycle, or 4 μs. A data structure with a repetition value of 999 will last 4 ms before the next data structure is read and executed. Figure 4 shows the data structures used in our digital output boxes and our analog output boxes.

```
Digital Out          Analog Out
Data Format          Data Format
```

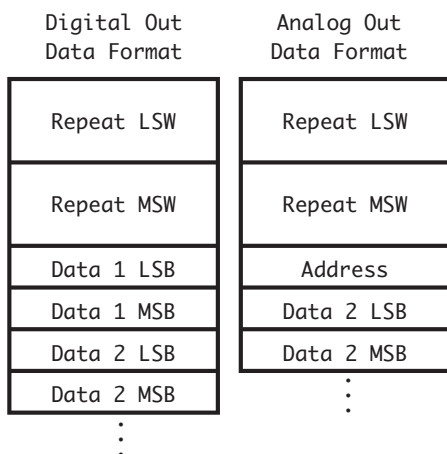| Repeat LSW	Repeat LSW
Repeat MSW	Repeat MSW
Data 1 LSB	Address
Data 1 MSB	Data 2 LSB
Data 2 LSB	Data 2 MSB
Data 2 MSB	⋮
⋮	

FIG. 4. Graphical representation of the data structures we use in the data streamers.

One advantage of our data structures is that there is only one data structure for every action. For an analog output circuit, an action constitutes changing a single channel. For our digital output circuits, an action constitutes setting all 32 bits. The 4-byte repetition value can be used for values up to around 4×10^9. At the FIFO clocking rate we use (one instrument circuit command every 4 μs), a single data structure could cause the ETHERNUT to wait over 4 h and still cause an event to happen with submicrosecond accuracy. When distinct actions occur every cycle, the programs become much longer. For example, our software on the ETHERNUT 2.1 can store a 480 kbytes program. At 8 bytes per data structure (for the digital output devices), the memory can contain 61 440 actions. If we assume an action occurs every clock cycle, then at our clock rate of 4 μs/cycle the memory will be exhausted in about a quarter of a second. In short, a full program has over 60 000 actions, which can span anywhere from about a quarter of a second to many years. These values can be changed by altering the clock divider on the interface circuit, although this is about the fastest clock that the ETHERNUT 2.1 can handle while filling the FIFOs faster than they drain.

C. Master box

As described so far, all boxes, in streaming mode, wait for a START pulse before anything actually happens. We prefer to have at least one box (which we call the *master box*) start automatically.

The MASTERBOX compiler flag in the ETHERNUT software designates which boxes are master boxes by controlling the behavior of the boxes on entering streaming mode. In all cases, once the ETHERNUTs begin streaming, they start filling the FIFOs. If the MASTERBOX flag is set, then, when the FIFOs fill up, that ETHERNUT forces a START pulse to start the FIFO output clock. If the MASTERBOX flag is not set, then the output clock will not start until a START pulse is received by the interface circuit. This allows us to trigger all the boxes in a repeatable manner. We currently have one master box, and some of the outputs of the master box are connected to the START inputs of the other boxes. We can then send our UDP ping to all the nonmaster boards, and

they will wait for the START pulse from the master box. A UDP ping to the master box will then trigger all the other boards in a repeatable, precise fashion, with possibly different trigger times.

One could designate a few boxes to trigger off some other condition, such as a beam being blocked during an experiment. The master box could also control START and STOP pulses for several different boxes, separately, turning them each on and off multiple times and at different times during a single experiment.

VIII. A CLIENT-SIDE EXAMPLE

The telnet thread is intended for simple manipulations, such as turning a single line high or low, and the menu system seems sufficient for that. Telnet programs can be used for human interface to the telnet thread, and shell scripts can communicate with the telnet thread via built-in TCP functionality, or programs such as NETCAT.

The data streamer is intended for long, complicated series of events, and so we found it handy to have an external library to handle most of the programming for us. Our particular library is written in PERL, and so should be portable to any operating system that supports TCP sockets, UDP sockets, and some version of multithreading.

Our library uses "events" to describe what actions to perform and when. The most basic event describes something happening to one device at one time, such as changing a single bit on a digital output box or setting a single channel on an analog output box. These can be concatenated into larger events that span multiple boxes and times. Once the entire set of events has been dictated, we call a subroutine called program_boxes(), passing the event list as an argument. The subroutine figures out what initial state all the boxes should be in, and, if necessary, connects to each one via the telnet thread to put them in that state. The subroutine then generates and sends all the necessary data to the ETHERNUTs for the data streamer. In our current setup, there is only one master box and the library knows which one it is, so this routine can also send out UDP pings to every box but the master.

