Vanderbilt University

School of Engineering

ECE 4375 - Embedded Systems

User and Technical Documentation

DeliverPi: An Automated Warehouse Package Picking System

April 30, 2025

Ben Mongirdas, Daniil Shatokhin

Contents

User Manual

Introduction 3

Environment 4

Hiwonder TurboPi Robot 5

Getting Started 6

Troubleshooting 8

Technical Documentation

Introduction 9

Networking 9

Controller 11

User Manual

Introduction

Welcome to DeliverPi! This is a project undertaken by Ben Mongirdas and Daniil Shatokhin for ECE4375 - Embedded Systems at Vanderbilt University. The goal of DeliverPi is to provide software that runs on the Hiwonder TurboPi platform that accurately simulates the warehouse picking process. For those unfamiliar, picking is the process of retrieving the item(s) within an order from storage and delivering them to the correct destination for further processing. This process is quite monotonous and prone to human error if you are not careful, and thus it is one of the first targets of warehouse automation, evidenced by Amazon's and other large retailers' swift adoption of warehouse robots.

DeliverPi is organized into three main programs - the central server, the picking hub, and the robot controller. All three are command line Python programs. The central server is a stand-in for an already existing order database and simply distributes orders to the picking hub. The picking hub receives orders from the central server and then distributes these orders to the robot controller based on their deadlines. The robot controller receives orders from the picking hub, finds the optimal picking route, and then controls all robot movement and action, collecting packaging and publishing status updates to the picking hub along the way while ensuring to not run into obstacles.

We hope that the DeliverPi platform can be a jumping-off point for supply chain engineers to explore how an automated picking system may function and how it can integrate with and help streamline their warehouse operations. We recommend viewing the demo video located in the GitHub repository at https://github.com/bmongird/DeliverPi to see DeliverPi in action!

Environment

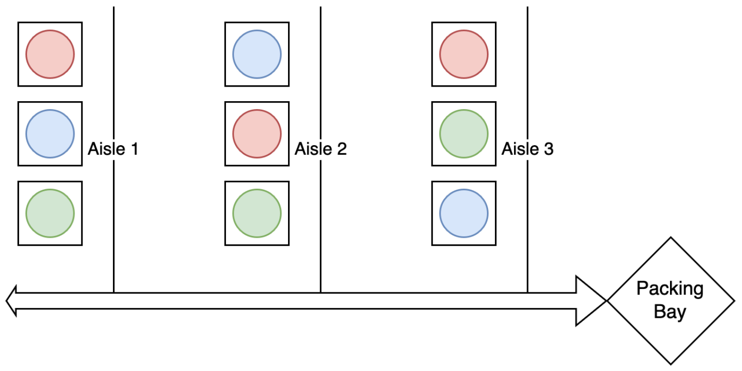


Figure 1. Simulated warehouse environment diagram

The robot software was intended to operate in a very specific environment designed to be similar to that of a real warehouse while remaining simple enough to allow the use of the robot's built-in sensors. The robot's movement implements a line-following algorithm to navigate the aisle geometry depicted in Figure 1. Each aisle, as well as the central lane (bottom of the image) have to be solid black lines with a width of 1-1.25 inches on an otherwise white or very light surface. The end of each aisle as well as the packing bay are indicated by using short black lines perpendicular to the aisles and the lane, respectively. The number of aisles in this setup has to be specified by changing a constant in the code, as detailed below.

Each aisle must contain exactly three packages, which are modeled by the differently-colored plastic balls. The three balls in a given aisle have to have the colors of red, green and blue, and be located approximately 1 foot away from the aisle line. The minimum recommended separation between the adjacent packages is 6 inches. In order for the robot to properly detect the balls, they have to be placed on the same height as the robot's camera and have a solid white background behind each aisle. For the same reason, the location of this arrangement should be moderately lit with the materials used (especially that of the black lines on the surface) being as unreflective as possible. To enhance the color detection quality in a particular setting, the calibration routine, described in the Getting Started section of this document, should be performed. An example setup (with not all balls in place) is depicted below:

