21:06 Reversing the Fairplay 710 Baseball Scoreboard

by EVM

The local baseball league where my kids play has some old electronic FairPlay 710 scoreboards that needed rehabilitation. FairPlay is a line of scoreboards made since 1975 by the Fairtron Corporation, which is still around in some fashion. The boards in our league date to 1992 and have become disused because of the way they were originally wired. At a league meeting over the summer, somebody asked what it would take to make them WiFi controlled. In this article, I'll walk through my RE process and my WiFi controller implementation, in the off chance that any of you fine neighbors want to rig up something similar.

At installation the boards were wired up to 110V AC power and a low voltage signal line. The processor box inside the board takes a 1/4 inch audio cable, and controls the bulbs. The board is comprised of standard E26/A19 bulb sockets (for the Ball/Strike/Out/Hit/Error lights) and E26/A15 bulb sockets (for all of the digit displays). The signal line is usually terminated indoors with a corresponding audio jack. The controller box can then be plugged into the jack to control the board.

On the two primary fields of play these jacks were put in the top level of a snack shed—a sort of scorekeeping booth. The problem is that no parent wants to be banished to the booth, so the boards don't get used. We wanted to make it so that parents could easily operate the boards from their phone, sitting comfortably in the bleachers.

The Controller

Since it was going to be logistically difficult to haul an oscilloscope out to the field, I decided to attack the controller box. After popping open the case I saw a beautiful little old lady of a board, featuring a Motorola 68HC11 and a 128K EPROM. Normally I would be all about popping that EPROM into a reader and dropping the image into a disassembler, but I figured that would be a long path to getting results, since the signal was probably pretty straightforward. And like a batting practice fastball, it sure was.

Stealing Signals (like an Astro)

I could easily see the red (signal) wire hooked up to the "ring" part of the 1/4 inch audio jack, and the black (ground) wire hooked up to the "tip" part. When I clipped my oscilloscope probes onto these parts of the connector, I could immediately see the data pulse train output by the controller and the encoding was very clear. (Figure 5.)

```
 _
/ \
\ / <- Tip: Signal ground
/_\
|_| <- Ring: Signal
| |
| | <- Sleeve: Chassis ground
```

The FairPlay signal uses RS-232 signal levels (±5V), but uses a proprietary protocol. In RS-232, each bit takes the same amount of time, with a 1 being a logical high (+5V) and a 0 being a logical low (−5V). The length of each bit is determined by chosen baud rate. In the FairPlay protocol each symbol contains both a high part and low part, and the difference between a 1 and a 0 is the length of the high part. Each symbol is 30 microseconds long,

Figure 2: FairPlay 710 scoreboard and internal processor box.

Figure 3: FairPlay controller buttons and label.

