COLUMBIA HYBRID COMPUTER USER'S GUIDE v10

(DRAFT)

By Ning Guo,
Yipeng Huang (yipeng@cs.columbia.edu),
and Ke Lei

Columbia University

October 31, 2017

DISCLAIMER

BY USING THIS MANUAL AND EQUIPMENT DESCRIBED HEREIN, YOU ACKNOWLEDGE THAT YOU HAVE READ ALL OF THE TERMS AND CONDITIONS AS SET FORTH BELOW, UNDERSTAND THEM, AND AGREE TO BE BOUND BY THEM. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS AS SET FORTH BELOW, YOU MUST NOT USE THE MANUAL OR EQUIPMENT.

THIS MANUAL IS STILL UNDER DEVELOPMENT. THIS VERSION, ANY UPDATES OR MODIFICATIONS OF THE MANUAL PROVIDED, AND THE EQUIPMENT DESCRIBED HEREIN, ARE PROVIDED “AS IS” AND WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING ANY GUARANTEE, PROMISE, OR WARRANTY OF CORRECTNESS OR USEFULNESS OF THE RESULTS OBTAINED, OR OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE PROCEDURES OR THE USE OF THE EQUIPMENT DESCRIBED IN THIS MANUAL WILL NOT POSE A HEALTH OR SAFETY RISK. COLUMBIA AND COLUMBIA PROVIDING SCIENTIST SPECIFICALLY DISCLAIM ANY WARRANTY THAT THE PROCEDURES IN THE MANUAL OR THE EQUIPMENT DESCRIBED HEREIN WILL MEET YOUR REQUIREMENTS OR WILL OPERATE IN A MANNER SELECTED FOR USE BY YOU, OR THAT THE OPERATION THEREOF WILL BE UNINTERRUPTED OR ERROR FREE.

THIS MANUAL MAY NOT BE COPIED OR DISTRIBUTED BY THE RECIPIENT TO ANY OTHER PERSON OR ORGANIZATION WITHOUT COLUMBIA’S AND THE COLUMBIA PROVIDER SCIENTIST’S PRIOR WRITTEN APPROVAL, EXCEPT TO THOSE SCIENTISTS AND ENGINEERS AFFILIATED WITH AND WORKING WITH THE RECIPIENT SCIENTIST.

TABLE OF CONTENTS

INTRODUCTION

The 2nd-generation Hybrid Continuous-Discrete Computing (HCDCv2) chip features clockless, continuous-time computation. Eliminating the clock signal during computation implies that there will be no convergence issues in solving differential equations and no aliasing in signal processing. By supporting efficient analog integration and multiplication operations, which are sometimes expensive in digital discrete-time computation, the HCDCv2 chip has a potential to accelerate digital iterative algorithms. For instance, on an implementation that simulates continuous-time systems, e.g. physical systems like a mass-spring damper, the ordinary differential equation (ODE) describing the system need not be discretized in time, thus avoiding intermediate discretization steps. This can potentially improve performance and energy efficiency significantly.

The HCDCv2 chip also features low power computation. With only milliwatt-level power for each chip, we can solve complex math problems, e.g. nonlinear differential equations. This low power feature will hopefully motivate the users to explore energy-efficient computation in emerging applications.

The setup of the HCDCv2 computer board is easy. By connecting a micro USB cable to your laptop, you can also explore the benefits of hybrid computation by using the HCDCv2 board as an accelerator for the digital computer. This user manual gives preliminary instructions to get you started using the HCDCv2 computing board development kit.

Preferred background of HCDC users: Analog circuits, Arduino programming/testing, control systems, numerical methods for ODEs/PDEs. Users with the above backgrounds can explore the speed and energy benefits of hybrid/analog computing. We also recommend that users read some old books and papers on hybrid computing and analog computing for extensive examples.

Attention:

1. Electrostatic charges on the human body may cause damage to the board, so we recommended that users discharge themselves (e.g., by touching a large metal structure) before using the board.
2. If you would like to feed external signals to the board, please double-check that the voltage levels are compatible with the HCDCv2 board (3.3 V for digital I/Os).
3. Please note that the HCDCv2 board is not a commercial product. Although we have fully tested the functionality of all boards and tried our best to make them as robust as possible, you may experience errors during operation. If you are 101% sure that there are issues with the hardware, please contact us.
4. Unless you have the necessary electrical engineering expertise and are very clear of what you are doing, we DO NOT RECOMMEND that users open the front cover of the HCDCv2 board to debug/probe/solder any components. Further, any such work must be pre-approved by the originators.
5. This user manual is for research purposes of groups pre-approved by the originators. It should not be distributed or published online.

CONNECTING THE HCDCV2 BOARD

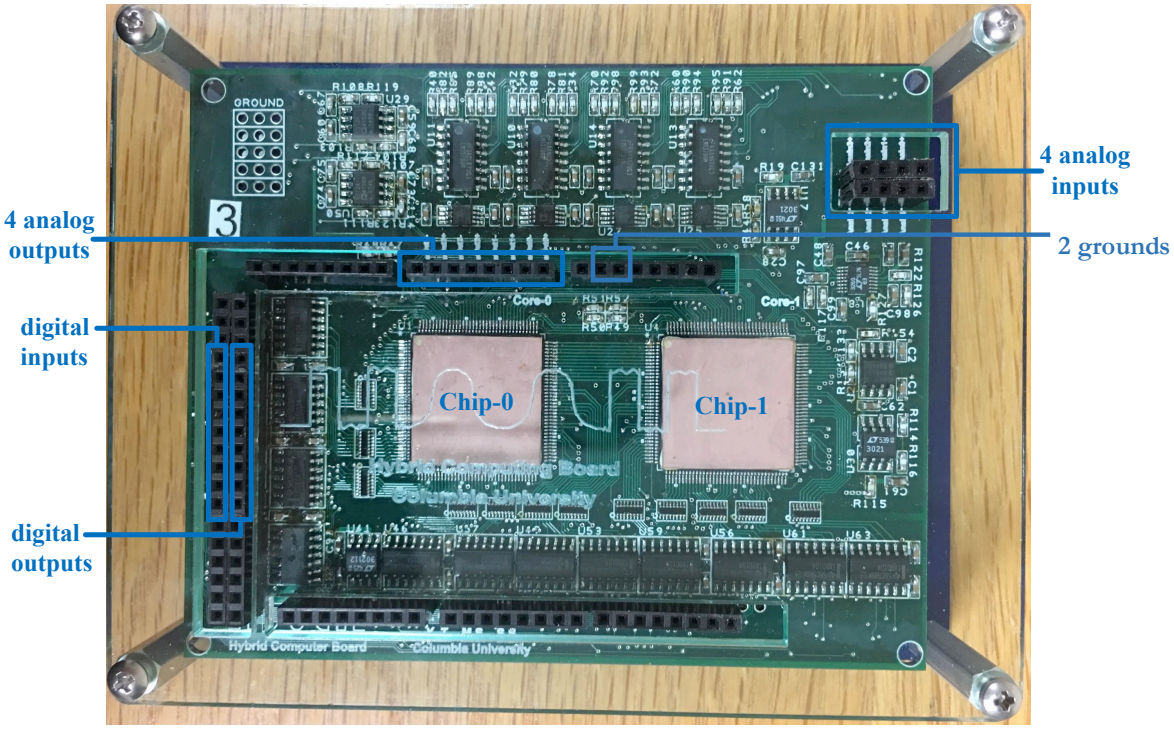


Figure 1: Front look of the HCDCv2 computer board.

Fig. 1 shows the most commonly used I/O interface of the board, i.e. four analog differential inputs, four analog differential outputs, digital inputs (8-bit data + one trigger signal) and digital outputs (8-bit data + one trigger signal). The demo boards are numbered; see, for example, the number "3" here; note this number as boardIndx.

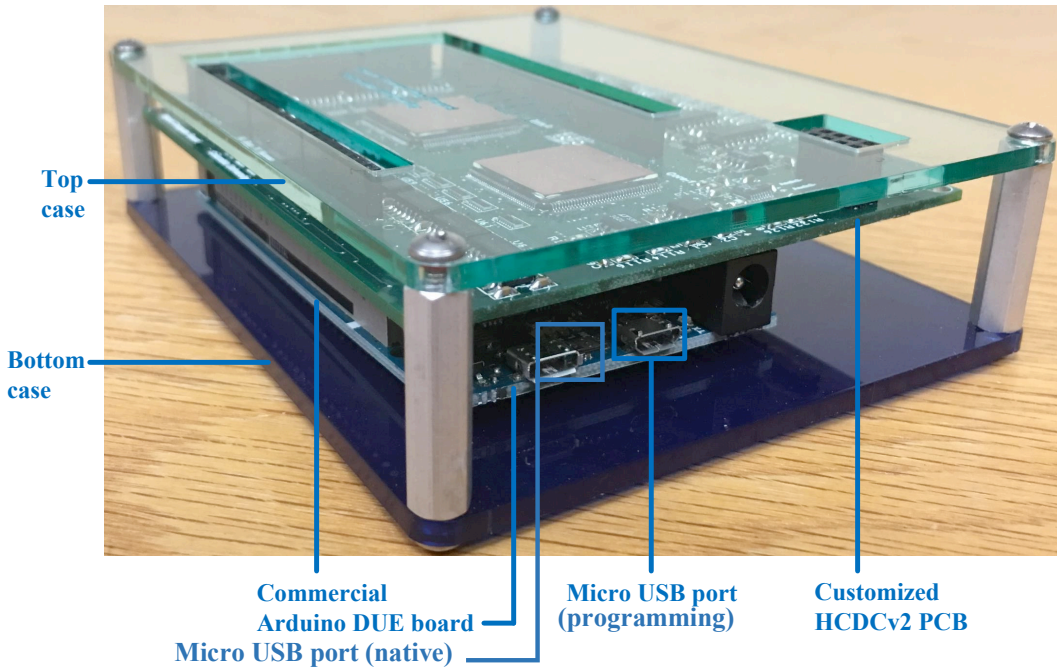


Figure 2: Side look of the HCDCv2 computer board.

Fig. 2 shows the side look of the HCDCv2 computer. It is composed of two boards: the customized HCDCv2 computing board on top and the Arduino Due board underneath. The purpose of the Arduino Due board is to power up, program and calibrate the HCDCv2 board. There is a reset button on the Arduino board, which allows you to manually reset the board.

