Nicholas Kottenstette, Panos J. Antsaklis, "neclab: The Network Embedded Control Lab," Networked Embedded Sensing and Control, Proceedings of Workshop NESC'05: University of Notre Dame, USA, October 17-18, 2005, Panos Antsaklis and Paulo Tabuada (Eds.), Lecture Notes in Control and Information Sciences (LNCIS) 331, pp. 107-126, Springer 2006.

neclab: The Network Embedded Control Lab

Nicholas Kottenstette and Panos J. Antsaklis[1]

Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556 USA
{nkottens,antsaklis.1}@nd.edu

Abstract — The network embedded control lab, *neclab*, is a software environment designed to allow easy deployment of networked embedded control systems, in particular wireless networked embedded control systems (*wnecs*). A *wnecs* is a collection of interconnected plant sensors, digital controllers, and plant actuators which communicate with each other over wireless channels. In this paper *neclab* is introduced and explained using a simple ball and beam control application. We focus on *wnecs* which use the MICA2 Motes.

1 Introduction

Typically, when a controls engineer needs to develop a new closed-loop control system she develops the control system in phases. The first phase is to develop a mathematical model of the system and synthesize a controller. The second phase is to simulate the control system using tools such as MATLAB [23]. In the third phase, using the results from the simulations, the engineer integrates sensors, actuators, remote data acquisition and control equipment into the system. This is done in order to acquire additional data and refine the models in order to optimize the controller. When the third phase is complete, the engineer has optimized and deployed a robust control system. Systems with a higher degree of autonomy will also have fault detection and remote monitoring systems. Typically these digital control systems are developed using a dedicated data acquisition system attached to a cable interfaced to a computer running a real-time-control software, such as RTLinux [1]. For control systems in which a wired control system is not possible or desired, the available design tools for the engineer are limited at best.

In this paper, a software environment is introduced called *neclab*, that is a software environment designed to allow easy deployment of networked embedded control systems, in particular wireless networked embedded control systems called *wnecs*. The components of *neclab* are presented in the following and described in terms of a classical control experiment, the ball and beam.

Note that most of the tools currently available to aid the engineer develop software for wireless embedded systems are geared specifically for sensing. The majority uses Berkley's *TinyOS* [2]. Note also that the majority of the *TinyOS* applications listed in [2], are not designed to be wirelessly reconfigurable. For example, one reconfigurable system which uses *TinyOS* is Harvard's moteLab [3], where each mote is connected to a dedicated programming board that is connected to an Ethernet cable. This is necessary in order for each mote to be reconfigured in order to use *TinyOS*. A reliable protocol, called Deluge, to enable wireless programming of *TinyOS* applications has been developed [4]. Deluge is currently part of the *TinyOS* development tree, and should be an integral part of the next stable release of *TinyOS*.

We considered Deluge but in view of our sensor and control applications of interest we decided to work with an alternative to the *TinyOS* operating system called *SOS* [16]. *SOS* offered an alternative working design for network reprogramming for the three following reasons. First the *SOS* operating system utilizes a basic kernel which should only have to be installed on the mote once. The second key element is that the *SOS* kernel supports small, typically one-twentieth the size of a *TinyOS* application, dynamically loadable *modules* over a network. Last, the *SOS* kernel supports a robust routing protocol, similar to MOAP [5], to distribute *modules* over a wireless network.

We built *neclab*, our networked embedded control system software environment using *SOS*. Specifically, *neclab* is a collection of software consisting of five main components. The first component, *build utilities*, is a set of utilities designed to build and download all required software tools and libraries in order to use *neclab*. The second component, *SOS*, is an operating system developed by the Networked and Embedded Systems Lab (NESL) at UCLA. *SOS* is a highly modular operating system built around a message passing interface (MPI) which supports various processor architectures, including those on the MICA2 Motes. The third component, *sos utilities*, are the utilities to facilitate code development and deployment of *SOS modules*. The fourth component, *necroot*, is a file system structure and language designed to seamlessly interconnect individual motes for distributed control. The fifth component, *FreeMat utilities*, are a set of utilities to facilitate *wnecs* design using FreeMat. FreeMat is a free interpreter similar to MATLAB but has two important advantages. First, FreeMat supplies a direct interface for C, C++, and FORTRAN code. Second, FreeMat has a built-in API for MPI similar to MatlabMPI [6].

neclab provides a mini-language built on facilities similar to those supported by UNIX. A modern UNIX OS supports facilities such as pipes, sockets and filters. *neclab* allows the engineer to develop a *wnecs* by designing control *modules* which can be interconnected using *networking message pipe*s. A *networking message pipe* is an abstraction to pass arrays of structured binary data from one module to another over a network. A *networking message type* indicates how the data should be handled, for example, descriptors are used to indicate standard, error, routing, and control messages which are passed over a network; e.g. control messages are indicated by the *control message type*.

Specifically, a *networking message pipe* is used to interconnect data flows between *networking source*s, *networking filter*s, and *networking sink*s. A *networking*

source creates data which will be sent over a network to *networking filter*s, and *networking sink*s. Similarly, a *networking filter* will receive data from either a *networking source* or another *networking filter*. The *networking filter* will proceed to optionally modify the data and send the new data to another *networking filter* or *networking sink*. A *networking sink* is where the network data flow for a given route ends. In order to implement *networking message pipe*s we will use the network message passing protocol provided by *SOS*. Like UNIX, *SOS* provides a way to run and halt programs which have a corresponding process id. These executable programs on *SOS* are known as *modules*. *neclab* provides an interface to pass *networking configuration* messages to a module in order to configure and enable the network flow of data between *modules* in the *wnecs* at run-time.

Using these facilities we will demonstrate an implementation of a highly parallel *wnecs* in which a secondary controller is reconfigured, while the primary controller maintains a stable control-loop. Once reconfigured, the roles of the two controllers will be switched. Other, highlights will illustrate that a controls engineer can actually create concise routing tables by simply describing a *wnecs* with *neclab*. This is a natural result of *wnecs* in general.

neclab's use of *SOS*'s dynamic memory allocation services, easily allows for a system which enables a control engineer to work through the second and third phases of her design project. This highly configurable environment without wires would have been difficult to implement with *TinyOS* since *TinyOS* does not support dynamic memory management. *SOS* on the other hand does. Other *SOS* features which *neclab* utilized are the ability to dynamically share functions and load executable files (modules) over a wireless channel while the *SOS* kernel is still running. *SOS* implies flexibility, and as a result it was chosen as the core component to *neclab*. For a more detailed discussion on the advantages and differences of *SOS* as compared to other solutions, refer to [7]. *neclab* is not the first project to utilize *SOS*. Other projects such as Yale's XYZ Sensor Node project [8] and various projects at NESL are starting to use *SOS* such as the RAGOBOT [9].

