ALORIUM TECHNOLOGY

XLR8™

FPGA
APPLICATION ACCELERATOR
&
DEVELOPMENT BOARD

User's Manual and Release Notes

February 8, 2017

Version	DATE	AUTHOR	CHANGES
1.0	April 3, 2016	mwebber	Initial Version
1.1	Dec 14, 2016	jpeterson	Added Best Practices section for SPI
1.2	Feb 1, 2017	hjuedes	Updated 3.3 Reconfigurability Instructions
1.3	Feb 8, 2017	Hjuedes	Updated Register Summary description

Table of Contents

1 Overview

XLR8 is an FPGA-based application accelerator and development board that has been specifically designed to look, feel, and act like a standard Arduino. It is programmed with the popular and easy to use Arduino IDE. The heart of XLR8 is an FPGA chip that is configured with an ATMega328 microcontroller clone as well as additional accelerator functions.

XLR8 provides an option for Arduino developers to achieve significantly improved performance in the same physical footprint and using the same tool chain as standard Arduino Uno and other similar Arduino compatible boards, even when incorporating custom hardware functions via accelerator blocks.

2 Usage

XLR8 can be programmed with Arduino sketches from the Arduino IDE by selecting "Arduino Uno" as the target board, in the same way as the Uno. The pin headers are compliant with the Arduino Uno R3 layout. It has the same analog pins, the same digital pins, and the same pins with PWM functionality as the Uno. So, using it as a clone of the Uno is a simple as using an Uno. The real fun starts after installing our Arduino Board package as described in the instructions at https://github.com/AloriumTechnology/Arduino_Boards, enabling advanced features such as doubling the microcontroller's clock speed or utilizing Xcelerator Blocks (XBs). But you don't need to do that until you are ready.

3 Differences between XLR8 and Arduino Uno/Sparkfun RedBoard

3.1 USB

Similar to the Sparkfun RedBoard, XLR8 uses a USB mini-B connector and an FTDI chip to do the USB-to-Serial conversion, which is slightly different than the Uno. If you haven't already, you'll likely need to install FTDI drivers. Note that this is not needed and should not be done if you are using Mac OS El Capitan or later. The OS will have the correct drivers.

Sparkfun has a great tutorial showing just how to do that https://learn.sparkfun.com/tutorials/how-to-install-ftdi-drivers). We have noticed for the Mac that Sparkfun's tutorial doesn't mention restarting your computer after installing the driver, but we've generally needed to do that and everything has worked great afterwards. We have also noticed that the FTDI driver in older Linux kernels (specifically, version 2.6 or older) does not appear to support the newer FTDI chips that are used on XLR8, Sparkfun's RedBoard, and many other products. This can be fixed by upgrading your Linux kernel to version 3.10 or newer.

3.2 Xcelerator Blocks (XBs)

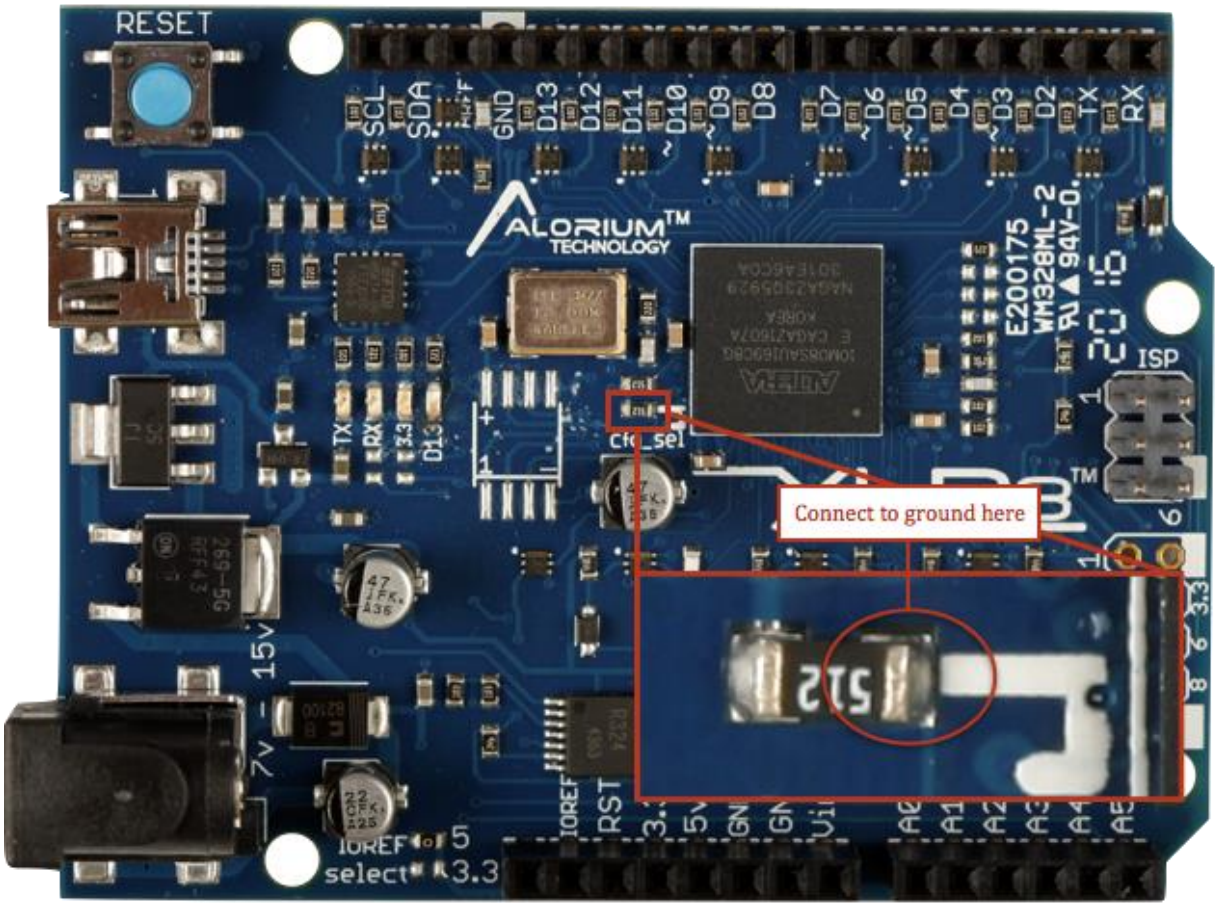
Xcelerator Blocks give XLR8 a performance advantage through a combination of custom hardware in the FPGA fabric along with a software library that is able to communicate with that hardware and make it easy for users to take advantage of the performance. The software libraries are available on our github site (https://github.com/AloriumTechnology), but it is

easy to install them without even going to github. In the Arduino IDE, go to the menu **Sketch -> Include Library -> Manage Libraries**, which will open the Library Manager in a new window. Enter **XLR8** in the search bar and you will find the entries for the various XLR8 libraries available. Click on the desired library and an **Install** button will appear for it.

3.3 Reconfigurability

One of the most awesome things about XLR8 is its re-configurability. XLR8 is able to hold two different FPGA images. Image 1 can be reconfigured from the Arduino IDE to take advantage of increased functionality as new XBs are introduced. The reconfiguration files are obtained by installing our Arduino Board package as described in the instructions at https://github.com/AloriumTechnology/Arduino_Boards. Image 0 is never changed and is typically unused unless the primary image 1 becomes corrupted. If necessary, a "factory reset" of XLR8 can be performed by grounding the FPGA side of R51 (cfg_sel) while applying power to the board. It only takes a momentary grounding to cause this to happen. The factory image is then loaded. However, any attempt to run a sketch or reset after loading the factory image will lead to the bad image being reloaded. The user will immediately want to reburn a known-good image into image 1 before attempting anything further.

3.3.1 Connecting to Ground



3.4 I/O

The FPGA at the heart of XLR8 is a 3.3v device. However, unlike other 3.3v boards, we've made a concerted effort to make XLR8 as compatible as possible with the 5v Arduino Uno. We've done this by adding level shifting and protection circuitry to the I/O so that existing shields can be used without fear of damaging the board. The digital I/O's (RX, TX, and D2-D13) can both drive and receive 5V signals providing great compatibility with all shields. The analog I/O's (A0-A5) can tolerate both analog and digital inputs up to 5.0V, and as digital outputs can drive up to 3.3V and that should work fine for the majority of shields and other circuits. XLR8 provides 5.0V on the IOREF pin for newer shields that use that feature.

