DOCUMENTATION

VERSATILE ARDUINO NETWORK (VAN)

CONTENTS

DESIGN

NETWORK

An Arduino microcontroller (MCU) is often able to provide cheap, easy, realtime and satisfactory control of a single actuator, or monitoring of a single sensor. However, many systems, such as a robotic arm for example, may require tight control of many actuators using feedback from many sensors. Problems can soon arise- Are there enough interrupt pins to keep track of all encoder positions? How does the computational load from controlling one actuator affect the timing of the others? How will an abundance of wiring be routed to reach the microcontroller?

The VAN's purpose is to facilitate the implementation of such systems by using multiple MCUs as nodes, with reliable, low latency communication between them. Each node can send information it determines is relevant to any other, and each node can act on information another node has decided is relevant to it. However, for the most part, individual nodes can get on with their own limited number of realtime tasks without being affected by the complexity of the whole system.

Source files exist for running a VAN on basic Arduinos (Uno, Nano), the more capable Mega, and the yet more capable ESP32. Serial ports are used as the primary means of data transfer on all MCUs, as each port is provided with hardware send/receive buffers, minimizing disruption imposed by handling communications. Esp-now is also used to provide a "wireless serial port" between multiple esp32s. Lacking hardware buffers for this purpose, esp32s are briefly interrupted when receiving communication.

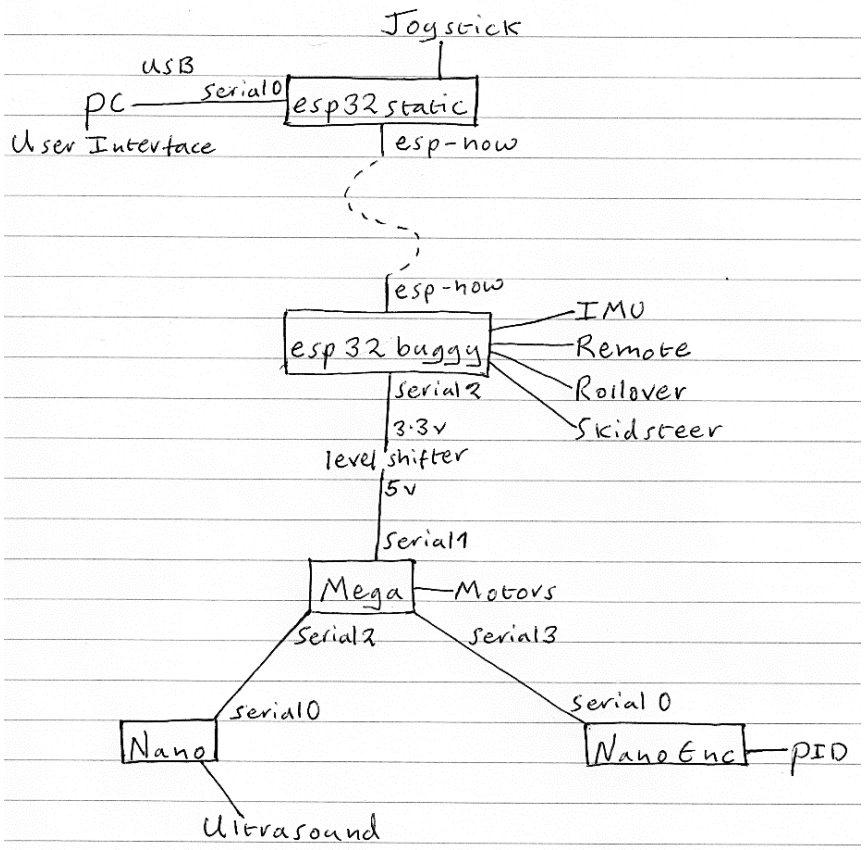
The VAN is implemented using a connectionless protocol, information is sent as discrete messages, each containing sufficient information to reach their destination. Each node requires an individual network map defining the locations of all other nodes and devices in relation to itself. By default, no acknowledgement protocol is used, and devices are unaware if a sent message has been correctly received. Optionally an acknowledgement protocol can be used. This provides sending devices the option to let an important message be automatically repeated until its correct reception by the destination node has been verified.