In presenting *neclab*, we will illustrate its use by presenting a typical undergraduate control lab problem modified to be a *wnecs*. We will then generalize this problem, by describing a tutorial application, which a user can create if five MICA2 Motes and a programming board are available. As the tutorial application is described we will highlight the various components of *neclab* which highlight the many issues that have to be addressed in order to develop a robust *wnecs*.

2 Problem Description

Consider an undergraduate controls laboratory experiment that teaches a student how to control the position of a metal ball on a beam. The experiment uses a motor as an actuator, and two variable Wheatstone bridges for sensing. The bridges measure angular position of the motor, and the absolute position of the ball on the beam. In the first laboratory experiment, the students are required to determine the actual model parameters of the ball and beam plant [10]. The next lab [11] teaches the

student how to control the exact position of the ball on the beam. The student designs and implements the control system using MATLAB, Simulink [23], and the Wincon server for real-time control [12]. The sensor inputs and outputs are accessible through the MultiQ board. We are going to replace this system using *neclab*, the MICA2 motes and a general purpose I/O boards developed for the MICAbot [13].

Figure 1 illustrates such a system. With *neclab* installed on a host computer a MICA2_N_gateway mote is typically accessed via a serial port. Figure 1 indicates that MICA2_N_gateway is interconnected to a MIB510 programming board. See [14] for additional details on the MIB510. In order to control the ball and beam plant the following control loops (jobs) need to be implemented (spawned). First the MICA2_N_actuator needs to reliably control the angular position α of the beam. This is achieved by controlling the angular position θ_l of the motor. The actuator will receive a desired angular position set-point and will control the motor. In networking terms, the MICA2_N_actuator behaves as a *networking sink* for networking messages, and takes actions based on the messages sent to its *control standard input*. According to [11] the desired response time for controlling α should be around 0.5 seconds. This is a fairly aggressive target to meet for the low data rate wireless network control system. As a result, we have initially kept this control loop internal to the MICA2_N_actuator. In order to do this we link this code statically to the kernel in order to guarantee a stable control loop on start-up. The second control loop involving the actual position of the ball, requires around a 4 second settling time, which is reasonable to implement over the wireless channel. This loop is accomplished by MICA2_N_sensor sampling data from the ball position sensor output with the ATmega 128L built-in 10 bit A/D converter (see [15] for additional information on this chip's features). The MICA2_N_sensor behaves as a *networking source* for generating networking messages, sending its data along to MICA2_N_controller-A and MICA2_N_controller-B respectively. Depending on which controller is enabled, the enabled controller will behave as a *networking filter* by calculating an appropriate command to send to MICA2_N_actuator based on the users desired set-point received. Figure 2, illustrates how this system can be implemented using *SOS* modules and messages which we will refer to as we discuss *neclab*.

3 *neclab*

Looking at figure 2, one can appreciate the number of distinct software components required for an engineer to obtain a working *wnecs*. In this figure the engineer has successfully built and installed kernels on five motes, loaded all the required modules on to the network, and created routing tables in order to create a stable closed loop controller to monitor and maintain the position of the ball. In order to use *neclab* the engineer must first download the 1.x version of *SOS* to a unix based PC. The location *SOS* is installed will be referred to as SOSROOT; *neclab* will be in the SOSROOT/contrib/*neclab* directory which will be referred to as NECLABROOT. From there all the engineer needs to do is follow the instructions in the NECLABROOT/README_SOS-1.X file. The first task that *neclab* will do for the user is

download, build and install all the necessary tools to build and install software on the MICA2 motes so to work with the FreeMat environment. Once the tools are installed, *neclab* will apply some minor patches in order to fully utilize the *SOS* software. From there the engineer should test and fully understand the example blink_lab. The blink_lab is discussed in Appendix A.

3.1 *build utilities*

neclab has been designed so that a user does not require root access to build and install her tools. This offers two distinct advantages, the first being that the user can modify any appropriate component that is needed to maximize the performance of the system. For example, *neclab* actually downloads and allows the user to build an optimized BLAS (Basic Linear Algebra Subprograms) library using ATLAS (Automatically Tuned Linear Algebra Software) [17]. Second it provides all users with a consistent tool-kit eliminating potential software bugs associated with not using a consistent tool-chain. The key tool used is the build_tool_makefiles program which reads a configuration file located in NECLABROOT/etc/build.conf and generates a custom makefile for all the following tools:

- perl – key tool for various utilities in *neclab* [18]
- avr-binutils – used for the MICA2 and MICAz motes [19]
- avr-gcc – used for the MICA2 and MICAz motes [20]
- avr-libc – used for the MICA2 and MICAz motes [21]
- ATLAS – used with FreeMat[17]
- FreeMat [22]
- uisp – used to load an image on to the MICA2 and MICAz motes [24]
- tcl – (Tool Command Language) required for the tk library [25]
- tk – graphical user interface toolkit library required for python [25]
- python – [26] used in conjunction with pexpect [27] for automation
- SWIG – [28] is a tool to connect programs written in C and C++ with high-level programming languages such as perl and python.

The auto-generated makefiles are then installed in each corresponding NECLABROOT/src/build_utilities/$< tool >$ directory and invoked to download, build and install each $< tool >$. The installation directory is in NECLABROOT/tools/. As a result any user can build and use her own tools, without having to ask the system administrator for permission!

