The Brody, Mainen, and Zador Lab

RTLinux Finite State Machine and Sound Triggering System

Quick Overview Guide

Calin A. Culianu <calin@rtlab.org>

Oct. 13, 2005

Overview

What is RTLinux? It is a modification to the Linux Kernel (the core of the operating system that interacts with hardware) to allow for "Hard Real-time" task scheduling. For more information, papers, etc. go to the RTLinux website: http://www.fsmlabs.org.

How are RTLinux 'programs' written? Traditionally RTLinux programs are written as LKM, or Linux Kernel Modules in C (although recent developments allow for RTLinux programs to run as regular userspace processes, there are still some performance reasons to stick to Kernel modules).

Kernel Modules

Kernel modules are analogous to Windows drivers. They are compiled 'object files' that can be loaded and into the kernel's address space at *runtime* and executed. For more information on kernel modules one should google for 'Linux Kernel Modules' and there is a plethora of literature on the subject.. :)

Note: Since modules are programs that run in the kernel, they typically produce no user-visible output to the screen. However, they may *print messages* (including error messages if they fail to load). It is important to review the output of kernel modules (especially if you are loading and unloading them) using the '***dmesg***' command.

Commands for operating upon existing Kernel Modules: Below is a short list of programs or commands that should be run as root to work with Linux Kernel Modules.

`lsmod` – lists the currently loaded modules. Sample output from lsmod:

```
Module                  Size  Used by    Tainted: P
rtl_sched              27680   0 (unused)
```

```
mbuff                   5996   0 (unused)
rtl                    17936   0 (unused)
```

The leftmost column is the MODULE NAME. It is basically the name of the module file without the filename extention. On Kernel 2.4 modules all end in '.o'. On 2.6 the module extention was renamed to '.ko'.

`modprobe` – Loads a module into the kernel from the *system module directory* which is usually /lib/modules/KERNELVERSION/. Modprobe automatically resolves module dependencies (some modules depend on other modules to load because they may 'call into' those other modules.

`insmod` – Loads a module file into the kernel from **outside** the *system module directory*. Use this to load individual module files that may or may not be 'properly' installed yet. Insmod does not resolve dependencies (as it operated on 'unintalled' modules and thus there is no dependency information for them, presumably).

`rmmod` – Unloads a loaded module (one visible using the lsmod command).

`modinfo` –Shows you the list of parameters that a particular module takes, along with descriptions for those parameters (if any).

`dmesg` –Shows you the Kernel log. While this isn't a module-specific command, modules typically print messages to the Kernel log and it is important to read the kernel log when working with modules (loading, unloading, getting errors, etc).

Module Descriptions and Parameters

Kernel modules can take (optional) *parameters* which are either integers or strings. They typically are used to configure the module's behavior or tell the module something about your hardware, etc. Use the *modinfo* command (see above) to find out what parameters (if any) a module takes.

Example: `insmod ./RatExpFSM.o sampling_rate=12000 num_in_chans=8 num_cont_chans=5 num_trig_chans=5`

Would load RatExpFSM.o kernel module giving it a sampling_rate parameter (see below) of 12000, num_in_chans of 8, etc.

If you fail to give a module any parameters, don't worry – usually the defaults work well enough.

The RTLinux programs I developed for the Finite State Machine and the Sound Triggering all take parameters and it is important (particularly for the finite state machine) to know what these parameters are as they have a strong influence on the operation of the FSM (see below).

The kernel modules for the FSM and sound triggering are broken up into three separate realtime kernel modules:

RatExpFSM.o

This lives in the CSHL_FSM/ source directory (ask Carlos about his CVS repository if interested in retreiving this code). It implements the Realtime Finite State Machine. It is analogous to the pervious RP-2 box that Lung-Hao programmed to implement the Finite State Machine. This module runs at a specified *sampling rate* or task rate.