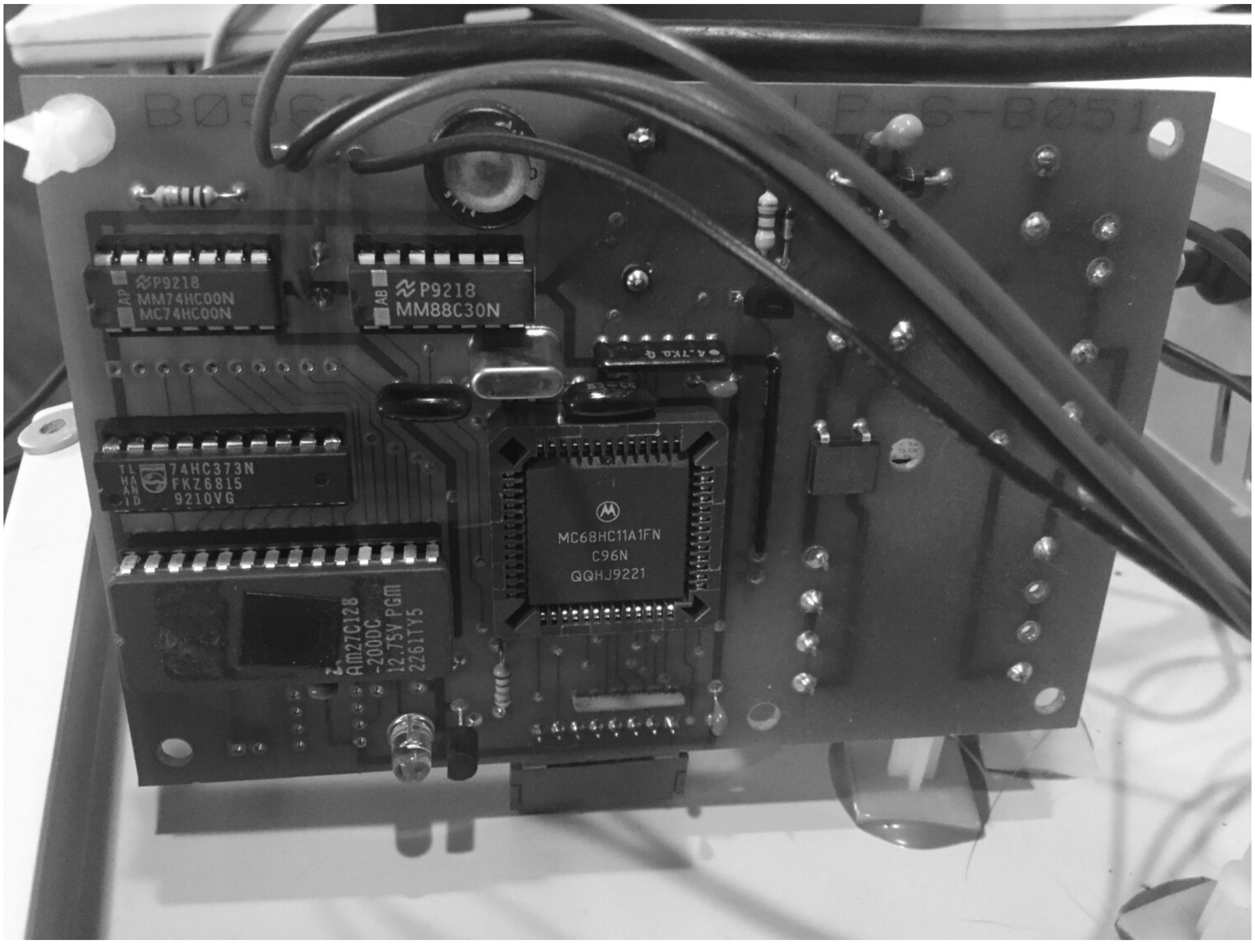
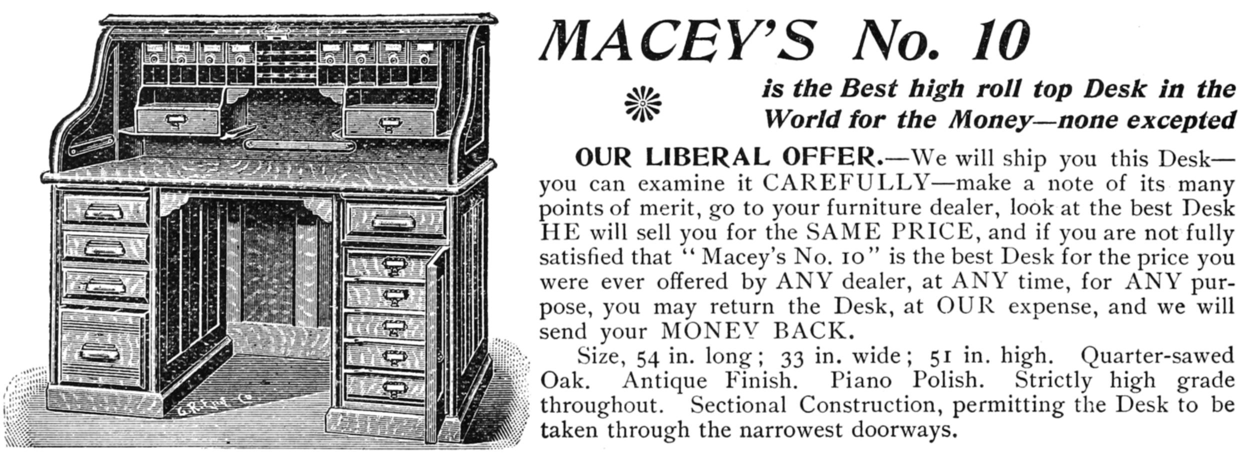


Figure 4: FairPlay BA41A controller board.



