Hex Logic Detector
with AVR128DB28

2022-04-23 / Henning Schou **http://henning-schou.dk/electronics/HexLogicDetector**

The motivation for this project was partly that I needed one for debugging more complicated projects, and partly that I wanted to explore some of the new features of the AVR DB series.

The Hex Logic Level Detector uses **all** of the AVR128DB28's IO-pins in the prototype-friendly 28-pins DIL package. It doesn't come near to use all of the available memory – an AVR32DB28 would be completely adequate.

This construction goes hand in hand with the Hex Logic Generator project:
http://henning-schou.dk/electronics/HexLogicGenerator

Features

- Six logic channels.
- Logic levels: Low, high, or floating displayed by red and green LEDs
- Edge detection: Blue LEDs flash when level changes
- Channels 2, 3, 4 & 5 detect pulses as short as 25 ns. (Channels 0 & 1: 50 ns)
- Supply voltage span: 2.5 – 5.0 V (U_B – MCU and channels 3, 4 & 5)
- Three channels can operate at different voltage: 1.65 – 5.0 V (U_A – channels 0, 1 & 2)
- System resets when primary supply voltage is too low (U_B < 2.45 V)
- Indicator for insufficient secondary supply voltage (U_A < 1.65 V)

Logic levels

An input voltage is identified as Logic LOW in the interval 0 % to 20 % of U_{SUPP}.

An input voltage is identified as Logic HIGH in the interval 80 % to 100 % of U_{SUPP}.

The supply voltage U_{SUPP} is U_A for channels 0, 1 and 2, and U_B for channels 3, 4 and 5.

"Invalid" voltage levels, i.e. lying in the interval 20 % to 80 % of U_{SUPP}, will be flagged by a special LED combination (see below). The hardware ensures that a floating input will be at an invalid voltage level.

LED colour combinations explained

R	G	B	Interpretation
■ ■			LOW level
	■ ■		HIGH level
(any)		■	Blue flash indicates a level change (LOW to HIGH or HIGH to LOW)
■ ■		■	Short, positive pulse
	■ ■	■	Short, negative pulse
■ ■	■ ■	■ ■	Lots of logic transitions. Intensity differences Red/Green may give a hint of duty cycle
■ ■	■ ■		Invalid input (floating). (Notice: NO blue flashes)
■ ■	■ ■		"Waving" R/G intensity variations in ch. 0, 1 & 2: U_A is too low
			All LEDs off: U_B is too low

Hardware

MCU: AVR128PB28, 28-pin DIL package (any AVR DB processor is fine)

LEDs: Red, green, blue (6 each) – all Ø 3 mm

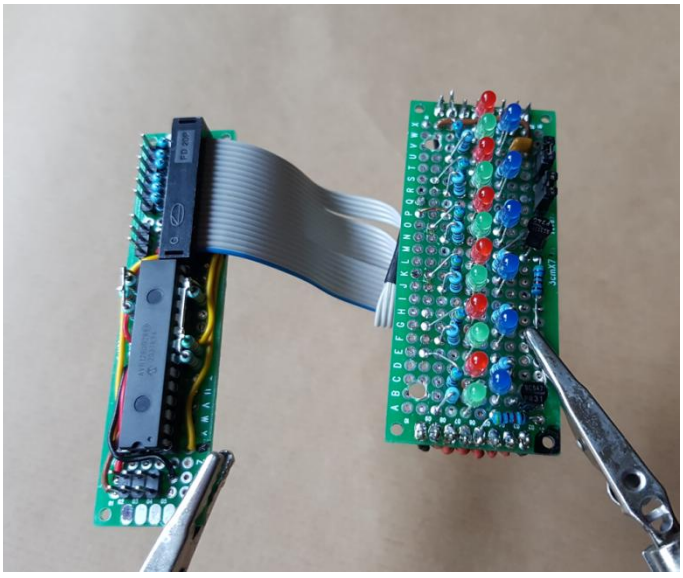
Resistors: 1 kΩ (18), 4.7 kΩ (6), 33 kΩ (6) – all standard 5 %, 0.25 W or similar *)

Capacitors: 100 nF ceramic (4), 2.2 µF tantalum (3)

Misc.: Housing, prototyping boards, 28-pin socket, pin headers, wires...

