Documentation for FPGA-based digital PID controller

Ryan Thomas

June 5, 2020

1 Introduction

This document is intended to describe how to use the FPGA-based digital PID controller included in this directory. It will begin with an overview of the devices and their connections, then a description of the programmable logic, and finally a description of the two MATLAB classes used to control the device and some examples of their use.

Three different ISE project directories are included: "single-servo", "dual-servo", and "quad-servo". The "single" version is an FPGA architecture with a *single* digital controller which uses the entire RAM for itself. This means that it can record up to 4×10^6 measurements from the ADC. The "dual" version has *two* independent controllers implemented within the FPGA logic, and each one gets half of the memory; therefore, each can store up to 2×10^6 measurements from the ADC. The "quad" version has *four* independent controllers implemented within the FPGA logic, and each one gets one-fourth of the memory; therefore, each can store up to 10^6 measurements from the ADC. Due to time delays on SPI signals, you will need to use slower SPI clock frequencies with the quad controller compared to the single and dual controllers; for instance, use a 5 MHz SPI clock (200 ns period) with the quad control servos.

2 Devices

This section provides an overview of the different devices used with the controller as well as their configuration.

2.1 FPGA

The controller uses a Numato Labs Saturn Spartan 6 module as the FPGA core of the device. The Saturn module consists of a Xilinx Spartan 6 FPGA (part number XC6SLX45 CSG324-3) with 512 Mb of LPDDR RAM. Interfacing with a computer is done through FTDI's FT2232H dual-channel USB device. An on-board 100 MHz oscillator provides the clock signal for the FPGA. On-board SPI flash memory allows the FPGA bitstream to be downloaded directly using a USB cable and software provided by Numato Labs.

2.1.1 Jumper configuration

The Saturn module should be correctly configured out of the box. Check on header P11 that jumpers are set between positions 1-2 and 5-6, and that on header P10 that jumpers are set on positions 1-2 and 5-6. On header K1 (near the power plug), a jumper should connect the middle position and the position closest to the edge of the board.

2.1.2 Driver installation

Drivers for the FTDI chip can be found on FTDI's website. Install the drivers, then plug in the FPGA. You will also need to configure the USB chip so that channel B uses a serial connection. Instructions for how to do this with a Windows computer can be found on Numato Lab's website.

2.1.3 Xilinx software

While most of the code used in this project is VHDL, the project as a whole was created using Xilinx's ISE Design Suite 13.1. The largest issue when moving to a different version of Xilinx's software is that some of the IP Cores, such as the multipliers, the digital clock management (DCM) blocks, and the memory interface generator, may need to be updated. Additionally, ISE 13.1 does not function on Windows 10, and may not function at all on the latest macOS, so you may need to download virtual machine software and install a version of Linux on it to ensure compatibility.

2.1.4 Uploading FPGA architecture

Numato labs provides a utility for uploading FPGA architectures to the on-board SPI flash memory on the Saturn module, so you do not need any fancy cables. This utility can be found on the downloads tab on the Saturn webpage. A different version is included with the project called `saturnflashconfig.exe`. Once downloaded and with the Saturn module connected, open the program and click on the "Load Binary File" button. Find the "topmod.bin" file in the appropriate project, and then click "Program Flash". Occasionally the dialog will say that "Programming failed...", but if you click "Program Flash" again it will almost certainly program the device successfully on the second try.

2.2 AD5791 evaluation board

The actuator signal is the voltage created by an Analog Devices AD5791 20-bit digital-to-analog convertor (DAC). To improve ease-of-use and performance, we use the EVAL-AD5791 evaluation board from Analog Devices. It allows for communication via a serial-peripheral interface (SPI) compatible scheme and there is a header on the board that provides a breakout (header J6) for the necessary signals. The board is powered from both ±15 V (for the output analog circuitry) and +5 V for digital circuitry – these are connected to the board using the two screw terminals. The digital input/output voltage levels can be set with the voltage supplied to the IOVcc pin on the breakout header.

The evaluation board does not come with its own voltage reference, so you will need to either supply a +5 V reference or a +10 V and/or a −10 V reference. The simplest solution is to provide a +5 V reference. Depending on your application, you will want to have either a unipolar 0 − 10 V output or a bipolar −10 to 10 V output. You should set the jumpers on the board as in Table 1.

LK1	Set to position B to source digital power supply from connector J2
LK3	Set to position B to source digital voltage levels from pin 5 of J6
LK4, LK5, LK6	Uninstall to pull up the $\overline{\text{LDAC}}$, $\overline{\text{CLR}}$, and $\overline{\text{RESET}}$ pins
LK8	Set to position A to get +5 V voltage reference from connector VREF.
LK9	Set to position B to get unipolar operation, or set to position C to get bipolar operation from −10 V to 10 V.

Table 1: Header connections for the AD5791 evaluation board.

While noise present in the DAC output is suppressed when used as part of a feedback loop, you will get better performance if you use a low-noise source. However, for testing purposes the +5 V output from a normal power supply will suffice.

2.3 ADS127L01 evaluation board

Measurements are made using a Texas Instruments ADS127L01 24-bit analog-to-digital convertor (ADC) mounted on the manufacturer supplied evaluation board (ADS127L01EVM Rev. B). The ADC can also be programmed using SPI once the Tiva microcontroller on the evaluation board has been disabled. To power the device once without using USB, connect jumpers between JP1 and JP2 and disconnect the jumper on JP3. You will then need to attach +5 V to pin 2 of JP3 and attach a ground to one of the GND test points. Connect the rest of the headers as in Table 2

JP10, JP7, and JP5	Disconnect these jumpers to enable the ADC, the initial differential amplifier, and the clock
JP8	Connect a jumper between pin 2 and the pin labelled "HR/LP".
JP9	Connect a jumper to enable the internal LDO for the LVDD supply
JP11	Connect a jumper between the middle position (labelled DVDD) and the position labelled 3.3 V to set the logic level to 3.3 V.
JP6	Connect a jumper across the pins labelled HR for high-resolution mode.

Table 2: Header connections for the ADC127L01 evaluation board (revision B).

In addition to the headers, the ADC has a number of settings that can be changed using the DIP switch S3. Set these as in Table 3. The filter settings are also changed using switch S3. For a full description of the filter options,

$\overline{\text{HWEN}}$	Set to 0 to use switch S3 to change ADC settings
HR	Set to 1 to use high-resolution mode
FORMAT	Set to 0 to use SPI for communication
FSMODE	Set to 0, although disabled when using SPI

Table 3: ADC settings using switch S3.

you should refer to the ADC data sheet. Changing the filters will change the rate at which data is converted in the ADC and also the signal-to-noise ratio of that data. Note that due to the overhead in the PID loop that the fastest data rate of 512 kSPS is not useable with the PID controller. Our work has used the low-latency filter, which is accessed using FILTER[1:0] = 0b10, with a 512x oversampling using OSR[1:0] = 0b10. With the on-board 16 MHz clock, this gives a sample rate of 31.25 kSPS.

