Debugging via data capture

Intro

A combination of some theoretical ranting and an actual project.

In this project I wanted to test a form of debugging which mearly records the system state and then enables one to replay it in an offline environment.

<Dusty academic mode on>
In this lab I wanted to test an idea I've had for some time, given a system with an initial state and then also given its input in a time series the end state of the system should be known. By storing the inputs in a sufficiently high resolution and asserting that the input values recorded are identical to the input values actually presented to the system, it should be possible to replay the scenario step by step, making observations along the way.
<Dusty academic mode off>

When doing some programming it is often not a good idea to use a debugger even if one is present. In real-time systems, timing often gets skewed and the only thing one can do is watch the state of the system at the break point, assuming that when hitting "continue" or similar we are almost certain to get bitten by "heisenbugs", i.e. bugs that either never manifests themselves in the debugger or only does manifest themselves in the debugger. So the procedure is usually:

1. Make an inspired guess of where the bug is
2. set a breakpoint
3. Spend a lot of manual time trying to get the system to produce the bug and hit the breakpoint
4. Look at the code, review your inspired guess.
5. goto 2)

Designing a simple system, an electronic dice

We've all been there, lets design something simply like out of basic engineering with some digital logic, an electronic dice and - oh - a push button would be nice to have as well!

Just some resistors+leds down to ground on mpu pins and then give them +5v feed for some extra juice. Put an arduino in and a few lines of code later you have at least one bug:

Code is available in the project pages

Of course you dove into the code and found the bug faster than I could write this sentence, but for the excercise we shall go about the troubleshooting in a much more complicated way. Just pretend this bug is truly obscure.

A simple "stub" for simulating the dice on a x86 platform

Since not all you guys have an ardu just lying around, after all they are like 5 euros and take some time to get from the webshop. So here is a simulator together with the original code. It's a rather simple thing defining a main and some stubs for the I/O including the dice.ino file.
The "in.txt" file contains the I part of the I/O operations and as since it feeds the inputs to the simulator when it does a 'digitalRead' the leds will flicker when we read a 0. In the simulator the flicker is replaced with patterns on stdout:

```
---------
|       |
|   X   |
|       |
---------

---------
|     X |
|       |
| X     |
---------

---------
|     X |
|   X   |
| X     |
---------

...
```

The patterns present themselves at about 1000 Hz so each time we read the button we spend 100 cycles counting.

in.txt looks something like this:

```
1
1
1
0
0
1
...
(much boring)
```

In our case the resolution is 1/10 of a second and each line corresponds to a read, so here after 0.3s the button was pressed for 0.2s etc.

The need for debugging arises

Following this scroll for a long time will make the "seven" pattern show up

```
---------
| X X X |
|   X   |
| X X X |
---------
```

This is a bit of a cheat since we haven't made it stop at seven, right now it is mearly counting in the 1000 Hz loop. To get to the real bug one has to set a breakpoint where the button is not pressed and all leds are lit (or prepare to scroll through a lot of text).

What we saw above when the dice was rolling we did not get this pattern, so this is most likely the bug we saw with the real dice. One way to now get to chase this bug will be a conditional break

```
(gdb) b dice.ino:88 if store[1]==0 && store[3]==0
(gdb) r ... ...
(gdb)
The leds correspond to this pattern of numbers in the store[] array in
dicesim.c:

---------
| 0 1 2 |
|   3   |
| 4 5 6 |
---------
```

After some time gdb will stop and two more "next"'s will lead us to the erroneous pattern when all leds are lit, i.e. they are all set to zero, we simplify this by using the fact that store[1] and store[3] are never set together, in fact store[1] is normally only set at "six" and store[3] is set at odd numbers only.

Traditional, conventional systems

Systems for this kind of troubleshooting, in circuit emulators and alike of course already exists and of course are out of budget for most hobbyists and also most professionals since debugging is as good as always a very low priority at most places of work. Consider just how much time is wasted on troubleshooting with silly printouts (that are also prone to introducing Heisenbugs). Add to that long turn-around times in building code and releasing and you get very large numbers which translates into cosy. It is something of an oddity that employers (and their customers) are still willing to pay for this.