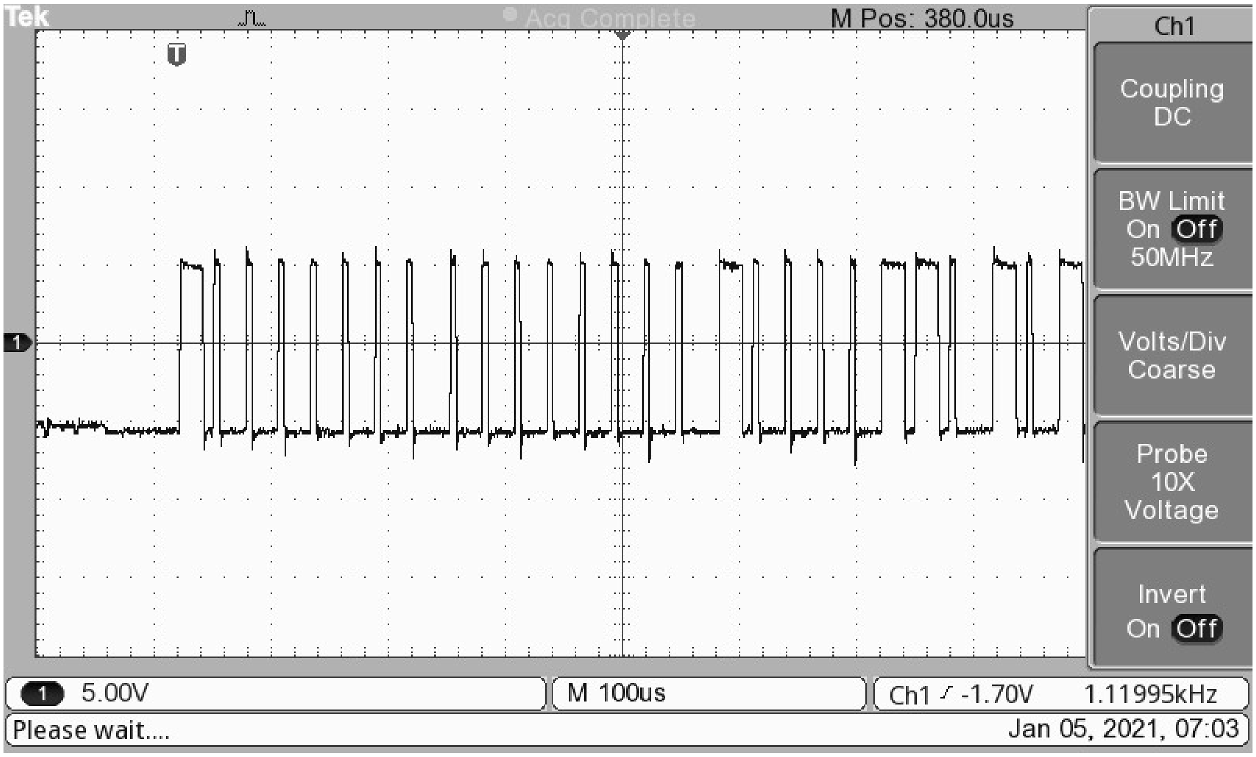


Figure 5: Scope capture of FairPlay signal.

a 0 symbol is 5 microseconds high and 25 microseconds low, a 1 symbol is 20 microseconds high and 10 microseconds low. The messages go from controller to board, there is no path for a response from the scoreboard.

This particular model uses a 56 bit message word that is repeated every 50 milliseconds. (See Figure 6.) I determined the fields by pressing controller buttons while it was hooked up to the oscilloscope and watching which bits change. For the digit encoding I cycled through all possibilities once I had a proof-of-concept implementation running on a Raspberry Pi. See Figure 7 for an explanation of the bitfields in the message. Notice that this is how it is transmitted on the wire.

Overkill: The Correct Amount of Kill

You might be doing the math in your head and thinking that there are precisely zero things that happen in a baseball game that require 50ms timing in a scoreboard. But I think it's likely that this same protocol is used in FairPlay scoreboards for sports like basketball or hockey that have a game clock. (Such a clock needs to be accurate to tenths of seconds.) My guess is that other FairPlay boards of similar vintage for other sports probably use the same encoding and timing, with different message words.

You might expect a protocol like this to have the controller transmit numerical values and then the scoreboard would figure out which bulbs to turn on, but it doesn't work that way. For the Ball, Strike, and Out fields, each bulb directly maps to bits in the message. The score and inning digits are controlled by a single byte in the message, but each digit is made up of 13 bulbs. This means not every bulb can be directly controlled. Nor does it work like a seven-segment display.

