Washington University in St. Louis

RichArduino V4.0 Microcontroller System Design

CSE 462M Spring 2022 Semester Project

Tomson Li

CSE 462M: Computer Systems Design

Dr. William Richard

6 May. 2022

Abstract

The paper describes a microcontroller system that can be used to digitize input analog signals and output digitized data. The system is based on RichArduino V3.0 design, which uses a Digilent CMOD A7 module that has a RSRC CPU, a pins module, and a UART serial interface that allows us to communicate with the microcontroller. We implement a wrapper design for the XADC that is integrated on to the die of the FPGA in the CMOD to digitize input analog signals. We then design the I^2C serial interface based on the UART design to transfer data out to the DAC that we soldered on our custom shield. We also build a PCB board which is our custom shield that we could test and demonstrate the functionality of our RichArduino V4.0 design. In addition, we programmed a UART demo app to communicate with the microcontroller using our computer and we write the assembly program based on the V3.0 assembly code to support our UART demo app. The design is developed for educational purposes which is finished in reasonable time frame and therefore limitations exist. Further development and design could improve the performance of the microcontroller.

I. Introduction

The RichArduino V4.0 is based on the RSRC CPU that we have been used in previous coursework. We implement an I^2C interface to transfer data out to DAC and a wrapper design to the XADC on the Xilinx Artix-7 FPGA die. We also design a custom shield PCB board that we could plug our CMOD A7 module on to the shield to evaluate and demonstrate our design. In terms of software, we write an assembly code and stored in our EPROM to serve as our bootloader. In addition, we implement a host downloader (UART Demo App) that we could interact with our RichArduino V4.0. The end goal is to upload a program in assembly using the host downloader and demonstrate the functionality where we could digitize incoming analog signal using XADC and output the data to the DAC on our shield where it can generate the same analog signal.

II. RichArduino Microcontroller System Architecture

The RichArduino V4.0 is built based on the RichArduino V3.0 design. The full block diagram of the system shows the content of Testbench that includes work that was done by the previous iteration of the course [Fig. 1]. The RSRC CPU, EPROM and SRAM are provided as the basic controller architecture. The PINS module and the serial UART interface were previous contribution of the course. We utilize the UART module to debug and upload our program to the microcontroller. The DCM (Digital Clock Manager) is used to provide a 48 MHz clock to the system based on the input clock of 12 MHz that is on the CMOD A7 board.

The I^2C serial interface and the XADC wrapper is our focus design this year. The I^2C module has 2 bit of address lines, 32 bit of data lines, a chip enable, a write enable, an output enable, an input clock from the DCM, a reset signal and a pair of data lines out to the I^2C devices. The SCL is the serial clock line and SDA is the serial data line. Although I^2C slave devices can drive the SDA, in our application, the microcontroller drives both SCL and SDA, except when the slave device ACK, where the slave drives SDA for very short period. The microcontroller output a 400 KHz clock on the SCL as it is the max speed support by the I^2C slave device. Two bit of address lines allow us to have 4 WORD of address space or 4 registers. We utilize three of them: a BUSY flag register [Fig. 2], an Output Register [Fig. 3], and an ACK register [Fig. 4]. The BUSY flag will be set when the transmitter is sending data. The transmitter will start sending when Output Register is set, and the ACK register will be set when slave device ACK at the right time. The I^2C bus is byte oriented and our slave device needs to receive three bytes, therefore there are total of three ACKs. Our slave device is a 10-bit I^2C DAC, according to its data sheet [1]. In order to write to the DAC, master device (RichArduino) needs to write three bytes of data and the DAC will ACK each time it successfully receives a byte. In

my design, I implement the I^2C a word-based version, where the output register receives all three bytes that will be sent and send the data bits in sequence all at once. The ACK register then stores all three ACK bit in one register.

The XADC wrapper creates a module that wraps the XADC LogiCore IP and maps the corresponding pins out to the testbench. The configuration for the XADC IP is shown in [Fig. 5] where we use DRP as our data interface and Single Channel mode for the operation of the XADC. We also enable the reset_in pin and set the timing to Continuous Mode so that we can get a sample any time we want. For the DRP timing, we set the DCLK frequency to match our DCM frequency of 48 MHz, and the ADC Conversion Rate is set to the lowest possible value, 39 KSPS. Our I^2C DAC is only capable of running at 400 KHz where it acquires 3 Bytes or 16 Bits per sample. Through calculation, 39 K samples per second is still much more than the DAC can handle. We also set the acquisition time to the lowest possible value and not check the Bipolar selection since our circuitry is running in unipolar mode [Fig. 6]. The diagram shows two analog signals where we can switch between the two. AN15 is connected to the electret microphone on our shield and AN16 is connected to the coaxial port on our shield. My implementation also adds a done signal where it is not shown on the block diagram. According to the XADC user guide [2], the digitized data from the XADC should only be captured when the DRDY signal is set and therefore a done signal should be used to tell the CPU that the data is ready. More details on the implementation will be discussed in the VHDL overview section.

III. RichArduino Microcontroller FPGA VHDL Overview

We use Xilinx Vivado CAD tool to design our internal circuit inside the Artix-7 FPGA. VHDL (Very High-Speed Integrated Circuit Hardware Description Language) uses RTL (Register-transfer level) abstraction to create high-level representations of circuitry, where it can derive lower-level circuit wiring using logic gates. It usually contains registers that hold data, combination logic that defines state input and clocks that control state changes.

In our design, we reference many of the VHDL code from the VHDL tutorial document [3] that Prof. Richard provides to us. On the top level, we have a testbench that holds all the components that is shown in the block diagram. The structural VHDL of the testbench will instantiate lower-level components. Testbench has an input clock from the CMOD A7 board, a reset button input, serial in and out that is used in the UART module, io pins that is used in the PINS module, SCL and SDA used in I^2C module and VAUX analog pair input from analog pins. We declare each component that we need to port into the testbench, and we tie those wires from each component to the wires in the testbench using PORT MAP. We also use SIGNALs as internal wires. In addition, we create a memory map [Fig. 7] and tie the specific memory space to each component through their chip enable signal. When the CPU receives specific address, it will enable the corresponding component in testbench. Our EPROM component is mapped to the bottom of the memory, at location 0000 0000H, having 1K word or 4KB. Therefore, we put our bootloader in the EPROM module so that it will start running our bootloader program when the system startup. SRAM is mapped on top of the EPROM, starting at location 4096, also have 4KB. PINS module is at the top of memory map, at location -16, having 16B. UART module is one down below the PINS module, at location -32, having 16B. I^2C is one down below the UART module, at location -48, having also 16B. Between XADC and I^2C, there is hole because

we would like to map XADC at the multiple of its address space, which 512B. Therefore, XADC is mapped at location -1024, having 512B of address space.

In our implementation of the I^2C component, we use the basic code layout that was developed for the UART module. We use counter to generate a 400 KHz clock. Our system clock from DCM is 48 MHz, we cannot directly use our system clock, so we have to use counter to manually generate a slower clock in our I^2C component. According to the I^2C Specification Sheet [4], SCL and SDA have pull up resistors. This indicate that both SCL and SDA are open drain circuits. Therefore, when we indicate a logic one or high in VHDL for SCL and SDA lines, we use 'Z (High Impedance)' instead of '1'. When we start transmitting data, we first drop SDA low then drop SCL low to indicate a start bit. During our transmission, we make sure that SDA is only changed when SCL is logic low, and SDA stays stable when SCL is high. We first send the address byte that is given on the datasheet and look for an ACK bit. Then we send another byte that includes two power down option bits and four data bits. Finally, we send the last byte that includes the last six data bits. When we finish transmitting, we first pull SDA high then pull SCL back to high to indicate a stop bit. We also make sure that the timing requirements is met in our design.

