Building an Open Artificial Pancreas System

openaps

Published with GitBook

Table of Contents

Introduction 0
- Table of Contents 0.1
- Conventions 0.2
- Ways to Contribute 0.3
- Where to Go for Help 0.4

Phase 0: General Setup 1
- Baseline Data 1.1
- Hardware 1.2
- Seting Up Your Raspberry Pi 1.3
- Setting Up openaps and Dependencies 1.4

Phase 1: Logging, Cleaning, and Analyzing Your Data 2
- Configuring and Learning to Use openaps Tools 2.1

Phase 2: Build a Manual System 3
- Using oref0 Tools 3.1
- [Understanding oref0-determine-basal recommendations] (docs/Build-manual-system/Understand-determine-basal.md) 3.2
- Creating a Loop and Retry Logic 3.3

Phase 3: Automate Your System 4
- Creating a Schedule 4.1
- Visualization and Monitoring 4.2
- Validating and Testing 4.3
- Keeping Up To Date 4.4

Phase 4: Iterate and Improve 5

Resources 6
- Technical Resources 6.1
- Troubleshooting 6.2
- #OpenAPS Overview and Project History 6.3
- Other Projects, People & Tools 6.4
- FAQs 6.5
- Glossary 6.6

Introduction

Welcome to the openaps documentation!

openaps is part of a set of development tools to support a self-driven Do-It-Yourself (DIY) implementation of an artificial pancreas based on the OpenAPS reference design. (Click here to view an image of the common physical components of an OpenAPS setup.)

By proceeding to use these tools or any piece within, you agree to the copyright (see LICENSE.txt for more information) and release any contributors from liability.

The tools may be categorized as:

- **monitor** collecting data and operational status from devices, and/or aggregating as much data as is relevant into one place
- **predict** make predictions about what will happen next
- **control** enacting changes, and feeding more data back into the **monitor**, closing the loop

You may also find this flowchart helpful to further break down the areas of monitor, predict, and control into various stages of general setup; logging, cleaning, and analyzing data; building a manual system; automating your work; and iterating and improving.

A Note on DIY and the "Open" Part of OpenAPS

This is a set of development tools to support a self-driven DIY implementation. Any person choosing to use these tools is solely responsible for testing and implementing these tools independently or together as a system.

The DIY part of OpenAPS is important. There are very good reasons why this isn't a single downloadable script. While formal training or experience as an engineer or a developer is not a prerequisite, a growth mindset is required to learn to work with the "building blocks" that will help you develop your OpenAPS instance. Remember as you consider this project that this is not a "set and forget" system; and OpenAPS implementation requires diligent and consistent testing and monitoring to ensure each piece of the system is monitoring, predicting, and controlling as desired. The performance and quality of your system lies solely with you.

This community of contributors believes in "paying it forward," and individuals who are implementing these tools are asked to contribute by asking questions, helping improve documentation, and contributing in other ways.

OpenAPS System Development Phases

This documenation is organized into a series of phases that progressively build upon the openaps development tools towards a working OpenAPS system. Click here for a visualization and breakdown of the phases. The phases are as follows:

- Phase 0: General Setup
  Record baseline data, acquire and configure hardware, install software, and become familiar with the openaps environment
- Phase 1: Logging, Cleaning, and Analyzing Your Data
  Create tools for logging and analyzing pump and CGM data
- Phase 2: Build a Manual System
  Use the logged data with oref0 tools to suggest insulin dose adjustments, review and refine algorithms, test different scenarios for safety, prepare for creating a loop and implementing retry logic
- Phase 3: Automate Your System
  Apply the recommendations automatically and in real time by creating a schedule and continuing to validate and assess outputs
- Phase 4: Iterate and Improve
  Improve the functionality of the system with additional software or hardware development

At the beginning of each phase, an outline summarizes:

- Upon Completion—The capabilities that the user and system should have after completing the phase
- Phase Tasks—Tasks or steps to take during the phase
- Community Contribution—How users should be able to contribute to the #OpenAPS project and/or openaps development tool set during and after the phase

Summary

- Introduction

    - Table of Contents
    - Conventions
    - Ways to Contribute
    - Where to Go for Help
- Phase 0: General Setup
    - Baseline Data
    - Hardware
    - Seting Up Your Raspberry Pi
    - Setting Up openaps and Dependencies
- Phase 1: Logging, Cleaning, and Analyzing Your Data
    - Configuring and Learning to Use openaps Tools
- Phase 2: Build a Manual System
    - Using oref0 Tools
    - [Understanding oref0-determine-basal recommendations] (docs/Build-manual-system/Understand-determine-basal.md)
    - Creating a Loop and Retry Logic
- Phase 3: Automate Your System
    - Creating a Schedule
    - Visualization and Monitoring
    - Validating and Testing
    - Keeping Up To Date
- Phase 4: Iterate and Improve
- Resources
    - Technical Resources
    - Troubleshooting
    - #OpenAPS Overview and Project History
    - Other Projects, People & Tools
    - FAQs
    - Glossary

Conventions

Some conventions used in this guide:

- Wherever you see text that is formatted `like this`, it is a code snippet.
- You will see a `$` at the beginning of many of the lines of code. This indicates that it is to be entered and executed at the terminal prompt. Do not type in the `$`.
- Wherever there are `<bracketed_components>` in the the code, these are meant for you to insert your own information. Most of the time, it doesn't matter what you choose **as long as you stay consistent throughout this guide**. That means if you choose `Barney` as your `<my_pump_name>`, you must use `Barney` every time you see `<my_pump_name>`. Choose carefully. Do not include the `< >` brackets in your name.

Some familiarity with using the terminal will go a long way, so if you aren't comfortable with what `cd` and `ls` do, take a look at some of the Linux Shell / Terminal commands on the Troubleshooting page and the reference links on the Technical Resources page.

One helpful thing to do before starting any software work is to log your terminal session. This will allow you to go back and see what you did at a later date. This will also be immensely helpful if you request help from other OpenAPS contributors as you will be able to provide an entire history of the commands you used. To enable this, just run `$ script <filename>` at the beginning of your session. It will inform you that `Script started, file is <filename>`. When you are done, simply `$ exit` and it will announce `Script done, file is <filename>`. At that point, you can review the file as necessary.

Ways to Contribute

OpenAPS doesn't require you to be a formally trained engineer/developer/anything to get started or use these tools. The main requirement is interest and willingness to safely DIY new technology that may help improve your life as well as others.

If you're not sure where to get started, here are some ways to get involved:

- Do a fresh install using this guide and see where you get stuck; if you have to do something off-script, there is a reasonable chance the script is either wrong or your case is "special" and should be accounted for here. Make edits and submit a pull request to change the document and assist others.
- Additionally, this guide needs more work, always. If there is something that is not documented, do one of two things: a) submit a pull request (see above). or b) log an "issue" (go here to see the open issues) about the section or thing that needs more documentation.
- Ask questions on gitter; if your question wasn't answered in this doc and gets answered there, chances are it should be included here—go ahead and add it to the appropriate section.
- Test the openaps tools for different use cases and report back with your findings. Logs files and, if you're comfortable, associated device data (CareLink CSV files, for example) are extremely helpful for debugging.
- Submit issues on GitHub and work with other contributors to get them resolved.
- Develop a plugin to enhance the functionality or ease-of-use of openaps.
- Spread the word about #OpenAPS and get others involved; the more, the merrier. (You can direct them to OpenAPS.org for more information.)

If you would like to work on the core openaps code, take a look at the openaps contributing guidelines before getting started.

Communication Channels

There are several ways to communicate with other participants and contributors in the #OpenAPS project. See also the Resources section for additional assistance.

Gitter

Gitter is a messaging/chat service similar to IRC. It provides integration with GitHub and several other services. The nightscout/intend-to-bolus channel is where you will find active #OpenAPS discussions ranging from technical issues with openaps tools to control theory to general information. It is a great place to introduce yourself and get some help from those who are a few steps further down the road.

Google Groups

A private google group focused on #OpenAPS development work can be found here. Request access to participate and see some of the archived discussions. If you're new, make sure to introduce yourself!

Issues on openaps GitHub

For reporting issues on the openaps tools formally, the openaps issues page on GitHub is the proper forum. Feel free to try and get through the issues by working with others on the Gitter channel first if you think it may be something unrelated to the codebase.

Other online forums

Those in the #OpenAPS community are frequently found in other forums, such as on Twitter (using the #OpenAPS hashtag, as well as #WeAreNotWaiting) and on Facebook in the "CGM In The Cloud" group.

Setup

The setup process is broken into two parts: configuring the Raspberry Pi and installing the openaps tools and dependencies. After completing these steps, you will be able to use the openaps tools to communicate with your insulin pump and CGM.

At this stage, you may want to begin documenting each step that you take. This will help in two ways.

First, this enables you to better ask for assistance if you run into errors, bugs, etc. By explaining where you are in the documentation and what you're seeing (by copying and pasting your last command and the output), someone can better provide tips on what you should consider next.

Second, this will enable you to help us improve our documentation. Did we skip a step, or not explain clearly? After you get through the setup instructions, you should consider forking a copy of these docs, editing with any changes you think should be made, and submitting a pull request (PR) back to the master. Others will be able to review & discuss any edits, make further changes, and pull this edits into the main file. This helps us all "pay it forward" as we go!

Summary

- Introduction
- Phase 0: General Setup
  - Baseline Data
  - Hardware
  - Seting Up Your Raspberry Pi
  - Setting Up openaps and Dependencies
- Phase 1: Logging, Cleaning, and Analyzing Your Data
- Phase 2: Build a Manual System
- Phase 3: Automate Your System
- Phase 4: Iterate and Improve
- Resources

Baseline data

There is no requirement to share your data to use the openaps toolset or participate in the OpenAPS project. Individuals within the project who share their data do so at will and you should do the same only if you feel comfortable. That being said, it is always a good idea to record your data before embarking on a new set of experiments. This will be helpful to understand the effects of the system as well as gain a better understanding of your response to different control strategies.

CGM Data

Before getting started, we ask that you store at least 30 days of CGM data. For now, the easiest way to do that is to upload your Dexcom receiver to Dexcom Studio or, if you use a Medtronic CGM, upload your CGM data to CareLink. We suggest you get in the habit of doing this regularly so that you have ongoing data to show trends in your overall estimated average glucose (eAG, a good indicator in trends in A1c) and variations in your "time in range."

Recent A1c

Go ahead and document your most recent A1c and keep it somewhere handy. This will allow you to compare your before/after results as well as be able to share it (if you choose) once there is a request from OpenAPS researchers, who may aggregate anonymous data to show what happens when people use OpenAPS.

Hardware

This section describes the hardware components required for a 'typical' OpenAPS implementation. There are numerous variations and substitutions that can be made but the following items are recommended for getting started. If you come across something that doesn't seem to work, is no longer available, or if you have a notable alternative, feel free to edit this document with your suggestions.

If you're interested in working on communication for another pump (Omnipod, Animas, etc), click here to join the collaboration group focusing on alternative pump communication.

Required Hardware

- **Insulin Pump**: Medtronic MiniMed model #:
  - 512/712
  - 515/715
  - 522/722
  - 523/723 (with firmware 2.4A or lower)
  - 554 (European Veo, with firmware 2.6A or lower)
- **Pump Communication**:
  - Medtronic CareLink USB stick
- **Continuous Glucose Monitor (CGM)**:
  - Dexcom CGM (G4 Platinum or Platinum with Share system) OR
  - Medtronic CGM (MiniMed Paradigm REAL-Time Revel or Enlite)
- **Other Supplies**:
  - Raspberry Pi 2 Model B ("RPi2")**(see note below)
  - 8 GB (or greater) micro SD card
  - Micro SD card to regular SD card converter [optional, but recommended so that you can use the micro SD card in a regular sized SD card drive]
  - Low-profile USB WiFi adapter
  - 2.1 Amp (or greater) USB power supply or battery
  - Micro USB cable(s)
  - AAA batteries (for pump)
  - Case [optional]
  - Cat5 or Cat6 Ethernet cable [optional]
  - HDMI cable [optional, used for connecting the RPi2 to a screen for initial setup ease]
  - USB Keyboard [optional, used to interact with the RPi2 via its own graphics

interface on your TV screen]
  - USB Mouse [optional, for the same purpose]

** Note: Several #OpenAPS contributors recommend the Raspberry Pi 2 CanaKit, which includes several essential accessories in one package and can be purchased through Amazon

The CanaKit has the RPi2, SD card, WiFi adapter, and wall power supply. It also comes with a case, HDMI cable, and heat sink, none of which are required for an OpenAPS build. The kit does not have a micro USB cable (required to connect a Dexcom G4 receiver to the RPi) or a battery, which can be used in lieu of the wall power supply for portability.

Additionally, for the Raspberry Pi and peripherals, verified sets of working hardware can be found here.

Eventually, once you have an entire OpenAPS build up and running, it is recommended that you have backup sets of equipment in case of failure.