BUGGY

To demonstrate and evaluate the VAN, it was used control a small buggy via multiple user controls. This is intended to represent a potential real-life use case. Mobile industrial equipment, such as a forklift or JCB, could benefit from a distributed design to simplify adding modules such as hydraulic actuators or proximity sensors. The inclusion of alternative methods of control could allow workers to remain positioned with a good view of their task, rather than of the equipment's cab. Communication with a PC was also implemented, which may allow remote monitoring, or shutdown in the event of an emergency.

In this system the VAN was running across five nodes: two esp32's, two Arduino Nanos, and one Arduino Mega. Motors and encoders were used to demonstrate that the network is capable of real-time actuator control with feedback, even across nodes. A joystick, remote, and PC key capture were used as controlling sensors, connected to multiple nodes, demonstrating that the network is capable of routing control from different origins to its intended actuator. Inclusion of the IMU and ultrasound sensor allow control to be overridden by safety-critical information, bypassing the node providing control. Additionally, these sensors also send continuous readings to the PC. Two purely software devices, skid steer and rollover, were used to convert a control data format, and detect rollovers respectively. This was done to demonstrate that the VAN allows separation of sensing/processing requirements, improving modularity and potential allowing more capable nodes to be used as application servers for less capable ones.

A network map is shown below, with the boxes around nodes, but not their child devices. The VAN is not running on the PC, and so is shown here as a device. However, it can communicate with the network using a python user interface.



User interaction is provided by passed control messages:

- Joystick->skidsteer->PID->motors
- Remote->PID->motors
- PC->motors

The joystick and remote stop sending control when they are not in use, rather than continuously instructing the PID to remain motionless. This allows either to be used without first having to lock out the other. The final message from the remote when relinquishing control is sent requiring acknowledgement, as it is critical that the buggy stops moving when it is no longer under control. However, it is possible for multiple users to send conflicting messages using these controls simultaneously. If key capture is used from the python user interface, the PID is automatically disabled, and then reenabled upon exit. This has the effect of locking out the other modes of control.

Safety cutoffs are provided by changing parameters of the motor device:

- Ultrasound->motors
- IMU->rollover->motors

IMPLEMENTATION

MESSAGES

So that all nodes are 'on the same page', communication is limited to discrete message structures of seven bytes, with a standard format: start, source, destination, command, data, data, validity. This structure is defined in message.h.

The start byte indicates if this message is standard, requires acknowledgement or is itself an acknowledgement response. However, its main purpose is to indicate where in a stream of bytes the beginning of a message is, allowing the following bytes to be correctly interpreted.

The source byte is the address of the device which originally produced the message, specifying that the reported information pertains to a specific sensor, for example. Each device must have a unique byte long address, giving an upper limit of addressable devices of 127. A node may have multiple child devices, which can be sensors, actuators, combinations of the two, or purely software functions. Nodes exist in the same address space.

The destination byte specifies which node or device is the intended recipient of a message, allowing messages to be routed through intermediary nodes, and allowing the recipient device to act only on information addressed to it. A message sent between two devices on the same node will not leave that node, and will be passed directly to the recipient device.

The command byte holds the messages purpose- Is it a sensor report, or an error? Is it intended to modify another devices parameters? If so which one?