While ATmega328p inputs can enable or disable the internal weak pullup resistors, in XLR8, the pullups differ by pin as follows:

- RX, TX, D2-D13: Strong pullups on XLR8 board (1K ohms) are always enabled. The initial value of the PINB and PIND registers is therefore 0x3F and 0xFF respectively.
- A0-A5: No pullups
- SDA, SCL: Strong pullups on XLR8 board (1K ohms) are included. When using Arduino libraries, Wire.begin() enables the pullups for the I2C case (assuming the PUD bit of the MCUCR (0x35) register is not set) without any additional work required from the user. If desired, disabling the pullups in Arduino is as simple as doing both `digitalWrite(SDA, LOW);` and `digitalWrite(SCL, LOW);`. Outside of the Arduino environment the pullups are enabled when PUD is low, TWEN is high, and both PORTC4 and PORTC5 are high.

3.5 ADC

The ADC used in XLR8 is higher performing in both speed and resolution than the ADC found in the ATMega328p. By default, doing an analogRead will give the same speed and 10 bit resolution as an Arduino Uno. By using the XLR8ADC library (https://github.com/AloriumTechnology/XLR8ADC) you can get 12 bit resolution. Further enhancements will be available in the future.

One ADC difference between XLR8 and Arduino Uno/Sparkfun RedBoard occurs when running from USB power. The USB may supply a voltage that is somewhat lower than 5V and because the Uno and RedBoard use this for the default analog reference it results in higher ADC readings compared both to what would be expected and to what is measured when powered from the barrel connector. The XLR8 board does not suffer from this issue; it will give the same ADC reading whether powered from the barrel or from USB. Some shields, however, can be deceiving. If an analog voltage that the shield is creating drops along with the 5V USB power, it will appear on the Uno/Redboard that the ADC readings stay somewhat constant while on XLR8 the ADC readings will be reduced. If it is desired to have XLR8 vary the reference voltage with the USB power similar to an Uno/Redboard, this can be accomplished quite easily by wiring from the 5V header pin to the AREF header pin and using Arduino's `analogReference(EXTERNAL);`.

While the ATMega328p is able to switch its ADC reference voltage between internal 5V, internal 1.1V, and external references, XLR8 uses circuitry on the board to make this selection. From the user's point of view, there shouldn't be a noticeable difference.
To ensure that XLR8 can tolerate ADC inputs up to 5V, the analog inputs run through an op-amp and voltage divider circuit before entering the FPGA. Again, from the user's point of view, there shouldn't be a noticeable.

While the ADC channel selection is the same for measuring the primary six analog inputs (ADMUX=0000 through 0101), there are differences in the other channels. Channel 6 (ADMUX=0110) reads from the voltage divider created by R30 and R59, and should be around the max ADC value (1023 when in 10 bit mode) if the 5V supply is at or above 5.0V, but may be slightly less if the 5V supply is lower. Channel 7, and the temperature sensor are not implemented (ADMUX=0111 and 1000). Using the ADC to read the bandgap (ADMUX=1110) does not actually do a measurement but returns a calculated value equivalent to 1.1/Aref. Using the ADC to read ground (ADMUX=1111) does not actually do a measurement and instead returns a fixed value of 0.

3.6 Bootloader and ICSP header

XLR8 uses the Optiboot bootloader that is used by the Arduino Uno, with a slight modification that allows it to run correctly at different CPU speeds. As an extra indication of XLR8's current CPU speed, you may notice when running at 32MHz that the beginning of an upload sequence blinks the pin 13 LED three times rather quickly just like an Arduino Uno, while at 16MHz you get two slightly slower blinks. This bootloader is hardcoded into the design and cannot be changed by the user. If you have a need for something different in the bootloader, we'd be interested to hear about it. Consistent with using Optiboot, the BOOTRST "fuse" is hardcoded to zero (programmed) and the IVSEL and IVCE bits of the MCUCR (0x35) register are hardcoded to zero.
XLR8 is programmed only via the USB-UART path. While the XLR8 board includes the ICSP header it is useful only as a convenient connector for the SPI interface; it can't be used to load programs/sketches into the design.

3.7 I2C / Two Wire Interface (TWI)

Although the Arduino Uno's Rev3 layout has separate SDA and SCL pins for I2C/TWI, they are still connected on the Uno board to the original A4/A5 pins. This means that when using the I2C interface, two of your analog inputs are no longer available. On XLR8, I2C/TWI is by default only available on the SDA/SCL pins and the A4/A5 pins can still be used as analog or digital inputs (but not outputs). If you have existing projects that need to use A4/A5 as the I2C pins, you can reconnect them to SDA/SCL by populating 0ohm resistors at R60 and R61.
Another difference between the I2C/TWI on XLR8 and Arduino that you will likely never notice is related to the TWBR register. While Arduino allows running with non-standard SCL clock frequencies, XLR8 is optimized for 100KHz and 400KHz operation and setting TWBR and the prescaler bits of TWSR for other speeds will not be operational.

3.8 UART

XLR8 has a Universal Asynchronous serial Receiver and Transmitter (UART) block similar to the ATmega328p USART. However, XLR8's UART does not implement the Synchronous UART

(USART) in SPI Modes that are available on the ATmega328p. Therefore, the UMSEL01, UMSEL00, and UCPOL0 bits of the UCSR0C (0x) register, while implemented, have no effect and the UART block always runs in "Asynchronous UART" mode. We have not yet seen a case where anything else is needed in an Arduino environment; if you have one, please let us know.

3.9 SPI

The SPI interface on XLR8 should operate the same as the SPI interface on an Uno or Redboard. The only item to note is that we've seen some SPI examples where the XLR8/Uno/Redboard is the SPI slave and instead of being driven from the SPI master, the SS pin is left floating. Although a sloppy design practice, an Uno or Redboard will often still work in this arrangement. Our XLR8 board, due to its I/O pullups, needs to have SS driven and not floating.

3.9.1 SPI Best Practices

In rare cases, we have observed an issue with the SPI clock being properly sampled by a shield. We suggest adding a 100-200 ohm pulldown resistor to the SPI clock pin D13 for more robust operation.

3.10 Analog Compare

XLR8 does not have the Analog Compare function that is found in the ATmega328p. The ACME bit and analog compare triggering (ADTS=001) of the ADCSRB (0x7B) register, the ACSR (0x30) register, and the DIDR1 (0x7F) register are not implemented. If an analog compare function is desired, pins 6 and 7 on the (unpopulated) JTAG connector are wired to MAX10 pins DIFFIO_TX_RX_B1P/N which is a differential I/O. Using the OpenXLR8 platform, a user could implement an analog compare function that is very similar to the ATMega328's, although the pin voltage would need to be limited to 3.3V.

3.11 EEPROM

XLR8 does not have the EEPROM memory that is found in the ATmega328p. However, there is an SOIC-8 location that could be populated with an EEPROM and using the OpenXLR8 platform, a user could implement an EEPROM function that is very similar to the one found in the ATMega328. The SOIC-8 pinout is:

FPGA pin	Typical Use	SOIC-8 pin		SOIC-8 pin	Typical Use	FPGA pin
C12	Addr0	1	1 +	8	Vcc	3.3V
B13	Addr1	2		7	WP	D11
C11	Addr2	3		6	SCL	D12
Gnd	Vss	4	-	5	SDA	E13

3.12 Power

The Arduino Uno and the Sparkfun RedBoard are only able to supply 50mA from the 3.3V pin header. XLR8 has a 1.5A regulator creating its 3.3V power (an LT1963EST-3.3) and the Max10 FPGA is estimated to draw less than 100mA, leaving a significant amount of 3.3V power available to the user for shields or other circuits. (Keep in mind that USB power is limited to about 500mA.)

3.13 Pin13 LED

Digital pin 13 is used for both the on-board LED as well as the SPI clock, SCK. While the Arduino Uno R3 adds circuitry to prevent the LED and its pulldown resistor from affecting SCK, the Sparkfun Redboard does not take those precautions. On XLR8 we've decided to follow Arduino's lead and keep SCK separated from the LED.