3.2 *SOS*, and *sos utilities*

Referring back to Figure 2, on the local host PC (HOST_PC), the engineer has just finished creating a *wnecs* using the *neclab_run* tool provided by *neclab*. Each mote has a kernel; however, they do not have to be unique, as is clearly shown in Figure 2. For example, the MICA2_N_gateway mote has the sosbase kernel which has a statically linked module which we will refer to by it's process id SOSBASE_PID. Other motes such as MICA2_N_sensor, MICA2_N_controller-A, and

MICA2_N_controller-B have a blank_sos kernel with no statically linked modules. Finally the MICA2_N_actuator has a custom kernel, custom_sos, with statically linked modules ANG_POS_SEN_PID (a module which manages the angular position sensor), and MOT_ACT_PID (a module which controls the angular position on the motor). The custom_sos kernel was required to generate a pwm output in order to drive the H-Bridge on the MICAbot board. The remaining modules which are dynamically loaded and unloaded over the network, are either in an active or inactive state. When in an inactive state they do not consume any additional processor RAM but do take up program space in the flash.

In order to load and unload the modules the following programs are required. First, the sos server, sossrv, needs to be built, installed and started. *neclab* manages this with the makerules and *neclab_{sim,run}* commands. The makerules first manage building and installing sossrv into NECLABROOT/tools/bin. The *neclab_{sim,run}* commands can be used either to begin a simulation of a *wnecs* with the *neclab_sim* command or to run a *wnecs* using the *neclab_run* command. Either command can be treated as equivalent for discussion. The *neclab_run* command starts the sossrv and connects it to the MICA2_N_gateway mote. Figure 2 indicates that sossrv is listening for clients on HOST_PC via the loop-back interface on port 7915 while listening for incoming packets filtered through the MICA2_N_gateway attached to /dev/ttyS0.

Next the *neclab_run* tool creates an input fifo and starts the *SOS* modd_gw client. The modd_gw client is an *SOS* application that runs natively on a PC, it provides a shell like interface in which user input can be redirected to the fifo for automation. The modd_gw client maintains a database file .mod_version_db local to where it is started. This database tracks the different dynamic modules which are loaded and unloaded from the network. If another engineer chose to use the same motes in the lab, they will either need access to this file or re-install new kernels on all the motes. As a result *neclab* makes sure that the modd_gw is started such that the database is located in a publicly accessible directory such as /tmp so others can access and run their own experiments. The *neclab_run* tool then proceeds to build all the required network modules, load them on to the network, and establish the networking routes for the control system. Lastly, the closed loop control system is enabled and can be monitored and modified as necessary.

Another tool *neclab* provides is the create_module_proto tool to generate a new module prototype for beginning development. *neclab* has built into its makerules a mechanisms to generate tags files to assist in tracking all the interrelationships that modules have with the kernel and other modules. Once the engineer has a satisfactory implementation she can use the appropriate options from the *neclab_run* tool to easily rebuild and re-install new module images as necessary. These tools provide a streamlined mechanism for simulating and generating *wnecs*. Building off of *SOS*'s novel technique of tracking module version numbers as they are dynamically loaded and unloaded. *neclab* has created a file-structure known as *necroot* to track, simulate, and develop *wnecs*.

3.3 *necroot*

Modules are tracked by their process id, similar to a process id generated by the ps command on a unix machine. This id is actually used to direct messages which are passed on an *SOS* network. Every message passed on the *SOS* network has a corresponding destination process id and address. The process id field; however, is only 8 bits, which supports only 255 unique process ids. Clearly more than 255 unique modules will be developed to be run on the *SOS* operating system, so there needs to be a clean way to address this limitation. *SOS* uses two files to define the association of a process id with a given module, mod_pid.h and mod_pid.c. The gen_mod_pid tool combined with makerules and the *necroot* design allow for dynamic generation of mod_pid.h and mod_pid.c for a corresponding lab project.

In *necroot* the NECLABROOT/src/*necroot*/modules.conf provides a line by line list in which each entry consists of a full-path to a corresponding module and a short description of the module. Each item is delimited by a colon. The module process id is parsed directly from each modules.conf entry and added to mod_pid.h. The corresponding description is then added to mod_pid.c for simulation and debugging. Furthermore, static modules and modules to be loaded over the network are uniquely identified by grouping these modules between the $< static >, < /static >$, and $< network >, < /network >$ tags respectively. The modules.conf is to serve as a global file, in it the key modules for *neclab* are listed. These entries include the moduled_pc module (MOD_D_PC_PID) and *neclab*'s configuring module (CONFIGURING_PID). The moduled_pc module, is a statically linked module which runs on the modd_gw client. The configuring module provides an interface for creating *networking source*s, *networking sink*s and *networking filter*s. It also provides the interface to configure modules and enable the *control standard input*/*control standard output* networking design referred to in the introduction. The remaining modules are identified in the engineer's NECLABROOT/labs/ball_beam_lab/etc/modules.conf.local file.

The next issue is to create a design which would allow a user to easily manage programming and tracking each corresponding mote's kernel and module configuration. This is solved by the construction of the NECLABROOT/src/*necroot*/etc/network.conf and the corresponding
NECLABROOT/labs/ball_beam_lab/etc/network.conf.local files. Each entry follows nearly the same language structure as the build.conf file described in the *build utilities* section. The only difference instead of identifying a tool, each entry describes a mote and all the properties local to that mote. Using the build_mote_makefiles tool, a custom makefile for each mote described in the network configuration files is generated. Then the *necroot* directory structure is generated in the ball_beam_lab directory. Furthermore the engineer can use *neclab_run* with the appropriate options to build and install the corresponding kernel on to her motes. For reference, a typical network.conf entry for a mote is as follows:

```
<mote>
SOS_GROUP = 13
ADDRESS = 2
NECLAB_PLATFORM = mica2
```

```
KERNEL_DIR = ${NECLABROOT}/src/patched_sos-1.x/sosbase  #$
X = 6
Y = -12
Z = 5
LOC_UNIT = UNIT_FEET
TX_POWER = 255
#CHANNEL = ? has no effect for mica2
</mote>
```