To setup the connection between the HCDCv2 computer board and your laptop, you need a micro USB cable. Connect the micro USB terminal to one of the Micro USB ports found on the Arduino DUE board, as shown in Fig. 3:

The left port is the native port. The native port has higher communication rate. We recommend you use this port if you are using the Arduino DACs and ADCs for mixed signal conversion. The downside of using the native port is the board may hang if your code has a memory error (e.g., pointer out of bounds). You need to manually reset or use the programming port to reboot if the board is hung. Make sure to correctly set the Arduino IDE to native port if using this port. According to the Arduino documentation, the function to print to the console from this port is **`SerialUSB.print();`**

The right port is the programming port. The programming port has lower communication rate and is useful for rebooting the board. Make sure to correctly set the Arduino IDE to programming port if using this port. According to the Arduino documentation, the function to print to the console from this port is **`Serial.print();`**

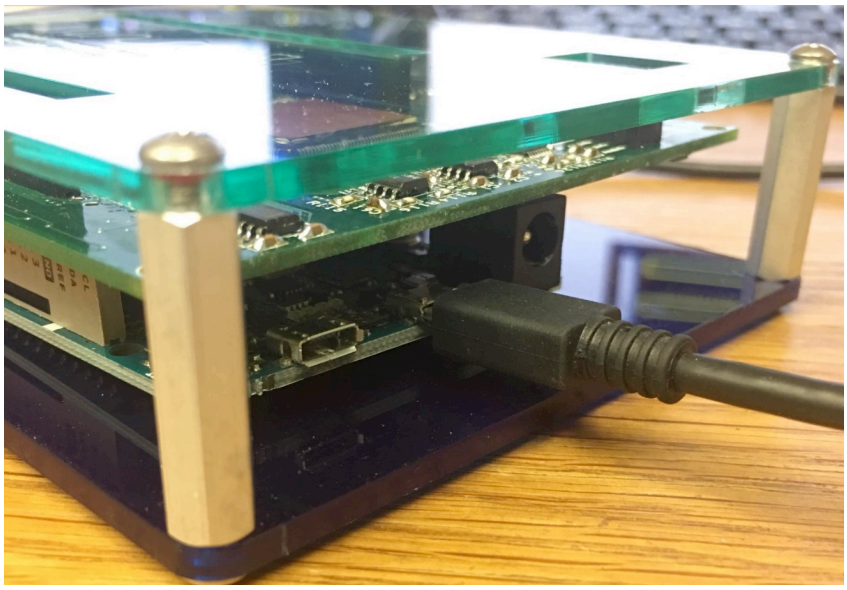


Figure 3: Connecting the micro USB port to HCDCv2 computer.

SETTING UP THE PROGRAMMING ENVIRONMENT

Now you have successfully connected the HCDCv2 computer board to your laptop. Next, install the Arduino IDE software from the Arduino official website (https://www.arduino.cc/en/main/software). Download the right version for your operation system and follow the installation instructions.

LOADING SOFTWARE LIBRARIES

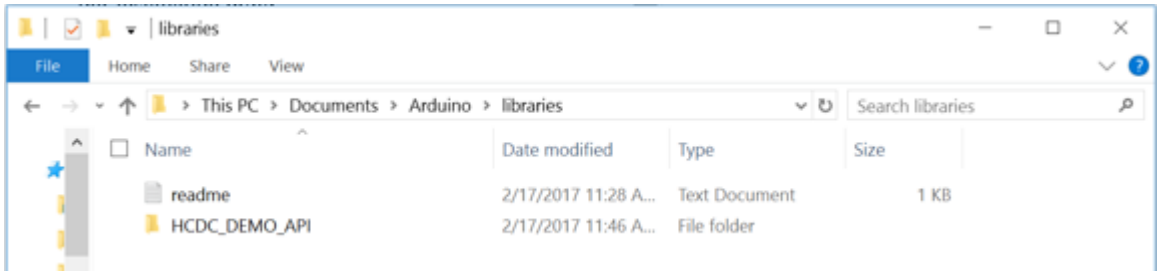
After you have successfully installed Arduino IDE, you need to put the "HCDC_DEMO_API" folder, a customized Arduino library for the HCDCv2 computer board, inside the "Arduino libraries" folder, to let the Arduino IDE compile and download our codes onto the HCDCv2 computer board correctly. Normally, for Windows users, the Arduino library is found by default at C:\Users**YOUR_USER_NAME**\Documents\Arduino\libraries; for Mac users, the Arduino library is located by default at …\Documents\Arduino\libraries. (The exact Arduino library location may differ depending on installation preference. It is not hard to find its location.)

After you find the Arduino libraries folder location, click the following link, and download the "HCDC_DEMO_API" folder and put it inside the Arduino's "libraries" folder on your laptop:

https://www.dropbox.com/sh/7khfuxs8f2vwwcn/AAAYX0ml6yO0YyKXEZzY6Byba/Libraries?dl=0

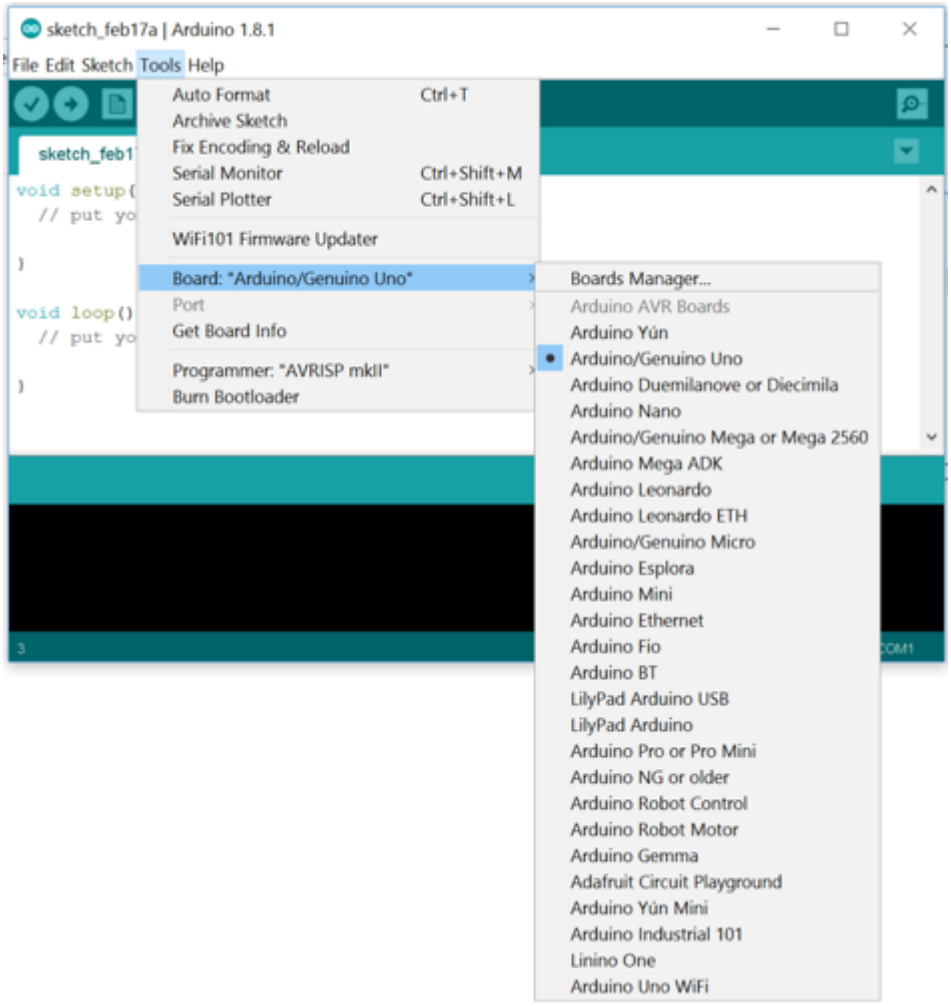
This is also the folder where you should put other libraries you use for software development. For example, the DueTimer library, which is useful for precisely triggering mixed signal conversions when using the Arduino DACs and ADCs.

The folder structure should be like the following (We use Windows as illustration; the situation for Mac OS is similar):



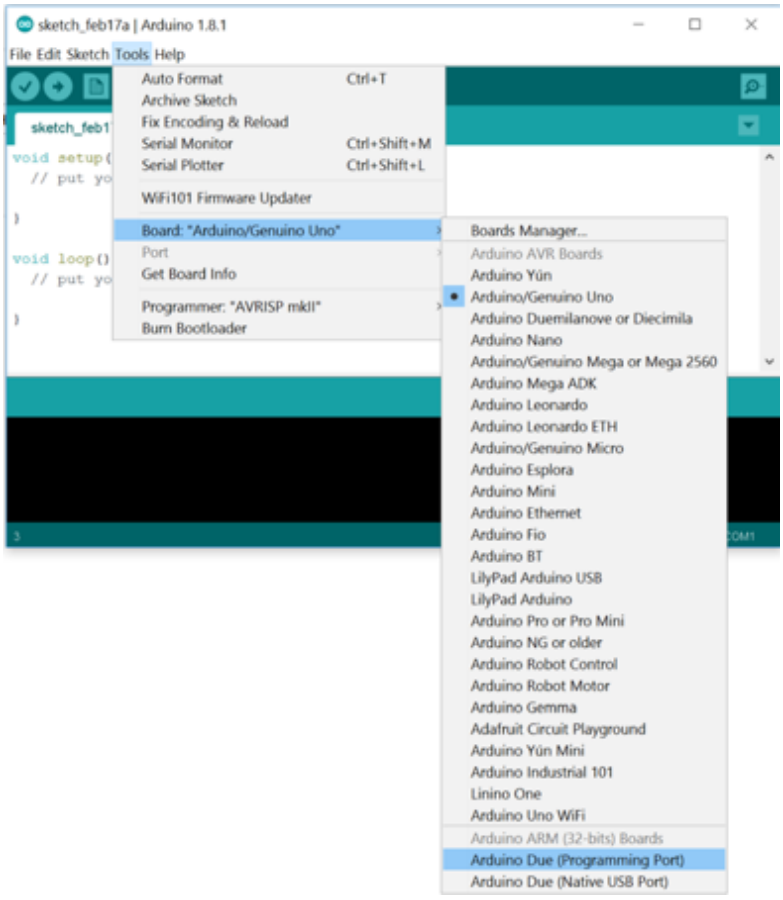
CONFIGURING THE BOARD AND PORT

Then you need to do some configuration on the Arduino software. Open the Arduino IDE software; make sure to allow internet access for it if anything pops up asking for this. Once the Arduino window opens, click the tab Tools – Board as in the following:



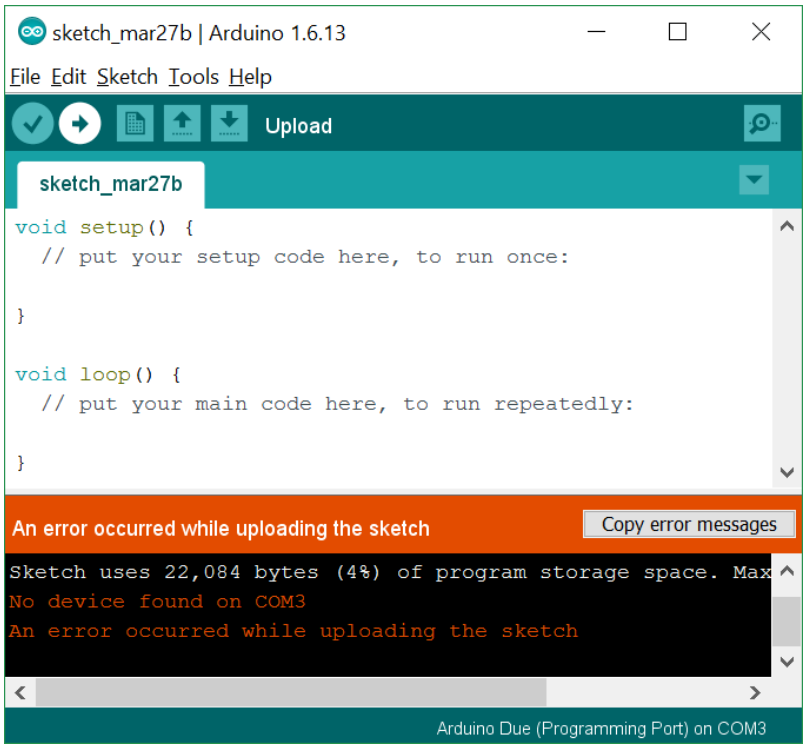
As you can see, there is no Arduino Due choice in the list. You need to install support for the Arduino Due yourself by clicking the "Board Manager" item on the top. In the popped-up window, type "DUE" in the search bar and you should see the following (if you do not see anything, make sure that nothing is blocking Arduino from accessing the internet):