4 ATmega328p features not implemented

The ATmega328p microcontroller includes several features that are seldom or never used in an Arduino environment. In XLR8, these features are hard coded to match the Arduino usage. Therefore, when comparing to the ATmega328p specification the following differences may be noted, however they do not impact functionality in an Arduino environment. One exception is that Arduino Uno allows programming via the SPI interface while, as mentioned in section 3.6, XLR8 has not implemented this feature (SPIEN fuse).

4.1 Fuses

The Arduino Uno runs with the following fuse settings.

- Low Fuse 0xFF (all unprogrammed)
  - CKDIV8: Default to clock division factor of 1.
  - CKOUT: Don't output system clock on port B pin 0
  - SUT1/0: Start up time
  - CKSEL3/2/1/0
    - Always use external clock (supplied by 16MHz) crystal.

    - XLR8 does not have the internal 128kHz RC Oscillator or the Calibrated Internal RC Oscillator that exists in ATmega328p. The OSCCAL (0x66) register is not implemented.
    - Because ATmega328 shares the TOSC1/2 pins with the XTAL1/2 pins, and XTAL1/2 are used to bring in the external 16MHz clock, the asynchronous mode of timer/counter 2 is not available and the ASSR register (0xB6) is not implemented.
- High Fuse 0xDE in Arduino (SPIEN and BOOTRST programmed), 0xFE in XLR8 (BOOTRST programmed)
    - RSTDISBL: External reset is always enabled. This is how the microcontroller returns to the bootloader to upload a new sketch from the Arduino IDE to program memory
    - DWEN: The debug wire feature is not enabled in Arduino and is not implemented in XLR8.
    - SPIEN: While Arduino programs this bit, XLR8 instead functions as if it were not programmed (see section 3.6)
    - WDTON: Watchdog always on not enabled
    - EESAVE: EEPROM not preserved through chip erase. Not applicable on XLR8.
    - BOOTSZ1/0: Bootloader is 256 words starting at 0x3F00
    - BOOTRST: Reset starts at the bootloader address
- Extended Fuse 0x05 in Arduino (BODLEVEL1 programmed), 0x07 in XLR8 (all unprogrammed)
    - BODLEVEL2/1/0: In XLR8 the brown out detection circuitry is disabled and not implemented. The BODS and BOSE bits of the MCUCR (0x35) register are not implemented, as well as the BORF bit of the MCUSR (0x34) register.

4.2 Power Reduction

Because we don't see it being used very often (or at all) in Arduino projects, XLR8 has not implemented the ATmega328p's Power Reduction Register PRR (0x64), Sleep Mode Control Register SMCR (0x33), or Clock Prescale Register CLKPR (0x61). They could be added if customer feedback indicates that would be a good thing to do.

5 Xcelerator Blocks (XBs)

Xcelerator Blocks are custom hardware blocks implemented within the XLR8 FPGA chip and are tightly integrated with the ATmega328 clone that is also implemented inside the FPGA chip. These custom hardware blocks can implement almost any functionality you can dream up, and can then be loaded into the XLR8 as with the Arduino toolset. Since an FPGA can be reprogrammed many times, a single XLR8 can be reconfigured to incorporate different XB depending on the project requirements.

XLR8 ships with three sample XBs: Floating Point, NeoPixel, and Servo Control. As mentioned in section 3.2, the software libraries are delivered as .zip files from our github site (https://github.com/AloriumTechnology). They are installed like other Arduino .zip libraries as described here (https://www.arduino.cc/en/Guide/Libraries - toc4).

Note: There is one thing to be aware of with all Arduino libraries. If you download an updated version and simply rename the directory of the old version (perhaps you think you might want to go back to it or maybe you are just feeling nostalgic), Arduino very likely will still end up using the old version as it simply looks for the first place it can find the #include header file name, regardless of what directory it is in. If you want to save an old version of a library, move it to someplace outside of your `Arduino/libraries` folder.

5.1 Floating Point

As an 8 bit microcontroller, the ATmega328p struggles with floating point math. The Floating Point XB provides functions that will give you floating point results in about ¼ the time that it takes software floating point to get the same answer. Available functions include add, subtract, multiply, and divide.

5.2 Servo Control

It is common for the standard Servo.h library to cause jitter in the servo control due to timing uncertainties caused by interrupt processing. The Servo Control XB completely eliminates this jitter by putting a dedicated hardware timer behind all 20 digital and analog pins. The XLR8Servo.h library is a drop-in replacement for the standard Servo.h library, so taking advantage of this XB is as simple as changing one line in your sketch from

```
#include <Servo.h>
```

to

```
#include <XLR8Servo.h>
```

5.3 NeoPixel Control

The standard Arduino struggles to meet the timing requirements of NeoPixels. With the XLR8 NeoPixel hardware and library, interrupts remain enabled, data memory is saved, and pixel color information does not need to be rewritten when brightness is lowered and then brought back up. The XLR8NeoPixel library can be a drop-in replacement for the standard Adafruit_NeoPixel library. It is common for other libraries, such as Adafruit_NeoMatrix, to build on the Adafruit_NeoPixel library and with just three lines added to the top of your sketch, it is possible to get the XLR8 advantages used by those libraries as well. For example,

```
// 3 new lines added
#include <XLR8NeoPixel.h>
#define Adafruit_NeoPixel XLR8NeoPixel
#define ADAFRUIT_NEOPIXEL_H
// Existing lines kept
#include <Adafruit_GFX.h>
#include <Adafruit_NeoMatrix.h>
#include <Adafruit_NeoPixel.h>
```

As explanation, working from the bottom up:

- Including Adafruit_NeoPixel.h remains because other library files that we are using but are not modifying (Adafruit_NeoMatrix.h in this case) have `#include <Adafruit_NeoPixel.h>`. The way Arduino handles compiling and linking generally requires that lower level #includes also be included in the sketch.
- Including Adafruit_NeoMatrix remains because that is the library that this particular sketch is using

- Including Adafruit_GFX remains for the same reason that it was there initially. It is included by Adafruit_NeoMatrix, and the way Arduino handles compiling and linking generally requires that lower level #includes also be included in the sketch.
- Even though we are still including the Adafruit_NeoPixel.h, adding `#define ADAFRUIT_NEOPIXEL_H` causes that header file to appear empty instead of having the actual Adafruit library code.
- By adding `#define Adafruit_NeoPixel XLR8NeoPixel`, any place in your sketch, or in the libraries that your sketch is using, where an Adafruit_NeoPixel string is instantiated, it will instantiate an XLR8NeoPixel string instead.
- And of course, the XLR8NeoPixel library is included.

Easy. Powerful. Fun.
And designed to become even more powerful in the future. Watch for updates.

6 Register Summary

The registers used in XLR8 are listed below. Those with a grey background have identical function to the equivalent ATmega328p register. Those with just a touch of grey are reserved in both XLR8 and ATmega328p but are noted because they are used in the ATmega328pb. Those with green background function like the equivalent ATmega328p register, but with some differences as noted. Those with blue background are new registers added for XLR8 functions. Those with white background are ATmega328p registers that do not exist in XLR8.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xFF)	Reserved	–	–	–	–	–	–	–	–	
(0xFE)	Reserved	–	–	–	–	–	–	–	–	
(0xFD)	SVPWH	–	–	–	–	Servo Pulse Width High Register				6.1.5
(0xFC)	SVPWL	Servo Pulse Width Low Register								6.1.5
(0xFB)	Reserved	–	–	–	–	–	–	–	–	
(0xFA)	SVCR	SVEN	SVDIS	SVUP	SVCHAN					6.1.5
(0xF9)	Reserved	–	–	–	–	–	–	–	–	
(0xF8)	Reserved	–	–	–	–	–	–	–	–	
(0xF7)	NEOD2	NeoPixel Data 2 register, function depends on NEOCMD								6.1.6
(0xF6)	NEOD1	NeoPixel Data 1 register, function depends on NEOCMD, often high half of pixel address/length								6.1.6
(0xF5)	NEOD0	NeoPixel Data 0 register, function depends on NEOCMD, often low half of pixel address/length								6.1.6
(0xF4)	NEOCR	NEOPIN				NEOCMD				6.1.6
(0xF3)	Reserved	–	–	–	–	–	–	–	–	
(0xF2)	Reserved	–	–	–	–	–	–	–	–	
(0xF1)	Reserved	–	–	–	–	–	–	–	–	
(0xF0)	Reserved	–	–	–	–	–	–	–	–	
(0xEF)	Reserved	–	–	–	–	–	–	–	–	
(0xEE)	Reserved	–	–	–	–	–	–	–	–	
(0xED)	Reserved	–	–	–	–	–	–	–	–	
(0xEC)	Reserved	–	–	–	–	–	–	–	–	
(0xEB)	Reserved	–	–	–	–	–	–	–	–	
(0xEA)	Reserved	–	–	–	–	–	–	–	–	
(0xE9)	Reserved	–	–	–	–	–	–	–	–	
(0xE8)	Reserved	–	–	–	–	–	–	–	–	
(0xE7)	Reserved	–	–	–	–	–	–	–	–	