For our XADC wrapper, we first instantiate the XADC IP in the VHDL. The tool will generate an instantiation template file that we can use to declare and instantiate the XADC IP component. Since we already set the corresponding clock frequency in the configuration GUI [Fig. 5], we can directly tie the DCLK to our system clock in our wrapper. We are also only reading samples from the XADC but not writing to it. Therefore, we feed all zeros to the input data bus and tie a logic zero to the write enable pin to indicate read operation. The VP pair is also tie to logic zero because we are using auxiliary analog input pair instead of the dedicated analog

input pair. We do not instantiate all the ports for the IP that we do not necessarily need for simplicity. In my implementation, I add a tristate buffer and a done signal because the data coming out from the XADC is only valid when it indicates that the data is ready. Therefore, we should only pass the valid data out to the main data bus when the data is ready. As our reset_l signal is active-low signal and the reset_in signal for the XADC is active-high, I also add a synchronous flip-flop to revert our reset signal.

IV. RichArduino Boot Monitor

The boot monitor program serves our bootloader of the microcontroller where we put the program in the EPROM, so it runs at startup. The boot monitor is capable of downloads a binary RSRC assembly program and executes (P), peek (R) and poke (W) for the specific register at specific address and check current microcontroller version (V), which is V4.0 for this semester. The UART serial interface makes the boot monitor possible to communicate with our host device, which is our PC. Previous course work has developed the functionality of downloads the assembly program and executes it. Our boot monitor is built based on Meagan Konst's boot monitor code [5].

The basic concept for each functionality is similar. Any time we want to read from the UART, we first check if the RX data flag is set. RX data flag will tell whether there is incoming data through the UART. Then, the program checks the incoming character from the host to see which function to proceed. If the character is P, then it will branch to the section where it starts to download assembly binary program and executes it. The program will be stored into our SRAM, which is located at address 4096 and then branch to address 4096 to start executing program in the SRAM. If the character is W, then it will branch to poke section. It will first receive a 32-bit address, one byte at a time. Then receive a 32-bit data, also one byte at a time. Both address and data are in big-endian format. It would then write the 32-bit data to the 32-bit address. Whenever we need to send data, we also need to check the TX busy flag. TX busy flag will tell whether the microcontroller is still sending data through the UART. If the character is R, it would receive a 32-bit address and read the data from the address. Then it would send the 32-bit data back to host. If the character is V, then it would simply send back 04H, which indicates current version number is V4.0. We use the SRC Simulator [6] to compile our assembly program

into binary where we can use it to download to our microcontroller. Our boot monitor is converted from the binary file into eprom.vhd file as it is hard coded into the EPROM module when we use Vivado the wire our internal logic circuits. The full code block with comments is attached to the bottom [Fig. 8].

V. Host Downloader

To communicate with our microcontroller, we also need to program a host downloader on our PC. Python has many great libraries to build simple serial communication GUI app therefore I choose to use python to write my host downloader (UART GUI App) [Fig. 9]. One of the pre-installed GUI libraries is tkinter. For a simple serial comm app, tkinter is sufficient. For serial communication programming, python also pre-installed the serial library that we can use to do UART communication. Our microcontroller has hard coded UART baud rate of 115200 with no parity bit and one stop bit. Therefore, I can also hard code these properties in my host downloader. However, the COM port number is not hard coded as it might be changed every time and that is also useful in real-life application.

For each functionality, it would first write a single character to notify the microcontroller which function that will be used. For example, it would write ascii character ‘W’ to indicate poke function. Then it would start writing 32-bit address and 32-bit data in big-endian format to the microcontroller. All write operations use similar method except the download program function will need to have extra steps to parse the binary file, calculate the size of the binary file, and send the data line by line. The host downloader has a separate thread that monitors all the incoming data from the microcontroller because the read function is a blocking function. It will convert all incoming data into hexadecimal form and display it on the receiving data section. All the input form for sending is also in hexadecimal form.

The download program functionality supports both direct path input of a binary file or browse file explorer to select a binary file for simplicity [7]. Information window would also pop out when COM port is not connected or unable to open. Read operation might also pop out information window to indicate exception when reading from the UART.

VI. RichArduino Demo Shield Architecture

Our host downloader in a way demonstrates that our RSRC CPU, EPROM, SRAM, and UART works. To assess the functionality of other module, we build a PCB board to serve as our demo shield that we can plug our CMOD A7 board into it. We sketch our schematic based on the demo shield block diagram [Fig. 10]. We use a LDO (Low-drop regulator) voltage regulator to convert our 5V voltage from the CMOD A7 board to 3.3V. Electret microphone and a preamplifier is used to feed in audio analog signal to one of our analog input pins. Another analog input would be from the coaxial port. One the output side, we have a I^2C DAC as well as an LED. The LED is only for testing purposes and the I^2C DAC is used to output our received digitized signal. We use ExpressSCH to sketch the schematics and ExpressPCB to draw the layout of the PCB board since our manufacture service is also provided by ExpressPCB.

In the full detailed schematic [Fig. 11], we first sketch the circuit for the op amp and the electret microphone. By reading the datasheet of the MAX4466 preamplifier and the electret microphone [8, 9], we can apply the recommended application circuit from the datasheet to our schematic, including microphone bias network circuit, preamplifier bypass and supply filter circuit. Then we also complete the circuit for the LDO using the recommended circuit drive on its datasheet [10]. For the I^2C DAC, the datasheet recommends that supply should be decoupled with 10 µF in parallel with a 0.1 µF capacitor to GND [1]. The specification of the I^2C serial bus also gives recommendations on the value of the pull up resistors. Here we use 2K resistors as our pull ups. We also tie A0 pin to GND for I^2C addressing. For all the supply filtering, we add ferrite beads in parallel to our VCC to filter high frequency noise. By reading the schematic of the power supply on the CMOD A7 board [11], we think that ferrite beads are necessary for our circuit.

After finishing schematic, we can draw the layout for the PCB board for fabrication. We follow the specifications of the ExpressPCB Miniboard Pro 4 layer [12] as it would our actual fabrication standard. The layout of the PCB is basically following the schematic with some design consideration on the wiring such as trace impedance, the size of the pins, and the size of those components. We also need to make a custom component for the preamp as its package standard is not in the library of the ExpressPCB tool. All my components are on the top layer of the PCB board where only one trace is on the bottom layer because the orientation of SCL and SDA pins on the DAC is the opposite to our CMOD A7 pins layout.

VII. RichArduino Demo Shield Demo Program

The shield demo program is also programmed and compiled using the SRC Simulator [6]. The program starts at address 4096 as it will be stored into SRAM when the bootloader downloads the program. The main function of the program is reading the XADC data register then check the UART busy flag. If the UART is busy, then it would continue branch back to read XADC data register until the UART is not busy. Once UART is not busy, it would convert the data from the XADC to the data that the would be needed. The XADC output 12 bits of data but the DAC needs only 10 bits of data. Therefore, we need to convert it down to 12 bits and set the corresponding power down mode bits to zeros. Lastly, we send the data with the right format by storing the data in the specific register at the address for the UART output register. And then branch back to continue looping the entire process.

VIII. Results/Discussion

We first solder all the components on the PCB board before we complete our microcontroller design. We apply little solder paste on all the pads and put the corresponding components on the pads. We then heat it, and those components would recenter themselves to appropriate position thanks to the soldering mask on the PCB board. We also solder a 48 DIP socket in the through holes because we want to leave more flexibility on the CMOD A7 board where we can take it off if we use the socket. We also end up cleaning up some solder where it might cause shorts on some the small IC package pins using microscope. The finished board [Fig. 12] have solid performance for what we need to accomplish. The supply voltage from the LDO is stable and the performance of the DAC as well as the preamplifier are also as expected.