Hardware Details & Recommendations

Medtronic Insulin Pump

See currently known working list of pumps above. The easiest way to navigate to the Utilities / Connect Devices menu on your pump. If "PC Connect" is present in this menu, your pump is *not* compatible with OpenAPS.

Due to changes in the firmware, the openaps tools are only able to function in full on the above pump models. Security features were added in firmware version 2.5A that prevent making some remote adjustments via the CareLink USB stick. Each pump series is slightly different, and openaps functionality is still being ironed out for some of them. For 512/712 pumps, certain commands like Read Settings, BG Targets and certain Read Basal Profile are not available, and requires creating a static json for needed info missing to successfully run the loop (see example here).

If you need to acquire an appropriate pump check CraigsList or other sites like Medwow or talk to friends in your local community to see if there are any old pumps lying around in their closets gathering dust. MedWow is an eBay-like source for used pumps. Note: If you're buying a pump online, we recommended you ask the seller to confirm the firmware version of the pump. (You may also want to consider asking for a video of the pump with working functionality before purchasing.)

There are several #OpenAPS participants working on ways to use other pumps (including non-Medtronic models). If you would like to get more information on the progress in these areas, take a look at the #OpenAPS Google Group or click here to join the Slack channel.

CareLink USB Stick

Currently, the only supported device* for uploading pump data and interfacing on the #OpenAPS is the CareLink USB stick. We recommend you purchase at least two sticks because if one breaks, acquiring another stick will take time and will delay development. Additionally, due to the short range of communication between the CareLink stick and the Medtronic pumps, some users set up multiple sticks in different locations to maximize the chances of successful transmissions.

Medtronic

American Diabetes Wholesale

A limitation of the Carelink USB stick is the short range of radio communications with the Medtronic pump. The radio signals are trasmitted from the end of the stick opposite the USB connector, on the flat grey side of the stick (see this set of experiments for details). Using a USB extension cable and angling the stick appropriately will assist in improving the connection.

Rerii 90 Degree USB Extension Cable

Mediabridge Products USB Extension Cable

**Note that there are a few options in progress that may become alternatives to the Carelink, although they are not documented in the OpenAPS docs yet. "RileyLink" is another DIY piece of hardware that is in development and has potential to replace the CareLink stick & Raspberry Pi to communicate with a pump. It is not yet built out and reliable to be a part of an OpenAPS yet, and thus is not currently recommended. There are also other pieces of DIY hardware that are works in progress that may be alternatives to using the CareLink stick and Raspberry Pi, so stay tuned.*

CGM: Dexcom G4 Platinum System (with or without Share) OR Medtronic

The openaps tool set currently supports two different CGM systems: the Dexcom G4 Platinum system (with or without the Share functionality) and the Medtronic system. With Dexcom, the Share platform is not required as communication with the receiver is usually accomplished via USB directly to the Pi. You can also pull CGM data from Nightscout as an alternative (documentation coming soon), or use xDrip (see below). The Medtronic CGM system communicates directly with the associated pump, so the data can be retrieved using the CareLink USB stick.

Using the Dexcom CGM:

Note: Your Dexcom should be nearly fully charged before plugging it in to your Raspberry Pi. If, when you plug in your receiver, it causes your WiFi dongle to stop blinking, that is a sign that it is drawing too much power and needs to be charged. Once the receiver is fully charged, it will stay charged when connected to the Pi.

Your OpenAPS implementation can also pull CGM data from a Nightscout site in addition to pulling from the CGM directly.

- You can find more documentation about pulling CGM data from a Nightscout site here.
- If you have an Android phone, you can use the xDrip app to get your data from the Dexcom to Nightscout, to then be used in OpenAPS.
    - If you have a Share receiver follow these directions to set up your Android uploader and Nightscout website.
    - You could also build a DIY receiver. Directions to build the receiver, set up your uploader and Nightscout can be found here.
    - You can also use part of the DIY receiver set up - the wixel – directly to the raspberry pi. Learn more about the wixel setup here and here.

Using the Medtronic CGM:

Because the Medronic pump collects data directly from the Enlite sensors, OpenAPS will retrieve CGM data in addition to your regular pump data from your pump. While you use the same OpenAPS commands to get it, the Medtronic CGM data need a little special formatting after being retrieved. We'll discuss these special circumstances as they come up later.

Raspberry Pi 2 Model B

The Raspberry Pi 2 (RPi2) model B is a credit-card sized single-board computer. The RPi2 primarily uses Linux kernel based operating systems, which must be installed by the user onto a micro SD card for the RPi2 to work. The RPi2 currently only supports Ubuntu, Raspbian, OpenELEC, and RISC OS. We recommend installing either Ubuntu or Raspbian. In this tutorial, you will learn how to do a "cableless" and "headless" install of Raspbian. You will be able to access and control the RPi2 via an SSH client on Windows, Mac OS X, Linux, iOS, or Android.

The RPi2 has 4 USB ports, an ethernet port, an HDMI port, and a micro USB power-in jack that accepts 2.1 Amp power supplies. In this tutorial, you will need to access the USB ports, micro USB power-in jack, and possibly the Ethernet jack (if wireless failure occurs). You will not require the HDMI port or a monitor.

Raspberry Pi 2 Model B

Micro SD Card

An 8 or 16 GB micro SDHC card is recommended. Get one that is class-4 or greater and is a recognized name brand, such as SanDisk, Kingston, or Sony. A list of verified working hardware (including SD cards) can be found here.

SanDisk Ultra 16GB Ultra Micro SDHC UHS-I/Class 10 Card with Adapter

Sony 16GB Class 10 UHS-1 Micro SDHC

Note: A known issue with the Raspberry Pi is that the SD card may get corrupted with frequent power cycles, such as when the system gets plugged and unplugged frequently from an external battery. Most core developers of openaps recommend purchasing extra SD cards and having them pre-imaged and ready to use with a backup copy of openaps installed, so you can swap it out on the go for continued use of the system.

WiFi Adapter

A minimalistic, unobtrusive WiFi USB adapter is recommended. The low-profile helps to avoid damage to both the RPi2 and the adapter as the RPi2 will be transported everywhere with the user.

Edimax EW-7811Un 150Mbps 11n Wi-Fi USB Adapter

Buffalo AirStation N150 Wireless USB Adapter

2.1 Amp USB Battery Power Supply

A large-capacity power supply that is greater than 8000 mAh (milliAmp-hours) is recommended for full day use. Make sure that the battery has at least one 2.1 Amp USB output. A battery with a form-factor that minimizes size is recommended, to allow the patient to be as ambulatory as possible. When you have a full OpenAPS implemented and working, you will want to have multiple batteries to rotate and recharge. A battery that can deliver power while it charges is ideal as you will be able to charge it on-the-fly without shutting down and restarting the RPi2.

TeckNet® POWER BANK 9000mAh USB External Battery Backup Pack

USB Cables

USB cables with a micro connector on one end and a standard (Type A) connector on the other are used to connect the power supply and the Dexcom receiver to the RPi2. Most cables will work fine, but some prefer to select lengths and/or features (such as right-angled connectors) to improve portability.

Rerii Black Golden Plated 15 cm Length Micro-B Male Left Angle USB cable

Monoprice Premium USB to Micro USB Charge, Sync Cable - 3ft

AAA Batteries

Repeated wireless communication with the pump drains the battery quite quickly. With a loop running every five minutes, a standard alkaline AAA—recommended by Medtronic—lasts somewhere between four to six days before the pump goes to a "Low Battery" state and stops allowing wireless transmission. Lithium batteries last significantly longer but do not give much warning when they are about to die, but alerts can be created to provide warning about the status of the battery. For further information on batteries, see this study on AAA battery use in a looping pump.

Cases

The Raspberry Pi is extremely minimalistic and does not come in a protective case. This is fine for development work, but presents an issue for day-to-day use. There are hundreds of cases available, but here some examples of what others are using in their OpenAPS builds.

JBtek® Jet Black Case for Raspberry Pi B+ & Raspberry Pi 2 Model B

Raspberry Pi B+ /PI2 Acrylic Case

Additionally, for mobile use, it is helpful to have something besides a lunchbox to carry the entire rig around. The size and weight of the component set as well as the limited range of the CareLink USB stick constrains the options here, but there are still some workable solutions. Waist-worn running gear and camera cases seem to work well.

FlipBelt

Lowepro Dashpoint 20

Setting Up Your Raspberry Pi

In order to use the RPi2 with openaps development tools, the RPi2 must have an operating system installed and be set up in a very specific way. There are two paths to the intial operating system instalation and WiFI setup. Path 1 is recommended for beginners that are very new to using command prompts or "terminal" on the Mac. Path 2 is considered the most convenient approach for those with more experience with coding and allows the RPi2 to be set up without the use of cables, which is also known as a headless install. Either path will work and the path you choose is a matter of personal preference. Either way, it is recommended that you purchase your RPi2 as a CanaKit, which includes everything you will need for a GUI install.

For the Path 1 GUI install you will need:

- A Raspberry Pi 2 CanaKit or similar, which includes several essential accessories in one package
- USB Keyboard
- USB Mouse
- A TV or other screen with HDMI input

For the Path 2 Headless install, you will need:

- Raspberry Pi 2
- 8 GB micro SD Card [and optional adapter so that you can plug in the micro SD Card into your computer]
- Low Profile USB WiFi Adapter
- 2.1 Amp USB Power Supply
- Micro USB cable
- Raspberry Pi 2 CanaKit
- Console cable, ethernet cable, or Windows/Linux PC that can write ext4 filesystems

Download and Install Raspbian Jessie

Note: If you ordered the recommended CanaKit, your SD card will already come imaged. However, if you don't already know whether it's Raspbian 8 Jessie or newer (see below), just treat it as a blank SD card and download and install the latest versian of Raspbian (currently version 8.0, codename Jessie).

Download Raspbian

Raspbian is the recommended operating system for OpenAPS. Download the latest version (Jessie September 2015 or newer) of Raspbian here. Make sure to extract the disk .img from the ZIP file. Note that the large size of the Raspbian Jessie image means its .zip file uses a different format internally, and the built-in unzipping tools in some versions of Windows and MacOS cannot handle it. The file can be successfully unzipped with [7-Zip] (http://www.7-zip.org/) on Windows and [The Unarchiver] (https://itunes.apple.com/us/app/the-unarchiver/id425424353?mt=12) on Mac (both are free).

Write Raspbian to the Micro SD Card

Erase (format) your SD card using https://www.sdcard.org/downloads/formatter_4/

Write the Raspbian .img you extracted from the ZIP file above to the SD card using the instructions at https://www.raspberrypi.org/documentation/installation/installing-images/

Detailed Windows Instructions

- First, format your card to take advantage of the full size it offers
    - If you got your through CanaKit, when you put it in your PC it will look like it is 1GB in size despite saying it is 8GB
- Download and install: https://www.sdcard.org/downloads/formatter_4/
- Run SDFormatter
    - Make sure your Micro SD Card is out of your Raspberry PI (shut it down first) and attached to your computer
    - Choose the drive where your card is and hit "Options"
    - Format Type: Change to Full (Erase)
    - This will erase your old Rasbian OS and make sure you are using the full SD card's available memory

    -
    - Format the card
- Download Rasbian 8 / Jessie
    - https://www.raspberrypi.org/downloads/raspbian/
    - Extract the IMG file
- Follow the instruction here to write the IMG to your SD card
    - https://www.raspberrypi.org/documentation/installation/installing-images/README.md
- After writing to the SD card, safely remove it from your computer and put it back into your RPi2 and power it up

Connect and configure WiFi

- Insert the included USB WiFI into the RPi2.
- Next, insert the Micro SD Card into the RPi2.

Path 1: Keyboard, Mouse, and HDMI monitor/TV

- First, insert your USB keyboard and USB mouse into the RPi2.
- Next, connect your RPi2 to a monitor or T.V. using the included HDMI cable.
- Finally connect your RPi2 using the power adapter.
- You should see the GUI appear on sceen.
- Configure WiFi per the instruction pamphlet included with your CanaKit.
- Once you have installed Rasbian and connected to WiFI, you can disconnect the

mouse, keyboard and HDMI cable.

Remember to keep your RPi2 plugged in, just disconnect the peripherals. Also remember to never disconnect your RPi2 without shutting it down properly using the `sudo shutdown -h now` command. If you are unable to access the Pi and must power it off without a shutdown, wait until the green light has stopped flashing (indicating the Pi is no longer writing to the SD card).

You can now skip to Test SSH Access and SSH into your RPi2.

Path 2: Console or Ethernet cable

- Get and connect a console cable (use this guide),
- Temporarily connect RPi to a router with an ethernet cable and SSH in (see below), or
- Connect the RPi directly to your computer with an ethernet cable (using this guide) and SSH in (see below)

Configure WiFi Settings

Once you connect to the Pi, you'll want to set up your wifi network(s). It is recommended to add both your home wifi network and your phone's hotspot network if you want to use OpenAPS on the go.