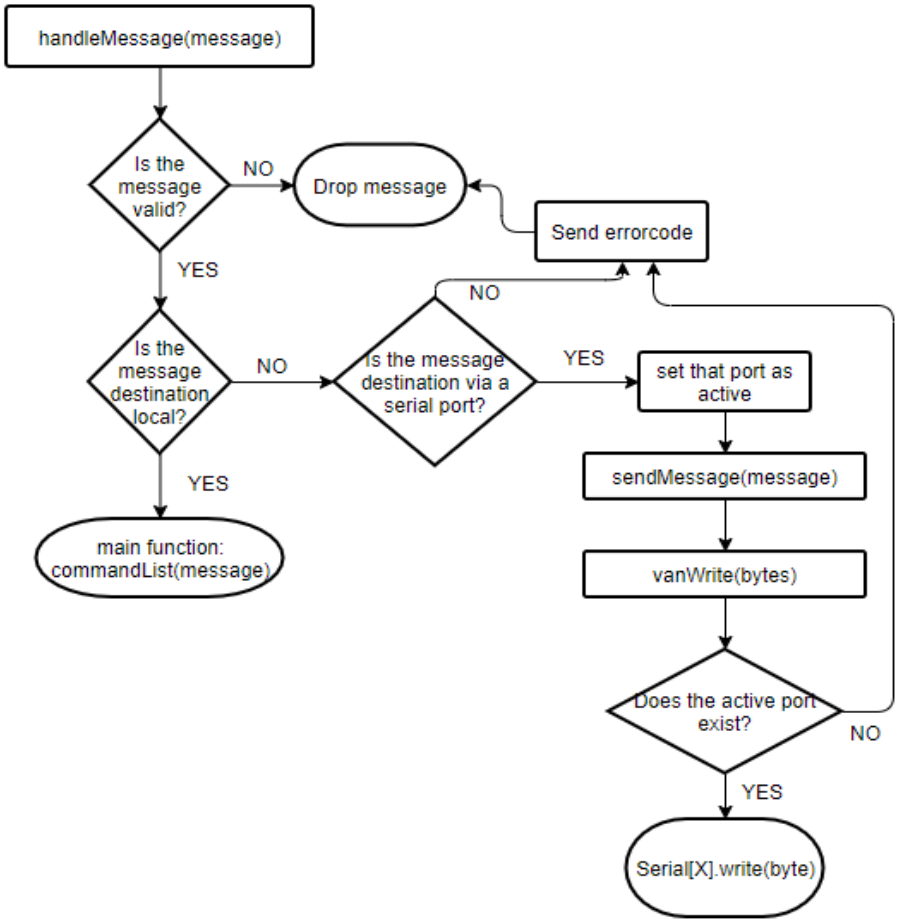
The two data bytes usually hold numeric information, such as a sensor reading or a commanded motor speed. The source and destination devices must have a shared expectation of the datatype: it could be a signed 16bit int, two unsigned 8bit ints, a Boolean value or a network address.

The final byte is the messages Boolean validity, if false, this indicates the message is corrupt, not yet complete, or has been deleted.

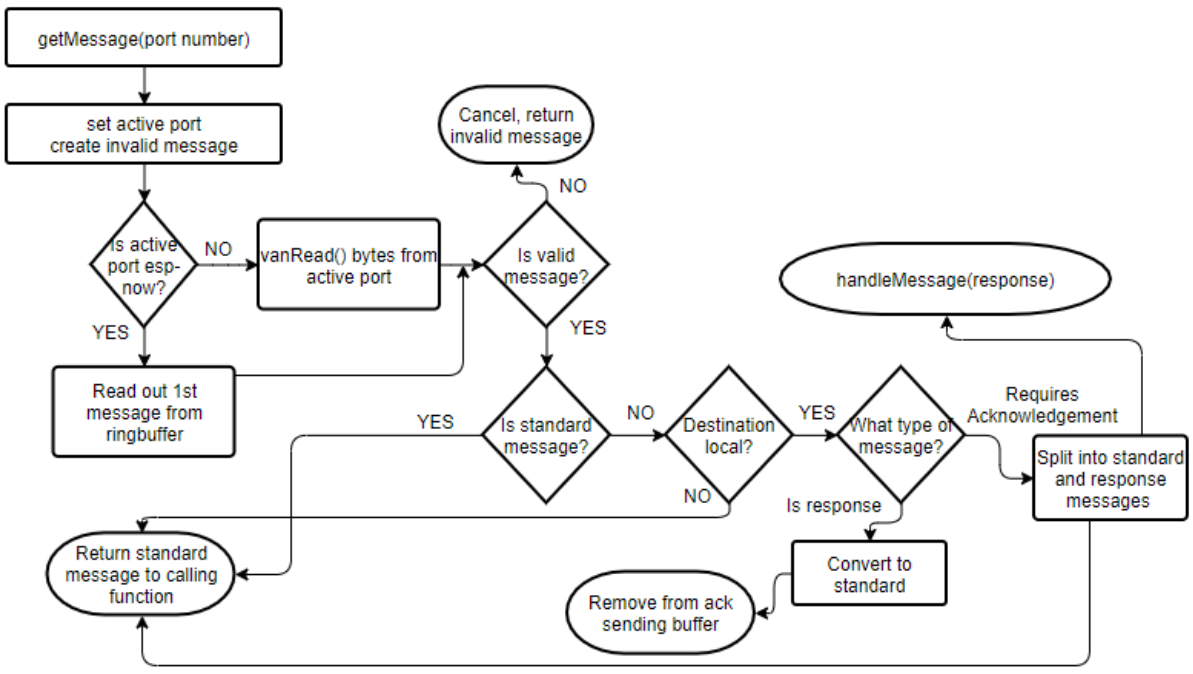
However, the validity byte is never sent outside the node. When sending, VAN replaces it with a checksum, which is the XOR of all the other message bytes. This is computed once when the message is sent, and again when it is received. If the result is equal to the received checksum, the message is marked as valid. Otherwise, the VAN assumes the message has been corrupted during transmission and drops it.