1. It basically wakes up every task period
2. Decides if new inputs have arrived on the DIO lines
3. OR decides if a timeout has expired (timeouts are part of the State Matrix)
4. If either of the above are true, it performs a state transition to a new state, recording the transition in the "event history"
5. It possibly issues digital outputs on DIO lines.

Parameters for RatExpFSM.o

parm: **minordev** int, description "The minor number of the comedi device to use."

parm: **sampling_rate** int, description "The sampling rate. Defaults to 6000."

parm: t**rigger_ms** int, description "The amount of time, in milliseconds, to sustain trigger outputs. Defaults to 1."

parm: **debug** int, description "If true, print extra (cryptic) debugging output. Defaults to 0."

parm: **avoid_redundant_writes** int, description "If true, do not do comedi DIO writes during scans that generated no new output. Defaults to 0 (false)."

parm: **num_in_chans** int, description "The number of channels to use as DIO input. Set this

appropriately for devices that require a block of, say, 8 channels to be input channels together, even if you only need, say 3 of these for actual inputs. These channels will be numbered 0-(num_in_chans- 1). Defaults to 8 (appropriate for most devices)."

parm: **num_cont_chans** int, description "The number of digital output channels to use for 'continuous' outputs. These channels are numbered [num_in_chans,(num_in_chans+num_cont_chans)]. Defaults to 11."

parm: **num_trig_chans** int, description "The number of digital output channels to use for 'trigger' outputs. Note that these are separate from the vtriggers. These channels will be numbered [(num_in_chans+num_cont_chans), (num_in_chans+num_cont_chans+num_trig_chans)]. Defaults to 5."

parm: **ready_for_trial_jumpstate** int, description "The state number to use as the 'ready for trial' jumpstate. If you know what this means, great. If not, leave it be. :) Defaults to 35."

parm: **num_vtrigs** int, description "The number of 'virtual' triggers to use. These triggers are not output using DIO but are intended to be used with the Lynx sound board, and are sent to LynxTrig-RT.o via the shared memory buffer "LynxVirtTrigShm". They are numbered from the number of total channels on (num_in_chans + num_cont_chans + num_trig_chans) onward. Defaults to 8."

Note: By far the most important parameters above are **num_in_chans**, **num_cont_chans**, and **num_trig_chans**. They affect how the DIO lines are used as well as which lines on the DAQ card correspond to what in the *State Matrix*.

LynxTWO-RT.o

This module lives in the LynxTWO-RT/ directory (as Carlos about getting the code from CVS if interested). It is a low-level driver module that directly talks to the Lynx22 sound board for playing triggered sounds. This module normally doesn't need to be touched or configured in any way as it 'just works' the way it is.

Parameters for LynxTWO-RT.o

parm: **debug** int, description "If set, produce very verbose (and slow) debug output. Not recommended unless you are a developer. :) Defaults to 0."

parm: **dma_size** int, description "The size in bytes of each of the DMA Buffers used for host DMA transfers. Tweak this to balance out latency versus CPU load versus underrun risk. The default is good enough for an RTOS-based driver. Default is (8*1024)."

parm: **do_preloads** int, description "If true, CHalWaveDMADevice->Start() does DMA preloads, which can take a few dozen microseconds to complete. Not recommended for realtime drivers. Defaults to (0)."

parm: **glimit** int, description "Low-level parameter to control sample threshold for generating DMA interrupts. When the number of samples in the board's CB buffer go below glimit, the DMA engine is told to continue copying samples from DMA buffers to the on-board CB. 0 means auto-set. Defaults to (0)."

parm: **dma** int, description "If nonzero, use DMA transfers to load sound

data onto the board. Using DMA causes fewer interrupts to be generated and reduces the load on the CPU for copying samples to the audio buffers, it also reduces the risk of buffer underruns. The drawback to DMA is that play commands have some fixed (non-jittered) latency that is a function of the sample rate (apparently the DMA engine isn't kicked-on by the board right away). Thus, DMA introduces some fixed latency before sounds play. In contrast, not using DMA has the effect of causing play commands to take effect nearly instantaneously, with no latency, at the expense of more time spent in interrupt-service-routines and more risk of underruns. DMA is recommended. Defaults to (1)."