To configure wifi:

Type `sudo bash` and hit enter

Input `wpa_passphrase "<my_SSID_hotspot>" "<my_hotspot_password>" >> /etc/wpa_supplicant/wpa_supplicant.conf` and hit enter (where `<my_SSID_hotspot>` is the name of your phone's hotspot and `<my_hotspot_password>` is the password).

(It should look like: `wpa_passphrase "OpenAPS hotspot" "123loveOpenAPS4ever" >> /etc/wpa_supplicant/wpa_supplicant.conf`)

Input your home wifi next: `wpa_passphrase "<my_SSID_home>" "<my_home_network_password>" >> /etc/wpa_supplicant/wpa_supplicant.conf` (and hit enter)

You can now skip to Test SSH Access and SSH into your RPi2.

Path 3: Headless WiFi configuration (Windows/Linux only)

Keep the SD card in the reader in your computer. In this step, the WiFi interface is going to be configured in Raspbian, so that we can SSH in to the RPi2 and access the device remotely, such as on a computer or a mobile device via an SSH client, via the WiFi connection that we configure. Go to the directory where your SD card is with all of the files for running Raspbian on your RPi2, and open this file in a text editor.

`/path/to/sd/card/etc/network/interfaces`

Edit the file so it looks like this:

```
auto lo
iface lo inet loopback
iface eth0 inet dhcp

auto wlan0
allow-hotplug wlan0
iface wlan0 inet dhcp
wpa-ssid <your-network-name>
wpa-psk <your-password>
```

Replace `<your-network-name>` and `<your-password>` with your own credentials (just text, no quotes). Save the file (without adding any additional extensions to the end of the filename).

Boot your Pi. (Put the SD card into the RPi2. Plug in the compatible USB WiFi adapter into a RPi2 USB port. Get a micro USB cable and plug the micro USB end into the side of the RPi2 and plug the USB side into the USB power supply.)

If you are unable to access this file on your computer:

- Connect your Pi to your computer with an ethernet cable and boot your Pi
- Log in using PuTTY. The Host Name is `raspberrypi.local` and the Port is 22. The login is `pi` and the password is `raspberry`.
- Type `sudo nano /etc/network/interfaces` and edit the file as described above, or follow the OS X directions below.

Test SSH Access

Windows

Make sure that the computer is connected to the same WiFi router that the RPi2 is using. Download PuTTY here. Hostname is `pi@raspberrypi.local` and default password for the user `pi` is `raspberry`. The port should be set to 22 (by default), and the connection type should be set to SSH. Click `Open` to initiate the SSH session.

Mac OS X / Linux

Make sure that the computer is connected to the same WiFi router that the RPi2 is using.

Open Terminal and enter this command:

`ssh pi@raspberrypi.local`

Default password for the user `pi` is `raspberry`

iOS

Make sure that the iOS device is connected to the same WiFi network that the RPi2 is using. Download Serverauditor or Prompt 2 (use this if you have a visual impairment). Hostname is `pi@raspberrypi.local` and the default password for the user `pi` is `raspberry`. The port should be set to 22 (by default), and the connection type should be set to SSH.

You probably also want to make your phone a hotspot and configure the WiFi connection (as above) to use the hotspot.

Android

Make sure that the Android device is connected to the same WiFi network that the RPi2 is using. Download an SSH client in the Google Play store. Hostname is `pi@raspberrypi.local` and the default password for the user `pi` is `raspberry`. The port should be set to 22 (by default), and the connection type should be set to SSH. You may need to ssh using the ip address instead; the app "Fing - Network Tools" will tell you what the address is if needed.

You probably also want to make your phone a hotspot and configure the WiFi connection (as above) to use the hotspot.

Note: If connecting to the RPi2 fails at this point, the easiest alternative is to temporarily connect RPi to your router with an ethernet cable and SSH in, making sure both the computer and the RPi2 are connected to the same router.

Configure the Raspberry Pi

Verify your Raspian Version

- In order to do this, you must have done Path 1 or Path 2 above so that you have an environment to interact with
- Go to the shell / Terminal prompt. If running the GUI, look at the Menu in the upper left and click the icon three to the right of it (looks like a computer)
- Type `lsb_release -a`
- If it says anything about Release 8 / Jessie, you have the correct version and can continue.

- If it says anything else, you need to go back to Download and Install Raspbian Jessie

Run raspi-config

Run

```
sudo raspi-config
```

to expand filesystem, change user password and set timezone (in internationalization options). This will take effect on the next reboot, so go ahead and reboot if prompted, or run `sudo reboot` when you're ready.

Setup Password-less Login [optional]

Windows

If you don't already have an SSH key, follow this guide from GitHub to create one.

Create a .ssh directory on the Pi: run `mkdir .ssh`

Log out by typing `exit`

and copy your public SSH key into your RPi2 by entering

```
ssh-copy-id pi@raspberrypi.local
```

Now you should be able to log in without a password. Try to SSH into the RPi2 again, this time without a password.

Mac and Linux

If you don't already have an ssh key, then on your local computer (*not* on the Pi), by running `ssh-keygen` (keep hitting enter to accept all the defaults)

Next create a .ssh directory on the Pi: `ssh pi@raspberrypi.local` , enter your password, and run `mkdir .ssh`

Next copy your public key to the Pi: `scp ~/.ssh/id_rsa.pub pi@raspberrypi.local:~/.ssh/authorized_keys`

Finally `ssh pi@raspberrypi.local` to make sure you can log in without a password.

Disabling password login [optional]

To secure the Pi, you should either set a password (using `sudo raspi-config` above, or with `sudo passwd`), or disable password login completely. If you want to disable password login (so you can only log in with your ssh key), open the `sshd_config` file in nano text editor on the Pi as follows

`sudo nano /etc/ssh/sshd_config`

Change the following

```
PermitRootLogin yes
# PasswordAuthentication yes
```

to

```
PermitRootLogin no
PasswordAuthentication no
```

Note that the second line was previously commented out.

From now on you will be able to SSH in with your private SSH key only.

Wifi reliability tweaks [optional]

Many people have reported power issues with the 8192cu wireless chip found in many wifi adapters when used with the Raspberry Pi. As a workaround, we can disable the power management features (which this chip doesn't have anyway) as follows:

```
sudo bash -c 'echo "options 8192cu rtw_power_mgnt=0 rtw_enusbss=0" >>
/etc/modprobe.d/8192cu.conf'
```

Watchdog [optional]

Now you can consider installing watchdog, which restarts the RPi2 if it becomes unresponsive.

Enable the built-in hardware watchdog chip on the Raspberry Pi:

`sudo modprobe bcm2708_wdog`

`sudo bash -c 'echo "bcm2708_wdog" >> /etc/modules'`

Install the watchdog package, which controls the conditions under which the hardware watchdog restarts the Pi:

`sudo apt-get install watchdog`

Next, add watchdog to startup applications:

```
sudo update-rc.d watchdog defaults
```

Edit the config file by opening up nano text editor

```
sudo nano /etc/watchdog.conf
```

Uncomment the following: (remove the # from the following lines, scroll down as needed to find them):

```
max-load-1
watchdog-device
```

Finally, start watchdog by entering:

```
sudo service watchdog start
```

Update the Raspberry Pi [optional]

Update the RPi2.

```
sudo apt-get update && sudo apt-get -y upgrade
```

The packages will take some time to install.

Setting Up openaps and Dependencies

This section provides information on installing the base openaps toolkit and its dependencies.

Easy install of openaps and dependencies

Using the package manager

This is the recommended way to install:

```
curl -s https://raw.githubusercontent.com/openaps/docs/master/scripts/quick-packages.sh |
bash -
```

This uses this script to install all the dependencies in one step.

If the install was successful, the last line will say something like:

openaps 0.0.9 (although the version number may have been incremented)

If you do not see this or see error messages, try running the script multiple times.

Installing from source

It's possible to use the package manager to install development branches. If you are hacking on the code, you'll need a way to develop using versions you control. Here's a quick way to do that:

```
curl -s https://raw.githubusercontent.com/openaps/docs/master/scripts/quick-src.sh | bash
-
```

If successful, the last line will say something like:

openaps 0.0.10-dev (although the version number may have been incremented)

Manual install [optional]

Install Python and Node.js Packages System-Wide [optional]

Run

`sudo apt-get install python python-dev python-setuptools python-software-properties python-numpy python-pip nodejs-legacy npm`

This installs a number of packages required by openaps.

Install openaps [optional]

Run

`sudo easy_install -ZU setuptools` `sudo easy_install -ZU openaps`

Running this command will also update openaps on your system if a newer version is available.

Install udev-rules [optional]

Run

`sudo openaps-install-udev-rules`

Enable Tab Completion [optional]

Run

`sudo activate-global-python-argcomplete`

Set up Git

Run

`sudo apt-get install git`

In order to set your git account's default identity, you will need to run the following two commands:

`git config --global user.email "you@example.com"`

`git config --global user.name "Your Name"`

replace `you@example.com` and `Your Name` with your own information, but keep the quotes.

Phase 1: Logging, Cleaning, and Analyzing Your Data

Phase 1 focuses on accessing, logging, cleaning up, and analyzing data from the pump and CGM. Data fidelity is extremely important, especially when dosing is being considered. Take the time to review what the openaps tools are outputting and carefully compare the logs against your own CareLink and CGM reports.

Outline of Phase 1

- Upon Completion
  - Be able to download, record, and recall data logs from pump and CGM
  - Be able to download “real time” data
  - Have data being sent via wireless connection to a cloud database for user access
  - Have built a system able to collect data and be able to help others do the same
- Phase Tasks
  - Initalize new openaps environment
  - Add device tools, including oref0 virtual devices
  - Use tools to download data from pump and CGM (add reports)
  - Log data for future analysis, use in algorithms (invoke reports, create aliases)
  - Ensure fidelity and accuracy of recorded readings
  - Test edge cases
  - Log data “in real time” to test hardware functionality
  - Set up cloud database to accept uploaded data (optional)
  - Have logged data syncing to cloud database (optional)
  - Have your system backed up
- Community Contribution
  - Provide feedback on ...
    - Model-specific data issues (e.g. Medtronic 722 v 723)
    - Using software tools
    - Following guides to accomplish above phase tasks
  - Provide improved or alternative implementations

Summary

- Introduction

- Phase 0: General Setup
- Phase 1: Logging, Cleaning, and Analyzing Your Data
    - Configuring and Learning to Use openaps Tools
- Phase 2: Build a Manual System
- Phase 3: Automate Your System
- Phase 4: Iterate and Improve
- Resources

Logging, cleaning, and analyzing your data

Configuring and Learning to Use openaps Tools

This section provides an introduction to intializing, configuring, and using the openaps toolset. The purpose is to get you familiar with how the different commands work and to get you thinking about how they may be used to build your own closed loop. Make sure you have completed the Setting Up the Raspberry Pi 2 and Setting Up openaps sections prior to starting.

The openaps readme has detailed information on the installation and usage of openaps. You should take the time to read through it in detail, even if it seems confusing at first. There are also a number of example uses available in the openaps-example repository.

Some familiarity with using the terminal will go a long way, so if you aren't comfortable with what `cd` and `ls` do, take a look at some of the Linux Shell / Terminal links on the Technical Resources page.

Some conventions used in this guide:

- Wherever there are `<bracketed_components>` in the the code, these are meant for you to insert your own information. Most of the time, it doesn't matter what you choose **as long as you stay consistent throughout this guide**. That means if you choose `Barney` as your `< my_pump_name >`, you must use `Barney` every time you see `< my_pump_name >`. Choose carefully. Do not include the `< >` brackets in your name.
- You will see a `$` at the beginning of many of the lines of code. This indicates that it is to be entered and executed at the terminal prompt. Do not type in the `$`.

One helpful thing to do before starting is to log your terminal session. This will allow you to go back and see what you did at a later date. This will also be immensely helpful if you request help from other OpenAPS contributors as you will be able to provide an entire history of the commands you used. To enable this, just run `$ script <filename>` at the beginning of your session. It will inform you that `Script started, file is <filename>`. When you are done, simply `$ exit` and it will announce `Script done, file is <filename>`. At that point, you can review the file as necessary.

Configuring openaps

Initialize a new openaps environment

To get started, SSH into your Raspberry Pi. Go to your home directory:

`$ cd`

Create a new instance of openaps in a new directory:

`$ openaps init <my_openaps>`

As mentioned above, `<my_openaps>` can be anything you'd like: `myopenaps`, `awesome-openaps`, `openaps4ever`, `bob`, etc.