The theory of recording

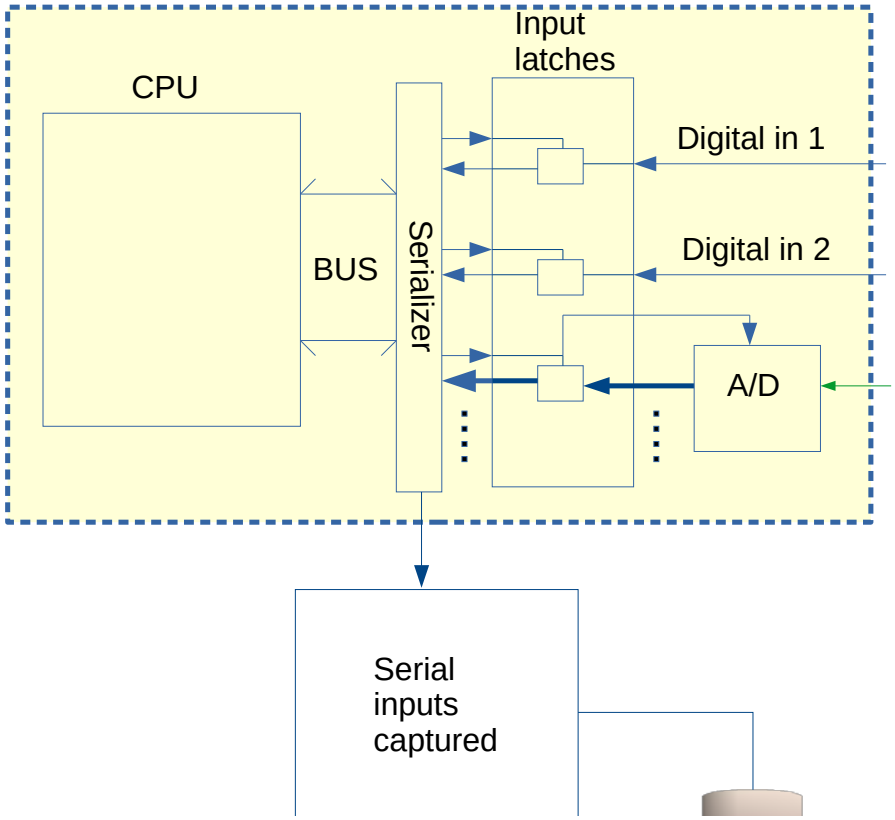
As stated above in the "dusty" section: If one could record all input to a given system, knowing its initial state, it would be possible to replay the whole scenario in a simulated environment.

What a computer like this could look like

Problem is to know at what time the computer reads its inputs, we could have an arbitrairy granularity of sampling the inputs and still miss important events or simply going out of sync in the replay later on and thus yielding a differing result from the live run.

We then conclude that one interesting alternative to todays systems would be if the computer as such outputed its read input data on some special pin. Obviously the data transfer rate on that pin would need to be enough to accomodate all input reads that the CPU performs, give and take some buffering. Alternatively an indicator could reliably signal an overflow, then allowing the designer to limit the amount of read from the inputs. One might for instance start to ask questions like:"Do we really need to sample this ambient temperature sensor at a steady 10 Mhz?"

This figure illustrates a possible implementation of the circuit inside the MCU chip as such.



Adding circuitry to enable recording

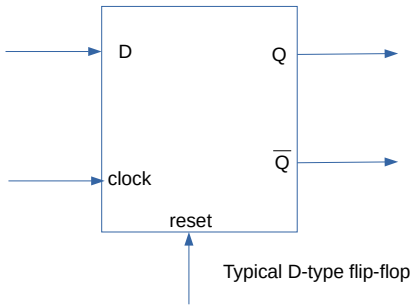
Since this make-your-own ASIC is clearly to expensive for an everyday hack one could consider making a proof of concept inside an fpga or similar, but for this we wanted an even simpler solution.

Realizing that the circuitry needn't be in the chip as such, adding external circuitry would also be a possibility. What we need is a mechanism that will tell us when the MPU wants an input sample and make sure this is also stored in our external circuitry holding the same value.

