Contents

Chapter 1

Installation

The `Legno` compiler has been tested on OSX and Linux systems. It is a command-line utility – all commands in this document should be executed using the command line.

1.1 Setting up the Compilation Environment

1.1.1 Mac OS

In order to install the dependencies for the `Legno` compiler, the `brew` package manager (https://brew.sh/). Brew is installed using the following command:

```
/usr/bin/ruby -e \
"\$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once brew is installed, it may be accessed using the command `brew`. Next, we should install the required dependences.

```
brew install python3 git graphviz
```

You might need to execute the following command to get get `pip3` setup:

```
brew postinstall python3
```

Testing the installation:If this step executed correctly, each of these commands should return the usage information for the command:

```
pip3 --help
python3 --help
git --help
dot --help
```

If any of these commands return the following error (where program is `pip3`, `python3`, `git` or `dot`), please contact me for help:

```
-bash: <program>: command not found
```

1.1.2 Linux

First, install the required dependencies

```
sudo apt-get install -y python3-pip libatlas-base-dev git graphviz
```

1.2 Setting up the Compilation Toolchain

First, we need to download the compilation toolchain and make sure we have selected the `master` branch:

```
git clone git@github.com:sendyne/legno-compiler.git
git checkout master
```

This should create a `legno-compiler` directory; inside this directory is the legno compiler toolchain.

1.2.1 Setting up the Compiler

The compiler is a pure python program, so almost all of the dependences can be installed using the python package manager, `pip`. Execute the following command from the root directory of the `legno-compiler` project to execute all the required packages:

```
pip3 install -r requirements.txt
```

1.3 Setting up the HCDCv2 Device

These steps are only necessary if you wish to execute the generated programs on the HCDCv2 device. To write the device firmware to the Arduino Due microcontroller attached to the SA100ASY development board, you must install the Arduino IDE. To install the Arduino IDE, download the executable from the following link and drag it do your `Applications` directory:

```
https://www.arduino.cc/en/Main/Software
```

After installing the Arduino IDE, navigate to `Tools/Board: <Board Name>` in the menu, and select the `Boards Manager` option. After doing so, search for `Due` and install the `Arduino Sam Boards (32 bits ARM-Cortex M3` package. We installed `1.6.12` of the `Sam` board firmware.

You also want to install the `DueTimer` package. This can be done by navigating to the `Sketch/Include Library/Manage Libraries` option in the Arduino IDE and searching for the `DueTimer` package. We installed version `1.4.7` of `DueTimer`.

Linux Command

Download `arduino.tar.xz` from the above arduino website and run the following to install and open the Arduino IDE:

```
tar -xvf arduino-1.8.12-linux64.tar.xz
cd arduino-1.8.12-linux64
./arduino-linux-setup.sh $USER
sudo ./install.sh
arduino
```

1.3.1 Installing Arduino-Make

The firmware is built using a Makefile that extends a specialized set Arduino Makefiles. The Arduino makefiles are provided in the `arduino-mk` package.

MacOS Command

The following installs the arduino makefiles on OSX:

```
brew tap sudar/arduino-mk
brew install --HEAD arduino-mk
```

Linux Command

The following installs the arduino Makefiles on linux. Note that the package-manager sometimes has an outdated version of `arduino-mk` that is missing `Sam.mk`. Please refer to the troubleshooting section below if this is the case to install from source.

```
sudo apt-get install -y arduino-mk
```

1.3.2 Installing the Grendel Runtime Dependences

The firmware communicates with a runtime which is also written in python. Execute the following command from the root directory of the `legno-compiler` project to install all the required packages.

```
pip3 install -r requirements.txt
```

1.3.3 Writing Firmware to the HCDCv2 Device

After following these steps, it should be possible to flash the HCDCv2 firmware to the analog device. Navigate to the following directory:

```
legno-compiler/lab_bench/arduino/grendel_interp
```

To build the firmware, type the following command:

```
make
```

This should complete without incident.

[Linux] `Sam.mk` not found

Some linux package managers reference an outdated version of `arduino-mk` that does not include the `Sam.mk` makefile. In these cases `arduino-mk` must be installed from source. To do so, first clone the following repository:

```
git clone git@github.com:sudar/Arduino-Makefile.git
sudo mkdir -p /usr/local/opt/arduino-mk/
sudo mv Arduino-Makefile/* /usr/local/opt/arduino-mk/
```

We tested the firmware with version `1.6.0` of `arduino-mk`. This version of `arduino-mk` includes version `1.6.12` of the SAM libraries.

[Linux] `ARDUINO_DIR` not defined

Sometimes, the arduino environment variables don't get set up properly. Add the following to your `.profile` or `.bashrc` file. Execute `whereis arduino` to figure out the directory to assign `ARDUINO_DIR` to (it is `/usr/local/bin` for me).