At this point, all the boxes except for the master have loaded their FIFOs and are waiting for a START pulse. We then call the trigger() subroutine. If the master box has had data sent to it, this just sends a UDP ping to the master box, which assumes that the events contain whatever START pulses are necessary to trigger the other devices at the proper times. If the master box has not had data sent to it, the trigger() routine uses the telnet thread to generate a START pulse for the other boxes. At this point, all the devices should start streaming.

As a final step, we call the sleep_for_boxes() routine. This waits a given amount of time and then sends ICMP pings to all the programmed boxes. Once the boxes have finished their streaming, they should respond to the ICMP ping, allowing the script to tell that the experiment has finished.

Currently, we run all the boxes at speeds such that the FIFOs will never run empty. It is conceivable, though, that some applications will want to run a few select devices very briefly, at speeds that the ETHERNUTs cannot maintain. In these cases, it would be handy to know whether the FIFOs ever emptied during the streaming, as that would disrupt the timing. We have not implemented this feature in software; however, the EMPTY flag on the FIFOs is buffered and supplied to the ETHERNUT. With a small amount of extra code, the ETHERNUTs could be programmed to clear the flag at the start of streaming (while waiting for a START pulse) and check the status of this flag immediately after finishing streaming. The `sleep_for_boxes()` routine could then poll the ETHERNUTs to determine if the FIFOs ever emptied during the run.

We now provide some sample pseudocode demonstrating how simple a program using this system could be. In the example, we use three extra subroutines.

- `digital_action(box,high,low)`: Given a list of bits to set high (`high`) and a list of bits to set low (`low`), this creates an event that sets those bits on a digital output box at time zero. Since we have two digital output circuits in every digital output box, and every digital output circuit has 16 output ports each, each element of the bit lists is an integer from 0 to 31, inclusive. The first argument, `box`, is the network name of the box.
- `analog_action_set(box,channel,value)`: Creates an event that sets the given channel on an analog output box to the given value, at time zero. `box` is the network name of the box. `channel` is an integer from 0 to 15 (inclusive) and determines which of the 16 outputs on the two analog output circuits to change, and `value` is the voltage to set.
- `event(time,[list of events])`: This creates an event that is a concatenation of all the listed events, with all of them starting at the given time (we use milliseconds for the units). It can be used on a single event (such as the ones returned by the previous two functions) to assign a nonzero time for the event. It can also be used with a 0 value of time to combine several events at various times together, without affecting when those events occur.

We have a few other event functions in our library, including the ability to ramp the output channels of the analog output devices.

Here is the sample (pseudo-) code. `event_list` is a list of the events we want to happen. This also assumes that there is a master box which is not mentioned in the code:

These initial event definitions would normally be in a separate library that described how the boxes were hooked up to various parts of our experiment.
The main laser beam is turned on and off by some device that is controlled by a digital switch that takes a TTL input. This TTL input is hooked to the zeroth bit of a digital output box:

```
main_laser_on=
    digital_action("digital1", [0], [])
main_laser_off=
    digital_action("digital1", [], [0])
```

The secondary laser has a similar setup, but a separate TTL signal:

```
second_laser_on=
    digital_action("digital1", [1], [])
second_laser_off=
    digital_action("digital1", [], [1])
```

Each laser's intensity is also controlled by some analog input voltage, which is set by two channels on an analog output box. Here, we have two settings for each laser:

```
main_laser_low=
    analog_action_set("analog1", 0, 2)
main_laser_high=
    analog_action_set("analog1", 0, 5)
second_laser_low=
    analog_action_set("analog1", 1, 2)
second_laser_high=
    analog_action_set("analog1", 1, 5)
```

Events at negative times determine how the boxes are initialized before the start of the streaming. Initialization does not happen at fixed times. We want the lasers to all be off:

```
event_list=event(-1,main_laser_off,
                    second_laser_off,
                    main_laser_low,
                    second_laser_low)
```

Turn the lasers on at the start of the experiment. Defining a separate event with a useful name and then adding it to event_list *is a little more readable:*

```
TIME=0
start_exp=event(TIME,main_laser_on,
                    second_laser_on,
                    main_laser_low,
                    second_laser_low)
event_list=event(0,event_list,
                    start_exp)
```