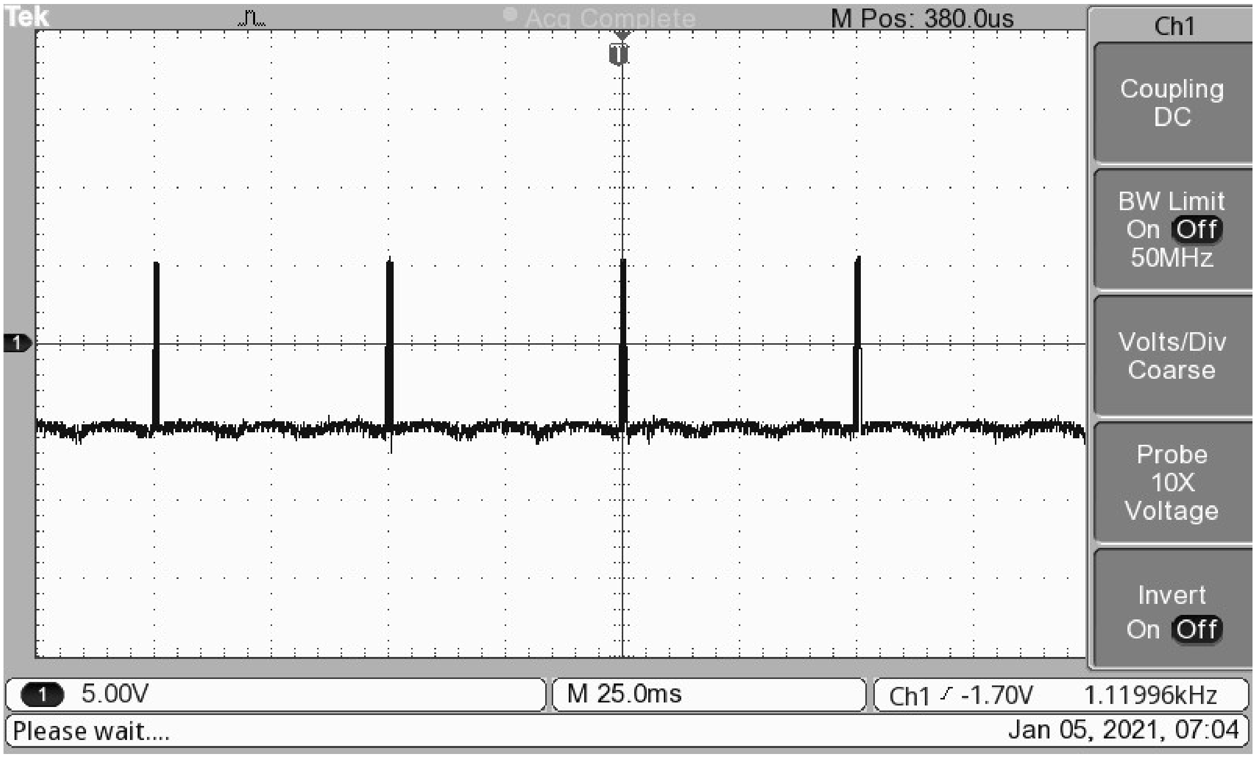


Figure 6: Repeating messages in FairPlay protocol.

```
Byte #     Bitfield                        Key
0          | br x  x  x  e     it(3)  |    br — bright (1) / dim (0)
1          | h  o  o  st st b  b  b  |    h  — hit    e — error
2          | inning ones digit        |    it — inning tens digit (3 bits)
3          | home ones digit          |    x  — unused
4          | home tens digit          |    o  — out
5          | guest ones digit         |    st — strike
6          | guest tens digit         |    b  — ball
```

Figure 7: Bitfields in the FairPlay protocol.

The bulbs map to the 8 bits of the byte in the following format:

```
007
5 1
567
4 2
337
```

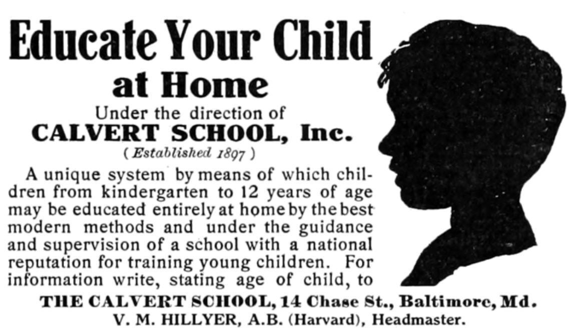
For instance you can render the digit 3 in two ways, with either the pattern `0x4F` `0xCF`.

```
XX                  XXX
  X                   X
 X    or like this:  XX
  X                   X
XX                  XXX
```

Here are the mappings for every digit:

```c
unsigned char pattern[] = {
  // 0     1     2     3     4
  0xBF, 0x86, 0xDB, 0x4F, 0xE6,
  // 5     6     7     8     9
  0xED, 0xFD, 0x87, 0xFF, 0xEF
};
```

To fully implement the WiFi control, I hooked up a Raspberry Pi Zero to the new Pi Pico board via UART and then I have a Pi Pico GPIO output hooked up to a MAX3232. (Thanks to good neighbor Goodspeed for that tip.) I have the Pi serve up a pretty simple PHP script that writes the current settings to a file, and a little server program that converts these settings into the proper 56-bit message word. The Pico program just reads the current 56-bit message and generates the signal which is converted to $\pm$5V by the MAX3232. Code is available, of course.[7]



[7] `git clone https://github.com/evm-apl/FairPlay || unzip pocorgtfo21.pdf FairPlay.zip`