3 Overview

The FPGA programmable logic is defined in behavioural terms using the VHDL language. A schematic of the main components is shown in Fig. 1. The module "topmod" is the top-most level and it is what connects to the physical

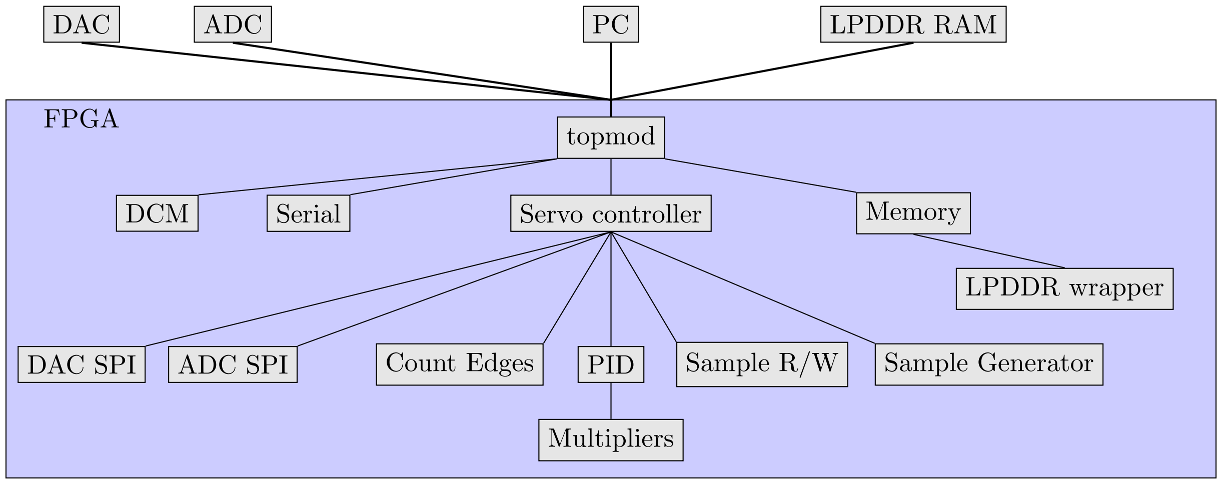

Figure 1: Diagram of the different components comprising the FPGA programmable logic and outside connections.

world through its input and output pins. Within topmod there are four different components: the digital clock management (DCM), the serial interface, the memory controller, and the actual servo controller.

3.1 Top level components

The DCM takes the 100 MHz on-board clock and generates a phase-locked internal 100 MHz clock used for clocking the LPDDR RAM and a 50 MHz clock for clocking everything else.

The serial interface handles communication with the PC using the universal asynchronous receiver/transmitter protocol with a baud rate of 1 MHz. While the FTDI chip can handle baud rates of up to 12 MHz, a rate of 1 MHz was chosen as being the fastest rate with the minimum of lost bytes during transmission. Higher baud rates do not seem to work well, and if faster transmission is required a different protocol should be used. The serial interface assumes that data will be sent in one byte at a time with 1 start bit, 1 stop bit, and no parity bits. Each complete transmission should be four bytes starting with the least-significant byte, and within each byte the data should be sent least-significant bit first. When four bytes have been received, the complete 32-bit word is presented on one of two output signals and a data ready flag is raised for one clock cycle. The assumption behind the serial interface in the FPGA is that most data will be sent first as a 32-bit address and then (possibly) a 32-bit data word. For some commands, such as a software start trigger, it is necessary to send only the address. A flow chart of incoming serial data is processed is shown in Fig. 2. In case a data byte has been lost, the serial interface resets itself after 10^5 baud periods, or 100 ms, if it has not received four bytes in that time.

The memory controller module acts as a simple interface for the LPDDR interface created by Xilinx's Memory Interface Generator. It is suitable for low rates of reading and writing. It is essentially just two state machines that handle read and write requests. A write request is initiated by presenting a 28-bit address, a 32-bit data word to be written, and a write trigger signal that should be high for one 50 MHz clock cycle. The state machine takes care of

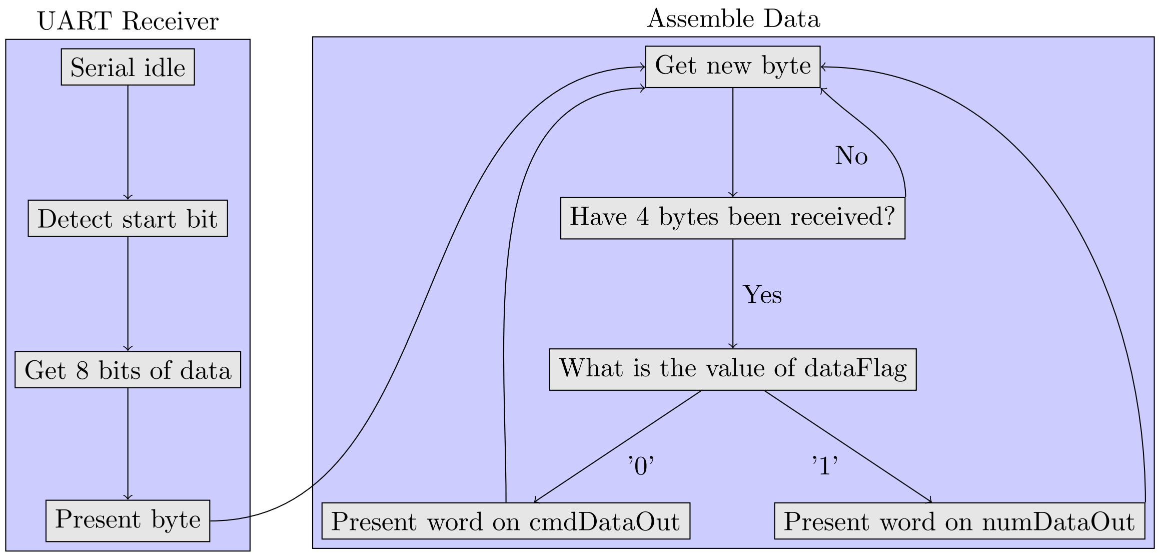
UART Receiver
Serial idle
Detect start bit
Get 8 bits of data
Present byte
Assemble Data
Get new byte
No
Have 4 bytes been received?
Yes
What is the value of dataFlag
'0'
'1'
Present word on cmdDataOut
Present word on numDataOut

Figure 2: Flow chart of serial interface.

writing data to the memory. For simplicity, no writes can take place while read operations are occurring – they will simply be paused until the read operation is complete. A read operation is started by presenting a 28-bit address and a read trigger to the controller. Once the data has been retrieved, it is presented on the output of the memory controller along with a data valid signal that is high for one clock cycle.

