

DIY Insulin Pump Code Manual

This manual provides a detailed explanation of the code for a DIY insulin pump. It includes information on the functionality of each component and how the code works.

IINTS Project Overview

IINTS (Insulin Is Not The Solution) is an open-source insulin pump project designed with affordability and accessibility in mind. It is built using MicroPython on a Raspberry Pi Pico and controls insulin delivery via stepper motors and a user-friendly interface.

Key Features:

- Customizable
- Affordable
- Open-source
- Made for everyone

Project History

IINTS was founded in 2024 as a personal project to learn more about insulin pumps and their mechanics. As someone with 13 years of experience living with diabetes, I was curious about how these devices work and wanted to build one myself. While not intended as a commercial alternative, I share this project as an open-source learning experience for anyone interested in electronics, programming, and medical technology.

Major Milestones:

- May 2024: First working prototype with Raspberry Pi Pico
- July 2024: OLED screen and user interface added
- December 2024: Introduced microstepping for improved precision
- April 2025: Presentation at Coolest Projects Belgium

Special Thanks

A huge thank you to the coaches from CoderDojo Genk and Hasselt for their incredible guidance and support throughout this project. Their mentorship has been invaluable in bringing this idea to life!

Project Features

- **Stepper Motor Control:** Precise insulin delivery using stepper motors
- **User-Friendly Interface:** Buttons and display for simple operation
- **Safety Mechanisms:** Basic fail-safes for reliable functioning
- **Customizable:** Adjust dosage and settings as needed
- **3D Printable:** Open-source STL files for hardware
- **Open Source:** Licensed under MIT, free for personal and medical research

Hardware Requirements

Core Hardware:

- Raspberry Pi Pico (RP2040)
- OLED/TFT display (for UI)
- Stepper motor + Driver (e.g., A4988, ULN2003)
- Push buttons (for input)
- Battery / Power supply

Insulin Pump Mechanism:

- Syringe pump setup (or peristaltic pump)
- 3D-printed mounts (STL files included!)

3D Print Files

All required 3D-printable parts can be found in the `/stl` folder.

- **Download STL files:** [STL Folder](link to STL folder)

Recommended Print Settings

- Material: PLA or PETG
- Layer height: 0.2mm
- Infill: 20%
- Supports: Not required
- Bed adhesion: Brim or Skirt

3D Print Timelapse

- Watch the 3D printing process in action! [Timelapse video](link to timelapse video)

Installation Guide

1. Install MicroPython

- Make sure you are using the correct MicroPython version:
 - MicroPython v1.23.0 (2024-06-02) for Raspberry Pi Pico
- **Download here:** [MicroPython Download](#)
- **Check Your MicroPython Version:**
 - Connect to your Raspberry Pi Pico and run:

```
import os
os.uname()
```

2. Install Thonny IDE

- Thonny is recommended for writing and uploading MicroPython scripts.
- **Download:** [Thonny Download](#)
- **Steps:**
 - Open Thonny
 - Select Raspberry Pi Pico as the interpreter
 - Install MicroPython firmware if not already installed

3. Clone This Repository

- Run this command to download the project:

```
git clone https://github.com/python35/IINTS.git
```

- Or manually **Download ZIP** from GitHub

4. Upload to Raspberry Pi Pico

- Connect the Raspberry Pi Pico via USB
- Open Thonny
- Copy `main.py` and other files to the Pico
- Click Run

Configuration

Edit `config.py` to set parameters:

```
INSULIN_RATE = 1.0 # Units per second
STEP_MOTOR_SPEED = 200 # Steps per second
DISPLAY_BRIGHTNESS = 0.8 # 80% brightness
```

Adjust according to your needs.


License

This project is MIT Licensed, meaning you're free to use, modify, and distribute it. However, this is NOT a certified medical device — use responsibly.


Contributions

Want to improve the project? Fork the repository and submit a Pull Request.

Disclaimer

 **Warning:** This project is for educational and research purposes only. It is not an FDA-approved medical device. Always consult a medical professional before using any insulin pump.

Download & Start Building!

 Clone the repository and start experimenting!

1. Libraries and Initialization

```
import machine
import utime
import framebuf
from st7789 import ST7789
```

Libraries:

- `machine` : Provides access to the microcontroller's hardware (in this case, a Pico), such as SPI, pins, etc.
- `utime` : For time functions like `sleep` and `sleep_ms`.
- `framebuf` : For creating a framebuffer, a memory area where you can draw pixels before sending them to the screen.
- `st7789` : A library for the ST7789 TFT LCD screen driver.

