TinyWatch Project

An AtTiny85 OLED Wearable

Goal + Background

Honestly, this project is just an excuse for me to stuff an attiny85 in more places it shouldn't be. Some of the secondary goals for the project are:

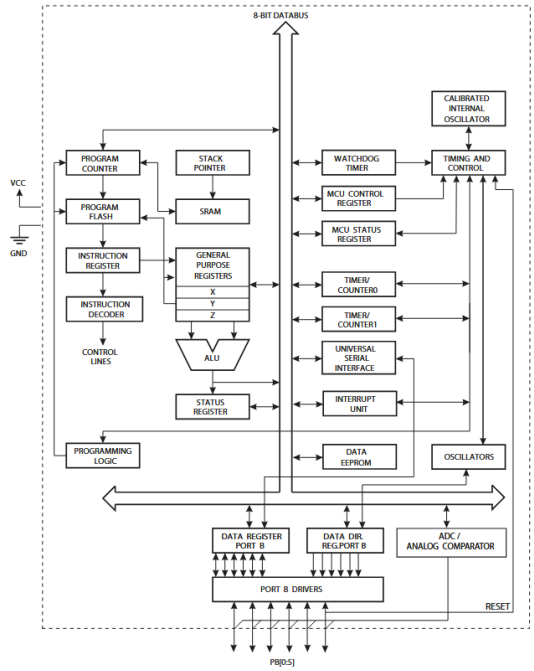
- Develop a cool wearable
- Use high density OLED display
- RTC chip
- Radio capabilities
- Laser pointer
- Learn

Another key aspect of this is that I had limited resources for quite a while, so keeping to parts I already have is a fun challenge.

This project was something I started in late eighth grade. I soldered together the perfboard and hooked a microcontroller up to a Nokia display with lithium batteries from various tech trash I found. This was a starting point I couldn't ignore, and I continued to be fascinated with embedded graphics and I2C, which seemed like magic to a kid with no EE education whatsoever.

Components

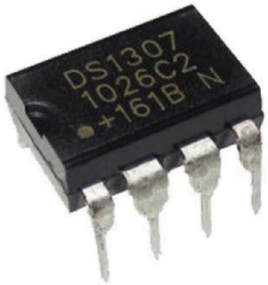
The Attiny85 (in the DIP package) will be used, mainly because ATMEL (now Microchip) provided free samples for anyone with an email. This also implicates an ISP, for which the arduino-as-isp setup will be useful. The chip provides flexibility with input voltage, ranging from around 5.5-2.7 volts until brownout or damage occur. It features a primary and watchdog timer, internal and external interrupts, 5 IO pins, a 8MHz internal clock, and similarities to the Arduino UNO system. And to make things fun, we are entitled to a generous 512 bytes of RAM!



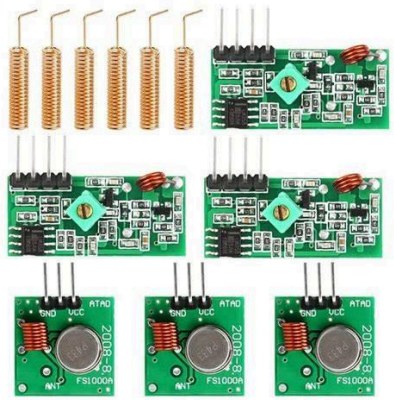
The [SSD1306] was selected as a low cost, widely available breakout with I2C input and various sizes, colors, and configurations. The display has a flushed out feature set with configurable addressing, backlight, indexing, and all that nice stuff. Plenty of pixels to work with (“128 x 64 dot matrix”) too. The datasheet has plenty of information and it is very well communicated.

Yes, IK this is the SPI version pictured

The DS1307 chip is a real time clock that can be accessed via I2C and provides more accuracy than I know what to do with. It can be commonly found on modules that include a small battery cell. For this project I decided to use a PDIP version with an external crystal. It also isolates the battery inputs from the rest of the system, greatly simplifying the power design. Looking back, not my smartest choice (other chips exist that don't use BCD or weird register layouts or some of the latter eccentricities of this primitive RTC).

