```
#include <MD_MAX72xx.h>
#include <MD_Parola.h>
#include <SPI.h>
#include <WiFi.h>
#include <esp_now.h>

////////////////////
#define MAX_DEVICES 2 ///< The maximum number of devices.

#define CLK_PIN 18  ///< The pin for the clock signal.
#define DATA_PIN 23 ///< The pin for the data signal.
#define CS_PIN 5    ///< The pin for the chip select signal.

#define HARDWARE_TYPE                                      \
    MD_MAX72XX::PAROLA_HW ///< The type of hardware being
                          ///< used.

MD_Parola P = MD_Parola(
    HARDWARE_TYPE, DATA_PIN, CLK_PIN, CS_PIN,
    MAX_DEVICES); ///< Instance of the MD_Parola class.
int g_iStop = 0;  ///< Stop variable.

uint8_t g_uiScrollSpeed = 25; ///< The speed of the scroll.
textEffect_t g_eScrollEffect =
    PA_SCROLL_LEFT; ///< The effect of the scroll.
textPosition_t g_eScrollAlign =
    PA_LEFT; ///< The alignment of the scroll.
uint16_t g_uiScrollPause =
    200; ///< The pause time of the scroll in milliseconds.
#define BUF_SIZE 75 ///< The size of the buffer.

char g_szCurMessage[BUF_SIZE] = {
    ""}; ///< The current message.
char g_szNewMessage[BUF_SIZE] = {
    "OUT OF SERVICE."}; ///< The new message.
bool g_bNewMessageAvailable =
    true; ///< Flag to check if a new message is available.
char g_szBuffer[3] = " "; ///< Buffer to hold the numbers.

uint8_t g_auiBroadcastAddress[] = {
    0xEC, 0x64, 0xC9,
    0x86, 0x13, 0xC0}; ///< The broadcast address.

int g_iFlag = 0;            ///< Flag variable.
unsigned long g_ulTime;     ///< Time variable.
unsigned long g_ulTimeLast; ///< Last time variable.
bool g_bService = true;     ///< Service flag.
int g_iHalt = 0;            ///< Halt variable.
int g_iService = 0;         ///< Service variable.
int g_iInitial = 0;         ///< Initial variable.

int g_iStopID = 2; ///< Unique ID of the operating stop.
```

```
const int g_iBusNum = 1;      ///< Total number of buses.
const int g_iStopNum = 2;     ///< Total number of stops.
int g_aiEstimated[g_iBusNum]; ///< Array to store estimated
                              ///< times in seconds.
int g_aiScheduled[g_iStopNum][g_iStopNum] = {
    {100, 50}, {50, 100}}; ///< 2D array to store scheduled
                           ///< times in seconds.

esp_now_peer_info_t g_peerInfo; ///< Peer information.

bool g_bHalt = false; ///< Halt flag.
int g_iShowTime = 0;  ///< Show time variable.

uint8_t g_auiBusMAC[] = {
    0xE8, 0x6B, 0xEA, 0xD4,
    0x27, 0x7C}; ///< MAC address of the receiver (bus).

/**
 * @struct Message
 * @brief Structure to send data.
 *
 * @var Message::buttonPressed
 * @brief Button pressed flag.
 */
typedef struct
{
    bool buttonPressed;
} Message;
volatile bool g_bButton =
    false; ///< Flag to indicate if the button is pressed.

/**
 * @brief Interrupt service routine for handling button
 * press.
 *
 * This function is called when an interrupt is triggered.
 * It sets the button flag to true.
 *
 * @note This function is placed in IRAM, which is a region
 * of RAM that is directly accessible by the CPU.
 */
void IRAM_ATTR handleInterrupt()
{
    g_bButton = true;
}

int g_iButtonPin = 0; ///< The pin connected to the button.
//////////////////////////////////////

// void UpdateInterrupt(int nBusIndex,int nStopIndex);
// int  UpdateEstimated();
// void SendRequest(int *retarr);

/**
```

```
 * @brief Sends data to a peer device over ESP-NOW.
 *
 * @param peer_addr The MAC address of the peer device.
 * @param data The data to be sent.
 * @param len The length of the data.
 *
 * This function sends data to a peer device over ESP-NOW.
 * It prints a success message to the serial port if the
 * data is sent successfully, otherwise it prints an error
 * message.
