A Sample Data Acquisition and Control Application Using RTLab

Calin A. Culianu and David J. Christini
Weill Medical College of Cornell University
520 E. 70th St., New York, NY 10021
calin@rtlab.org, dchristi@med.cornell.edu

Abstract

RTLab is a software project which provides a cross-variant (RTAI and RTLinux) conceptual framework for biomedical experiment interface. The intent is to provide abstract services on top of RTAI/RTLinux and COMEDI for the rapid development of a certain class of control and measurement applications typical in the biomedical field. Applications that follow a periodic acquire-analyze-control pattern can be quickly written using a plugin-style architecture. A simple control application, complete with code, will be illustrated in this paper.

1 Introduction

Real-time computing tasks, such as periodic tasks for which timing must be guaranteed, are ubiquitous in biomedical research. As an example of a real-time application, consider a feedback control system where the amplitude of an electrical stimulus (perhaps stimulating a section of biological tissue) is to be recomputed repeatedly as a function of the most recent analog input sample acquired from a biological preparation. If an internal multifunction board collects data at a sampling rate of 1kHz (a rate which is often sufficient for biological experiments), the computer must retrieve each input scan from the multifunction board, process that scan to determine the stimulus amplitude, and send the appropriate signal to the stimulator, all in less than 1 ms. If the application fails to accomplish this in time, the consequences may be unacceptable[1].

It might appear that failure to maintain the task rate would be a predictable (and possibly preventable) function of hardware speed, sampling rate, and/or algorithmic complexity. However, if the computer is controlled by a multitasking operating system such as a standard Linux kernel, it is more likely that real-time accuracy will be hindered by the operating system itself; Linux's standard scheduler has scheduling policies for tasks which are are inherently non-deterministic and make no guarantees with respect to time, making real-time processing unreliable (even for relatively slow and computationally simple tasks). Unpredictable events outside of the control of a process (sometimes even events originating outside the computer itself – such as interrupts generated by incoming network packets) can potentially interfere with even the most carefully crafted program[5].

In order to circumvent these shortcomings, two particular projects have sprung up to add real-time capability and features to Linux – RTAI[3] and RTLinux[5]. By running the Linux kernel itself as an idle task and turning off unneeded interrupts during the execution of a real-time task, realtime scheduling guarantees can be achieved.

While the quality of the software provided by both projects is high, from the point of view of the typical biomedical researcher, they still provide intimidatingly low-level abstractions and somewhat incompatible software APIs to their services. The function calls and data types used by RTLinux follow a strict POSIX pthreads interface. While RTAI offers some POSIX support, its pthreads implementation is slightly incomplete and not the preferred method of working with RTAI. In fact it is not possible to get the most out of RTAI through pthreads – one must instead use the alternate rt_*() series of functions[3].

In addition, neither RTAI's nor RTLinux's APIs address the specific needs of the biomedical experimenter – nor should they, as they are non-application specific and follow a "mechanism over policy" philos-

ophy. However, in a specific application domain such as biomedical research, certain higher-level concepts are useful to developers of real-time control applications. For instance, it might be useful for a program running an experiment to be able to easily track the frequency of a periodic (or aperiodic) signal. Rather than write the rather lengthy code for analyzing the signal and computing a running frequency, it is more convenient to instead have a signal monitor service with features similar to that of a digital oscilloscope. The experimenter then could simply ask the monitor to start monitoring a channel, then later ask the monitor for a status update on the current frequency. Several lines of intricately low-level code are now reduced to two high-level calls to an abstract service.

In order to address these experiment issues, the RTLab project was started in July of 2001 to provide a software framework for real-time Linux-based data acquisition and control for biomedical experiments. The software project uses existing Realtime Linux variants (RTLinux and RTAI) and COMEDI[4] drivers as a core. On top of these it adds some higher-level facilities useful in building a biomedical control application. The intended users of the code and services provided by RTLab are to be application programmers and researchers in the biomedical field who are interested in developing their own software-based control algorithms.

RTLab offers concepts and services such as waveform generation, waveform analysis, electrical pacing, stimulation and hardware/driver neutrality. These concepts are useful to experimenters working with real-time control of biological systems. RTLab offers an API that works on top of core facilities in either RTLinux or RTAI and shields researchers from the details of COMEDI – allowing him or her to focus on the algorithm they are trying to implement and the experiment they are trying to run rather than details of the platform they are using. [Details such as the exact gain setting on their analog output board needed to achieve 250mV output[4] or whether their RTIME value in RTAI[3] is in nanos or clock ticks, etc.]. What's more, RTLab allows one to easily substitute RTLinux for RTAI and viceversa, since its entire API is an abstraction on top of RTLinux/RTAI and COMEDI.