Select the "Arduino SAM Boards (32-bits ARM Cortex-M3)" and click Install to start the installation. After the installation finishes, click again the tab Tools – Boards and you should see two new Arduino DUE options at the bottom of the list, as in the following:

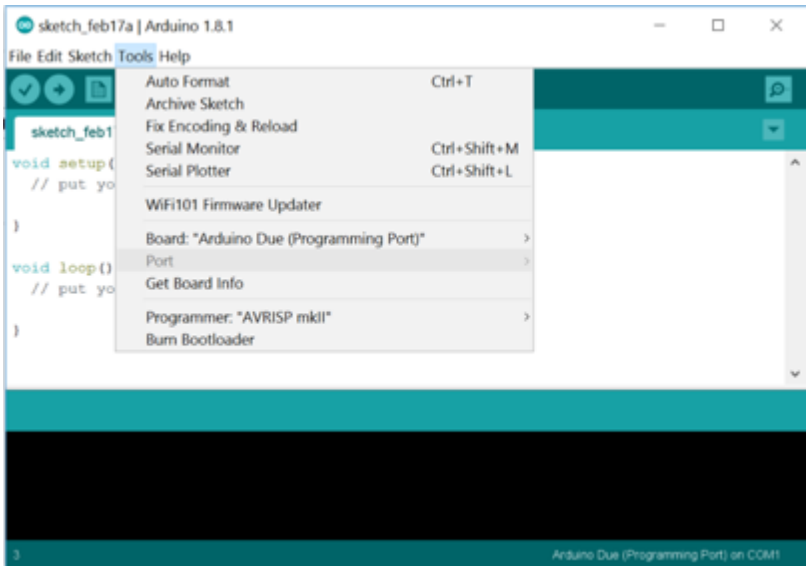


Click the “Arduino Due (**Native USB Port**)” if you are using the left port, or “Arduino Due (**Programming Port**)” if you are using the right port.

Set USB port: connect the board to one of your USB ports and click the “Upload” arrow. If no error occurs, you are ready to go and can skip this section. If you get the “No device found on COM” error as in the screenshot below,

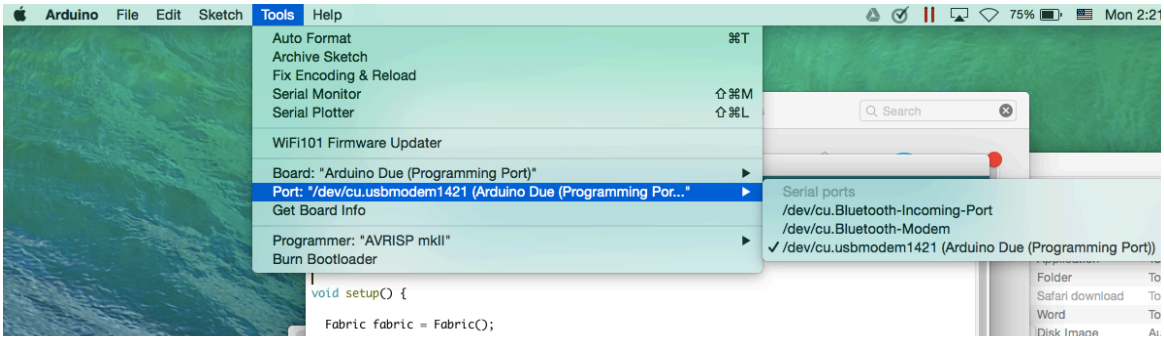


go to Tools – Port. If it is grey as shown below, connect the board to another USB port and click “Upload”. Try this with different ports until no error occurs.



If you do have options for Port, as shown below, select one, likely the one that ends with “(Arduino Due (… Port)),” and click the Upload arrow. If you are still getting the error, switch USB ports to connect the board and try different choices in

Port. Do this by trial & error until the Upload command goes through without error. Then keep this Port setting and remember the USB port to always connect to.



At this point, you can download and run the example codes given in the Appendix.

OVERVIEW OF THE HCDCV2 CHIP AND BOARD

1. **Basic math operations available:**

   Addition, Subtraction, Integration, Multiplication, Arbitrary nonlinear function generation (of one independent variable).

2. **Types of computation:**

   Continuous-time computation, current-mode.

3. **Overall accuracy for computation:**

   Around 5% errors, but it is highly problem-dependent.

4. **Nominal current range available for computation:**

   -2 μA ~ +2 μA. (-0.2 μA ~ +0.2 μA and -20 μA ~ +20 μA current ranges are also available on chip).

5. **Number system used on HCDC:**

   Analog: Bipolar, push-pull, differential analog currents, mapping to real numbers.

   Digital: 8-bit offset binary (https://en.wikipedia.org/wiki/Offset_binary), plus a trigger signal.

FUNCTIONAL UNITS FOR COMPUTATION ON THE HCDCV2 CHIP

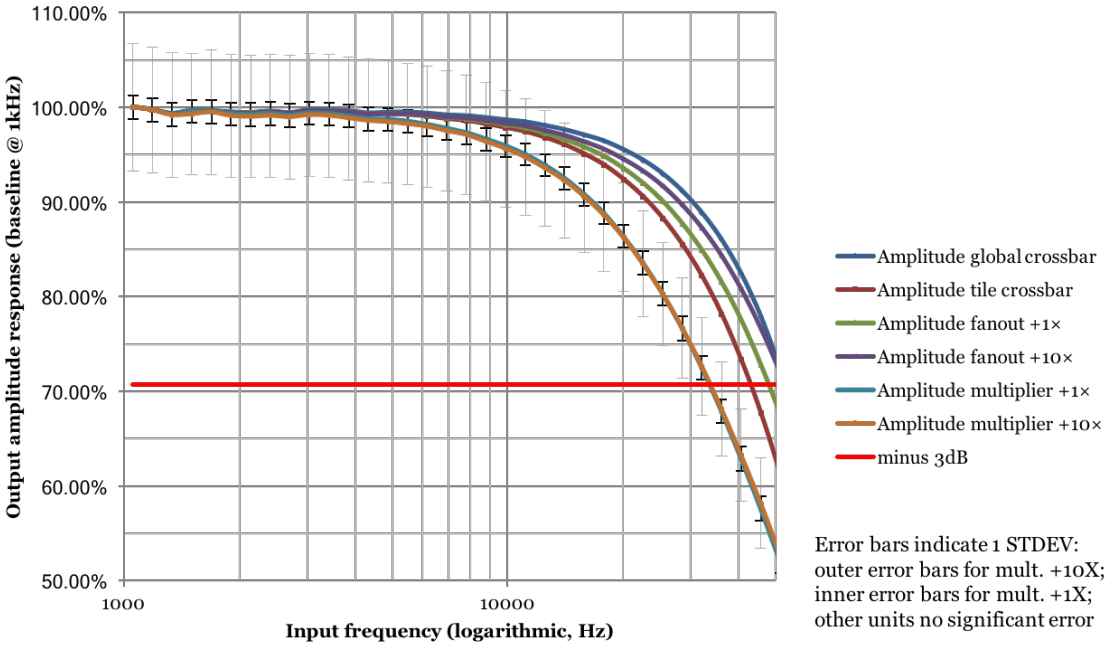
Name	Function
Fanout	Take one input current signal and generate three identical current signals
Multiplier / VGA	In multiplier mode, the module serves as a four-quadrant analog multiplier, which takes two current signals as input; in VGA mode, the module serves as a variable-gain amplifier, which takes one current signal as input
Integrator	Do integration of input current signal with respect to time
ADC	Convert an input current signal into an 8-bit digital code
DAC	Convert an 8-bit digital code into an output current signal
SRAM	Store 256 (address 0 - 255) 8-bit digital codes for table look-up purposes

Ideal input and output relations of each type of computing block:

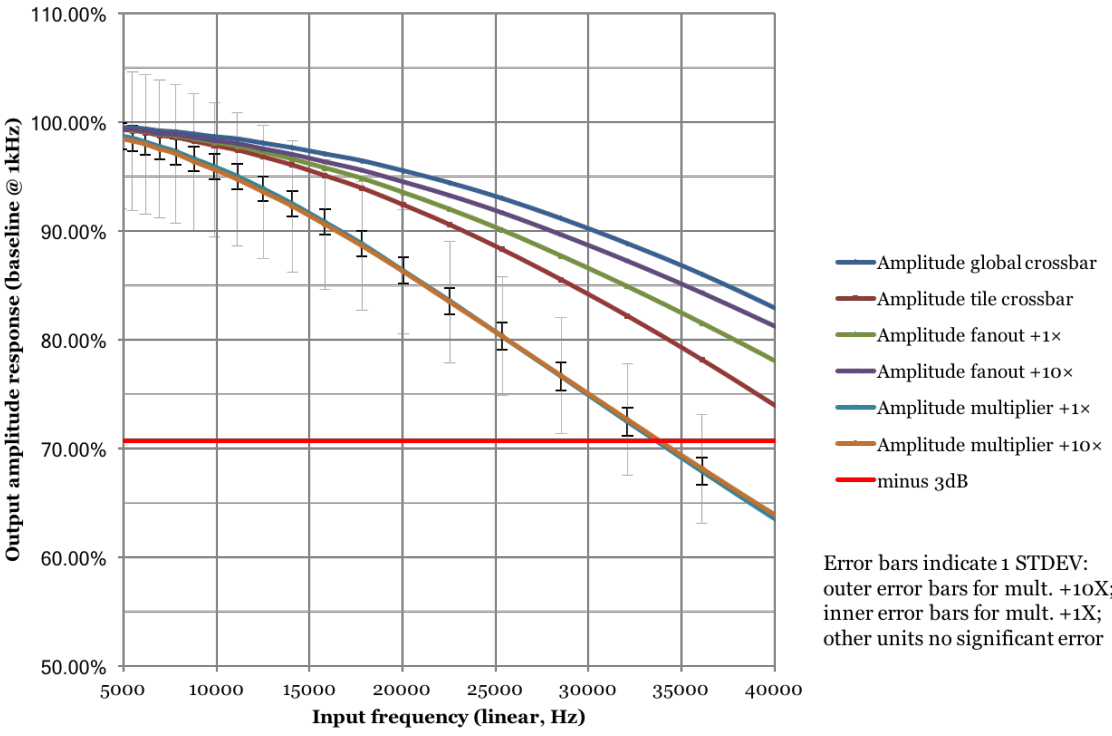
1. **Fanout:** $I_{out0}(t) = I_{out1}(t) = I_{out2}(t) = I_{in}(t)$
2. **Multiplier:** $I_{out}(t) = \frac{I_{in0}(t) * I_{in1}(t)}{I_{full\_scale}}$, where $I_{full\_scale} = 2\ \mu A$ for the current version of the computing board. Special care needs to be taken when we use the multipliers for computing. Please follow the examples to get a better understanding.
3. **VGA (Multiplier in VGA mode):** $I_{out}(t) = I_{in0}(t) * coeff$, where $coeff$ is a digitally-controlled coefficient (gain). Digital code of 0 ~ 256 is mapped to coefficient of -1 ~ +1(but remember that 256 is not available for an 8-bit digital system).
   Digital code 0 -> $coeff = -1$
   Digital code 128 -> $coeff = 0$
   Digital code 255 -> $coeff = +0.992$
4. **Integrator:** $I_{out}(t) = \omega_c * \int_0^t I_{in}(t)dt + I_{IC}$, where I_{IC} is the initial condition of the integrator, set by an 8-bit digital word. ω_c is the unity gain frequency, where $\omega_c = 2\pi \times 20\ kHz = 126k\ rad/s$. The term ω_c allows the analog computer to have a time scaling feature (usually speeding up) for solving ordinary differential equations. Three ω_c values are available on the board, namely $\omega_c = 12.6k, 126k\ and\ 1260k\ rad/s$. We recommend you start with using $\omega_c = 126k\ rad/s$.
5. **Analog, arbitrary, nonlinear function generator:** $I_{out}(t) = f(I_{in}(t))$, where the function $f()$ is implemented as a table lookup, with 256 data points of granularity. The function $f()$ is stored in the SRAM; the ADC converts the analog input signal into an 8-bit digital word, which is used as the address to read out the content stored in SRAM.