During initial testing, we first test the functionality of the I^2C interface by sending the right data bits to the I^2C DAC. When send maximum value to the DAC, we can measure that the DAC is outputting maximum voltage, which is 3.3V in our case. We then tie the input analog signal pin to the input of the microphone preamp. We are able to watch the output register changes as we speak to the microphone. For the final completion of the project, we tie the input analog signal pin to the input of the coaxial port, which allows us to get a sine wave from a function generator into the coaxial port and output the same sine wave by probing the output of the DAC on an oscilloscope.

When we first design the I^2C interface, we use roughly the same frequency as the 115200 baud rate UART module in our counter. It works but it does not max out the performance of the DAC. In addition, when we probe it under the oscilloscope, we can see clearly that the output of the DAC does not give a very continuous wave because the frequency was too low. After

remodifying the I^2C design to max out its performance, we can see a much more beautiful sine wave coming out of the DAC based on the input.

One of the possible outcomes of the project is to digitize the analog audio signal and transmit the data using a faster speed UART design to our PC. In my opinion, this idea is much cooler than what I have finished. If we had more time, I would definitely adopt this demo idea. It would involve more research and learning on the XADC setup, and we would look more into the audio signal that we transmit to our PC for sampling. Our current design only looks into the minimum of the XADC function, where we only use single channel and not use any of the calibration, averaging or multiplexer settings. However, take into account of the time we have for this semester to complete the project, we have done much to learn a lot on computer systems design.

IX. Conclusions

The RichArduino V4.0 is a good educational project that we complete for this year's capstone design. We managed to learn and implement one serial communication protocol, the I^2C interface and learn to implement the onboard XADC IP. We tried to finish the project with the highest standards possible with the limited time we have. With hands on PCB wiring and interacting with ICs, we can learn more about real-life computer system design applications. There are a lot more things we need to consider into our design besides all the theories that we learn from circuit class or logic class. I think the only downsides is that our time is so limited that we cannot get to learn more about PCB using better CAD tools such as Cadence. We could not also get to do more fun things like digitizing audio analog signal and transmit it through UART to our PC. Overall, the project is fun, and I like it. If we could do a little more than what we have accomplished, that would be even better.

X. References

[1] 2.5 V to 5.5 V, 120 μA, 2-Wire Interface,Voltage-Output 8-/10-/12-Bit DACs Data Sheet, Analog Devices

[2] 7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide, Xilinx, Inc.

[3] EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT SYNTHESIZABLE VHDL BUT WERE AFRAID TO ASK, William D. Richard, Ph.D.

[4] The I^2C-BUS Specification, Version 2.1, Philips Semiconductors

[5] Boot Monitor Code Developed by Meagan Konst

[6] SRC Simulator (SRCTools Version 3.1.1), Computer Systems Design and Architecture, 2nd Edition, Heuring and Jordan

[7] "File explorer in python using Tkinter," GeeksforGeeks, 15-Feb-2021. https://www.geeksforgeeks.org/file-explorer-in-python-using-tkinter/.

[8] Low-Cost, Micropower, SC70/SOT23-8, Microphone Preamplifiers with Complete Shutdown Data Sheet, Maxim Integrated

[9] PUI Audio's omnidirectional electret condenser microphones Data Sheet, PUI Audio, Inc.

[10] 150mA Low Noise μCap CMOS LDO Data Sheet, Micrel Semiconductor

[11] Cmod A7 Schematic Rev B.1, Digilent, Inc.

