**RASPBERRY PI as a DMX Show Controller**

**By Adela Baciu, Radu Gheorman,**

**Nuri Cingillioglu and Kaiwen Song,**

**First year students at Imperial College London**

After discussion and research made for our extension on the first year’s ARM project, we chose the idea of using the Pi as a controller to lighting systems. We did choose this idea knowing that it has the potential to amaze and grow with commercial significance (one of the main factors that we believe it is that no one ever used a Raspberry PI as a DMX Show Controller).

1. **Overall Description**

The industrial standard to controlling light systems involves using the DMX-512 protocol, which is a specified way of transmitting serial data. The protocol states that data are transmitted in DMX packets, which consist of padding around a byte of data, which is sent to devices connected in a serial fashion. A DMX packet consists of a break signal, which is a series of 0s, lasting minimum 88 microseconds, followed by 2 Mark after Break bits at 1, a start bit at 0, 8 bits data then 2 stop bits at 0.

The challenge for us was to use the Pi to simulate a controller that was able to generate outputs in this particular format. We needed to be able to read the data byte by byte from each memory allocation, adding the required padding to the data obtained, before finally outputting the data in a serial fashion. What was more challenging was the fact that we would have to do all of this in a timed fashion, as the protocol only accepts signals of valid time frames. This meant that we had to find a way to output serial data at the speed, which we state, to the Pi.

We used our own assembler as the operating system for the Pi, by writing an entire assembly file consisting of instructions, stating the behaviours of the Pi. In the assembly file, the output pin was set then cleared, and we manually written data into memory addressed and polled from it. If our design works, then hopefully these instructions would enable the GPIO pins stated in the file to be outputting serial, industrial standard data.

1. **Testing**

**2.1) USB DMX Reader**

Luckily, we had a reliable method that we could use to test our implementations. We had access to a DMX controller, which was an industrial standard device, which when supplied with serial data stream, along with its negated stream, which we will refer to as data minus, and a ground, was able to determine the validity and the contents of the signal transmitted, with the aid of an accompanying software. This device was also helpful in the respect that it was able to give us industrial standard outputs when specified with data, hence we could use such outputs to compare against our own to debug.

The DMX Reader is the software accompanied our DMX controller. Through the DMX Reader we are able to read in the inputs from the Pi and display it on a computer. If everything works, the DMX Reader will be able to decode the serial input stream and obtain the data transferred to it by removing the padding and display the decimal number represented by the 8 bits data.

**2.2) Oscilloscope**

The DMX Reader and controller enabled us to test whether our design was able to output industrial standard data, however, if the format of our output was wrong, then testing it on the DMX will only produce junk data outputs that did not help us. So we wanted to find a remote and portable way to test our output. Under the suggestion of the CSG(the Computing Service Support Department), we obtained ourselves a working oscilloscope. The idea behind the oscilloscope was that it was able to output visual representation of the data being transmitted by the pins. The oscilloscope helped us to analyse the data signal, to ensure that we had all the padding bits surrounding the data bits, and also to check our timing against the standard outputted by the DMX controller by reading the wavelength of outputs of the Pi.

1. **Implementation**

**3.1) Manually turning on and off the pins**

The initial attempt at outputting a DMX signal was to manually turn on and off GPIO pins, as it would have been quite similar to flashing a LED on the Pi. However, this approach bared significant problems, one of which was timing. After trying a few times to turn on and off the pins 14 and 15 to simulate data plus and data minus, we realised it was impossible to achieve 4 microsecond precision using this method.

**3.2) Enabled the UART to output data**

Going over the Broadcom Peripherals spec, we discovered that there was a built-in serial output function, UART at GPIO pin 14. Though the specification was quite vague on how to the use this function, we managed to output the 8-bit data with 1 start bit and 2 stop bits. However, again the timing was off and the DMX protocol required a break.

**3.3) Using an inverter to simulate the data**

After confirming that we were outputting the correct data using an oscilloscope, we tried to simulate data minus as the standard on which DMX is based, RS-485, uses differential signalling. After double-checking that the UART inside the Broadcom 2835 chip only provided a single output without any built-in data minus option, we sought an inverter to invert the signal. Our concern was if the delay of the inverter would corrupt the data that was being sent from the original data plus line. Fortunately, the delay was negligible.

**3.4) Simulating break manually by disabling pin 14**

One of the most challenging part of outputting a valid DMX signal was simulating the break. The UART of the Pi outputted logical high when on idle and then the data. However, the protocol required a low signal for more than 88 microseconds to signal the packet. Our initial attempt was to turn off the GPIO output pin such that it will run low for some period of time and then turn back on to output the data. The idea failed quickly as we realised that the pin was dependent on the UART and continued to output high.

**3.5) Using the multiplexer to simulate breaks**

In this attempt to simulate the break, we incorporated a multiplexer to flip the signals of idle and output low for a period of time. We used GPIO 8 as a control signal to the multiplexer. However, the multiplexer did not behave as expect it to be, introducing delays when switching between signals. The multiplexer together with the GPIO 8 made timing 88 microseconds incredibly difficult.

**3.6) Cross referencing with the original output of the DMX**

Our biggest progress happened when we analysed a proper DMX signal from the USB-DMX device using our oscilloscope. We noticed several crucial points, first of which was that our timing with the current baud rate was not at 4 microseconds and that the oscilloscope had not been calibrated. The initial UART clock of the Pi was at 3 MHz, which was insufficient to output DMX. Furthermore, we discovered that the break was not visible in the oscilloscope.

**3.7) Setting the UART clock and slow down the baud rate to simulate break**

We manually increased the UART clock to 4 MHz, which gave an exact baud rate of 250k required by the DMX protocol. We confirmed this by checking out bit lengths in the oscilloscope with respect to the proper output we recorded in the previous step. Since we had control of the output speed, we simulated the break by slowing the baud rate and outputting data 0 from UART. The slowing down ratio was 2.5 to fit the 8-bit data, 1 start bit, 2 stop bits in the range of 88 to 176 microseconds. This method not only outputted consistent breaks, but also maintained a smooth stream of data, as the UART was not manually turned off.

**3.8) Final Product**

Our final product involved an assembly program, which was able to output data stored as binary inside the kernel image itself. The output data is divided into scenes, which usually described what each light should do. We put this data inside a text file and wrote a helper program, which converted into a binary file. Then we wrote another helper program, which concatenated the assembly program binary and the data binary such that the assembly program would access the data from the memory locations. There was a slight problem as we were not able to access the memory locations of the data directly as we thought the kernel image started at address 0x0. It turned out the kernel image start address was 0x8000 (which we had to search for online). After resolving addressing issues, the final assembly program was able to output a set of data, defined as a scene, for a certain amount of delay specified inside the show file and then move onto the next scene looping forever.

Finally, a demonstration of our project can be found at: https://www.youtube.com/watch?v=ou4OAfCO0nw&feature=youtu.be

