"Anything that can be connected, will be connected."

As the initial project idea would have been almost impossible to calibrate and demonstrate without the proper material, it was converted to something with less practical application but a greater technical challenge: building an environment monitoring system with mostly pre-internet era technology. While still using contemporary sensors and a high-end embedded device to relay data on the cloud, the data acquisition and processing is performed by a headless homebrew computer based on the MOS 6502, with Commodore 64 spares parts and various salvaged components.

For the system to work, the following components were needed:

- CPU
- RAM
- ROM
- Timers and IO pins
- Analog-to-digital converter
- Serial communication
- Address decoder
- Reset circuit
- Software
- ROM (flash) Programmer

Additionally, to aid in the development & troubleshooting, a 16x4 character LCD was added to display sensors data and a software-controlled sound generator was used to test interrupts and as a general indication that the system is alive and running.

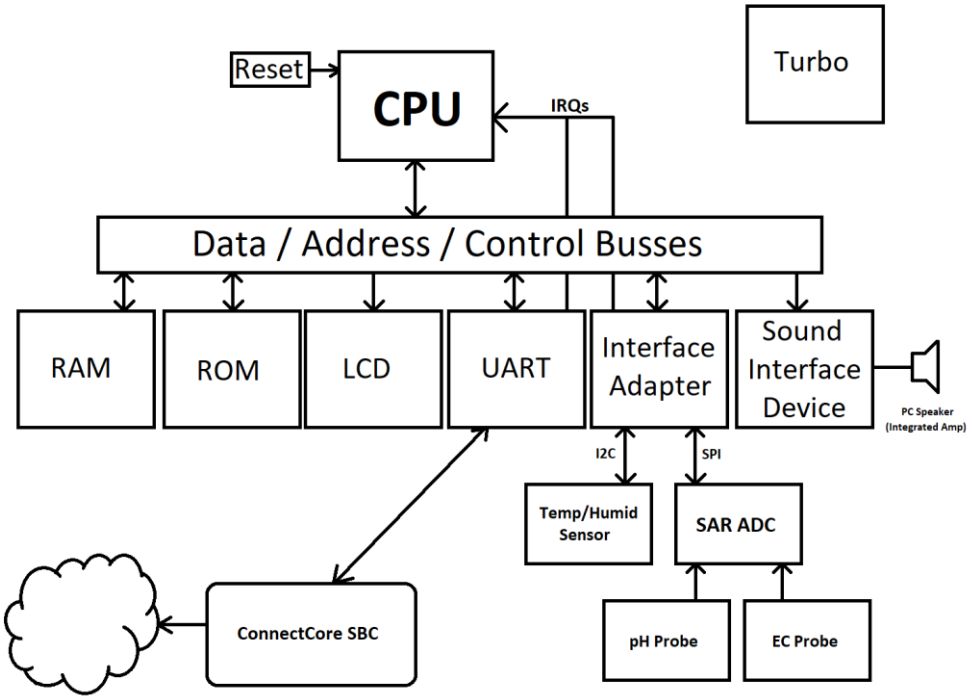

Figure 1.

Architecture Diagram

Development principle:

The development followed an incremental build model with regression testing. Hardware components were added to the system one at a time alongside their respective software. Previously added components were tested at each step to make sure no conflict arose. Using this method simplified troubleshooting, although some problems (which will be discussed in their respective sections) were caught much later with both the RAM and the UART chip, and modifications to the address decoder & control bus were needed along the way.

Development tools:

The software is first assembled sector by sector (4kB blocks) using ACME, a multi-platform cross-assembler offering support for the 65c02 variant. The binary file is fed to a script (made with python at first, then an enhanced version was done in bash) to create an 8-bit unsigned integer array from that data, which is then used by the programmer to store the program into the flash.

The CY8CKIT-059 (PSoC5 LP) is used as the flash programmer. A firmware has been created to replicate the sector-erase and program cycle timing and send the command sequences to the flash chip. The program on the flash is split on 3 independent sectors: music routine on $C000, ISR on $E000 and the main program, reset and interrupt vector on $F000, allowing modification to an individual part without having to reprogram the flash entirely.