```
export ARDUINO_DIR=/usr/local/bin/
export ARDMK_DIR=/usr/local/opt/arduino-mk/
export AVR_TOOLS_DIR=/usr/include
```

[Linux] `DueTimer` library not found

On some linux distributions, the installed libraries are stored in a different place than the device firmware. Execute the following command to move the `DueTimer` library to the location derived by `arduino-mk` (the `ARDUINO_PLATFORM_LIB_PATH` directory).

```
mv ~/Arduino/libraries/DueTimer/ \
    ~/.arduino15/packages/arduino/hardware/sam/1.6.12/libraries/
```

1.3.4 Writing Firmware to the Arduino Due

Next we will upload the firmware to the device. Please ensure the device is programmed via USB, and that the USB cable is plugged into the programming port of the device. This is the port closest to the DC power port. After connecting the device to the computer, type the following command to flash the firmware to the device:

```
make upload
```

When you're done, navigate back to the `legno-compiler` directory.

```
legno-compiler/
```

[Linux] Permission denied: `'/dev/ttyACM0'`

This occurs on linux installations where the current user doesn't belong to the `dialout` group. Execute the following command to add yourself to the necessary group:

```
usermod -a -G dialout $USER
```

After executing this command, you may need to log out and log back in again. If the command worked successfully, you should be able to read from `/dev/ttyACM0` without getting a permission denied error. Verify this by typing in the following command:

```
cat /dev/ttyACM0
```

[Linux] SAM-BA Operation Failed

If you get the following error, there is issue with serial write operations. This usually happens if the microusb connector is plugged into the wrong port on the microcontroller. Verify that the microcontroller is properly setup before continuing.

1.3.5 Setting up the Sigilent 1020XE Oscilloscope

This toolchain supports collecting signals with either the microcontroller or the the Sigilent 1020XE Oscilloscope. To use the oscilloscope, make the following connections between the oscilloscope and the board:

1. Connect the first channel to pin A0 of the analog device (positive channel of chip[0,3,2])
2. Connect the second channel to pin A1 of the analog device (negative channel of chip [0,3,2]).
3. Connect the EXT lead to pin 25/SDA1 of the Arduino. The Arduino triggers the oscilloscope to record data.
4. Connect the probe ground to the board ground.

Next, we have to connect the oscilloscope to the network. First, connect the oscilloscope to an Ethernet jack (the port is in the back of the device). We also need the IP address of the device. To retrieve the IP address, press the `Utility` button on the Oscilloscope control panel. This should change the menu at the bottom of the screen. Navigate to page 2 of the menu, select the `IO` option, and then select the `IP Set` option. This should display the IP address on the oscilloscope's screen. Write it down and check you can reach the oscilloscope from your computer using the following command:

```
ping <oscilloscope-ip>
```

Trigger Pin (SDA1): While this pin is typically used to trigger the oscilloscope, it can be used to trigger other external peripheral devices (such as external signal generators) as well.

1.4 Configuring the Legno Toolchain

The `Legno` toolchain must first be configured before it can be used. First, copy the reference configuration:

```
cp util/config_local.py util/config.py
```

The following fields may be adjusted as needed in the config file. Note that the only field that should definitely be changed is the `OSC_IP` field:

1. `OSC_IP`: populate this field with the IP address of the oscilloscope. If you are not using an oscilloscope, you can leave this field as is. Refer to Section 1.3.5 for instructions on how to get the IP address of the oscilloscope.
2. `DEVSTATE_PATH` (default recommended): set this field to the directory where all the device data (calibration information, analog block models, etc) should be stored. We recommend leaving this field unchanged.
3. `OUTPUT_PATH` (default recommended): set this field to the directory where all the compiler outputs should be stored. We recommend leaving this field unchanged.

The remaining fields are automatically populated from these three fields. We don't recommend directly setting any of these. The `ARDUINO_FILE_DESC` field points to the USB port that is linked to the Arduino and is automatically populated by looking for the proper file descriptor in the `/dev/` directory.

1.4.1 Setting up the Calibration, Delta Model, and Profiling Databases

Next, we want to unpack the characterization data that was collected during chip testing. First, locate the tag that was shipped with the HCDCv2 board – this is the model number of the board. For example, the tag may indicate the board has the model number `<c3>`.
Important: This manual will use `<model-number>` to indicate where the model of the board should be filled in when dispatching commands. Whenever you see the `<model-number>` argument, replace the argument with the model number: `c4`, for example.
Getting the Data: Contact Sendyne and ask for the characterization and benchmark data for the board you have. Provide them with the model number. They will provide you with the device state archive `<model-number>-devstate.zip` and benchmarks archive `<model-number>-bmarks.zip`.

The device state archive contains all of the characterization data collected from the chip. The benchmark archive contains the end-to-end application executions from the chip on hand. Note that this data can be automatically rebuilt from scratch if the archives cannot be located.

Unpacking and Analyzing the Data

Execute the following command to unpack the data:

```
python3 scripts/unpack_char_data.py <model-number>
```

This will unpack the relevent databases to the `device-state/hcdc/<model-number>` directory and the benchmark executions to the 'outputs/legno/unrestricted' directory. Execute the following command to visualize all of the characterized data.