3.2 Servo controller components

The servo controller consists of a number of different components and processes along with combinatorial logic. Most of these components are interdependent on each other. A normal sequence for running the servo controller might be as follows. First, a start signal is received either from the serial interface as a specific command or as the rising edge of an external pin. This start signal starts the reading of ADC data using the ADC SPI component. Depending on various settings, this data may be written to memory using the Sample R/W component. At the same time, the sample generator starts. Depending on the sample source, new samples are either created on-the-fly using piece-wise linear ramps or they are read from RAM. Each sample value is associated with an 8-bit sample code; currently only the least-significant bit is used. When this LSB is 0, the corresponding sample is interpreted as a control signal and both this and the measurement signal from the ADC are passed to the PID component which calculates the next value for the DAC. If the sample code's LSB is 1, then the sample is interpreted as a value to be written *directly* to the DAC. This means that the PID process is disabled for this sample. The process continues until the sample generator determines that it has produced as many samples as requested, and data is collected and sent to memory for this period.

A separate process in the servo controller module keeps track of the total time since the start signal. When this value reaches a so-called "off time" then the DAC values from either the PID component or directly from the sample generator are overridden and an "off value" is written to the DAC. This is useful because it may be that one wants to switch off the DAC output with a time resolution better than that achieved using the sample generator. The "off value" is programmable to account for systems that may require a voltage different than 0 V to switch off. After a hold off period a reset command is sent to the sample generator which stops it and resets its state.

Data that was saved to the RAM during this sequence can be retrieved using the serial interface. Data from the RAM is read out as one block – a command is sent to read data and it is streamed as fast as possible to the host computer. As significant amounts of data can be stored, and the serial interface has a limited speed, a starting and ending sample can be specified as well as a sample step size. This allows the user to retrieve only the data range of interest.

The servo controller component defines four constants related to locations in memory. These are listed in Table 4. All memory locations accessed in the servo controller are 28 bits long and the four most-significant bits are always 0b0000. The next two most-significant bits define what is stored in memory, and the rest of the bits define which sample is being accessed.

3.2.1 SPI drivers

The FPGA communicates with both the DAC and the ADC using SPI, and the functionality is encapsulated in the `SPI_DRIVER` component. Although the driver can read and write data at the same time, the DAC driver operates

`SAMPLE_LOCATION`	0b00	Location of sample values
`LOOP1_LOCATION`	0b01	Location of the first loop register comprised of (K_i, K_p).
`LOOP2_LOCATION`	0b10	Location of the second loop register comprised of (N, K_d).
`DATA_LOCATION`	0b11	Location of saved measurement data.

Table 4: Table of memory locations. See Eq. (1) for descriptions of the loop registers.

only in write mode and the ADC driver operates only in read mode. SPI has four different modes of operation which relate to the polarity of the clock signal and the edge of the clock signal on which data is written/read from the slave device. In the driver module, the CPOL generic sets the idle polarity of the SCLK (serial clock) signal: if CPOL is 0, then the idle polarity of SCLK is 0, and if CPOL is 1 then the idle polarity is 1. The CPHA generic controls the edge on which data is clocked into or out of the device. If CPHA is 0, the data is valid on the CPOL $\rightarrow \overline{\text{CPOL}}$ transition, and if CPHA is 1 data is valid on the opposite transition.

The driver only operates when the enable signal is high. A write cycle proceeds when the driver is idle (output port `busy` is low), data is present on the `dataToSend` input port, and a the driver receives a trigger. If the trigger is an external signal that is not necessarily synchronized with the FPGA clock, then there is a generic option to create a synchronized signal on either the rising or falling edge of the input trigger. When the trigger is received, the SYNC signal is pulled to its active level (low for both the DAC and the ADC), the serial clock is started at the given `spiPeriod`, and data is sent to the device in accordance with the CPOL and CPHA parameters. After the final bit of data has been transmitted, the SYNC signal is pulled to its non-active level (high for the DAC and the ADC). An input signal, `syncDelay`, can be used to delay this final transition of the SYNC signal. While not needed for the AD5791, if a different DAC is used – such as the MAX5719 – a delay between the final bit and the SYNC edge may be needed. In the servo controller, triggers for the DAC process are all internal triggers.

The read process occurs in much the same way as the write process. For the servo controller, the start trigger for the SPI process is the falling edge of the $\overline{\text{DRDY}}$ signal from the ADC. Once this edge is registered, the serial clock starts and data from the ADC is clocked in on the appropriate edge. Once all data has been received, the assembled data word is presented on the output of the driver and a data ready signal is raised high for one FPGA clock cycle.

3.2.2 Count edges

The Count Edges component is quite simple: it counts the number of clock cycles between either rising or falling edges of its `edgeIn` signal. This component is used to determine the ADC sample rate using the `ADC_DRDY` signal and report it back to the PC. It can be a useful diagnoses for issues with the ADC clock.

3.2.3 Sample Generator

The sample generator component either calculates new samples as piecewise linear ramps or retrieves these values from memory. The component consists of three processes: Update, LinearRamp, and MemoryRamp. The Update process increments the sample counter and delays new samples by the input signal `updateTime`. On the receipt of a trigger, the generator's enable signal is set to high which is used by other components in the servo controller. The process then loads the appropriate sample data into its active registers depending on the input signal `sampleSource`: if '0', then new samples come from LinearRamp, and if '1' then new samples come from MemoryRamp. An update trigger is then raised high for one clock cycle which indicates to other components that a new sample is valid. The Update process then waits for the time `updateTime` before retrieving the next sample. This process continues until the sample count exceeds the user-specified input signal `numSamples`.

The LinearRamp process calculates new output samples using three arrays of integers: `rampValues`, `rampBndyTimes`, and `rampRates`. At the start of a sequence, the first sample value is `rampValues(0)` and the ramp index `rampIdx` is set to 0. On each update trigger, the sample count is checked against the values of the array of `rampBndyTimes` which define the *boundary times* of the linear ramps. If the sample count is equal to the boundary time corresponding to the current ramp index, then the output sample is set to `rampValue(rampIdx)`. When the sample count is less than the boundary time corresponding to the *next* ramp, then the corresponding `rampRate` is added to the current value of the output sample. The ramp index is incremented when the sample count is equal to the next boundary time *minus 1*.

The MemoryRamp process retrieves new samples from memory as well as new loop parameters. When the process receives an update trigger, it calculates the appropriate memory address from the sample count to retrieve the new sample. The process then waits for the memory controller to indicate that new data is ready before reading that data in. Sample values are interpreted as the least significant 24 bits of the 32-bit data word, and sample codes are the 8 most-significant bits. The controller then reads the updated values for the PID controller into the `loopReg1` and `loopReg2` signals, both of which are 32-bits wide. The MemoryRamp process knows where to retrieve the sample and loop data due to generics in its instantiation. Additionally, this process reads the first sample on the receipt of a reset signal which means that the first sample is always immediately available to the Update process when a sequence is first started.