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xE6)	Reserved	–	–	–	–	–	–	–	–	
(0xE5)	Reserved	–	–	–	–	–	–	–	–	
(0xE4)	Reserved	–	–	–	–	–	–	–	–	
(0xE3)	Reserved	–	–	–	–	–	–	–	–	
(0xE2)	Reserved	–	–	–	–	–	–	–	–	
(0xE1)	Reserved	–	–	–	–	–	–	–	–	
(0xE0)	Reserved	–	–	–	–	–	–	–	–	
(0xDF)	Reserved	–	–	–	–	–	–	–	–	
(0xDE)	Reserved	–	–	–	–	–	–	–	–	
(0xDD)	Reserved	–	–	–	–	–	–	–	–	TWAMR1
(0xDC)	Reserved	–	–	–	–	–	–	–	–	TWCR1
(0xDB)	Reserved	–	–	–	–	–	–	–	–	TWDR1
(0xDA)	Reserved	–	–	–	–	–	–	–	–	TWAR1
(0xD9)	Reserved	–	–	–	–	–	–	–	–	TWSR!
(0xD8)	Reserved	–	–	–	–	–	–	–	–	TWBR1
(0xD7)	Reserved	–	–	–	–	–	–	–	–	
(0xD6)	XLR8VERT	XLR8 Version Number Flags								6.1.1
(0xD5)	XLR8VERH	XLR8 Version Number Register High Byte								6.1.1
(0xD4)	XLR8VERL	XLR8 Version Number Register Low Byte								6.1.1
(0xD3)	Reserved	–	–	–	–	–	–	–	–	
(0xD2)	FCFGDAT	FPGA Reconfiguration Data Register								6.1.4
(0xD1)	FCFGSTS	FCFGDN	0	FCFGFM	FCFGRDY	–	–	–	–	6.1.4
(0xD0)	FCFGCTL	–	FCFGSEC			–	FCFGCMD		FCFGEN	6.1.4
(0xCF)	FCFGCID	Chip ID register								6.1.2
(0xCE)	Reserved	–	–	–	–	–	–	–	–	UDR1
(0xCD)	Reserved	–	–	–	–	–	–	–	–	UBBR1H
(0xCC)	Reserved	–	–	–	–	–	–	–	–	UBBR1L
(0xCB)	Reserved	–	–	–	–	–	–	–	–	UCSR1D
(0xCA)	Reserved	–	–	–	–	–	–	–	–	UCSR1C
(0xC9)	Reserved	–	–	–	–	–	–	–	–	UCSR1B
(0xC8)	Reserved	–	–	–	–	–	–	–	–	UCSR1A
(0xC7)	Reserved	–	–	–	–	–	–	–	–	
(0xC6)	UDR0	USART I/O Data Register								
(0xC5)	UBRR0H	–	–	–	–	USART Baud Rate Register High				
(0xC4)	UBRR0L	USART Baud Rate Register Low								
(0xC3)	Reserved	–	–	–	–	–	–	–	–	UCSR0D
(0xC2)	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01/ UDORD0	UCSZ00/ UCPHA0	UCPOL0	3.8
(0xC1)	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	
(0xC0)	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	
(0xBF)	Reserved	–	–	–	–	–	–	–	–	
(0xBE)	Reserved	–	–	–	–	–	–	–	–	
(0xBD)	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	–	
(0xBC)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	
(0xBB)	TWDR	2-wire Serial Interface Data Register								
(0xBA)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	
(0xB9)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	
(0xB8)	TWBR	2-wire Serial Interface Bit Rate Register								
(0xB7)	Reserved	–	–	–	–	–	–	–		
(0xB6)	Reserved	–	–	–	–	–	–	–		ASSR 4.1
(0xB5)	Reserved	–	–	–	–	–	–	–	–	
(0xB4)	OCR2B	Timer/Counter2 Output Compare Register B								
(0xB3)	OCR2A	Timer/Counter2 Output Compare Register A								
(0xB2)	TCNT2	Timer/Counter2 (8-bit)								
(0xB1)	TCCR2B	FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20	

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xB0)	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20	
(0xAF)	Reserved	–	–	–	–	–	–	–	–	
(0xAE)	Reserved	–	–	–	–	–	–	–	–	SPDR1
(0xAD)	Reserved	–	–	–	–	–	–	–	–	SPSR1
(0xAC)	Reserved	–	–	–	–	–	–	–	–	SPCR1
(0xAB)	Reserved	–	–	–	–	–	–	–	–	OCR4BH
(0xAA)	Reserved	–	–	–	–	–	–	–	–	OCR4BL
(0xA9)	Reserved	–	–	–	–	–	–	–	–	OCR4AH
(0xA8)	Reserved	–	–	–	–	–	–	–	–	OCR4AL
(0xA7)	Reserved	–	–	–	–	–	–	–	–	ICR4H
(0xA6)	Reserved	–	–	–	–	–	–	–	–	ICR4L
(0xA5)	Reserved	–	–	–	–	–	–	–	–	TCNT4H
(0xA4)	Reserved	–	–	–	–	–	–	–	–	TCNT4L
(0xA3)	Reserved	–	–	–	–	–	–	–	–	
(0xA2)	Reserved	–	–	–	–	–	–	–	–	TCCR4C
(0xA1)	Reserved	–	–	–	–	–	–	–	–	TCCR4B
(0xA0)	Reserved	–	–	–	–	–	–	–	–	TCCR4A
(0x9F)	Reserved	–	–	–	–	–	–	–	–	
(0x9E)	Reserved	–	–	–	–	–	–	–	–	
(0x9D)	Reserved	–	–	–	–	–	–	–	–	
(0x9C)	Reserved	–	–	–	–	–	–	–	–	
(0x9B)	Reserved	–	–	–	–	–	–	–	–	OCR3BH
(0x9A)	Reserved	–	–	–	–	–	–	–	–	OCR3BL
(0x99)	Reserved	–	–	–	–	–	–	–	–	OCR3AH
(0x98)	Reserved	–	–	–	–	–	–	–	–	OCR3AL
(0x97)	Reserved	–	–	–	–	–	–	–	–	ICR3H
(0x96)	Reserved	–	–	–	–	–	–	–	–	ICR3L
(0x95)	Reserved	–	–	–	–	–	–	–	–	TCNT3H
(0x94)	Reserved	–	–	–	–	–	–	–	–	TCNT3L
(0x93)	Reserved	–	–	–	–	–	–	–	–	
(0x92)	Reserved	–	–	–	–	–	–	–	–	TCCR3C
(0x91)	Reserved	–	–	–	–	–	–	–	–	TCCR3B
(0x90)	Reserved	–	–	–	–	–	–	–	–	TCCR3A
(0x8F)	Reserved	–	–	–	–	–	–	–	–	
(0x8E)	Reserved	–	–	–	–	–	–	–	–	
(0x8D)	Reserved	–	–	–	–	–	–	–	–	
(0x8C)	Reserved	–	–	–	–	–	–	–	–	
(0x8B)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte								
(0x8A)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte								
(0x89)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte								
(0x88)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte								
(0x87)	ICR1H	Timer/Counter1 – Input Capture Register High Byte								
(0x86)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte								
(0x85)	TCNT1H	Timer/Counter1 – Counter Register High Byte								
(0x84)	TCNT1L	Timer/Counter1 – Counter Register Low Byte								
(0x83)	Reserved	–	–	–	–	–	–	–	–	
(0x82)	TCCR1C	FOC1A	FOC1B	–	–	–	–	–	–	
(0x81)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	
(0x80)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	
(0x7F)	Reserved	–	–	–	–	–	–	–	–	DIDR1 3.9
(0x7E)	DIDR0	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	
(0x7D)	XLR8ADCR	AD12EN	–	–	–	–	–	–	–	6.1.8
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	3.5
(0x7B)	ADCSRB	–	-	–	–	–	ADTS2	ADTS1	ADTS0	ACME 3.9