2 Architectural Overview

RTLab consists of two parts. The kernel-side rtlab.o module and user-side C++ classes.

2.1 Kernel

rtlab.o is a kernel module which is responsible for all real-time operation. It does things such as all low-level interaction with RTAI/RTLinux and COMEDI, and provides some basic services useful to a scientific experiment (such as waveform generation, waveform analysis, stimulation, etc.). rtlab.o communicates to userspace via the standard shared memory/RT-FIFO techniques.

2.2 User

This is Qt-based C++ code responsible for all user inteface interactions (such as would be done by an operator using a dialog box to tweak experiment parameters) and all visualization of data via graphs.

2.3 Kernel vs. Userspace

As most seasoned RTAI/RTLinux developers will tell you, your real-time control application is usually split into two parts: the critical pieces that *must* meet real-time deadlines, and the non-critical parts that have no such constraints[6].

Examples of critical parts of a real-time application include code that must analyze incoming samples and make a control decision within a limited time window. For example consider the following C function, which you might find inside the kernel module of an RTLab-based application:

Listing 2.3 - Example real-time control decision

```
/* Makes a control (electrical stimulation)
   decision based on incoming_voltage */
void
make_control_decision
  (
    double incoming_voltage
  )
{
  struct rtlab_stim_params p =
    {
      off_voltage: 0,
      num_per_train: 0,
      num_trains: 1
    };

  /* is it less than 250 mV? */
  if (incoming_voltage < 0.250) {
    p.on_voltage = 3.3; /* 3.3 V stim */
    p.duration_ms = 1; /* for 1ms     */
```

```
    p.when_ms = 3;    /* 3ms from now */
  } else {
    p.on_voltage = 1.3; /* 1.3 V stim */
    p.duration_ms = 2; /* for 2ms     */
    p.when_ms = 1;    /* 1ms from now */
  }

  /* schedule the stim. */
  rtlab_stimulate(my_stimulator,
                  &p);
}
```

The above example sends out two possibly different stimulation pulses depending on the incoming sample. If the input sample has amplitude less than 250mV, it sends out a 3.3V stimulation pulse with duration 1 ms and schedules it to go out 3 ms in the future. Any sample with amplitude greater than 250mV produces a stimulation pulse of 1.3V which lasts 2ms and is scheduled to occur 1 ms in the future.

The above example is code that would be time-critical. If this code were used in a context where failure to meet time deadlines were unacceptable, then we would say this code is hard real-time critical. Such code should always run under hard real-time priority.

RTLab (like many RTAI/RTLinux based programs) always runs real-time critical code in the kernel as a real-time kernel module[5]. The kernel module rtlab.o provides services (exported functions and other goodies) to real-time applications. Application developers are expected to write their own kernel modules that link against and make calls to rtlab.o. The above code example, for instance, would exist in its own kernel module (along with the rest of the real-time critical code specific to that application) but would use rtlab.o's exported symbols for services. The call to `rtlab_stimulate()` is a function call made into rtlab.o.

Thus, rtlab.o provides the kernel-side hard real-time services for an experiment application. These services include the following:

1. Data Acquisition and Measurement: allows client code to acquire samples from analog input channels synchronously and at a fixed rate. Clients are notified synchronously via function callbacks as each scan is completed.

2. Waveform analysis (used in conjunction with 1): allows client code to instruct rtlab.o to monitor a particular channel and analyze the waveform on that channel. Properties such as period and amplitude are made available to client code.

3. Waveform generation (control and stimulation): enables client code to generate specified waveform types on an AO channel as well as schedule hard-realtime stimulation pulses. Waveforms are parametrically specified. Such features are very critical to biomedical experimentation.

   The intent here is to replace the hardware pacer/stimulator one would normally use in a cardiology experiment, for instance. In addition to all of the functionality of the hardware version (at a fraction of the cost) this approach would give the user the additional ability to design feedback-adapted stimulation schemes[7].

4. Hardware abstraction: Focus is on 'cooked' values such as sample voltage rather than sample value (a double versus an `lsampl_t` in COMEDI parlance), experiment time in milliseconds and microseconds rather than 'clock ticks' or 'jiffies', etc.

