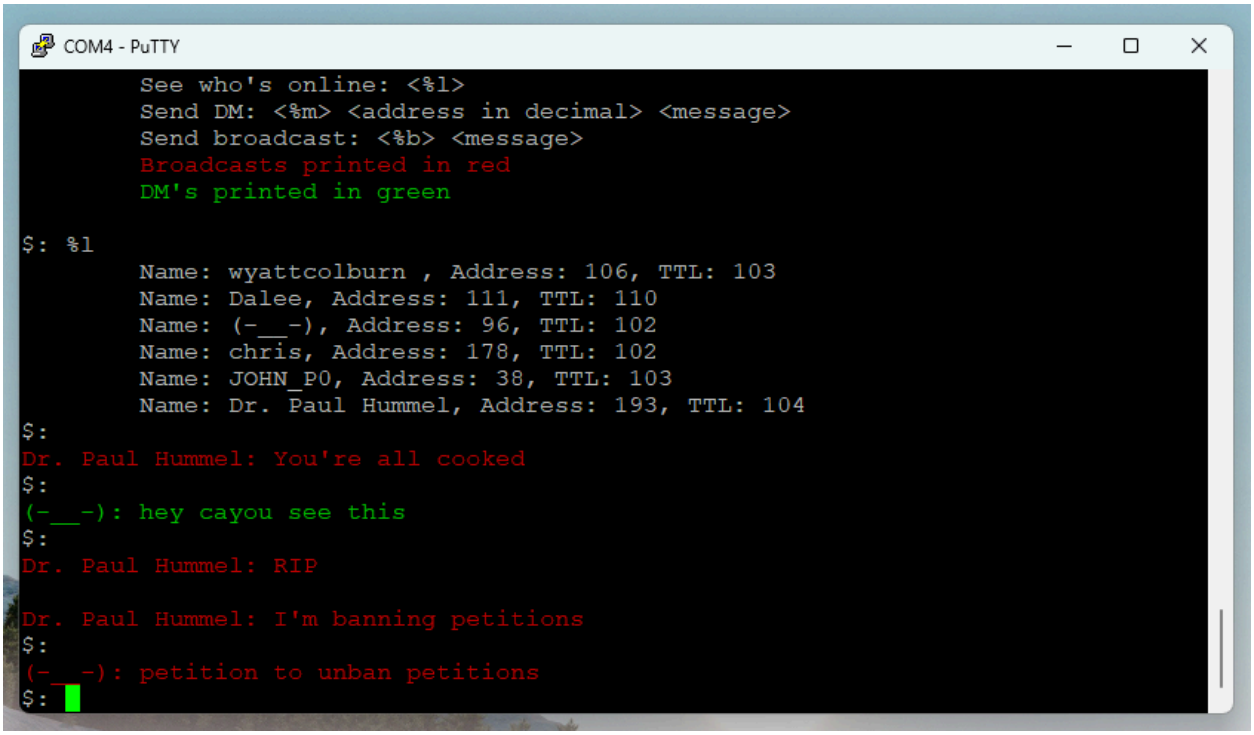


Wireless Chat

Diego Renato Curiel

21 May 2024

Behavior Description

The device I designed works as a node in a mesh network using the Spirit1 RF module and FreeRTOS on an STM32 microcontroller. Its main functions are node discovery and communication, message exchange, task management, user interaction, and data handling. Each node regularly sends out a "heartbeat" message to show it's active, helping other nodes keep track of who's online. The device can send direct messages to specific nodes or broadcast messages to the whole network, with each message including the type, sender's username, and content.

FreeRTOS is used to handle multiple tasks at the same time, like logging onto the network, receiving messages, updating the list of active nodes, sending heartbeat messages, and processing user inputs. To ensure smooth operation, different tasks are assigned different priorities. For example, tasks like logging onto the network and handling user input have higher priorities, while tasks like sending heartbeat messages or updating the list of active nodes have lower priorities. Task delays are used to manage these priorities and ensure that higher priority tasks can preempt lower priority ones when necessary.

Users interact with the device via a USART interface, letting them send commands to clear the screen, list active nodes, send direct messages, or broadcast messages. The device echoes user inputs and displays messages from other nodes on the terminal. The USART interrupt handler manages user inputs, including handling backspace for editing the input buffer. Node management functions handle adding, updating, and removing nodes based on their activity, keeping the mesh network intact.