In addition to sample values, the user provides *sample codes* to either the linear ramp or arbitrary ramp generators. The sample code is an 8-bit word that can control the sample process. Currently only two bits are used. The least-significant bit (LSB) controls whether or not the servo controller should consider the new sample value to be an ADC value to stabilise to (LSB = 0) or a DAC value that should be directly written to the DAC (LSB = 1).

When using arbitrary ramps retrieved from memory, there is also the possibility of *compressing* the sample values in the situation when the sample is held at a particular value for a long period of time. For instance, if one is using the servo to control the current in a magnetic trap that will be held at a particular current for 10 s, then rather than writing 312500 samples to the memory one can write just *two* values: the value itself and how many samples to hold for. This hold phase is enabled by writing a sample where the sample code's most-significant bit is '1' and the sample value is the number of update cycles to delay for.

3.2.4 Sample R/W

This module has two purposes: it records sample data on each edge and keeps track of how many samples have been recorded, and it streams data back to the PC when requested. When the module gets a `dataReady` signal, it issues a write request to the memory controller with the new data and an address composed of the current sample count and the appropriate memory location, in this case `DATA_LCOATION`. At the same time, it updates a counter called `sampleCountMax` that keeps track of how many samples have been recorded so that when reading samples back the entire memory is not read.

When reading data, the controller has four different read modes: data, sample, loop 1, or loop 2, each corresponding to a different memory location as specified in Table 4. Data is streamed directly to the host computer. To limit the amount of data, the user can enable the `useSampleLimits` signal and set the minimum and maximum sample counts to retrieve (inclusive) and the step size to take. To check for errors, the controller will also calculate the signal `numRequestedSamples` which can be read using the serial interface and is useful for checking for missing samples that may have been lost during transmission using the serial interface.

3.2.5 PID

The PID module is what actually implements the proportional-integral-derivative control law. The module calculates a new actuator value u_n at time step n according to

$$u_n = u_{n-1} + 2^{-N}\left[K_p\underbrace{(e_n - e_{n-1})}_{\text{propTerm}} + \frac{K_i}{2}\underbrace{(e_n + e_{n-1})}_{\text{integralTerm}} + K_d\underbrace{(e_n - 2e_{n-1} + e_{n-2})}_{\text{derivativeTerm}}\right] \tag{1}$$

where $e_n = r_n - y_n$ is the error signal, K_p, K_i, K_d are the proportional, integral, and derivative gains, respectively, N is an overall divisor used to rescale the gains to approximate arbitrary values, r_n is the control signal, and y_n is the measurement value. The controller can use either fixed gain and divisor values, or it can use variable values obtained from the sample generator. The fixed gains are input using the signal `loopRegFixed`, and the variable gains are input using `loopRegVary`. Both registers are 64-bits wide, and each of the gains and the divisor are 16 bits wide. The encoding in the registers is, from most-significant bit to least-significant, (N, K_d, K_i, K_p). For the servo controller, the actuator value is the voltage code for the DAC, the measurement is the voltage code from the ADC, and the control signal is a voltage code corresponding to the ADC encoding.

On the receipt of a `measReady` signal, the PID module parses the appropriate loop register depending on the value of `useFixed`, and it also calculates the values of propTerm, integralTerm, and derivativeTerm and stores the last two values of the error signal. The error signal itself is calculated using combinatorial logic as *either* $e_n = r_n - y_n$ for the normal, negative polarity PID controller, or as $e_n = y_n - r_n$ for positive polarity. Positive polarity control is needed in situations where a positive change in the actuator signal causes a negative change in the measurement. The main process then waits for seven clock cycles to let the multipliers compute the new values and provide updated results. Once multipliers have produced updated values, the sum $K_p(\text{propTerm}) + K_i(\text{integralTerm})/2 + K_d(\text{derivativeTerm})$ is used to increment the 56-bit wide signal `pidDivide`. `pidDivide` is clipped at maximum and minimum values provided using the serial interface before being shifted right by N bits to approximate division by 2^N. It is necessary that the division occurs after summation in order to eliminate round-off errors. This shifted value is then converted from a signed integer into an unsigned integer appropriate for the DAC by adding it to a `dacZero` signal and then converting it to the appropriate width. At the end of the process, a `trigOut` signal is raised high for one clock cycle to indicate that the output signal is now valid.

4 Use

This section describes how to use the controller, in terms of connections to the FPGA, how parameters are encoded, and how to use the MATLAB classes. It concludes with examples of how to program the device.

4.1 Connections

The physical pins that each internal signal of the FPGA is connected to can be found in the constraints file of the project. Below is a summary of these connections. For the "single" version of the FPGA architecture, connect signals as for "Servo 2".

Name	**Description**	**Pin on Saturn module**			
		Servo 0	**Servo 1**	**Servo 2**	**Servo 3**
`IOVcc`	3.3 V power on FPGA, connect to IOVcc on DAC	Connect to a Vcc pin, such as P2-3			
`GND`	Ground on FPGA, connect to GND on DAC	Connect to a GND pin, such as P2-7			
`DAC_SCLK`	DAC serial clock, connect to SCLK on DAC	P3-26	P3-28	P2-11	P3-87
`DAC_SDOUT`	DAC serial data out, connect to SDIN on DAC	P3-22	P3-24	P2-15	P3-83
`DAC_LDAC`	DAC load DAC signal, connect to LDAC on DAC	P3-14	P3-16	P2-19	P3-79
`DAC_SYNC`	DAC serial SYNC or CE signal, connect to SYNC on DAC	P3-18	P3-20	P2-23	P3-75
`DAC_SWITCH`	Switch signal. If you have an external switch for the DAC voltage, connect to this.	P3-10	P3-12	P2-11	P3-71
`GND`	FPGA ground, connect to GND on ADC.	Connect to a GND pin, such as P2-7			
`ADC_SDOUT`	ADC serial data out, connect to DIN on ADC	P3-13	P3-15	P2-8	P3-88
`ADC_SCLK`	ADC serial clock, connect to SCLK on ADC	P3-17	P3-19	P2-12	P3-84
`ADC_SDIN`	ADC serial data in, connect to DOUT on ADC	P3-9	P3-11	P2-16	P3-80
`ADC_DRDY`	ADC data ready, connect to DRDY on ADC	P3-5	P3-7	P2-20	P3-76
`ADC_START`	ADC start signal. Leave unconnected	P3-25	P3-27	P2-24	P3-72
`ADC_SYNC`	ADC serial SYNC signal, connect to SYNC on ADC	P3-21	P3-23	P2-28	P3-64
`extStartTrig`	External start signal. Connect to an external, 3.3 V logic start signal	P3-29	P3-31	P3-85	P3-63
`EXT_SWITCH`	External switch input signal. Use if you want to have an external signal control the DAC switch and have its falling edge switch off the controller	P3-30	P3-32	P3-86	P3-59

Table 5: FPGA connections to the DAC and ADC.