2.4 User Space Facilities

All user space UI code for RTLab is built around the Qt Widget set. Additional support classes are provided for utility tasks such as data serialization to disk and communication with the realtime process. The intent is to create useful basic classes for scientists who want to build a UI to control their experiments.

User space services offered by the RTLab class hierarchy are as follows:

1. Data graphing/visualization: basic data visualization facilities are accomplished via 2D graphing classes.

2. Realtime Process control: variant-neutral treatment of shared memory buffers (mbuff.o and rtai_shm.o) and RT-FIFOs. Utility classes provide functional wrappers to read() and write() system calls on FIFOs which integrate well with Qt's own asynchronous event system. This frees scientists from the tast of writing efficient code to read from FIFOs and provides a natural integration between Qt and Realtime Process IPC mechanisms.

3. Data serialization to disk: The .NDS data file format is a versatile and extensible file format

for storing acquired analog input data. All floats/doubles/ints are stored in network byte order, which is the preferred method of effecting CPU architecture neutrality in binary data.

This file format can be extended to add features and to store additional meta-data without rendering files unopenable by existing tools – tools which might expect to read an 'older' format. This is accomplished by way of a named segmenting scheme for storing binary and ASCII data. Existing tools encountering an unknown segment can simply disregard it, while still processing the standard analog data segment present in every data file.

This type of compatibility is essential to scientific researchers who may wish to develop their own extentions to the file format for adding application-specific meta-data. For example, cardiac researchers may need to annotate a heart ECG in order to flag anomalous beats. This type of custom meta-data can be added without 'breaking' the file format.

4. Plugin architecture: allows scientists to extend the functionality of existing applications built on RTLab via a plugin mechanism. For example, the DAQSystem application that comes with RTLab can be easily extended for control applications (as we will show below). It is compiled as position-independent, dynamic-exporting code (-fPIC and -export-dynamic in gcc) that exports symbols to other objects. Plugins are written as shared object files (.so) that follow a specific interface. Plugins can be loaded on demand, adding and removing functionality from DAQSystem as needed.

3 Using RTLab to Perform Control Experiments

Using RTLab for the purposes of a specific control experiment requires two discrete modules to be written:

1. RTLab-based kernel module: A kernel module that plugs into rtlab.o and registers itself with hooks inside rtlab.o. This module is responsible for driving the experiment in real time. This code is, in essence, the brains of the experiment. We will focus on this because it typically contains all essential experiment logic.

2. User-space shared-object plugin: A user space module (.so) file that gets loaded by the DAQSystem user program and which should provide a GUI for controlling experiment variables and which is responsible for saving experimental data to disk.

3.1 Inside rtlab.o

The fundamental engine behind rtlab.o is a single periodic loop. Since RTLab is an acquisition-driven system, this loop runs at the sampling frequency of the acquisition. All resultant features and services provided by rtlab.o are based upon hooks inside this loop that run after a particular scan has been acquired.

Listing 3.1 - pseudocode for main rtlab.o loop

```
void main_rtlab_loop()
{
  MultiSampleStruct *m;

  while (!end_rt_task) {
    m = grab_scans_off_boards();
    for_each(callback,
             /* in */ callbacks) {

      callback(m);

      /* the callback above needs to be
         'lean' as if it takes too long
         to execute our realtime task
         misses its next period!! */

    }

    /* generate waveforms for
       stimulation or pacing -- A
       complicated stimulation
       pattern or waveform is
       implemented as a series of
       atomic voltage output commands
       (comedi_data_write()) queued up
       and executed at precise times */
    do_analog_output();

    /* this is the core of the monitor
       service */
    do_analog_input_analysis(m);

    /* sleep the realtime task until
       the next period, which is a
       function of our data acquisition
       sampling rate */
    wait_next_period();
  }
}
```

The hooks are as follows:

1. Scan-related callback functions: Modules can attach themselves to rtlab.o by registering callback functions to be executed once per scan. These are registered via the `rtp_register_function()` function. The callback functions are passed a pointer to a MultiSampleStruct which is a struct that encapsulates a single scan.

   This mechanism allows modules to perform computations on an incoming scan and possibly change their own internal state and/or take actions based on incoming scans.

   Possible actions including registering a stimulus with the stimulation engine or altering stimulation frequency or amplitude.