[12] ExpressPCB Manufacturing Specifications, ExpressPCB, LLC

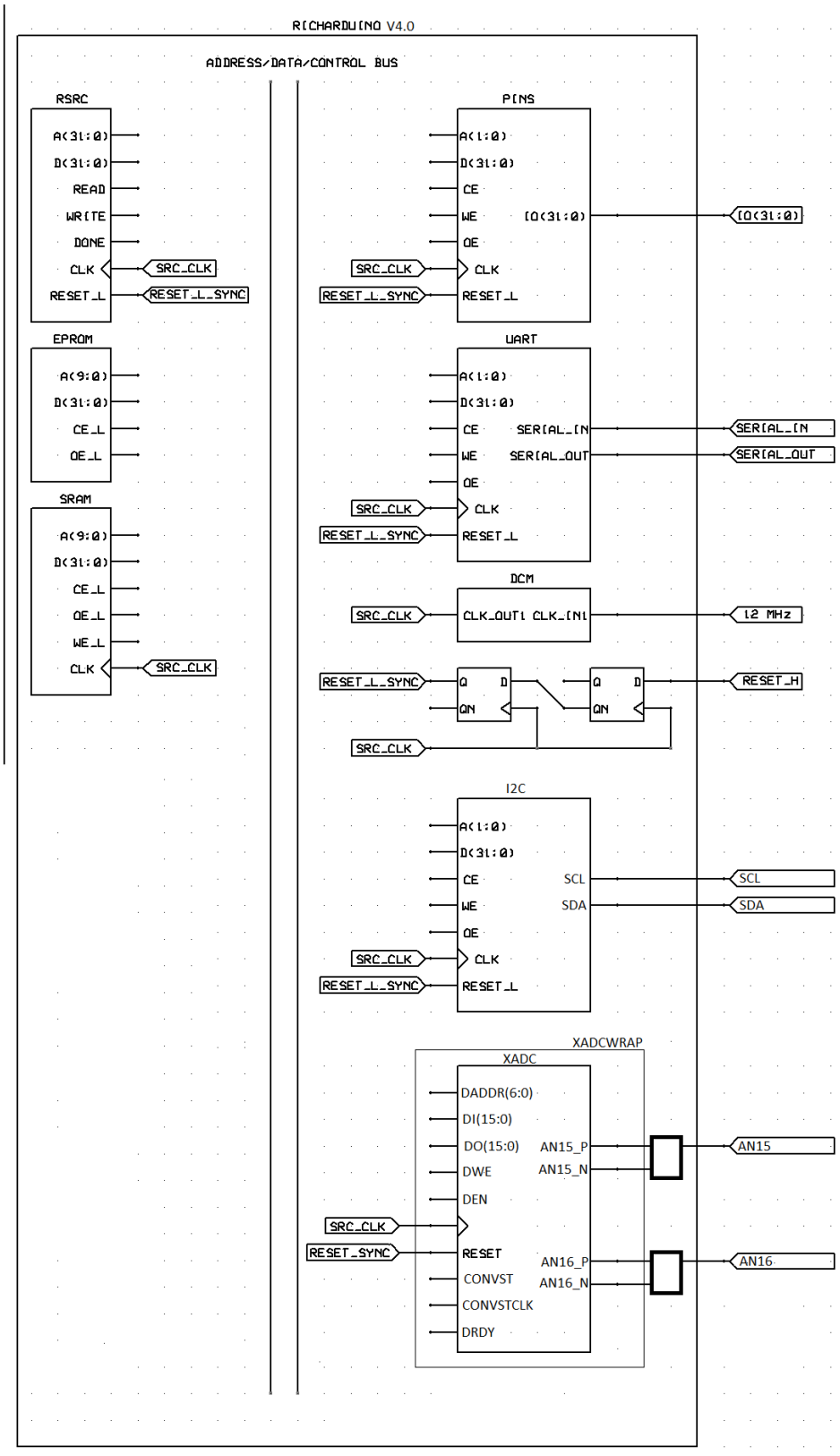

Fig. 1 RichArduino V4.0 Full Block Diagram

Address FFFF FFD0H – I2C BUSY Flag Register

BIT	TYPE	FUNCTION	DEFAULT	COMMENT
Bit 31	R	RESERVED(31)	0	
Bit 30	R	RESERVED(30)	0	
Bit 29	R	RESERVED(29)	0	
Bit 28	R	RESERVED(28)	0	
Bit 27	R	RESERVED(27)	0	
Bit 26	R	RESERVED(26)	0	
Bit 25	R	RESERVED(25)	0	
Bit 24	R	RESERVED(24)	0	
Bit 23	R	RESERVED(23)	0	
Bit 22	R	RESERVED(22)	0	
Bit 21	R	RESERVED(21)	0	
Bit 20	R	RESERVED(20)	0	
Bit 19	R	RESERVED(19)	0	
Bit 18	R	RESERVED(18)	0	
Bit 17	R	RESERVED(17)	0	
Bit 16	R	RESERVED(16)	0	
Bit 15	R	RESERVED(15)	0	
Bit 14	R	RESERVED(14)	0	
Bit 13	R	RESERVED(13)	0	
Bit 12	R	RESERVED(12)	0	
Bit 11	R	RESERVED(11)	0	
Bit 10	R	RESERVED(10)	0	
Bit 9	R	RESERVED(9)	0	
Bit 8	R	RESERVED(8)	0	
Bit 7	R	RESERVED(7)	0	
Bit 6	R	RESERVED(6)	0	
Bit 5	R	RESERVED(5)	0	
Bit 4	R	RESERVED(4)	0	
Bit 3	R	RESERVED(3)	0	
Bit 2	R	RESERVED(2)	0	
Bit 1	R	RESERVED(1)	0	
Bit 0	R	I2C_BUSY_FLAG	NA	

This 32-bit read-only register supplies the I2C_BUSY_FLAG in the least significant bit when read.

Fig. 2 I2C Busy Flag Register Notation

Address FFFF FFD4H – I2C Output Register

BIT	TYPE	FUNCTION	DEFAULT	COMMENT
Bit 31	W	RESERVED(31)	NA	
Bit 30	W	RESERVED(30)	NA	
Bit 29	W	RESERVED(29)	NA	
Bit 28	W	RESERVED(28)	NA	
Bit 27	W	RESERVED(27)	NA	
Bit 26	W	RESERVED(26)	NA	
Bit 25	W	RESERVED(25)	NA	
Bit 24	W	RESERVED(24)	NA	
Bit 23	W	RESERVED(23)	NA	
Bit 22	W	RESERVED(22)	NA	
Bit 21	W	RESERVED(21)	NA	
Bit 20	W	RESERVED(20)	NA	
Bit 19	W	RESERVED(19)	NA	
Bit 18	W	RESERVED(18)	NA	
Bit 17	W	RESERVED(17)	NA	
Bit 16	W	RESERVED(16)	NA	
Bit 15	W	I2C_DATA(15)	0	
Bit 14	W	I2C_DATA(14)	0	
Bit 13	W	I2C_DATA(13)	0	
Bit 12	W	I2C_DATA(12)	0	
Bit 11	W	I2C_DATA(11)	0	
Bit 10	W	I2C_DATA(10)	0	
Bit 9	W	I2C_DATA(9)	0	
Bit 8	W	I2C_DATA(8)	0	
Bit 7	W	I2C_DATA(7)	0	
Bit 6	W	I2C_DATA(6)	0	
Bit 5	W	I2C_DATA(5)	0	
Bit 4	W	I2C_DATA(4)	0	
Bit 3	W	I2C_DATA(3)	0	
Bit 2	W	I2C_DATA(2)	0	
Bit 1	W	I2C_DATA(1)	0	
Bit 0	W	I2C_DATA(0)	0	

This 32-bit write-only register supplies data to the I2C transmitter when written in the lower 16 bits. These 16 bits are the data in the second two bytes of the transfer. The contents of the first byte are hard coded in this implementation.

Fig. 3 I2C Output Register Notation

Address FFFF FFD8H – I2C ACK Register

BIT	TYPE	FUNCTION	DEFAULT	COMMENT
Bit 31	R/W	RESERVED(31)	0	
Bit 30	R/W	RESERVED(30)	0	
Bit 29	R/W	RESERVED(29)	0	
Bit 28	R/W	RESERVED(28)	0	
Bit 27	R/W	RESERVED(27)	0	
Bit 26	R/W	RESERVED(26)	0	
Bit 25	R/W	RESERVED(25)	0	
Bit 24	R/W	RESERVED(24)	0	
Bit 23	R/W	RESERVED(23)	0	
Bit 22	R/W	RESERVED(22)	0	
Bit 21	R/W	RESERVED(21)	0	
Bit 20	R/W	RESERVED(20)	0	
Bit 19	R/W	RESERVED(19)	0	
Bit 18	R/W	RESERVED(18)	0	
Bit 17	R/W	RESERVED(17)	0	
Bit 16	R/W	RESERVED(16)	0	
Bit 15	R/W	RESERVED(15)	0	
Bit 14	R/W	RESERVED(14)	0	
Bit 13	R/W	RESERVED(13)	0	
Bit 12	R/W	RESERVED(12)	0	
Bit 11	R/W	RESERVED(11)	0	
Bit 10	R/W	RESERVED(10)	0	
Bit 9	R/W	RESERVED(9)	0	
Bit 8	R/W	RESERVED(8)	0	
Bit 7	R/W	RESERVED(7)	0	
Bit 6	R/W	RESERVED(6)	0	
Bit 5	R/W	RESERVED(5)	0	
Bit 4	R/W	RESERVED(4)	0	
Bit 3	R/W	RESERVED(3)	0	
Bit 2	R/W	I2C_ACK(2)	0	
Bit 1	R/W	I2C_ACK(1)	0	
Bit 0	R/W	I2C_ACK(0)	0	

The bottom three (3) bits of this register returns the complement of the ACKs supplied by the I2C device during the data transfer. This register is initialized to all zeros. This register is cleared when the I2C output register is written so it can be updated during the new transfer with the new ACK values.