MIDDLEWARE

The middleware VAN is provided by including vanEsp, vanMega, or vanUno, depending on the board. Sending a message is handled by calling handleMessage(message), which (with a correct network map in netDef.h) will route it towards its destination. Its operation is shown in the following flowchart.



Receive a message by calling getMessage(port number to check). If the message requires an acknowledgement a response will be sent automatically, and a standard message is returned. If it is a response to an acknowledge message being sent repeatededly by this node, this message will be removed from the acknowledge sending buffer, AKB0.



From the top level Arduino sketch, repeatedly calling a chain of these two functions on every VAN node, handleMessage(getMessage(x)) provides the core message routing functionality of the VAN. In the same loop a reportList() function allows devices, and AKB0, to send regular messages (the period between sends is handled by the devices themselves). A commandList(message) function must also be present to allow the VAN to pass messages to their destination devices, but does not need to be called by oher methods.

NODES AND DEVICES

Nodes are the networked microcontrollers using one of the VAN middleware headers: vanEsp, vanMega, or vanUno. These allow the node to send, receive, route, and respond to messages.

Devices are software objects running on the nodes, often using GPIO pins to interact with sensors or actuators. All devices included are classes to allow multiple instances of the same sensor or actuator on a single node, and to encapsulate them from each other. However, they can also be created in the main sketch or as a header.

Generally, their behavior is defined by two main functions:

- .command(message) is called when a message is destined for that device, and contains logic to alter device parameters or GPIO values based on the message.
- .autoReport() is called repeatedly, if a report is 'due', .instantReport() is called which generates, and sends, messages based off GPIO or parameter values.

However, some devices only need to send messages when directly commanded to, and others are only required to send regular updates without ever being modified. In such cases one of these functions can be ommited. Blocking functions should not be used, as this will also block the entire node.

ACKNOWLEDGMENT PROTOCOL

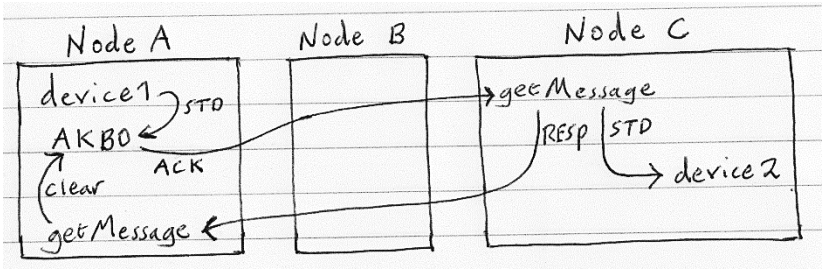
The acknowledgement protocol addresses a shortcoming of sending connectionless messages: a sending device is unaware of weather a sent message has reached its destination. If the messages purpose is critical, such as disabling motors, the safest course of action for the sending device to take is to keep repeating the message. Consequentially, bandwidth is wasted.

This shortcoming is addressed by allowing nodes to instead add this message to an acknowledgment buffer, from ackbuff.h, an instance of which, AKB0, is used by VAN. Messages in this buffer will be repeatedly sent at a set period, with a special start byte: acknowledge, ACK.

Upon an ACK message being read in at its destination node by VAN function getMessage, two messages are automatically produced. In one, the start byte is switched to back to standard, STD and it is routed to its destination device. The other is given the start byte RESP, flagging it as a response. Its source and destination are swapped, before routing it back to the device which sent the ACK message.

Upon a RESP message being read in at its destination node by VAN function getMessage, the source and destination are swapped back, and it is flagged as STD, recovering the original message stored in the ackbuffer triggering this exchange. If this message can be found in the ackbuffer, it is then removed, completing the transaction.

This permits indefinate chaining through intermediary nodes, where both the ACK and RESP messages are simply passed on.