To get this thing going, besides an input signal we also need some way of knowing when the sampling is to take place, in our case we need to tell the circuitry when to clock in the value of the dice button.

The D-flop

The sampling is done via a simple D-flop, the classic 74hc74 dual flip-flop out of which we use one half.

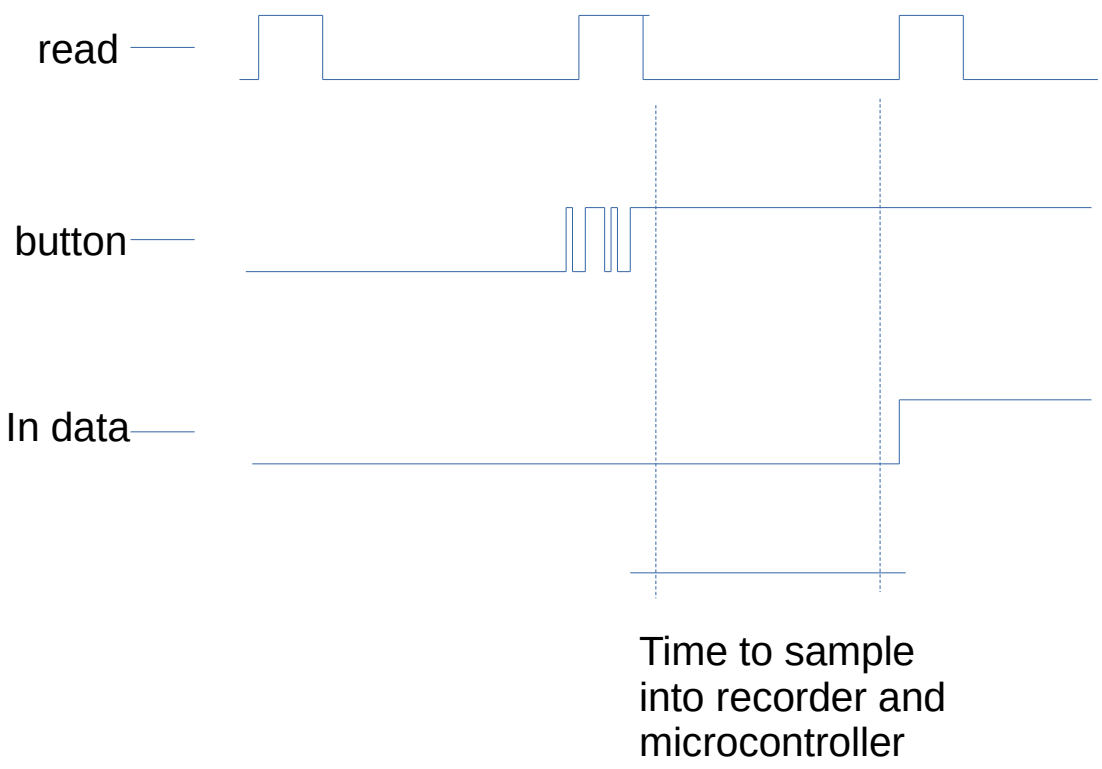


Typical D-type flip-flop

Here the idea is that one output from the nano is responsible for triggering the flip-flop and thus sample the value of the button. Connected to the 'D' pin. The observant code reader might already have noted that this takes place in the dice.ino code. Yes we need code modifications for this hack to happen since we are not doing it inside the MCU with dedicated hardware as in the first figure.

When the data has been sampled we wait for the output pin to go back to its original state (1) to avoid glitches, this is when the sampler should read in the data. Given a low enough frequency this should work. In this example we run the I at a really low freq at 10 Hz which should be low enough for most samplers. A person with amazing abilities might even be able to write this down with paper and pen given a synced light at 10 Hz...

After flipping the ck from the MPU we also should read the q output of the flipflop into the MPU for actual use by the code. This method should assert that they both get the same value.