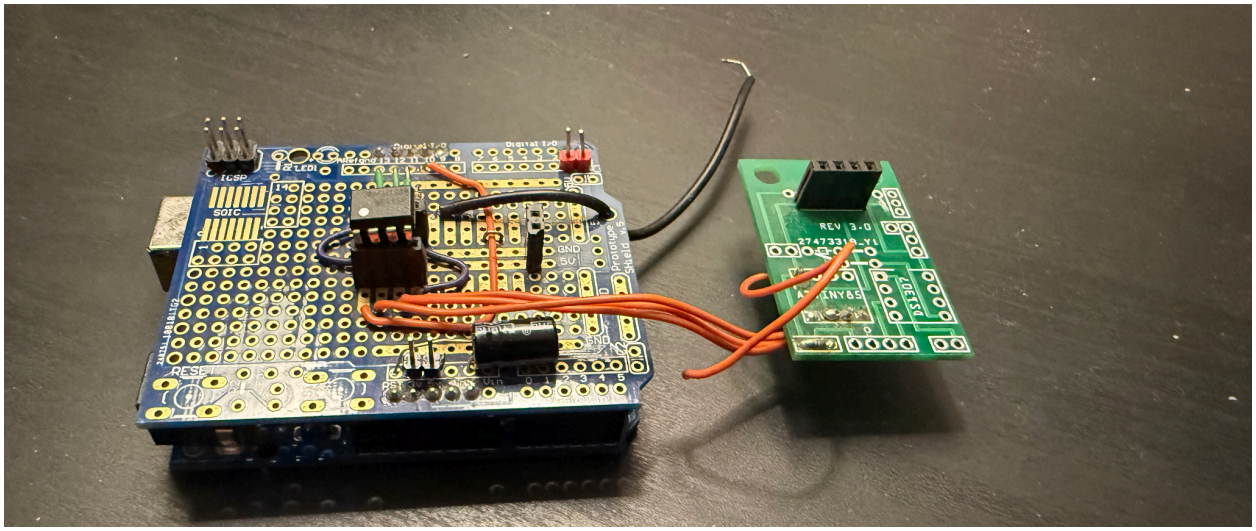


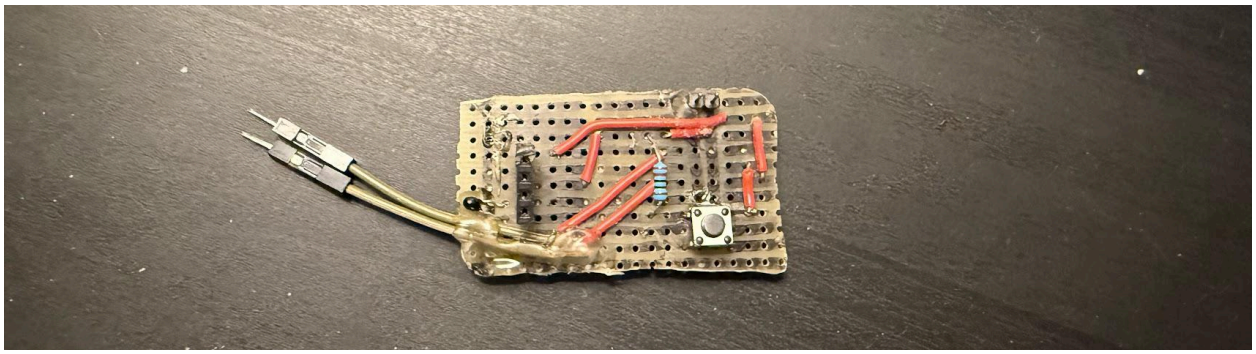
Keeping with the trend of super basic modules, we find ourselves before the 433Mhz radio module. A garage door opener sort of quality, this module is battle tested and FCC compliant. Initially, I just saw this as a cool single pin solution for wireless communication, and I intended for it to be used as a simple input. Nothing crazy like defining symbols and transient waveforms and mathematical transformations like LoRa pieces, but still itches at a curiosity. These can carry UART signals (not without losses and noise) so a lot of simple schemes can be used, like a checksum or hamming code or just a parity bit.