4.2 Parameter addresses and encoding

The controller has a number of different parameters that need to be set appropriately for it to work as desired. All of these parameters are accessed using the serial interface. A table of parameters, including their address, data type, and use, is shown in Table 6. The two most-significant address bits are reserved for use as a controller ID, although this is not implemented in the single controller version and thus these bits can be anything. The next two address bits are used to determine whether to read/write data to a parameter or to memory. Their encoding is:

- 00 → write parameter
- 01 → write to memory
- 10 → read parameter
- 11 not used.

Most parameters are written to using a two step procedure. First, an address is sent over serial to the FPGA. Second, some kind of numerical data is sent over. How the numerical data is parsed in the FPGA is indicated in the "Numerical data type" column of Table 6. Parameters where that column is "null" are those where numerical data does not need to be sent; typically, these are trigger signals for which numerical data is unnecessary. Numerical data should be aligned with the least-significant bit, so a 16-bit parameter sent over serial as 32 bits should be 0bxxxx xxxx xxxx xxxx dddd dddd dddd dddd where x is “don’t care” and d are data bits.

Name	**Address**	**Numerical data type**	**Description**
		Manual parameters	
`manDataDAC`	0x00 00 00 xx	24-bit `std_logic_vector`	24 bits to send directly to the DAC. Used for setting up the DAC and setting the DAC voltage manually.
`manTrigDAC`	0x00 00 01 xx	null	Tells the controller to trigger the DAC SPI.
`manSwitchDAC`	0x00 00 02 (00,01)	null	Manually sets the switch signal (if used). If the LSB of the address is 0 then the switch signal is low, if the LSB is 1 then the switch signal is high.
`manTrigADC`	0x00 00 03 xx	null	Tells the controller to manually trigger the ADC to collect `numSamples` samples of data. Useful for debugging.
		Software triggers	
`resetMan`	0x01 00 00 xx	null	Issues a reset command to the sample generator.
`manTrigRamp`	0x01 00 01 xx	null	Issues a start trigger to the ramp generator. A software trigger for the sequence.
`startTransmit`	0x01 00 02 (00,01,10,11)	null	Tells the sample R/W module to start reading back data from memory and sending to the host computer using serial. The last two bits of the address indicate which data to access, and these correspond to the memory locations in Table 4.
		Global parameters	
`spiPeriod`	0x02 00 00 xx	unsigned 31-bit integer	Number of clock cycles between rising edges of the SCLK for both the ADC and the DAC.
`transmitType`	0x02 00 01 xx	unsigned integer [0,3]	Type of data to save to memory and then transmit. 0: ADC data, 1: DAC output, 2: control signal, 3: error signal.
`enableTrig`	0x02 00 02 xx	`std_logic`	Set to high to enable external trigger. This signal is set to 0 on power up to prevent the controller from running without being programmed.
`offTime`	0x02 00 03 xx	unsigned 32-bit integer	Number of clock cycles before the switch is shut and the override DAC process starts.
`ADC_START`	0x02 00 04 (00,01)	null	Sets the value of the `ADC_START` signal, which can be connected to the START pin on the ADC to synchronize the start of conversions if using more than one ADC.
`samplePeriodADC`	0x02 00 05 xx	null	Read-only parameter. Tells the controller to send over serial the current ADC sample period as measured using the Count Edges module.
		DAC settings	
`minValueDAC`	0x03 00 00 xx	unsigned 20-bit integer	Minimum DAC code allowed. DAC codes corresponding to smaller values will be clipped.

`maxValueDAC`	0x03 00 01 xx	unsigned 20-bit integer	Maximum DAC code allowed. DAC codes corresponding to larger values will be clipped.
`syncDelay`	0x03 00 02 xx	unsigned 8-bit integer	Delay between last bit written to DAC and rising edge of SYNC, in clock cycles.
`dacMode`	0x03 00 03 xx	`std_logic`	Polarity of DAC. '0' corresponds to unipolar operation, and '1' to bipolar operation.
`dacOffValue`	0x03 00 04 xx	20-bit `std_logic_vector`	DAC code corresponding to the code the DAC should have when the servo is "off". Written to DAC at `offTime`
`useExternalSwitch`	0x03 00 05 xx	`std_logic`	Uses an external switch line to control both the switch output from the FPGA and the DAC override process. If enabled, the `DAC_SWITCH` signal will be high only when `EXT_SWITCH` is also high, and the override process will start on the falling edge of the `EXT_SWITCH` line.
Sample generator settings			
`sampleSource`	0x04 00 00 xx	`std_logic`	Selects the sample source. '0' is for piece-wise linear ramps, and '1' draws samples from RAM.
`updateTime`	0x04 00 01 xx	unsigned 31-bit integer	Number of clock cycles between successive samples.
`numSamples`	0x04 00 02 xx	unsigned 31-bit integer	Number of samples to write in automatic mode, or number of ADC samples to collect when manually triggered using `manTrigADC`.
Linear ramp settings			
`rampValues`	0x05 00 00 *ii*	32-bit signed integer	Sets the starting value for each linear ramp. *ii* indicates the ramp index to write to. $ii \in [0,7]$
`rampBndyTimes`	0x05 00 01 *ii*	32-bit signed integer	Sets the boundary time for each linear ramp. *ii* indicates the ramp index to write to. $ii \in [0,7]$
`rampRates`	0x05 00 02 *ii*	32-bit signed integer	Sets increment to add at each update time to change the ramp value. *ii* indicates the ramp index to write to. $ii \in [0,6]$
`rampCodes`	0x05 00 03 *ii*	8-bit unsigned integer	Sets the ramp code for each linear ramp. *ii* indicates the ramp index to write to. $ii \in [0,7]$
PID settings			
`polarity`	0x06 00 00 xx	`std_logic`	PID polarity. '0' is negative, '1' is positive.
`Kp`	0x06 00 01 xx	16-bit unsigned integer	Value of the proportional gain.
`Ki`	0x06 00 02 xx	16-bit unsigned integer	Value of the integral gain.
`Kd`	0x06 00 03 xx	16-bit unsigned integer	Value of the derivative gain.
`divisorPID`	0x06 00 04 xx	16-bit unsigned integer	Overall divisor for the PID process. See Eq. (1).
`useFixed`	0x06 00 05 xx	`std_logic`	Set to '0' to use variable loops, and set to '1' to use fixed gain loops.
Memory settings			
`numMemSamples`	0x07 00 00 xx	31-bit unsigned integer	Maximum number of samples written to memory. Use to retrieve sample values from memory.