```
python3 grendel.py vis <model-number>
```

The benchmark execution data can be viewed without any special processing. Simply navigate to the `outputs/legno/unrestricted` directory and peruse the `lgraph-vis` and `lscale-vis` for visualizations of the generated unscaled and scaled circuits, respectively.
Reproducing Benchmark Data: It is trivial to re-execute the unscaled and scaled adps. To reproduce the benchmark `<bmark>`, remove the following directories:

```
rm outputs/legno/unrestricted/<bmark>/out-waveform*
rm outputs/legno/unrestricted/<bmark>/plots*
```

Removing these directories clears out the stored waveforms. Note that these directories can be repopulated by re-running the unpacking script. After removing these directories, you can re-execute the benchmark applications with the following commands:

```
python3 legno.py lexec <bmark>
python3 legno.py lwav <bmark> --summary-plots --individual-plots
```

The first command re-runs the adps on the HCDCv2 device and records the emitted waveforms to the `out-waveforms` directory. The second command processes the collected waveforms and plots the data – the plotted data is written to the `plots` directory.

1.5 Troubleshooting

The following section provides fixes for common errors that occur during installation.

1.5.1 `brew link` when installing `python3`

If you get the following error trying to install `python3` using brew:

```
Linking /usr/local/Cellar/python/3.7.5... Error: Permission denied @ dir_s_mkdir
- /usr/local/Frameworks
```

Execute the following commands to fix it:

```
sudo mkdir /usr/local/Frameworks
sudo chown -R \$(whoami) /usr/local/Frameworks/
brew link python
brew postinstall python3
```

1.5.2 `pyserial` missing when making firmware

If you get the following error when trying to compile the firmware:

```
import failed, serial not found
```

This is likely because `arduino-mk` is using a different version of python than the rest of the project. To mitigate this, install pyserial for `python2`:

```
pip install pyserial
```

1.5.3 ARDUINO_DIR not defined when making firmware

If you get the following error while trying to compile the firmware:

```
/usr/share/arduino/Arduino.mk:279: *** "ARDUINO_DIR is not defined". Stop.
```

This is likely because the Arduino IDE is not properly installed. Please make sure that the Arduino IDE is in the `Applications` folder in OSX.

Chapter 2

Legno Quickstart

The following quickstart walks through compiling a dynamical system program that implements the cosine (`cos`) function. This section requires you know the model number of your board. Refer to the tag included with your board for this information.

2.1 Compiling the `cos` Program

The following section describes how to compile the cosine program. We use the `legno.py` script to compile the program. This script is invoked multiple times to produce a scaled analog device program (`adp`). The first step of compilation is circuit synthesis. This step assembles together a circuit of analog blocks which implements the provided program. We will be generating one unscaled circuit that models the cosine function with the `lgraph` compilation pass:

```
python3 legno.py lgraph --adps 1 cos
```

This command generates one unscaled circuit (`--adps 1`) which implements the cosine function. This circuit cannot be directly executed on the analog device because it may contain constant values and signals which lie outside of the operating range restrictions of the device. For this reason, we need to scale the unscaled circuit. We will be generating ten scaled circuits that respect the hardware restrictions with the `lscale` pass:

```
python3 legno.py lscale --scale-adps 10 --calib-obj minimize_error
--scale-method ideal --objective qty --model-number <model-number> cos
```

This command generates 10 scaled circuits (`--scale-adps 10`) from each unscaled circuit. Each of these scaled circuits take into account the operating range restrictions in the device. The `--objective qty` argument tells the compiler to maximize the best-case signal-to-noise ratio on all the wires.

This invocation of `lscale` assumes all blocks implement the promised functions in the hardware specification and are not subject to manufacturing variations (`--scale-method ideal`). The resulting scaled circuit is agnostic to the underlying board and calibration objective. Note that we still tell the compiler which board we're targeting (`--model-number <model-number>`) and how this board is calibrated (`--calib-obj minimize_error`) – this information is needed for execution

and therefore included in the scaled circuit. Refer to Section **??** for details on circuit calibration and calibration objectives.

`lscale` also supports compensating for block-specific manufacturing variations which were not eliminated during calibration. The following command takes into account manufacturing variations when scaling the circuit to target the board `<model-number>` when it has been calibrated with the `maximize_fit` calibration objective:

```
python3 legno.py lscale --scale-adps 10 --calib-obj maximize_fit
--scale-method phys --objective qty --model-number <model-number> cos
```

The above invocation generates 10 scaled circuits which take into account the manufacturing variations present in the board (`--scale-method phys`). It is directed to compensate for the manufacturing variations which exist when the board is calibrated with the `maximize_fit` objective (`--calib-obj maximize_fit`).

It is likely the above command will complain that it's missing empirical models and fail. This scaling approach requires the blocks which are in use be calibrated and characterized. It uses the characterization data to build models which capture the behavioral deviations that it needs to compensate for. Execute the command below to characterize any outstanding blocks.