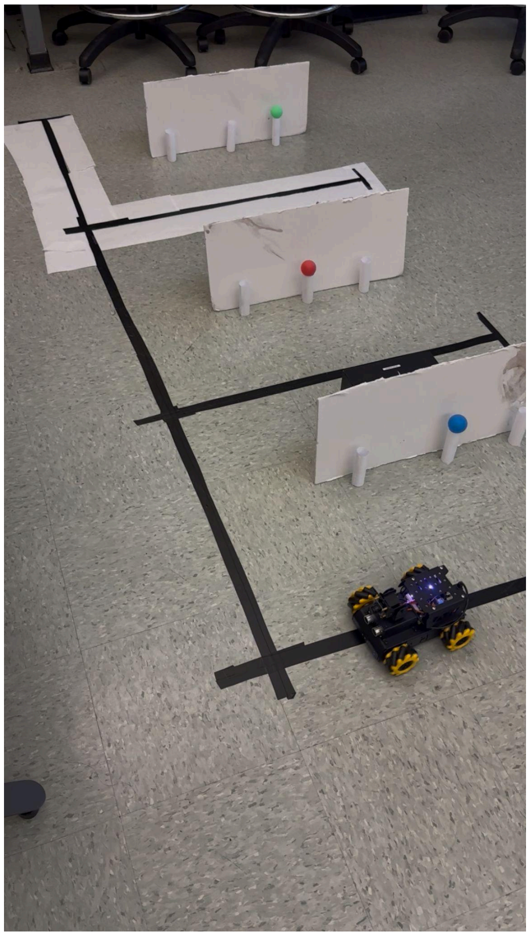


Figure 2. Sample simulated warehouse environment. Having an ending line at the end of each aisle is important, in addition to extending the side aisle lines slightly past the main aisle to form a cross shape for proper aisle detection.

Hiwonder TurboPi Robot

For this project, we used the Hiwonder TurboPi car kit with Raspberry Pi 5 platform. The car took 2-3 for two people to assemble. For best compatibility, we recommend getting the same model. One large drawback of the platform is that it does not include a robotic arm, so the robot does not actually pick up the packages; rather, it beeps and flashes the LEDs on board. The kit also includes red, blue, and green ping pong balls which we used as packages for the robot to detect. The robot functionality used in this project includes the line follower, ultrasonic sensor, camera, LEDs, wheels, and buzzer.

Because the kit is relatively affordable, some of the sensors can be a bit finicky and require calibration. The line follower has a sensitivity potentiometer which we had to adjust, and the TurboPi documentation provided guidance on calibrating the camera to properly detect the color of the balls. The SD card provided in the kit also contains a lot of testing software that you can use to verify functionality of the robot itself.

Getting Started

The SD card provided in the kit came preloaded with many of the Python packages required, but there are a few that you need to install in addition to cloning the git repository. We recommend following the TurboPi documentation on how to connect to the car and view the remote desktop. We also recommend reviewing the documentation and sample programs to get an idea for the capabilities of the car and how it functions before diving into the DeliverPi program.

With the remote desktop open, open the Terminal and enter `cd TurboPi/Functions`. Next, you will want to clone the GitHub repository using `git clone https://github.com/bmongird/DeliverPi.git`. This will download all of the files into a directory called `DeliverPi`. Navigate to that directory by typing `cd DeliverPi`. Next, execute the command `pip install -r requirements.txt` to ensure that all of the required packages are installed on the machine. This may take a while.

Now you are almost ready to run the robot! The implementation of the central server provided with the project reads the prepared order data from a JSON file, making it only suitable for testing. For that purpose, the “generate_orders.py” script is also included. When running it using a Python interpreter and entering the requested restrictions when prompted, it randomly generates a set of orders satisfying those constraints and writes it to the specified JSON file in the format accepted by the test server. To run Python programs from the terminal, ensure you are in the same directory as the file and type `python <file_name>.py`. A few parameters have to be changed to ensure proper communication between the server, hub, and robot. In the “controller.py” and “hub.py”, you will find lines near the top of the file to modify host variables, including SERVER_HOST, CONTROLLER_HOST, and HUB_HOST. Ensure that each of these is set to the correct local IPV4 address of the machine that they’re running on. If the server, controller, and hub are all running on the TurboPi car, all of these can be set to “localhost”.

With the software setup out of the way and your environment set up, you are ready to run DeliverPi! Set the robot at the “packing bay” and run the hub, controller, and server files in that order. You should see that the controller connects to and repeatedly requests orders from the hub until the hub receives an order from the server, at which point the robot will receive the order from the hub and begin operation. Note that if there is an obstruction detected by the ultrasonic sensor, the robot will stop movement and only resume operation if it is cleared. If any small errors arise, read the error messages carefully. If the issue is larger, contact your software engineering team (or yourself if you are a software engineer) to diagnose the problem. Possible issues include missing/outdated packages and connection problems. You can also refer to the troubleshooting section below.