`useSampleLimits`	0x07 00 01 xx	`std_logic`	Set to '0' to read back all data from memory. Set to '1' to use sample limits (below).
`minSample`	0x07 00 02 xx	31-bit unsigned integer	Sample at which to start read-back.
`maxSample`	0x07 00 03 xx	31-bit unsigned integer	Sample at which to stop read-back.
`sampleStep`	0x07 00 04 xx	31-bit unsigned integer	Step size when reading samples. If this is 0, no read-back will occur.
`lastReqSample`	0x07 00 05 xx	31-bit unsigned integer	Read only parameter. Indicates how many samples should have been sent over UART. Useful for detecting dropped values.

Table 6: Table of parameters in the servo controller. The most-significant four bits of the address assume that "Servo 0" is being addressed and data is being written to the parameter. If the numerical data type is null, then no numerical data needs to be sent.

When writing to memory locations, such as sample data or loop parameters, one needs to set address bits 29 to 28 to be "01". Bits 27 to 26 indicate the memory location as per Table 4, except location "11" is forbidden as the host computer cannot write information to the data storage location. Bits 21 to 0 are then the address in memory to which data should be written. Once the address has been sent over serial, one can send a 32-bit value over serial, and this will be stored in the memory.

4.3 MATLAB drivers

To simplify communication with the servo controller, two MATLAB classes are included. These are the `servoCmd` and `servo` classes. The `servo` class defines an object that represents the servo controller as a whole, with properties and methods that relate to the parameters in the FPGA logic. The properties of the `servo` class that represent parameters inside the FPGA logic are instances of the `servoCmd` class. These parameters are named as in Table 6.

The `servoCmd` class has publicly accessible properties that describe the serial address as a hexadecimal string, the data type of the parameter (which determines how it is sent over serial), and upper and lower limits on the physical value of the parameter. The class has protected properties that are the physical value of the parameter, such as a time in seconds or a value in volts, the integer representation of that value as it appears in the FPGA, and conversion functions that take an integer value to a physical value and vice-versa. Generally speaking, the methods that an end-user will use the most are the set, get, write, and read functions. All other methods are used when defining the parameter as a property of the `servo` class. As an example, suppose that one wants to manipulate the `offTime` parameter. To set the value of a `offTime` to, say, 1.5 s, use `offTime.set(1.5)`. To write this new value to the FPGA, use `offTime.write()` – note that this requires that the `device` property of the `servoCmd` object be set to an instance of class `servo` using the method `servoCmd.setDevice(dev)`. Otherwise, one can set directly specify the serial object to use as `offTime.write(ser)` where `ser` is the serial object. To read the value from the servo controller use the `offTime.read()` method and to convert the integer value to a physical value use the `offTime.get()` method.

Note that the parameters pertaining to linear ramps do not convert from physical units to integer values because one may want these ramps to be specified in terms of a voltage output from the DAC or as a voltage read from the ADC. These two cases require a different conversion function. However, the user can specify conversion functions if they wish using the `servoCmd.setFunctions('to',toIntFuncHandle,'from',fromIntFuncHandle)` method.

The `servo` class mostly contains parameters of type `servoCmd`, and these parameters are manipulated using the `servoCmd` methods. There are additional functions that the `servo` class contains that are used to set up the DAC, manually set the DAC voltage, upload parameters, and write and retrieve data from memory. A summary of these commands is shown in Table 7.

Method name	**Description**
`setID(id)`	Sets the ID of the controller to `id` $\in [0,3]$. Used for multi-controller configurations.
`open`	Creates and then opens the serial connection using the public property `comPort`.
`close`	Closes and deletes the current serial object.
`dacSetup`	Sets up the AD5791 DAC so that is active and uses offset-binary encoding. Finishes by writing 0 V to the DAC.
`dacWrite(V)`	Writes the voltage `V` to the DAC.

`dacSwitch(v)`	Writes the on/off state `v` to the DAC switch signal.
`adcTrig`	Triggers manual collection of data from the ADC. Number of samples controlled by `numSamples` parameter.
`reset`	Resets the FPGA memory pointer.
`start`	Starts the sequence.
`setSampleLimits(minSample,maxSample,sampleStep)`	Sets the sample retrieval limits to `minSample`, `maxSample`, and optionally `sampleStep`.
`setTimeLimits(dt,ti,tf,sampleStep)`	Sets the sample retrieval limits using starting and ending times `ti` and `tf`. The expected temporal separation between successive points must be specified as `dt`, and a sample step can be specified using `sampleStep`.
`read(cmd)`	Issues a read request associated with the command `cmd`. If the number of bytes retrieved is not a multiple of 4, then it retries the read process.
`data = readMemory(readType)`	Reads the memory corresponding to the given `readType`. Allowed values are "sample", "loop1", "loop2", or "data". Returns the data read from the device.
`[data,t] = getMemSamples(cnv)`	Retrieves data under `SAMPLE_LOCATION` in memory. Applies a conversion `cnv` of either "adc" or "dac" to the data to convert from integer values to either ADC voltages or DAC voltages, respectively.
`[Kp,Ki,Kd,n] = getMemLoop`	Retrieves data under `LOOP1_LOCATION` and `LOOP2_LOCATION` in memory, and parses data into the different loop gain parameters.
`[data,t] = getData(cnv)`	Retrieves data under `DATA_LOCATION` in memory. Applies a conversion `cnv` of either "adc" or "dac" to the data to convert from integer values to either ADC voltages or DAC voltages, respectively. Data is stored in the properties `servo.v` and `servo.t`.
`adcStart(v)`	Writes the on/off state `v` to the ADC start signal.
`sampleUpload(V,codes)`	Uploads a series of sample voltages `V` and associated sample codes `codes` to the device. `codes` can be either the same length as `V` or a single element that applies to every sample.
`loopUpload(Kp,Ki,Kd,N)`	Uploads loop parameters `Kp`, `Ki`, `Kd`, and `N` for set-point dependent loop parameters.
`upload`	Uploads all parameters to the device.
`[v,t,c] = simLinRamps(cnv)`	Simulates the linear ramp process with conversion `cnv`. Outputs voltage `v`, time `t`, and sample code `c`. Useful for comparing what you want with what you actually get in case of integer overflow problems.
`dacConvert(in,mode,refVoltage,direction)`	Static method that converts a value `in` from a DAC code to a voltage or from a voltage to a DAC code depending on `direction` (either 'code' or 'volt'). Requires that the `dacMode` be specified in `mode` and the reference voltage.
`adcConvert(valIn,direction)`	Static method that converts an ADC code to a voltage or a voltage to an ADC code depending on `direction` (either 'code' or 'volt').

Table 7: Table of `servo` methods.

Finally, a note about the sample compression feature discussed in Sec. 3.2.3. Sample compression can be enabled by setting the `servo` property `useCompression` to `true`. This will automatically use the static methods `compressSamples` and `decompressSamples` to write and retrieve sample and data values. **Note** that sample compression neglects input checking, so you need to ensure that when varying the loop parameters the loop parameters are static at the same time as the sample values; otherwise, you will get unexpected results.