*) Check the brightness of the LEDs and select resistor values so that the red and green LEDs looks equally bright. It is OK if the blue LED looks a bit brighter.

Wires for the six inputs were chosen to correspond to the resistor colour codes for 0 to 5 (black, brown, red, orange, yellow, green).
Wires for supply voltages and ground were chosen to be different from the input wires:
Ground – Gray / U_A – Violet / U_B – Blue.



Practical construction

To fit inside the housing, the circuit was laid out on two perforated prototyping boards. One board holds the LEDs, the series resistors, and a few other components The other board accommodates the rest. The circuit was built by point-to-point wiring. The two boards are connected by a 20-lead flat cable.

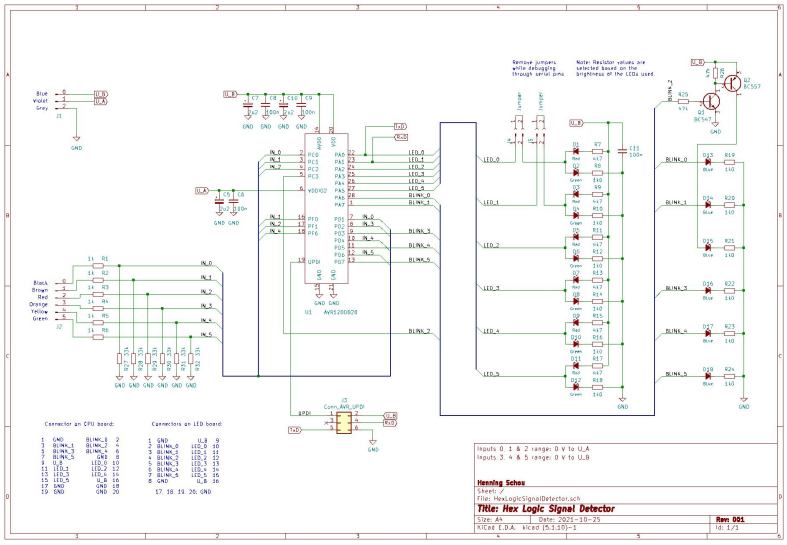
The prototyping boards makes the LED positions align with a 0.1 inch grid. You can use a third perfboard as a drilling template for the front panel.

The power and input cables are detachable. The connector pins are not very solid and you will need to replace these cables when a connector is damaged.

Hardware design choices

Get a larger version of the schematics by clicking on the link below:

http://henning-schou.dk/common/fdml.php?f=HexLogicDetector/f/HexLogicSignalDetector_Schematics.pdf



If you haven't got it yet, grab the AVR128DB28/32/48/64 datasheet from Microchip:

https://ww1.microchip.com/downloads/en/DeviceDoc/AVR128DB28-32-48-64-DataSheet-DS40002247A.pdf

The paragraph on "IO Multiplexing and Considerations" shows in compact form the functions of the various pins.

Input and output pins

We need to control the red and green LEDs as sketched previously. But there are not enough pins if red and green LEDs were to be controlled individually. Instead the LEDs (with their series resistors) are connected in series from U_B to ground, with the MCU output connecting to the centre of the series connection. Using PWM (Pulse Width Modulation) outputs, the LED brightness can be varied from red fully turned on to green fully turned on. Setting the PWM output for around 50% duty cycle, both LEDs will be lit equally. (The eyes won't judge 50% intensity to be much less than 100%. Nature seems to favour sensors with logarithmic characteristics.)

The requirement to deploy 6 channels of PWM – along with the other decisions below – leads to the use of timer TCA0 in split mode, and the use of pins PA0 to PA5.

In order to accommodate two different voltage domains, three of the four PORTC pins must be used for inputs. PC0, PC1 & PC2 were chosen.

If very short pulses are to be detected, we need to use edge triggered interrupts. The AVR DB series allows us to use any input pin to trigger interrupts. Checking the input state is usually synchronous with the internal clock, but each port features two bits (2 & 6) that has *fully asynchronous edge detection*. We use these pins as much as possible for inputs – but only one of them falls in the available PORTC range. Therefore, inputs 0 & 1 are slightly slower than the rest. The fully asynchronous pins used are PC2, PD2, PD6 & PF6.