Fig. 4 I2C ACK Register Notation

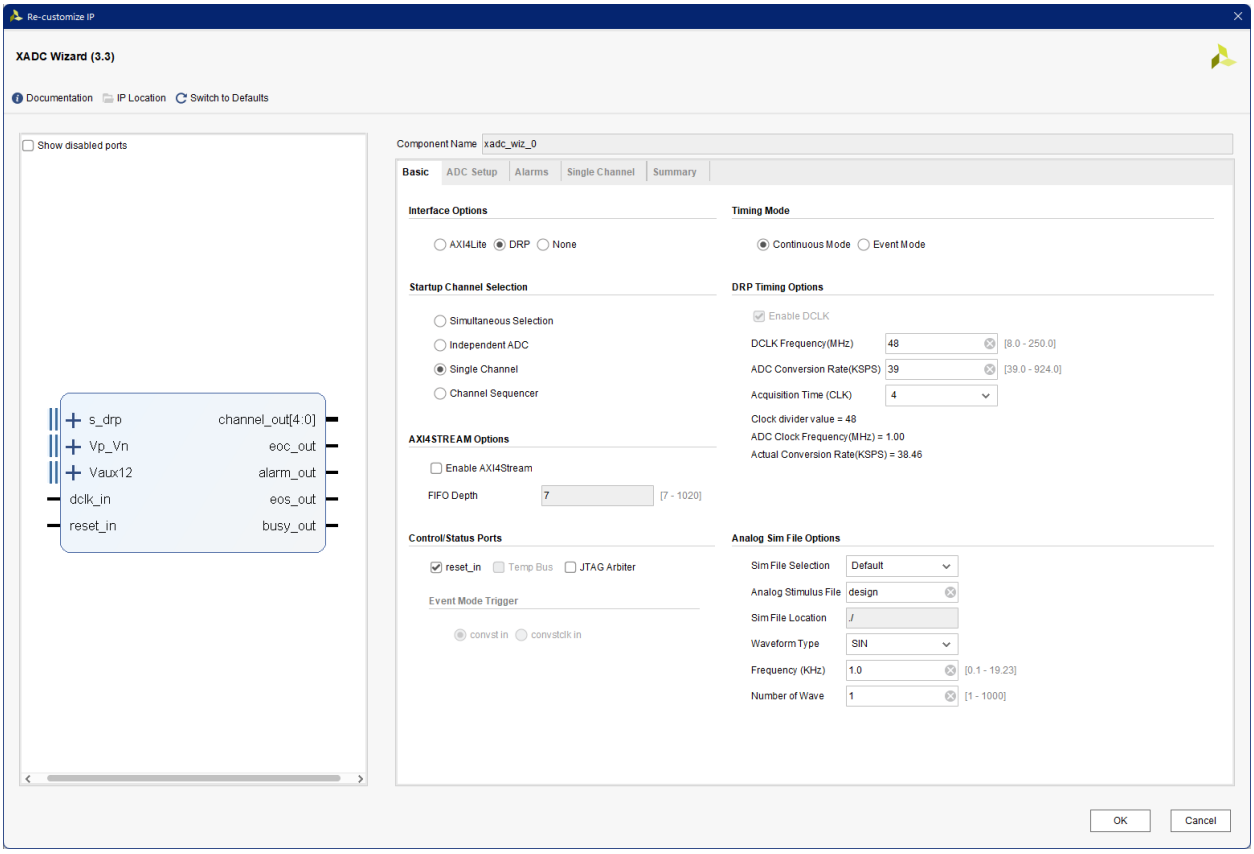

Fig. 5 XADC Wizard Configuration Basic Setup

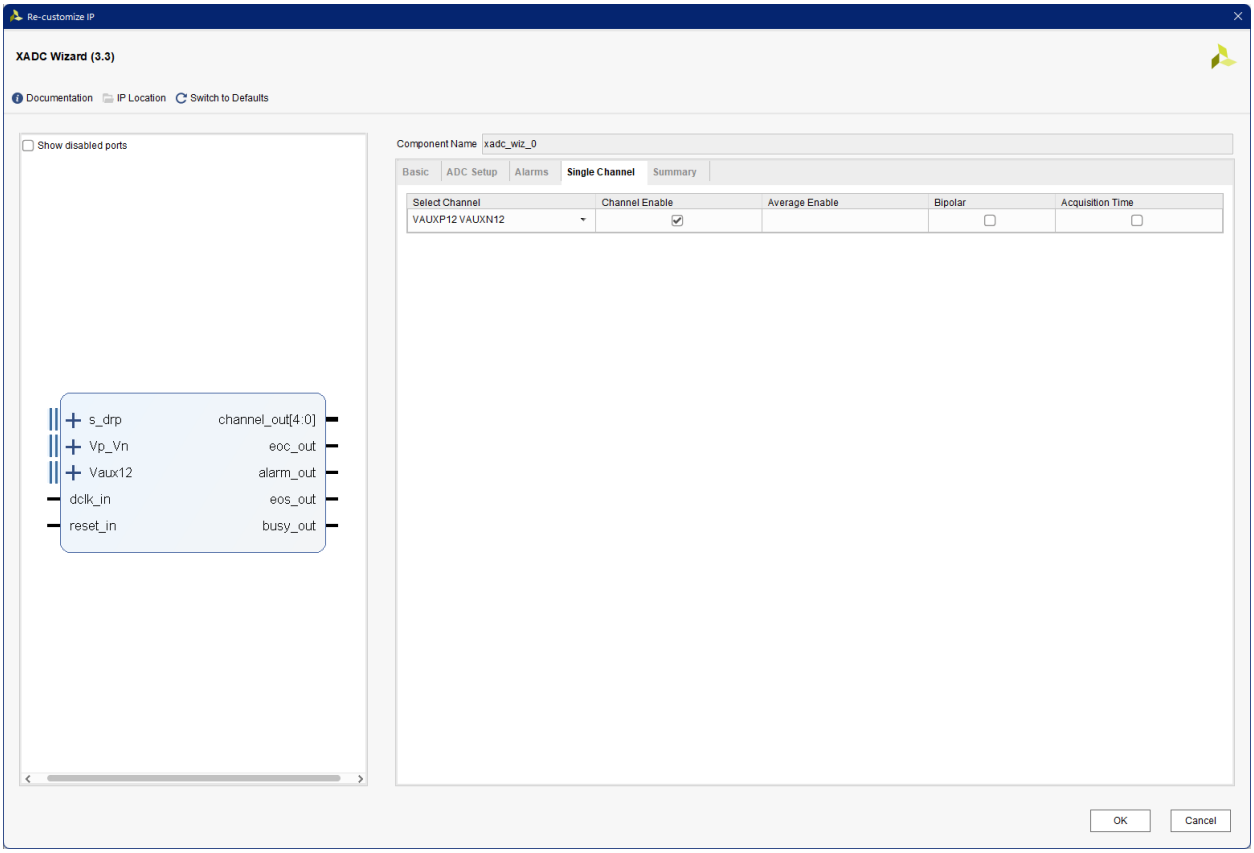

Fig. 6 XADC Wizard Single Channel Setup showing channel running at unipolar mode

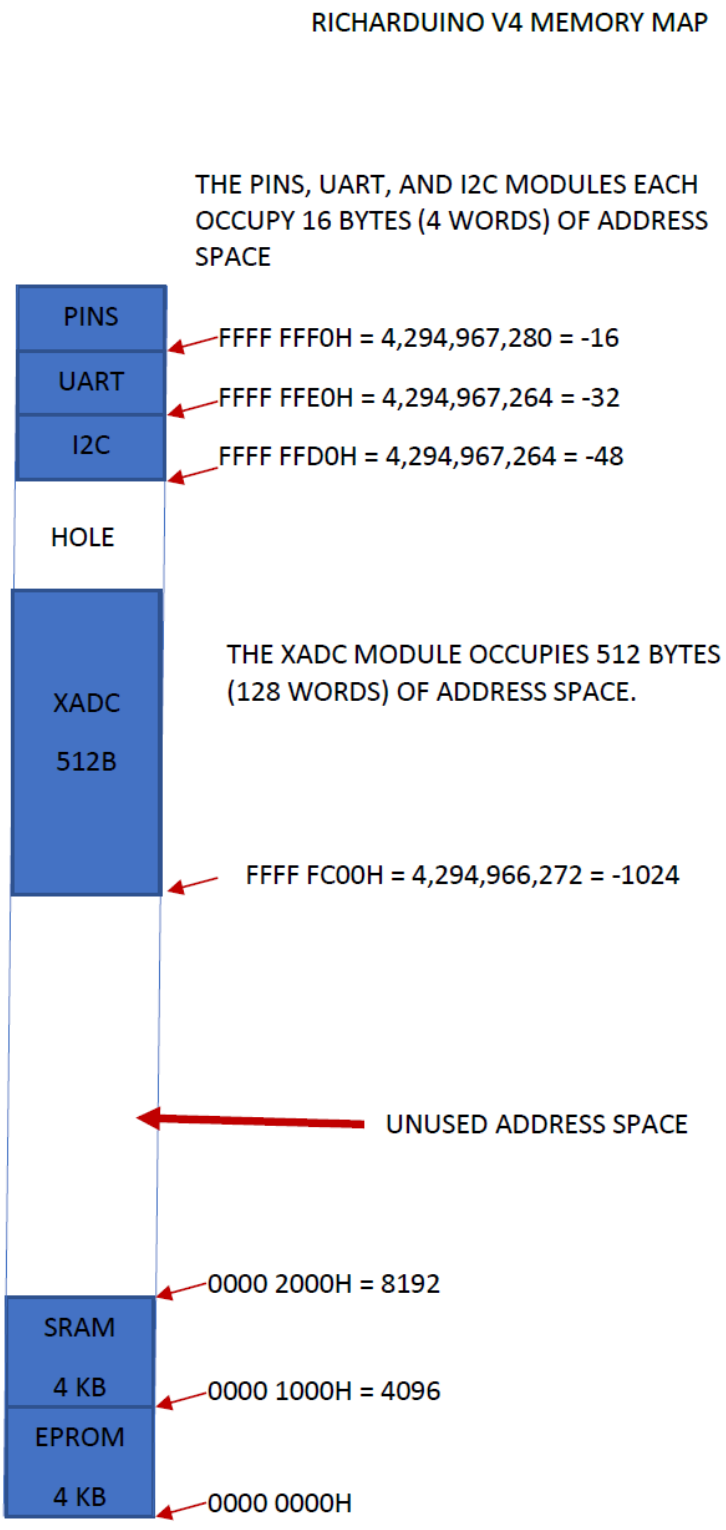

Fig. 7 RichArduino V4.0 Memory Map

```
      .org  0          ; Program starts at address 0
      la    r24,TOP    ; TOP Label
      la    r23,LB2    ; P Program Branch Label
      la    r22,LB10   ; W Program Branch Label
      la    r21,LB18   ; R Program Branch Label
      la    r20,LB26   ; V Program Branch Label

      la    r1,END     ; END Label Address
      la    r2,4096    ; SRAM Starting Address
      la    r3,4096    ; SRAM Address Pointer
      la    r4,80      ; ASCII Code for P
      la    r5,87      ; ASCII Code for W
      la    r6,82      ; ASCII Code for R
      la    r7,86      ; ASCII Code for V

TOP:  ld    r31,-24(r0)
      brzr  r24,r31    ; Checking RX Data Flag

      ld    r31,-20(r0); Load the input

      sub   r30,r4,r31
      brzr  r23,r30    ; Branch to P Program if it is P

      sub   r30,r5,r31
      brzr  r22,r30    ; Branch to W Program if it is W

      sub   r30,r6,r31
      brzr  r21,r30    ; Branch to R Program if it is R

      sub   r30,r7,r31
      brzr  r20,r30    ; Branch to V Program if it is V

      br    r24        ; Branch back to TOP if none matched

LB26: ld    r31,-32(r0)
      brnz  r20,r31    ; Checking TX Busy Flag
      la    r30,4      ; byte 04H to send
      st    r30,-28(r0); Store to TX_DATA, send data out

      br    r24        ; Branch back to TOP


LB18: la    r19,LB19   ; Loop Label 19
      la    r18,LB20   ; Loop Label 20
      la    r17,LB21   ; Loop Label 21
      la    r16,LB22   ; Loop Label 22
      la    r15,LB23   ; Loop Label 23
      la    r14,LB24   ; Loop Label 24
      la    r13,LB25   ; Loop Label 25

      ld    r31,-24(r0)
      brzr  r21,r31    ; Checking RX Data Flag
```

```
      ld    r30,-20(r0); Load the input
      shl   r29,r30,24 ; Shifting according to Big Endian System

LB19: ld    r31,-24(r0)
      brzr  r19,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r28,r30,16 ; Shifting according to Big Endian System

LB20: ld    r31,-24(r0)
      brzr  r18,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r27,r30,8  ; Shifting according to Big Endian System

LB21: ld    r31,-24(r0)
      brzr  r17,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input

      or    r26,r29,r28; Assembling Address in r26
      or    r26,r26,r27
      or    r26,r26,r30

      ld    r30,0(r26) ; Load 32 bit data of the address in r26

      shr   r29,r30,24 ; Shift right 24 bits for Big Endian System
LB22: ld    r31,-32(r0)
      brnz  r16,r31    ; Checking TX Busy Flag
      st    r29,-28(r0); Store to TX_DATA, send data out

      shr   r29,r30,16 ; Shift right 16 bits for Big Endian System
LB23: ld    r31,-32(r0)
      brnz  r15,r31    ; Checking TX Busy Flag
      st    r29,-28(r0); Store to TX_DATA, send data out

      shr   r29,r30,8  ; Shift right 8 bits for Big Endian System
LB24: ld    r31,-32(r0)
      brnz  r14,r31    ; Checking TX Busy Flag
      st    r29,-28(r0); Store to TX_DATA, send data out

LB25: ld    r31,-32(r0)
      brnz  r13,r31    ; Checking TX Busy Flag
      st    r30,-28(r0); Store to TX_DATA, send data out

      br    r24        ; Branch back to TOP


LB10: la    r19,LB11   ; Loop Label 11
      la    r18,LB12   ; Loop Label 12
      la    r17,LB13   ; Loop Label 13
```

```