The code is organized into several key modules. The main application (main.c) sets up the system, initializes FreeRTOS tasks, and starts the scheduler. The USART communication module (usart.c) handles UART initialization and communication, including user input and string transmission to the terminal. Node management (node.c) takes care of the linked list of active nodes. Utility functions (utils.c) provide helper functions for processing inputs, creating Protocol Data Units (PDUs), and managing message payloads. Program functionality is rudimentary, but robust.

Software Architecture

The software architecture for this project is designed to create a node in a wireless mesh network using the Spirit1 RF module and FreeRTOS on an STM32 microcontroller. The program is organized into several tasks managed by FreeRTOS, with each task handling a specific function.

First, the system goes through an initialization phase where the system clock, GPIO, SPI, and USART peripherals are configured. This setup ensures that the hardware and RTOS are ready for the concurrent execution of tasks. The main tasks created start with the LogOnTask, which broadcasts a logon message when the node starts to announce its presence to other nodes in the network. The ReceiveTask continuously listens for incoming messages, processes them to update the list of active nodes or display a message using a mutex for proper synchronization.

The UpdateListTask periodically checks and removes inactive nodes based on a timeout mechanism, maintaining an updated list of active nodes. The HeartbeatTask sends periodic heartbeat messages to inform other nodes that it is still active, helping to maintain network integrity. The InputTask handles user inputs received via the USART interface, processes commands such as sending direct messages or broadcasts, and updates the system accordingly.

A crucial part of the architecture is the USART interrupt service routine (ISR), which processes characters as they are typed, managing special cases like the carriage return for completing a message and the backspace for editing the input. The ISR interacts with the FreeRTOS message queue to send completed messages to the appropriate task for processing.

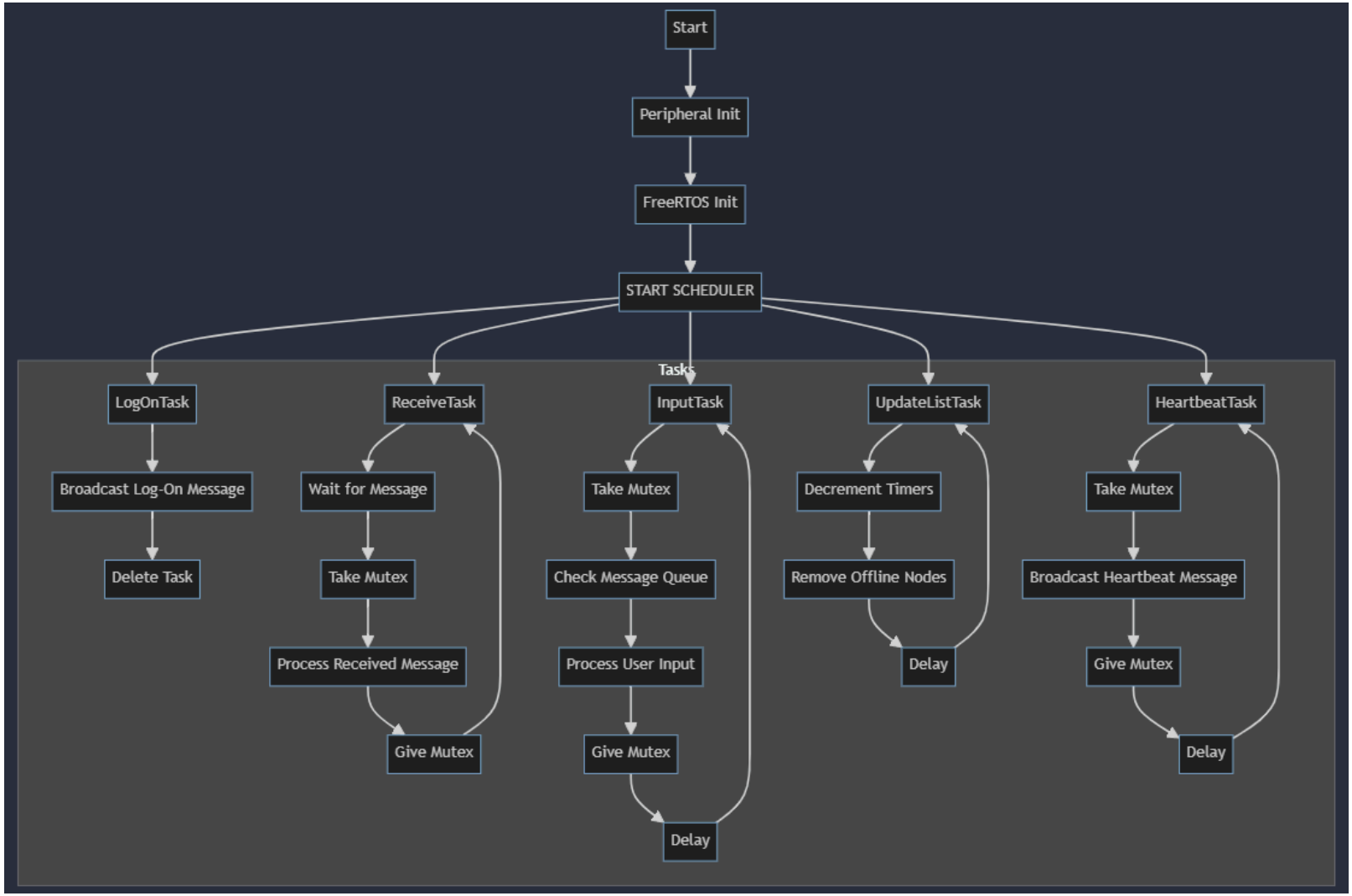
Message processing involves structuring messages into Protocol Data Units (PDUs) before transmission. Incoming messages are parsed to determine their type (e.g., logon, heartbeat, direct message), and appropriate actions are taken, such as updating the node list or displaying messages.