Detecting floating inputs requires that the actual voltage on the inputs are measured. Two of the pins chosen so far allow both analog and digital inputs simultaneously (PD2 & PD6). The rest need to be coupled in parallel with an analog input. We use PD1, PD4, PF0 & PF1.

This leaves the following pins for the blue LEDs: PA6, PA7, PC3, PD3, PD5 & PD7. However, the HIGH level of PC3 is U_A – which is allowed to vary between 1.65 and 5 V. This is not suited for driving an LED, so it is level shifted to U_B using two transistors.

Note that a high channel 0, 1 or 2 input can power the input section for these channels through a protection diode, so that the diodes will show correct levels even if the U_A power input is left with no connection. This situation should be avoided.

Detection of floating inputs

Inputs have pull-up resistors enabled (nominally 20 kΩ). Each input has an external 33 kΩ resistor to ground. When left floating, this results in a voltage around 60 % of VDD.

This voltage can directly be measured by the ADC. It is possible to make the ADC raise a flag when the result lies within a given interval. We use this technique - being careful *not* to read the ADC result too early as this will automatically reset the flag. We define the “floating” interval to go from 20% to 80% of the relevant supply voltage.

- Floating inputs 0, 1, 2

Digital port pins: PC0, PC1, PC2 (VDDIO2 domain). PC2 interrupt is “fully asynchronous”.
Analog port pins: PD1 (AIN1), PF0 (AIN16), PF1 (AIN17).

Digital ports remain connected while sensing – the analog inputs are connected in parallel. We enable pullup on *one* of the pins in each pair.

VDDIO2/10 (i.e. 10% of VDDIO2) can be directly measured by the ADC. Left shifting this value 1 bit makes 20 % of VDDIO2; left shifting 3 bits gives 80 % of VDDIO2. It is assumed that U_B (which is used as reference for the ADC) doesn’t fluctuate too much.

- Floating inputs 3, 4, 5

Digital port pins: PD2, PF6, PD6. *All* three pin interrupt are “fully asynchronous”. Using PF6 requires ~RESET to be disabled - this is set by one of the fuses (more about this later).
Analog port pins: PD2 (AIN2), PD4 (AIN4), PD6 (AIN6).

PF6 and PD4 are coupled in parallel – only *one* of them has pull-up enabled. The other two pins work simultaneously as digital and analog inputs.

Because we use U_B as the reference for the ADC, we can define the 20% and 80 % interval limits simply as 0.2*ADC_MAX, resp. 0.8*ADC_MAX.

Firmware and programming

The firmware was developed with Arduino IDE using DxCore.

DxCore is documented along with a detailed description of the AVR DA and DB series here:
https://github.com/SpenceKonde/DxCore

A direct link to installation instructions:
https://github.com/SpenceKonde/DxCore/blob/master/Installation.md

The new AVR MCUs don't use the ICSP interface known from e.g. the classic Arduino boards. Instead they are programmed through the so-called UPDI (Unified Program and Debug Interface). "Official" programmers usually connect via a 6-pin connector just like ICSP. You can find detailed descriptions of a couple of different ways to build a UPDI programmer here:
https://github.com/SpenceKonde/AVR-Guidance/blob/master/UPDI/jtag2updi.md

The 6-pin connector actually leaves room for some extra signals. This is great for adding serial communication for low-level debugging. I saw this "UPDI+" interface described at LeoNerd's Tindie pages (where you will also find a couple of UPDI programmers and some other very fine boards):
https://www.tindie.com/stores/leonerd/

LeoNerd's "big" programmer was sold out when I built the Hex Logic Signal Detector, so I had to roll my own using two separate USB to serial adapters (one for UPDI, and one for serial communication).
If interested, you can find the programmer described in detail here:
http://henning-schou.dk/electronics/UpdiProgrammer

(A note for serial communication: As all IO pins are already in use, you will need to disconnect the LEDs normally controlled by the pins used for TxD and RxD. As shown in the schematics, two jumpers can be removed to accomplish this.)