The configuration language is GNU Make combined with some XML-like tags to separate each mote entry. Although, not true for mobile motes, each non-mobile mote has a fixed location. This information is typically used for geographic routing protocols, such as those implemented on Yale's XYZ platform. *SOS* has built in the capability to track the location of each mote into its kernel; hence, the X, Y, Z, and LOC_UNIT entries. The *neclab* project assisted in this design by introducing the Z dimension, a LOC_UNIT to associate relative position with a given dimension, and a gps_loc variable to track GPS location information. The gateway mote should typically assign itself a GPS location in order to assist with routing and interconnecting of other laboratories around the globe. Presently, gps_loc maintains precision down to one second. A sample GPS entry, here at Notre Dame would have the following format (note the Z coordinate is an elevation relative to sea-level in feet).

```
GPS_X_DIR = WEST
GPS_X_DEG = 86
GPS_X_MIN = 14
GPS_X_SEC = 20
GPS_Y_DIR = NORTH
GPS_Y_DEG = 41
GPS_Y_MIN = 41
GPS_Y_SEC = 50
GPS_Z_UNIT = UNIT_FEET
GPS_Z = 780
```

Note, each platform that *SOS* supports has a wireless radio setting unique to each mote. In the above sample the radio was set to full power. *neclab* is also working on formalizing the interface to modify the frequency and channel of the radio. As the comment notes, the MICA2 mote does not currently support the channel interface; however, the MICAz mote does. The other parameters such as the *SOS*_GROUP id is used to filter radio packets out in the network that do not belong to that particular group. Each mote entry should have a unique, (*SOS*_GROUP, ADDRESS) pair. Being able to group motes and designate different channels, provides one way to allow multiple laboratories to interact and perform co-operative tasks.

For example, the ball and beam lab, can be modified to simulate the transfer of one ball from one station to another, as might be done in transferring parts from one assembly cell to another. Furthermore, by building in the ability to group the motes and assign each group to a separate radio channel, we have created a mechanism to create groups which can co-operate without worrying about generating radio

interference. This feature we plan to exploit with the MICAz motes when developing routing algorithms between groups. The routing will be implemented between the gateway motes of a group. The gateway mote will typically be attached to their respective HOST_PC and the routing of packages will occur over the Internet.

Finally, the issue of declaring the desired kernel is addressed with the KERNEL_DIR entry. The KERNEL_DIR entry provides the full path to the desired kernel to be loaded and built. Any modules that the engineer plans to statically link in her kernel should be addressed in her respective kernel makefile.

After the engineer has successfully built her newly created *necroot* file structure, the following root tree will result.

```
[user@host_pc ball_beam_lab]$ find ./ -name "mote*"
./root/group13/mote1
./root/group13/mote2
./root/group13/mote3
./root/group13/mote4
./root/group13/mote5
./root/group14/mote1
./root/group14/mote2
./root/group14/mote3
./root/group14/mote4
./root/group14/mote5
[user@host_pc ball_beam_lab]$
```

In each mote directory there is a module configuration file, corresponding to every module identified in the corresponding modules.conf and modules.conf.local files. These are used in conjunction with the configuring module to set module parameters and set up routes in the network. Looking in the mote1 directory for example the user will see the following additional files.

```
[user@host_pc ball_beam_lab]$ ls -1 root/group13/mote1/*.mod
root/group13/mote1/ball_beam_con.mod
root/group13/mote1/ball_pos_sen.mod
root/group13/mote1/configuring.mod
root/group13/mote1/moduled_pc.mod
root/group13/mote1/ang_pos_sen.mod
root/group13/mote1/mot_act.mod
[user@host_pc ball_beam_lab]$
```

These user editable files provide an interface in order to use the configuring module to create routes. These routes can be configured using the command line or can optionally be declared in the routing.conf file. All lines starting with a # are comments. What the following segment of routing.conf should illustrate is that we have created a compact language to describe networking routes. The net_source, net_filter, and net_sink commands are intended to emphasize that a module will be configured to be in one of those three states. The optional arguments such as –d{0,1} are to show that the engineer can describe up to two destinations and the behavior is determined by

the respective –m{0,1} option which describes the *networking message type* for each destination. This allows an engineer to redirect control data flow into error data flow for monitoring for example. It should also be noted that although we are using the *necroot* file system to maintain and describe our routes, an ad-hoc routing module could be designed to utilize the configuring module interface in order to build arbitrary networking routes based on some metric. This language can further be utilized to describe these routes using a graphical user interface.

The ball_beam_lab's etc/routing.conf file is as follows:

```
#!/bin/sh
#routing.conf
#define some constants
TIMER_REPEAT=0;TIMER_ONE_SHOT=1
SLOW_TIMER_REPEAT=2;SLOW_TIMER_ONE_SHOT=3
MSG_STANDARD_IO=35;MSG_ERROR_IO=36
MSG_CONTROL_IO=37;MSG_ROUTING_IO=38
cd $NECLABROOT/labs/ball_beam_lab/root/group13
#Main control-loop
#MICA_2_N_sensor -> MICA_2_N_controller-A -> MICA_2_N_actuator
#Secondary control-route
#MICA_2_N_sensor -> MICA_2_N_controller-B
#Configure the MICA_2_N_sensor (mote2/ball_pos_sen) routes
net_source -m mote2/ball_pos_sen -t "$TIMER_REPEAT:1024"  \
           --d1=mote3/ball_beam_con --m1=$MSG_CONTROL_IO  \
           --d2=mote4/ball_beam_con --m2=$MSG_CONTROL_IO
#Configure MICA_2_N_controller-A (mote3/ball_beam_con) routes
net_filter -m mote3/ball_beam_con -t "$TIMER_REPEAT:2048" \
           --d1=mote5/mot_act --m1=$MSG_CONTROL_IO        \
           --d2=mote1/moduled_pc --m2=$MSG_ERROR_IO
#Configure MICA_2_N_controller-B (mote4/ball_beam_con)
net_sink -m mote4/ball_beam_con -t "$TIMER_REPEAT:2048"
#Configure MICA_2_N_actuator (mote5/mot_act)
net_sink -m mote5/ball_beam_con -t "$TIMER_REPEAT:256"
#Activate the routes $
net_enable {mote5/mot_act,mote4/ball_beam_con}
net_enable {mote3/ball_beam_con,mote2/ball_pos_sen}
```

As shown in Figure 2, three routes are clearly established. One route establishes the control loop from sensor to controller to actuator. A second route delivers sensor and controller debugging information back to a gateway mote. A third route enables a second controller to act as a slave. The ball-position-sensor module on mote2 is configured as a *networking source*, generating a *control standard input output* message to be sent to the ball-beam-controller module, on mote3 every second or 1024 counts. The ball-beam-controller on mote3 is configured as a *networking filter*. As a *networking filter* it handles *control standard input output* messages from mote2's ball-position-sensor, computes the appropriate command and sends a *control standard*

input output message to mote5's motor-actuator module. Furthermore mote3's ball-beam-controller module will generate a debugging message destined for the gateway mote1's non-resident moduled_pc module every two seconds. By sending information to the gateway mote, *neclab* can support remote monitoring. The moduled_pc module was arbitrarily chosen to illustrate that an arbitrary non-resident module id can be reserved in order to pass a message up to the sossrv program and a client can be designed to handle and display this message on the engineers HOST_PC display. Terminating the control loop route as a *networking sink*, mote5's motor-actuator, handles *control standard input output* messages from mote3's ball-beam-controller and actuates the beam. The motor-actuator controls the angular position of the beam and requires a faster control loop of a quarter of a second; hence, the timer is set to 256 counts.