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	
(0x79)	ADCH	ADC Data Register High byte								
(0x78)	ADCL	ADC Data Register Low byte								
(0x77)	Reserved	–	–	–	–	–	–	–	–	
(0x76)	Reserved	–	–	–	–	–	–	–	–	
(0x75)	Reserved	–	–	–	–	–	–	–	–	
(0x74)	Reserved	–	–	–	–	–	–	–	–	
(0x73)	Reserved	–	–	–	–	–	–	–	–	PCMSK3
(0x72)	Reserved	–	–	–	–	–	–	–	–	TIMSK4
(0x71)	Reserved	–	–	–	–	–	–	–	–	TIMSK3
(0x70)	TIMSK2	–	–	–	–	–	OCIE2B	OCIE2A	TOIE2	
(0x6F)	TIMSK1	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	
(0x6E)	TIMSK0	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	
(0x6D)	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	
(0x6C)	PCMSK1	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	
(0x6B)	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	
(0x6A)	Reserved	–	–	–	–	–	–	–	–	
(0x69)	EICRA	–	–	–	–	ISC11	ISC10	ISC01	ISC00	
(0x68)	PCICR	–	–	–	–	–	PCIE2	PCIE1	PCIE0	
(0x67)	Reserved	–	–	–	–	–	–	–	–	
(0x66)	Reserved	–	–	–	–	–	–	–	–	OSCCAL 4.1
(0x65)	Reserved	–	–	–	–	–	–	–	–	
(0x64)	PRR	–	–	–	PRINTOSC	–	–	–	–	4.2, 6.1.3
(0x63)	Reserved	–	–	–	–	–	–	–	–	
(0x62)	Reserved	–	–	–	–	–	–	–	–	XFDCSR
(0x61)	Reserved	–	–	–	–	–	–	–	–	CLKPR 4.2
(0x60)	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	
0x3F(0x5F)	SREG	I	T	H	S	V	N	Z	C	
0x3E(0x5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8	
0x3D(0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	
0x3C(0x5C)	Reserved	–	–	–	–	–	–	–	–	
0x3B(0x5B)	XFR3	XLR8 Function (floating point) 32 bit Result High Byte								6.1.7
0x3A(0x5A)	XFR2	XLR8 Function (floating point) 32 bit Result Byte								6.1.7
0x39(0x59)	XFR1	XLR8 Function (floating point) 32 bit Result Byte								6.1.7
0x38(0x58)	XFR0	XLR8 Function (floating point) 32 bit Result Low Byte								6.1.7
0x37(0x57)	SPMCSR	SPMIE	RWWSB	SIGRD	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	
0x36(0x56)	Reserved	–	–	–	–	–	–	–	–	
0x35(0x55)	Reserved	–	–	–	–	–	–	–	–	MCUCR 4.1,3.4,3.6
0x34(0x54)	MCUSR	–	–	–	–	WDRF	–	EXTRF	PORF	BORF 4.1
0x33(0x53)	Reserved	–	–	–	–	–	–	–	–	SMCR 4.2
0x32(0x52)	Reserved	–	–	–	–	–	–	–	–	
0x31(0x51)	Reserved	–	–	–	–	–	–	–	–	DWDR
0x30(0x50)	Reserved	–	–	–	–	–	–	–	–	ACSR 3.9
0x2F(0x4F)	Reserved	–	–	–	–	–	–	–	–	ACSRB
0x2E(0x4E)	SPDR	SPI Data Register								
0x2D(0x4D)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X	
0x2C(0x4C)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	
0x2B(0x4B)	GPIOR2	General Purpose I/O Register 2								
0x2A(0x4A)	GPIOR1	General Purpose I/O Register 1								
0x29(0x49)	CLKSPD	Clock speed programming used by XLR8 bootloader								6.1.3
0x28(0x48)	OCR0B	Timer/Counter0 Output Compare Register B								
0x27(0x47)	OCR0A	Timer/Counter0 Output Compare Register A								
0x26(0x46)	TCNT0	Timer/Counter0 (8-bit)								

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
0x25(0x45)	TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	
0x24(0x44)	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	
0x23(0x43)	GTCCR	TSM	–	–	–	–	–	PSRASY	PSRSYNC	
0x22(0x42)	EEARH	EEPROM Address Register High Byte								EEPROM function could be added - 3.11
0x21(0x41)	EEARL	EEPROM Address Register Low Byte								
0x20(0x40)	EEDR	EEPROM Data Register								
0x1F(0x3F)	EECR	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	
0x1E(0x3E)	GPIOR0	General Purpose I/O Register 0								
0x1D(0x3D)	EIMSK	–	–	–	–	–	–	INT1	INT0	
0x1C(0x3C)	EIFR	–	–	–	–	–	–	INTF1	INTF0	
0x1B(0x3B)	PCIFR	–	–	–	–	–	PCIF2	PCIF1	PCIF0	
0x1A(0x3A)	Reserved	–	–	–	–	–	–	–	–	
0x19(0x39)	Reserved	–	–	–	–	–	–	–	–	TIFR4
0x18(0x38)	Reserved	–	–	–	–	–	–	–	–	TIFR3
0x17(0x37)	TIFR2	–	–	–	–	–	OCF2B	OCF2A	TOV2	
0x16(0x36)	TIFR1	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	
0x15(0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14(0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13(0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12(0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11(0x31)	XFSTAT	XFDONE	XFERR	–	–	–	–	–	–	6.1.7
0x10(0x30)	XFCTRL	–	XFSTART	–	–	–	XFCMD			6.1.7
0x0F(0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E(0x2E)	Reserved	–	–	–	–	–	–	–	–	328PB PORTE
0x0D(0x2D)	Reserved	–	–	–	–	–	–	–	–	328PB DDRE
0x0C(0x2C)	Reserved	–	–	–	–	–	–	–	–	328PB PINE
0x0B(0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	pullup 3.4
0x0A(0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	
0x09(0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	reset 3.4
0x08(0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	pullup 3.4
0x07(0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	
0x06(0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	
0x05(0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	pullup 3.4
0x04(0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	
0x03(0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	reset 3.4
0x02(0x22)	Reserved	–	–	–	–	–	–	–	–	
0x01(0x21)	Reserved	–	–	–	–	–	–	–	–	
0x0(0x20)	Reserved	–	–	–	–	–	–	–	–	
		= unchanged from ATmega328p								
		= ATmega328p registers not implemented in XLR8								
		= Some differences in XLR8 compared to ATmega328p								
		= new registers for XLR8 Blocks								
		= Reserved registers that are best not used for XLR8 blocks because ATmega328PB uses them								

6.1 XLR8 and XB Register Descriptions

6.1.1 XLR8VERL, XLR8VERH, XLR8VERT – Version Number Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xD6)	XLR8VERT	XLR8 Version Number Flags								

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xD5)	XLR8VERH	XLR8 Version Number Register High Byte								
(0xD4)	XLR8VERL	XLR8 Version Number Register Low Byte								
Read/Write		R	R	R	R	R	R	R	R	
Initial Value		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The version number register provides the FPGA design revision, while the version flags register indicates if the build had a mixed or modified version. The registers have a constant value for a particular design, but the value changes for each version. The easiest way to use these registers is with the XLR8Info library (https://github.com/AloriumTechnology/XLR8Info).

6.1.2 FCFGCID – Chip ID Register

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xCF)	FCFGCID	Chip ID register								
Read/Write		R	R	R	R	R	R	R	R	write-reset
Initial Value		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The chip ID register is a read-only register that provides chip ID information. Multiple bytes of chip ID information are available and each read presents the next byte. Writing the register (with any value) resets the read pointer back to the beginning (and does not store the write data in any way).