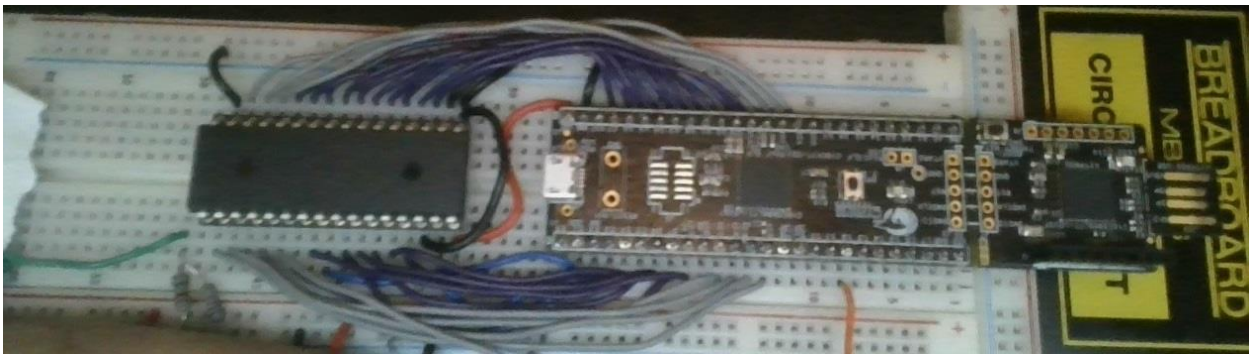

Figure 2. Flash Programmer

Step 1: Clock, CPU, ROM, reset

The selected microprocessor is the WDC 65C02, a CMOS variant of the classic MOS6502. It is an accumulator-based, little-endian 8-bit processor with a 16-bit address bus. Although this specific model can be clocked up to 24MHz a 1 MHz clock is used for this project, to minimize the effect of the gate delay introduced by the bulky address decoding circuit and because this is the standard operating frequency of the Sound Interface Device (a different clock rate would require recalculating the sound envelopes and frequencies parameters).

Since the processor need to have its reset pin held low for a couple cycles after the voltage stabilizes at power on, a reset circuit was needed. The first attempt was made with a one shot multivibrator using a 555 timer, following the design used on the Commodore 64. For reasons which are still unknown it lacked stability and would fire in an apparently random pattern every 10-20 seconds. It was abandoned and replaced by an inverting Schmitt-trigger and a capacitor, which both introduce a delay on start-up and help debouncing the reset switch.

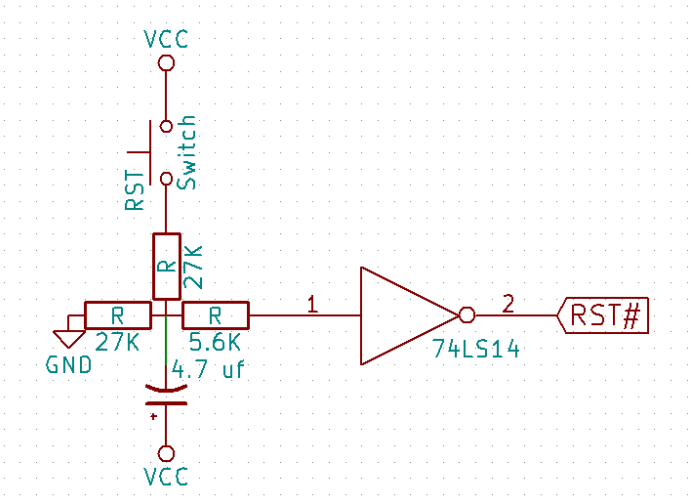

Figure 3. Reset Circuit

A Microchip SST39SF040 (4 Mbit / 512K x 8) Multi-Purpose Flash IC is used for the ROM. Only 44 Kbyte are used at a given time, the address pins A16 to A18 being tied to either ground or VCC. Changing the configuration of the A16-A18 wires allowed to store multiple versions of the program, making it possible to fallback to a previous version if the system stopped functioning. By keeping a previously tested version in another part of the flash memory it was easier to identify if an emerging problem was caused by the new software or was hardware related. For the initial testing a short program was written, alternatively loading the accumulator with the values $AA and $55 in a loop, an easily recognizable pattern to sniff on the data bus with a logic analyzer. Once this was working and stable, indicating the processor, reset circuit and flash programmer were working fine, the next step was to design, build and test the address decoder.