Turn the second laser off and increase the power of the main laser at some later time:

```
TIME+=100
switch_to_main_laser=event(TIME,
                    second_laser_off,
                    main_laser_on,
                    main_laser_high)
event_list=event(0,event_list,
                    switch_to_main_laser)
```

Finally, turn all the lasers off even later:

```
TIME+=500
lasers_off=event(TIME,main_laser_off,
                    second_laser_off)
event_list=event(0,event_list,
                    lasers_off)
```

Program the boxes:
```
program_boxes(event_list)
```
Start the streaming:
```
trigger()
```
Wait for the boxes to finish (sleep_for_boxes() takes milliseconds, which is what TIME is already in:
```
sleep_for_boxes(TIME)
```
Print some warning if the boxes do not respond.

The library, along with a script that implements the above code in PERL, is available on our website.[9]

IX. FUTURE DEVELOPMENT

This is an ongoing project. We have the telnet thread properly functioning for the direct digital synthesis circuits,
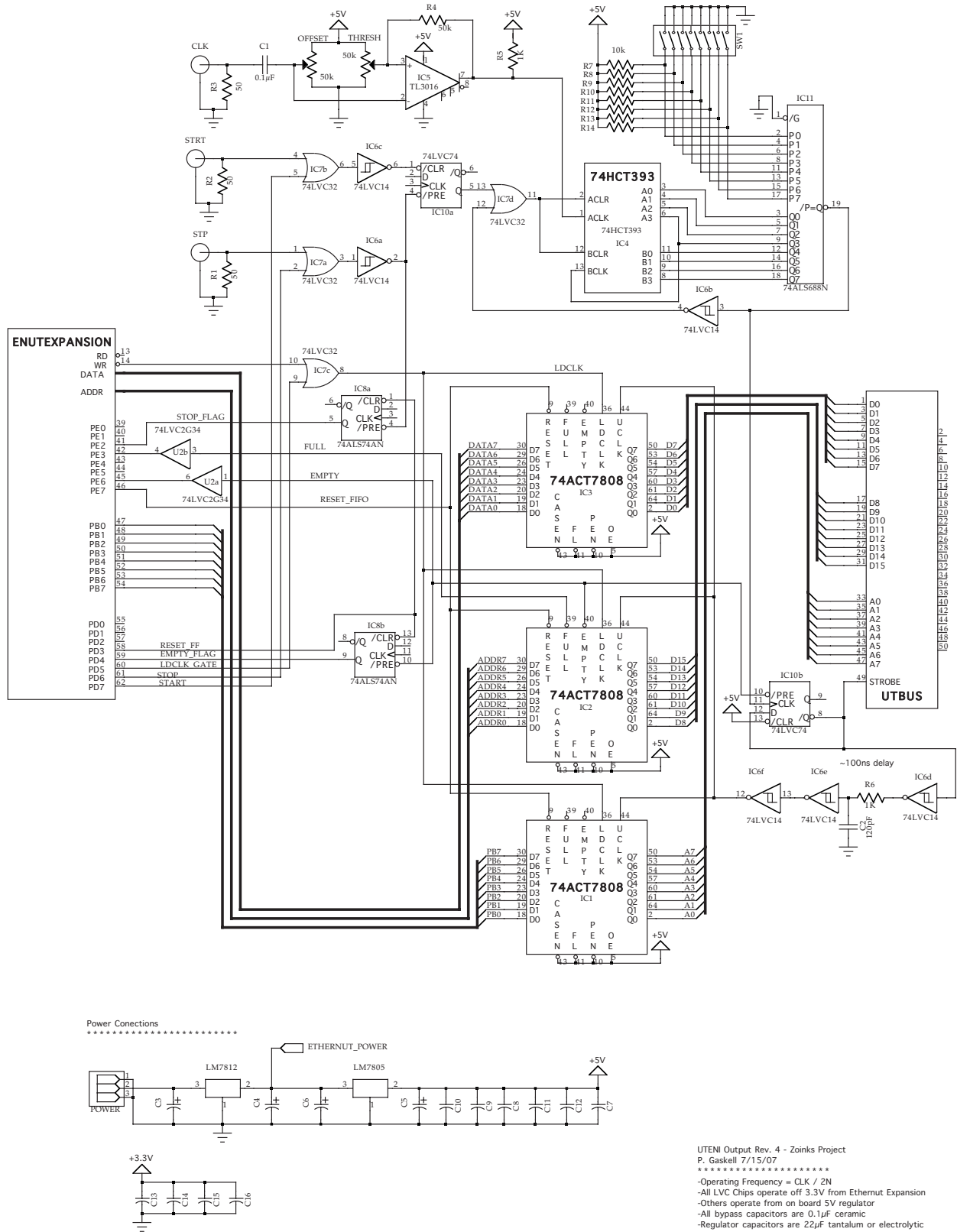
FIG. 5. Schematic for the output interface circuit.

but still need to implement the data streamer for those circuits. We also are also working on an input version of the interface circuit, with associated ETHERNUT software, to interface with an analog input instrument circuit that we are concurrently developing.