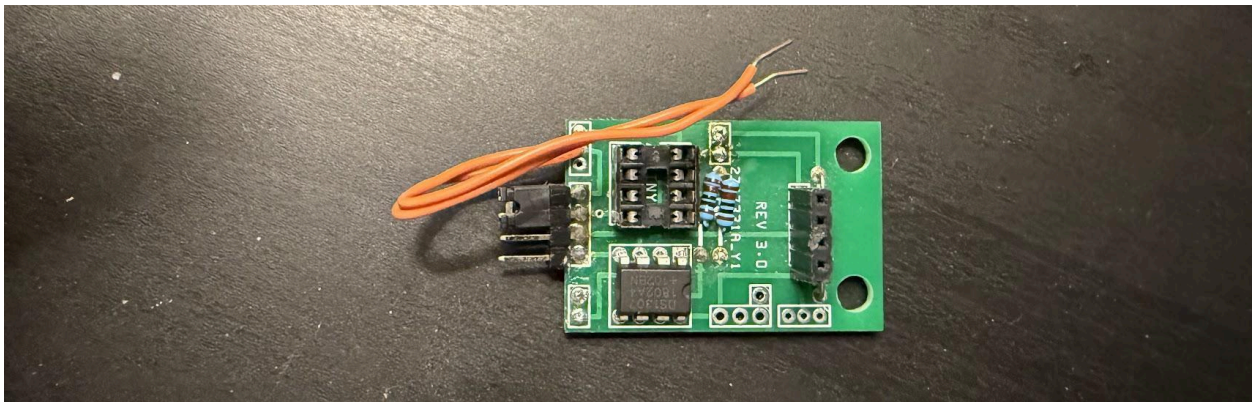
Early Prototypes

The first steps in development involved crude assembly of components and an equally crude ISP. Here's the gore:





So these only really included the display and IC. Looking at the Arduino-as-ISP setup, I used a basic shield PCB with some headers to program removable ICs. I would be a more sane man if I knew about ZIF sockets when I made this. Moving on. Next I made a simple PCB in KiCad (The gerber/schematic files have been lost to time):



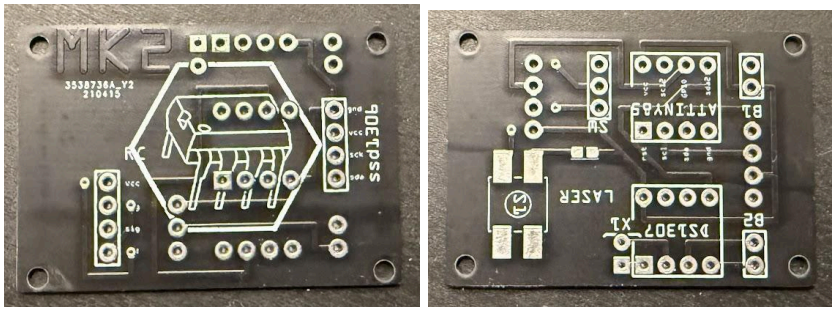
So some major improvements here!

- Battery hookups for separate batteries (coin cell and LiPo)
- DS1307 PDIP package soldered in
- Pulldown resistors

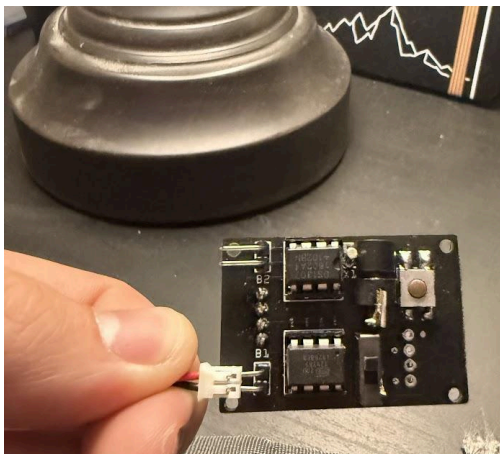
Still very crude, I was able to implement some basic watch functions though. The ISP was retrofitted with a copy of the PCB so that removal of the microcontroller from the ISP wasn't necessary to test out I2C and graphics. Another feature I experimented with was a capacitive touch input. Using the internal ADC on the microcontroller it was possible to get values and interpret them using a 1M Ohm resistor and measuring rise time. I later changed this to use a digital pin, which had a trigger threshold. Capacitive touch is pretty simple and elegant when you mess around with it enough. See my github where I published a [capacitive touch routine in AVR ASM].