 */
void send_data(const uint8_t *pucPeerAddr,
               const uint8_t *pucData, size_t uiLen)
{
    esp_err_t eResult =
        esp_now_send(pucPeerAddr, pucData, uiLen);

    if (eResult == ESP_OK)
    {
        Serial.println("Sent with success");
    }
    else
    {
        Serial.println("Error sending the data");
    }
}

/**
 * @brief Updates the estimated time of arrival for a bus at
 * a stop.
 *
 * @param nBusIndex The index of the bus.
 * @param nBusStop The index of the bus stop.
 * @param nStopID The ID of the stop.
 *
 * This function updates the scheduled times for the given
 * bus when an interrupt occurs.
 */
void UpdateInterrupt(int iBusIndex, int iBusStop,
                     int iStopID)
{
    // On Interrupt update the scheduled times for the given
    // bus
    g_aiEstimated[iBusIndex] =
        g_aiScheduled[iBusStop][iStopID];
}

/**
 * @brief Updates the estimated time of arrival for all
 * buses and returns the minimum.
 *
 * @return The minimum estimated time of arrival.
 *
 * This function decrements the estimated times and takes
 * the minimum.
 */
int UpdateEstimated()
{
    // Decrementing the times and taking minm
    for (int iBusID = 0; iBusID < g_iBusNum; iBusID++)
    {
        g_aiEstimated[iBusID] =
            max(0, g_aiEstimated[iBusID] - 1);
    }
    int iMinm = g_aiEstimated[0];
    for (int iBusID = 0; iBusID < g_iBusNum; iBusID++)
    {
        iMinm = min(iMinm, g_aiEstimated[iBusID]);
    }
    return iMinm;
}

/**
 * @struct struct_message
 * @brief Structure for sending and receiving data.
 *
 * @var struct_message::a
 * @brief A string message.
 * @var struct_message::nBusStop
 * @brief The index of the bus stop.
 * @var struct_message::nBusIndex
 * @brief The index of the bus.
 * @var struct_message::nDirection
 * @brief The direction of the bus.
 *
 * This structure must match the sender structure.
 */
typedef struct struct_message
{
    String strMessage;
    int iBusStop;
    int iBusIndex;
    int iDirection;
} struct_message;
// Create a struct_message called myData

struct_message g_SendData;    ///< Data to send.
struct_message g_RecieveData; ///< Data received.
// callback function that will be executed when data is
// received
/**
 * @brief Callback function that is executed when data is
 * received.
 *
 * This function handles the incoming data and performs
 * different actions based on the received message. It
 * updates the status of the bus (halted, moving, out of
 * service, etc.) and the estimated time of arrival.
 *
 * @param mac The MAC address of the sender.
 * @param incomingData The incoming data.
 * @param len The length of the incoming data.
 */
void OnDataRecv(const uint8_t *pucMac,
                const uint8_t *pucIncomingData, int iLen)
{
    g_iInitial = 1;
    memcpy(&g_RecieveData, pucIncomingData,
           sizeof(g_RecieveData));
    Serial.println("received");
    Serial.println(g_RecieveData.strMessage);
    if (g_RecieveData.iBusStop != 0)
    {
        if (g_RecieveData.strMessage == "BUS OOS")
        {
            g_bService = false;
            g_iService = 0;
        }
        else if (g_RecieveData.strMessage == "BUS NOOS")
        {
            g_bService = true;
        }
        else if (g_RecieveData.strMessage == "BUS HALTED")
        {
            g_bHalt = true;
            g_iHalt = 0;
        }
        else if (g_RecieveData.strMessage == "BUS MOVING")
        {
            g_bHalt = false;
        }
        else
        {
            g_iInitial = 1;
            Serial.println("Bus Departing from other stop");
            if (g_RecieveData.iDirection == 0)
            {
                UpdateInterrupt(g_RecieveData.iBusIndex - 1,
                                g_RecieveData.iBusStop - 1,
                                g_iStopID - 1);
            }
            else
            {
                UpdateInterrupt(g_RecieveData.iBusIndex - 1,
                                g_iStopID - 1,
                                g_RecieveData.iBusStop - 1);
            }
        }
    }
    if (g_RecieveData.iBusStop == 0)
    {
        if (g_RecieveData.strMessage == "BUS STOPPED!")
        {
            g_iStop = 1;
            Serial.println("BUS ARRIVED!");
            // nothing
        }
        else if (g_RecieveData.strMessage == "BUS STARTED!")