Now that it has been created, move into the new openaps directory:

`$ cd <my_openaps>`

All subsequent openaps commands must be run in this directory. If you try to run an openaps command in a different directory, you will receive an error:

`Not an openaps environment, run: openaps init`

The directory you just created and initialized as an openaps environment is mostly empty at the moment, as can been seen by running the list files command:

```
$ ls
openaps.ini
```

That `openaps.ini` file is the configuration file for this particular instance of openaps. It will contain all of your device information, plugin information, reports, and aliases. In the subsequent sections, you will be configuring your openaps instance to add these components. For now, however, it is blank. Go ahead and take a look:

`$ cat openaps.ini`

Didn't return much, did it? By the way, that `cat` command will be very useful as you go through these configuration steps to quickly check the contents of files (any files, not just `openaps.ini`). Similarly, if you see a command that you are unfamiliar with, such as `cat` or `cd`, Google it to understand what it does. The same goes for error messages—you are likely not the first one to encounter whatever error is holding you back.

Add pump as device

In order to communicate with the pump and cgm receiver, they must first be added as devices to the openaps configuration. To do this for the pump:

```
$ openaps device add <my_pump_name> medtronic <my_serial_number>
```

Here, `<my_pump_name>` can be whatever you like, but `<my_serial_number>` must be the 6-digit serial number of your pump. You can find this either on the back of the pump or near the bottom of the pump's status screen, accessed by hitting the ESC key.

Important: Never share your 6-digit pump serial number and never post it online. If someone had access to this number, and was in radio reach of your pump, this could be used to communicate with your pump without your knowledge. While this is a feature when you want to build an OpenAPS, it is a flaw and a security issue if someone else can do this to you.

Add Dexcom CGM receiver as device

Now you will do this for the Dexcom CGM receiver:

```
$ openaps device add <my_dexcom_name> dexcom
```

Note this step is not required if you are using a Medtronic CGM. The pump serves as the receiver and all of the pumping and glucose functionality are contained in the same openaps device.

Check that the devices are all added properly

```
$ openaps device show
```

should return something like:

```
medtronic://pump
dexcom://cgms
```

Here, `pump` was used for `<my_pump_name>` and `cgms` was used for `<my_dexcom_name>`. The names you selected should appear in their place.

Your `openaps.ini` file now has some content; go ahead and take another look:

```
$ cat openaps.ini
```

Now, both of your devices are in this configuration file:

```
[device "pump"]
vendor = openaps.vendors.medtronic
extra = pump.ini

[device "cgms"]
vendor = openaps.vendors.dexcom
extra = cgms.ini
```

Again, `pump` was used for `<my_pump_name>` and `cgms` was used for `<my_dexcom_name>`. Your pump model should also match your pump.

Because your pump's serial number also serves as its security key, that information is now stored in a separate ini file (here noted as `pump.ini`) that was created when you created the pump device. This makes it easier for sharing the `openaps.ini` file and also for keeping `pump.ini` and `cgms.ini` more secure. Be careful with these files. Open the pump's ini file now (use the name reported to you in the line labeled `extra` in the `openaps.ini` file).

```
$ cat pump.ini
```

It should show something like this:

```
[device "pump"]
serial = 123456
```

The serial number should match that of your pump.

If you made a mistake while adding your devices or simply don't like the name you used, you can go back and remove the devices as well. For example, to remove the pump:

```
$ openaps device remove <my_pump_name>
```

Then, you can add your pump again with a different name or serial number.

Check that you can communicate with your pump

Now that you have added these devices, let's see if we can establish communication with them. First, the pump:

```
$ openaps use <my_pump_name> model
```

should return something like:

```
"723"
```

Congratulations, you just pulled data from your pump! The `model` command is a very useful one to verify whether you can communicate with the pump. It is not, however, the only thing you can do. Take a look at the help file to see all of the possibilities:

`$ openaps use <my_pump_name> -h`

This returns a healthy bit of useful information, including a list of all the commands that can be done with `$ openaps use <my_pump_name>`. Of course, each one of those uses has its own help file as well:

```
$ openaps use <my_pump_name> model -h
usage: openaps-use pump model [-h]

 Get model number



optional arguments:
  -h, --help  show this help message and exit
```

The `-h` argument is your friend. If you ever forget what a command does, what arguments it requires, or what options it has, `-h` should be your first resource.

Go ahead and try some more pump uses to find out what they do. Note that some of the commands require additional inputs; these are detailed in the specific help files.

Check that you can communicate with your Dexcom receiver

Now let's try communicating with the Dexcom receiver.

Hint: Your Dexcom should be nearly fully charged before plugging it in to your Raspberry Pi. If, when you plug in your Dexcom, it causes your WiFi dongle to stop blinking, that is a sign that it is drawing too much power and needs to be charged.

`$ openaps use <my_dexcom_name> iter_glucose 1`

should return something like:

```
[
  {
    "trend_arrow": "FLAT",
    "system_time": "2015-08-23T21:45:29",
    "display_time": "2015-08-23T13:46:21",
    "glucose": 137
  }
]
```

Hint: if this doesn't work, check to make sure that your Dexcom receiver is plugged into your Raspberry Pi ;-)

Just like with the pump, you can use the `-h` argument to call the help files. For example:

```
$ openaps use <my_dexcom_name> iter_glucose -h
usage: openaps-use cgms iter_glucose [-h] [count]

 read last <count> glucose records, default 100, eg:

positional arguments:
  count       Number of glucose records to read.

optional arguments:
  -h, --help  show this help message and exit

* iter_glucose   - read last 100 records
* iter_glucose 2 - read last 2 records
```

Pulling blood glucose levels from Nightscout

Some people have found it more beneficial to pull blood glucose values from Nightscout rather than directly from the Dexcom receiver. In order to do that, two steps are needed:

1) Similar like above, we need to create a device that talks to Nightscout. Add this device called "curl" to your list of devices in your openaps.ini file:

[device "curl"]
fields =
cmd = bash
vendor = openaps.vendors.process
args = -c "curl -s https://yourwebsite.azurewebsites.net/api/v1/entries.json | json -e 'this.glucose = this.sgv'"

In addition, you need to alter your monitor/glucose.json report to use this device rather than the cgms device you setup above. The report will look like this in your openaps.ini file:

[report "monitor/glucose.json"]
device = curl
use = shell
reporter = text

Many people will actually setup both ways to pull the blood glucose level and switch between the different devices depending on their needs. If you are going to pull it directly from Nightscout then you will have to have internet access for the Raspberry Pi.

Adding and Invoking Reports

At this point, you should be comfortable communicating with your pump and cgm receiver with the `openaps use` command. This is great for learning and for experimenting, but it lacks the ability to generate output files. You'll notice that running

```
$ openaps use <my_dexcom_name> iter_glucose 100
```

prints *a lot* of data to the terminal. It would be great to save that data somewhere so that it can be used for logging historical records, performing calculations, and verifying actions. That is what `report` does.

Generating reports involves two steps: adding the report structures to your openaps configuration and invoking the reports to produce the desired outcome.

Adding Reports

As an example, let's suppose you would like to gather the last four hours of records from your pump. With the `use` command, that would be:

```
$ openaps use <my_pump_name> iter_pump_hours 4
```

This dumps the past four hours of pump records directly to the terminal.

Now, let's add this as a report instead:

```
$ openaps report add last_four_pump_hours.json JSON <my_pump_name> iter_pump_hours 4
```

If done correctly, the only thing returned in the terminal is:

```
added pump://JSON/iter_pump_hours/last_four_pump_hours.json
```

Let's take a closer look at each section. `openaps report add` is adding a report to your openaps configuation. The report name is `last_four_pump_hours.json`. The format of the report is `JSON`. The command that will be used to generate the report is `<my_pump_name> iter_pump_hours 4`. You will notice that this last section is identical to what was called above when you printed the output to the terminal window, except there it was done with the `use` command. The report is simply running that same command and writing the output to the file you specified in the format you specified.

Much like adding devices, this report configuration is saved to your `openaps.ini` file. You can view all of your reports there with `$ cat openaps.ini` or by using `$ openaps report show`. Similarly, you can remove reports with `$ openaps report remove <report_name>`.

Invoking Reports

Adding the report does not actually generate the output file. To do this, you need to invoke the report:

`$ openaps report invoke last_four_pump_hours.json`

Again, the terminal output will be minimal:

```
pump://JSON/iter_pump_hours/last_four_pump_hours.json
reporting last_four_pump_hours.json
```

This time, however, a new file was created. Check and see using `$ ls`; you should see a file called `last_four_pump_hours.json` in your directory. Take a look at the file with `$ cat last_four_pump_hours.json`. The file's contents should look very familiar—the same data that was printed to ther terminal window when you performed `$ openaps use <my_pump_name> iter_pump_hours 4`.

Each time you add a new report to your configuration, you should immediately invoke it and check the resulting file. This means **open the file and actually check to make sure the output is what you expect**. Don't assume that it worked just because you didn't see an error.

The reports you add are reusable—each time you would like new data, simply invoke the report again and it will overwrite the output file. If you would like to see when the file was last edited, use the command `$ ls -l`. This will help you make sure you are getting up-to-data data.

Go ahead and create (and check) some reports for the the commands you have been using the most.

Aliases

Now that you have some reports added, you may notice that you end up calling some of them in combinations. For example, you might always want to get your updated pump records and your updated cgm records. To do that, you would normally run two commands each time:

```
$ openaps report invoke last_four_pump_hours.json
$ openaps report invoke last_four_cgm_hours.json
```

For this example, we assume that you have added a second report called `last_four_cgm_hours.json` that is similar to the `last_four_pump_hours.json` we walked through previously, except that it is using your `<my_dexcom_name>` device and the `iter_glucose_hours` command. Go ahead and do that so you can follow along.

Calling two sequential commands for each update is a bit annoying, but imagine calling five or ten. Luckily, openaps has a built-in way to group these commands: aliases. Aliases allow generation of single-word commands to invoke a series of reports. For this example, create an alias called `last_four_hours`:

```
$ openaps alias add last_four_hours "report invoke last_four_pump_hours.json
last_four_cgm_hours.json"
```

Go ahead and execute this command:

```
$ openaps last_four_hours
```

You will see that it invokes each of the reports you specified in the order you specified. It prints each step out to the terminal window, and you will find that the corresponding output files have been created.

Just like with devices and reports, the alias is now part of your openaps configuration. You can view all of your aliases with `$ cat openaps.ini` or by using `$ openaps alias show`. Similarly, you can remove aliases with `$ openaps alias remove <alias_name>`.

Aliases are not limited to reports, but we will leave that up to you to explore.

Putting the Pieces Together

Take a moment to consider putting these commands to work in the larger context of a closed-loop system. Components of that system that you might need to `add` and `invoke` would be recent glucose data, recent pump history, the time, battery status, pump settings, carb ratios, the current basal profile, insulin sensitivities, blood glucose targets, and the status of the pump.

Go ahead and add and invoke reports for these components of a future closed-loop system.

Are there groupings of these reports that you imagine would be called at the same time? For example, in a closed-loop setup, the pump settings, blood glucose targets, insulin sensitivities, the basal profile, and carb ratios would not need to be checked as often as the current pump status, battery status, clock, recent blood sugars, and recent pump history.

Take some time to create aliases for groups of reports that would be called at the same time and verify that they invoke the expected reports.

Backing Up Your openaps Instance

There are numerous ways to back up your system, from making a copy of the entire SD card to copying over individual files. Here, we will discuss one method of using git to back up just the openaps instance you've created. Note that this will not back up the entire sysem (and all the work you did in Setting Up the Raspberry Pi 2 and Setting Up openaps), but it will enable you to skip all of the configuration steps above if something happens.

For this backup method, we will take advantage of the fact that your openaps instanace is a git repository. We won't go over git here, but take a look at the references on the Resources page to get familiar with the basics. Typically, we would do this backup using GitHub, since that is where most of the openaps repositories are located and you should already have an account. However, GitHub only provides free repositories if they are public, and since this repository has your `<my_pump_name>.ini` file—and thus your pump's serial number—in it, we want to make sure that is private. If you are comfortable with sharing your glucose and pump history publicly, you can make sure the secret .ini files remain so by creating a .gitignore file listing `<my_pump_name>.ini` (and any other .ini files with secret information). This will prevent git from uploading those files to GitHub, but will still allow you to backup and publicly share all your other configuration and data. Alternatively, you can purchase a monthly GitHub plan and then follow [these instructions] (https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/) if you'd like to go with GitHub, or use a service like Bitbucket instead.

Bitbucket offers a similar service to GitHub, but permits users to create free private repositories. Go ahead and sign up and then create a repository. You can call it whatever you like, but make sure that on the "Create a new repository" setup page you leave the "This is a private repository" box checked. Once created, you will be directed to a "Repository setup" page. Under the "Command line" section, click on the "I have an existing project" option and follow the instructions.

Once you have completed this step, all of the files in your `<my_openaps>` directory will be saved in your online Bitbucket repository. Whenever you would like to update your backup, simply go into your `<my_openaps>` directory and `$ git push`. This process can be automated, but we'll save that for another day.

Phase 2: Build a Manual System

Phase 2 focuses on deploying a suitable algorithm to recommend necessary changes to basal rates. This is essentially a practice closed-loop system, with you completing the loop by manually calculating what you would do in that scenario. This can be performed in real time or by using historical data and making retroactive suggestions. Pay special attention to situations where CGM readings are not smooth (after calibration, with a new sensor, or with errors such as ???) or when there are issues with data connectivity or fidelity. Assume there will be issues with connectivity.