Once the message has been added to AKB0 by a sending device, the exchange is then completed by VAN, no further actions are required by either device. The acknowledgment protocol does not gaurentee that the destination device has got the message, or acted upon it. It only garentees that the recipient device's parent node has recieved it correctly i.e It is intended to correct communication errors, not software errors. No error messages are raised by indefinatelty waiting ACK messages. If the sending device needs to cancel the transaction, or be certain that an obsolete transaction will not take place calling AKB0.cancel(message) will remove messages which are identical in all but data values.

ACKBUFFER

An ackbuff object, defined in ackbuff.h, is a software buffer which stores messages but does not retain their order, for the purpose described in the previous section. Calls to .add(message) add that message to the buffer, duplicate messages will not be added again. Calls to .clear(message) removes only that exact message from the buffer. Calls to .cancel(message) removes that message disregarding data values. Calling clearAll resets the buffer.

ESP-NOW WIRELESS MESSAGING

ESP-NOW is a connectionless Wi-Fi communication protocol which can be used between multiple ESP32s, the VAN utilizes this to form a wirless link in its network.

The MAC address of the receiving node must be added to the sending nodes netDef.h, and espNowSetup() must be called before use. The network map must be difined with a serial port that the ESP32 does not have, SPORT[3]. When the VAN function handleMessage(message) determines a message should be routed over ‘serial port 3’ it is intead transmitted to the receiving nodes MAC address.

Upon reception via ESP-NOW, an interupt service routine, espNowISR, is triggered. It is not reccomended to act upon the message within the ISR, received messages would have absolute priority over those from other ports, potentially locking them out altogether. After its validity is checked, the message is instead stored in a ringBuffer, RB0.Store, before the ISR exits.

When ‘serial port 3’ is later checked via getMessage(3), the VAN instead reads the oldest message from the ringBuffer, RB0.Read which is then returned in the same way it retrieves messages from the hardware serial buffers.

RINGBUFFER

A ringbuff object, defined in ringBuff.h, is a FIFO software buffer used to store messages in order, for the purose described in the previous section. Calls to .store(message) add that message to the buffer, calls to .Read return the oldest message stored in the buffer. If no messages are avialable in the ringBuffer, an invalid message is returned which should not be acted upon. The capacity is set to 10 messages, overloading the buffer triggers an ELog. Variables available, overload, and msgCnt indicate whether at least one message is available, wether it’s overloaded and the number of messages available. These are not used by the VAN.

IMU DEVICE

Allows the use of a 6050 MPU (combined accelerometer and gyroscope) to provide measurement of:

- Buggy longitutional acceleration, AY, indicating climb/descent angle. Value in g's X1000.
- Rotation velocity as viewed from above, GZ, indicating turning velocity. Value in deg/s
- Absolute rotation around buggy longitutional axis, ANGY, indicating sideways roll. Value in degrees

It must call a main sketch function IMU0Update() to get readings from a MPU6050 wire object (from library MPU_6050 light)
Commands PARAM0,1,2 set destination of sensor channels independantly.

PID DEVICE

Continously sends velocity commands to motors, based off it's last commanded target velocities and encoder counts. Member functions .isrLeft and .isrRight must be called by interupt service routines setup in main sketch. Messages with command SET use DAT0, DAT1 to set left, right target velocities from a single message. Input target velocites range from 0 to 128 to 255, respectivly full reverse, stop, full forwards. PARAM0 sets encoder gain, a multiplier is needed since encoder count resolution is very low. PARAM1,2,3 respectively modify P,I and D gains. Output motor PWM messages use the same format as PID input, described earlier.

JOYSTICK DEVICE

Continously sends readings from two axes of an analogue joystick, by default to device skidsteer, which uses them to produce PID commands. Period and destination are the only run-time modifyable parameters. To prevent sending messages based on sensor noise, a deadband is used, only sequential readings outside the deadband are used.

SKIDSTEER DEVICE

Skidsteers purpose is to receive messages from a joystick, and convert velocity, heading values to target velocities for left, right motors. These are then sent to the PID device. As such there is no autoreporting, this device only acts when it recieves a message to convert. Only the output destination in modifyable.

REMOTE DEVICE

The remote device decodes received infra-red codes, and produces PID control messages for the up, down, left, right buttons. Message output format is as described in PID section. It must regularly call a main function, IMU0Update() to operate. When no longer in use, once this has been acknowledged by its destination node, no further messages are sent.