Setting up the Arduino IDE

After installing DxCore, you have a bunch of selections available in the Tools menu. Use these:

Board:	AVR DB-series (no bootloader)
Chip:	AVR128DB28
Clock speed:	24 MHz (internal)
Millis() Timer:	TCB2 (recommended)
BOD Level:	2.45 V [1])
BOD Mode:	Enabled/Enabled [1])
Save EEPROM:	EEPROM retained [1])
Reset Pin Function:	Input ("no output, ever") [1])
Startup Time:	8 ms [1])
Flash writing:	Disabled
MVIO:	Enabled
AttachInterrupt():	"On all pins as usual" [2])
printf():	"Full, 2.6k, prints floats" [3])
Programmer:	"SerialUPDI with 4.7k resistor or diode (239499 baud)" [4])

[1]) Several of the settings require you to click "burn bootloader" to activate. Just do that, then upload the real firmware using the programmer.

[2]) I have no idea what this is - but I guess it is described in the DxCore documentation in case you need to work with Arduino-style interrupt handling.

[3]) Flash memory space is not a problem here. Not used in this application anyway...

[4]) This works with my home-built programmer mentioned above.

Firmware details

Source code can be downloaded from the project web:
http://henning-schou.dk/electronics/HexLogicDetector

The code is extensively commented.
The following paragraphs will detail some parts of the code.

Turning on the blue LEDs

The code manipulates a lot of registers. We will typically do this through arrays of pointers to the registers involved; the index of the arrays being simply the channel number. Typically in combination with arrays of bitmasks, again indexed by the channel number.

For instance, we define these arrays to help us set the pins PA6, PA7, PC3, PD3, PD5, PD7:

```
register8_t* setBlink[MAX_CH+1] = {       // Registers for setting blue LED outputs HIGH:
    &PORTA.OUTSET, &PORTA.OUTSET, &PORTC.OUTSET, &PORTD.OUTSET, &PORTD.OUTSET, &PORTD.OUTSET };

const byte blinkBM[MAX_CH+1] = {          // Corresponding bitmasks for the output pins:
    bit(6), bit(7), bit(3), bit(3), bit(5), bit(7)};
```

Now it is easy to perform this task in a loop over channels:

```
*(setBlink[ch]) = blinkBM[ch];            // Set blue LED output HIGH for channel ch
```

Reacting on input changes

As mentioned previously, changes in the inputs trigger interrupts. The inputs are spread across three ports, so we need tree different interrupt routines. The ISRs set bits in a single, common flag byte. The bits in the flag byte are not all the same as their position in the port - there would be overlaps.
In the example below, the input pins PD2 and PD6 are left-shifted before they are transferred to the variable `blinkFlags`. Note that the original bit positions are used for resetting the interrupt flags. After an interrupt is triggered, the ISR disables it for the pin involved. The interrupt is re-enabled in the main code after a given time.

```
ISR(PORTD_PORT_vect) {
  byte flagsD; // Input pins used in PORTD: PD2 (IN_3), PD6 (IN_5)

  // Handling these bits in blinkFlags (after left shift):
  // bit3 ~ IN_3  ,  bit7 ~ IN_5
  flagsD = PORTD.INTFLAGS & (bit(6)|bit(2));
  blinkFlags |= (flagsD << 1);
  PORTD.INTFLAGS = flagsD; // clear interrupt flags

  // Disable interrupt(s):
  if ((flagsD & bit(2)) != 0) *(ctrlInputs[3]) = PIN_CTRL_INT_DIS;  // No interrupt (but pull-up)
  if ((flagsD & bit(6)) != 0) *(ctrlInputss[5]) = PIN_CTRL_INT_DIS; // No interrupt (but pull-up)
}
```

There are similar ISRs for servicing PORTC and PORTF.

This example shows how the edge/pulse indication works: Any change in an input will trigger an interrupt which sets a bit in a flag byte. In the main loop(), a function is called that checks each of the flag bits and turns on the blue LED for the corresponding input channel(s).

The same function checks how long the LED has been on, and turns the LED off again when the short blink period is over.

Static input levels

The voltages of the six input channels and the U_A pin are all checked periodically.

