Embedded Systems Software Integration

Overview:

In order to develop the computer vision targeting system, it was necessary to streamline the development process so that the vision code can focus on implementation.

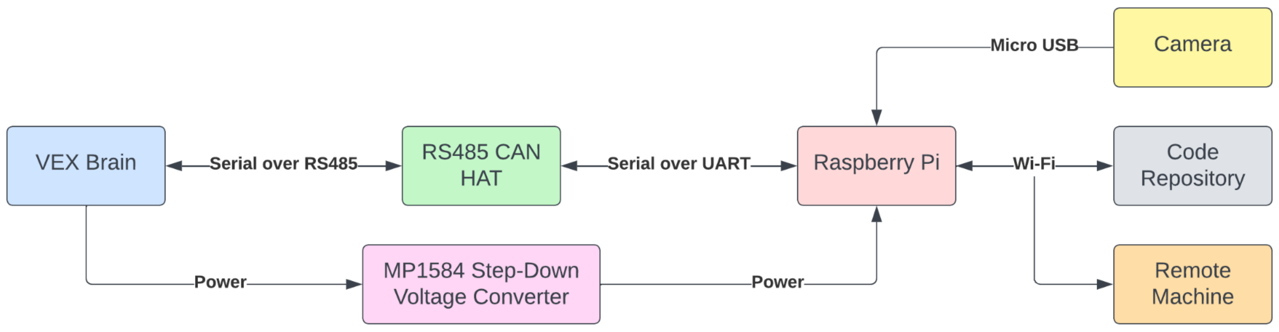
Step: 1	Date: 12/21/2022	Engineer: Jeremy Eastham
Witness: Mariana Sanchez	Date Reviewed: 02/21/2023	

Objective:

- Design the software counterpart to the Raspberry Pi assembly
- Discuss and document design challenges

Remarks:

The hardware design of the Raspberry Pi assembly necessitates disabling terminal access over USB. The reason for this is that the virtual serial port that is used by the UART pins on the Raspberry Pi for communication with the VEX Brain is the same port that is used for SSH over USB. If terminal access were left enabled, the communication link between the Pi and the Brain would be polluted with bash commands and command output. Due to this limitation, the first software problem that needed to be solved was the difficulty of deploying new code to the Raspberry Pi. It is also not optimal to only have one copy of the working vision code. The team uses a git repository to manage changes, so the team designed an automated system that automatically keeps the Raspberry Pi up to date with the latest code over the internet. The Pi polls the repository for updates every few seconds and pulls any new changes, and then restarts the vision program. This allows for quick iteration as long as the Pi has Wi-Fi access. This program can be daemonized so that the Raspberry automatically runs this program when it boots up and restarts it if it crashes. This will allow the Raspberry Pi to respond to commands from the robot as soon as it boots up without human intervention.



VEX Brain to Raspberry Pi Communication Diagram

Made with LucidChart

Next, the team needed to design a communication protocol between the VEX Brain and the Raspberry Pi. In preliminary testing, the communication link did not suffer from any data loss. However, the communication needs to be resilient in order to recover from malformed messages. The vision program crashing during a match could be catastrophic. This can be broken down into four problems: packet framing, packet structure, packet verification, and packet recovery. The solutions to these problems needed to be designed with performance in mind. Further research will need to be done to implement algorithms that meet all of these requirements.