For many people, CPU stands for "Cost Per Unit"

This kind of mechanism could most likely be implemented without a substantial cost of the CPU's. This is especially true since the sampler itself needs only be added when troubleshooting the system. Suppose you have a batch of 10000 toasters you might only need to check the three that are actually broken and perhaps a few healthy speciments to compare with. thus minimising the cost for the system, less the debugging machinery itself.

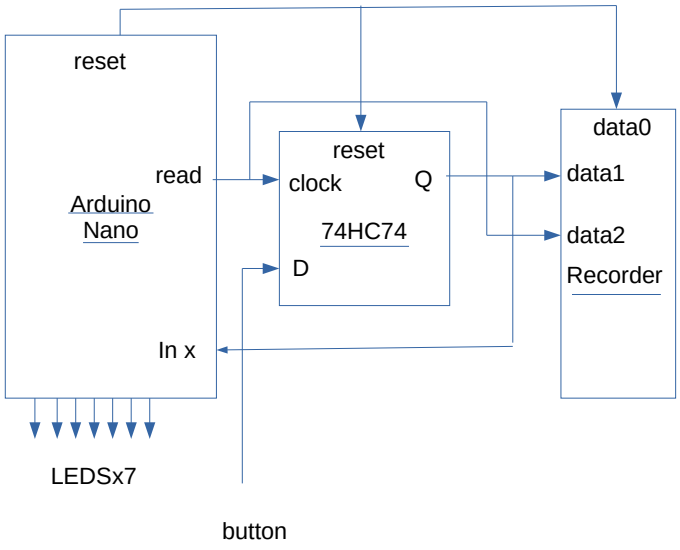
Implementing the full system

In this example we have:

The Arudino nano, a bunch of LED's and a button.

The 74hc74 flipflop previously described. Here I made an adaptor from the soic to 0.1" pin header experiment board. It is also possible to buy very nice ones or simply get a DIP version but I was to impatient to wait for delivery.

A Raspberry Pi Zero to act as the sampler through three gpio pins.



Currently the ck for the flipflop, its "q" output and the reset signal is drawn out from the Arduino. The reset signal is used for the initial synchronization.
The program "gpioread.c" will wait for a press of the reset button, and then sample all data on the clock positive flank from the q signal of the flipflop. (We need a library for raspi called "wiringpi" in Debian). The user can then press the dice button repeatedly, samples will be taken 10 times a second so actually it is quite easy to cheat by a quick press. But what we are aiming at is finding the "strange" bug that gives us all seven LED's lit. So keep pressing until this happens, on average this requires seven presses. After this you might want to press reset a second time, this will instruct 'gpioread' that we are finished and it will terminate. Or use ctrl-C ayor..

The Pi also has been equiped with a USB hub for keyboard, mouse and ethernet. One connector goes to the Arduino nano and the whole Arduino gui is installed on the Pi together with a working gcc and gdb for its arm platform (This should work just as well with a PC but the data acquisition would require some special circuitry).

After this pattern occurs we can now press the arduinos reset button again and the gpioread program should terminate, leaving us an "in.txt" file containing all the 1's and 0's read from the button.

Now comes the fun part, we compile the "main.c" program including dice.ino and start simulating it. Running this program will take the in.txt and each time the simulation asks for an input value one will be provided from the in.txt file!

So we will see the dice patterns scroll by and every now and then we will see the "full" dice pattern of the "illegal seven" roll by. Remember however that it is only when the dice is still, i.e. when the button is not

pressed that a stable "seven" pattern is observed. It is left to the reader to set a breakpoint and "find" this bug via the debugger. Loads of printf's can also be added to the code since we don't need cycle-exact emulation, we only need to sync with the input data feed from in.txt.

Future development

If this catches enough attention (and something similar can't already be bought at Adafruit for $5) I'll make an attempt at something a whole lot more tricky, capturing interrupts and replaying! This will involve:

- Cycle exact emulator for Atmel Avr's (*emulare* is being considered but also a python variety).
- CPLD, such as xc9536 or coolrunner xc2s series, this will shape interrupts enough to be grabbed by data capture device.
- Most likely the raspi will be used for capture but a mechanism will be added so we get a detection if data is missing due to too high data rates.