Node management is implemented using a linked list to track active nodes. Nodes are added when they log on, updated with each heartbeat, and removed when they are deemed inactive. This ensures that the network's state is always current and reliable.

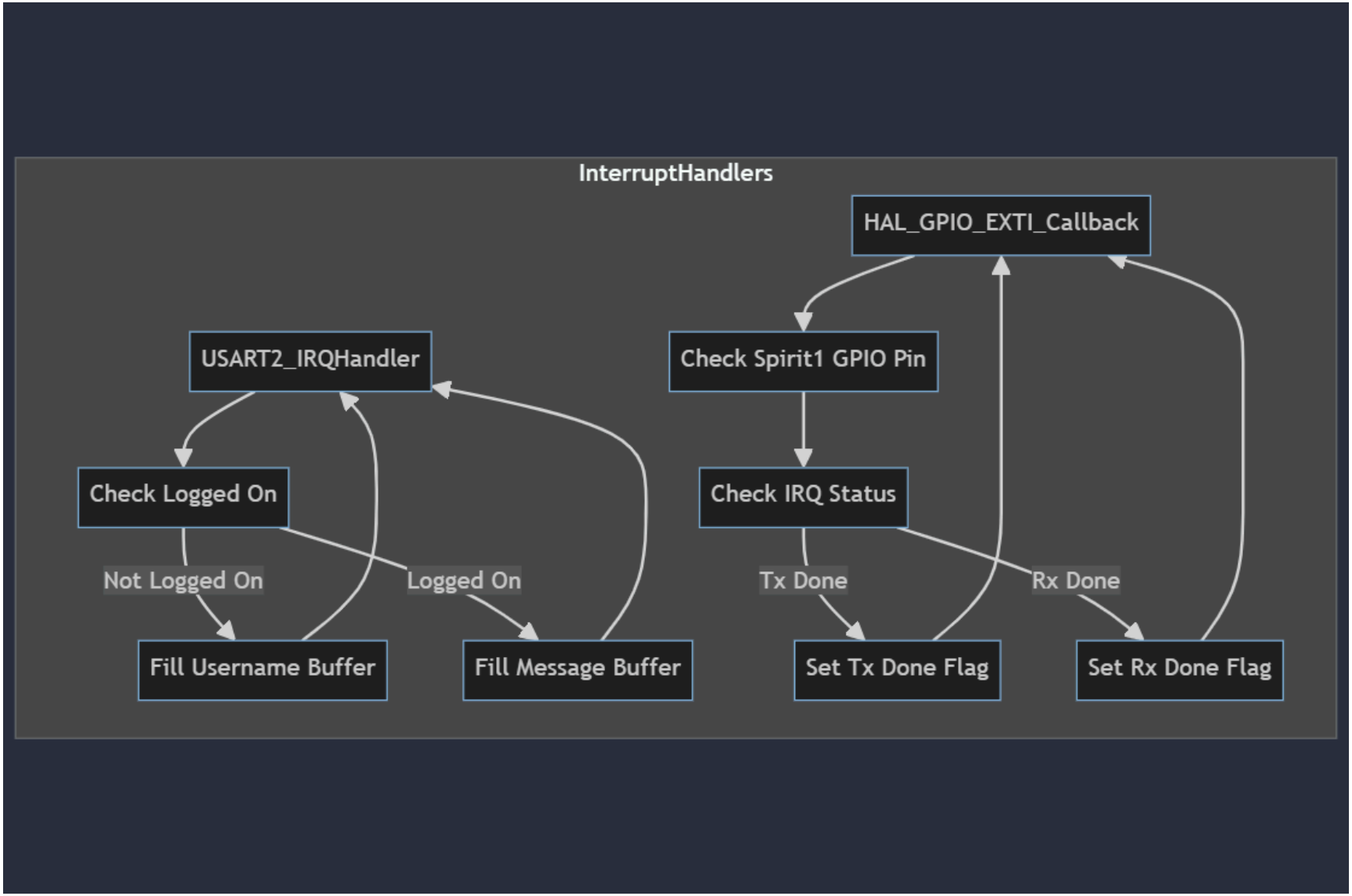
In the following section, there are flowcharts and state diagrams to visualize the program.There is one main diagram that holds all the tasks together, and then there is a diagram for each individual task.