6.1.3 CLKSPD – Clock Speed Register

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
0x29(0x49)	CLKSPD	Clock speed programming used by XLR8 bootloader								
Read/Write		R	R	R	R	R	R	R	R	
Initial Value		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	
0x29(0x49)	CLKSPD	–	–	–	–	–	–	–	OSCOUT	
Read/Write		W	W	W	W	W	W	W	W	
Initial Value		N/A	N/A	N/A	N/A	N/A	N/A	N/A	0	
(0x64)	PRR	-	–	–	–	PRINTOSC	–	–	–	
Read/Write		R	R	R	R	R/W	R	R	R	
Initial Value		0	0	0	0	0	0	0	0	

The clock speed register holds a constant value that represents the value to be programmed into the UBRR0L register to run the UART at a baud rate of 115200. It is used by the modified bootloader mentioned in section 3.6 to allow it to run correctly regardless of whether XLR8 is running 16MHZ, 32MHz, or some other speed.

XLR8 includes in on-chip oscillator that currently isn't being used, but a divide-by-1024 version of it can be output to digital pin 8 by writing bit 0 of the CLKSPD register high. This is a write-only operation, it does not change the value that is read from the CLKSPD register. The internal oscillator can be turned off entirely by setting the PRINTOSC bit of the PRR register. As described in section 3.12, the other bits of this register are currently unused.

6.1.4 FCFGDAT, FCFGSTS, FCFGCTL – FPGA Reconfiguration Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xD2)	FCFGDAT	FPGA Reconfiguration Data Register								
Read/Write		W	W	W	W	W	W	W	W	
Initial Value		0	0	0	0	0	0	0	0	
(0xD1)	FCFGSTS	FCFGDN	FCFGOK	FCFGFAIL	FCFGRDY	–	–	–	–	
Read/Write		R/W1C	R/W1C	R/W1C	R	R	R	R	R	
Initial Value		0	0	0	0	0	0	0	0	
(0xD0)	FCFGCTL	–	FCFGSEC			–	FCFGCMD		FCFGEN	
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	

These registers are used during reconfiguration of the FPGA and are not intended for customer use. FCFGEN auto-clears after a reconfiguration is complete. The Data register is a write-only register.

6.1.5 SVPWH, SVPWL, SVCR – Servo XB Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xFD)	SVPWH	–	–	–	–	Servo Pulse Width High Register				
Read/Write		R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	
(0xFC)	SVPWL	Servo Pulse Width Low Register								
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	
(0xFA)	SVCR	SVEN	SVDIS	SVUP	SVCHAN					
Read/Write		R/W	W	W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	

The servo data registers SVPWH and SVPWL represent the desired servo pulse width in microseconds. The value is programmed to the channel selected by SVCHAN when the SVCR register is written with the update (SVUP) bit set. The channel can be enabled to begin at the same time by also setting the enable (SVEN) bit. A channel is disabled by writing SVCR with the desired channel in the SVCHAN field, the SVEN bit clear and the SVDIS set. The pulse width of a channel can be changed without changing its enabled/disabled status by leaving the SVEN and SVDIS bits clear when writing the SVCR register. SVDIS and SVUP are strobes and will always read zero. Reading SVEN will give the current enabled/disabled status of the channel read in the SVCHAN field. The value of SVCHAN corresponds to the Arduino pin to use (i.e. 0=RX, 1=TX, 2=D2, …, 14=A0, etc.). Multiple pins can be driven simultaneously, each with a different pulse width…with a small limitation. The 32 possible values of SVCHAN directly alias to the 16 available timers (e.g. channels 1 and 17 could both be enabled, but they would always have the same pulse width of whichever one was programmed most recently). The easiest way to use these registers is with the XLR8Servo library (https://github.com/AloriumTechnology/XLR8Servo).

6.1.6 NEOD2, NEOD1, NEOD0, NEOCR – NeoPixel XB Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0xF7)	NEOD2	NeoPixel Data 2 register, function depends on NEOCMD								
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	
(0xF6)	NEOD1	NeoPixel Data 1 register, function depends on NEOCMD, often high half of pixel address/length								
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	
(0xF5)	NEOD0	NeoPixel Data 0 register, function depends on NEOCMD, often low half of pixel address/length								
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	
(0xF4)	NEOCR	NEOPIN				NEOCMD				
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	

The NeoPixel register usage varies depending on the value of the NEOCMD field in the NeoPixel control register (NEOCR).

NEOPIN : For the show command, this specifies which pin to send the data to. For no-op a pin number or other ID can be used. For other commands this field is ignored and not stored. For show the value clears to zero when the command completes and thereby can be used as a busy indication. While the hardware doesn't provide any locks, this may help software manage having multiple objects share this hardware nicely. The outputs are indexed starting at 1 because using the clear to zero to indicate busy/notbusy wouldn't work with pin 0.

NEOCMD : 0000 = no-op, but set PinNum/BusyID (won't autoclear until after show)
0001 = get memsize. D2=cmd buf size (entries), D1/D0= pixel mem size (bytes)
0010 = show WS2812. D2=starting cmd buffer D1/D0=length (bytes)
0011 = show WS2811. D2=starting cmd buffer D1/D0=length (bytes)
0100-0111 = Reserved.
1000 = set color. D2=color value, D1/D0=memory address (autoincrement)
1001 = set cmd buf entry-addr. D2=cmd buf addr, D1/D0=section start addr
1010 = set cmd buf entry-length. D2=cmd buf addr, D1/D0=section length
1011 = set cmd buf entry-bright. D2=cmd buf addr, D1=reserved, D0=brightness
1100 = get color. D2=color value, D1/D0=memory address (autoincrement)
1101 = get cmd buf entry-addr. D2=cmd buf addr, D1/D0=section start addr
1110 = get cmd buf entry-length. D2=cmd buf addr, D1/D0=section length
1111 = get cmd buf entry-bright. D2=cmd buf addr, D1=reserved, D0=brightness

NEOD2/D1/D0 : 8b data registers used as described above

For set, and show operations, D2/D1/D0 are loaded before doing the set/show cmd in the CNTL register (except possibly doing a no-op to set the busy ID).

For get color, D1/D0 are set first, then CNTL, then the color value can be read from D2.

For get cmd buf, D2 is set first, then CNTL, then the cmd buf info can be read from D1/D0.

For get memsize, CNTL is written, and then the memsize can be read from D2/D1/D0.

The easiest way to use these registers is with the XLR8NeoPixel library (https://github.com/AloriumTechnology/XLR8NeoPixel).

6.1.7 XFCTRL, XFSTAT, XFR0, XFR1, XFR2, XFR3– Floating Point XB Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
0x3B(0x5B)	XFR3	XLR8 Function (floating point) 32 bit Result High Byte								
0x3A(0x5A)	XFR2	XLR8 Function (floating point) 32 bit Result Byte								
0x39(0x59)	XFR1	XLR8 Function (floating point) 32 bit Result Byte								
0x38(0x58)	XFR0	XLR8 Function (floating point) 32 bit Result Low Byte								
Read/Write		R	R	R	R	R	R	R	R	
Initial Value		0	0	0	0	0	0	0	0	
0x11(0x31)	XFSTAT	XFDONE	XFERR	–	–	–	–	–	–	
Read/Write		R	R	R	R	R	R	R	R	
Initial Value		0	0	0	0	0	0	0	0	
0x10(0x30)	XFCTRL	–	XFSTART	–	–	–	XFCMD			
Read/Write		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value		0	0	0	0	0	0	0	0	