Screen Initialization

```
spi = machine.SPI(1, baudrate=40000000, polarity=1, phase=1,
sck=machine.Pin(10), mosi=machine.Pin(11))
dc = machine.Pin(13, machine.Pin.OUT)
reset = machine.Pin(12, machine.Pin.OUT)
bl = machine.Pin(17, machine.Pin.OUT)
display = ST7789(spi, 240, 240, reset=reset, dc=dc)
```

```
display.init()
bl.value(1)
```

- `spi = machine.SPI(1, ...)` : Initializes the SPI interface (Serial Peripheral Interface), used to communicate with the LCD screen. The parameters specify the SPI bus, speed, signal polarity and phase, and the pins used for SCK (Serial Clock) and MOSI (Master Output Slave Input).
- `dc = machine.Pin(13, machine.Pin.OUT)` : Defines the Data/Command pin for the LCD. This pin is used to toggle between sending pixel data and commands.
- `reset = machine.Pin(12, machine.Pin.OUT)` : Defines the reset pin for the LCD. A short signal to this pin resets the screen.
- `bl = machine.Pin(17, machine.Pin.OUT)` : Defines the backlight pin for the LCD. Setting this pin high turns on the backlight.
- `display = ST7789(...)` : Creates an `ST7789` object representing the LCD screen. Parameters include the SPI interface, screen width and height, and the data/command and reset pins.
- `display.init()` : Initializes the LCD screen by sending a sequence of commands to set it up.
- `bl.value(1)` : Turns on the screen backlight.

2. Text Function

```
buffer = bytearray(240 * 240 * 2)
fb = framebuf.FrameBuffer(buffer, 240, 240, framebuf.RGB565)

def draw_text_centered(text, color=0xFFFF):
    fb.fill(0x0000)
    text_width = len(text) * 8
    text_x = (240 - text_width) // 2
    text_y = 100
    fb.text(text, text_x, text_y, color)
    display.blit_buffer(buffer, 0, 0, 240, 240)
```

Framebuffer:

- `buffer = bytearray(240 * 240 * 2)` : Creates a bytearray large enough to hold pixel data for the entire screen (240x240 pixels, 2 bytes per pixel for RGB565 color).
- `fb = framebuf.FrameBuffer(...)` : Creates a `FrameBuffer` object which allows you to draw on the buffer using pixel and shape functions. RGB565 is a 16-bit color format.

Text function:

- `draw_text_centered(text, color=0xFFFF)` : Draws centered text on the screen.
 - `fb.fill(0x0000)` : Clears the framebuffer (fills it with black).
 - `text_width = len(text) * 8` : Calculates the text width (each character is 8 pixels wide).
 - `text_x = (240 - text_width) // 2` : Calculates the X position to center the text horizontally.
 - `text_y = 100` : Sets the vertical position of the text.
 - `fb.text(...)` : Draws the text to the framebuffer with the specified color.
 - `display.blit_buffer(...)` : Sends the framebuffer content to the display.
-

3. Startup Screen Function

```
def show_startup_screen():
    try:
        with open("start.bmp", "rb") as f:
            f.seek(18)
            bmp_width = int.from_bytes(f.read(4), 'little')
            bmp_height = int.from_bytes(f.read(4), 'little')
            f.seek(54) # Start of pixel data
            x_offset = ((240 - bmp_width) // 2) - 40
            y_offset = (240 - bmp_height) // 2
            row_buffer = bytearray(bmp_width * 2)
            display.fill(0x22C5)
            for y in range(bmp_height):
                f.readinto(row_buffer)
                f.seek(54 + (bmp_height - 1 - y) * bmp_width * 2)
                for i in range(0, len(row_buffer), 2):
                    row_buffer[i], row_buffer[i + 1] = row_buffer[i + 1],
row_buffer[i]
                display.blit_buffer(row_buffer, x_offset, y + y_offset,
bmp_width, 1)
                utime.sleep(5)
    except OSError:
        draw_text_centered("BMP not found", color=0xFFFF)
        utime.sleep(3)
        display.fill(0x000000)
```

`show_startup_screen()` : Displays a BMP image on startup.

- `try...except OSError` : Attempts to load the BMP image; if not found, displays an error message.