Step 2: Address decoder

The address decoder was designed with a few prerequisites in mind. The 6502 needs the first two pages of memory to be pointing to RAM: the zero page is used for a faster immediate addressing mode as well as for some indirect addressing and the stack is located at address $100 to $1FF (8-bit stack pointer). ROM need to occupy the last few bytes of memory since the processor fetch the NMI, reset and IRQ/BRK vectors from $FFFA-$FFFF. Additionally, to be able to play C64 music with minimal adjustments it is preferable to keep both the SID and the CIA at their original location: $D400 and $DC00, so the 4KB block from $D000-$DFFF was reserved for the I/O devices.

The following division was made, leaving 16KB for RAM and 44 KB for ROM:

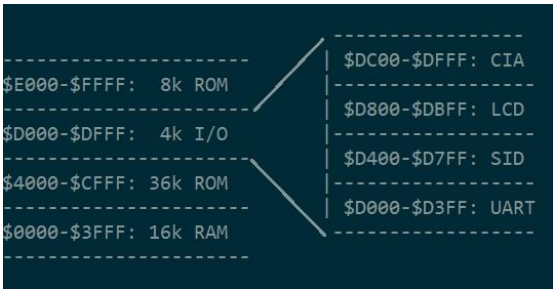

Figure 4. Memory map

Placing the I/O space at $D000-$DFFF, although required for SID music portability, turned out to be a counter-intuitive idea and the resulting CS circuit ended up being quite bulky.

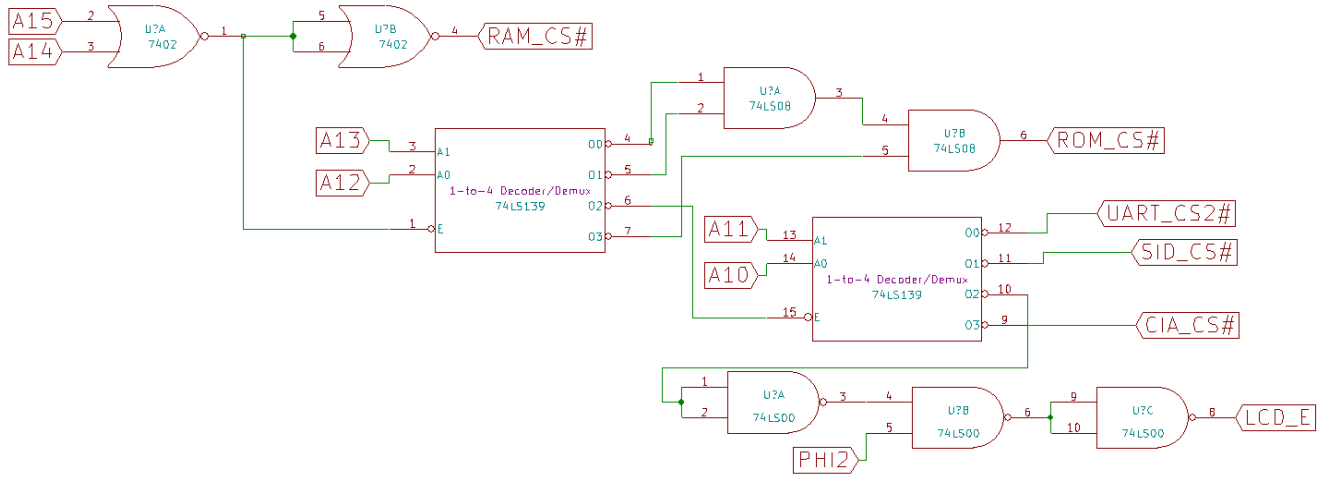

Figure 5. Address Decoder

Step 3: RAM

To be able to unlock most addressing modes, call functions and make general use of the stack, adding RAM was the next logical step. The first 16kB of an AS6C1008, 128K x 8 bit CMOS SRAM was used for this purpose. Although it was quite simple to connect, the address decoder already being in place, and passing all the initial tests (write, read, stack operations, subroutine jumps), it was the first component to cause some problem afterward.