Diagrams

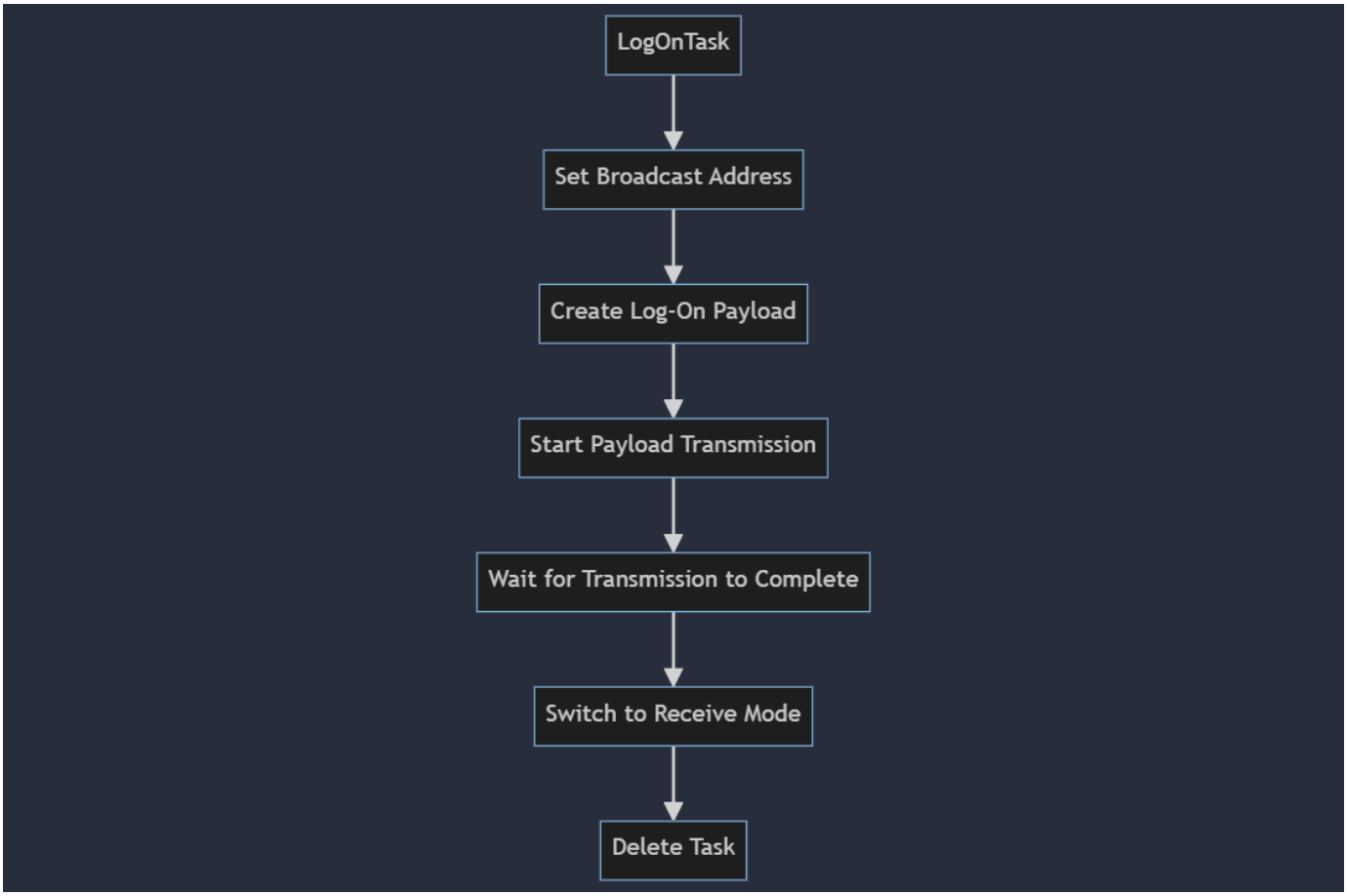
Main Program Flowchart: Behavior between tasks