In this section, you'll dive into how to use the oref0 tools and learn about creating a loop and retry logic. oref0—short for "OpenAPS Reference Design 0"—is our first (zero-th) implementation of the OpenAPS Reference Design. It consists of a number of "Lego block" tools that, when combined with the core openaps toolset, create a full closed loop artificial pancreas system—an OpenAPS implementation.

By this stage, you should have already set up your pump and cgm as openaps devices. You will now add the oref0 tools as virtual devices, create openaps reports for commonly used queries and calculations, and add openaps aliases that bring together those reports into higher-level activities. Finally, you can combine those into a single command (or small set of them) that can do everything required to collect data, make a treatment recommendation, and enact it on the pump.

If you haven't already done so, this is also an excellent time to calibrate your inputs, such as insulin sensitivity factor (ISF), carbohydrate ratio (CR), basal rates, et cetera.

Outline of Phase 2

- Upon Completion
  - Have a set of algorithms designed to keep blood glucose in target range with temp basals
  - Have provided patient-specific inputs required for these algorithms
  - Have those algorithms coded so as to take the data collection built in Phase 1 as input
  - Have the code output recommendations as to how it would change temp basal rates
  - Have the ability to utilize code on real-time and historical data to evaluate efficacy
  - Have implemented, tested, and tuned one or more set of algorithms and be able to help others do the same

- Phase Tasks
  - Research the different strategies and tools for single-hormone closed loop systems
  - Complete relevant patient-specific studies to determine key inputs (e.g. carb absorption, although not necessary for OpenAPS, may be helpful)
  - Code one or more of these strategies such that they can accept live and/or historic patient data and output suggestions for temp basal changes. (oref0 is currently popular.)
  - Evaluate the efficacy of these algorithms by testing their suggestions against live and historic data
  - Iterate on the algorithms and their implementation to improve their output
  - Test at least these cases: data corruption, lack of data, lack of connectivity, and other non-ideal operating conditions, and provide error checking. Also, create a Òpreflight checkÓ.
  - Create a loop alias and retry logic
- Community Contributions
  - Provide feedback on efficacy of algorithms
  - Help edit instructions for Phase 2
  - Provide results of testing for comparing and contrasting with others
  - Summarize research findings and synthesize them into jump-off points for others' research

Summary

- Introduction
- Phase 0: General Setup
- Phase 1: Logging, Cleaning, and Analyzing Your Data
- Phase 2: Build a Manual System
  - Using oref0 Tools
  - [Understanding oref0-determine-basal recommendations] (../../docs/Build-manual-system/Understand-determine-basal.md)
  - Creating a Loop and Retry Logic
- Phase 3: Automate Your System
- Phase 4: Iterate and Improve
- Resources
- Glossary

Using oref0 Tools

Add the oref0 Virtual Devices

In Phase 1, you added two physical medical devices to openaps—your pump and your cgm. This was done using the command `$ openaps device add` and then specifying the device name, type, and parameters. OpenAPS tools to gather system profile parameters such as pump settings, calculate the current insulin on board (IOB), and determine if the pump temp basal should be updated or not, are contained in the OpenAPS reference system oref0. Since there is no physical oref0 device, you are essentially adding it to the openaps environment as a virtual device or plugin.

First, you can add a catch-all oref0 device using

```
$ openaps device add oref0 process oref0
```

and then you can be more specific and add individual oref0 processes as virtual devices using the following commands:

```
$ openaps device add get-profile process --require "settings bg_targets insulin_sensitivi
$ openaps device add calculate-iob process --require "pumphistory profile clock" oref0 ca
$ openaps device add determine-basal process --require "iob temp_basal glucose profile" o
```

In these commands, `--require` specifies the arguments required by each of the oref0 processes. Most of the arguments to the oref0 processes should look familiar to you from your experimentation with `openaps` tools earlier. Now it's time to put together reports that the oref0 processes use as inputs, as well as reports and aliases that invoke the oref0 processes themselves.

Organizing the reports

It is convenient to group your reports into `settings`, `monitor`, and `enact` directories. The `settings` directory holds reports you may not need to refresh as frequently as those in `monitor` (e.g. BG targets and basal profile, vs. pump history and calculated IOB). Finally, the `enact` directory can be used to store recommendations ready to be reviewed or enacted (sent to the pump). The rest of this section assumes that you have created `settings` , `monitor` , and `enact` as subdirectories in your openaps directory. The following shell command creates these three directories:

```
$ mkdir -p settings monitor enact
```

The get-profile process

The purpose of the `get-profile` process is to consolidate information from multiple settings reports into a single JSON file. This makes it easier to pass the relevant settings information to oref0 tools in subsequent steps. Let's look at what kind of reports you may want to set up for each of the `get-profile` process arguments:

- `settings` outputs a JSON file containing the pump settings:

  ```
  $ openaps report add settings/settings.json JSON pump read_settings
  ```

- `bg_targets` outputs a JSON file with bg targets collected from the pump:

  ```
  $ openaps report add settings/bg_targets.json JSON pump read_bg_targets
  ```

- `insulin_sensitivities` outputs a JSON file with insulin sensitivites obtained from the pump:

  ```
  $ openaps report add settings/insulin_sensitivities.json JSON pump read_insulin_sensi
  ```

- `basal_profile` outputs a JSON file with the basal rates stored on the pump in your basal profile

  ```
  $ openaps report add settings/basal_profile.json JSON pump read_basal_profile_std
  ```

- `max_iob` is an exception: in contrast to the other settings above, `max_iob` is not the result of an openaps report. It's a JSON file that should contain a single line, such as: `{"max_iob": 2}` . You can create this file by hand, or use the oref0-mint-max-iob tool to generate the file. The max_iob variable represents an upper limit to how much insulin on board oref0 is allowed to contribute by enacting temp basals over a period of time. In the example above, `max_iob` equals 2 units of insulin.

Make sure you test invoking each of these reports as you set them up, and review the corresponding JSON files using `cat`. Once you have a report for each argument required by `get-profile`, you can add a `profile` report:

```
$ openaps report add settings/profile.json text get-profile shell settings/settings.json
```

Note how the `profile` report uses `get-profile` virtual device, with all the required inputs provided. At this point, it's natural to add an alias that generates all the reports required for `get-profile`, and then invokes the `profile` report that calls `get-profile` on them:

```
$ openaps alias add gather-profile "report invoke settings/settings.json settings/bg_targ
```

Remember, what you name things is not important - but remembering WHAT you name each thing and using it consistently throughout is key to saving you a lot of debugging time. Also, note that the name of your report and the name of the corresponding file created by the report are the same. For example, you invoke a report called "settings/settings.json" and the results are stored in "settings/settings.json". The corresponding output file is created by invoking the report.

The calculate-iob process

This process uses pump history and the result of `get-profile` to calculate IOB. The IOB is calculated based on normal boluses and basal rates over past several hours. At this time, extended boluses are not taken into account. The `get-profile` arguments, and suggested reports are as follows:

- `profile`: report for `get-profile`, as discussed above
- `pumphistory` stores pump history in a JSON file

  ```
  $ openaps report add monitor/pumphistory.json JSON pump iter_pump_hours 4
  ```

In this example, pump history is over a period of 4 hours. Normally, you would want oref0 to operate based on pump history over the number of hours at least equal to what you assume is your active insulin time.

- `clock` outputs the current time stamp from the pump

  ```
  $ openaps report add monitor/clock.json JSON pump read_clock
  ```

You can now add a report for the `calculate-iob` process:

```
$ openaps report add monitor/iob.json JSON calculate-iob shell monitor/pumphistory.json s
```

As always, it is a good idea to carefully test and examine the generated reports.

The determine-basal process

This process uses the IOB computed by `calculate-iob` , the current temp basal state, CGM history, and the profile to determine what temp basal to recommend (if any). Its arguments and reports could be setup as follow:

- `iob` : your report for `calculate-iob`
- `profile` : your report for `get-profile`
- `temp_basal` reads from pump and outputs the current temp basal state:

  ```
  $ openaps report add monitor/temp_basal.json JSON pump read_temp_basal
  ```

- `glucose` reads several most recent BG values from CGM and stores them in glucose.json file:

  ```
  $ openaps report add monitor/glucose.json JSON cgm iter_glucose 5
  ```

In this example, glucose.json will contain 5 most recent bg values.

Finally, a report for `determine-basal` may look like this:

```
$ openaps report add enact/suggested.json text determine-basal shell monitor/iob.json mon
```

The report output is in suggested.json file, which includes a recommendation to be enacted by sending, if necessary, a new temp basal to the pump, as well as a reason for the recommendation.

If you are using a Minimed CGM (enlite sensors with glucose values read by your pump), you might get this error message when running this report `Could not determine last BG time` . That is because times are reported differently than from the Dexcom receiver and need to be converted first. See the section at the bottom of this page.

Adding aliases

You may want to add a `monitor-pump` alias to group all the pump-related reports, which should generally be obtained before running `calculate-iob` and `determine-basal` processes:

```
$ openaps alias add monitor-pump "report invoke monitor/clock.json monitor/temp_basal.jso
```

For consistency, you may also want to add a `monitor-cgm` alias. Even though it's invoking only a single report, keeping this consistent with the `monitor-pump` alias makes the system easier to put together and reason about.

```
$ openaps alias add monitor-cgm "report invoke monitor/glucose.json"
```

Checking your reports

At this point you can call

```
$ openaps report show
```

and

```
$ openaps alias show
```

to list all the reports and aliases you've set up so far. You'll want to ensure that you've set up a report for every argument for every oref0 process and, *more importantly*, that you understand what each report and process does. This is an excellent opportunity to make some `openaps report invoke` calls and to `cat` the report files, in order to gain better familiarity with system inputs and outputs.

You can also test the full sequence of aliases and the that which depend on them:

```
$ rm -f settings/* monitor/* enact/*
$ openaps gather-profile
$ openaps monitor-pump
$ openaps monitor-cgm
$ openaps report invoke monitor/iob.json
$ openaps report invoke enact/suggested.json
```

It is particularly important to examine suggested.json, which is the output of the `determine-basal` process, i.e. the output of the calculations performed to determine what temp basal rate, if any, should be enacted. Let's take a look at some suggested.json examples:

```
{"temp": "absolute","bg": 89,"tick": -7,"eventualBG": -56,"snoozeBG": 76,"reason": "Event
```

In this example, the current temporary basal rate type is "absolute", which should always be the case. The current BG values is 89, which dropped from 96 by a "tick" value of -7. "eventualBG" and "snoozeBG" are oref0 variables projecting ultimate bg values based on the current IOB with or without meal bolus contributions, an average change in BG over the most recent CGM data in glucose.json, and your insulin sensitivity. The "reason" indicates why the recommendation is made. In the example shown, "eventualBG" is less than the target BG (100), no temp rate is currently set, and the temp rate required to bring the eventual BG to target is -0.435U/hr. Unfortunately, we do not have glucagon available, and the pump is unable to implement a negative temp basal rate. The system recommends the best it can: set the "rate" to 0 for a "duration" of 30 minutes. In the oref0 algorithm, a new temp basal rate duration is always set to 30 minutes. Let's take a look at another example of `suggested.json` :

```
{"temp": "absolute","bg": 91,"tick": "+6","eventualBG": -2,"snoozeBG": 65,"reason": "Even
```

In this case, the evenatual BG is again less than the target, but BG is increasing (e.g. due to a recent meal). The actual "tick", which is also referred to as "Delta", is larger than the change that would be expected based on the current IOB and the insulin sensitivity. The system therefore recommends canceling the temp basal rate, which is in general done by setting "duration" to 0. Finally, consider this example:

```
{"temp": "absolute","bg": 95,"tick": "+4","eventualBG": 13,"snoozeBG": 67,"reason": "Even
```

which is similar to the previous example except that in this case there is no temp basal rate to cancel. To gain better understanding of oref0 operation, you may want to also read [Understanding oref0-determine-basal recommendations] (Understand-determine-basal.md) and spend some time generating and looking through suggested.json and other reports.

Enacting the suggested action

Based on suggested.json, which is the output of the `determine-basal` oref0 process, the next step is to enact the suggested action, i.e. to send a new temp rate to the pump, to cancel the current temp rate, or do nothing. The approach one may follow is to setup an `enacted.json` report, and a corresponding `enact` alias. Thinking about how to setup the `enact` report and alias, you may consider the following questions:

- Which pump command could be used to enact a new basal temp, if necessary, and what inputs should that command take? Where should these inputs come from?
- How could a decision be made whether a new basal temp should be sent to the pump or not? What should `enact` do in the cases when no new temp basal is suggested?

Once you setup your `enact` alias, you should plan to experiment by running the required sequence of reports and by executing the `enact` alias using `$ openaps enact`. Plan to test and correct your setup until you are ceratin that `enact` works correctly in different situations, including recommendations to update the temp basal, cancel the temp basal, or do nothing.