        {
            g_iInitial = 1;
            g_iStop = 0;
            Serial.println("BUS DEPARTING!");
            g_SendData.strMessage = "BusStop A";
            g_SendData.iBusStop = g_iStopID;
            g_SendData.iBusIndex = g_RecieveData.iBusIndex;
            g_SendData.iDirection =
                g_RecieveData.iDirection;
            // Send_Data.from_where=0;
            UpdateInterrupt(g_RecieveData.iBusIndex - 1,
                            g_iStopID - 1, g_iStopID - 1);
            // flag = 1;
            delay(1000);
            send_data(g_auiBroadcastAddress,
                      (uint8_t *)&g_SendData,
                      sizeof(g_SendData));
            /*esp_err_t result =
            esp_now_send(broadcastAddress, (uint8_t *)
            &Send_Data, sizeof(Send_Data));

            if (result == ESP_OK) {
              Serial.println("Sent with success");
            }
            else {
            Serial.println("Error sending the data");
            }*/
        }

        else if (g_RecieveData.strMessage == "BUS HALTED")
        {
            g_bHalt = true;
            g_iHalt = 0;
            Serial.println("BUS HALTED");
            g_SendData.strMessage = "BUS HALTED";
            g_SendData.iBusStop = g_iStopID;
            g_SendData.iBusIndex = g_RecieveData.iBusIndex;
            g_SendData.iDirection =
                g_RecieveData.iDirection;
            send_data(g_auiBroadcastAddress,
                      (uint8_t *)&g_SendData,
                      sizeof(g_SendData));
        }
        else if (g_RecieveData.strMessage == "BUS MOVING")
        {
            g_bHalt = false;
            Serial.println("BUS MOVING");
            g_SendData.strMessage = "BUS MOVING";
            g_SendData.iBusStop = g_iStopID;
            g_SendData.iBusIndex = g_RecieveData.iBusIndex;
            g_SendData.iDirection =
                g_RecieveData.iDirection;
            send_data(g_auiBroadcastAddress,
                      (uint8_t *)&g_SendData,
                      sizeof(g_SendData));
        }

        else if (g_RecieveData.strMessage == "BUS OOS")
        {
            g_bService = false;
            g_iService = 0;
            Serial.println("BUS OOS");
            g_SendData.strMessage = "BUS OOS";
            g_SendData.iBusStop = g_iStopID;
            g_SendData.iBusIndex = g_RecieveData.iBusIndex;
            g_SendData.iDirection =
                g_RecieveData.iDirection;
            send_data(g_auiBroadcastAddress,
                      (uint8_t *)&g_SendData,
                      sizeof(g_SendData));

            /* esp_err_t result =
             esp_now_send(broadcastAddress, (uint8_t *)
             &Send_Data, sizeof(Send_Data));

             if (result == ESP_OK) {
               Serial.println("Sent with success");
             }
             else {
             Serial.println("Error sending the data");
             }*/
        }
        else
        {
            g_bService = true;
            Serial.println("BUS NOOS");
            g_SendData.strMessage = "BUS NOOS";
            g_SendData.iBusStop = g_iStopID;
            g_SendData.iBusIndex = g_RecieveData.iBusIndex;
            g_SendData.iDirection =
                g_RecieveData.iDirection;
            send_data(g_auiBroadcastAddress,
                      (uint8_t *)&g_SendData,
                      sizeof(g_SendData));

            /*esp_err_t result =
            esp_now_send(broadcastAddress, (uint8_t *)
            &Send_Data, sizeof(Send_Data));

            if (result == ESP_OK) {
              Serial.println("Sent with success");
            }
            else {
            Serial.println("Error sending the data");
            }*/
        }
    }
}

/**
 * @brief Callback function that is executed when data is
 * sent.
 *
 * This function prints the status of the last packet sent.
 * It prints "Delivery Success" if the packet was sent
 * successfully, otherwise it prints "Delivery Fail".
 *
 * @param mac_addr The MAC address of the receiver.
 * @param status The status of the send operation.
 */
void OnDataSent(const uint8_t *pucMacAddr,
                esp_now_send_status_t eStatus)
{
    Serial.print("\r\nLast Packet Send Status:\t");
    Serial.println(eStatus == ESP_NOW_SEND_SUCCESS
                       ? "Delivery Success"
                       : "Delivery Fail");
}
/**
 * @brief Setup function for the Arduino sketch.