An issue with the control logic which caused it to output data for a few hundred nanoseconds too long was initially ruled out to be just general system noise and did not cause any issue before the interrupts were tested. The only situation where the problem manifested were when a second interrupt triggered while one was already serviced. The rapid consecutive reads and writes when returning from then immediately reentering the ISR corrupted the return address on the stack. After a few days of misled investigation, initially thinking the problem was coming from the stack pointer or the stack itself, the control logic was changed from CE# to WE# controlled cycles, since both were specified in the

datasheet. Instead of NANDing CS# with PHI2 to CE# and having WE# directly in R/W#, CS# got connected directly to CS# and WE# connected to the result of PHI2 and R/W#. Briefly, enabling write only on the second half of the processor cycle fixed the problem.

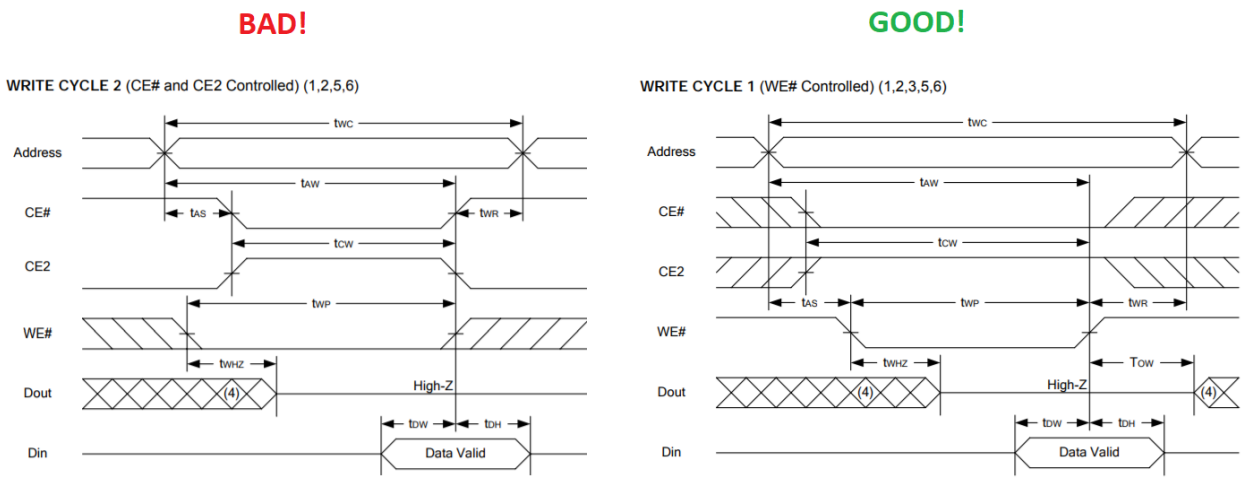

Figure 6. RAM write cycles

Step 4: Complex Interface Adapter

The MOS 6526 / CIA was added for timers' interrupts and GPIO. It offers, amongst other things, two 16 bit programmable timers and two 8 bit parallel I/O ports. Designed to be used with a 6502, it was straightforward to connect it to the system, sharing the same control signals as the CPU. The ports pins direction can be set individually, but to avoid issues caused by misconnections Port A was configured as output and Port B as input. For the timers it was initially planned to use one timer for the music routine and another timer to control the main loop, but after some testing both spares available were exhibiting a well-known bug: if the interrupt control register is read a short time before a timer expire, no interrupt is being generated. The ICR is read both to ACK the interrupts and determine its source, so it will become problematic only if both timer expire in a very short window. Thanks to the deterministic nature of the 6502, it took only a bit of arithmetic and trial and error to build a test program.

Even though the odds of encountering this problem in real life are very thin, especially if one timer is a multiple of the other, using timer B was ruled out for normal operations and a counter was included in timer A ISR. Counting 20 ms slices up to 50, it then raise a flag to signal the main loop to run once a second.