The input interface circuit is very similar to the output interface circuit, except that the ETHERNUT only loads data onto one of the three FIFOs (the address FIFO). The other two FIFOs will store data from the instrument circuits, to be read by the ETHERNUT. We are concurrently developing our only input instrument circuit, which is an analog input circuit.

The current analog input instrument circuit design utilizes four high quality, 16-bit, 250 kHz analog to digital converters capable of measuring ±10 V. The inputs are fully differential and the input gain can be chosen by soldering a single resistor onto the circuit. The 8 address bits will select a particular input on a particular circuit to perform a conversion, which will get loaded into the FIFO buffers on the input interface circuit. The ETHERNUT, at its convenience, will load these into memory. This setup should allow us to either monitor a few channels nearly continuously, as a relatively slow but cheap oscilloscope, or take a few discrete measurements at specific critical times during an experimental run.

X. KNOWN ISSUES

We know of one significant issue on the ETHERNUT 2.1 boards. If the ETHERNUTs receive data over the network faster than they read it, the network interface hangs, and the only way we know to restore network connectivity is to reboot the ETHERNUT. This is usually not a problem but two situations can trigger the issue. First, high network loads can cause the problem. The processor on the ETHERNUT 2.1 is not fast enough to handle the full network speed, so, for example, a prolonged ping flood crashes the network interface. The second time is when the ETHERNUT is otherwise busy. This happens when the ETHERNUT is in streaming mode. During this time, the processor is instructed to ignore the network interface (and all other interrupts), and so any nonzero data rate is enough to cause a network problem eventually, but it takes more than a few pings to do it.

We have not attempted to track the problem down or find a solution other than not using the network when the ETHERNUTs are in streaming mode. We do suspect that the issue is a buffer overflow on the network processor on the ETHERNUT board (which is separate from the main processor). We think when data arrive on the network, the network processor stores it and signals the main processor that there is data. The main processor then reads the data. If the main processor does not read the data, the buffer on the network processor slowly fills up until it either gets read by the main processor, or it fills completely up. We think that once the buffer fills up, the network interface hangs. If this is the case, then there might be a way to restore the network processor in software, by sending some reset command. However, for now, we reboot the ETHERNUT if the problem occurs. This currently needs to be detected manually, as the ETHERNUT software does not probe to see if the network interface is working.

APPENDIX: SCHEMATICS

The schematic for the output interface circuit is given in Fig. 5, and includes the three external inputs, clock divider, FIFOs, ETHERNUT interface bus, power line for the ETHERNUT, and instrument circuit interface bus.

[1] D. L. Whitaker, A. Sharma, and J. M. Brown, Rev. Sci. Instrum. **77**, 126101 (2006).
[2] A. Combo, A. J. N. Batista, J. Sousa, and C. A. F. Varandas, Rev. Sci. Instrum. **74**, 1815 (2003).
[3] D. S. Hall, Rev. Sci. Instrum. **75**, 562 (2004).
[4] R. M. Gao and K. W. Madison, ETHERNUT-GPIB interface manual, 2005, URL: http://www.physics.ubc.ca/~qdg/publications/InternalReports/GPIBE NUTMan.pdf.
[5] J. J. Thorn, E. A. Schoene, T. Li, and D. A. Steck, Phys. Rev. Lett. **100**, 240407 (2008).
[6] J. J. Thorn, E. A. Schoene, T. Li, and D. A. Steck, Phys. Rev. A **79**, 063402 (2009).
[7] Egnite GmbH, URL: http://www.egnite.de/, 2009.
[8] T. Meyrath and F. Schreck, A laboratory control system for cold atom experiments: Hardware and software, URL: http://iqoqi006.uibk.ac.at/users/c704250/, 2009.
[9] P. E. Gaskell, URL: http://atomoptics.uoregon.edu/zoinks, 2009.