   In the sample in listing 3.1, this corresponds to the line that reads:

   ```
   callback(m);
   ```

2. Signal monitoring callbacks: Similar to the scan-related callbacks above, except the registered functions are executed once a particular signal has completed an analysis cycle and the state of the signal variables has been updated.

   This mechanism allows an experimenter's code to change its internal state based on important parameters. (Such as a change in frequency, mean amplitude, etc.)

3. Waveform generation notification: Similar to signal monitoring callbacks, except that a user's callback function is executed with respect to the waveform generation facilities provided by rtlab.o. An example would be that perhaps a subscribed client of this code would want to be notified once a stimulation has been generated by the pacing service of rtlab.o. Essentially, client code gets notified of interesting events that have occurred with respect to analog output so that it can change its state accordingly.

4 Building A Sample Control Application

Appendix A illustrates the kernel code for a realtime control application that does the following:

1. Monitors an analog input channel's average voltage (V) and frequency (F) and updates its internal state variables accordingly.
2. Every time the monitored input channel above finishes a cycle, a stimulus is immediately sent down the wire via an analog output channel. The duration of this stimulus is a function of F, and its magnitude is a function of V. The time of the stimulus, F, and V are all recorded and sent to userspace via an RT-FIFO.

(See appendix A at the end of this document)

4.1 Listing walkthrough

Lines 22-28 Some useful kernel headers.

Line 30 The header to the rtlab.o kernel facilities.

Lines 31-47 Linux kernel module entry and exit functions.

Lines 48-51 Some stateful information that is used to determine the nature of our outgoing control signal. `(voltage_avg / 10)` determines the outgoing stimulation voltage, and one fifth the monitored frequency `(1000.0/frequency_hz)/5.0`.

Line 52 The opaque `rtlab_monitor` type which is used as a handle for signal monitoring. (see *Signal Monitoring Callbacks* above)

Lines 57-63 Handles to our realtime fifo and our analog output channel. All channels in RTLab are positive integers starting at 0. The first AI channel on the first comedi device is 0, the second channel is 1, etc.. all the way to the last comedi device.

All realtime FIFOs correspond to minor numbers of `/dev/rtf*` files in userspace.

Lines 83-120 Module initialization: Register this module with rtlab.o.

`rtp_register_function()` tells rtlab.o that a callback function is to be added to the list of callbacks that get executed once a full scan is complete. This happens once per rtlab.o engine loop iteration.

`rtp_find_free_rtf()` tells rtlab.o to create an RTF and allocate some space for it. A handle to the rtf is placed in the first parameter.

`rtp_reserve_ao_chan()` tells rtlab.o to atomically find and reserve the first free analog output channel. This function is intended to allow

more than one realtime experiment module to share analog output resources.

`rtlab_monitor_channel()` tells rtlab.o to begin monitoring a AI channel 0. In this case we are monitoring the channel's frequency (`RTLAB_MONITOR_FREQUENCY`). Other possible values include `RTLAB_MONITOR_AMPLITUDE` to keep track of the max amplitude per period. Once a period completes the callback we passed to this function (`monitored_values_changed`) gets executed.

Lines 138-145 Keep a current average of the incoming channel 0 voltage. This is an application-specific statistic and is recomputed once per scan, because this function is called after each scan completes.

Lines 147-158 Callback that gets executed whenever the rtlab.o monitor service notices that the monitored signal has updated values. This usually happens after the incoming periodic signal has finished a period. We ask the monitor service what the last period's frequency was via `rtlab_get_frequency()`.

Lines 160-177 Function that does some setup on a `struct rtlab_stim_params` to generate a stimulus in real time. This stimulus is sent to the wire when `rtlab_stimulate_ao_chan()` is called. The stimulus's amplitude, duration, and whether or not it is continuous or a oneshot are specified by various parameters in the `struct rtlab_stim_params`.

Lines 179-190 This function generates a human-readable string that gets put on the fifo by rtlab.o's wrapper to `rtf_put()`, `rtp_rtf_put()`.

5 Significance of this development project

As mentioned earlier, real-time experiment control systems are used in a wide range of biomedical research areas. Currently, most researchers use either custom-programmed or proprietary systems. Custom-programmed systems require a large time investment and are at risk of becoming unmaintained legacy code in the event that the main programmer leaves or graduates. Proprietary systems have a number of disadvantages. One is price. Perhaps most critically, users have no control over the future of their proprietary software: they are subject to the whims of the software company (or worse, an unknown entity if the company goes out of business) when it comes to future development, bug elimination, or even long-term support.

