DECODER FIRMWARE

The firmware was written in C, and compiled with Microchip's free XC8 compiler. It's a fairly trivial thousand-line affair, and fits easily into the PIC.

Frankly there's little more worth saying about the code: it's that straightforward. Most of the I/O is handled by an interrupt routine running at about 2kHz and hung off TMR2. All the inputs are polled in the same handler, so we don't need to worry about a flood of interrupts being generated by bouncing switches.

UNI/O support

The only tricky bit was the code which reads the UNI/O memory. In essence it's simple: there's a simple serial protocol to implement, but all we have to is read a small amount of data from the flash chip.

In practice it proved tricky to get the timing right, though the following tricks helped:

- Put the code in macros rather than subroutines.
- Use TMR2 (running at 62.5kHz) to synchronize all the state changes and reads.

Essentially the key idea is to make all the UNI/O changes immediately after a TMR2 tick. We can then run random code, provided that we're finished some time before the next TMR2 tick.

Things were made easier because the code only has to run once, and for a short-time, at startup. So we can use TMR2 as we will, and disable interrupts too.

It's fair to say that the UNI/O code isn't particularly robust. If I were actually deploying it properly I'd make at least three changes:

- The UNI/O state machine doesn't look very hard for errors.
- There's no checksum on reading data from the flash chip.
- I've not checked that the jitter on outgoing bitstream is within tolerance (there's probably scope to reduce jitter with a bit of assembler though).

For testing I've used a 11LC160 16kb chip, of which only the first 208 bits actually matter! You could use any device with address 0xA0 instead e.g. any of the 11xxyyy family.

Configuration

Configuration data are held in a 26-byte array:

```
static uint8_t inbuff[26] = {
  // Start up message: remember that the LSB is on the right `so'
  // this is backwards
  seg_p, seg_p, seg_p, font_O, font_L, font_L, font_E, font_H,
  // North offset: N 51 12.345 => 0x002ee159
  0x59, 0xe1, 0x2e, 0x00,
  // East offset:  E  0  9.876 => 0x00002694
  0x94, 0x26, 0x00, 0x00,
  // Skipjack32 Key
  0x00, 0x99, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11,
};
```

You can see it's possible to change the message displayed on startup, and the two decoder parameters: the coordinate offset and skip32 key. You'll see too that these match the values used in the Haskell skip32 implementation.

At startup, this block might be over-written by data from the UNI/O flash chip. However for this to work SCAN_FLASH must be set at compile time:

```
/ Set this to non-zero to scan the flash on startup
#define SCAN_FLASH (1)
```

Source: http://www.mjoldfield.com/atelier/2012/12/coord-decoder.html