ULTRASOUND DEVICE

Takes distance measurments using an SR04 ultrasonic sensor continously, or optionally as a one off reading. Readings output as cmd:REPORT, dat0&dat1: int16 (us). Function .assessDanger detects if an object has moved infront of the buggy or been removed. Upon either of these state changes, an acknowledge message is sent to device motors, disabling or enabling forward motion. Message format: cmd = PARAM0, data = 255, 255, or 0, 0 for enable disable respectively. Only the distance reports are sent if there is no state change. The threshold for blocking obsticle distance is modifiable with cmd = PARAM0. Period is modifiable, this affects both functions. Destination modifiable only for reports.

MOTORS DEVICE

Sets PWM, direction pins for two motors, based on messages received in same format as described in PID section. Messages with CMD PARAM0, PARAM1, respectively enable/disable forwards motion, or all motion. Upon change of either enable state, this is reported to the PC.

ROLLOVER DEVICE

Device receives messages from IMU, with CMD = PARAM4, with data as int16 angle around buggy Y axis. Function .assessDanger determines if this angle is within a safe range of +/- 45 degrees, and if the state has changed between safe/unsafe an acknowledge message containing the new state is sent to the motors.

PYTHON USER INTERFACE

Requires python 3 or later, on Ubuntu 18.04. Started by running run.py. A serial device must be connected on /ttyUSB0. A monitor window opens once the buggy is powered on. The last message received is displayed as plaintext, if addresses/commands are known. Otherwise message bytes are shown numerically. Ultrasound, X axis inclination, Z rotation velocity and motor state messages are not displayed as they are instead shown graphically.

The terminal input allows the construction and sending of messages. Message destinations and commands, listed in definitions.py must be entered in plaintext (not case sensitive). These are listed in the console for reference. The message source is not entered, source 'PC' is used automatically. Message data is then added numerically, as two words for individual bytes, or as one for int16. Messages are not sent if formed incorrectly, a promt will detail why the message is incorrect. Control of buggy via keycapture is activated by pressing C, W,S,D,A keys are used, keycapture is disabled by pressing C again.

Threading is used to allow these purposes to run indepentantly. After connecting, run.py spawns a display thread for monitoring, another for serial communication, and another for terminal input. These correspond to display.py, serialComs.py and userInput.py files. An additional temporary thread is used for keycapture. Information is passed between threads using a single class 'Threadshare'. All threads are closed by entering 'q' in the terminal.

EVALUATION

NETWORK

The 'Message' structure used was designed to communicate 16 bit signed integer values, on the basis that this would be an appropriate length for most of the control/sensing used. In practice, sometimes less precision was required and two 8 bit 'channels' could be communicated in a single message, as used for the motors, PID, Joystick, or even just Boolean values. These special cases required packing/unpacking code on each device using them. A better design would have been to include some data content datatype information as part of the Message, and generic packing/unpacking functions included in the structure for every data format. In other cases, it would have been preferable to send and receive floats, without losing precision. Variable message lengths could have allowed this, without wasting bandwidth for shorter messages, and is a better technical solution. However, this was not attempted because it would add a more complexity to the message routing, passing, verification and buffering functions.

The XOR checksum is lightweight, but not foolproof. It will not reject messages with an even number of bit errors at the same byte position, or with a wrong ordering of bytes. Additionally, there is no means of message recovery/resending if an error is detected.

The acknowledgement protocol does not inform the sending device of any response, it has to rely on a 'best attempt' being made by the VAN. However, it is possible for the sending device to check AKB0 to see if its message is still awaiting a response, and it could take action based on that. Alternatively, an acknowledgement exchange could have been implemented at the device level, instead of in VAN, which would not require an ackBuffer, and could guarantee the sending device that the destination device has acted on the message. However, this would have required additional states to be added to every device that uses this protocol.

The PID and motors devices could act dangerously in the event of a partial network failure, as no timeout was implemented for target speed commands. An encoder failure would not be detected by the PID device, which could then overshoot its target speed.

Nothing prevents the user from setting a devices destination to itself, which could then become trapped in an unrecoverable loop.

Adding, removing or relocating devices necessitates updating the netDef files for all nodes, which is tedious and error prone. Potentially a script could update all of them based on a single map of the entire network.

The user interface should not have been made in Python, as a 'standalone' application. A better solution would have been to adapt the C++ VAN code to also run on a PC, allowing the PC to be a node, with the various indicators and inputs as its child devices. This would have been consistent with the rest of the network, and likely would have had better performance, especially if interrupt driven behavior was used to replace polling.