parm: **auto_recalibrate** int, description "If nonzero, enable board auto-recalibration for when things like the sample clock rate change. Auto-recalibration is normally a good thing for playback accuracy, but it happens to cause the device to play silence for 1000 DWORDs when the sampling rate changes. If you change the sampling rate often right before playing a sound, then it is necessary to disable auto_recalibrate as the 1000 DWORDs of silence could overlap with actual sound playback. Defaults to (1)."

You should really never need to give LynxTWO-RT any parameters as the defaults work really well.

LynxTrig-RT.o

This module is responsible for triggering the Lynx22 sound board driver (LynxTWO-RT.o) to play sounds in realtime. It also stores a list of sounds in memory that get loaded via the Matlab interface (see Matlab section below).

Parameters for LynxTrig-RT.o

It has **two** modes of operation. Either it uses 'COMEDI' drivers to receive triggers from *actual* hardware such as a DAQ board, or, the default, is that it uses 'software' triggering to receive triggers from **RatExpFSM.o** via 'shared memory'.

To enable hardware triggering, it is necessary to specify the *comedi_triggers=1* module parameter. In hardware-triggered mode, two parameters that might be useful are also *first_trig_chan* and *num_trig_chans* which together specify the exact DIO channels to monitor for hardware triggers. Note that in software triggered mode these parameters are ignored.

parm: **minordev** int, description "The minor number of the comedi device to use."

parm: **sampling_rate** int, description "The sampling rate. Defaults to 10000. Ignored for software triggering as software triggering does not use a periodic task"

parm: **debug** int, description "If true, print extra (cryptic) debugging output. Defaults to 0."

parm: **first_trig_chan** int, description "The first DIO channel to use for board triggers. Note that this plus num_trig_chans determine exact lines we scan to get the binary trigger id. Defaults to 0."

parm: **num_trig_chans** int, description "The number of DIO channels from first_trig_chan to scan for triggers. Defaults to 0 (which means take all channels left until the last)."

parm: **comedi_triggers** int, description "If set, use COMEDI DIO device to do triggering. Otherwise use a SHM which contains a singe int for 'virtual' triggering from another RT module. The SHM name is "LynxVirtTrigShm" of size sizeof(int). Defaults to 0 (virtual)."

parm: **audio_memory** int, description "The number of MB to preallocate for audio buffers. Note that this should be a large value otherwise you will soon run out of realtime audio buffers when sending sounds from userspace! Defaults to 1/3 the system's physical RAM."

parm: **trig_on_event_chan** int, description "If non-negative, the DIO channel (not related to first_trig_chan or num_trig_chans at all!) to set 'trigger' for one cycle whenever an event occurs. This feature is intended to be used for debugging the timing of the play command. Defaults to -1 (off)."

Matlab Integration

The two functional components above that implement the FSM and the Sound Trigger are integrated with Matlab via TCP/IP sockets (over the local internet, basically). Two userspace processes, ***RatExpFSMServer*** and ***LynxTrigServer*** listen for TCP/IP connections on port 3333 and 3334 respectively. They use a simple text-oriented protocol to receive commands from the outside world and to pass parameters over to their respective kernel modules.

Thus, it is possible to remotely specify a state matrix for the RatExpFSM.o kernel module, or sounds to play based on trigger id for the LynxTrig-RT.o kernel module, among other things.

Matlab objects have been written to abstract out the specifics of communicating with these servers. They are called RTLSM (Realtime Linux State Machine) and RTLSoundMachine (Realtime Linux Sound Machine).