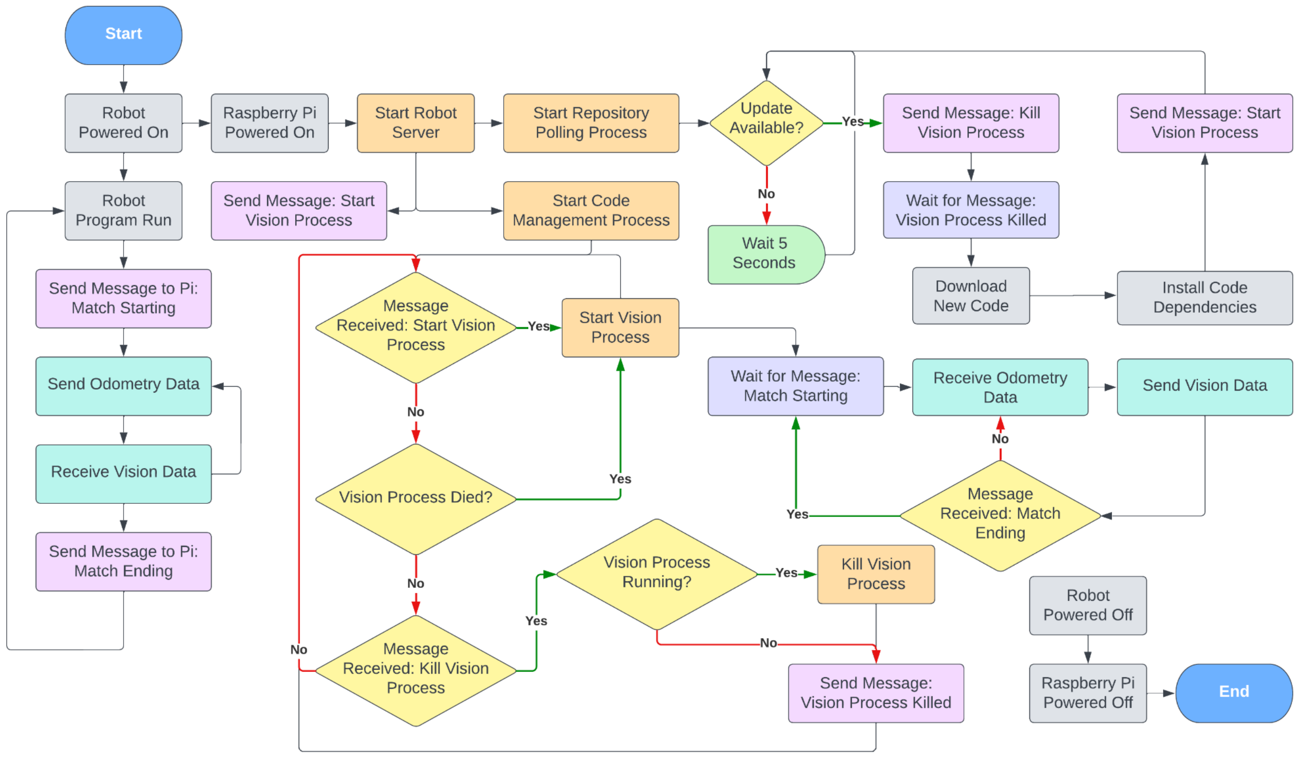
Accomplishments:

- Designed the software architecture for the Raspberry Pi to VEX Brain link

Step: 2	Date: 01/04/2023	Engineer: Jeremy Eastham
Witness: Mariana Sanchez	Date Reviewed: 02/21/2023	

Objective:

- Build an automatic deployment system for the Raspberry Pi


Start
Robot Powered On
Raspberry Pi Powered On
Start Robot Server
Start Repository Polling Process
Update Available?
Yes
No
Wait 5 Seconds
Send Message: Kill Vision Process
Wait for Message: Vision Process Killed
Download New Code
Install Code Dependencies
Send Message: Start Vision Process
Robot Program Run
Send Message: Start Vision Process
Start Code Management Process
Send Message to Pi: Match Starting
Send Odometry Data
Receive Vision Data
Send Message to Pi: Match Ending
Message Received: Start Vision Process
Yes
Start Vision Process
No
Vision Process Died?
Yes
No
Message Received: Kill Vision Process
Yes
No
Vision Process Running?
Yes
Kill Vision Process
No
Send Message: Vision Process Killed
Wait for Message: Match Starting
Receive Odometry Data
Send Vision Data
No
Message Received: Match Ending
Yes
Robot Powered Off
Raspberry Pi Powered Off
End

Control Flow Diagram of the Robot Vision Server
Made with LucidChart

Remarks:

The automatic deployment system was relatively simple in theory, but complex in practice. The main aspect of this program that added complexity was the fact that the Raspberry Pi needs to concurrently poll the repository for updates and also manage the vision program. It should also relaunch the vision program if it crashes. The team ultimately decided to build this project with multiple concurrent processes. An interprocess communication protocol was designed that manages four separate processes: the server (process monitor), the repository poller (code updater), and the code manager (process starter/killer). This architecture was complex to build, but it should be resilient against unpredictable circumstances.