Step 5: Sound interface Device

With all the basic building blocks down (CPU, memory, interrupts) it was time to add some audio capabilities to the system. It was a welcome addition since it served many different purposes other than joyfully annoying people in the project space. As an unmissable power LED it prevented from doing one scary thing which happened way to often in the early development phases: taking the ROM off the board or messing with other connections while the power was on. And with the newly acquired

interrupts, it was a good tool to test the timing and detect system crashes without having to rely on the logic analyzer. A cheap PC speaker with an integrated amplifier was harvested in a computer in the takeaway area and hooked to the sound output to provide maximum annoyance to everyone in range.

Again, just like the Complex Interface Adapter, it was a part designed for the Commodore 64 so there was no problem connecting or operating it. Thousands of music files for the C64 in assembly format (with header) are available so it was only a matter of writing a bash script to retrieve the relevant information from the header (load address, init address, play address, timing), strip it and convert the binary to a C array with xxd. Music data has its own sectors into ROM, so it is not necessary to flash it every time with the main program. A short block of code copies it into RAM at startup and run the init routine, then the play routine is executed inside the interrupt.

The CIA being at the same memory space as on the C64 insure maximum compatibility with most tracks, even those with variable timing (to play music samples on the volume register, for example). If the play routine must change the timers latch values it will have no problem doing so, even if it is set to use the timers from the CIA 2 (C64 has 2) since it is mirrored every $100 bytes across $DC00-$DFFF (CIA 1 at $DC00 and CIA 2 at $DD00). The only unwanted side effect would be to run the main program loop slower or faster, but since there is nothing timing critical in there it will not cause any malfunction.

Only the play routines with variable timing using raster interrupts from the C64 video chip will fail, but it is a small minority which must be ruled out. The only other possible issue is overlapping Zero Page memory usage with the main program. Most of the main program use Zero Page addresses which are reserved for the KERNAL on the original hardware and effort was made to avoid using those who are most likely to be used in music routines.

Problem encountered at this stage: manifestation of the RAM issue mentioned above.

Step 6: LCD

Besides having to invert, strobe with PHI2 then invert again the CE# signal for the Enable pin (see figure 4), the HD44780U LCD did not cause any problem connection-wise. Interfacing it on an 8-bit data bus and controlling it via its 2 registers ended up being way simpler than the usual interface with microcontrollers I/O pins. A few subroutines were made to facilitate positioning, printing strings or characters or blanking lines.

Unrelated to the LCD there is an issue with the clock which happened at this stage. Connecting the PHI2 clock from the 6502 directly caused complete failure of the system. A few hypotheses were made. No voltage drops which could have been caused by excessive current draw on the pin was noticed on the oscilloscope, nor any LPF shaped rounding of the square wave which could have been caused by capacitance was visible on the signal. A timing problem due to gate delay is another possibility which have not been investigated. In retrospective the right thing to do would probably have been to try buffering the clock with a 74244 or something similar but due to lack of space on the breadboard it was not even considered.

From then on, the hackiest solutions were used here and there to supply clock signal to components: NOTing PHI1, connecting directly on the can oscillator instead of on the microprocessor PHI2 out and even giving the ADC its own personal clock.

Step 7: UART

The WDC65C51/ACIA would have been perfectly suited for this purpose, being bus compatible with the 6502, but having to deal with things at hands, a Startech ST16C450 along with its clock circuit was harvested from an old 8032 board. The minimal configurations for its task were made: baud rate, word length, parity, stop bit and RX ready interrupt. Once the initial test was successful, sending the value from a potentiometer on the ADC, the baud rate was raised from 9600 to 115200 and it then started to become unreliable: skipping data while still signaling data was sent.

The first solution was to lower the baud rate back to 9600, which did not solve the issue. Suspecting failure from the chip it was swapped with a brand new one with no better result. With careful observations while sweeping the potentiometer values a pattern was detected: there was a periodic gap in values sent when the 5th bit was set, which made absolutely no sense. A bit of probing on the address bus revealed that the address was changing before the end of the IORead/IOWrite signals.