Adding error checking

Before moving on to consolidating all of these capabilities into a single alias, it is a good idea to add some error checking. There are several potential issues that may adveresely affect operation of the system. For example, RF communication with the pump may be compromised. It has also been observed that the CareLink USB stick may become unresponsive or "dead", requiring a reset of the USB ports. Furthermore, in general, the system should not act on stale data. Let's look at some approaches you may consider to address these issues.

Ensuring that your openaps implementation can't act on stale data could be done by deleting all of the report files in the `monitor` directory before the reports are refreshed. YOu may simply use `rm -f` bash command, which removes file(s), while ignoring cases when the file(s) do not exist. If a refresh fails, the data required for subsequent commands will be missing, and they will fail to run. For example, here is an alias that runs the required bash commands:

```
openaps alias add gather '! bash -c "rm -f monitor/*; openaps gather-profile && openaps m
```

This example also shows how an alias can be constructed using bash commands. First, all files in `monitor` directory are deleted. Then, aliases are executed to generate the required reports. A similar approach can be used to remove any old `suggested.json` output before generating a new one, and to check and make sure oref0 is recommending a temp basal before trying to set one on the pump. You may want to make sure that your `enact` alias includes these provisions.

It's also worthwhile to do a "preflight" check that verifies a pump is in communication range and that the pump stick is functional before trying anything else. The oref0 `mm-stick` command can be used to check the status of the MM CareLink stick. In particular, `mm-stick warmup` scans the USB port and exits with a zero code on success, and non-zero otherwise. Therefore,

```
$ mm-stick warmup || echo FAIL
```

will output "FAIL" if the stick is unresponsive or disconnected. You may simply disconnect the stick and give this a try. If the stick is connected but dead, `oref0-reset-usb` command can be used to reset the USB ports

```
$ sudo oref0-reset-usb
```

Beware, this command power cycles all USB ports, so you will temporarily loose connection to a WiFi stick and any other connected USB device.

Checking for RF connectivity with the pump can be performed by attempting a simple pump command or report and by examining the output. For example,

```
$ openaps report invoke monitor/clock.json
```

returns the current pump time stamp, such as "2016-01-09T10:47:56", if the system is able to communicate with the pump, or errors otherwise. Removing previously generated clock.json and checking for presence of "T" in the newly created clock.json could be used to verify connectivity with the pump.

Collecting all the error checking, a `preflight` alias could be defined as follows:

```
$ openaps alias add preflight '! bash -c "rm -f monitor/clock.json && openaps report invo
```

In this `preflight` example, a wait period of 120 seconds is added using `sleep` bash command if the USB ports have been reset in an attempt to revive the MM CareLink stick. This `preflight` example also shows how bash commands can be chained together with the bash && ("and") or || ("or") operators to execute different subsequent commands depending on the output code of a previous command (interpreted as "true" or "false").

You may experiment using `$ openaps preflight` under different conditions, e.g. with the CareLink stick connected or not, or with the pump close enough or too far away from the stick.

At this point you are in position to put all the required reports and actions into a single alias.

Cleaning CGM data from Minimed CGM systems

If you are using the Minimed Enlite system, then your report for `iter_glucose` uses your pump device because the pump is the source of your CGM data. Unfortunately, the pump reports CGM data a bit differently and so your glucose.json file needs cleaning to align it with Dexcom CGM data. The simplest way to handle this is with this excellent plug-in:

https://github.com/loudnate/openaps-glucosetools

After installing glucosetools, add the vendor and device with:

```
$ openaps vendor add openapscontrib.glucosetools
$ openaps device add glucose glucosetools
```

Now you can create a report to clean your glucose data like this:

```
openaps report add monitor/glucoseclean.json JSON glucose clean monitor/glucose.json
```

And you should then make sure that your enact/suggested.json report uses monitor/glucoseclean.json instead of monitor/glucose.json. You can add the `clean` report to your `monitor-cgm` alias, as long as it comes after the `iter_glucose` report.

Note that if you use Nightscout visualization as described later, you can use the built-in tool `mm-format-ns-glucose` to help formatting the Minimed glucose data. If you do, run the tool against the original `iter-glucose` output (monitor/glucose.json), *not* the output from glucosetools.

Creating a loop and retry logic

To pull all of oref0 together, you could create a "loop" alias that looks something like `openaps alias add loop '! bash -c "openaps monitor-cgm 2>/dev/null && ( openaps preflight && openaps gather && openaps enact) || echo No CGM data."'` . If you want to also add some retry logic to try again if something failed, you could then do something like `openaps alias add retry-loop '! bash -c "until( ! mm-stick warmup || openaps loop); do sleep 5; done"'` .

Once all that is working and tested, you will have a command that can be run manually or on a schedule to collect data from the pump and cgm, calculate IOB and a temp basal suggestion, and then enact that on the pump.

Phase 3: Automate Your System

Phase 3 focuses on creating a schedule to automate the manual system you developed in Phase 2. Again, at this stage testing is critical and output of the system should be tracked and validated over a series of time, and include thorough edge case testing as you move from communicating with the pump and CGM to closing the loop. If you haven't already, it's helpful to consider this experience as a set of experiments. For experimenting, you'll need to have plenty of time (and patience) to track what the loop is doing; what you think it should do; and what is needed to bring your BG up/down and decide if you need to intervene.

At this stage, you should have a suitable algorithm to manually recommend necessary changes to basal rates that you have tested thoroughly. That was essentially a practice closed-loop system, with you completing the loop by manually calculating what you would do in that scenario. Now, you're ready to automate your loop. This section focuses on creating a schedule to collect data from the pump and cgm, calculate IOB and a temp basal suggestion, and then enact that on the pump. Again, at this stage testing is critical and output of the system should be tracked and validated over a series of time, and include thorough edge case testing to ensure that the loop is working, the schedule is as designed, and that you can quick-check when the system is running and trouble shoot any runtime challenges. You will likely also want to visualize what the system is doing.

Once you get the loop automated, you should set an alarm and check your glucose levels every 15-30 minutes and continue to watch the system closely. You should not test overnight until you are supremely confident in the operation of the system! And you should probably have someone committed to watching your response while you sleep, and alarms to ensure that everything goes as planned.

```
* Upon Completion
    * Have a schedule to run a set of algorithms designed to keep blood glucose in target
    * Have repeatedly validated the output of your scheduled work  and understand how to

* Phase Tasks
    * Create schedule to collect & validate input data, calculate IOB, generate temp basa
    * Carefully analyze output and outcomes from the scheduled system over a series of ti
    * Test at least these cases: data corruption, lack of data, lack of connectivity, and

* Community Contributions
    * Provide feedback on efficacy of algorithms
    * Help edit instructions for Phase 3
    * Provide results of testing (including start date for first overnight evaluation, pl
    * Summarize research findings and synthesize them into jump-off points for others' re
```

Summary

- Introduction
- Phase 0: General Setup
- Phase 1: Logging, Cleaning, and Analyzing Your Data
- Phase 2: Build a Manual System
- Phase 3: Automate Your System
    - Creating a Schedule
    - Visualization and Monitoring
    - Validating and Testing
    - Keeping up to date
- Phase 4: Iterate and Improve
- Resources

Using cron to create a schedule for your loop

You should use cron to create a schedule for your loop.

Visualization and Monitoring

Nightscout Integration

Integrating OpenAPS with Nightscout is a very helpul way to visualize what OpenAPS is doing using a web browser or an app on a mobile device, as opposed to logging into your Raspberry Pi and looking through the logs. The integration requires setting up Nightscout and making changes and additions to your OpenAPS implementation.

Nightscout Setup

OpenAPS requires the latest (currently dev) version of Nighthscout, which can be found here: https://github.com/nightscout/cgm-remote-monitor/tree/dev.

Note: currently there is a bug in the dev version, which doesn't allow you to set up a new profile using the profile editor. If you are starting a fresh install of Nightscout, you should first deploy the master version of the code. Once the master version is up an running, you can create your profile with information on basal rates, etc. After that, you can deploy the dev version. If you have an existing version of Nightscout, then make sure you create your profile before moving to the dev version. Or, you may keep your existing Nightscout as is, and start a new Nightscout deployment (master first, followed by dev), specifically to test OpenAPS integration.

The steps discussed here are essantially the same for both Azure and Heroku users. Two configuration changes must be made to the Nightscout implementation:

- Add "openaps" (without the quotes) and, optionally, "pump" (without the quotes) to the list of plugins enabled, and
- Add a new configuration variable DEVICESTATUS_ADVANCED="true" (without the quotes)

For Azure users, here is what these configuration changes will look like (with just "openaps" added): https://files.gitter.im/eyim/lw6x/blob. For Heroku users, exactly the same changes should be made on the Config Vars page. The optional "pump" plugin enables additional pump monitoring pill boxes. For example, assuming you have added "pump" to the list of enabled plugins, you may add a new configuration variable PUMP_FIELDS="reservoir battery" to display pump reservoir and battery status on the Nightscout page. The "pump" plugin offers a number of other options, as documented on the Nightscout readme page: https://github.com/nightscout/cgm-remote-monitor/blob/dev/README.md#built-inexample-plugins

Next, on your Nightscout website, go to the Settings (3 horizontal bars) in the upper right corner. At the very bottom of the Settings menu, in the "About" section, you may check the Nightscout version (e.g. version 0.9.0-dev). Just above is a list of Plugins available. OpenAPS should show up. Click the check box to enable. Similarly, in the case you've enabled the "pump" plugin, "Pump" should also show up in the list, and you may chekc the box to enable. You should now see the OpenAPS pill box (and any optional pump monitoring pill boxes) on the left side of the Nightscout page near the time. You may also want to graphically show the basal rates: select "Default" or "Icicle" from the "Render Basal" pull-down menu in the Settings.

Environment Variables for OpenAPS Access to Nightscout

To be able to upload data, OpenAPS needs to know the URL for your Nightscout website and the hashed version of your API_SECRET password, which you have entered as one of your Nightscout configuration variables. Two environment variables, NIGHTSCOUT_HOST and API_SECRET, are used to store the website address and the password, respectively.

To obtain the hashed version of the API_SECRET, go to http://www.sha1-online.com/ (keep the default sha-1) and hash your API_SECRET. For example, if your enter "password" (without quotes), the hashed version returned will be 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8.

In your Raspberry PI terminal window, you may now define the two environment variables as follows:

```
$ export  NIGHTSCOUT_HOST="https://<your Nightscout address>"
$ export  API_SECRET="<your hashed password>"
```

These variables will stay defined as long as the current terminal session remains active. Note that it is important to enter https:// (not http://) in front of your Nightscout address. To have these variables defined each time you login, you may include the two `export` lines (without quotes) in the .profile file, which is in your home directory.

Configuring and Uploading OpenAPS Status

Integration with Nightscout requires couple of changes to your OpenAPS implementation, which include:

- Adding a new `ns-status` device, and generating a new report `monitor/upload-status.json`, which consolidates the current OpenAPS status to be uploaded to Nightscout
- Uploading the status report to Nightscount, using the `ns-upload` command

Upon successful completion of these two steps, you will be able to see the current OpenAPS status by hovering over the OpenAPS pill box on your Nightscount page, as shown here, for example: https://files.gitter.im/eyim/J8OR/blob

The `ns-status` is a virtual device in the oref0 system, which consolidates OpenAPS status info in a form suitable for upload to Nightscout. First, add the device:

```
$ openaps device add ns-status process --require "clock iob suggested enacted battery res
```

The corresponding entry in your openaps.ini file should look like this:

[device "ns-status"]
fields = clock iob suggested enacted battery reservoir status
cmd = ns-status
vendor = openaps.vendors.process
args =

Then use the `ns-status` device to add the `monitor/upload-status.json` report, which you may do as follows:

```
$ openaps report add monitor/upload-status.json JSON ns-status shell monitor/clock-zoned.
```

The reports required to generate upload-status.json should look familiar. If you have not generated any of these required reports, you should set them up and make sure they all work. In particular, note that monitor/clock-zoned.json contains the current pump clock time stamp, but with the timezone info included. If you have not generated that report already, you may do so using the following commands, which add a `tz` virtual device and use it to create clock-zoned.json starting from clock.json.

```
$ openaps vendor add openapscontrib.timezones
$ openaps device add tz timezones
$ git add tz.ini
$ openaps report add monitor/clock-zoned.json JSON tz clock monitor/clock.json
```

At this point, you may want to update your monitor-pump alias to make sure that it produces all the required reports, so that uploading status to Nightscount can be automated. After you've generated a monitor/upload-status.json report, you can try to manually upload the OpenAPS status to Nightscout using the `ns-upload` command:

```
$ ns-upload $NIGHTSCOUT_HOST $API_SECRET devicestatus.json monitor/upload-status.json
```

If successful, this command will POST the status info to your Nightscout site, and return the content of the Nightscout devicestatus.json file, which you can examine to see what status information is sent to Nightscout.