3.4 *FreeMat utilities*

The *FreeMat utilities* will provide an interface for users familiar with MATLAB to appropriately modify configuration parameters for *neclab* modules designed and written in C. These utilities are currently the least developed for *neclab*; however, the major design problems have been confronted. There were numerous obstacles which had to be overcome in order to develop these utilities. The first accomplishment was getting FreeMat to build and install. Second was to integrate ATLAS, so that the engineer can have an optimized matrix library for simulation and development. Third was to develop the configuring module to allow a higher level networking protocol to be implemented in order to interconnect modules in a manner similar to the MPI protocols identified in the introduction. Lastly, was identifying SWIG as a possible candidate to assist with generating shared libraries to allow us to interface our *SOS* modules with FreeMat. We have used SWIG to generate interface files and shared libraries for python which we have used to create our initial graphical user interface application. We have also used SWIG to interface to our configuring module and do initial testing of the module. Although, FreeMat does provide a native interface for C and C++ programs, we feel that learning to effectively use SWIG to handle interfacing with *SOS* will allow a more flexible development environment in which users of *neclab* can use whatever high-level software tools they desire to use.

The *FreeMat utilities* will allow an engineer to generate her own routing tables while allowing users to receive and monitor data from modules using the *FreeMat client*. The *FreeMat client* will connect to the sossrv and relay all data destined for the *FREEMAT_MOD_PID*. Setting parameters and displaying data should be transparent due to the configuring interface provided by the combined configuring module and the data flash swapping interface provided by *SOS*. The configuring interface provides all the necessary elements for a module to establish up to two routes, configure a timer and change up to 8 bytes of parameter data in the modules RAM. To handle larger data-sizes the user can either rely on the larger *SOS* RAM memory block of 128 bytes. The difficulty is that there are only four 128 byte RAM blocks available for the MICA2 and MICAz motes on *SOS*. The problem is further compounded in that the radio requires at least one 128 byte block to receive an *SOS* message over the

network. In order to simultaneously send and receive a 128 byte message, a second 128 byte block of RAM needs to be allocated by the radio. This means that the user essentially has only two 128 large blocks of RAM available for allocation and they should be used for temporary operation such as being allocated to perform linear algebra routines. The second option is to dedicate a section of internal flash memory for storing configuration parameters and reading the configuration parameters into the local stack during module run-time. This is a preferred option because swapping in 256 bytes of data into the stack should only require around a tenth of a millisecond. This is a feasible option as long as the engineer utilizes the co-operative scheduler provided by *SOS*, and avoids interrupt service routines, and nested function calls which require large amounts of the stacks memory.

Being able to effectively manage the RAM and stack will allow *neclab* to support a much larger design space. For example, by gaining the ability to configure up to 256 bytes of data, the engineer can begin to develop four-by-four full-state observers. The following sections of configuring.h illustrate both the *networking configuration* and *control standard input* interface. The *networking configuration* is defined by the configuring_message_type. The *control standard input* is handled using the standard_io_message_t and indicated by the MSG_CONTROL_IO message id.

```
#define MSG_CONFIGURING         MOD_MSG_START
#define MSG_CONFIGURING_ENABLE  (MOD_MSG_START + 1)
#define MSG_CONFIGURING_DISABLE (MOD_MSG_START + 2)
/* #define MSG_*_IO */
#define MSG_STANDARD_IO         (MOD_MSG_START + 3)
#define MSG_ERROR_IO            (MOD_MSG_START + 4)
#define MSG_CONTROL_IO          (MOD_MSG_START + 5)
#define MSG_ROUTING_IO          (MOD_MSG_START + 6)
/* #define MSG_*_IO */
#define CONFIGURING_SOURCE        (1<<0)
#define CONFIGURING_SINK          (1<<1)
#define CONFIGURING_FILTER        (1<<2)
#define CONFIGURING_ENABLED       (1<<3)
#define CONFIGURING_CONFIGURED    (1<<4)
#define CONFIGURING_RESERVED      (1<<5)
#define CONFIGURING_TIMER_0_BIT_0 (1<<6)
#define CONFIGURING_TIMER_0_BIT_1 (1<<7)
#define CONFIGURING_TIMER_0       0
#define SIZE_OF_MULTI_HOP_PACKET  SOS_MSG_HEADER_SIZE
#define SIZE_OF_MSG_SMALL_BLOCK   (32)
#define SIZE_OF_MSG_LARGE_BLOCK   (128)
typedef struct destination_type{
    uint16_t daddr;  /* Destination address  */
    sos_pid_t did;   /* Destination pid      */
    uint8_t  type;   /* Message type to send */
} __attribute__ ((packed))
```

```
destination_t;

#define CONFIGURING_N_DESTINATIONS 2
typedef struct configuring_type{
    uint8_t flag;  /* source, sink, etc. */
    uint8_t size;  /* Size of user data  */
    sos_pid_t pid; /* pid of user module */
    uint8_t pad;   /* pad for alignment  */
    /* destinations takes up to 8 bytes */
    destination_t destinations[CONFIGURING_N_DESTINATIONS];
    uint8_t       data[8]; /* 8 bytes for user to store
                              user specific data */
} __attribute__ ((packed))
configuring_t;/* 20 bytes leaving 12 bytes for 6 function
                 pointers in the app_state_t structure
                 = 32 bytes = SIZE_OF_SMALL_BLOCKS
                 for the avr processor */

typedef struct configuring_message_type{
    int32_t interval; /* The ticks/counts you want to place
                         for the timer if timer < 0 the
                         timer is disabled by definition! */
    configuring_t conf;
} __attribute__ ((packed))
configuring_message_t; /* 24 byte packet leaving 8 bytes
                          (8 required) for multi-hop routing
                           worst case */

typedef enum {UINT8_T, INT8_T, UINT16_T, INT16_T,
              UINT32_T, INT32_T, FLOAT_T,
              USER_STRUCT_T} standard_io_types;

typedef struct standard_io_message_type{
    /* Engineer can send a vector of data_types */
    uint8_t data_r;
    /* Engineer can send an array of structures
       as long as the size_of is specified */
    uint8_t data_size_of;
    /* One of standard_io_types */
    uint8_t data_type;
    /* sid of sender of standard io message */
    sos_pid_t sid;
    uint8_t  data[MAX_STANDARD_IO_LENGTH]; /*32-8-4=20*/
} __attribute__ ((packed))
standard_io_message_t;
```