One major issue with this server is that code dependency installation could not be fully automated. In addition to installing Python packages, certain package-specific commands need to be run to ensure that certain binaries are installed correctly. However, once a package is installed, code updates are completely automatic.

The team also had issues deploying the deployment server. The Raspberry Pi only has 1 GB of RAM. Once the server started launching processes, this filled up quickly. By default, the Pi is set up with a statically allocated 100 MB pagefile. After diagnosing the problem, the pagefile was set to automatically resize as needed so that memory will not be an issue in the future.

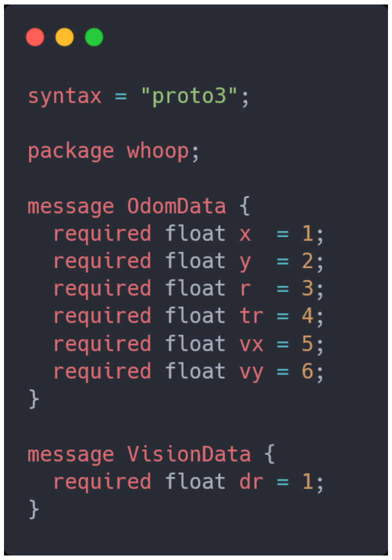
Accomplishments:

- Successfully set up automatic deployment on the Raspberry Pi

Step: 3	Date: 01/14/2023	Engineer: Jeremy Eastham
Witness: Mariana Sanchez	Date Reviewed: 02/21/2023	

Objective:

- Design and implement a serial communication protocol between the VEX Brain and the Raspberry Pi



```
syntax = "proto3";

package whoop;

message OdomData {
  required float x  = 1;
  required float y  = 2;
  required float r  = 3;
  required float tr = 4;
  required float vx = 5;
  required float vy = 6;
}

message VisionData {
  required float dr = 1;
}
```

Remarks:

As mentioned previously, the problem of reliable communication between the robot and the Raspberry Pi can be broken up into four tasks: packet framing, packet structure, packet verification, and packet recovery.

For the problem of packet framing, the team decided to use the simple and widely regarded Consistent-Overhead Byte Stuffing algorithm. This algorithm encodes bytes of data in such a way that the packet does not contain any null bytes. This means that we can encode a packet using COBS and then send a null byte to indicate that the packet is complete. The process on the other end of the connection will continue receiving bytes until it sees a null byte and then attempt to parse the packet.

For the problem of packet structure, the team decided to use the industry-standard ProtoBuffers library created by Google. ProtoBuffers can encode any data structure according to a specification written in an easy to use domain-specific language. The library is language-agnostic and protocol-agnostic so we can use it to generate parsing code for both sides of the serial link.

For the problem of packet verification, the team decided to use a popular library called xxHash. This is a hashing algorithm that is faster than cryptographically secure algorithms such as MD5 or SHA-1. Using this algorithm, we can attach a hash to each packet and use that hash to verify the integrity of the packet's data on the other end of the serial link.

For the problem of packet recovery, the team decided to declare two different types of packets: message packets and data packets. Message packets are important. They contain data such as "the match is starting" or "the vision program is still running". These packets require a confirmation packet to be sent back to the sender to ensure that it is received. Data packets contain data such as odometry position or goal angle. It is not a problem if a few of these are missed, so it is more efficient to just send the next one instead of requiring confirmation each time a packet is sent.

Accomplishments:

- Designed and implemented a robust serial communication protocol