The ESP-NOW link is vulnerable to malicious attack, if an attacker was aware of how messages are structured. They wouldn't even need the MAC address of the targeted ESP32, as malicious messages could be sent as a broadcast transmission. However, ESP-NOW does include a built-in encryption method which could be utilized.

There is no way for nodes to check that messages genuinely originated from the source they are claim to, allowing any device to send messages pretending it is any other device.

No varying levels of user privilege were implemented, although user control via the remote or joystick are limited to control of a single actuator. Additionally, key control via the python UI overrides them by bypassing the PID controller. Limited network access via the UI could be implemented by omitting the definitions for sensitive parameters, such as PID gains, from the definitions.py file.

BUGGY

The PID system was disadvantaged by residing on a dedicated Nano, having only enough interrupt pins to be triggered by just a single encoder channel per motor, it could only receive half the maximum resolution the encoders could otherwise provide.

The ultrasound sensor was attached aiming slightly downwards, sometimes detecting features which were not actually blocking the buggy, such as carpet edges. In addition, it was found to be inaccurate when objects were presented at an angle over 35 degrees. The use of multiple ultrasound sensors at a range of angles, or servo adjustment of one sensor are possible solutions.

The megas 5v regulator provided power to all systems on board other than the motors. For any more microcontrollers or actuators to be used, a more capable power supply would be required.

The use of four microcontrollers on this buggy, if they were not serving as a demonstration of this network, would have unnecessarily complicated its design and debugging. Generally, use of a distributed system requires more justification, such as leveraging the strengths of heterogeneous hardware, a requirement for redundancy, or is it unavoidable due to large distances between sensors or actuators.

TESTING

Testing involved gathering experimental results for message passing latency, resilience to poor line quality and wireless range. These may indicate the VANs suitability for time-critical systems, with components spread over large physical distances. Experimental results were also gathered from the motors, PID and ultrasound to determine their limitations, reliability or potential software bugs. Code tests were developed for the more complex aspects of the VAN code: the message passing functions, acknowledgement protocol, and software buffers. These required such tests for their development, and verification of predictable operation. All results are presented in testing.pdf, but the code used to produce them is available on GitHub in the folder testingNodes.

REFERENCES

VANesp:
Used to create functions: espNowISR, espNowSetup, and to transmit messages via ESP-NOW in handleMessages
https://techtutorialsx.com/2019/10/20/esp32-getting-started-with-esp-now/

espBuggy.ino
Used for wiring MPU6050 with ESP32
https://www.instructables.com/Connecting-MPU6050-With-ESP32/
Library used to read MPU6050 from ESP32
https://github.com/rfetick/MPU6050_light
It's documentation
https://github.com/rfetick/MPU6050_light

Library used to read IR remote
https://github.com/Arduino-IRremote/Arduino-IRremote
Example code used in function REM0Update()
https://github.com/Arduino-IRremote/Arduino-IRremote/blob/master/examples/IRreceiveDemo/IRreceiveDemo.ino

Van_dev_ultrasound
Code used in function instantReport(), to trigger and time ultrasound pulse from SR04
https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/

Van_dev_pid
Example code from initio buggy manufacturer used for counting encoder pulses with ISRs
http://www.4tronix.co.uk/arduino/sketches/initio_03.ino

Python user interface (van_dev_pc)
Third party modules are used:
serial, time, threading, tkinter, matplotlib, numpy

run.py
Code used from forum to implement threading
https://www.reddit.com/r/learnpython/comments/8kfys3/running_code_while_waiting_for_user_input/

definitions.py
Code used from forum for swapping key, value pairs in dictionaries
https://stackoverflow.com/questions/1031851/how-do-i-exchange-keys-with-values-in-a-dictionary

display.py
Code used from online cheatsheet to create tkinter window, labels, progress bars
https://gist.github.com/athiyadeviyani/b18afdc8136f003956b1a71d94a6c696

serialComs.py
Code used from pySerial documentation to open, read, write serial to port
https://pyserial.readthedocs.io/en/latest/index.html
Originally adapted from
https://github.com/recantha/EduKit3-RC-Keyboard/blob/master/rc_keyboard.py

userInput.py
Code used to implement getch() function, allowing keycapture without pressing return
https://www.jonwitts.co.uk/archives/896