Refer to the *neclab* documentation for additional details.

4 Conclusion

In developing *neclab* many challenging design issues were addressed and solved. *neclab* first addressed the need to have a consistent tool chain, such as cross-compilers, for the MICA2 motes. This design problem was solved using *neclab*'s *build utilities*. The next design issue to be addressed was automating all the tasks associated with loading software on to the MICA2 motes. This was addressed by developing *sos utilities*. These utilities assisted in generating new modules, loading modules on to the network, building and installing new kernels on to the MICA2 motes. The next design issue was to create a mechanism to track the type of kernels and properties of each MICA2 mote on the network. This was accomplished, using the *necroot* design. Building on the tools developed from *build utilities*, the network.conf file provides a simple interface to manage each motes kernel, static-modules, radio configuration, location, group, and address properties. Next by showing a well-designed configuring module interface combined with *necroot* we demonstrated an efficient way to establish routes using a routing.conf file. Finally, all these components allow us to build a set of *FreeMat utilities* to enable control-engineers to develop *wnecs*, using a MATLAB like interface.

Using the routing.conf file, we also illustrated how control systems naturally create their own routes. The sensor is a *networking source*, creating a time-base for the system and generating messages to be routed to either a *networking filter*, or *networking sink*. The controller is a *networking filter*, receiving messages from the sensor, modifying the data and sending an appropriate message to either a *networking filter*, or *networking sink*. In the ball and beam example the controller sent a command message to the motor actuator. The motor actuator behaved as a *networking sink*, receiving the messages from either a *networking source* or *networking filter* and applied the appropriate input to the motor. We also showed in the ball and beam example how by utilizing a redundant controller we can accomplish the reprogramming on the fly. Specifically, the second controller, initially configured as a *networking sink*, can be reconfigured with new control parameters and then switched to become a *networking filter* while configuring the initial controller to be a *networking sink*. This shows that if a control system requires extensive time to reconfigure itself, the parallel architecture just described, can safely do so without having to risk having the system go unstable. This is a valuable feature, which may be used for control, reconfiguration and fault diagnostics and identification, switching control, etc.

Working with the memory constrained MICA2 and MICAz motes, we identified a unique way to increase our module configuration space. State-space control applications, in particular, those which require a state-observer, require large amounts of memory to describe a model of the system. In order to describe the system and make it re-configurable, we have identified that a section of internal flash can be reserved for reading in module control parameters into the stack during module runtime. Using *SOS* with it's co-operative scheduler, an engineer can wirelessly send

new control parameters to the internal flash section in order to reconfigure and tune a state-observer. The current implementation is slower, however, due to the increased memory read times from the internal flash section.

Although, *neclab*, is in its initial release, it has become a fairly advanced piece of software. One limitation, is related to the limited message routing infrastructure. Presently, *neclab* relies on the built in *SOS* message routing structure, which is quite advanced; however, configuration messages can only be established if the gateway mote can access an individual node directly on the network. We are particularly interested in adding a more advanced routing interface, to enable multiple-hop routing for initial configuration. We are going to be actively using *neclab* for designing *wnecs*, and hope the control-community will find *neclab* a useful tool for their own research.

5 Acknowledgment

We would like to thank Simon Han from NESL, UCLA for inviting us to contribute to the *SOS* project and making *neclab* one of many projects which will make extensive use of the *SOS* operating system. Simon has provided valuable development, feedback and support for integrating *neclab* with *SOS*. We would also like to acknowledge the rest of the *SOS* team: Ram Kumar, Roy Shea, Andrew Barton-Sweeney, Eddie Kohler and Mani Srivastava for their collective feedback and support. Finally, we would like to acknowledge Joey Ernst for testing and assisting in the development of *neclab*.

References

1. Victor Yodaiken, FSMLabs Lean POSIX for RTLinux , http://www.fsmlabs.com/fsmlabs-lean-posix-for-rtlinux.html
2. Berkley WEBS: Wireless Embedded Systems, TinyOS Community Forum: Related , http://www.tinyos.net/related.html
3. Harvard University: Maxwell Dwoorkin Laboratory, moteLab Harvard Network Sensor Testbed , http://www.motelab.eecs.harvard.edu
4. Jonathan Hui, Deluge 2.0 - *TinyOS*Network Reprogramming, http://www.cs.berkeley.edu/∽jwhui/research/deluge/deluge-manual.pdf
5. Thanos Stathopoulos, John Heidemann and Deborah Estrin, "A remote Code Update Mechanism for Wireless Sensor Networks," CENS Technical Report # 30, December 2003, http://lecs.cs.ucla.edu/∽thanos/moap-TR.pdf
6. Dr. Jeremy Kepner, Parallel Programming with MatlabMPI http://www.ll.mit.edu/MatlabMPI/
7. Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava, SOS: A dynamic operating system for sensor networks. Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (Mobisys), 2005
8. D. Lymberopoulos and A. Savvides, XYZ: A Motion-Enabled, Power Aware Sensor Node Platform for Distributed Sensor Network Applications. Proceedings of IPSN 05, Los Angeles, CA, April 25-27 2005