      la    r16,LB14   ; Loop Label 14
      la    r15,LB15   ; Loop Label 15
      la    r14,LB16   ; Loop Label 16
      la    r13,LB17   ; Loop Label 17

      ld    r31,-24(r0)
      brzr  r22,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r29,r30,24 ; Shifting according to Big Endian System

LB11: ld    r31,-24(r0)
      brzr  r19,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r28,r30,16 ; Shifting according to Big Endian System

LB12: ld    r31,-24(r0)
      brzr  r18,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r27,r30,8  ; Shifting according to Big Endian System

LB13: ld    r31,-24(r0)
      brzr  r17,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input

      or    r26,r29,r28; Assembling Address in r26
      or    r26,r26,r27
      or    r26,r26,r30

LB14: ld    r31,-24(r0)
      brzr  r16,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r29,r30,24 ; Shifting according to Big Endian System

LB15: ld    r31,-24(r0)
      brzr  r15,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r28,r30,16 ; Shifting according to Big Endian System

LB16: ld    r31,-24(r0)
      brzr  r14,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r27,r30,8  ; Shifting according to Big Endian System

LB17: ld    r31,-24(r0)
      brzr  r13,r31    ; Checking RX Data Flag
```

```
      ld    r30,-20(r0); Load the input

      or    r25,r29,r28; Assembling Instruction in r25
      or    r25,r25,r27
      or    r25,r25,r30

      st    r25, 0(r26); Storing Data into Address

      br    r24        ; Branch back to TOP


LB2:  la    r19,LB3    ; Loop Label 3
      la    r18,LB4    ; Loop Label 4
      la    r17,LB5    ; Loop Label 5
      la    r16,LB6    ; Loop Label 6
      la    r15,LB7    ; Loop Label 7
      la    r14,LB8    ; Loop Label 8
      la    r13,LB9    ; Loop Label 9

      ld    r31,-24(r0)
      brzr  r23,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r29,r30,24 ; Shifting according to Big Endian System

LB3:  ld    r31,-24(r0)
      brzr  r19,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r28,r30,16 ; Shifting according to Big Endian System

LB4:  ld    r31,-24(r0)
      brzr  r18,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r27,r30,8  ; Shifting according to Big Endian System

LB5:  ld    r31,-24(r0)
      brzr  r17,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input

      or    r26,r29,r28; Assembling # of Code Bytes (N) in r26
      or    r26,r26,r27
      or    r26,r26,r30

      brzr  r1,r26     ; Branch to END if N = 0

LB6:  ld    r31,-24(r0)
      brzr  r16,r31    ; Checking RX Data Flag
```

```
      ld    r30,-20(r0); Load the input
      shl   r29,r30,24 ; Shifting according to Big Endian System

LB7:  ld    r31,-24(r0)
      brzr  r15,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r28,r30,16 ; Shifting according to Big Endian System

LB8:  ld    r31,-24(r0)
      brzr  r14,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input
      shl   r27,r30,8  ; Shifting according to Big Endian System

LB9:  ld    r31,-24(r0)
      brzr  r13,r31    ; Checking RX Data Flag

      ld    r30,-20(r0); Load the input

      or    r25,r29,r28; Assembling Instruction in r25
      or    r25,r25,r27
      or    r25,r25,r30

      st    r25,0(r3)  ; Place Instruction in SRAM
      addi  r3,r3,4    ; Move Pointer to Next Word Address
      addi  r26,r26,-4 ; Decrement Number of Bytes to Read
      brnz  r16,r26    ; Branch if More Bytes to Read

      br    r2         ; Branch to SRAM to start executing downloaded
code

END:  stop
```

