Wired System Communications

The communications within the distributed system are key to its successful operation, without communication channels there is no distribution.

The different microcontrollers being used within the system have differing capabilities. The Arduino Nano has no native wireless capability, and only one hardware UART which is connected to the USB port on the development board. The ESP32 has Wi-Fi, Bluetooth, BLE, and 3 hardware UARTs. A hardwired network was essential as the Arduino devices would need to be supported by a much more capable device in order to enable them to communicate wirelessly, which would render them redundant in the system.

When considering the application the system was designed for (multi-level, multi-room, installed hardware system) a hybrid of wired and wireless networks provides a level of redundancy and resilience within the system. This enables the hardware network to continue to function should wireless communications be disrupted, albeit with slightly diminished functionality. The core functions would still be able to operate.

The options considered when specifying the system were: Wi-Fi, Blue tooth, ESPnow, UART serial, I2C, SPI.

I2C: Inter Integrated Circuit (I2C, I2C) is a 2 wire communication protocol data line (SDA) and clock line (SCL), this allows for synchronised communications between devices at up to 100Kbps. The system uses 7 bit addressing allowing up to 127 devices to be on a single bus. I2C is a master/slave interface allowing a master to communicate with multiple slaves, it is possible to set up as a multi-master bus. At 100 KHz the maximum length of the bus is 1 meter, if the speed is reduced to 10KHz this can be extended to 10 meters assuming a suitably low capacitance bus.

I2C was discounted as a communication channel due the length restrictions on the bus, as the system was specified to be a multi-room system.

SPI: Serial Peripheral Interface (SPI) is a 4 wire synchronous serial protocol with speeds up to MHz. It is a master/slave system where the master generates the clock signal. Slaves are unable to initiate communications. There is no multi-master provision within SPI. Again SPI is a short range protocol designed primarily for communications within a device or PCB.

SPI was discounted due to the bus length restriction, and the single master requirement, all nodes within the system need to be able to initiate communication.

Wi-Fi was chosen as a channel for the communications between the ESP32 devices and the cloud-based database and dashboard processing elements of the system.

Bluetooth was chosen as a serial communication channel allowing an interface between the access gate and android cell phone.

ESPnow is a proprietary protocol allowing ESP32 devices to communicate wirelessly with each other. This was not considered as other wireless options were preferred.

UART serial: Using UART communications allows all of the non-cloud based nodes to communicate with each other, the Nanos were able to use a software serial port mapped to a pair of GPIO pins, and the ESP devices had spare hardware UART capacity.

The transmission protocol chosen was RS485, as RS232 has a limited range and is primarily a protocol used between a pair of devices. RS485 is a protocol that takes the serial output of the UART and uses a transceiver to convert it into a differential pair of signals. The transceiver has TX & RX pins connected to the respective UART pins, and an active high TX enable pin, and an active low RX enable pin. The two enable pins are tied together on the breadboard and driven by a single chip select pin on the MCU. It is important to connect these pins to ground on any unconnected transceiver boards on the network as they default to high. The schematic for the transceiver boards is given in figure 1 below. The 120 Ω resistor R7, is a termination resistor designed to match the characteristic impedance of the transmission line used for the bus, this ensures that the line sees an impedance matched load and prevents reflections coming back down the line. Its close but the characteristic impedance of both cat 5, and cat 6 cable is 100 Ω. In practice the test circuit demonstrates no adverse effects on the signal quality due to this. As the termination resistor should only be fitted at the ends of the line R7 was removed from all but two of the transceiver boards.