It is a perfectly normal behavior on the 6502, as the R/W# used to control these signals do keep its state for some nanoseconds after the end of the cycle. Usually it is not a problem, since the PHI2 clock generally control the transaction. Although the UART is not controlled by PHI2, that same change on address pin 2 did not cause problem for most values.

After some puzzled glares at the datasheet, it became clearer:

A2 A1 A0	Register [Default] Note *2	BIT-7	BIT-6	BIT-5	BIT-4	BIT-3	BIT-2	BIT-1	BIT-0
General Register Set									
0 0 0	RHR [XX]	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0 0 0	THR [XX]	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0 0 1	IER [00]	0	0	0	0	modem status interrupt	receive line status interrupt	transmit holding register	receive holding register
0 1 0	FCR [00]	RCVR trigger (MSB)	RCVR trigger (LSB)	0	0	DMA mode select	XMIT FIFO reset	RCVR FIFO reset	FIFO enable
0 1 0	ISR [01]	FIFO's enabled	FIFO's enabled	0	0	INT priority bit-2	INT priority bit-1	INT priority bit-0	INT status
0 1 1	LCR [00]	divisor latch enable	set break	set parity	even parity	parity enable	stop bits	word length bit-1	word length bit-0
1 0 0	MCR [00]	0	0	0	loopback enable	-OP2	-OP1	-RTS	-DTR
1 0 1	LSR [60]	FIFO data error	trans. empty	trans. holding empty	break interrupt	framing error	parity error	overrun error	receive data ready
1 1 0	MSR [X0]	CD	RI	DSR	CTS	delta -CD	delta -RI	delta -DSR	delta -CTS
1 1 1	SPR [FF]	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0

Figure 7. In addition of sending data to the transmit holding register, the last moment change on the address bus enabled loopback and the values were only sent internally.

After devising dirty schemes like inserting some NOPs to insure the next instruction would be at an address which would point to a read only register on the UART chip and slowly witnessing my sanity fade away, the datasheet came to the rescue once again.

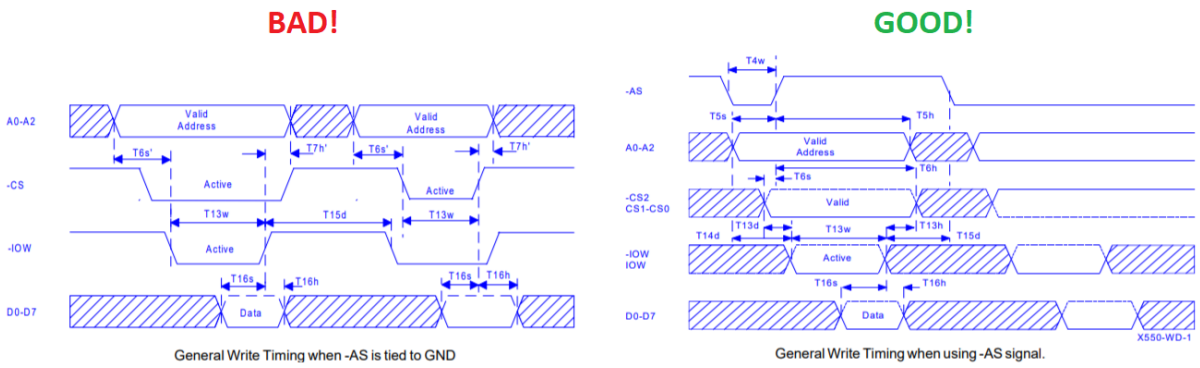

Figure 8. Using -AS to latch the address instead of tying it to ground

The modifications required to add an address decoder and a NAND to generate the -AS signal and completely change the IOR and IOW generation. New space has just been added right next to it to house the sensors so the only inconvenient was more dangling wires from one end of the board to the other. Even if it took less time than fixing the RAM issue and could have been avoided by not blindly picking the simplest control scheme, it was the most interesting and challenging problem in all this ordeal.