Troubleshooting

- **Robot missing turns or straying off the line**

  The cause of this problem is the inconsistent line sensor readings and it can be mitigated by increasing the brightness contrast between the line and the surrounding surface, using non-reflective materials, ensuring proper environment lighting, and adjusting the line sensor sensitivity potentiometer.

- **Camera not turning when entering the aisle**

  This occurs due to the robot occasionally missing certain commands. The only reliable solution to this problem is to issue several such commands each time. This can be done by modifying the project code and adding repeated calls to the problematic functions.

- **Server/hub/controller not connecting to each other**

  This may be caused by not specifying the correct IP-addresses or hostnames in the code or by the routing setup and/or firewall rules preventing the connection.

Technical Documentation

Introduction

The DeliverPi platform was tested and demonstrated using the Hiwonder TurboPi car kit with Raspberry Pi 5 running the custom Raspberry Pi OS image provided by Hiwonder. DeliverPi was tested and working on Python 3.11 with all of the packages listed in the requirements.txt file. A full commit history and all code can be found at https://github.com/bmongird/DeliverPi. We recommend reading through the user documentation first to get an idea of the project before diving into the technical details.

Networking

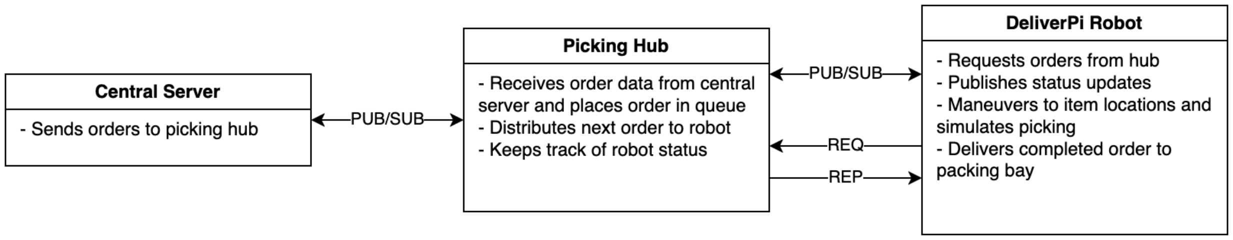

Central Server
- Sends orders to picking hub
PUB/SUB
Picking Hub
- Receives order data from central server and places order in queue
- Distributes next order to robot
- Keeps track of robot status
PUB/SUB
REQ
REP
DeliverPi Robot
- Requests orders from hub
- Publishes status updates
- Maneuvers to item locations and simulates picking
- Delivers completed order to packing bay

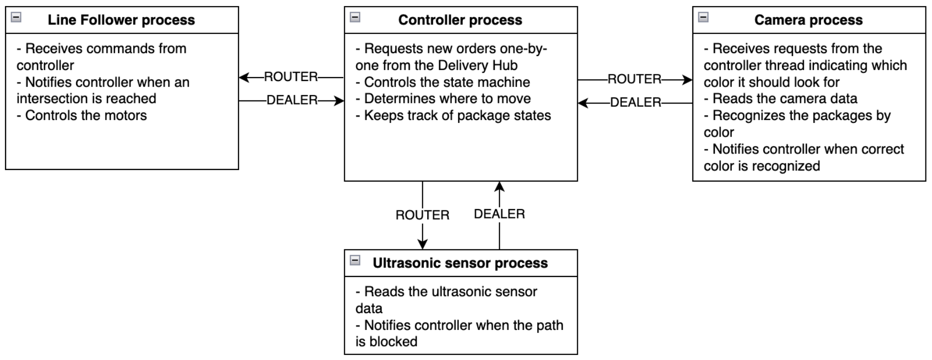
Figure 3. The networking architecture of the project.

The network is used for communication between the central server, picking hub and robot controller. As detailed in the User Documentation, the provided central server implementation is only suitable for testing purposes. To use the project in a production environment, a custom implementation of the server, which would retrieve the orders from another data source, has to be created. It would have to comply with the network message format expected by the picking hub. That is, the order data transmitted as JSON-serialized object compatible with the "OrderData" Python type specified in the "common.py" file. The transmission itself must adhere to the protocol implemented by ZeroMQ's PUB-SUB socket architecture with the server acting as the publisher. In case of deploying several picking hubs operated by a single central server, the message format can be augmented with the target hub identifier for each order, allowing the addition of the appropriate message filtering in the picking hub code.