We need the voltage on the U_A pin to specify the allowed input voltage range for channel 0, 1 and 2. The ADC has a separate channel that is connected to 0.1 times U_A. The ADC output is smoothed and stored in a variable VDDIO2_10. The logic limits at 20 % and 80 % of U_A are obtained as 2, resp. 8, times the value of VDDIO2_10.

The corresponding limits for channels 3, 4 and 5 are constant values calculated as 0.2, resp. 0.8, times the maximum ADC output (which is 2^{12}-1 for a 12-bit ADC):

```
const word ADC_MAX = (1<<12)-1;             // Max result for 12-bit ADC
const word ADC_CENTER = ADC_MAX/2;          // Center of 12-bit ADC
const word ADC_20_PCT = 0.2*ADC_MAX;        // Low logic level maximum
const word ADC_80_PCT = 0.8*ADC_MAX;        // High logic level minimum
```

For each channel, we want to check if the voltage falls inside the 20% to 80% window (“illegal” values). If it does, the input is either floating or simply passing through the region on its way from one legal logic state to the other. Therefore we only set a boolean first time an “illegal” level is detected. If the level is OK next time we check, the “illegal” level was only a transient state and the boolean is reset. On the other hand, if the level is “illegal” two times in a row, the input is marked as floating.

The check is performed in hardware. The ADC has threshold registers that we set up like this:

```
if (channel<3) {                            // Set window according to logic levels
   ADC0.WINLT = VDDIO2_10 << 1;             // 20 % of VDDIO2
   ADC0.WINHT = VDDIO2_10 << 3;             // 80 % of VDDIO2
} else {
   ADC0.WINLT = ADC_20_PCT;                 // 20 % of ADC_MAX
   ADC0.WINHT = ADC_80_PCT;                 // 80 % of ADC_MAX
}
```

If the result of an ADC conversion falls within these thresholds, an interrupt flag is raised. We don’t enable the corresponding interrupt, but check the flag manually.

If the input voltage falls outside the “illegal” window, the logic level is still not accepted immediately. The level must be stable for a hold-off time (chosen as 40 ms) before we allow the LEDs to display the result.

Setting up and using TCA0 for PWM

For turning on *both* the red and the green LEDs (to indicate a floating input) - with only one output - we need to toggle the pin fast enough that the eye will not observe any flickering. This leads us to use pulse width modulation (PWM). This is also used for making the "waving" indication when VDDIO2 is too low. We use TCA0, the type A timer/counter for this. When running in split mode, it provides just the six PWM outputs we need.
The DxCore has by default some Arduino'ish use for TCA0 and has even routed the outputs to alternative pins. We will flush all of this and set up everything by hand:

Restoring the TCA0 back to scratch:

```
  takeOverTCA0();                     // Required to prevent DxCore from using TCA0 (for PWM)
  PORTMUX.TCAROUTEA = 0;              // DxCore has routed these pins - reset!
```

Changing to split mode, setting clock frequency, and enabling the counter:

```
  TCA0.SINGLE.CTRLA = 0;                                  // TCA0 disabled
  TCA0.SINGLE.CTRLESET =  TCA_SINGLE_CMD_RESET_gc;        // Reset counter
  TCA0.SINGLE.CTRLD =     TCA_SINGLE_SPLITM_bm;           // Set split mode
  TCA0.SPLIT.CTRLA =      TCA_SPLIT_CLKSEL_DIV64_gc |     // Clock freq DIV 64 (f_PWM = 1.465 kHz)
                          TCA_SPLIT_ENABLE_bm;            // TCA0 enabled
```

Setting the duty cycle of the PWM is performed by the function below. Duty cycle is `val`/256.
The output pins have been defined as outputs and have been set HIGH in setup(). This means that disabling the PWM output results in a constant HIGH output = 100% duty cycle.

```
void setDutyCycle(byte ch, int val) {
  const byte bm[MAX_CH+1] = {0,1,2,4,5,6};              // Bitmap (See doc. for CTRLB register)
  if (val>=256) {
    TCA0.SPLIT.CTRLB &= ~bit(bm[ch]);                   // Disable PWM output (normal pin HIGH
                                                        // output takes over)
  } else {
    TCA0.SPLIT.CTRLB |= bit(bm[ch]);                    // Enable PWM output
    *(cmpRegs[ch]) = val;                               // Set output compare register
  }
}
```