Matlab Command Reference

RTLSM (State Machine)

Below is a list of Matlab method calls for the RTLSM (state machine) object that communicates with the RTLinux state machine.

```
% sm = RTLSM(hostNameOfServer_String, portNumberOfServer_Numeric)
%      Constructs a new RTLSM instance and connects to the server using
%       hostname and port specified.
%
% sm = Initialize(sm)   This is equivalent to a reboot of the
%                StateMachine. It clears all variables, including
%                the state matrices, and initializes the
%                system. Initialize() does not start the
%                StateMachine running.
%
% sm = Halt(sm)  Stops the StateMachine: puts it in a Halt
%                state. In this state, no events have any
%                effect. Variables are not cleared, and can be read by
%                other programs.
%
%                A freshly Initialize()'d StateMachine is in the Halt
%                state. Halting an already halted StateMachine has
%                no effect.
%
% sm = Run(sm)   Restarts the StateMachine: events have an effect
%                again. After an Initialize(), Run() starts the
%                machine in state 0. After a Halt(), Run() restarts
%                the machine in whatever state is was halted. Note
%                that calling Run() before the state matrices have
%                been defined results in a Matlab error.
%
% sm = Set_StateMatrix(sm, Matrix state_matrix)    This command defines
the
%                state matrix that governs behavior during
%                trials. This state_matrix can have a maximum of 128
%                different rows (i.e., states), and must have 10
%                columns. (Note (1) that the part of the state matrix
that
%                is being run during intertrial intervals should
%                remain constant in between any two calls of
%                Initialize(); and (2) that Set_StateMatrix()
%                should only be called in between trials).
%
% sm = ForceTimeUp(sm)  Sends a signal to the state machine that is
%                equivalent to there being a TimeUp event in the
%                state that the machine is in when the
%                ForceTimeUp() signal is received. Note that due to
```

```
%                the asynchronous nature of the link between Matlab
%                and StateMachines, the StateMachine framework
%                itself provides no guarantees as to what state the
%                machine will be in when the ForceTuneUp() signal
%                is received.
%
% sm = ReadyToStartTrial(sm)   Signals to the StateMachine that it is ok
%                to start a new trial. After this routine is called,
%                the next time that the StateMachine reaches state 35,
%                it will jump to state 0, and a new trial starts.
%
%
% [] = Close(sm) Begone! Begone!

%
% ------- Machine Outputs to the World -------
%
% sm = BypassDout(sm, int d)   Sets the digital outputs to be whatever
the
%                state machine would indicate, bitwise or'd with
%                "d." To turn this off, call BypassDout(0).
%
% [] = Trigger(sm, int d) Bypass the control over output triggers, and
set
%                off the indicated trigger. (In the RM1 implementation
%                in use in August 2005, these triggers go to the analog
%                outputs of the RM1 box.)
%
%
%
% ------- Reading from the Machine -------
%
% [int nevents] = GetEventCounter(sm)   Get the number of events that
%                have occurred since the last Initialize()
%
% [EventList]   = GetEvents(sm, int StartEventNumber, int
EndEventNumber)
%                Gets a matrix in which each row corresponds to an
%                Event; the matrix should have
%                EndEventNumber-StartEventNumber+1 rows and three
%                columns. (If EndEventNumber is bigger than
%                GetEventCounter(), this produces an error).
%
%                Each of the rows in EventList should have three
%                columns: the first is the state that was current when
%                the event occurred; the second is the event that
%                occurred (1=Cin, 2=Cout, 4=Lin, 8=Lout, 16=Rin,
%                32=Rout, 64=Tup, 0=no detected event, e.g. when a
%                jump to state 0 is forced); the third is the time,
%                in seconds, at which the event occurred.
%
% [double time] = GetTime(sm)    Gets the time, in seconds, that has
%                elapsed since the last call to Initialize().
%
% [int r]       = IsRunning(sm)  return 1 if running, 0 if halted
```

```
%
%
% ----- Control issues ----------
%
% sm = FlushQueue(sm)   Some state machines (e.g., RM1s, RTLinux boxes)
%                 will be self-running; others need a periodic ping
%                 to operate on events in their incoming events
%                 queue. This function is used for the latter type
%                 of StateMachines. In self-running state machines,
%                 it is o.k. to deine this function to do nothing.
%
% [intvl_ms] = PreferredPollingInterval(sm)    For machines that
%                 require FlushQueue() calls, this function returns
%                 the preferred interval between calls. Note that
%                 there is no guarantee that this preferred interval
%                 will be respected. intvl_ms is in milliseconds.
%


% Carlos Brody wrote me on 15-Aug-05
```