ANALOG CHARACTERIZATION FOR SELECT FUNCTIONAL UNITS

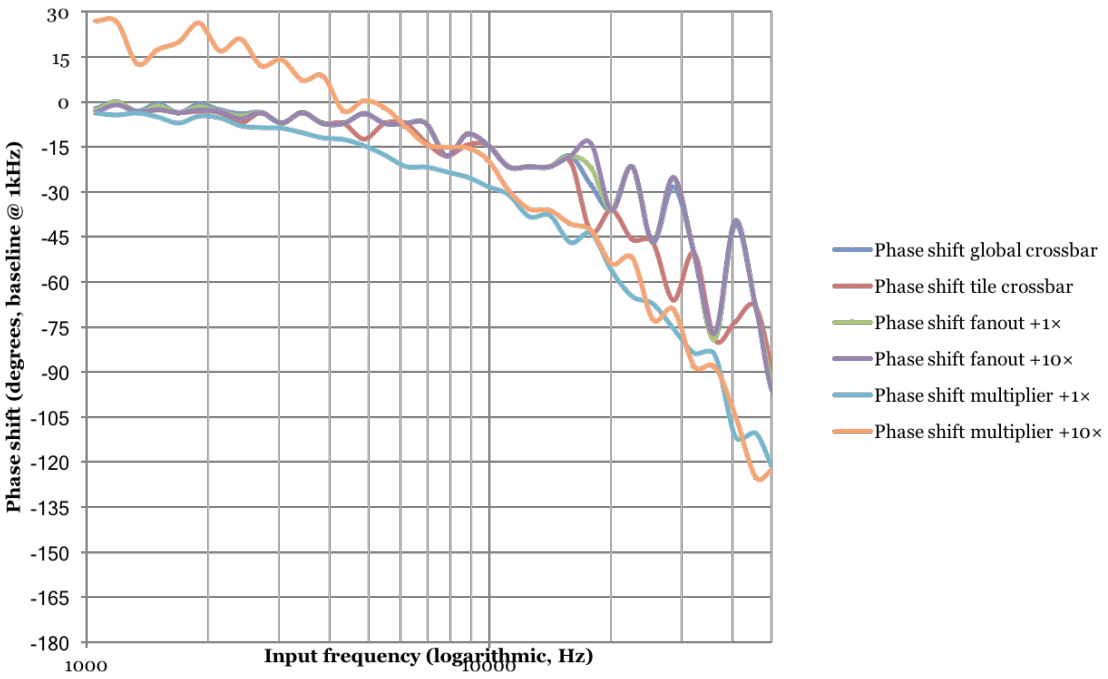
The analog characterization of the output amplitude and phase shift vs. input frequency for a few analog blocks follows:



Error bars indicate 1 STDEV:
outer error bars for mult. +10X;
inner error bars for mult. +1X;
other units no significant error



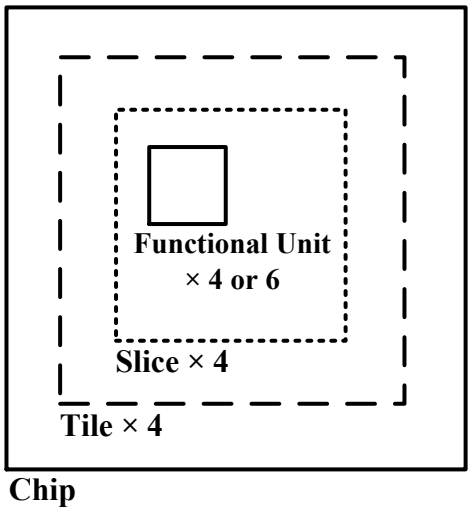
Error bars indicate 1 STDEV:
outer error bars for mult. +10X;
inner error bars for mult. +1X;
other units no significant error


30
15
0
-15
-30
-45
-60
-75
-90
-105
-120
-135
-150
-165
-180
Phase shift (degrees, baseline @ 1kHz)
1000
Input frequency (logarithmic, Hz)
10000
Phase shift global crossbar
Phase shift tile crossbar
Phase shift fanout +1×
Phase shift fanout +10×
Phase shift multiplier +1×
Phase shift multiplier +10×

ORGANIZATION OF THE HCDCV2 CHIP

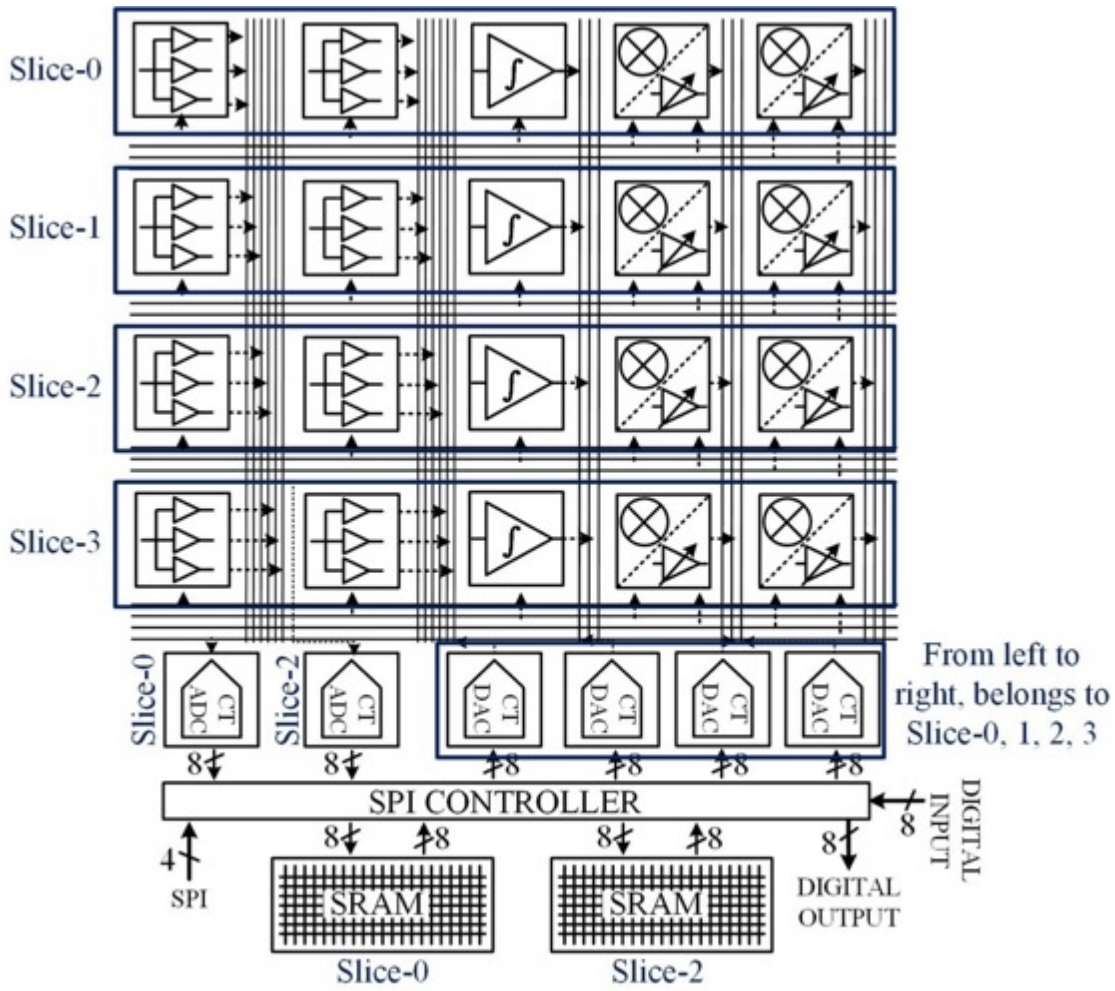
We built several virtual hierarchies on the HCDCv2 computer and adopted the object-oriented programming approach for programming the HCDCv2 chip; this approach is also consistent with Arduino IDE and its firmware library, which is written in C/C++.

As shown in the following figure, there are four hierarchies:



1. **"Chip"** is the top-level of our HCDCv2 chip.
2. **"Tile"** is the macro/core inside our HCDCv2 chip. There is a total of four macros, connected with each other through the global interconnection network (global crossbars). Each tile contains 28 functional units.
3. **"Functional Units"**: 8 fanouts, 4 integrators, 8 multipliers/VGAs, 2 ADCs, 4 DACs and 2 SRAMs. Between tile and functional units, there is another virtual hierarchy called "slice."
4. **"Slice"**: the purpose of slices is to divide all functional units into equal pieces. Since we only have 2 ADCs and 2 SRAMs, only two slices have ADC and SRAM block, and the other two do not. We use the following rules for the Slice:
   - Slice[0] and Slice[2] are called **full Slice**, which contains 2 fanouts, 1 integrator, 2 multipliers/VGAs, 1 ADC, 1 SRAM and 1 DAC.
   - Slice[1] and Slice[3] are called **half Slice**, which contains 2 fanouts, 1 integrator, 2 multipliers/VGAs and 1 DAC.

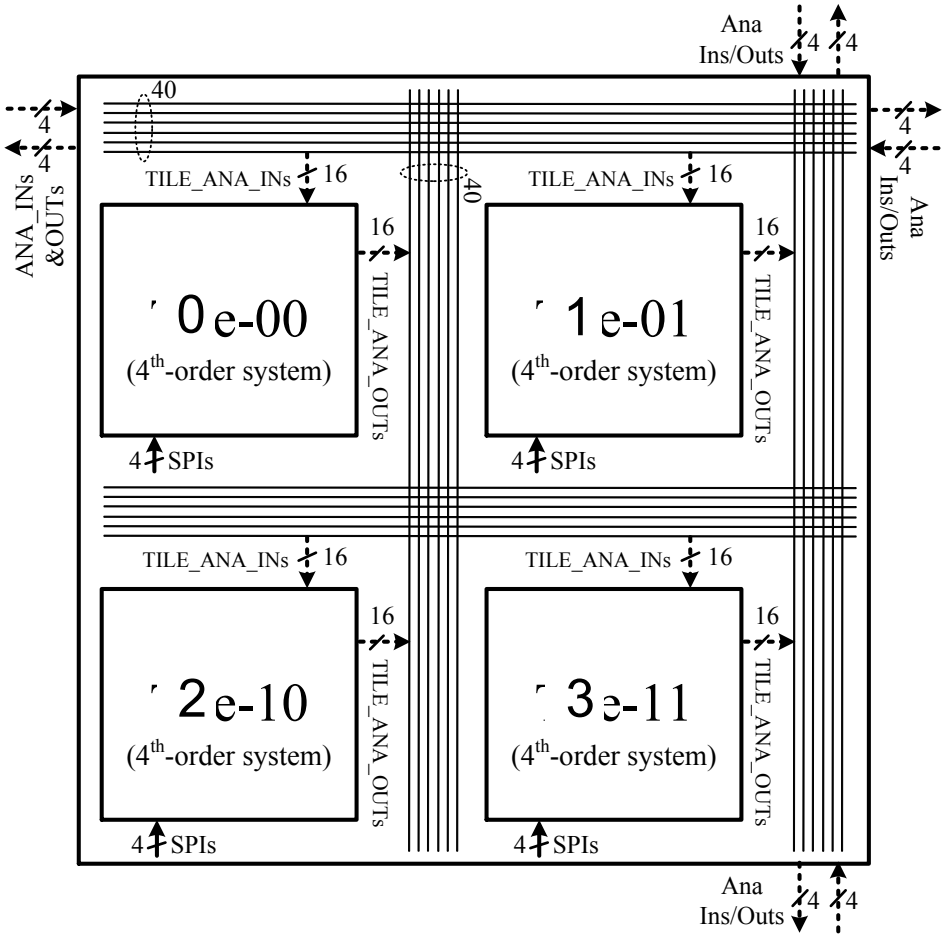
The following figure shows the orientation of Functional Units inside one Tile and how the Functional Units are divided into Slices.