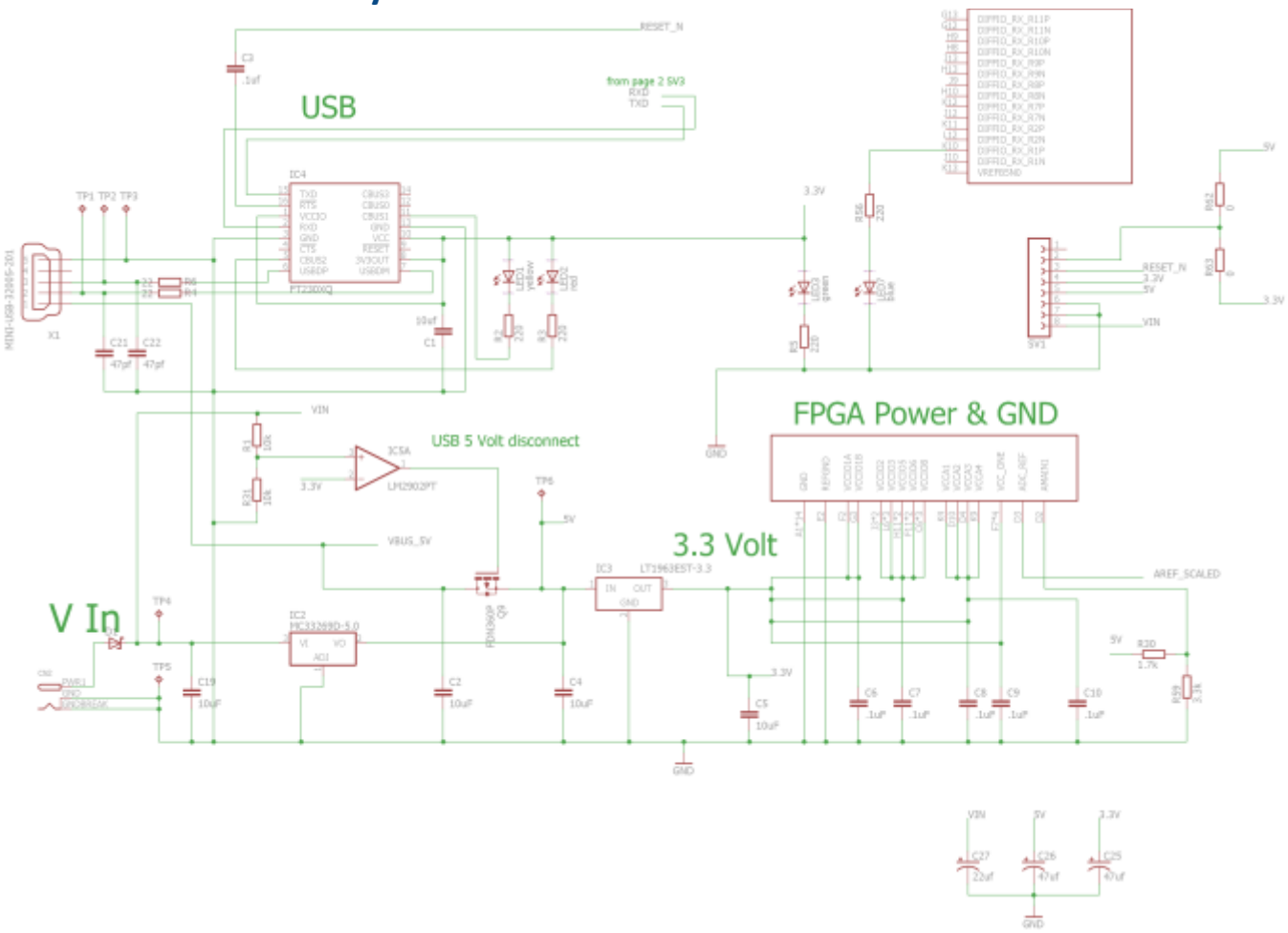
A floating-point calculation is started by writing the XFSTART bit in the XFCTRL register, along with the desired operation in the XFCMD field (1=add, 2=multiply, 3=divide). Operands come directly from the AVR's general-purpose register file (using our library ensures they will be in the right place). When the operation is done, the result appears in the XFR0/1/2/3 registers and the XFDONE status bit is set. If an unsupported XFCMD is used, the XFERR bit is also sets, allowing software to revert to using a software-based calculation. The XFSTAT register auto-clears when it is read, or when the next operation is started via writing the XFSTART bit.
The easiest way to use these registers is with the XLR8Float library (https://github.com/AloriumTechnology/XLR8Float).

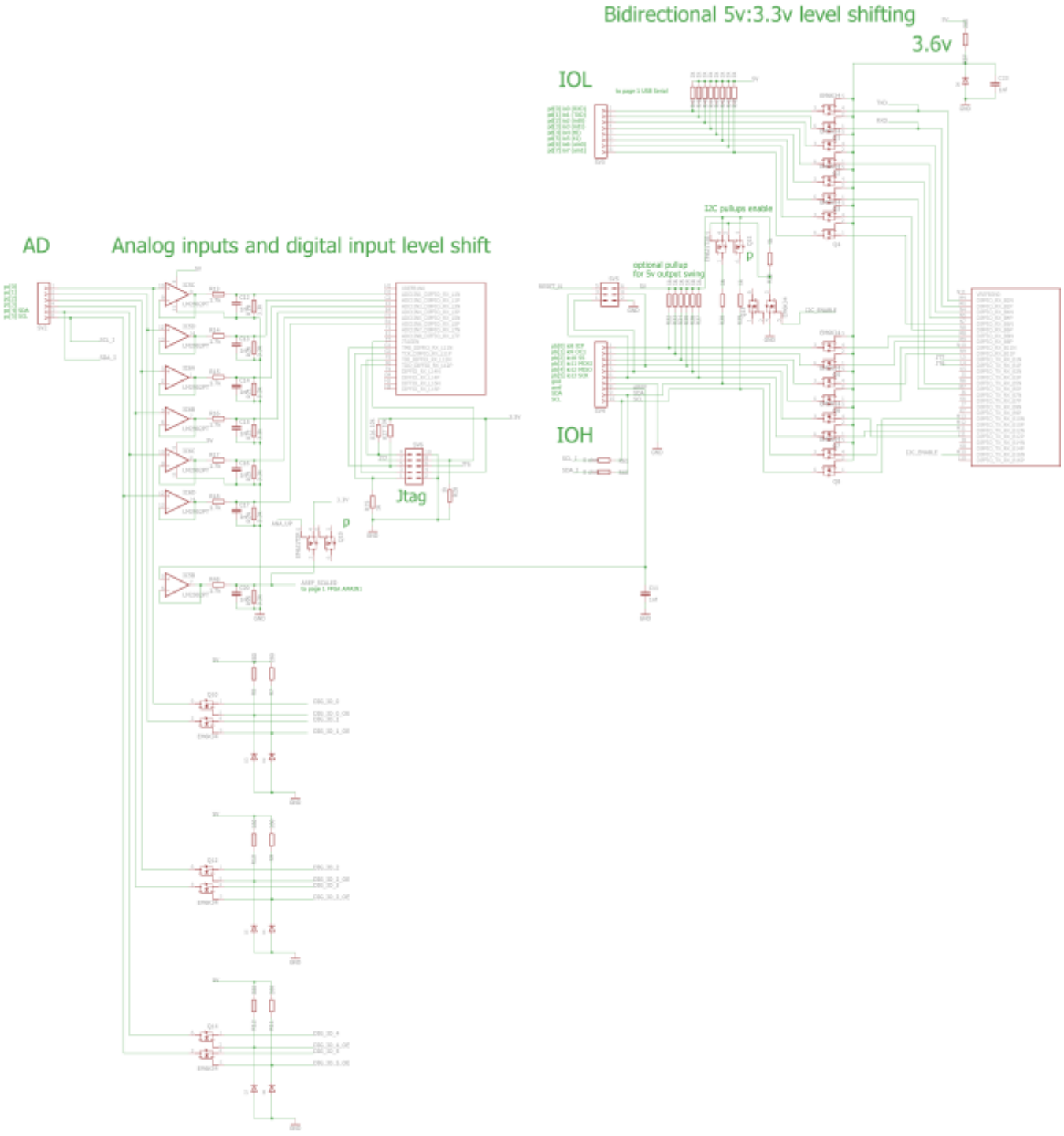
6.1.8 XLR8ADCR – XLR8 ADC Control Register