RTLSoundMachine

```
% This directory contains the definitins of the RTLSoundMachine object.
%
% The current definition is intended to exactly replicate the
% capabilities of the TDT RPBoxes in use in August 2005. Future
% RTLSoundMachine definitions will expand and modify this.
%
%
% Methods:
%
% sm = RTLSoundMachine('host', port)
%                 Construct a new RTLSoundMachine handle.
%                 The host and port that the sound server is listening on
%                 Defaults to 'localhost', 3334.
%
% sm = Initialize(sm)
%                 This is equivalent to a reboot of the
%                 Sound Server. It clears all variables, including
%                 the sound files, and initializes the system.
%
%                 It is not necessary to call this unless you want to clear
%                 things and start with a clean slate.
%
%                 Note that multiple sm objects could potentially point to the
%                 same real sound server so be careful when re-initializing
%                 the sound server!
%
%
% sm = SetSampleRate(sm, srate)
%                 Set the sample rate for future calls to LoadSound()
%                 Note that changing the sampling rate only has an effect
%                 on future soundfiles loaded via LoadSound().  Sound files
%                 that were already loaded will *not* be updated to use
```

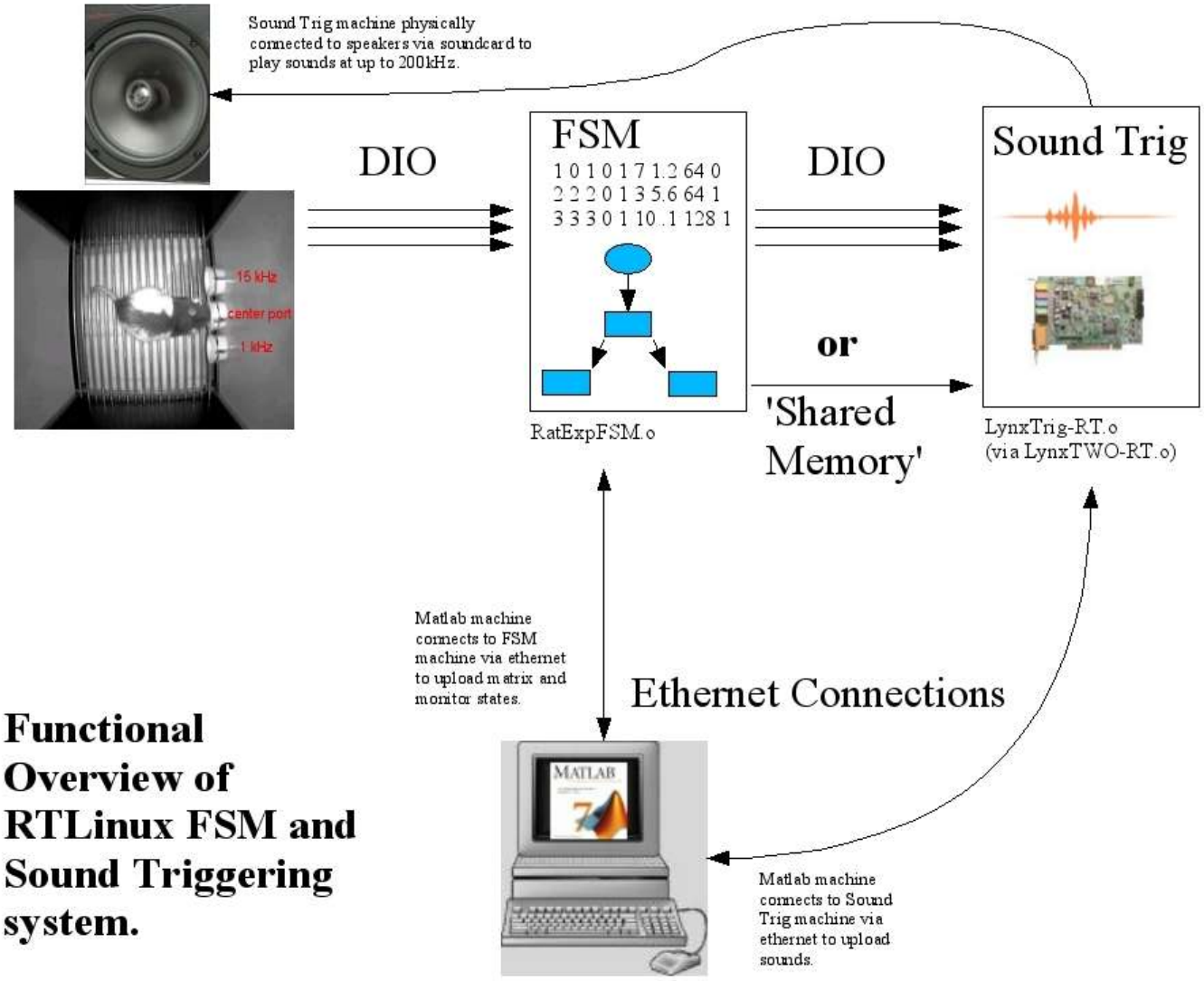
```
%                the new rate!  The default rate on a newly constructed object
%                is 44000 (44 kHz).
%
% sm = GetSampleRate(sm)
%                Get the sample rate that will be used for future calls to
%                LoadSound().
%
% sm = LoadSound(sm, unsigned triggernum, Vector_of_Int32 soundVector, String
side)
%                This command defines a particular sound to play for
%                a particular trigger number.
%
%                The soundVector is either a 1xNUM_SAMPLES array of int32s