Fig. 8 RichArduino V4.0 Boot Monitor Full Code Block

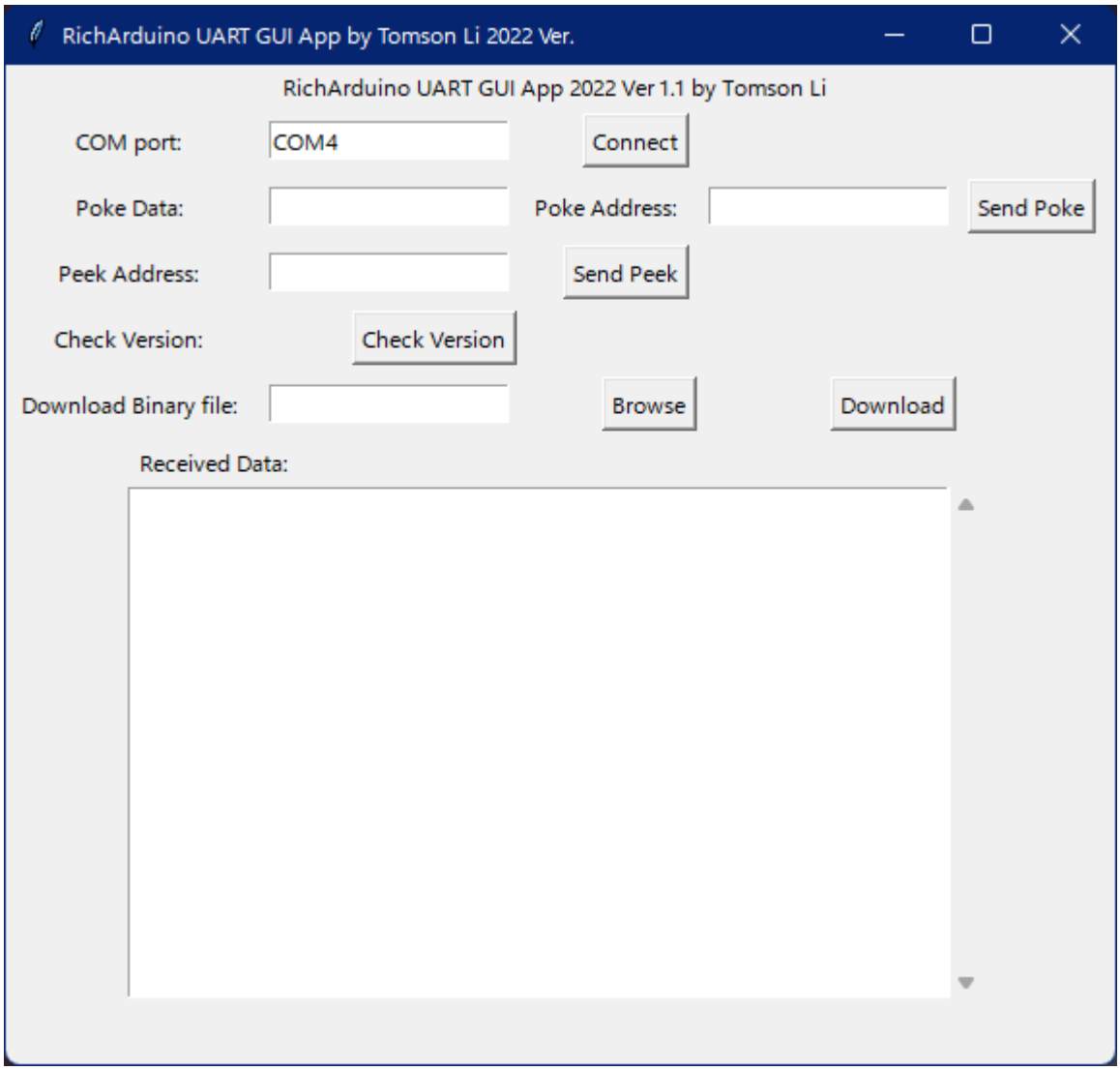

Fig. 9 RichArduino Host Downloader GUI App Interface

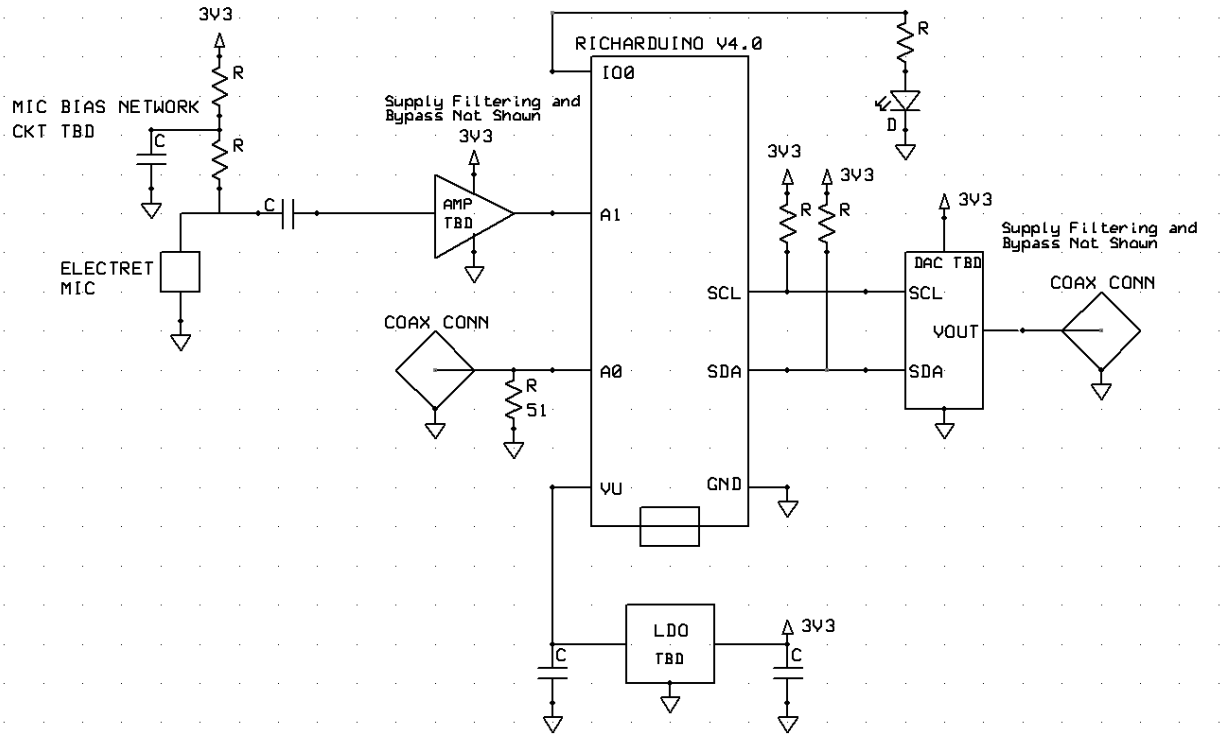