9. NESL (Networked and Embedded Systems Laboratory), RAGOBOT.COM, http://www.ragobot.com
10. University of Notre Dame Department of Electrical Engineering, Model identification of Ball and Beam Plant , http://www.nd.edu/∽eeuglabs/ee455/lab/lab3/lab3_2003.pdf
11. University of Notre Dame Department of Electrical Engineering, Ball and Beam Balancing Problem , http://www.nd.edu/∽eeuglabs/ee455/lab/lab4/lab4_2003.pdf
12. Quanser WinCon 5.0, http://www.quanser.com/english/html/solutions/fs_soln_software_wincon.html
13. M. Brett McMickell, Bill Goodwine, and Luis Antonio Montestruque. MICAbot: A Robotic Platform for Large-Scale Distributed Robotics. In Proceedings of the 2003 IEEE International Conference on Robotics & Automation Taipei, Taiwan, September 14-19, 2003
14. Crossbow Technology, Inc. MPR-MIB Series Users Manual, http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf
15. Atmel ATmega128(L) Users Manual, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf
16. SOS Operating System, http://nesl.ee.ucla.edu/projects/sos-1.x/
17. Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/
18. Larry Wall, Tom Christiansen, Randal L. Schwartz. *Programming Perl, Third Edition*. O'Rielly 2000
19. GNUBinutils, http://www.gnu.org/software/binutils/
20. Host/Target specific installation notes for GCC, http://gcc.gnu.org/install/specific.html#avr
21. AVR C Runtime Library, http://savannah.nongnu.org/projects/avr-libc
22. FreeMat, http://freemat.sourceforge.net/
23. MATLAB, http://www.mathworks.com/
24. AVR In-System Programmer, http://savannah.nongnu.org/projects/uisp/
25. Tcl/Tk, http://www.tcl.tk/software/tcltk/
26. Python Programming Language, http://www.python.org
27. Pexpect - a Pure Python Expect-like module, http://pexpect.sourceforge.net/
28. Simplified Wrapper and Interface Generator, http://www.swig.org

6 Appendix A

As shown in Figure 3, a basic network has been created and a blink module has been loaded and enabled on each MICA2 mote in order to control the blink rate of the yellow, green, and red LEDs. Each mote is a member of group 2, and each mote name corresponds to its address (mote1 has an address of 1). The blink module utilizes the configuring module interface and a module specific LED structure. The LED structure contains the state of each LED, a basic clock structure for each LED, and a mask to selectively filter the state of a given LED. Mote1 serves as a *networking source* in which its timer is configured to repeat every 1024 counts or one second. Once configured, and enabled by a MSG_CONFIGURING_ENABLE message, mote1's blink module will change the status (on/off) of the red LED every 4 seconds, green LED every 2 seconds, and the yellow LED every 8 seconds. Each time the led_state changes a MSG_STANDARD_IO message is delivered to mote2's

blink module. Figure 3 indicates that mote1 currently has the yellow and green LEDs on. The message received by mote2 contains the new led_state of mote1. Mote2, configured as a *networking filter* masks the received led_state such that only the red and green led state will pass. As a result only mote2's green LED is indicated as on. In this example, we chose to let the interval timer for mote2's blink module update the new status of the led state on the physical blink display so there could be a delay up to half a second. This is done to illustrate that we can isolate the handling of asynchronous events with a synchronous task. Last, the filtered led_state of mote2 is delivered to mote3 as a MSG_STANDARD_IO message. Mote3, configured as a *networking sink* updates the newly received led_state and displays the lit green LED with a lag of no greater than half a second. Note that modules on the network can be enabled/disabled by broadcasting a MSG_CONFIGURING_{ENABLE,DISABLE} message or motes can be enabled/disabled individually. Configuring messages are delivered reliably to individual motes, using the SOS_MSG_RELIABLE interface. As a result, configuring messages which are broadcast can not currently be acknowledged as being reliably sent; however, may still be deemed useful for a quick initial shutdown of a system.

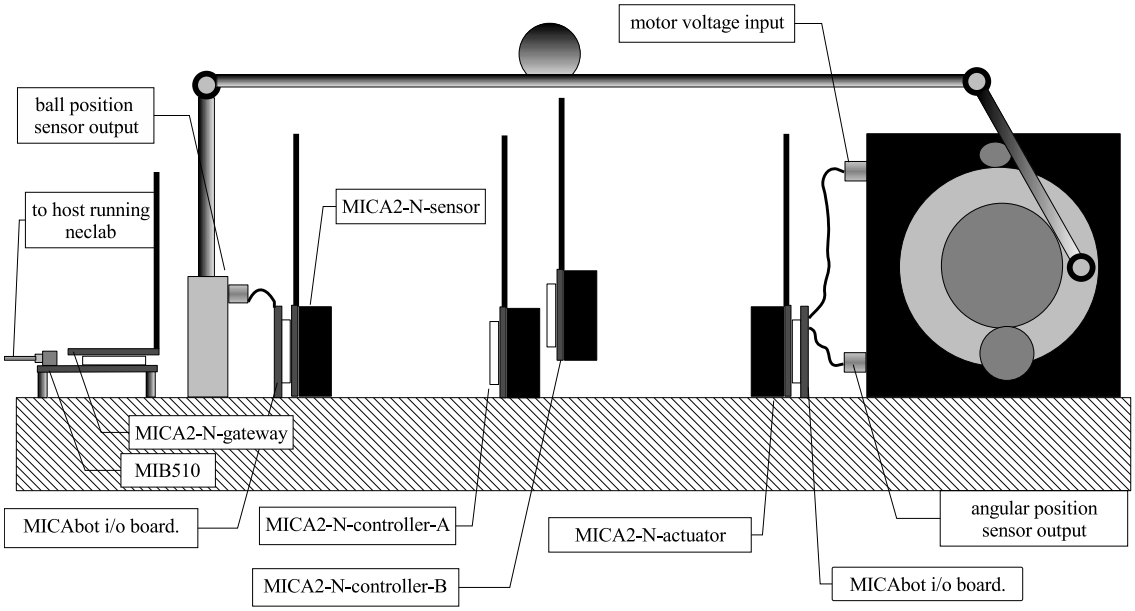

Fig. 1. Ball and Beam Network Embedded Control System.