The increasing prominence of open-source software is an important trend that has many positive benefits for the scientific community. Open-source software is typically favorable due to its low cost and the high degree of modifiability and thus control over program behavior. Because of open-source licensing, no entity can ever own the software — its future is only dictated by the software's continued utility and the willingness of people to further its development. In fact, the parallels between the availability of source-code in open-source software and the open exchange of information typical of scientific advancement are more than just coincidental: science can benefit greatly from utilizing open-source software.

In short, an open-source real-time experiment control system could be of great value to biomedical scientists. Such a system would be an important research and education tool in a wide variety of research and teaching laboratories and would decrease the time required for system implementation, eliminate software licensing fees, and greatly reduce the likelihood of a system becoming outdated legacy code.

References

[1] Christini DJ, Stein KM, Markowitz SM, Lerman BB. A practical real-time computing system for biomedical experiment interface. *Annals of Biomedical Engineering* 1999;27:180–186.

[2] Bianchi E, Dozio L, Ghiringhelli GL, Mantegazza P. Complex Control Systems, Applications of DIAPM-RTAI at DIAPM, *Realtime Linux Workshop* Milan 2001;

[3] Mantegazza P, et al. DIAPM RTAI Programming Guide 1.0 *http://www.rtai.org* ;

[4] Schleef D, Hess F. Comedi Documentation *http://www.comedi.org/comedi* ;

[5] Yodaiken V, Barabanov M. A Real-Time Linux *http://www.fsmlabs.org* 1996;

[6] Ripoll I. RTLinux versus RTAI *http://bernia.disca.upv.es/rtportal/* 2002;

[7] Hall K, Christini DJ, Tremblay M, Collins JJ, Glass L, Billette J. Dynamic control of cardiac alternans. *Physical Review Letters* 1997; 78:4518–4521.