4.4 Simple example programs

To illustrate how to use the MATLAB classes to control the device, several example programs have been included. In all of these example programs, you must change the `comPort` property to reflect the COM/serial port that the Saturn module uses on your computer. All examples assume that the DAC is set up for unipolar operation with a reference voltage of 10 V. Once each file has been run, the servo can be started using the `start()` method. Data can be retrieved using the `getData('adc')` method.

4.4.1 Linear ramps, fixed gain

The file `programServo_LinSamples_FixedGain.m` demonstrates how to set up the servo controller and generate samples using the internal piece-wise linear ramp generator. The ramp generator is programmed to output a 0 V for 50 ms, a linear ramp from 0 V to 1 V in 100 ms, hold for 500 ms, then ramp down to 0 V in 100 ms, then hold for another 50 ms. Lines 60 and 61 set the sample codes which control whether or not the PID controller is engaged. If line 60 is uncommented and line 61 is commented out, then the DAC output is exactly the values generated by the sample generator. If line 60 is commented out and line 61 is uncommented, then the PID controller is engaged and the DAC output is changed such that the ADC voltage equals the output of the sample generator. Note that the values written to the linear ramp parameters depend on which sample code is used. One can also mix and match sample codes, so that some ramps are written directly to the DAC and others use the PID.

The PID gain parameters on lines 90-95 should provide adequate regulation of the voltage when the DAC output `VOUT_BUF` is connected directly to the ADC input with the signal line on `AINN` and the ground line on `AINP`, since the input is inverted. The overall divisor `divisorPID` (N) is set to $10+5=15$, where the 5 is due to the difference in the voltage scales for the DAC and the ADC. Each increment in the voltage code for the ADC corresponds to $2\times 2.5/2^{24}$ V, while each increment in the voltage code for the DAC in unipolar mode corresponds to $10/2^{20}$ V. The ratio between these two is 2^5.

4.4.2 Arbitrary ramps, fixed gain

The file `programServo_ArbSamples_FixedGain` demonstrates how to set up the servo controller with an arbitrary signal by writing and reading samples directly from memory. The important difference between this file and the one in Sec. 4.4.1 is in lines 58 to 84, where the arbitrary signal is defined. This signal uses a minimum jerk function to change the voltage from 0 V to 1 V in 100 ms, then adds the voltage $0.1\sin^3(2\pi t/(10\text{ ms}))$ to the 1 V signal for 400 ms. The voltage is then changed from the end voltage to 0 V in 100 ms using another minimum jerk trajectory.

Like in Sec. 4.4.1, lines 81 and 82 can be changed to have the samples be written directly to the DAC or used in the PID process.

4.4.3 Arbitrary ramps, variable gain

The file `programServo_ArbSamples_VariableGain` uses the same parameters as in Sec. 4.4.2 but this time the gains are set to be time-dependent. In this example, they are constant as a function of time, but this can be easily changed. Note the different in the `useFixed` parameter (line 55) and the use of the `loopUpload` method on line 85 compared to Sec. 4.4.2.

4.5 Measuring open loop dynamics for a linear system

This example demonstrates how one can use the servo controller to measure the open-loop response of a simple RC filter and then use those measurements to shape the closed-loop response. Set the ADC filter to low-latency with 512x oversampling ratio. In addition to the evaluation boards, you will need a 4.7 kΩ resistor, a 100 nF capacitor, and a decent op-amp (like the OP27G). Connect these as in Fig. 3 The buffer is needed because of the relatively

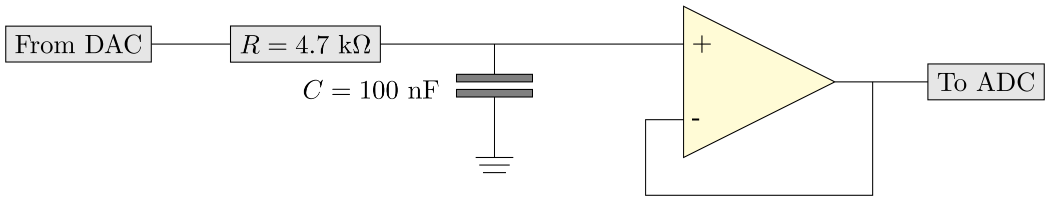

Figure 3: Electrical schematic for measuring the open-loop response of a low-pass filter.

high input impedance of the ADC evaluation board.

The folder "Open loop measurement example" contains the necessary files. To measure the open loop response, run the script `measureOpenLoopLinear`. This will measure the response to modulations in the frequency range 10 Hz to 10 kHz using a 32 μs DAC update period and ADC sample period. The data will be saved in the file "Open loop response.mat" in the same folder. To analyze the resulting data, run the file `calcOpenLoop`. This program will fit sine curves to the measured ADC voltages, using the modulation parameters to constrain the fit, to get the amplitude and phase shift. The amplitude and phase shift is then used in a fit to a second-order frequency response in addition to the known low-latency filter response (see the ADC datasheet). The second output argument from the `calcOpenLoop` function, pFilter, is a vector with the second order response parameters

$$G(\omega) = \frac{G(0)}{1 + i\frac{\omega}{\omega_1} - \frac{\omega^2}{\omega_2^2}} \tag{2}$$

where pFilter(1) $= G(0)$, pFilter(2) $= \omega_1/(2\pi)$, and pFilter(3) $= \omega_2/(2\pi)$. $G(0)$ should be very close to 1, and for the RC filter in Fig. 3 $\omega_1 \approx 338$ Hz. The term ω_2 is included to account for non-ideal effects in the buffered RC filter. An example of the open loop measurement is shown in Fig. 4. Here, the ADC filter was set to a low-latency

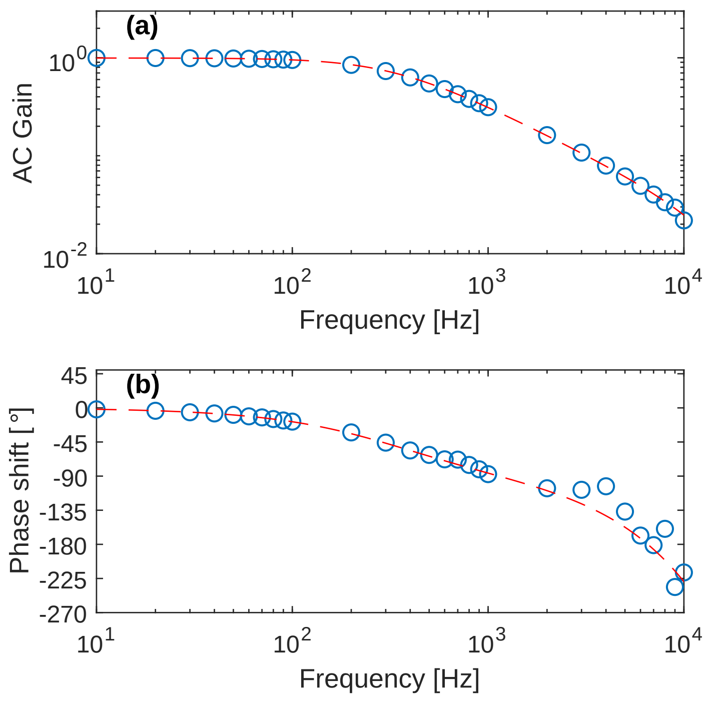

Figure 4: Bode plot of the open loop response of the buffered RC filter when measuring using the ADC with a low-latency filter and 512x OSR. Blue circles are from measurements, red dashed line is a fit to $G(\omega)M(\omega)$. Here, $G_0 = 0.9945$, $\omega_1/(2\pi) = 324.7$ Hz, and $\omega_2/(2\pi) = 2611$ Hz. **(a)** Amplitude of the response as a function of frequency. **(b)** Phase shift as a function of frequency.