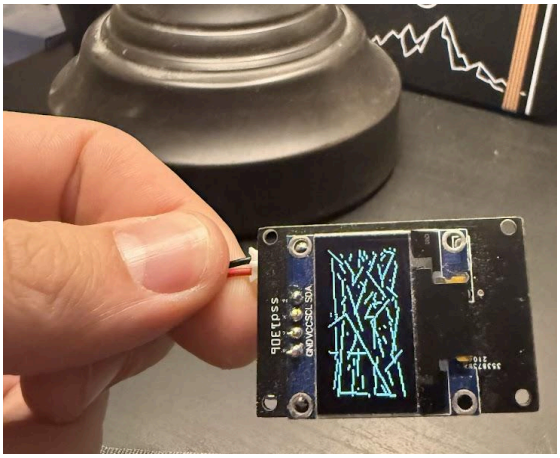
Final Hardware

After my first go around with PCB development, I started to have some fun. I made a cool silkscreen image, text underneath the black soldermask, surface mount components, decent mounting points, the list goes on. Additionally the display was moved to the side opposite the MCU and RTC. This version integrates a pushbutton, sliding on switch for the MCU, and laser pointer. This was made in sophomore (?) year of high school.

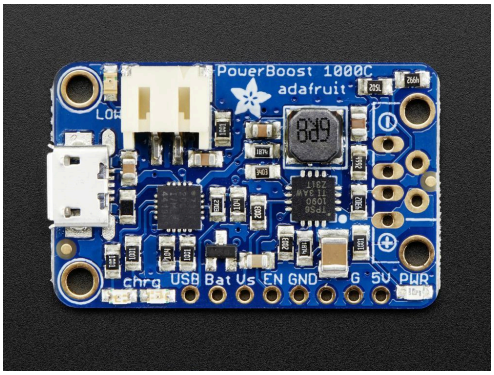


So the fully assembled version looks like this.

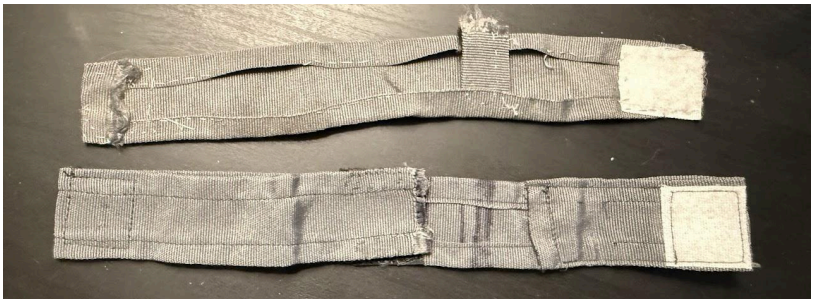




So I guess the obvious question is "Where's the BMS?". Well battery management was done with an adafruit Powerboost 1000c. It has a charge controller, standard battery connector, and boost converter to bring the system to 5 volts which is then regrettably converted to 3.3V with an LDO.



A big theme of the project is picking up whatever skills I needed to get to my goal, so I also got on a sewing machine and made a crude and ugly (but functional) band to hold everything.



Firmware

There's a lot going on here. First, the obvious issue, the microcontroller literally doesn't support I2C. This was honestly pretty simple. In early builds I just bit-banged it by studying the protocol and writing C functions. I just ran this at the 1Mhz microcontroller clock speed, which the peripherals were able to handle (ssd1306 specifies ~400kHz max, and capacitance was likely the limiting factor)

```
//special delay is inline asm nops
//digital_write is asm writing to PORTB

void i2c::write(uint8_t data) {
  uint8_t i;
  for ( i = 8; i > 0; i-- ) {
    if ( data & 0x80 ) {
      DIGITAL_WRITE_HIGH(DDR_REG, PORT_REG, m_sda)
    } else {
      DIGITAL_WRITE_LOW(DDR_REG, PORT_REG, m_sda);
    }
    data <<= 1;
    special_delay(I2C_RISE_TIME);
    DIGITAL_WRITE_HIGH(DDR_REG, PORT_REG, m_scl);
    special_delay(I2C_HALF_CLOCK);

    DIGITAL_WRITE_LOW(DDR_REG, PORT_REG, m_scl);
    special_delay(I2C_HALF_CLOCK);
  }
  DIGITAL_WRITE_HIGH(DDR_REG, PORT_REG, m_sda);
  special_delay(I2C_RISE_TIME);
  DIGITAL_WRITE_HIGH(DDR_REG, PORT_REG, m_scl);
  special_delay(I2C_HALF_CLOCK);
  DIGITAL_WRITE_LOW(DDR_REG, PORT_REG, m_scl);
  special_delay(I2C_HALF_CLOCK);
}
```