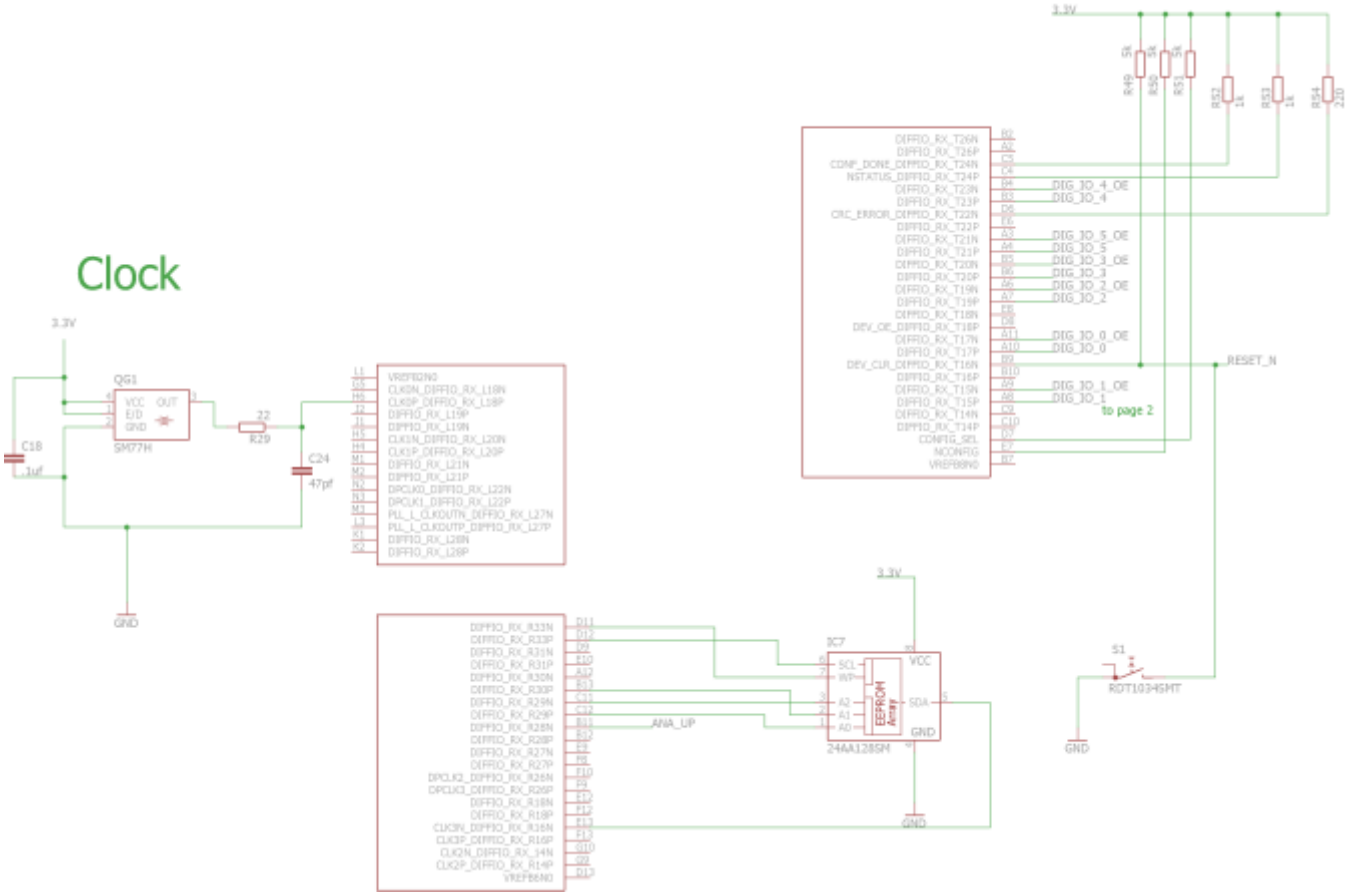
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Notes
(0x7D)	XLR8ADCR	AD12EN	–	–	–	–	–	–	–	
Read/Write		R/W	R	R	R	R	R	R	R	
Initial Value		0	0	0	0	0	0	0	0	

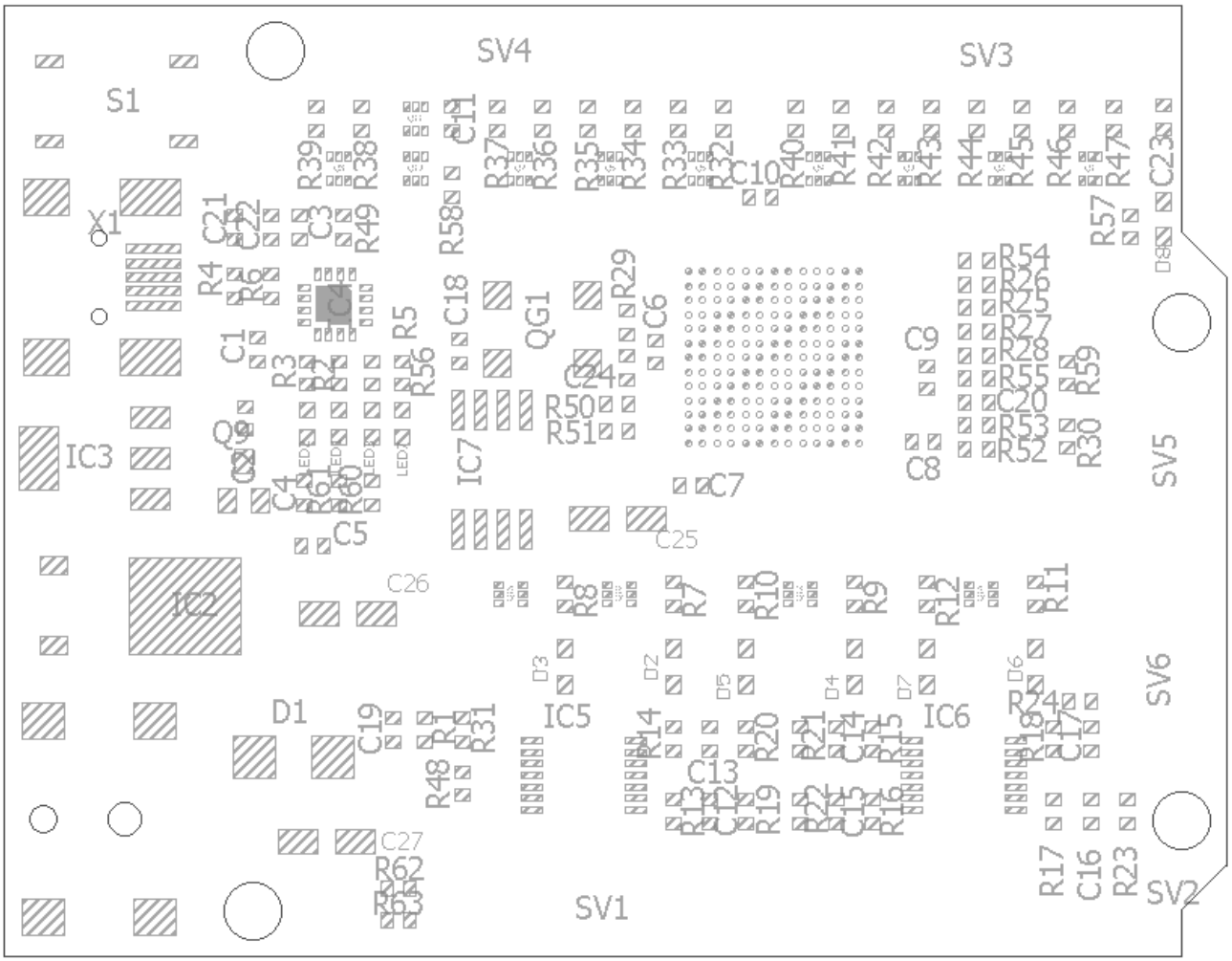
The AD12EN bit enables the ADC to run in 12 bit mode. The results reported in the ADCL and ADCH registers when running with ADLAR=0 can range from 0-4095, and when running with ADLAR=1, bits 5:4 of ADCL will include the least significant bits of the 12 bit ADC result. When running in 10 bit mode (the default to match Arduino), the result is truncated (not rounded) from the 12 bit result.

7 Schematics and Layout









8 Credits

Some code is used and modified from the AVR core written by Ruslan Lepetenok (lepetenokr@yahoo.com) that is available at http://opencores.com/project,avr_core. Ruslan's AVR core does not contain copyright or license notices, but we certainly wish to recognize its contribution to this project.

The I2C module builds upon the I2C core written by Richard Herveille (richard@asics.ws) that is available at http://opencores.org/project,i2c. The I2C core was released under BSD license with the following copyright statement:

> Copyright (C) 2001 Richard Herveille
> richard@asics.ws
>
> This source file may be used and distributed without
> restriction provided that this copyright statement is not
> removed from the file and that any derivative work contains
> the original copyright notice and the associated disclaimer
>
> THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY
> EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
> TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
> FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL THE AUTHOR
> OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
> INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
> (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
> GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
> BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
> LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
> (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
> OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
> POSSIBILITY OF SUCH DAMAGE.

All other portions of XLR8 were designed and verified for Alorium Technology by the excellent team at Superion Technology.