NOTE: Our block naming convention is from 0, not 1. For example, the first fanout block inside Slice is named as fanout[0].

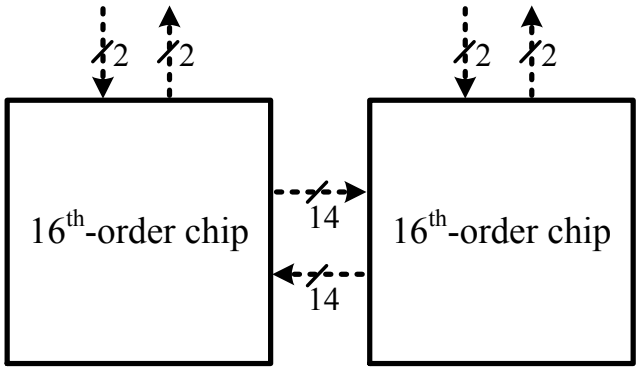
As we have seen, Function Units inside one Tile have full connectivity, i.e. any Functional Unit can connect to any other Function Unit through local crossbars.

The connectivity between four Tiles on the HCDCv2 chip is limited, as it is too expensive in area if we offer full connectivity between Functional units inside different Tiles. The following figure shows how Tiles are oriented and connected inside the chip:



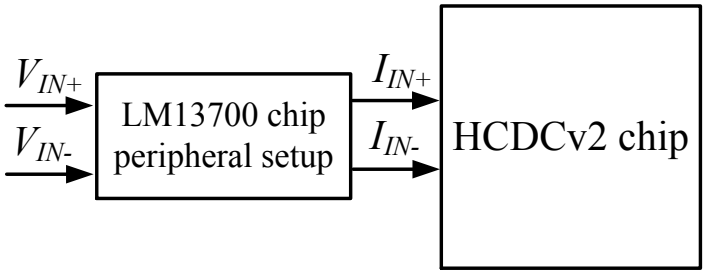
As shown, there are 40 global horizontal wires (on each row) and 40 global vertical wires (on each column). Each Tile has 16 analog inputs and 16 analog outputs. Thus, 32 of the 40 global horizontal and vertical wires are connected to the Tiles. The wires not connected to tiles contribute to the chip's 16 analog inputs and 16 analog outputs. At each intersection sit global crossbars that control the signal flows between tiles. Each Tile handles the programming of the adjacent global crossbar, i.e. the one to the top-right corner of each Tile.

ORGANIZATION OF THE HCDCV2 BOARD & CHIP INTERCONNECTIONS



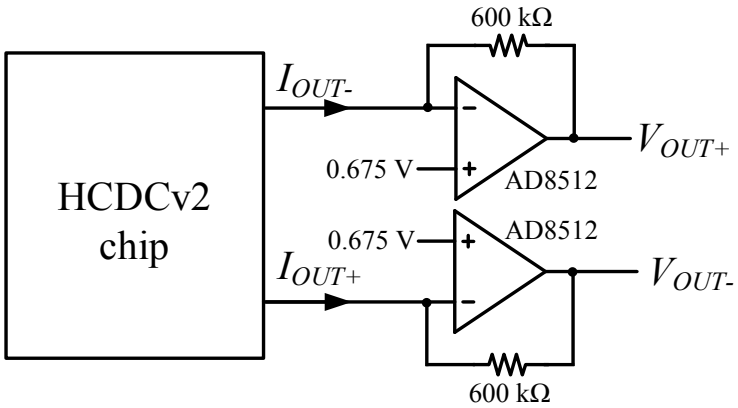
Each chip has 14 analog outputs connected to the 14 analog inputs of the other chip, and has 14 analog inputs connected to the 14 analog outputs of the other chip. Each chip has 2 analog outputs used as the HCDCv2 computer board's outputs and 2 analog inputs as the HCDCv2 computer board's inputs. Thus, the HCDCv2 computer board has 4 analog inputs and 4 analog outputs. The Appendix lists the analog interconnections between two chips.

VOLTAGE-MODE ANALOG INPUTS OF HCDCV2 BOARD



V-I conversion ratio: -55 mV ~ + 55 mV differential voltage is converted into -2 µA ~ +2 µA differential current.

VOLTAGE-MODE ANALOG OUTPUTS OF HCDCV2 BOARD



I-V conversion ratio: -2 µA ~ +2 µA differential current is converted into -1.2 V ~ + 1.2 V differential voltage.

PREPARING BLOCK DIAGRAMS TO SOLVE PROBLEMS

FROM EQUATION TO GENERAL BLOCK DIAGRAM

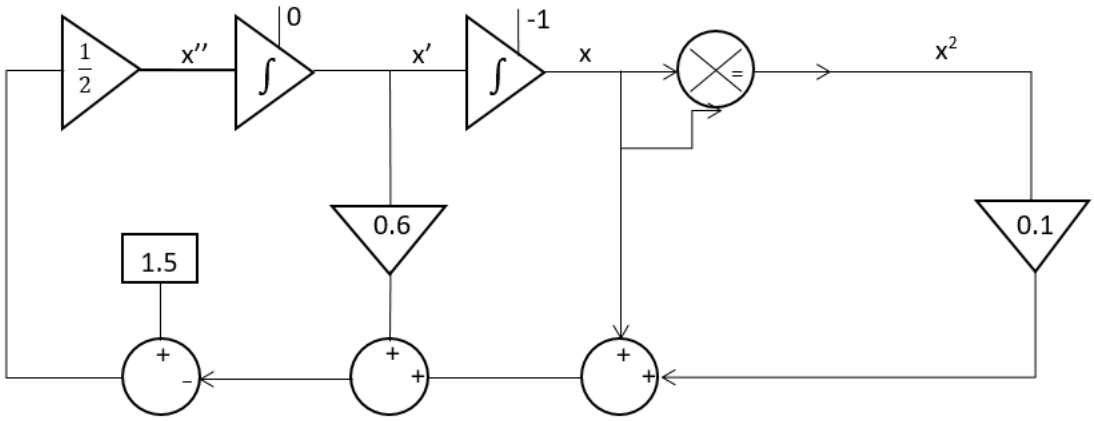
To setup the board for solving a specific differential equation, you first need a block diagram representation of the equation. If you do not know how to represent an equation with a block diagram, you can learn the systematic way of doing so here http://techteach.no/fag/tmpp250/v06/modeling/dynsys_models.pdf; a video tutorial can be found here https://youtu.be/_HHUmK0xjd0. Note that the elementary operations available on this computing board are limited to sum, constant gain, multiplication, and integration.

Then, make a few simple modifications to convert the regular (voltage-mode) block diagram to the one (current-mode) that that our board uses. Instead of using an adder, simply connect all signals (currents) to be summed to the input that takes the sum. Whenever there is a branching of a signal, use a Fanout. Below is an example of the conversion starting from the equation

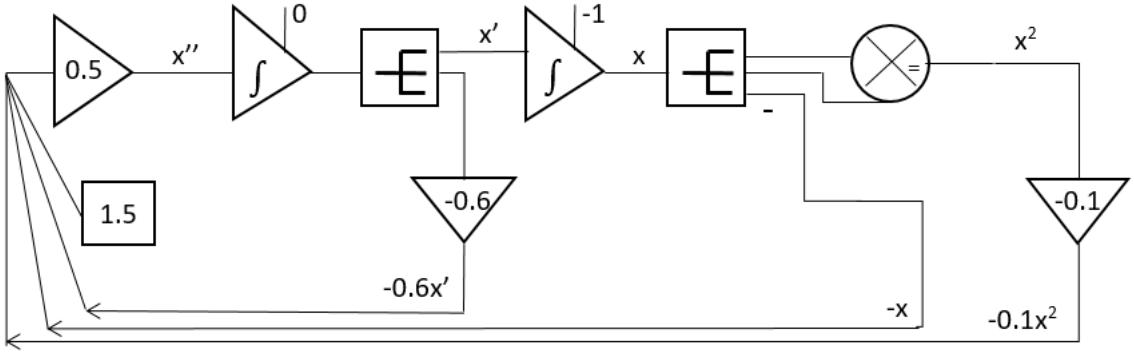
$$2x'' + 0.6x' + x + 0.1x^2 = 1.5, \qquad x'(0) = 0, x(0) = -1$$

to a current-mode diagram which will be further converted in the next section to a diagram for our chip.

Voltage-Mode Diagram



Current-Mode Diagram



FROM GENERAL TO CUSTOMIZED BLOCK DIAGRAM

With the diagram ready, decide on a scaling factor, and convert the initial condition of the integrators and the values of functions independent of the state variables (e.g. forcing function) into a 0-255 scale. For example, in the equation given in the last section, the signals are bounded by -2.5 (from -x) and 2.5 (from x). So, in our example we choose a full scale of -3.5 ~ 3.5 (M=3.5).

Convert an ordinary current-mode diagram to a more specialized diagram that you can directly translate to the code for our board:

- Pick a symmetric full-scale -M~M (i.e. -10 ~ 10, -500 ~ 500) that bounds all state variables in the system. Then convert the numerical values, i, of the initial conditions or the forcing function into an integer, σ, between 0 and 255. $\sigma = \frac{i+M}{2M} \times 256$, round to an integer. Write down the scaled initial conditions, σ, for the integrators in the customized diagram.

- It is hard to pick a perfect full scale without knowing the value or the bound of all signals in the system, so you need to find it by trial and error. Start with a relatively large guess for M; then observe the signals after you have the code ready and run the board. Unless you are using VGAs with a gain greater then unity, the signals you need to pay attention to are outputs of integrators. Probe the outputs (see section "Send signals concerned to board analog outputs") of integrators to see if they ever reach 1.2V. If any of them does, increase M. If the integrator outputs are all far away from 1.2V, then decrease M. You can also check if there is saturation at the output of integrators using the option introduced in section "Check for saturation". If you are using VGAs with a gain greater than one, then either probe their output or do a simple calculation by hand when you know the maximum value of their inputs already.