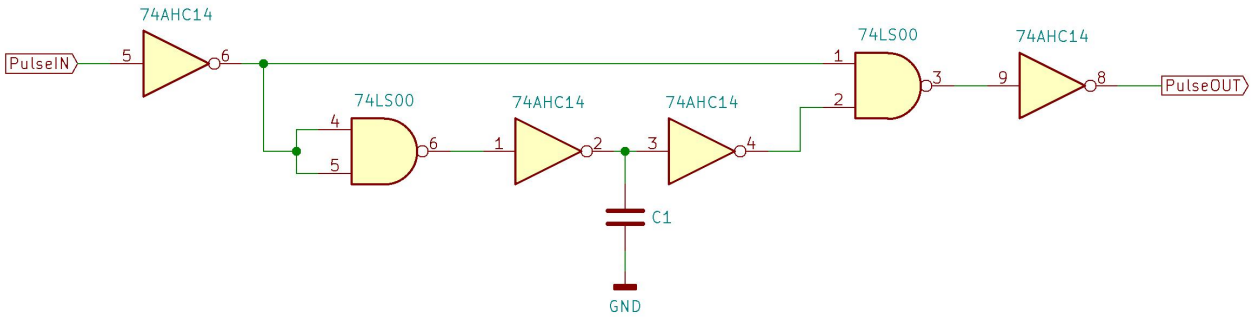
Testing

Testing was done with static input levels - both “legal” and “illegal” levels. Also with different supply voltages for the two input channel groups, including too low voltages. This is just a question of applying power supplies, a couple of potentiometers and a multimeter. Nothing exciting here.

The fun begins when you want to test the ability to detect short pulses.

I have a homebuilt pulse generator that can deliver pulse widths down to 50 ns. The repetition period can be made sufficiently long to check by naked eye that each and every pulse is marked by a blue LED blink.
This test was passed - for both polarities of pulses; 50 ns pulses were consistently detected by all inputs.

In order to test with shorter pulses, a couple of logic gates were used:



I used the outdated LS chip simply because that is what I had. More modern gates are probably better. The circuit was build on a solderless breadboard making it a bit sensitive to layout details. If you try to built this pulse shaper, be prepared to experiment and to monitor the results with an oscilloscope. Power lines must be decoupled close to the chips.
The capacitor C1 was in the range of 5 to 22 pF and was omitted for the shortest available pulse width.

The shortened pulses were applied to the inputs of channels 0 and 5. Of these, input 5 is fully asynchronous, input 0 is not. For comparison, the neighbouring channels 1 and 4 were connected to the incoming 50 ns pulses. The repetition frequency was 0.7 s. Each of the pairs of blue LEDs were observed for a couple of minutes.

With a pulse width of 37 ns, input 0 now and then failed to detect the pulses. Input 5 had no problems with 37 ms pulses.

Testing with 25 ns pulses - input 5 still OK.

Pulse width 18 ns resulted in some pulses missing from input 5. It doesn’t look like at sharp cut-off, it’s just that the probability of detections seems to fall off here. At 13 ns the drop-outs become more frequent.

Conclusion: The fully asynchronous inputs (channel 2, 3, 4 & 5) can reliably be used with pulses as short as 25 ns, whereas the remaining inputs (channels 0 & 1) are reliable down to 50 ns.

Boring stuff

Copyright

Disclaimer

The hardware and firmware described above has been build and tested with the results stated. It is my hope and belief that a moderately experienced hobbyist will be able to build a similar device based on this description. I cannot, however, provide any guarantee that this will be the case. The information contained in this description and the accompanying files is provides “as-is”. By choosing to use this information, you accept that the responsibility for obtaining a usable result is yours alone.

External links

Links within this document that point to external addresses were checked at the time of writing. If a link is broken or appears to point to irrelevant material, use Google and common sense to find what you need.
Likewise, information may get superseded by newer versions - this may especially be true for datasheets.