A simple way to transmit data to the ConnectCore was then devised. The ConnectCore would request data by sending a byte, raising an interrupt. The Dreadnought would then use that data as an index to fetch memory in ZP, where the processed value from the sensors are stored contiguously. If the value requested is pH, the only 2 bytes value, the second byte is transmitted shortly after. Some error detecting/retransmitting ideas were pitched, but due to the lack of time, potentially decreased reliability (ironically) and for the sake of simplicity, none of them were implemented.

This was the last step to have a working system, before adding the sensors. As shown on the next page.

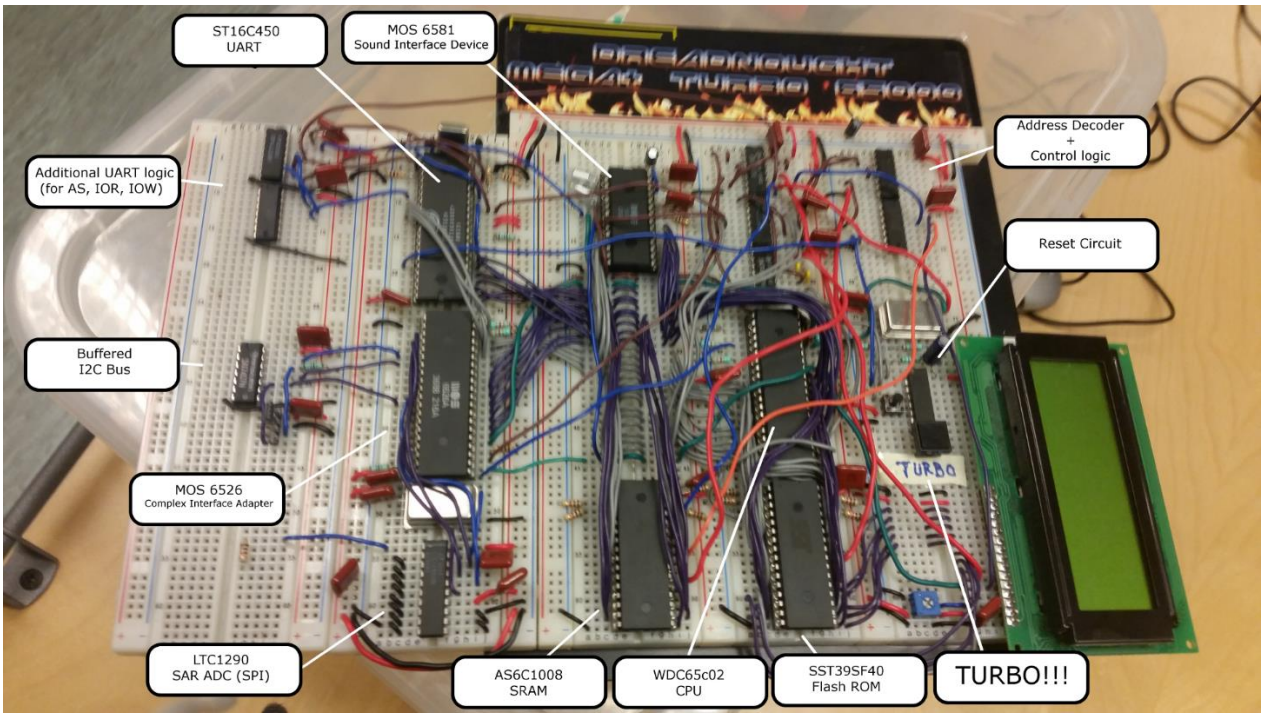

Figure 9. Final system, without the sensors

Step 8: Sensors

Two sensors made it to the project demo: an I2C Honeywell HumidIcon humidity and temperature sensor and an analog pH probe.

The HumidIcon is connected to the system through the I/O port of the Complex Interface Adapter. Writing and reading to/from the sensor is done through bit-banging subroutines. The implementation is only partial, not considering the acknowledges from the sensor or allow clock stretching and the software interface was not made with flexibility in mind but works fine for this specific purpose. Although the bitrate achieved (~43 kbps) is far below the 100 kHz minimum SCL clock frequency specified in the datasheet, no communication issues were found. The inverse transfer functions have been slightly changed. For the humidity, instead of data_in / (2^14-2) * 100, the minus two in the divisor has been dropped, allowing to do 14 bit shifts instead of running a lengthy division routine, and yield a error of only up to 0.02% when getting close to 100% humidity, which is insignificant. The same process has been done for the temperature data, again with an error of up to 0.02 Celsius at high temperature.