A diagram of a circuit

Description automatically generated

figure RS485 transceiver circuit

Combined with the physical construction of the bus (a pair of shielded, twisted wires) gives the system very good immunity to noise. This translates to a maximum bus length of 1200 meters. The maximum number of devices able to use the bus depends on the loading characteristics of the transceivers used, with the standard transceiver the maximum is 32 devices although this can be increased to 256 by using 1/8th unit load transceivers. The bus and transceivers are a 5 V system. This is fine for the Arduinos which are 5 V devices, the ESP32 however is a 3.3 V device so a MOSFET level shifter is used to interface these devices to the bus. The level shifter has a 3.3 V input and a 5V input, a 3.3 V signal put in comes out at 5 V and vice-versa. Figure 2 below shows the result of testing the level shifter before using in the circuit. The pink trace is the 3.3 V input, and the yellow trace is the corresponding 5 V output.

A screen shot of a computer

Description automatically generated

figure Testing the operation of the level shifter

Figure 3 below shows the serial bus and gate nodes with level shifters

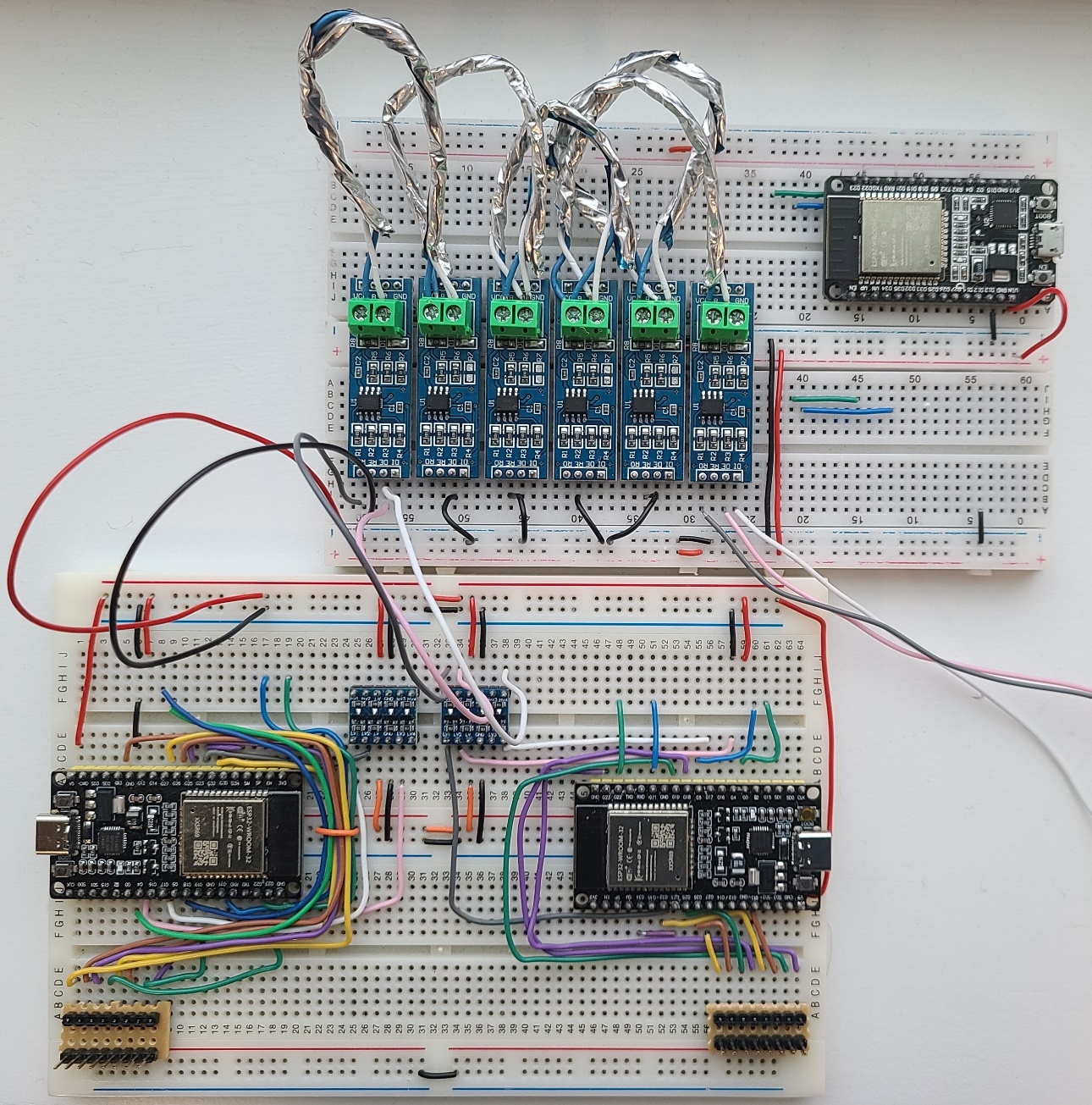


figure The serial bus test circuit

In order to maintain the communication signal integrity a communication software protocol was developed to set out a standard message frame and access conditions to the serial bus. This ensures that all devices connected to the network are following the same rules, to minimise data corruption/loss and avoid collisions on the bus. The protocol was saved to a folder containing Communication\_Protocol.cpp, Communication\_Protocol.h, and a ReadMe detailing how to use the protocol. This folder could then be added to all node code folders for a unified communication system.

The message frame is set out as:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Start byte | Sender address | Destination address | Sender node type | Payload length | Up to 33 payload characters | Checksum byte | End byte |

The checksum is calculated by a pair of functions at the TX and RX ends of the communications, and are an error detection byte. This combined with the start (0x02) and end (0x03) bytes help to ensure the data that is read is not corrupted.

The communication Protocol is made up of the following variable and functions:

struct TX\_Payload {

unsigned char length;

char message[35];

};

This is the standard variable used to create serial messages e.g. const struct TX\_Payload Alive = {8, "I'm here"};

There are four user functions which allow others writing code for nodes to simply use the protocol. These are:

1. void Comms\_Set\_Up();

This function is called to setup all of the hardware required for the serial communications. It uses compiler preprocessor commands to determine if the device is an Esp or Arduino and sets up the appropriate serial object, the pins (listed as #defines in the .h file), and the bus monitor interrupt. The Communication\_Protocol.cpp file also contains all of the common variables required for the system to operate, leaving the user to only declare the messages they wish to send.

1. void Transmit\_To\_Bus(struct TX\_Payload\* data, unsigned char\* message = TX\_Message);

This function is called with a pointer to the message payload to be sent, the second parameter does not need to be filled by the user. E.g. Transmit\_To\_Bus(&Alive);

The function will call the assemble message function (which calls the Calculate\_Checksum function), will then check if the bus is idle and safe to transmit on and transmit the message onto the bus, taking care of the writing and chip enable pin on the transceiver.

1. unsigned char Read\_Serial\_Port();

This function is called to read the incoming serial messages. It reads the port, and decodes the message when it is fully received, copying the data contained into the previously set up variables to which the decode function is pointed.

The function returns the address to which the serial communication was intended for.

1. void Acknowledge(unsigned char\* dest, unsigned char\* message = Ack\_message );

This function can be called with a pointer to Sender\_Address to send a pre-recorded acknowledgement byte back to the message origin.

These functions are primarily designed to be called within the functions defined above:

1. void Assemble\_Message(struct TX\_Payload\* data, unsigned char\* message);

This function put together the message string for transmission it is passed the message to be sent and the transmission char array address

1. unsigned char Calculate\_Checksum(struct TX\_Payload\* data);
2. unsigned char Calculate\_RX\_Checksum(unsigned char\* data, unsigned char length);
3. unsigned char Decode\_Message(unsigned char\* message, unsigned char\* Sender\_Address, unsigned char\* Sender\_Node\_Type, unsigned char\* payload);

The decode message function is designed to read the message check for evidence of corruption, and sort the important data into variables accessible within the main body of the code.

1. bool Collision\_Avoidance();

The clear to send function is designed to prevent data collision. It does this by setting a bus busy flag to 0. The flag is set by a change interrupt on a pin connected to the UART RX pin.