 *
 * This function initializes the serial monitor, sets the
 * device as a Wi-Fi Station, initializes ESP-NOW, registers
 * callback functions, sets up the button interrupt, and
 * initializes the display.
 */
void setup()
{
    // Initialize Serial Monitor
    Serial.begin(115200);

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_AP_STA);
    WiFi.softAP("BUS_STOP_B", NULL, 4);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK)
    {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    memcpy(g_peerInfo.peer_addr, g_auiBroadcastAddress, 6);
    g_peerInfo.channel = 4;
    g_peerInfo.encrypt = false;
    if (esp_now_add_peer(&g_peerInfo) != ESP_OK)
    {
        Serial.println("Failed to add peer");
        return;
    }

    // Once ESPNow is successfully Init, we will register
    // for recv CB to get recv packer info
    esp_now_register_recv_cb(OnDataRecv);
    esp_now_register_send_cb(OnDataSent);

    ///////////////////////////////////////////////////////////////

    pinMode(g_iButtonPin, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(g_iButtonPin),
                    handleInterrupt, FALLING);

    // Define and add the peer (the bus ESP32)
    esp_now_peer_info_t bus_peer;
    memset(&bus_peer, 0,
           sizeof(bus_peer)); // Clear peer structure
    memcpy(bus_peer.peer_addr, g_auiBusMAC,
           6);                // Set peer MAC address
    bus_peer.channel = 4;     // Use default channel
    bus_peer.encrypt = false; // No encryption

    if (esp_now_add_peer(&bus_peer) != ESP_OK)
    {
        Serial.println("Failed to add peer");
        return;
    }

    /////////////////////////////////////////////////////////////////

    // Initialize the display
    P.begin();
    P.setIntensity(1); // keep it 3 or below as we are
                       // powering off the chip/usb
    P.displayText(g_szBuffer, PA_CENTER, 0, 0, PA_PRINT,
                  PA_NO_EFFECT);

    g_ulTimeLast = millis();
}
/**
 * @brief Scrolls the text on the display.
 *
 * This function animates the display to scroll the text. It
 * resets the display after each scroll.
 */

void scroll()
{
    if (P.displayAnimate())
    {
        P.displayText(g_szNewMessage, g_eScrollAlign,
                      g_uiScrollSpeed, g_uiScrollPause,
                      g_eScrollEffect, g_eScrollEffect);
        P.displayReset();
    }
}

/**
 * @brief Main loop function for the Arduino sketch.
 *
 * This function continuously checks if the button is
 * pressed and sends a message if it is. It also updates the
 * estimated time of arrival and displays it on the screen.
 * If the bus is halted or out of service, it displays the
 * appropriate message.
 */
void loop()
{
    if (g_bButton == true)
    { // If the boot button is pressed
        Message msg;
        msg.buttonPressed = true;
        esp_err_t result = esp_now_send(
            g_auiBusMAC, (uint8_t *)&msg, sizeof(msg));

        if (result == ESP_OK)
        {
            Serial.println("Sent with success");
        }
        else
        {
            Serial.println("Error sending the data");
        }
        delay(100); // Debounce delay
        g_bButton = false;
    }


//////////////////////////////////////////////////////////////////////////////////
////

    if (!g_bHalt)
    {
        g_iShowTime = UpdateEstimated();

        Serial.println(g_iShowTime);
        if (P.displayAnimate())
        {
            int s = (g_iShowTime / 60) + 1;
            sprintf(g_szBuffer, "%01d",
                    g_iShowTime); // the number "1" is the
                                  // number of digits that
                                  // you want always shown
                                  // on the screen for this
                                  // font we can have 5
            P.displayText(g_szBuffer, PA_CENTER, 0, 0,
                          PA_PRINT, PA_NO_EFFECT);
        }

        delay(1000);
    }

    else
    {
        if (g_iHalt == 0)
        {
            Serial.println("BUS HALTED IN BETWEEN");
            g_iHalt = 1;
        }
        if (!g_bService)
        {
            if (g_iService == 0)
            {
                Serial.println("BUS OUT OF SERVICE");
                if (P.displayAnimate())
                {
                    P.displayText("OOS", PA_CENTER, 0, 0,
                                  PA_PRINT, PA_NO_EFFECT);

                    P.displayReset();
                }

                g_iService = 1;
            }
        }
    }
}
```