APPENDIX

A Listing of our Example Control Application

```
1  /*
2   * This file is part of the RT-Linux Multichannel Data Acquisition System
3   *
4   * Copyright (C) 2002 Calin Culianu
5   *
6   * This program is free software; you can redistribute it and/or
7   * modify it under the terms of the GNU General Public License
8   * as published by the Free Software Foundation; either version 2
9   * of the License, or (at your option) any later version.
10  *
11  * This program is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program (see COPYRIGHT file); if not, write to the
18  * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
19  * Boston, MA 02111-1307, USA, or go to their website at
20  * http://www.gnu.org.
21  */
22 #include <linux/module.h>
23 #include <linux/kernel.h>
24 #include <linux/init.h>
25 #include <linux/version.h>
26 #include <linux/types.h>
27 #include <linux/errno.h>
28 #include <linux/string.h>
29
30 #include "rtlab.h"
31
32 #ifdef MODULE_LICENSE
33 MODULE_LICENSE("GPL");
34 #endif
35
36 #define MODULE_NAME "Example RTLab Module"
37
38 MODULE_AUTHOR("David J. Christini, PhD and Calin A. Culianu");
39 MODULE_DESCRIPTION(MODULE_NAME ": A Real-Time stimulation and control "
40                    "add-on for daq system and rtlab.o.\n$Id: example_module.c,v 1.1 2002/11/04 21:28:43 calin Exp $
41
42 int  mod_init(void);      /* Module entry */
43 void mod_cleanup(void);   /* Module cleanup */
44
45 module_init(mod_init);    /* as per Linux Kernel Module interface */
46 module_exit(mod_cleanup); /* as per Linux Kernel Module interface */
47
48 #define N_VOLTAGES 10 /* the number of voltages to average */
49 static double voltage_avg = 0.0;
50 static int n_voltages = 0; /* the number of voltages in our average */
51 static double frequency_hz = 0.0; /* our ongoing frequency variable F */
52 static struct rtlab_monitor *monitor = 0; /* the opaque handle given to us by
53                                              rtlab.o */
```

```
54
55  /* just a struct holding our string message which gets copied to
56     user space via a realtime fifo */
57  struct fifo_msg {
58    char message[1024];
59  };
60
61  #define FIFO_SZ (sizeof(struct fifo_msg) * 100)
62  static int realtime_fifo = -1; /* handle to our outgoing RTF */
63  static int ao_chan = -1; /* the analog output channel handle reserved
64                              for us by rtlab.o */
65
66  /* this is called once per scan to recompute voltage_avg */
67  static void recompute_voltage_average(MultiSampleStruct *);
68
69  /* this is called automatically by the signal monitor as a notification
70     that the frequency of the monitored channel changed, or that one
71     period elapsed in the monitored channel.   */
72  static void monitored_values_changed(void);
73
74  /* internally called by monitored_values_changed() to send a stimulus
75     down the wire using rtlab_stimulate_ao_chan() */
76  static void stimulate(void); /* if called, a new stim is started */
77
78  /* internal function that writes a struct fifo_msg out the rt-fifo
79     so that a monitoring user process can serialize experiment data to disk */
80  static void out_to_fifo(void);
81
82
83  int mod_init (void)
84  {
85    int retval = 0;
86
87    if (
88        /* register our per-scan callback */
89        (retval = rtp_register_function(recompute_voltage_average))
90        /* find a non-reserved rtf (realtime fifo) and put its value
91           in realtime_fifo */
92         || (retval = rtp_find_free_rtf(&realtime_fifo, FIFO_SZ))
93        /* find a non-reserved analog output channel for control
94           and put its value in ao_chan */
95         || (retval = rtp_reserve_ao_chan(&ao_chan))
96         || (retval = rtlab_monitor_channel(
97                                           /* opaque 'handle' */
98                                           &monitor,
99
100                                          /* monitor the first AI chan
101                                             on the first comedi  board */
102                                          0,
103
104                                          /* tell the monitor we are
105                                             interested in frequency only */
106                                          RTLAB_MONITOR_FREQUENCY,
107
108                                          /* our notification callback
109                                             function */
110                                          monitored_values_changed))
111
112        /* activate our callback function once all of the above is ok */
113         || (retval = rtp_activate_function(recompute_voltage_average))
```

```
114       )
115      /* on failure call our module cleanup function to release
116         any partial resources */
117      mod_cleanup();
118
119    return retval;
120  }
121
122  void mod_cleanup (void)
123  {
124    rtp_deactivate_function(recompute_voltage_average);
125
126    /* free our channel monitor */
127    if (monitor)  rtlab_deactivate_monitor(monitor);
128
129    /* free up our reserved rt-fifo */
130    if (realtime_fifo >= 0)  rtp_rtf_destroy(realtime_fifo);
131
132    /* indicate that now this channel is free */
133    if (ao_chan >= 0)  rtp_free_ao_channel(ao_chan);
134
135    rtp_unregister_function(recompute_voltage_average);
136  }
137
138  void recompute_voltage_average(MultiSampleStruct *)
139  {
140    if (n_voltages < N_VOLTAGES)
141      n_voltages++;
142
143
144    avg_voltage +=  m->samples[0] / n_voltages;
145  }
146
147  void monitored_values_changed(void)
148  {
149    /* abort early if our voltage_avg value isn't yet complete
150       (recomput_voltage_average needs to be called N_VOLTAGES times
151       in order for our voltage_avg value to be considered value) */
152    if (n_voltages < N_VOLTAGES) return;
153
154    frequency_hz = rtlab_get_frequency(monitor);
155
156    stimulate();
157    out_to_fifo();
158  }
159
160  void stimulate(void)
161  {
162    struct rtlab_stim_params rsp = EMPTY_STIM_PARAMS;
163
164    /* stimulate with an amplitude that is 1/10th the ongoing computed
165       average voltage */
166    rsp.on_voltage = voltage_avg / 10.0;
167    rsp.off_voltage = 0;
168    rsp.when_ms = 0; /* send the stim right now (0 ms from now) */
169    /* the duration in ms is one tenth the current computed frequency */
170    rsp.duration_ms = (1000.0 / frequency_hz) / 10.0;
171    /* only 1 stim per 'stim train' */
172    rsp.num_per_train = 0;
173    /* only 1 stim cycle */
```

```
174     rsp.num_trains = 1;
175
176     rtlab_stimulate_ao_chan(ao_chan, &rsp);
177   }
178
179   static void out_to_fifo(void)
180   {
181     struct fifo_msg fm;
182
183     sprintf(fm.message,
184             "Scan index: %s   Avg: %s V    Freq: %s HZ",
185             rtlab_scan_index_str,
186             rtlab_double_to_string(voltage_avg),
187             rtlab_double_to_string(frequency_hz));
188
189     rtp_rtf_put(realtime_fifo, &fm, sizeof(fm));
190   }
```