- Reads BMP metadata (width and height), and pixel data from offset 54.
 - Calculates `x_offset` and `y_offset` to center the image, shifted 40 pixels to the left.
 - Reads each row of the image bottom-to-top (BMP stores pixels upside-down).
 - Swaps byte order from BGR to RGB.
 - `display.fill(...)`: Fills background with a color before showing the image.
 - If the image is not found, shows a centered "BMP not found" message.
-

4. Button Configuration

```
happy_button = machine.Pin(9, machine.Pin.IN, machine.Pin.PULL_UP)
# sad_button is broken, so we ignore it
sad_button = machine.Pin(5, machine.Pin.IN, machine.Pin.PULL_UP)
angry_button = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_UP)

def button_held(button, delay=0.2):
    """Returns True if the button is held for 'delay' seconds."""
    if button.value() == 0:
        utime.sleep(delay)
        if button.value() == 0:
            return True
    return False
```

- **Buttons:**
 - Buttons use `PULL_UP`, meaning they read `1` by default, and `0` when pressed.
 - Only the **happy** and **angry** buttons are functional.
 - `button_held(...)` checks if a button remains pressed for a short time to avoid accidental presses.
-

5. Motor Configuration

```
in1 = machine.Pin(2, machine.Pin.OUT)
in2 = machine.Pin(3, machine.Pin.OUT)
in3 = machine.Pin(4, machine.Pin.OUT)
in4 = machine.Pin(6, machine.Pin.OUT)

def disable_motor():
    in1.value(0)
```

```
in2.value(0)
in3.value(0)
in4.value(0)

disable_motor()
```

- Defines four motor control pins (for a stepper motor).
- `disable_motor()` turns off all outputs to save power and prevent overheating.

6. Stepper Motor Functions

```
full_step_sequence = [
    (1, 0, 1, 0), # Step 1
    (0, 1, 1, 0), # Step 2
    (0, 1, 0, 1), # Step 3
    (1, 0, 0, 1) # Step 4
]

def step_motor(delay, steps, direction=1):
    sequence = full_step_sequence if direction == 1 else
full_step_sequence[::-1]
    for _ in range(steps):
        for step in sequence:
            in1.value(step[0])
            in2.value(step[1])
            in3.value(step[2])
            in4.value(step[3])
            utime.sleep_ms(delay)
        disable_motor()

def continuous_step_motor(delay, steps, direction=1):
    sequence = full_step_sequence if direction == 1 else
full_step_sequence[::-1]
    for _ in range(steps):
        for step in sequence:
            in1.value(step[0])
            in2.value(step[1])
            in3.value(step[2])
            in4.value(step[3])
            utime.sleep_ms(delay)
    # Keeps the motor powered for continuous movement
```


- `full_step_sequence` : Sequence of signals to drive the stepper motor.
 - `step_motor(...)` : Moves a set number of steps, then turns off the motor.
 - `continuous_step_motor(...)` : Used to keep the motor running (e.g., during manual retraction).
-

7. Variables and Value Change

```
totale_dagdosis = 0
gram_koolhydraten = 0
stap = 0

def verander_waarde(waarde, richting, stap_grootte=1):
    if richting == 'up':
        waarde += stap_grootte
    elif richting == 'down' and waarde > 0:
        waarde -= stap_grootte
    return waarde
```

- `totale_dagdosis` : Total daily insulin dose.
 - `gram_koolhydraten` : Amount of carbs to be injected for.
 - `stap` : Tracks the current phase (0 = daily dose, 1 = carbs, 2 = inject).
 - `verander_waarde(...)` : Helper to increase/decrease a value with optional step size.
-

8. Main Program

```
show_startup_screen()

while True:
    # Step 0: Set total daily dose
    if stap == 0:
        draw_text_centered(f"Day dose: {totale_dagdosis}U")
        if button_held(happy_button, delay=0.3):
            totale_dagdosis = verander_waarde(totale_dagdosis, 'up')
            draw_text_centered(f"Day dose: {totale_dagdosis}U")
        if button_held(angry_button, delay=0.3) and totale_dagdosis > 0:
            stap = 1
            utime.sleep(0.3)
```

```

# Step 1: Set carbs in grams
elif stap == 1:
    draw_text_centered(f"Carbs: {gram_koolhydraten}g")
    if button_held(happy_button, delay=0.3):
        gram_koolhydraten = verander_waarde(gram_koolhydraten, 'up')
        draw_text_centered(f"Carbs: {gram_koolhydraten}g")
    if button_held(angry_button, delay=0.3) and gram_koolhydraten > 0:
        stap = 2
        utime.sleep(0.3)

# Step 2: Calculate insulin and inject
elif stap == 2:
    kh_ratio = 500 / totale_dagdosis
    eenheden_insuline = gram_koolhydraten / kh_ratio
    steps_per_revolution = 200
    insuline_per_revolution = 1.0
    steps_per_unit_insuline = steps_per_revolution /
insuline_per_revolution
    total_steps = int(eenheden_insuline * steps_per_unit_insuline)

    step_motor(5, total_steps, direction=-1)
    draw_text_centered(f"Insulin: {eenheden_insuline:.2f}U")
    utime.sleep(2)

    draw_text_centered("Retract?")
    utime.sleep(1)

    while angry_button.value() == 0:
        continuous_step_motor(5, 10, direction=1)
        utime.sleep(0.1)

    disable_motor()
    draw_text_centered("Motor retracted")
    utime.sleep(2)

    totale_dagdosis = 0
    gram_koolhydraten = 0
    stap = 0