```
python3 legno.py lcal cos --model-number <model-number>
```

Note that this may take several hours. Once it is finished, repeat the above `lscale` command once it is finished.

2.1.1 Inspecting the Compilation Outputs

The `legno.py` script produces unscaled circuits, scaled circuits and grendel scripts. All compilation and execution outputs for the cosine program is stored in the following directory:

```
legno-compiler/outputs/legno/unrestricted/cos/
```

We call this the *program directory*. The following sub-directories in the program directory contain compilation outputs:

- `lgraph-adp` and `lgraph-diag`: These directories contain the unscaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging.
- `lscale-adp` and `lscale-diag`: These directories contain the scaled analog device programs and associated diagrams.
- `out-waveforms`: This directory contains the waveforms collected from the HCDCv2 board.
- `plots`: This directory contains visualizations of the collected waveforms from the HCDCv2 board. The `wav` subdirectory contains HCDCv2 waveform visualizations. Software simulation visualizations are written to the `sim` subdirectory.
- `times`: this directory contains the runtime information for each pass.

We should see one `.adp` file in the `lgraph-adp` directories, and one `.gv.pdf`/`.gv` file in the `lgraph-diag` directory. These files should have the following naming scheme:

```
cos_g0.adp
```

The `g0` identifier indicates this was the first unscaled circuit generated by the `lgraph` pass. We should see ten `.adp` files in the `lscale-adp` directory and ten `.gv.pdf`/`.gv` files in the `lscale-diag` directory. These files should have the following naming scheme:

```
cos_g0_s1_ideal_qty_minerr_c1.adp
```

The `g0` identifier indicates this scaled circuit was derived from the unscaled circuit with the `g0` identifier. The `s1` identifier indicates this was the second scaled circuit generated from that particular unscaled circuit. The `qty` identifier indicates the generated circuit maximizes signal quality. The `ideal` identifier indicates the scaled circuit was generated with the `--scale-method ideal` flag. The `c1` identifier indicates this scaled circuit targets the board with model number `c1`.

2.2 Running the Cosine Scaled ADPs

The `lexec` subcommand in the legno compiler (`legno.py`) executes all generated scaled circuits on the analog hardware:

```
python3 legno.py lexec cos --model-number <model-number>
```

This command may complain that it can't find the calibration codes for a particular block. To rectify this, calibrate the chip with the following command:

```
python3 legno.py lcal cos --model-number <model-number>
```

Note that this may take several hours. Once it is finished, repeat the above `lexec` command once it is finished.

If the `lexec` command executed correctly, you should see a `.json` file appear in the `out-waveforms` directory. This file is the waveform downloaded from the microcontroller:

```
legno-compiler/outputs/legno/unrestricted/cos/out-waveform/
```

2.2.1 Execution with the Sigilent 1020XE Oscilloscope

:

If you are using the Sigilent 1020XE oscilloscope, include the `--osc` flag to `lexec` commands to tell the runtime from attempting to communicate with the measurement device. If you would like to use an unsupported oscilloscope, you will need to modify the `llcmd_sim.py` file so that it configures the voltage, and time divisions on the oscilloscope, waits for an edge trigger, and retrieves the waveforms from the oscilloscope.

If the script executed correctly, you should see a cosine waveform on your oscilloscope.

2.3 Analyzing Waveform Data

The `lwav` subcommand is used to analyze and render oscilloscope outputs. We execute the following command to produce visualizations from the collected waveforms:

```
python3 legno.py lwav cos
```

This command generates plots for each measured signal and then attempts to align the measured signal with the reference waveform. It inverts the scaling transform (time and voltage scale coefficients) to recover the original dynamics. The oscilloscope waveforms and visual analysis results are written to the `plots/wav` subdirectory in the program directory:

```
outputs/legno/unrestricted/cos/plots/wav
```

2.4 Debugging Analog Device Programs

You may observe dynamics which deviate significantly from the expected dynamics. This may occur for a few reasons:

- **Incorrect Dyanamical System Program / Analog Device Program**: You may have mis-specified the dynamical system or found an issue with the graph synthesis or scaling procedure. Refer to the *Troubleshooting ADPs and Dynamical System Programs* section for hints on how to pinpoint the issue.
- **Uncompensated Manufacturing Variations**: The ADP may have been configured to ignore manufacturing variations or insufficiently compensates for the manufacturing variations present in the circuit. Not all manufacturing variations can be compensated for in compilation. Refer to Section **??** for information on how to investigate the manufactuirng variations present in the hardware.
- **Unmodelled Behavior**: Some behaviors (e.g. frequency-gain responses) are not characterized in the device. These behaviors are not captured in the hardware specification or in the empirically derived hardware models. Please contact the developers if you suspect there's unmodelled behavior – some of these behaviors can be approximated by modifying the hardware specification.

We recommend generating multiple scaled circuits and running them all then selecting best performing circuit for further use.

2.4.1 Troubleshooting ADPs and Dynamical System Programs

`legno` supports simulating dynamical system programs and all ADPs in software. Software simulations are useful for identifying issues with the compiler. The following command simulates the `cos` dynamical system program:

```
python3 legno.py lsim cos --reference
```

It produces a single plot which contains all the program variables in the `plots/sim` directory. This plot has a `REF` identifier in its file name. We can simulate the circuit emitted by `lgraph` pass with the following command:

```
python3 legno.py lsim cos --unscaled
```

The following command simulates all the unscaled `cos` ADPs emitted by `lgraph` pass and writes a single plot (for each `.adp`) containing visualizations of the trajectories to the `plots/sim` directory. We can finally simulate circuits emitted by the `lscale` pass with the following command:

```
python3 legno.py lsim cos
```

This again will will simulate all the scaled `cos` ADPs emitted by the `lscale` pass and writes a single plot (for each scaled `adp`) containing visualizations of the trajectories to the `plots/sim` directory.

Simulation Accuracy: Note that the above simulation is rudimentary and does not use the board-specific empirical models or impose operating range restrictions when executing the circuit. It is therefore unwise to inspect circuit simulations for adps compiled with the `phys` scale method.

Chapter 3

Dynamical System Language

The `Legno` compiler (`legno.py`) enables developers to compile dynamical systems down to configurations for the analog hardware. The `Legno` compiler requires that dynamical systems be specified in the `dynamical system language`, a high level language that supports writing first-order differential equations.

3.1 Example: Cosine Program

The following dynamical system program implements the cosine function. This program can be found in `progs/quickstart/cos.py`. In order for `legno.py` to find the program, the program must be placed in the `progs/` directory.:

```
from dslang.dsprog import DSProg
from dslang.dssim import DSSim,DSInfo

def dsname():
  return "cos"

def dsinfo():
  return DSInfo(name=dsname(), \
                desc="cosine",
                meas="signal",
                units="signal")
  info.nonlinear = False
  return info

def dsprog(prob):
  params = {
    'P0': 1.0,
    'V0' :0.0
  }
```

```
    prob.decl_stvar("V","(-P)","{V0}",params)
    prob.decl_stvar("P","V","{P0}",params)
    prob.emit("P","Position")
    prob.interval("P",-1.0,1.0)
    prob.interval("V",-1.0,1.0)
    prob.check()
    return prob

  def dssim():
    exp = DSSim('t20')
    exp.set_sim_time(20)
    return exp
```

A dynamical system program must define four python functions in order to be used by the compiler:

- **dsname**: This function returns the name of the program. The user can later refer to this program by its name during compilation.
- **dsinfo**: This function returns detailed information about the program. This includes a short description of the program, the name of the signal being measured and the units of the signal being measured. Only one signal can be measured right now.
- **dsprog**: This function returns the dynamical system associated with the program. The dynamical system is defined as a collection of state variable and function declarations. The dynamical system also contains interval annotations for each state variable (this defines the bounds for each variable) and which variable is measured. The `check()` function checks to see that the system is bounded.
- **dssim**: This function returns the simulation parameters. This program is to be run for 20 simulation units.

3.1.1 Breaking Down `dsprog`

In this dynamical system, the position `P` corresponds to the amplitude of the cosine function. The V and P variables are internal variables that are used to model the dynamics of the system. We describe each line of the program below:

- `params = {..}`: This statement creates a parameter map. The parameter map associates names with values. This is used to fill in parameters when defining the dynamical system. Parameters are referred to by their name, enclosed in curly braces (for example `\{V0\}` refers to the parameter V0).
- `prob.decl_stvar("V","(-P)","{V0}",params)`: This statement defines a state variable V, whose dynamics are governed by $V' = -P$. The initial value of the state variable is the value of parameter V0. The last argument is the parameter map to use.
- `prob.decl_stvar("P","V","{P0}",params)`: This statement defines a state variable P, whose dynamics are governed by $P' = V$. The initial value of the state variable is the value of parameter P0. The last argument is the parameter map to use.

- `prob.emit("P","Position")`: This statement indicates the expression P is of interest, and should be observed. The observation is named `Position`.
- `prob.interval("P",-1.0,1.0)`: This statement defines the variable `P` as falling within the bounds [-1,1]. All state variables must be restricted to a user-defined interval in order for the program to pass the well formedness check (`check()` invocation).
- `prob.interval("V",-1.0,1.0)`: This statement defines the variable `V` as falling within the bounds[-1,1].
- `prob.check()`: This statement checks that all the variables in the dynamical system are bounded. This function must pass (not trigger an error) in order for the program to compile successfully.

3.1.2 Executing the `cos` Dynamical System with a ODE Solver

The `Legno` compiler supports digitally simulating dynamical systems for testing purposes.