Finally, you may define a new alias `status-upload`, to combine generating the report and uploading the status to Nightscout:

```
$ openaps alias add status-upload '! bash -c "openaps report invoke monitor/upload-status
```

To test this alias, you may first run your loop manually from command line, then execute `openaps status-upload`, examine the output, and check that the new status is visible on the OpenAPS pill box on your Nightscout page. To automate the status upload each time the loop is executed you can simply add `status-upload` to your main OpenAPS loop alias. The OpenAPS pill box will show when the last time your loop ran. If you hover over it, it will provide critical information that was used in the loop, which will help you understand what the loop is currently doing.

The OpenAPS pill box has four states, based on what happened in the last 15 minutes: Enacted, Looping, Waiting, and Warning:

- Waiting is when OpenAPS is uploading, but hasn't seen the pump in a while
- Warning is when there hasn't been a status upload in the last 15 minutes
- Enacted means OpenAPS has recently enacted the pump
- Looping means OpenAPS is running but has not enacted the pump

Some things to be aware of:

- Make sure that the timezones for the pi (if need be you can use `sudo raspi-config` to change timezones), in your monitor/clock-zoned.json report, and the Nightscout website are all in the same time zone.
- The basal changes won't appear in Nightscout until the second time the loop runs and the corresponding upload is made.
- You can scroll back in time and at each glucose data point you can see what the critical information was at that time

Uploading Latest Treatments to Nightscout

In addition to uloading OpenAPS status, it also very beneficial to upload the treatment information from the pump into Nightscout. This removes the burden of entering this information into Nightscout manually. This can be accomplished using `nightscout` command and adding a new `upload-recent-treatments` alias as follows:

```
$ openaps alias add latest-ns-treatment-time '! bash -c "nightscout latest-openaps-treatm
$ openaps alias add format-latest-nightscout-treatments '! bash -c "nightscout cull-lates
$ openaps alias add upload-recent-treatments '! bash -c "openaps format-latest-nightscout
```

Note that a pumphistory-zoned.json report is required, which can be generated from pumphistory.json using `tz`, following the approach described above for clock-zoned.json. After running your loop from command line, you may try executing `openaps upload-recent-treatments` manually from command line. Upon successful upload, the recent treatments will show up automatically on the Nightscount page.

Note: Currently extended boluses are not handled well and depending on the timing of the upload are either missed entirely or have incorrect information.

As a final step in the OpenAPS and Nightscout integration, you may add `status-upload` and `upload-recent-treatments` to your main loop, and automate the process using cron. Make sure you include the definitions of the environment variables NIGHTSCOUT_HOST and API_SECRET in the top part of your crontab file, without `export`, or quotes:

```
NIGHTSCOUT_HOST=https://<your Nightscout address>
API_SECRET=<your hashed password>
```

Validating and Testing

If you haven't read this enough already: the DIY part of this is really important. If you've been copying and pasting, and not understanding what you're doing up to this point - please stop. That's dangerous. You should be testing and validating your work, and asking questions as you go if anything is unclear. (And, if this documentation annoys you enough, put in a PR as you go through each part to update/improve the documentation to help the next person! We've all been there.) :)

That being said, at this stage, you have both a manual loop and a schedule (using cron) to create an automated loop. At this point, you're in the "test and watch" phase. In particular, you should make sure the loop is recommending and enacting the types of temporary basal rates that you might do manually; and that the communication is working between the devices.

Additionally, most loopers, after automating their system with cron jobs, begin to think through some of the following things and run the following tests, including unit tests:

How often should the cron run?

Think about how often you get new BG data and may want to act on it. Or, time how long it takes your loop to run, and add another minute to that. You probably want to add preflight checks to make sure a loop is not already running before your next one starts.

What should BG target range be?

In the early testing, the OpenAPS settings may cause your BG to go both high and low.

It's tempting to set your targets to "perfect" on day one, and start your looping with those values. The problem with this is that if the algorithm incorrectly gives you too much insulin you don't have very much room to handle emergencies.

To start off, you should set your glucose target range "high and wide". Once you can reproducibly get your sugars in a wider and higher band without going low, you can then *slowly* reduce the target range to your ideal range.

You should work toward a long-term goal here, rather than trying to do everything on day one.

Additionally, as you think about the lower end of your target range, remember the timing of your insulin activity and the fact that negative insulin corrections take about the same amount of time to go into effect; thus, you wouldn't want your low end of the target range set below 90, for example - otherwise the system will not be able to prevent lows by reducing the insulin.

What should pump settings be?

You should set your maximum temporary basal limit on your pump to a reasonable value, to try and make sure that you don't go low by accident.

To start off, you should think about taking the largest basal rate in your profile and multiply it by a 1.3. Set that as your maximum temporary basal on the pump.

Once you're happy things are functioning correctly, you can increase this value to about 2x your basal.

Note that for children especially this can vary a lot based on age, weight, and activity. Err on the side of caution.

What happens if the system gets out of range or gets bad data?

Test the range of the system. What happens if you walk out of range of the Carelink stick? What happens to the temporary basal rate? What happens with your cron job? What do you need to be aware of? Apply the same set of questions and thinking for other scenarios, including if you go out of range of your CGM and/or get ??? or another CGM error message.

Make sure you understand the limits of the transmitter and these other errors, especially in an overnight situation , and what the system can and can't do when you are out of effective range.

As your tests extends from minutes to hours, you'll want to check at least these scenarios: data corruption, lack of data, lack of connectivity, and other non-ideal operating conditions.

Unit testing

Additionally, you may want to consider some unit testing. There is a basic unit testing framework in oref0 that you can use, and add to.

To help with unit test cases:

If you'd like to help out with defining all the desired behaviors in the form of unit test cases:

1) Please clone / checkout [oref0] (https://github.com/openaps/oref0)

2) Type `sudo npm install -g mocha` and `sudo npm install -g should`

3) You should then be able to run `make` (or something like `mocha -c tests/determine-basal.test.js 2>&1 | less -r`) from the openaps-js directory to run all of the existing unit tests

4) As you add additional unit tests, you'll want to run `make` again after each one.

How to add more test cases:

We'll want to cover every section of the code, so if you see a "rT.reason" in bin/oref0-determine-basal.js that doesn't have a corresponding "output.reason.should.match" line in an appropriate test in tests/determine-basal.test.js, then you should figure out what glucose, temp basal, IOB, and profile inputs would get you into that section of the code (preferably representing what you're likely to see in a real-world situation), and create a test case to capture those inputs and the desired outputs. Then run the tests and see if your test passes, and the output looks reasonable. If not, then modify your test case accordingly, or if you think you've found a bug in determine-basal.js, ask on Gitter.

Keeping up to date

If you've gone "live" with your loop, congratulations! You'll probably want to keep a very close eye on the system and validate the outputs for a while. (For every person, this amount of time varies).

One important final step, in addition to continuing to keep an eye on your system, is letting us know that you are looping.

This is important in case there are any major changes to the system that we need to notify you about. One example where this was necessary is when we switched from 2015 to 2016: the dates were incorrectly reporting as 2000, resulting in incorrect IOB calculations. As a result, we needed to notify current loopers so they could make the necessary update/upgrade.

So that we can notify you if necessary, please fill out this form if you have been looping for 3+ days. Your information will not be shared in any way. You can indicate your preferred privacy levels in the form. As an alternative, if you do not want to input info, please email dana@openaps.org. Again, this is so you can be notified in the case of a major bug find/fix that needs to be deployed.

Iterating and Improving

At this point, you're probably familiar enough to understand some of the limitations or usability frustrations of this DIY system. (You get one point for having your own complaint, one point for hearing Dana complain about "frying a Pi", and one point to hear someone talking about using an Edison or a "RileyLink" or another tool to cut down on the size of their openAPS implementation.) The main point here is that this is still not a set-and-forget system, and there's lots to improve on. This page will serve as a landing page to point out to various other projects, or notes on how people might be improving the system.

Using other pumps

There's a group trying to figure out the Omnipod communication as well as Animas communication. There is an omnidocs repository that they may use to share their work, but they're more frequently chatting in a Slack channel. Click here to join that collaboration channel and check in on their progress.

Using other devices instead of a raspberry pi

- RileyLink - check out this repo or join the gitter channel. RileyLink is custom designed Bluetooth Smart (BLE) to 916MHz module. It can be used to bridge any BLE capable smartphone to the world of 916Mhz based devices. This project is focused on talking to Medtronic insulin pumps and sensors.
- Edison - There is also work to see if an Edison can be used instead of a Raspberry Pi.
- HAPP - Tim Omer has put together an Android-based app that is a manual temp-basal recommendation engine, based off the OpenAPS reference design. It's not a closed loop, but his work could be used to create an Android-driven loop implementation. Check out HAPP on Github here.

Summary

- Introduction
- Phase 0: General Setup
- Phase 1: Logging, Cleaning, and Analyzing Your Data
- Phase 2: Build a Manual System
- Phase 3: Automate Your System

- Phase 4: Iterate and Improve
- Resources

Resources

See the subsections for:

- Links to technical documentation
- Helpful troubleshooting commands
- A primer on the history of OpenAPS
- Other interesting projects
- OpenAPS contributors sharing their work publicly
- FAQs

Summary

- Introduction
- Phase 0: General Setup
- Phase 1: Logging, Cleaning, and Analyzing Your Data
- Phase 2: Build a Manual System
- Phase 3: Automate Your System
- Phase 4: Iterate and Improve
- Resources
    - Technical Resources
    - Troubleshooting
    - #OpenAPS Overview and Project History
    - Other Projects, People & Tools
    - FAQs
    - Glossary

Technical Resources

These represent a small selection of guides, tutorials, and quick references for some of the tools used to develop and document OpenAPS.

Raspberry Pi

Raspberry Pi Documentation

Pi Filler, Pi Finder, and Pi Copier

Git and GitHub

Official Git Documentation

Atlassian (BitBucket) Git Tutorials

Code School Interactive Git Introduction

Codecademy Git Course

Linux Shell / Terminal

Learn UNIX in 10 Minutes

Codecademy Command Line Course

Cron How To Guide

Python

Official Python Documentation

Learn Python the Hard Way

Automate the Boring Stuff with Python

Codecademy Python Course

Python 2.7 Quick Reference

NumPy for MATLAB Users

Useful Apps

Fing (Android and Apple): Identify IP address of devices on a network. Useful for finding the IP address of RPi on new networks.

Hotspot Manager (Android): Itentify IP address of devices on a hotspot. Useful for finding the IP address of RPi on hotspots.

JuiceSSH (Android): SSH client for Android devices

Markdown & GitBook

Daring Fireball (John Gruber) Markdown Introduction

Macdown: Open-source Markdown editor for OS X

StackEdit: Online Markdown editor

GitBook editor

GitBook Help

Troubleshooting

Even those who follow this documentation precisely are bound to end up stuck at some point. This could be due to something unique to your system, a mistyped command, actions performed out of order, or even a typo in this guide. This section provides some tools to help diagnose the issue as well as some common errors that have been experienced and resolved before. If you get stuck, try re-reading the documentation again and after that, share what you've been working on, attempted steps to resolve, and other pertinent details in #intend-to-bolus in Gitter when asking for help troubleshooting.

Generally useful linux commands

More comprehensive command line references can be found here and here. For the below, since these are basic linux things, also try using a basic search engine (i.e. Google) to learn more about them and their intended use.

`$ ls -alt` (List all of the files in the current directory with additional details.)

`$ cd` (Change directory)

`$ pwd` (Show the present working directory (your current location within the filesystem).)

`$ sudo <command>`

`$ tail -f /var/log/syslog`

`$ df -h`

`$ ifconfig`

`$ cat <filename>` (Display the contents of the file.)

`$ nano <filename>` (Open and edit the file in the nano text editor.)

`$ stat <filename>`

`$ pip freeze`

`$ sudo reboot`

`$ sudo shutdown -h now` (The correct way to shut down the Raspberry Pi from the command line. Wait for the green light to stop blinking before removing the power supply.)

`$ dmesg` (Displays all the kernel output since boot. It’s pretty difficult to read, but sometimes you see things in there about the wifi getting disconnected and so forth.)

`uptime`

[add something for decocare raw logging]

Dealing with the CareLink USB Stick

The `model` command is a quick way to verify whether you can communicate with the pump. Test this with `$ openaps use <my_pump_name> model`.

If you can't get a response, it may be a range issue. The range of the CareLink radio is not particularly good, and orientation matters; see range testing report for more information.

If you still can't get a response, trying unplugging and replugging the CareLink stick.

Once you're setting up your loop, you may also want to oref0-reset-usb (`oref0-reset-usb.sh`) if mm-stick warmup fails, to reset the USB connection. It can help in some cases of CareLink stick not responding. Just note that during USB reset you will loose your Wi-Fi connection as well.