- NOTE when using multipliers (not in VGA mode): in most of the cases, both input signals you feed to the multiplier are scaled to the full scale M, and then there is a virtual gain of 1/M from the multiplier (because the output of the multiplier is $I_{out}(t) = \frac{I_{in0} * I_{in1}}{I_{full\_scale}}$). You need to compensate back a gain of M for the multiplier output. The example diagram below uses myMul to generate x^2; as both of its inputs, x's, are scaled to M, its output is $\frac{x^2}{M}$. So, the following VGA takes a gain of -0.1*M=-0.1*3.5=0.35 to produce $-0.1x^2$ at the output of myVga2. In cases where one input (e.g. x(t)) is scaled to M and the other (e.g. sint(t)) scaled to 1, there is no virtual gain and the output of the multiplier will just by $In_1 * In_2$ (this also follows the multiplier output function). The "Feed analog signals into the chip" section discusses an example for this case later.

- Do the scaling for VGA gains. For the numerical gain $g \in [-1, 1]$ in your Current-Mode Diagram, use the formula Coeff = $\frac{g+1}{2} \times 256$ to get a rounded integer, Coeff, between 0 and 255, and put it in the customized diagram for the VGA blocks. For this example, in the diagram below, 83 in myVga2 comes from $\frac{-0.35+1}{2} \times 256$ and corresponds back to $g = -0.3516$. For a gain such that $|g| \in (1, 100]$, leave its numerical value in the customized diagram.

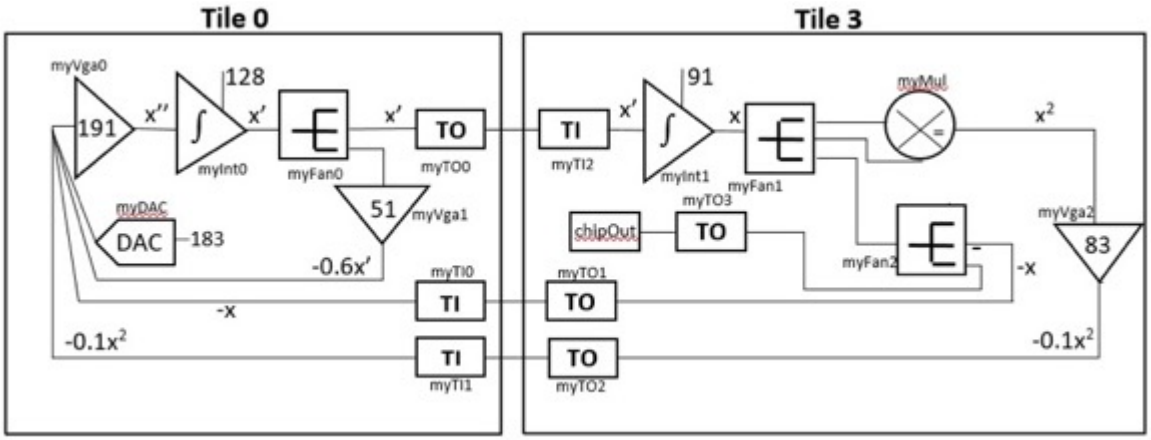
- Convert values of independent functions also to integers between 0 and 255 using the same formula for ICs, that is, scale them to M. In the example, the forcing function is a constant 1.5, thus we get $\frac{1.5+3.5}{2*3.5} \times 256 \approx 183$, where M=3.5 is the full scale chosen.

- Write down a name for each functional unit, which you plan to use in the code.

- Finally, do this only if you are using units from more than one tile: group units by the tiles they are in, and include Tile Inputs (TI) and Tile Outputs (TO) blocks.

The diagram below is the customized diagram for the example discussed before.

Customized Diagram for HCDC



WRITING CODE FOR BLOCK DIAGRAMS TO SOLVE PROBLEMS

After you have the Arduino software opened, click “File”, then “New” in Arduino to create a new .ino file where you write C++ code to configure the board. Your .ino file should begin with the following lines of code, then codes in following sections. Replace the BLUE parts in the code with numbers or Booleans following instructions in the paragraph above them. You should leave the rest of the black code as they are.

```
#define _DUE
char HCDC_DEMO_BOARD = boardIndx;
#include <HCDC_DEMO_API.h>
void setup() {
      Fabric fab = Fabric ();
      fab.calibrate();
      …
```

You can find the **boardIndx** on top of the board. Note that the bottom line of the above code is for calibrating the chip. Calibration takes a few minutes every time this code is uploaded. Open Tools -- Serial Monitor to see the status during the calibration. The outputs of the board can be problematic if you do not include this line of code, so it is recommended to include it most of the time, although leaving it out will save you some time if you upload the code many times (in the testing stage) and do not need an accurate output. If the scrolling serial monitor ends up showing “Exiting with status 1.”, the calibration fails, and you need to re-upload the code so that the board tries to calibrate again.

If calibration is taking too much time, and you are already experienced with the chip and library code, you may go into the library source code, and adjust the files fabric.cpp, chip.cpp, tile.cpp. The lines such as **`chips[1].calibrate(); tiles[1].calibrate(); slices[1].calibrate();`** enable and disable calibration for parts of the chip you don't need.

INITIALIZING THE FUNCTIONAL UNITS AND TILE INPUTS AND OUTPUTS

Pick which functional units – in which chip, tile, and slice – you want, and initialize an object for each of them using the code listed below. Using units from as few tiles as possible will save you some inter-tile connections. Include the lines of code below for initializing Fanout, Multiplier, VGA, Integrator, ADC, DAC, SRAM(LUT), Tile Input Channel, and Tile Output Channel respectively.

```
Fabric::Chip::Tile::Slice::Fanout * myFan =
&fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].fans[unitIndx];

Fabric::Chip::Tile::Slice::Multiplier * myMul =
&fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].muls[unitIndx];

Fabric::Chip::Tile::Slice::Multiplier * myVga =
&fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].muls[unitIndx];

Fabric::Chip::Tile::Slice::Integrator * myInt =
fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].integrator;

Fabric::Chip::Tile::Slice::Adc * myADC =
fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].adc;

Fabric::Chip::Tile::Slice::Dac * myDAC =
fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].dac;

Fabric::Chip::Tile::Slice::LookupTable * myLut =
fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].lut;

Fabric::Chip::Tile::Slice::TileInOut * myTI =
&fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].tileInps[ioIndx];

Fabric::Chip::Tile::Slice::TileInOut * myTO =
&fab.chips[chipIndx].tiles[tileIndx].slices[sliceIndx].tileOuts[ioIndx];
```

You can put whatever names (in RED) for each unit, as long as they are distinct and used consistently in the following parts. Fill in the corresponding index to set which exact unit to use. For '`chipIndx`' and '`unitIndx`', pick from 0 and 1; for '`tileIndx`', '`sliceIndx`' and '`ioIndx`' pick from 0 to 3; note that '`sliceIndx`' for myADC and myLUT can only be 0 or 2.

CONFIGURING THE FUNCTIONAL UNITS

First, enable every functional unit except for LUT, ADC and DAC (TI and TO are not functional units) by writing one line of the code below for each unit initialized above. For example,

```
myInt -> setEnable( true );
```

1. **Multiplier/Vga:** if you use this unit as a multiplier, no configuration is needed; just make sure when making connections later, that you are connecting a signal to each of its input ports: in0 and in1. If you want to use it as a VGA, include the two lines below, where `coeff` is what you obtained in the previous section, and put in your last diagram for all gains less than 1. Note a VGA has one input, in0, while a multiplier has two.

   ```
   myVga -> setVga( true );
   myVga -> setGainCode( coeff );
   ```

   For gains greater than unity, use the code below, where `g` is just the numerical value of the gain and can be -10~ -100, 10~100.

   ```
   myVga -> setGain( g );
   ```

   The **`setGain()`** function actually recalibrates the analog unit and makes sure the gain matches the intended value; as such it has higher effective precision higher 1/256.

2. **Integrator:** set the initial condition between -1 and 1 by the code below, where the '`σ`' is obtained in the previous section and noted in the customized diagram.

   ```
   myInt -> setInitial( σ );
   ```

   The **`setInitial()`** function actually recalibrates the analog unit and makes sure the initial condition matches the intended value; as such it has higher effective precision higher 1/256. In general, the integrator input offset calibration has the biggest impact on accuracy. So, we should set and calibrate the integrators last. To do this, just make sure **`setInitial()`** is called after everything else (parameters, connections) is set up.

3. **SRAM (LUT):** The LUT takes a digital input address and outputs a digital value corresponding one-to-one to that address. What the LUT does is simple; you can come up with usages specific to your problem. One example would be implementing the function f(x)=sin(x(t)). Since the LUT has limited memory, you need to know ahead of time the possible values the independent variable can take to fit them into the 256 addresses. Refer to EXAMPLE_4 in the Appendix for how to configure the LUT for a sine function.

   Write code to assign 256 values (integers from 0 to 255) to the 256 addresses (integers from 0 to 255). The output of the LUT changes whenever the input changes. Assign a `value` to each of the 256 `addresses` using the code below. Determine the 0~255 scaled value of '`value`' using the same convention as that used for figuring out the values for the initial conditions. Refer to EXAMPLE_4 in the Appendix for assigning LUT values using a loop.

   ```
   myLut -> setLut ( address, value );
   ```

   Then choose which source you are using as the input of the LUT. Include the code below where exactly one of the three Boolean arguments is 'true'. When '`boo0`' is 'true', LUT takes input from parallel input; when '`boo1`' is 'true', it takes input from output of ADC in Slice[0]; when '`boo2`' is 'true', it takes input from ADC in Slice[2]. So you do not need to draw connections between ADC and LUT using the **`setConn()`** function illustrated in the next section. Parallel input and output are the digital I/O of the board and will be explained later.

```
myLut -> setSource ( boo0, boo1, boo2 );
```

Include the code below if you want to send the digital output of the LUT to the board digital output.

```
myLut -> setParallelOut ( true );
```

4. **DAC and ADC:** Similar to LUT, you set the connection to DAC using its **`setSource()`** function shown below instead of **`setConn()`** function. When '**`boo3`**' is 'true', the DAC takes input from registers (not often used); when '**`boo4`**' is 'true', it takes input from chip parallel input; when '**`boo5`**' is 'true', it takes input from SRAM in Slice[0]; when '**`boo6`**' is 'true', it takes input from SRAM in Slice[2]. Put 'false' for all other three Boolean arguments.

   ```
   myDAC->setSource ( boo3, boo4, boo5, boo6 );
   ```

   The **`setConstantCode()`** function makes the DAC to output a constant voltage corresponding to a digital 'value' from 0~255. This function is useful when you have a constant forcing function in your equation. Use the same formula for the initial condition to determine the '**`value`**' per the numerical value of the constant in your mathematical equation.

   ```
   myDAC->setConstantCode ( value );
   ```

   Use the **`setConn()`** function to connect a signal to the input of the ADC. The output of each ADC has connections to both SRAM in the chip; which one is actually used is determined by the setSource function of the LUT. Include the code below if you want to send the digital output of the ADC to the board digital output.

   ```
   myADC->setParallelOut( true );
   ```

5. **Invert output:** there is no subtraction unit but you can negate a signal at a unit's output by the code below. The units whose outputs can be inverted are Fanout, Integrator, and DAC. Use it for every '-' sign in your current-mode diagram.

   ```
   myFan -> out0 -> setInv ( true );
   myInt -> out0 -> setInv ( true );
   myDAC -> out0 -> setInv ( true );
   ```

   You can also invert an output using a VGA with gain -1, that is **`setGainCode(0)`**.

MAKING CONNECTIONS BETWEEN FUNCTIONAL UNITS

Now that you have everything set up according to the last diagram for your specific equation, it is time to draw all the links. Note the numbering of the I/O ports of all units has the same format: in# and out#, starting from 0. For example, a Fanout has one input, called **in0**, and three outputs, **out0**, **out1**, **out2**; a Multiplier has two inputs, **in0**, **in1**, and one output called **out0**.