```

- **Step 0:** User sets total daily insulin dose.
- **Step 1:** User sets carbs in grams.
- **Step 2:**
 - Calculates required insulin using the 500 rule.
 - Converts insulin units into motor steps.

- Injects insulin (`direction=-1`).
- Asks if retraction is needed.
- While "angry" button is pressed, the motor retracts (`direction=1`).
- Resets variables and returns to step 0.

Scientific Explanation of Insulin Calculation

The calculation of insulin units in this DIY insulin pump is based on a fundamental concept in diabetes management: the carbohydrate ratio (CR). Here is a detailed explanation:

1. Carbohydrate Ratio (CR)

The CR is a measure of how many grams of carbohydrates are compensated by one unit of insulin. This ratio is individually determined and can vary depending on factors such as:

- Insulin sensitivity: Some people are more sensitive to insulin than others.
- Time of day: Insulin sensitivity can vary throughout the day.
- Activity level: Physical activity increases insulin sensitivity.

In the code, the CR is calculated using the following formula:

```
cr = 500 / total_daily_dose
```

This formula is a simplification. The value '500' is a general guideline, but in clinical practice, the rule of 450 or 500 is often used to divide the total daily dose of insulin to obtain an estimate of the carbohydrate ratio. This number (450 or 500) represents an estimate of the total amount of carbohydrates (in grams) that can be metabolized by one unit of insulin over a certain period.

Example: If someone uses a total daily dose of 25 units of insulin, the CR is:

```
cr = 500 / 25 = 20
```

This means that 1 unit of insulin can process 20 grams of carbohydrates.

2. Calculating Insulin Needs

Once the CR is known, the required amount of insulin for a specific meal can be calculated using the following formula:

```
units_of_insulin = grams_of_carbohydrates / cr
```

- **Total daily dose:** This is the total amount of insulin (in units) that a person needs per day. This value is usually determined by a doctor.
- **Grams of carbohydrates:** This is the amount of carbohydrates (in grams) that the person plans to consume.
- **Units of insulin:** The calculated amount of insulin needed to process the consumed carbohydrates.

Example: If the person wants to consume 60 grams of carbohydrates and the CR is 20, then the required amount of insulin is:

```
units_of_insulin = 60 / 20 = 3 units
```

3. Calculating Motor Steps

The code converts the calculated insulin units into the number of steps the stepper motor needs to rotate to deliver the correct dose.

```
steps_per_revolution = 200
insulin_per_revolution = 1.0
steps_per_unit_insulin = steps_per_revolution / insulin_per_revolution
total_steps = int(units_of_insulin * steps_per_unit_insulin)
```

- `steps_per_revolution` : This is the number of steps the stepper motor needs to take to complete one full revolution. This value depends on the motor used.
- `insulin_per_revolution` : This is the amount of insulin (in units) delivered for one full revolution of the motor. This value depends on the mechanical setup of the pump (e.g., the lead of the syringe screw). This is the calibration constant and is crucial for the accuracy of the pump.
- `steps_per_unit_insulin` : The number of steps the motor needs to take to deliver one unit of insulin.
- `total_steps` : The total number of steps the motor needs to take to deliver the calculated amount of insulin. The result is rounded down to a whole number using `int()` .

Example: If the stepper motor takes 200 steps per revolution and 1 revolution delivers 1 unit of insulin, and the calculated dose is 3 units, then the total number of steps is:

```
steps_per_unit_insulin = 200 / 1 = 200 steps/unit
total_steps = int(3 * 200) = 600 steps
```

Important Considerations

- **Calibration:** The value of `insulin_per_revolution` is crucial and must be accurately calibrated. Any deviation in this value will lead to incorrect insulin dosing.
- **Accuracy:** The accuracy of insulin delivery depends on the accuracy of the stepper motor, the mechanical construction of the pump, and correct calibration.
- **Safety:** Incorrect calculations or inaccurate insulin delivery can have serious health consequences. It is of the utmost importance that the code and hardware are thoroughly tested and validated in a safe environment. Always consult a qualified healthcare professional for advice on diabetes management.