filter with 512x OSR. From equation 4 on the ADC datasheet, the filter response is

$$M(\omega = 2\pi f) = e^{-2\pi i f T_s} \left| \frac{\sin\left(\frac{32\pi f}{f_{\text{CLK}}}\right)}{32 \sin\left(\frac{\pi f}{f_{\text{CLK}}}\right)} \right|^5 \left| \frac{\sin\left(\frac{32 N_s \pi f}{f_{\text{CLK}}}\right)}{N_s \sin\left(\frac{32\pi f}{f_{\text{CLK}}}\right)} \right| \tag{3}$$

where f_{CLK} is the ADC master clock frequency (16 MHz), and N_s is the second-stage oversampling ratio equal to the OSR setting divided by 32, OSR/32.

Using the measured $G(\omega)$, the response of the servo to any set of gains for Eq. (1) can now be calculated. Alternatively, one can calculate the gains needed to approximate a given response. Suppose that we want our closed loop response to be $T(\omega)^{-1} = 1 + 1i\omega/\omega'$ where ω' is the target cut-off frequency for a low-pass filter – i.e., we want to change the cut-off frequency of our RC filter. If the control law is $K(\omega)$, then the closed-loop response is

$$T(\omega) = \frac{G(\omega)K(\omega)M(\omega)}{1 + G(\omega)K(\omega)M(\omega)} \tag{4}$$

and thus we can calculate the needed control law if we know $T(\omega)$:

$$\begin{aligned} K(\omega) &= \frac{T(\omega)}{1-T(\omega)} G^{-1}(\omega) M^{-1}(\omega) \\ &= \left(\underbrace{\frac{\omega'}{\omega_1 G(0)}}_{K_p} + \underbrace{\frac{\omega'}{G(0)}}_{K_i} \frac{1}{i\omega} + \underbrace{\frac{\omega'}{\omega_2^2 G(0)}}_{K_d} i\omega \right) M^{-1}(\omega). \end{aligned} \tag{5}$$

Due to the complex nature of $M(\omega)$, in calculating the gain parameters of $K(\omega)$ we typically assume that $M(\omega) \approx 1$ which is accurate for $\omega T_s \ll 1$. Simpler measurement responses could be incorporated into $G(\omega)$ to form an effective open-loop response $\tilde{G}(\omega) = G(\omega)M(\omega)$. Note that to convert from a continuous description of the control law to a discrete version time needs to be measured in units of T_s.

The script file `measureClosedLoopLinear` measures the closed-loop response of the buffered RC filter using four different target cut-off frequencies for $T(\omega)$: $\omega'/(2\pi) = 250, 500, 1000,$ and 2000 Hz. The data collected by this script can be analysed using the file `calcClosedLoop`, which uses the function `PIDsim` to model the system response. Results are shown in Fig. 5. As both the ADC and DAC are well-characterized, and the FPGA latency is well-known,

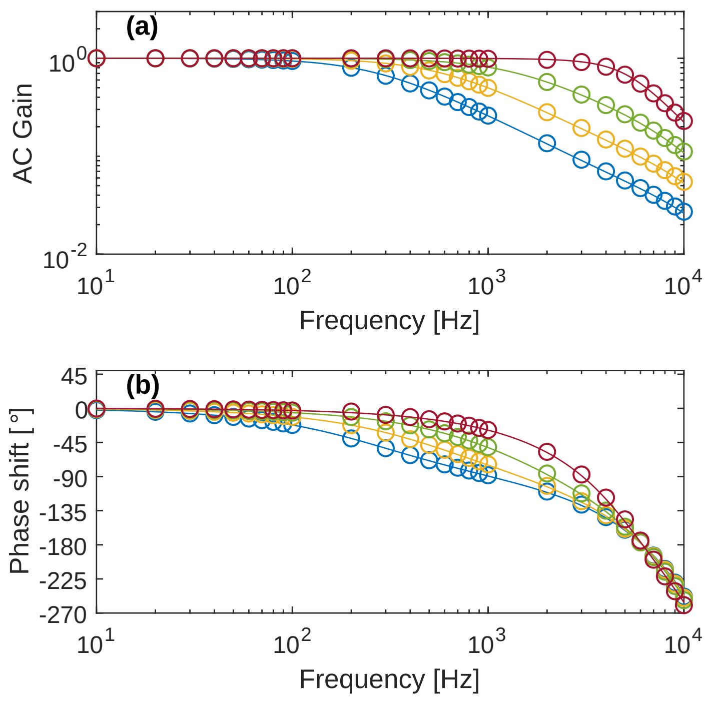

Figure 5: Bode plot of the closed loop response with different target low-pass cut-off frequencies of 250 Hz (blue), 500 Hz (yellow), 1000 Hz (green), and 2000 Hz (red). Open circles are the measured response, and solid lines are the modelled response based on the open loop parameters and the servo controller's PID calculation. **(a)** Amplitude of the signal as a function of frequency. **(b)** Phase shift of the signal as a function of frequency.

the model of the controller is a good match to the measurements. This model can thus be used to determine the response of the controller to any set of PID gains or when connected to a system with different open loop dynamics. We define the closed-loop response as

$$T(\omega) = \frac{L(\omega)}{1+L(\omega)}$$

where $L(\omega)$ is the so-called loop gain equal to

$$L(\omega) = K(\omega)H(\omega)G(\omega)M(\omega)$$

where $G(\omega)$ is the open-loop response, $M(\omega)$ is the measurement response, $H(\omega) = e^{-i\omega T_{\text{lag}}}$ is the phase shift due to the FPGA latency of $T_{\text{lag}} \approx 6$ μs. The control law $K(\omega)$ is modelled as

$$K(\omega) = \left[\left(K_p + \frac{K_i}{2} + K_d \right) + \left(-K_p + \frac{K_i}{2} - 2K_d \right) z^{-1} + K_d z^{-2} \right] \left[1 - z^{-1} \right]^{-1} \tag{6}$$

where $z^{-1} = \exp(-i\omega T_s)$.