```
python3 legno.py --subset extended simulate cos --reference
```

The `--reference` argument tells the compiler to perform a digital simulation and produce reference figures. To profile the runtime of the reference simulation, add the `--runtime` argument.

Note that the `--subset` argument is used to determine where to write the reference simulation – it does not impact the behavior of the simulation. The digital simulator writes the plots of the state variable trajectories to the following directory:

```
legno-compiler/outputs/legno/extended/cos/sim/ref/
```

3.2 Creating your own Dynamical System

The following section walks through how to create a dynamical system named `deg` that implements exponential decay. First, copy the dynamical system template to a new file. This file will automatically be picked up by the compiler:

```
cp progs/template.py progs/deg.py
```

Next we modify the contents of `test.py` to implement the desired dynamical system. First we change the name of the program to `decay`:

```
def dsname():
   return "decay"
```

Next, we implement the dynamical system by populating the `dsprog` function. The exponential decay function is implemented as $x' = -k \cdot x$, where k is the decay parameter. The dynamical system takes one additional parameter, $x(0)$, the initial condition of the system. We are interested in executing the dynamical system with $k = 0.1$ and $x(0) = 10$. We define the following parameter dictionary:

```
params = {'k':0.1,'x0':10.0}
```

This system has one state variable, x. We next define the dynamics of x with the statement below.

```
prog.decl_stvar("x","{k}*(-x)","{x0}",params)
```

This statement creates a new state variable named `x`, whose derivative is `k*(-x)` (`k` is a parameter) and initial condition is the parameter `x0`. Wrapping a variable in curly braces indicates it's a parameter – the compiler will look for the parameter in the parameter map that is passed in. The last argument, `params`, provides the parameter map to the compiler.

We are interested in observing this state variable; we tell the compiler we would like to forward this variable to an external pin for observation with the `emit` command. This signal will be named `OBS` in any generated plots:

```
prog.emit('x','OBS')
```

We also need to provide an interval range annotation for x. This annotation tells the compiler the range of values x may take on. It is important to set an appropriate dynamic range for each variable. If this range is smaller than the actual dynamic range of x, it may saturate. If this range is much larger than the actual dynamic range, then x might be overtaken by noise and error during execution.

We know that x starts at 10.0 and decays to 0.0. It therefore always falls within $[0, 10.0]$. We set the interval of x with the following command:

```
prog.interval('x',0,10.0)
```

The interval of OBS is automatically derived from x. The full `dsprog` function is shown below:

```
def dsprog(prog):
  params = {'k':0.1,'x0':10.0}
  prog.decl_stvar("x","-{k}*x","{x0}",params)
  prog.emit('x','OBS')
  prog.interval('x',0,10.0)
  prog.check()
  return
```

We next need to provide the simulation parameters of the system by changing the `dssim` function. We are fine with executing the simulation for 20 simulation units, so we can leave the `dssim` function as is:

```
def dssim():
  exp = DSSim('t20')
  exp.set_sim_time(20)
  return exp
```

Next, we change the program information to reflect the exponential decay system. The `decay` system is a linear system that tracks the trajectory of x, which is measured in arbitrary units.

```
def dsinfo():
  info = DSInfo(dsname(), \
                desc="exponential decay"
                meas="trajectory",
                units="units")
  info.nonlinear = False
  return info
```

We can test the dynamical system by digitally simulating it using the command below:

```
python3 lsim.py decay --reference
```

The command will write a plot with a `REF` identifier containing the trajectories of `x` and `OBS` to the following directory:

```
legno-compiler/outputs/legno/extended/decay/plots/sim/
```

3.3 Troubleshooting

3.3.1 `handle not in interval` when executing `check()`

If you get the following error:

```
Exception: handle not in interval: :h0
```

This indicates that the compiler could not compute intervals for all the variables in the dynamical system. Please make sure all state variables and functions (optionally, usually) are bounded using `interval` commands.

3.3.2 `Legno` cannot find my dynamical system program!

Make sure that (1) the program is a `.py` file that resides somewhere in the `legno-compiler/prog` directory (2) the `dssim`, `dsinfo`, `dsname` and `dsprog` functions are defined in the file.

Chapter 4

Compiler Overview

The compilation toolchain contains five major components:

- **Compiler** (`legno.py`): The compiler. It reads dynamical system programs (in the `prog` directory) and generates analog device programs. These programs can be executed by the runtime. The `legno.py` script also offers several convenience functions (which wrap the runtime) for running, simulating, and visualizing ADPs and characterizing the device.
- **Runtime** (`grendel.py` and `meta_grendel.py`): The runtime. It executes ADPs on the analog device by dispatching commands to the microcontroller. The core runtime (`grendel.py`) also offers calibration, profiling, and model elicitation subcommands which calibrate and characterize the board and build empirical models which capture the behavior of the board. These empirical models are used in the `lscale` pass of the compiler. The meta-runtime (`meta_grendel.py`) uses the core runtime to test the board, identify and characterize poorly performing blocks.
- **Firmware** (`lab_bench` directory): The firmware. The bare-metal grendel interpreter (`lab_bench/arduino/grendel_i` must be written to the microcontroller in order for the microcontroller to understand the grendel commands. The `lab_bench` directory also contains some host-side libraries for interacting with the microcontroller.

4.1 Legno Compiler

The `Legno` compiler (`legno.py`) enables developers to compile dynamical systems down to configurations for the analog hardware. The `Legno` compiler requires that dynamical systems be specified in the `dynamical system language`, a high level language that supports writing first-order differential equations. The `Legno` compiler also accepts a specification of the target analog device, described using the Analog Device API. The `Legno` compiler generates an *analog device program* (`adp`) which implements the target dynamical system on the specified analog device. The compiler is broken up into three passes:

- `LGraph`: This pass synthesizes an unscaled circuit that implements the specified dynamical system.

- `LScale`: This pass synthesizes one or more scaled circuits for each unscaled circuit generated by the `LGraph` pass.

The `Legno` compiler works with the `unrestricted` subset of block features. This limits the accepted modes of each block to the high (-20 uA to 20 uA) and medium (-2 uA to 2 uA) modes. These are the modes which have been most thoroughly tested by us. Note that the HCDCv2 also offers low modes (`0.2`) – these are untested and therefore unsupported by the compiler.

The `Legno` compiler writes all compilation outputs for some program `<prog>` compiled with the feature subset `<subset>` to the following directory:

```
outputs/legno/<subset>/<program>/
```

This directory has eight subdirectories:

- `lgraph-adp` and `lgraph-diag`: These directories contain the unscaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging. These files are produced by the `lgraph` pass of the `Legno` compiler (Section `??`).
- `lscale-adp` and `lscale-diag`: These directories contain the scaled analog device programs and associated diagrams. The diagrams are visual representations of the programs that are useful for debugging. These files are produced by the `lscale` pass of the `Legno` compiler (Section `??`), and are derived from files in the `lgraph-adp` directory.
- `times`: this directory contains the runtime information for each compilation pass.
- `sim`: this directory contains digital simulation results for the program. These results are generated by the `simulate` operation of the `Legno` compiler (see Section 3)
- `out-waveform`: this directory contains the output waveforms collected from the HCDCv2 during execution. This directory is populated by the grendel runtime (`grendel.py`) or by the legno compiler's `lexec` subroutine which invokes the grendel runtime.
- `plots`: this directory contains visualizations of the output waveforms collected by the HCDCv2 chip. These plots are generated by legno's waveform analysis (`lwav`) command and re are written to the `plots/wav` subdirectory. This directory may also contain software simulation plots. The software simulation plots are generated by legnos simulation (`lsim`) command.

4.2 Runtime System

The `Grendel` runtime executes analog device programs and calibrates and characterizes the board. It maintains all of the calibration information, profiling data, and empirically derived models (which capture the manufacutring variations) in the *device state database*. The device state database is located at the following location:

```
device-state/hcdcv2/<model-number>/hcdcv2-<model-number>.db
```

This database maintains two important tables – the delta model table and the profiling data table. The delta model table stores empirically derived symbolic models for each calibration strategy and the calibration information for each calibration strategy. The profiling data table maintains the

datasets which capture the behavior of each calibrated block. The firmware and grendel runtime offer several calibration strategies which inform how the block is calibrated:

- **Maximize Fit** (`minimize_error`,`minerr`): This calibration strategy allows for analog blocks to have variations in the gain they introduce. It optimizes for eliminating any constant offsets (biases) and eliminating any hard-to-characterize behavior (point errors). This calibration strategy is executed on the device firmware using a heuristic search.
- **Minimize Error** (`maximize_fit`,`maxfit`): This calibration strategy calibrates analog blocks to deliver behavior that closely follows the behavior in the hardware specification. This calibration strategy is executed on the device firmware using a heuristic search.
- **Bruteforce** (`bruteforce`,`brute`): This calibration strategy randomly samples the space of calibration codes and chooses the best one. This strategy is performed on the host-side and is usually reserved for blocks which perform poorly when calibrated with the previous two strategy.
- **Best** (`best`): This calibration strategy chooses the best performing block calibration strategy.

We will walk through the various operations which can be performed with the `Grendel` runtime in this section.
Advice on device state database: The device state database should be backed up regularly as it caches a great deal of calibration and profiling information which is time-intensive to recompute. Note that this database is not irreplaceable – it can easily be repopulated if it is lost with a few commands. We may periodically ask for end-users to submit their device state database as it provides valuable information on how the board is functioning and can help.

4.2.1 ADP Execution

The primary function of the `Grendel` runtime is to execute analog device programs. The `Grendel` runtime dispatches scaled `.adps` to the analog device and writes any collected waveforms to the following directory:

```
outputs/legno/<subset>/<program>/out-waveform/wav
```

The grendel runtime communicates with a microcontroller running the `Grendel` firmware. This firmware is found in the following directory:

```
legno-compiler/lab_bench/arduino/grendel_interp_V1
```

4.2.2 Calibrating Blocks in an ADP

Calibration is the first step to deriving the empirical models for each block. The grendel runtime supports invoking the on-board firmware calibration routines for the blocks in a target analog device program. Only the `maximize_Fit`/`minimize_error` calibration strategies may be selected as they are implemented directed in the firmware. Refer to Sections **??** or Sections **??** for information on how to do the bruteforce or best-performing calibration strategies. The following invocation calibrates all the blocks in the file `<file>.adp` using the `maximize_fit` strategy and writes the calibration information to the board's device state database:

```
python3 grendel.py cal --model-number <model-number> <file>.adp maximize_fit
```

The calibration routine takes a long time to complete because it calibrates the block under each possible block configuration. All this information is written to the `hcdc-<model_number>.db` database – if this procedure is interrupted, it will pick up from where it left off.

4.2.3 Profiling Blocks in an ADP

Calibrated blocks can be profiled with the `Grendel` runtime's `prof` subcommand. It accepts as input the analog device program which contains the blocks to profile and the calibration strategy which was used. It looks up the calibration information in the state database and exercises the block over the input space to collect a set of input-output pairs which describe the behavior of the calibrated block. We write an example invocation below:

```
python3 grendel.py prof --model-number <model-number> <file>.adp maximize_fit
--grid-size 17
```

The `--grid-size` argument tells the profiler how many points to break each input into. For example, if a block takes inputs `x` and `y`, the profiler will break up the input space for each input into 17 points. It will therefore try 17^2 test points to evaluate the behavior of the block being tested. The profiler skips over any calibrated blocks which have already been profiled to save time.

This routine also accepts other optional arguments, which we describe below:

- `--min-points` **argument**: This argument tells the runtime how many input-output pairs need to be in the dataset for the dataset to be complete. This argument is used to determine if a block has already been profiled.
- `--force` **argument**: This argument forces the profiler to re-profile every block in the analog device program. This can be useful if the profiling information is somehow incomplete or malformed.
- `--missing` **argument**: This argument tells the profiler to ignore the analog device program and instead calibrate any blocks in the database which are missing profiling data.

4.2.4 Inferring Delta Models from the Profiling Data

The `Grendel` runtime uses the profiling data to construct empirically-derived symbolic models which describe the behavior of each block instance in practice in each programming mode. The model inference algorithm is executed with the following command:

```
python3 grendel.py mkdeltas --model-number <model-number>
```

This command fits the profiling data in the state database to the delta model specification from the hardware specification. It updates the delta models in the database to include the fitted models. The `mkdeltas` subcommand skips over any already computed delta models – it can be made to recompute all delta models with the `--force` command. The `mkdeltas` subcommand can also be forced to skip over any profiling datasets which don't have enough points with the `--min-points` argument.

4.2.5 Calibrating, Profiling, and Modelling Blocks with `lcal`

The `Legno` compiler offers a convenience command for performing calibration, profiling, and delta model elicitation in one step. Simply run the following command to calibrate, profile, and infer models for all ADPs which implement a particular program `<prog>`:

```
python3 legno.py lcal <prog> --model-number <model_number>
```

The above command will calibrate and profile all blocks in the ADPs with both the `minimize_error` and `maximize_fit` calibration strategies.

4.2.6 Visualizing Block Behavior

The `Grendel` runtime offers a command for visualizing the errors associated with the profiled blocks. The following command populates the `device-state/hcdcv2/<model-number>/viz` directory with heatmaps detailing the percent error of each block over the input space:

```
python3 grendel.py vis --model-number <model-number> --histogram <calibrate_objective>
```

Because the visualization operation is time consuming, it requires the end user specify which calibration objective (`minimize_error`,`maximize_fit`,`best`,`brute`) to investigate. The `--histogram` argument tells the visualization algorithm to render histograms showing the distribution of errors across blocks.

4.2.7 Characterizing Badly Behaving Blocks and Bruteforce Calibration

Some blocks are badly behaved – we often want to take a closer look at these blocks to better understand why they are failing and potentially get a better calibration setting for them. The grendel runtime offers a convenience function for identifying and characterizing these blocks:

4.2.8 Selecting the Best Calibration Strategy

4.2.9 Full-Board Testing

Bibliography