A big improvement was switching to the TinyWireM library (shoutout adafruit + arduino forums). No more DIY bitbanging. So the next step is developing a hardware abstraction for the display. One problem. The microcontroller does not have enough RAM for a display buffer (128x64 pixels << 512 bytes). This creates issues where put_pixel type calls will overwrite the whole line, because we can only write one byte as per the addressing scheme (in the common configuration 8 pixel vertical lines are drawn from top left, across, and then when the row is done it moves down). So we use one byte of our precious RAM to cache pixel

writes and XOR data onto that line. Here's a full implementation of the SSD1306 library I made. The main reason I made my own library was to enable drawing circles, which no other library is capable of without a frame buffer.

```
#include "ssd1306.h"

#define wire TinyWireM

namespace {
  //2 bytes static ram usage
  void printc(char c, const uint8_t *font){
    if(font == 0) return;

    static uint8_t cursor_x = 0;
    static uint8_t cursor_y = 0;
    uint8_t start_offset = font[3];
    uint8_t width = font[1];

    ssd1306::cursor(cursor_x, (cursor_y % 8) << 3);

    cursor_x += width;

    if(cursor_x > 128 - width){
      cursor_x = cursor_x % 128;
      cursor_y++;
    }

    if(c == '\n'){
      cursor_x = 0;
      cursor_y++;
      return;
    }

    if(c == 0){
      cursor_x = 0;
      cursor_y = 0;
      return;
    }

    for(int j = 0; j < width; j++){
      uint16_t cur_offset = 4 + width * ((uint8_t)c - start_offset);
      ssd1306::data(pgm_read_byte(&font[cur_offset + j]));
    }
  }
  void cmd(const uint8_t *cmds, uint8_t n){
    wire.beginTransmission(SSD1306);
```

```
    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);
    for(uint8_t i = 0; i < n; i++){
      uint8_t cur = pgm_read_byte(&cmds[i]);
      wire.write(cur);
    }

    wire.endTransmission();
  }
  const PROGMEM uint8_t load_code[] = {
    OLED_CMD_DISPLAY_OFF,
    OLED_CMD_SET_DISPLAY_OFFSET,
    0x00,
    OLED_CMD_SET_DISPLAY_START_LINE,
    0x00,
    OLED_CMD_SET_CONTRAST,
    0x7F,
    0xa1,
    0xc8,
    OLED_CMD_DISPLAY_RAM,
    OLED_CMD_DISPLAY_NORMAL,
    OLED_CMD_SET_CHARGE_PUMP,
    0x14,
    OLED_CMD_SET_MEMORY_ADDR_MODE,
    0x00,
    OLED_CMD_DISPLAY_ON,
    0x2E
  };
    const uint8_t lookup_table[] PROGMEM = {
    0x00,
    0x1a,
    0x35,
    0x4e,
    0x67,
    0x7f,
    0x95,
    0xaa,
    0xbd,
    0xce,
    0xdc,
    0xe8,
    0xf2,
    0xf9,
    0xfd,
    0xff
  };
}
namespace ssd1306 {
  void begin(){
    wire.begin();
```

```
    cmd(load_code, 17);
  }

  void cursor(uint8_t x, uint8_t y) {
    wire.beginTransmission(SSD1306);
    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);
    wire.write(OLED_CMD_SET_COLUMN_RANGE);
    wire.write(x);
    wire.write(0x7F);
    wire.write(OLED_CMD_SET_PAGE_RANGE);
    wire.write(y >> 3);
    wire.write(0x7);
    wire.endTransmission();
  }

  void data(uint8_t d) {
    wire.beginTransmission(SSD1306);
    wire.write(OLED_CONTROL_BYTE_DATA_STREAM);
    wire.write(d);
    wire.endTransmission();
  }

  void clear() {
    cursor(0,0);
    for (uint16_t i = 0; i < 64; i++) {
      wire.beginTransmission(SSD1306);
      wire.write(OLED_CONTROL_BYTE_DATA_STREAM);
      for (uint8_t x = 0; x < 16; x++) wire.write(0x00);
      wire.endTransmission();
    }
  }

  void start_scrolling(uint8_t scroll_dir, uint8_t start_page, uint8_t
framerate, uint8_t end_page){
    /* WHAT THE ARGS MEAN
     * end_page = 0b111
     * start_page = 0b000
     * framerate ? see data sheet. 0b111 is fastest
     * scroll_dir = 0 → right, 1 → left
     */
    wire.beginTransmission(SSD1306);
    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);
    //setup scrolling
    wire.write((scroll_dir & 1) | 0x26);
    wire.write(0x00);
    wire.write(start_page);
    wire.write(framerate);