Fig. 10 Demo Shield Block Diagram

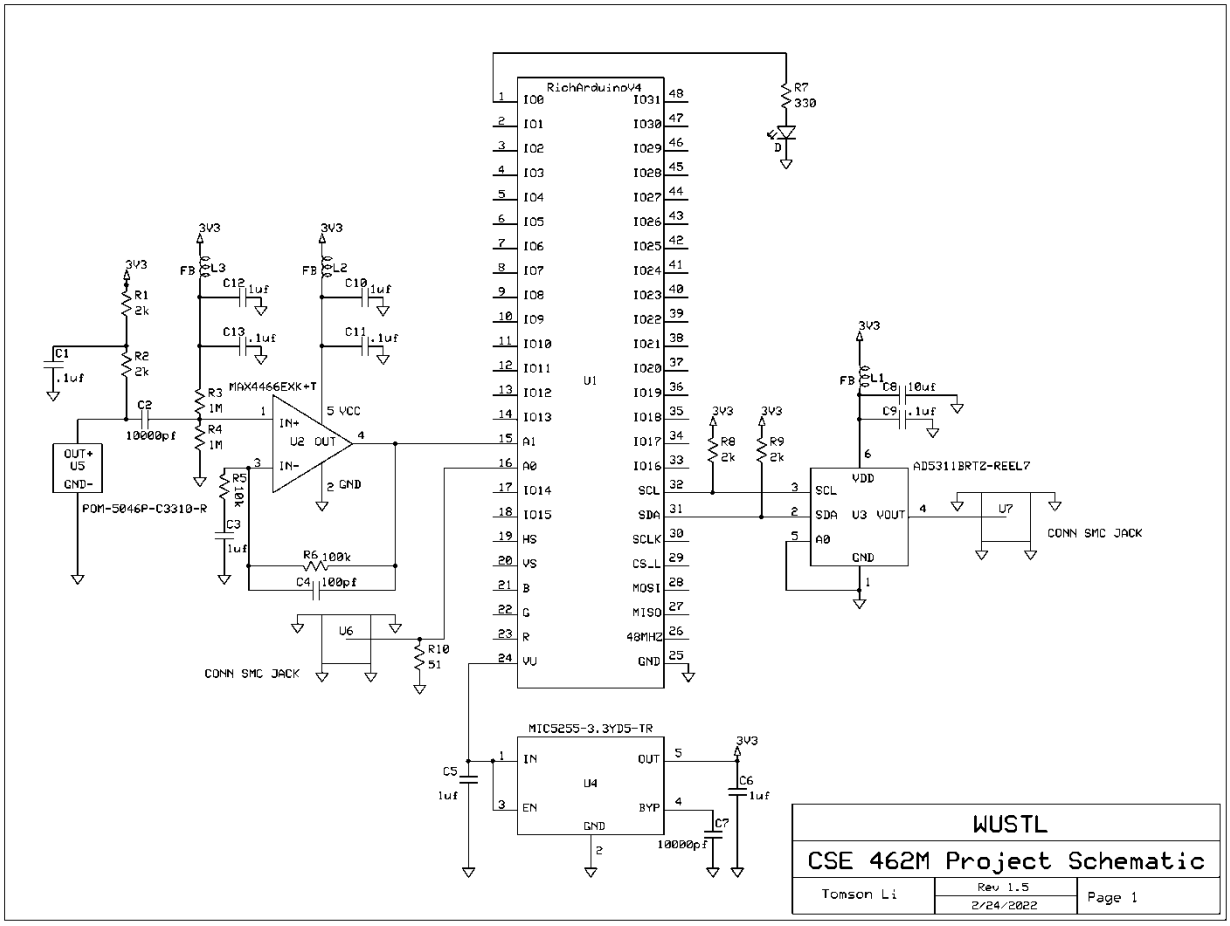

Fig. 11 Demo Shield Full Schematic

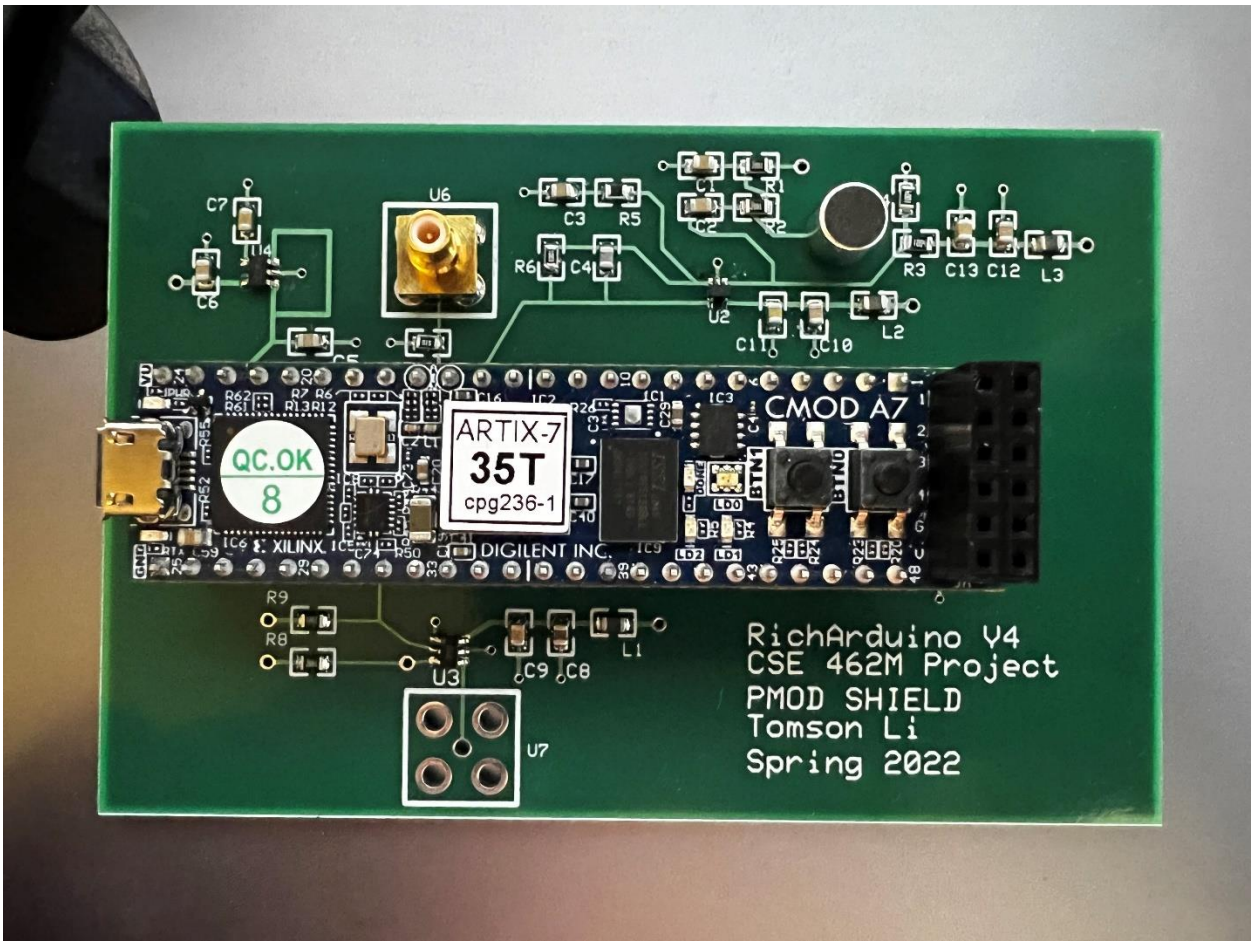

Fig. 12 Complete Demo Shield Product with CMOD A7 board