The analog pH probe is interfaced through SPI with a LTC1290 SAR ADC. The SPI is again implemented through bit-banging on the Complex Interface Adapter I/O ports. Requests for conversion are done at program startup and after each reading, and the data ready status is polled before taking the data in.

The inverse transfer for the pH electrode itself is relatively simple, yielding 0 volts at a pH of 7, and with steps of +59.16 mV for every point below, -59.16 mV for every point above. The probe came with a module taking care of the amplification and voltage offset though, with little documentation beside an Arduino code. According to the example code, the formula to get the pH value from the module output

is 3.5*voltage + calibration offset. A rough approximation combining it with the equation to get the voltage value from the 8-bit ADC (val*/5/255) was done.

Since the correct calculation would be:

pH value = 3.5 * adc value * 5 / 255 = adc value

to avoid dealing with floating point numbers and still give a 2 decimals precision, it was simplified to:

pH value = adc value * 7

giving a value equal to 100x the pH value. A "." Is then added after the first digit when printing on the LCD, and the device at the receiving end of the UART need to take care of the division by 100. This will be problematic only in the very unlikely scenario where the pH is below one. The calibration offset was left out of the equation, having no calibration solution at hands.

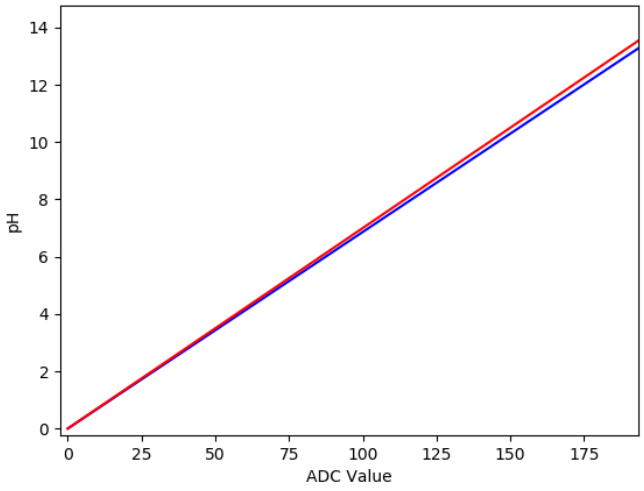

Figure 10. Error introduced by the pH value approximation.

The red line indicates the approximation value. The error is around 0.2 point around a pH of 7.

A very similar attempt to simplify the inverse transfer function of a conductivity meter failed. The concentration of dissolved solid in the water sample we had was making the value underflow, and due to a lack of time to fix it, it was simply removed from the project.

Technical References:

-The Western Design Center, “W65C02S Datasheet”, Nov. 2016

-D. Eyes and R. Lichty, “Programming the 65816 Including the 6502, 65C02 and 65802”, 1992 [Revised by The Western Design Center Inc., 2007]

-Microchip, “4 Mbit (x8) Multi-Purpose Flash”, SST39SF040 Datasheet, 2016

-Alliance Memory, “128K x 8 bit Low Power CMOS SRAM”, ASC1008 Datasheet, Feb. 2007

-Commodore Semiconductor Group, “6526 Complex Interface Adapter (CIA)”, MOS6526 datasheet, Nov 1981

-Commodore Semiconductor Group, “6581 Sound Interface Device (SID)”, MOS6581 datasheet, Oct. 1982

-Hitachi, “Dot Matrix Liquid Crystal Display Controller/Driver”, HD44780U (LCD-II) datasheet, Sept. 1999

-EXAR Corporation, “Universal Asynchronous Receiver/Transmitter (UART)”, ST16C450 datasheet, Sept. 2003

-Honeywell, “Honeywell HumidIcon Digital Humidity/Temperature Sensors” HIH9000 Series datasheet, May 2015