Dealing with a corrupted git repository

OpenAPS uses git as the logging mechanism, so it commits report changes on each report invoke. Sometimes, due to "unexpected" power-offs (battery dying, unplugging, etc.),the git repository gets broken. When it happens you will receive exceptions when running any report from openaps. As git logging is a safety/security measure, there is no way of disabling these commits.

To fix a corrupted git repository you can run `oref0-fix-git-corruption.sh`, it will try to fix the repository, and in case when repository is definitly broken it copies the remainings in a safe place (`tmp`) and initializes a new git repo.

Environment variables

If you are getting your BG from Nightscout or you want to upload loop status/resuts to Nightscout, among other things you'll need to set 2 environment variabled: `NIGHTSCOUT_HOST` and `API_SECRET`. If you do not set and export these variables you will receive errors while running `openaps report invoke monitor/ns-glucose.json` and while executing `ns-upload.sh` script which is most probably part of your `upload-recent-treatments` alias.Make sure your `API_SECRET` is in hashed format. Please see this page for details. Additionally, your `NIGHTSCOUT_HOST` should be in a format like `http://yourname.herokuapp.com` (without trailing slash). For the complete visualization guide use this page from the OopenAPS documentation.

Project History

In order to relieve the incredible burden of T1D, many research teams and manufacturers have developed and are testing Artificial Pancreas Systems (APSs) that connect CGMs to insulin pumps and use various algorithms to automatically adjust insulin dosing (and sometimes dose glucagon, a counter-regulatory hormone) to attempt to mimic some of the functions of a healthy pancreas, and keep blood sugar levels in a safe range. While quite successful in clinical trials so far, current APS systems have been in development for many years, and are still likely at least 3 years away from FDA approval. It is also unclear whether first-generation APS technology will be suitable for, or available to, all patients, even in rich countries.

To address some of the challenges of daily life with diabetes, and because #WeAreNotWaiting, several people worked to figure out how to connect up existing FDA-approved medical devices such as the Dexcom G4 CGM and the Medtronic Minimed insulin pump, using commodity computer / mobile phone hardware and open-source software, to create a complete closed loop Artificial Pancreas System (APS). The first public example of this was the #DIYPS closed loop system, created in their spare time by @DanaMLewis and @ScottLeibrand in the fall of 2013 based on their earlier work to build the #DIYPS remote monitoring and decision assist system. #DIYPS used the Nightscout project's uploader to get Dexcom CGM data off the device. #DIYPS was able to become a closed loop with the help of open-source decoding-carelink project created by @Ben West to communicate with Medtronic insulin pumps, retrieve data and issue insulin-dosing commands to pumps that support it. #DIYPS was the base system that led to #OpenAPS.

In light of the success of #DIYPS closed loop and other simple APS systems built by individuals, Dana and Scott decided to further apply the #WeAreNotWaiting ethos to APS research, believing safe and effective APS technology can be made available more quickly and to more people, rather than just waiting for current APS efforts to complete clinical trials and be FDA-approved and commercialized through traditional processes.

#OpenAPS is an open reference design for, and will be a reference implementation of, an overnight closed loop APS system that uses the CGM sensors' estimate of blood glucose (BG) to automatically adjust basal insulin levels, in order to keep BG levels inside a safe range overnight and between meals.

#OpenAPS is not intended to be a "set and forget" APS system. To maximize safety, a system designed from OpenAPS only doses basal insulin. Users still need to bolus for meals as they do today. However, OpenAPS can identify deviations from predicted blood sugar changes and change basal rates to prevent dangerous drops or rises that deviate from expected behavior.

After launching in early 2015, there are at least 15 known instances of OpenAPS that are live and running (as of 3 November 15), with several others in development and testing phases. For anecdotal experiences from those running OpenAPS, watch the #OpenAPS hashtag on Twitter and also check out the Resources section for a list of those sharing their experiences publicly.

Other People, Projects & Tools

People

These people have publicly identified as either supporting or running OpenAPS implementations and are sharing their experiences publicly. See below links.

Dana Lewis - blogs about personal experience at DIYPS.org and shares on Twitter as @DanaMLewis.

Scott Leibrand - contributes to Dana's instance of OpenAPS, and is on Twitter as @ScottLeibrand. (Scott and Dana collectively maintain OpenAPS.org.)

Ali Mazaheri - shares on Twitter as @AliMazaheri

Chris Hannemann - shares on Twitter as @hannemannemann

Nate Racklyeft - shares on Twitter as @LoudNate

Ben West - author of decoding-carelink and much of the openaps toolkit - on Twitter as @bewestisdoing

The following provide links to other related projects as well as commercial artificial pancreas work underway.

APS & Diabetes Data Tools

- **#DIYPS** (http://diyps.org/) - the project and personal experience that inspired #OpenAPS
- **simPancreas** (http://bustavo.com/category/simpancreas/) - another DIY closed loop, although not #OpenAPS related
- **NightScout** (http://www.nightscout.info/) - a visualization and remote monitoring tool for people with diabetes using CGM
- **xDrip** (http://stephenblackwasalreadytaken.github.io/xDrip/) - a DIY combination of a device and a software application which receives data sent out by a Dexcom G4 CGM transmitter/sensor and displays the glucose readings on an Android phone

- **RileyLink** (https://github.com/ps2/rileylink)
  A custom designed Bluetooth Smart (BLE) to 916MHz module. It can be used to bridge any BLE capable smartphone to the world of 916Mhz based devices. This project is focused on talking to Medtronic insulin pumps and sensors. There is also a Gitter channel dedicated to discussion on the RileyLink here.
- **Tidepool** (http://tidepool.org/ and https://github.com/tidepool-org)
  Notably, work on Boston University iLet UI (https://github.com/tidepool-org/bionicpancreas) and open-source tools for visualization.
- **Perceptus** (http://perceptus.org) - more data visualization tools

Commercial APS Efforts

There are currently several commercial closed-loop products in development by old and new companies in the diabetes treatment space. These include:

- Medtronic MiniMed 640G
- Medtronic MiniMed 670G
- TypeZero Technologies
- Bigfoot Biomedical
- Boston University's iLet, formerly known as the "Bionic Pancreas"

Frequently Asked Questions

What is a Closed Loop?

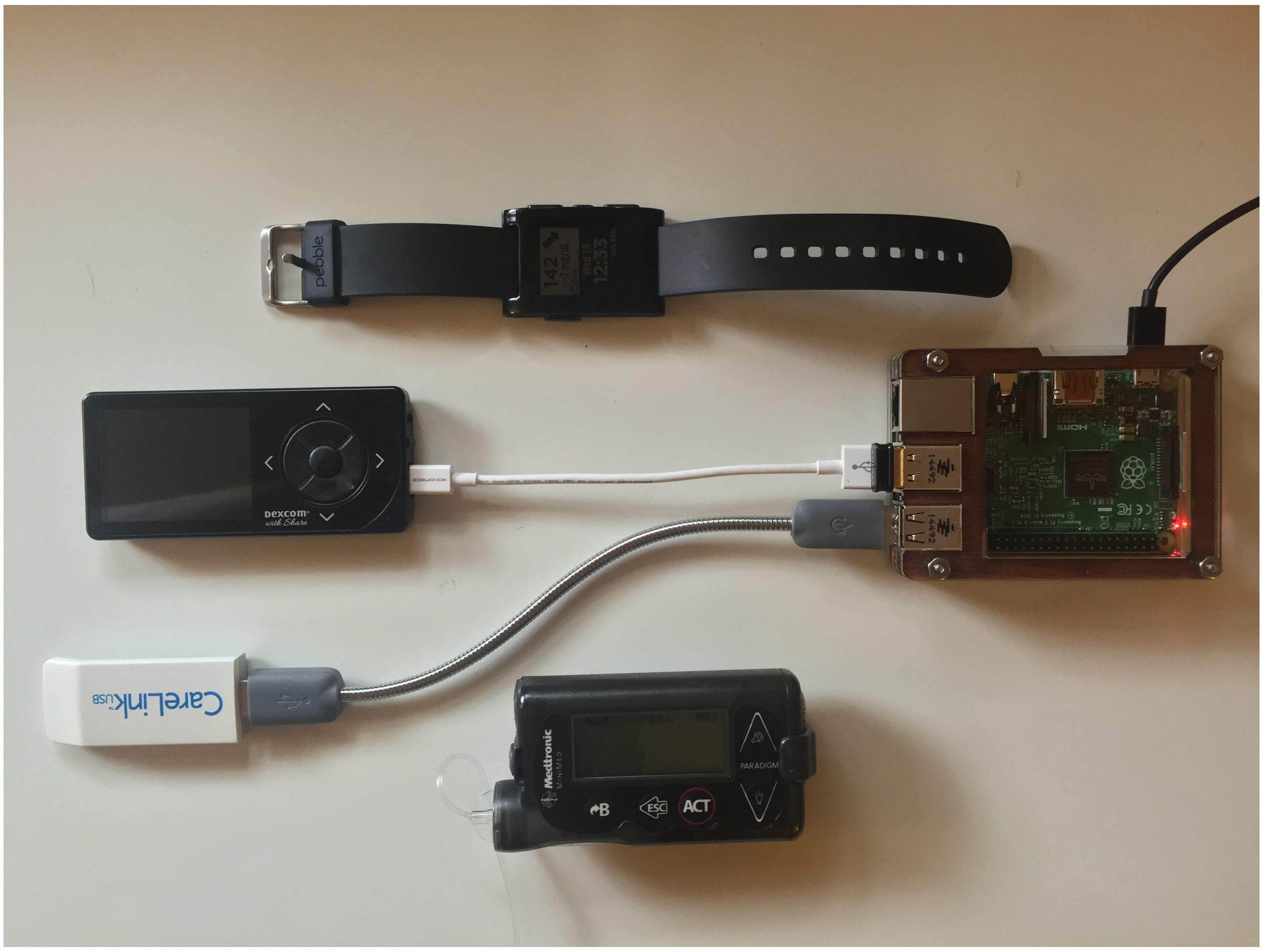
In general, a "closed loop" system for treatment of diabetes is considered to be one in which insulin dosingâ€”and in some cases glucagon dosingâ€”is partially or completely automated. This is in contrast to an "open loop" system, where the user evaluates the inputs and manually instructs the insulin pump to dose a specific amount. In both cases, the goal is to maintain blood glucose within the desired range through adjusting hormone doses.

There are numerous different types of closed loop systems, ranging from simple basal suspend systems designed to mitigate extreme hypoglycemia to dual hormone, fully automated systems. The JDRF Artificial Pancreas Project Plan page provides an overview of the current commercial and academic generation-based approach. Several commercial systems are currently in development; see Commercial APS Efforts for more information.

#OpenAPS is focused on a single-hormone hybrid closed-loop system. This is a system that uses only insulin (no glucagon) and still requires user input for mealtime insulin. For background on #OpenAPS, review the #OpenAPS Reference Design page.

What does an OpenAPS closed loop look like?

While there are numerous variations, this particular setup shows the key components—namely, a continuous glucose monitor, an insulin pump, a method for communicating with the pump (here, a CareLink USB stick), and a controller (here, a Raspberry Pi). Also shown is a Pebble watch, which can be used for monitoring the status of the OpenAPS. Not shown is the power supply (off-screen) and a way to interact with and program the Raspberry Pi, typically a computer or smartphone.



For more details on the exact hardware required to build an OpenAPS, see the Hardware section.

Glossary

APS - artificial pancreas system. Sometimes also referred to as "AP"

CGM - continuous glucose monitor, a temporary glucose sensor that is injected into your skin (the needle is removed) for 3-7 days and, with twice a day calibrations, provides BG readings approximately every 5 minutes.

#OpenAPS - stands for Open A(rtificial) P(ancreas) S(ystem). It is an open-source movement to develop an artificial pancreas using commercial medical devices, a few pieces of inexpensive hardware, and freely-available software. A full description of the #OpenAPS project can be found at openaps.org. #OpenAPS (with the hashtag) generally refers to the broad project and open source movement.

OpenAPS - refers to an example build of the system when used without a hashtag (#)

openaps - the core suite of software tools under development by this community for use in an OpenAPS implementation

Bolus - extra insulin given by a pump, usually to correct for a high BG or for carbohydrates

Basal - baseline insulin level that is pre-programmed into your pump and mimics the insulin your pancreas would give throughout the day and night

IOB - Insulin On Board, or insulin active in your body. Note that most commercially available pumps calculate IOB based on bolus activity only. An OpenAPS implementation calculates and refers most often to net IOB, which takes into account any adjusted (higher or lower) basal rates as well as bolus activity.

DIA - duration of insulin action, or how long the insulin is active in your body. (Ranges 3-6 hours typically)

CR - carb ratio, or carbohydrate ratio - the amount of carbohydrates for one unit of insulin. Example: 1 u of insulin for 10 carbs

ISF - insulin sensitivity factor - the amount of insulin that drops your BG by a certain amount mg/dl. Example: 1 u of insulin for 40 mg/dl

NS, or Nightscout - a cloud-based visualization and remote-monitoring tool.