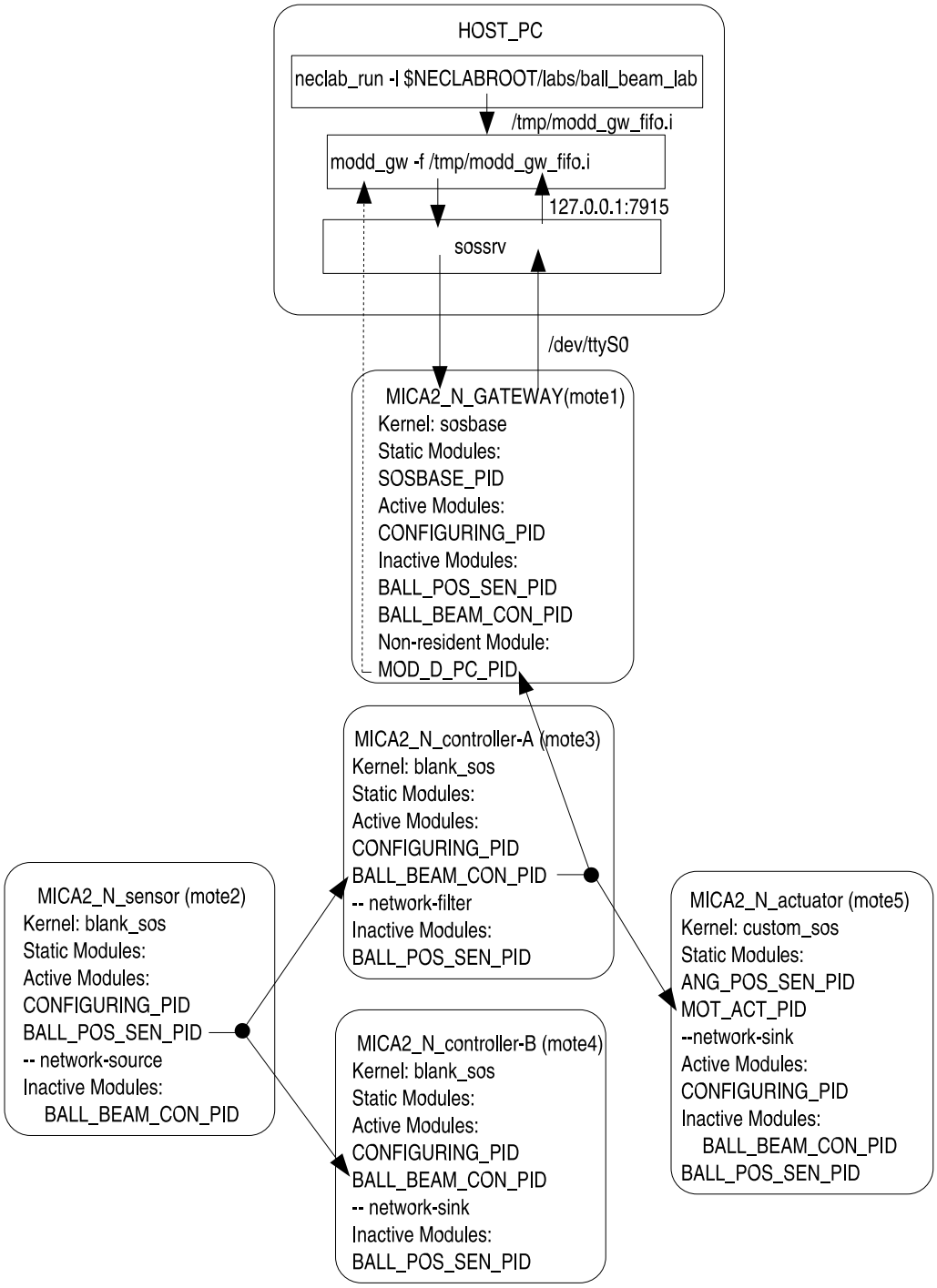

Fig. 2. *neclab* block diagram for ball and beam control.

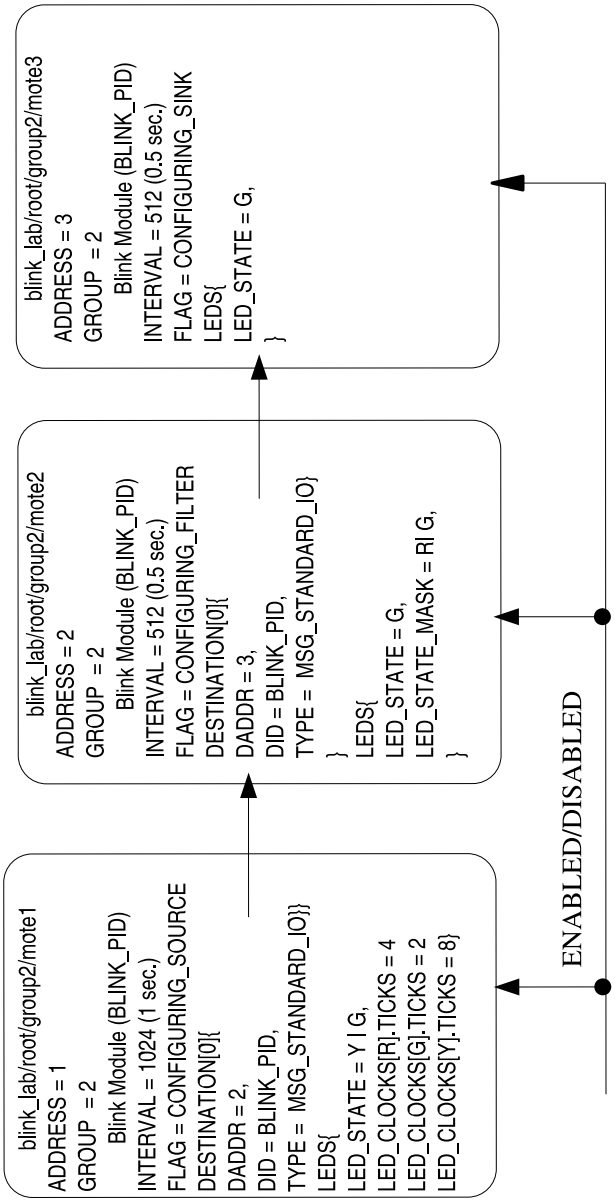

Fig. 3. blink_lab network routing diagram.