Connections inside a tile. There is no limitation on the connectivity within a tile; you can make as many connections between any two functional units as you need. Below is an example for connecting the second output of a Fanout called myFan to the first input of a Multiplier called myMul.

```
Fabric::Chip::Connection ( myFan->out1, myMul->in0 ).setConn();
```

Connections across tiles. When you need to connect an output of a unit, say myFan in Tile 0, to an input of another unit, say myInt in Tile 2, put the output signal through a tile output channel (myTO) in Tile 0 and a tile input channel (myTI) in Tile 2. The three lines of code below is an example of connecting the third output of myFan to myInt.

```
Fabric::Chip::Connection ( myFan->out2, myTO->in0 ).setConn();
Fabric::Chip::Connection ( myTO->out0, myTI->in0 ).setConn();
Fabric::Chip::Connection ( myTI->out0, myInt->in0 ).setConn();
```

Connections across chips. To send a signal across chips, first connect the functional unit in Chip 0 to one of the 16 tileOuts of its tile. Then from the tileOuts to one of the 14 chipOutput in Chip 0. Within a chip, any tile I/O can be connected to any chip I/O.

Then, connect the chipOutput to one of the 14 chipInput of Chip 1, then to a tileInps in the tile of that unit in Chip 1, and finally to that unit. There is one chipOutput and one chipInput in every slice.

But the chip I/O in Tile 3 Slice 2 and Slice 3 are connected to the board analog I/O, not to the other chip. See the Appendix for the mapping of chipOutputs to chipInputs.

ANALOG OUTPUT: ROUTING ANALOG SIGNALS TO CHIP OUTPUTS

Each chip has two output channels connected to the board's analog output; they are in tiles[3].slices[2] and tiles[3].slices[3]. To observe a signal from a unit using an oscilloscope or sample it using an Arduino ADC, you need to send it to one of the four board analog outputs.

First, connect the signal to one of its tile's 16 output channels, say myTO, then connect the output of myTO to `chipOutput->in0` in Tile 3 Slice 2 or 3, with the code below

```
Fabric::Chip::Connection ( myTO->out0, fab.chips[0].tiles[3].slices[2].chipOutput-
>in0 ).setConn();
```

The chipOutput in Chip 0 Tile 3 Slice 2 goes to the first analog output pairs on the board (printed as anaOut0+ and anaOut0-),
that in Chip 0 Tile 3 Slice 3 goes to anaOut1+/-,
that in Chip 1 Tile 3 Slice 2 goes to anaOut2+/-,
that in Chip 1 Tile 3 Slice 3 goes to anaOut3+/-.

ANALOG INPUT: FEEDING ANALOG SIGNALS TO CHIP INPUTS

The common use of these input channels is for non-constant forcing functions in your equation. Generate the forcing function you need using a function generator, which gives waveforms like sine, square, triangle, ramp etc. Like using the board analog outputs, connect **`chipInput->out0`** to one of the tileInps (illustrated by the code below), then output of the tileInps to a functional unit in that tile:

```
Fabric::Chip::Connection ( fab.chips[0].tiles[3].slices[2].chipInput-> out0,
fab.chips[0].tiles[3].slices[1].tileInps[1]-> in0 );
```

The first analog input pairs on the board (printed as anaIn0+ and anaIn0-) is in Chip 0 Tile 3 Slice 2,
anaIn1 is in Chip 0 Tile 3 Slice 3,
anaIn2 is in Chip 1 Tile 3 Slice 2,
anaIn3 is in Chip 1 Tile 3 Slice 3.

You only need to supply a single-ended signal to the board (i.e. the group from the function generator goes to the negative pin of the board analog input pair). Example 3 and 7 in the appendix use the analog input.

Here is how you find the amplitude, frequency and offset of the analog signal you want from the function generator. Say a function f(t) in your mathematical equation is a sine wave with amplitude A and frequency f. Then feed a sine wave with amplitude $\frac{A}{M}$ ×55mV, and frequency f *126000 for f(t). If your function generator cannot generate a signal small enough for a small A, connect the input signal through a VGA inside the chip. Ideally, the signal you supply does not need an offset other than what is explicitly required by your equation. But you do need to add an offset on the ideal case offset to compensate for the bias of the V-I conversion components. Refer to the chart here for the extra offset you need for the specific analog input pair you use:

https://docs.google.com/spreadsheets/d/1gzetbeWd78UoS3lmaWNEo0qy56nXgcrBJnk4U9y-SvI/edit?usp=sharing

Here is an example of using external analog signals specifically with multipliers. Since 55mV (quite low amplitude for function generators) of an external signal correspond to the full scale M and you usually don't need that large a signal, you may often need a VGA to decrease the input so it fits in the full scale. But recall that a multiplier comes with a gain of 1/M if both of its inputs are in the full scale. So one trick here is, if you want x(t)*0.9sin(t) for example, connect the M-scaled x(t) to one of the multiplier input as usual, but connect the external signal, 0.9sin(t), scaled to -1~1 to the other input. This way, you feed in a sine wave of amplitude 55*0.9=49.5mV and connect it directly to the multiplier; you do not need the extra VGA to bring it down to $\frac{49.5}{M}$ mV, and you do not need to compensate back a gain of M after the multiplier. The multiplier in this case gives you x(t)*0.9sin(t) right at its output. However, this only works if you are multiplying the analog input with a signal already in full scale. Otherwise, you still need to follow the standard scaling procedure stated in the paragraph above to fit it into the full scale before being added or integrated with other signals in full scale.

DIGITAL PARALLEL OUTPUT / INPUT

There is one digital parallel input and one digital parallel output for the board. Recall from section "Configuring functional units" that through the **`setSource()`** and **`setParallelOut()`** functions of the LUT, DAC and ADC, you can make use of the parallel I/O channels. Unlike board analog I/Os, no **`setConn()`** needs to be called here.

You can feed in an external 8-bit digital input+1 trigger/clock signal into the board as the parallel input through the 9 digital input pins shown in Figure 1. You can send the parallel output signal to the board output and connect the 9 pins to an external device to observe them.

To feed a digital signal (that is already at the board digital input) into a DAC or a LUT, first turn on the tile digital input of the tile where your DAC and LUT are, then use the **`setSource()`** function to set the parallel input as the source, using the code below.

```
fab.chips[chipIndx].tiles[tileIndx].setParallelIn(true);
myDAC->setSource ( false, true, false, false );
```

To get the digital output from a ADC or a LUT (to the board digital output to be observed by external devices), simply include one function call, but remember you cannot do this for more than one unit:

```
myADC -> setParallelOut(true);
```

There is also a more convenient option of generating a parallel input and obtaining the parallel output, which doesn't require external devices. But due to the speed limitation of the Arduino board, signals obtained through this method are of low frequency so it is only suitable for measuring constant signals.

To generate a digital signal inside the board as the parallel input, use the function below (usually in a loop), where **`value`** is found the same way as for initial conditions. Each call to this function sets the parallel input to a constant value which persists until the next call to this function with a different argument. Include a **`delay(#)`** in the loop to change the value of the signal every **#** miliseconds. Use **`delayMicroseconds(#`**) for a # microsecond delay.

```
fab.chips[chipIndx].writeParallel(value);
```

To obtain values of the parallel output, use the function below (usually in a loop). Each call to this function will give you one integer (between 0 and 255) equal to the value of the parallel output the instant it is called.

```
unsigned char myPout = fab.chips[chipIndx].readParallel();
```

Then you can use myPout as a variable. You can either print it (put the code below in the same loop) on the Arduino interface when you run the code, or save a series of its values over time into an array, or use it in other function calls, etc. However, note that for the numbers you get, you do not get the exact time they appear at. So, you should only observe a constant signal using this method.

```
// print to Arduino Serial Monitor:
Serial.println(myPout); // programming port
SerialUSB.println(myPout); // native port
```

You can see the serial outputs by going to Tools – Serial Monitor, the same place where the calibration details are shown. See example 7 in the appendix for putting some of the codes in this section together.

COMMITTING A CONFIGURATION, STARTING AND STOPING SIMULATION

The transient response of a system often fades out quickly, so you need to run the simulation starting from the initial state over and over to make sure the signal starting from time=0 can be captured by the oscilloscope at any time, unless you only need a steady state solution, in which case you include only the first and third line of the code below. With the code below, the board will keep simulating your equation starting from time=0, for 3 ms. Recall that 3 ms corresponds to about $3*10^{-3}*126000=378$ units of time in your equation.

The number in **delay()** is the interval in milliseconds in real time, the time you observe on oscilloscope, that each simulation runs for. The command **fab.cfgCommit()** will refresh the chip with all the settings you've written above this line. The commands **fab.execStart()** and **fab.execStop()** start and stop the integrators.

```
fab.cfgCommit();
while (true) {
      fab.execStart();
      delay(3);
      fab.execStop();
}
```

CHECKING FOR SATURATION

Besides probing the signal with an oscilloscope, you can check whether the output of an integrator (or an ADC) is saturating through the Arduino interface by adding the code below **after** the call to **fab.execStart()**. If it prints 'true' in the serial monitor, then the integrator is saturating and you need to increase M. Otherwise it will print false.

```
bool sat = myInt -> getException();
Serial.println( sat );
```

SWITCHING AMONG MULTIPLE CONFIGURATIONS IN ONE CODE

As a more advanced case, you may want the board to run a simulation for different diagrams (systems) and automatically switch among them, instead of uploading and running one configuration in one .ino file at a time. To do this, break the connections and turn off the functional units for the first configuration; then setup the connections and units for the second configuration. In this case, you may want to declare each connection you want to break later and name them, by using the following code:

```
Fabric::Chip::Connection myConn = Fabric::Chip::Connection ( myInt->out0, myFan->in0 );
```

The two functions below break a connection and disable a unit respectively:

```
myConn.brkConn();
myInt -> setEnable(false);
```

FINISHING THE CODE

Close the **`setup()`** section with one more **`}`**.

After this, your .ino file should end with the following line (already there when the new file is generated). That is, put nothing in the loop ( ) unless you are familiar with Arduino code and know what you are doing.

```
void loop () {}
```

To save your code, click File – Save, make sure you are inside the Arduino folder, give your code a name and click Save. Then a folder with that name will be created in the Arduino folder, with your .ino file with the same name in it. Note that an .ino file should always be in a folder of the same name, i.e. you cannot run an .ino file in the directory ../Arduino where the "libraries" folder is.

A complete code from the customized diagram of the example is included as the 6th working example in the appendix.

ACQUIRING AND INTERPRETING THE OUTPUT: OSCILLOSCOPE

After the board is connected to your computer through the USB cable and you have the code and probes ready, click the check mark on the upper left corner of the Arduino interface to verify the code. Then click the right arrow next to the check mark to upload the code to the board. Wait a few seconds for the board to setup before you can see a stable output on the oscilloscope (if the equation does give a stable system). You can see the serial output (i.e. output from **`Serial.print()`**) by going to Tools – Serial Monitor.

For each one of the four chip analog output pairs, you need two channels of an oscilloscope to probe it: CH.1 probing the positive (+) pin and CH.2 probing the negative (-) pin. You can read the print on the board (e.g. 'anaout0+') to determine which pin is the +/- pin of which pair of outputs. Be careful with connecting the grounds of the probes to the correct pins on the board, which are shown in Figure 1 and can also be found by looking to the side for 'GND'. Then, using the MATH function available on oscilloscopes, take the difference of the two channels, CH.1-CH.2. Export the voltage vs. time data from the MATH channel as your raw output.