%                in signed PCM32 format, or a 2xNUM_SAMPLES array for stereo.
%                side is either 'left', 'right' or 'both'.
%
%                tau_ms is the number of milliseconds to do a cosine^2
%                stop-ramp function when triggering the sound to 'stop'.
%                Default is 0, meaning don't ramp-down the volume.
%                Otherwise the volume will be ramped-down for a 'gradual
%                stop' over time tau_ms on trigger-stop events.
%
%                Calling this function with a stereo soundVector and any side
%                parameter other than 'both' is supported and is a good way to
%                suppress the output of one side.
%
%                Sampling rate note: Each file that is loaded to the
%                RTLSoundMachine takes the sampling rate currently
%                set via the SetSampleRate() method.  In other words, it is
%                necessary to call SetSampleRate() before calling LoadSound()
%                for each file you load via LoadSound() if all your sound
%                files' sampling rates differ!  Likewise, you need to reload
%                sound files if you want new sampling rates set via
%                SetSamplingRate() to take effect.
%
% sm = PlaySound(sm, trigger)
%                Forces the soundmachine to play a particular sound previously
%                loaded with LoadSound().
%
% sm = StopSound(sm)
%                Forces the soundmachine to stop any sounds it may
%                (or may not) be currently playing.
%
% [] = Close(sm) Begone! Begone!
%
% [double time] = GetTime(sm)    Gets the time, in seconds, that has
%                elapsed since the last call to Initialize().
%
% Calin Culianu wrote me on 26-Sep-05
```

Use Cases and Example Setups

Coming soon. :)

Installing RTLinux and Compiling The Code

Coming soon. :)



Functional Overview of RTLinux FSM and Sound Triggering system.