The picking hub implements a simple earliest-deadline-first approach to scheduling orders. To do that, it employs the min-heap data structure internally and updates it accordingly whenever it receives a new order from the server or sends the next order to one of the robots. The networking with the robot is implemented using the REQ-REP pattern to enable the robot to request a new order only whenever it is ready to start collecting it. The hub also subscribes to the robot's PUB-SUB socket for status updates. Currently those are simply logged, but an appropriate business logic can be implemented instead.

Whenever a robot finishes the previous order, it repeatedly tries to request a new order from the picking hubs with 1 second delays between consecutive requests. After the order is received, the robot's controller sorts them by their aisle number and schedules them for pick up in the resulting order. It sends the appropriate status updates back to the hub when it finishes the order, as well as when it cannot pick up one of the packages either due to the path to it being blocked or the ball with the appropriate color not being detected in the specified aisle.

Controller



Line Follower process
- Receives commands from controller
- Notifies controller when an intersection is reached
- Controls the motors
Controller process
- Requests new orders one-by-one from the Delivery Hub
- Controls the state machine
- Determines where to move
- Keeps track of package states
Camera process
- Receives requests from the controller thread indicating which color it should look for
- Reads the camera data
- Recognizes the packages by color
- Notifies controller when correct color is recognized
Ultrasonic sensor process
- Reads the ultrasonic sensor data
- Notifies controller when the path is blocked
ROUTER
DEALER
ROUTER
DEALER
ROUTER
DEALER

Figure 4. Robot control architecture.

The controller is the brains of the robot and controls all of the robot functionality and movement. Status updates and other debug info are logged using the logging package. When initiated, the controller launches 3 subprocesses - linefollower.py, color_detect.py, and ultrasonic.py. We’ll start by looking at the ultrasonic.py process as it is the easiest to grasp. In it, there is a msg() function that is launched as a thread and runs in an infinite loop, receiving requests from the controller on the ZMQ DEALER socket and modifying the state accordingly (starting/stopping). The main thread simply reads ultrasonic sensor data and notifies the controller via the DEALER socket whether the path ahead is blocked or not. The linefollower.py file follows an almost identical structure, with an extra turn() function for turning the car. The main thread is a match/case statement that matches the sensor data. False means no line detected, true means line detected. The car velocity is set accordingly (continue straight, turn slightly left/right, etc.). The color_detect.py file has a similar msg() function but it is much more complex than the previous two files. Much of its functionality is taken straight from the ColorDetect.py file located in the TurboPi/Functions folder provided by Hiwonder. The basic functionality is this: when the camera detection starts, camera data is pulled and run through some OpenCV color detection code. We found that this worked remarkably well in our application, and you can refer to Hiwonder's documentation for more information on exactly how it works. Some important things to note regarding the camera process follow.

- Line 260 checks for the maximum area of the color (ball), so you may have to adjust these values depending on the size of the colored object you're detecting.
- Directly below that, we check for the object to be in the middle of the frame. Adjust these values accordingly.
- The process opens an OpenCV window when running for easy debugging (and because it's cool to see) but this should be disabled if deploying in a long-running application to save execution time.
- The move() function is where the process will notify the controller via DEALER socket if it detects the color (line 171).

The controller communicates with all of the processes using ZMQ's ROUTER/DEALER socket. It operates its own state machine to control state. The most important functions to understand are process_event(), listen_for_messages() and execution_thread(). The listen_for_messages() thread will receive incoming messages from the router socket and process them accordingly. The execution_thread(), as the name implies, is the main execution thread of the program. Here, you will see some functionality implemented based on the state that the controller is currently in. You can add functionality that might have to run in a loop here. The process_event() function is the real "meat and potatoes" of the program and is where the state transitions and control logic is implemented. You'll see a match/case structure matching the event to be processed. Most event names are descriptive and you can get a sense of what they do by reading the commands that they send and updates they make. These event names are also used for transitions in the state machine. Note that this function should run pretty quickly to avoid slowdown. Much information on how this function (and the rest of the codebase) operates can be gleaned from the doc and inline comments left behind.