There is a delay of 1 ms (9.6 bit periods at 9600 baud. Long enough to guarantee picking up a start or stop bit), the flag is then checked. If the flag is not set then the function exits. If the flag is set indicating a transmission on the bus the function delays for a random period between 50 and 100 ms (the maximum message length is 40 characters taking 41.6 ms) then reruns. The delay period increases exponentially up to 10 s.

1. bool Clear\_To\_Send();

This function calls the collision avoidance function, if this returns 1 then a random back off delay between 1 & 10 ms is set before the function exits allowing the transmission to send.

1. void Bus\_Monitor\_Pin\_interrupt();

Sets the bus busy flag on a rising or falling edge of the monitor pin.

1. void Comms\_Setup\_Nano(struct Set\_Up\_Pins\* pin);
2. void Comms\_Setup\_Esp(struct Set\_Up\_Pins\* pin);
3. void Board\_Select(struct Set\_Up\_Pins\* pin);

Called by the comms set up function and checks which device is being used

1. void Introduction();

This function is designed to be called within the coms set up and transmits an introduction message to the network. It then waits for a reply to its default address from the server setting its location and address parameters.

1. void Forward\_Messasage();
2. void Print\_Message(unsigned char\* message, unsigned char length);
3. void print\_Struct(struct Set\_Up\_Pins\* message);

13 & 14 used for debugging purposes

The protocol has provided a stable and effective serial communications network, with the five nodes attached to the bus exchanging data and commands with no detectable collisions. Collisions are signified by the "Invalid start or end byte" message printed to the terminal.

The capability of the ESP32 to run Bluetooth was employed by the gate access node to allow users to bypass the key entry and log in or out using a Bluetooth app on an android mobile device.

The communication system was built up and each sub-system was tested on the network with functions altered, and new functions added as the need arose. The peripheral monitoring and communications were all set to run on core 0 of the ESP devices leaving core 1 for the Wi-Fi and MQTT message handling code. The biggest problem for the communication system was discovered with the integration of the Wi-Fi.

The first big issue was that the system would repeatedly crash on startup after extensive debugging and research it was determined that the issue was the Bluetooth transmitter and Wi-Fi were using the same antenna on the chip, and while it is theoretically possible to set up the timings of both these systems to share the antenna successfully there was not the time available for this to be done.

The solution was to add a second ESP device to the gate node to run the Bluetooth. This second device sets up the Bluetooth channel, the user send their access code over Bluetooth. The Bluetooth receiver transmits the code over an I2C bus to the gate node which processes the code and returns a value which determines the response sent to the users Bluetooth device. The timings of the data exchange proved to be very important, however once they were correctly set the Bluetooth sub-system is a very stable and user friendly platform.

The second bug introduced by the wireless integration was the total loss of serial comms on the affected device. The first symptom was the RX interrupt triggering the RX pin was probed with a scope to try and determine what was happening this showed a low signal with a repeated pattern regular spikes every 100 ms with a short break between. The signal line was traced through the level shifter and the RS485 transceiver. These devices were ruled out as the cause of the fault. The link between the monitor pin and the RX pin was removed, and the fault was discovered to be two faults. The Voltage spikes triggering the RX interrupt were traced to the monitor pin, and the ESP device was removed from the circuit to rule out any interference being picked up from the breadboard or wiring. The RX pin was discovered to be held low when it should be high when idle. The monitor pin trace is shown in figure 4 below.

A screenshot of a computer

Description automatically generated

figure Signal trace of bus monitor pin

The bus monitor pin(GPIO 2) is set as an input in the setup and worked perfectly as did the UART2 RX pin (GPIO 16) before the addition of the Wi-Fi code which did not set any pins. It was concluded that there must be some behind the scenes use of these two pins by the Wi-Fi setup. Once this was discovered it was a simple matter to remap the affected pins, and the serial communication network was restored.