The final step to get to the numerical solution, x(t) of your equation(s) is time and amplitude scaling. Multiply the time vector by 126000. Multiply the voltage vector by $\frac{M}{1.2}$. Recall that M is from the full scale chosen at the beginning.

ACQUIRING AND INTERPRETING THE OUTPUT: ARDUINO ADCS

We can also measure the analog output of the analog chip using the ADCs of the Arduino. The Arduino ADCs offer 12-bit precision, and can steadily sample every 3 microseconds. The timing of each Arduino ADC sample is done using timer interrupts in the Arduino microcontroller. The high sampling rate results in a lot of digital data to read back from the Arduino, so you should use the native USB port to read back the data.

The following code is an example of doing so:

```
#include <DueTimer.h>

volatile unsigned short timeIndx = 0;
const unsigned short timeSize = 256; // number of samples

// a buffer to store the samples:
volatile unsigned short adcCode[2][timeSize];

// timer delay between the ADC samples:
// timer is validated to go as frequent as 3us
const float delayTimeMicroseconds = 3;

void setup() {
      SerialUSB.begin(9600);
      while (!SerialUSB);
      // use DueTimer library timer to call myHandler every 3 uSeconds
      Timer3.attachInterrupt( myHandler );
}

void loop () {
      fabric->cfgStop();
      // start timer
      Timer3.start( delayTimeMicroseconds );
      // start integration
      fabric->execStart();
      while (timeIndx != timeSize) {};
      fabric->execStop();
      // convert the ADC codes to real values
      float val = adcCodeToVal(adcCode[0][k]);
      SerialUSB.println(val,6);
}

void myHandler() {
      // get values
      adcCode[0][timeIndx] = ADC->ADC_CDR[6]; // anaOut0
      adcCode[1][timeIndx] = ADC->ADC_CDR[4]; // anaOut1
      timeIndx++; if (timeIndx == timeSize) Timer3.stop();
}

float adcCodeToVal (float adcCode) {
      return (3.267596063219 - 0.001592251629 * adcCode) / FULL_SCALE;
}
```

APPENDIX: WORKING EXAMPLES

Codes for the examples below can be found at
https://www.dropbox.com/sh/7khfuxs8f2vwwcn/AACz2tHjFnHqK3N6Xnd-rt9oa/Test%20sets?dl=0

Here is how you run the example codes:

- Download the folder for each example and put them under the ../Document/Arduino directory where your "libraries" folder is.
- Click the folder, say "test_1_tdi_lut_tdo".
- Click open the .ino file and an Arduino window will pop up automatically with the code on it.
- Click the "Verify" check at the top left corner and wait for the code to compile.
- Click the "Upload" arrow and wait the code to be uploaded to the board.
- Now the board runs simulations as directed by the code.

EXAMPLE 1: TEST_1_TDI_LUT_TDO

This test checks the functionality of all digital pins. The signal paths are connected as TDI -> SRAM -> TDO.

The SRAM block is checked one by one, from Tile to Tile, by random write and read.

EXAMPLE 2: TEST_2_EQUATION_1

This test solves a second-order linear ODE. The solution can be measured at analog output 0 on the board.

See the following for more details: https://docs.google.com/spreadsheets/d/16OUbpPYWz2AkzahYhePIV-2jrkbWBkqfBp6dv6QCqi4/edit?usp=sharing

EXAMPLE 3: TEST_3_ANAIN_TO_ANAOUT

This test checks the analog input channels. The signal path is connected as Analog IN0 -> Analog Output0.

If we apply a -55mV - + 55mV sine input signal to board input, we should observe a -1.2 V - +1.2 V analog output signal.

EXAMPLE 4: TEST_4_SINE_LOOKUP

This test checks the nonlinear function generator configured as a sine lookup. The signal path is connected as DAC0 -> ADC -> SRAM -> DAC -> board output. DAC0 generates a ramp repeated signal to lookup the sine function stored inside the SRAM. You should observe a sine wave form at board output.

EXAMPLE 5: TEST_5_CHIP_INTERCONNECTIONS

This test checks the interconnections between two chips. The signal path is connected in a snake-like path. One end is connected to the DAC's output and the other end is connected to the board output. If the interconnections are perfect, we should measure an expected voltage, calculated based on DAC's output current.

EXAMPLE 6: TEST_6_MANUAL_EXAMPLE

The example presented in section WRITE CODE TO SETUP THE BOARD. You can probe the signal x(t) at the anaOut1+ and anaOut1- pair of the board.

EXAMPLE 7: TEST_7_PARALLEL_OUT

This simple example shows usages of the analog input, ADC, and parallel output. The analog input channel is connected to an ADC whose output is set to be the parallel output. The parallel output is read and printed to serial monitor. You can try feeding in a square wave of frequency less than 100Hz and amplitude 55mV to anaIn0 +/- pins, and you should get a series of integers which alternates between a cluster of values close to 0 and a cluster values close to 255.

APPENDIX: ARDUINO DUE PINS AND HCDC PINS MAPPING

The Arduino DUE board pinout diagram can be found on the following link:

http://www.robgray.com/temp/Due-pinout-WEB.png

The HCDCv2 computer board and Arduino DUE board pin mapping is recorded in the file "pinDocumentation.txt" located in the same folder.

ANALOG INTERCONNECTIONS BETWEEN THE TWO CHIPS

Use the following codes if you would like to do inter-chip connections:

```
chip 0 output 0 is hardwired to chip 1 input 11 (no inversion)
fab.chips[0].tiles[0].slices[0].chipOutput->in0 ->
fab.chips[1].tiles[1].slices[3].chipInput->out0

chip 0 output 1 is hardwired to chip 1 input 10 (no inversion):
fab.chips[0].tiles[0].slices[1].chipOutput->in0 ->
fab.chips[1].tiles[1].slices[2].chipInput->out0

chip 0 output 2 is hardwired to chip 1 input 9 (no inversion):
fab.chips[0].tiles[0].slices[2].chipOutput->in0 ->
fab.chips[1].tiles[1].slices[1].chipInput->out0

chip 0 output 3 is hardwired to chip 1 input 8 (no inversion):
fab.chips[0].tiles[0].slices[3].chipOutput->in0 ->
fab.chips[1].tiles[1].slices[0].chipInput->out0

chip 0 output 4 is hardwired to chip 1 input 7 (HAS inversion):
fab.chips[0].tiles[1].slices[0].chipOutput->in0 ->
fab.chips[1].tiles[2].slices[3].chipInput->out0

chip 0 output 5 is hardwired to chip 1 input 6 (HAS inversion):
fab.chips[0].tiles[1].slices[1].chipOutput->in0 ->
fab.chips[1].tiles[2].slices[2].chipInput->out0

chip 0 output 6 is hardwired to chip 1 input 5 (HAS inversion):
fab.chips[0].tiles[1].slices[2].chipOutput->in0 ->
fab.chips[1].tiles[2].slices[1].chipInput->out0

chip 0 output 7 is hardwired to chip 1 input 4 (HAS inversion):
fab.chips[0].tiles[1].slices[3].chipOutput->in0 ->
fab.chips[1].tiles[2].slices[0].chipInput->out0

chip 0 output 8 is hardwired to chip 1 input 3 (no inversion):
fab.chips[0].tiles[2].slices[0].chipOutput->in0 ->
fab.chips[1].tiles[0].slices[3].chipInput->out0

chip 0 output 9 is hardwired to chip 1 input 2 (no inversion):
fab.chips[0].tiles[2].slices[1].chipOutput->in0 ->
fab.chips[1].tiles[0].slices[2].chipInput->out0
```

```
chip 0 output 10 is hardwired to chip 1 input 1 (no inversion):
fab.chips[0].tiles[2].slices[2].chipOutput->in0 ->
fab.chips[1].tiles[0].slices[1].chipInput->out0

chip 0 output 11 is hardwired to chip 1 input 0 (no inversion):
fab.chips[0].tiles[2].slices[3].chipOutput->in0 ->
fab.chips[1].tiles[0].slices[0].chipInput->out0

chip 0 output 12 is hardwired to chip 1 input 12 (no inversion):
fab.chips[0].tiles[3].slices[0].chipOutput->in0 ->
fab.chips[1].tiles[3].slices[0].chipInput->out0

chip 0 output 13 is hardwired to chip 1 input 13 (no inversion):
fab.chips[0].tiles[3].slices[1].chipOutput->in0 ->
fab.chips[1].tiles[3].slices[1].chipInput->out0

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

chip 1 output 0 is hardwired to chip 0 input 11 (no inversion):
fab.chips[1].tiles[0].slices[0].chipOutput->in0 ->
fab.chips[0].tiles[1].slices[3].chipInput->out0

chip 1 output 1 is hardwired to chip 0 input 10 (no inversion):
fab.chips[1].tiles[0].slices[1].chipOutput->in0 ->
fab.chips[0].tiles[1].slices[2].chipInput->out0

chip 1 output 2 is hardwired to chip 0 input 9 (no inversion):
fab.chips[1].tiles[0].slices[2].chipOutput->in0 ->
fab.chips[0].tiles[1].slices[1].chipInput->out0

chip 1 output 3 is hardwired to chip 0 input 8 (no inversion):
fab.chips[1].tiles[0].slices[3].chipOutput->in0 ->
fab.chips[0].tiles[1].slices[0].chipInput->out0

chip 1 output 4 is hardwired to chip 0 input 7 (HAS inversion):
fab.chips[1].tiles[1].slices[0].chipOutput->in0 ->
fab.chips[0].tiles[2].slices[3].chipInput->out0

chip 1 output 5 is hardwired to chip 0 input 6 (HAS inversion):
fab.chips[1].tiles[1].slices[1].chipOutput->in0 ->
fab.chips[0].tiles[2].slices[2].chipInput->out0

chip 1 output 6 is hardwired to chip 0 input 5 (HAS inversion):
fab.chips[1].tiles[1].slices[2].chipOutput->in0 ->
fab.chips[0].tiles[2].slices[1].chipInput->out0

chip 1 output 7 is hardwired to chip 0 input 4 (HAS inversion):
fab.chips[1].tiles[1].slices[3].chipOutput->in0 ->
fab.chips[0].tiles[2].slices[0].chipInput->out0

chip 1 output 8 is hardwired to chip 0 input 3 (no inversion):
fab.chips[1].tiles[2].slices[0].chipOutput->in0 ->
fab.chips[0].tiles[0].slices[3].chipInput->out0

chip 1 output 9 is hardwired to chip 0 input 2 (no inversion):
fab.chips[0].tiles[2].slices[1].chipOutput->in0 ->
fab.chips[0].tiles[0].slices[2].chipInput->out0
```

```
chip 1 output 10 is hardwired to chip 0 input 1 (no inversion):
fab.chips[0].tiles[2].slices[2].chipOutput->in0 ->
fab.chips[0].tiles[0].slices[1].chipInput->out0

chip 1 output 11 is hardwired to chip 0 input 0 (no inversion):
fab.chips[0].tiles[2].slices[3].chipOutput->in0 ->
fab.chips[0].tiles[0].slices[0].chipInput->out0

chip 1 output 12 is hardwired to chip 0 input 12 (no inversion):
fab.chips[1].tiles[3].slices[0].chipOutput->in0 ->
fab.chips[0].tiles[3].slices[0].chipInput->out0

chip 1 output 13 is hardwired to chip 0 input 13 (no inversion):
fab.chips[1].tiles[3].slices[1].chipOutput->in0 ->
fab.chips[0].tiles[3].slices[1].chipInput->out0
```