    wire.write(end_page);
    wire.write(0x00);
```

```
    wire.write(0xFF);
    //start
    wire.write(0x2F);
    wire.endTransmission();
  }

  void invert_display(uint8_t inverse){
    wire.beginTransmission(SSD1306);
    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);
    wire.write((inverse & 1) | OLED_CMD_DISPLAY_NORMAL);
    wire.endTransmission();
  }

  void start_scrolling_v(uint8_t fixed_area, uint8_t scroll_area){
    wire.beginTransmission(SSD1306);
    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);
    //setup scrolling
    wire.write(0xA3);
    wire.write(fixed_area);
    wire.write(scroll_area);
    //start
    wire.write(0x2F);
    wire.endTransmission();
  }

  void stop_scroll(){
    wire.beginTransmission(SSD1306);

    wire.write(OLED_CONTROL_BYTE_CMD_STREAM);

    wire.write(0x2E);
    wire.endTransmission();

    wire.beginTransmission(SSD1306);
    wire.write(0x00);
    wire.endTransmission();

  }

  void pixel(uint8_t x, uint8_t y) {
    if (x ≥ 128 || y ≥ 64) return; //sanity check
    static struct graphics_cache { uint8_t x, y, buf; } cache =
{0,0,0};
    uint8_t newbuf = (1 ≪ (y % 8));
    if ((y ≫ 3 == cache.y ≫ 3) && cache.x == x) {
      cache = {x, y, cache.buf |= newbuf};
    } else {
      cache = {x, y, newbuf};
    }
```

```
  cursor(cache.x, cache.y);
  data(cache.buf);
}

void line(uint8_t x0, uint8_t y0, uint8_t x1, uint8_t y1) {
  int16_t steep = abs(y1 - y0) > abs(x1 - x0);
  uint8_t tmp;
  if (steep) {
    //swap(x0, y0);
    tmp = x0;
    x0 = y0;
    y0 = tmp;
    //swap(x1, y1);
    tmp = y1;
    y1 = x1;
    x1 = tmp;
  }

  if (x0 > x1) {
    //swap(x0, x1);
    tmp = x1;
    x1 = x0;
    x0 = tmp;
    //swap(y0, y1);
    tmp = y1;
    y1 = y0;
    y0 = tmp;
  }

  int16_t dx, dy;
  dx = x1 - x0;
  dy = abs(y1 - y0);

  int16_t err = dx / 2;
  int16_t ystep;

  if (y0 < y1) {
    ystep = 1;
  } else {
    ystep = -1;
  }

  for (; x0 ≤ x1; x0++) {
    if (steep) {
      pixel(y0, x0);
    } else {
      pixel(x0, y0);
    }
    err -= dy;
```

```
    if (err < 0) {
      y0 += ystep;
      err += dx;
    }
  }
}

void circle(uint8_t x0, uint8_t y0, uint8_t r) {
  int16_t f = 1 - r;
  int16_t ddF_x = 1;
  int16_t ddF_y = -2 * r;
  int16_t x = 0;
  int16_t y = r;

  pixel(x0, y0 + r);
  pixel(x0, y0 - r);
  pixel(x0 + r, y0);
  pixel(x0 - r, y0);

  while (x < y) {
    if (f ≥ 0) {
      y--;
      ddF_y += 2;
      f += ddF_y;
    }
    x++;
    ddF_x += 2;
    f += ddF_x;

    pixel(x0 + x, y0 + y);
    pixel(x0 - x, y0 + y);
    pixel(x0 + x, y0 - y);
    pixel(x0 - x, y0 - y);
    pixel(x0 + y, y0 + x);
    pixel(x0 - y, y0 + x);
    pixel(x0 + y, y0 - x);
    pixel(x0 - y, y0 - x);
  }
}


void clock_hand(int x0, int y0, int n, int l){
  uint8_t x , y;
  uint8_t y_l = lookup_table[(16 - 1) - (n % 16)];
  uint8_t x_l = lookup_table[n % 16];
  uint8_t x_a = ((uint16_t)l * (uint16_t)x_l) >> 8;
  uint8_t y_a = ((uint16_t)l * (uint16_t)y_l) >> 8;

  switch(n >> 4){
```

```
      case 0:
          x = x0 - x_a;
          y = y0 - y_a;
        break;
      case 1:
          x = x0 - y_a;
          y = y0 + x_a;
        break;
      case 2:
          x = x0 + x_a;
          y = y0 + y_a;
        break;
      case 3:
          x = x0 + y_a;
          y = y0 - x_a;
        break;
      }
    line(x0, y0, x, y);
  }

  void clock(int h, int m, int s){
    const int x0 = 96;
    const int y0 = 32;
    const int r =  20;
    const int h_l = 10;
    const int m_l = 13;
    const int s_l = 17;
    circle(x0, y0, r);
    clock_hand(x0, y0, h * 5, h_l);
    clock_hand(x0, y0, m, m_l);
    clock_hand(x0, y0, s, s_l);
  }

  void print(const char *s, const uint8_t *font){
    for(const char *c = s; *c != '\0'; c++)
      printc(*c, font);
  }
  void print_flash(const char *s, const uint8_t *font){
    for(const char *c = s; pgm_read_byte(c) != '\0'; c++)
      printc(pgm_read_byte(c), font);
  }
}
```

The attentive reader will notice a strange nondescript lookup table. This is because these microcontrollers take absolutely forever (400-800 cycles IIRC) to compute sine and cos. I decided to just hardcode some values for each of the 60 positions of a clock hand, which is simplified

by the fact that it is symmetrical horizontally (now needing 30 values), and also symmetrical vertically (we can cut that down to 15 positions). I avoid floating points and doubles by premultiplying the values, which results in “good enough” precision for the effect of a clock hand. Another optimization is storing certain strings and data in flash memory. There are special macros used to access this.I should have mentioned earlier, but the damellis attiny package is being used in conjunction with the arduino IDE to provide critical hardware abstractions that are familiar between AVR chips. Why are we using a namespace and not a class??? Less dynamic memory usage. Above photos of the final prototype vaguely show a demo that involves drawing lines between random points on the screen, primarily to test the pixel drawing caching.

How do we set the time if there's no buttons? Well the RTC will report if the clock is configured and running, so we can tell when it needs to be set. Additionally we have persistent memory between microcontroller power loss in the form of EEPROM. Last, our toolchain defines a **__time__** macro to get the compilation time of the code. Given that there is only a few seconds of upload time, this could be used to set the time if it is the first run of the microcontroller and the clock is not set. Heres what that code looks like.

```
static bool firstRun(){ //works, 10us on write
  for(byte i = 0;i<8;i++){
    if(EEPROM.read(i) ≠ (__TIME__)[i]){
      for(byte j = i; j < 8;j++) EEPROM.write(j,(__TIME__)[j]);
      return true;
    }
  }
  return false;
}
```

An undeveloped feature I dreamt up was synchronizing the clock using UART over 433Mhz radio communication, but the microcontroller just does not have the capability to pull this off without better technology for me to debug it.

All said and done, when a fully functional UI is implemented the flash memory is completely full. Strings and a font take up a majority of the ROM.

Reflections

If another iteration were to be made, significant changes would have to be made.

- Use QFN or at least TSSOP packages for all components
- Onboard BMS (maybe the TP4056?)
    - Thermal runaway protection
    - USB-C capable
    - Robust
    - Common and documented
- Direct soldered display, no breakout
- More capable IC
    - Actual I2C support
- RTOS
- Less archaic RTC
- Pogo pin ISP header
- 4 Layer PCB

I mean the list goes on and on. I think I learned a ton from this project, and even more about what not to do. The most enjoyable part was using the watch and designing the embedded code. The hardware constraints were a good challenge and I feel that a quality outcome was achieved, because I satisfied my design constraints and was able to render a clock. This was a great exploration in low tech debugging, hardware design (badly), firmware, and exploring datasheets. As an older project, the documentation has been created well after the development process so unfortunately a lot has been lost to time. Hope my process and thoughts are worth something!

```
ssd1306::clock_hand(63, 31,tinyrtc.second(),30);
ssd1306::clock_hand(63, 31,tinyrtc.minute(),22);
ssd1306::clock_hand(63, 31,5 * tinyrtc.hour(),15);
delay(1000);
//lmao
```

-no5up