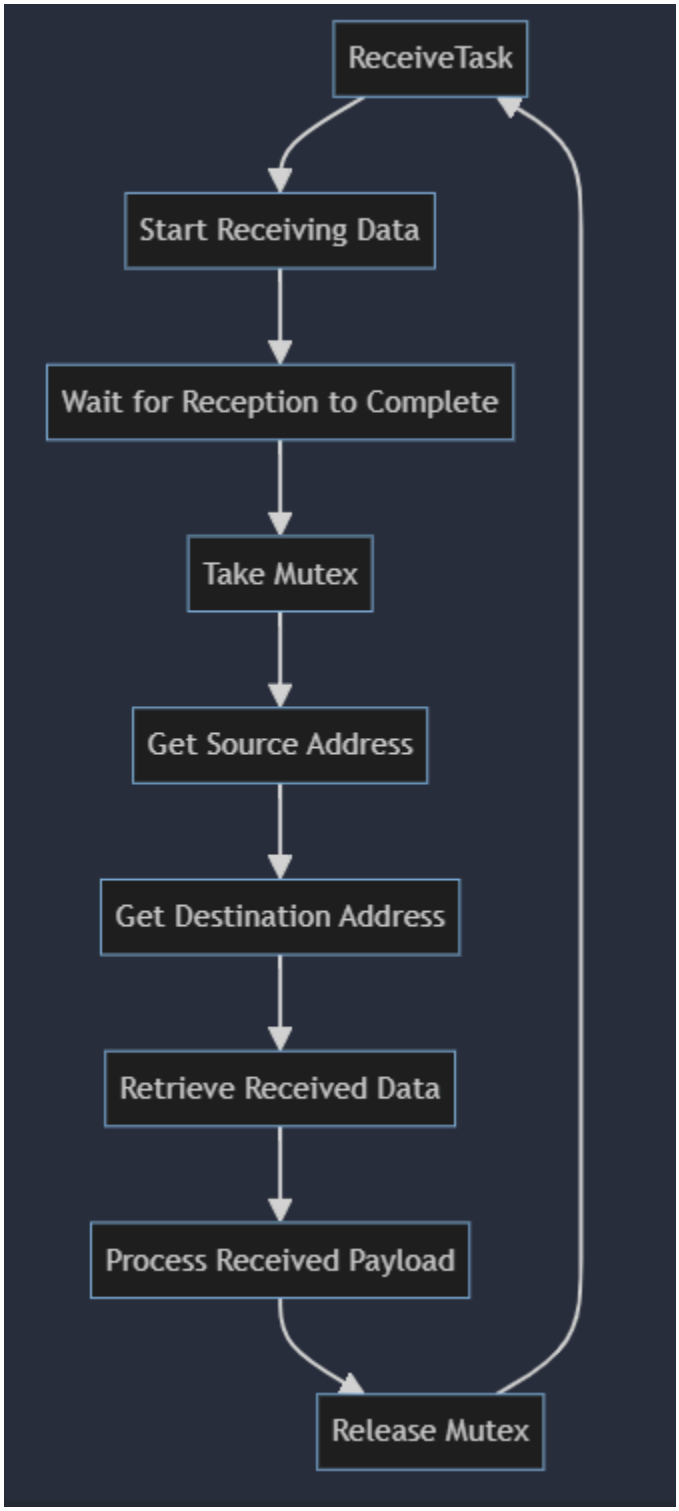
Interrupt Service Routines



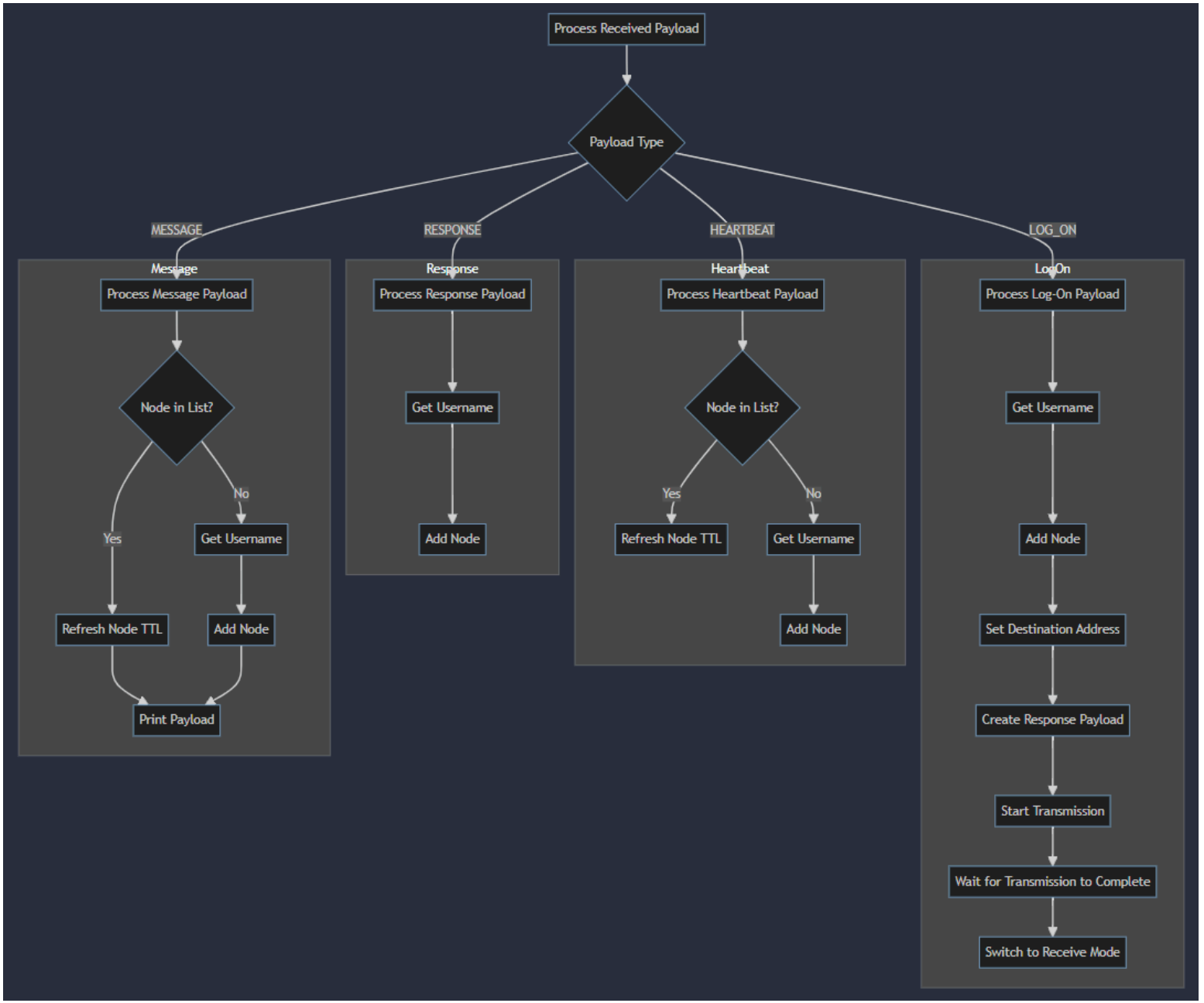
Log On Task


LogOnTask
Set Broadcast Address
Create Log-On Payload
Start Payload Transmission
Wait for Transmission to Complete
Switch to Receive Mode
Delete Task

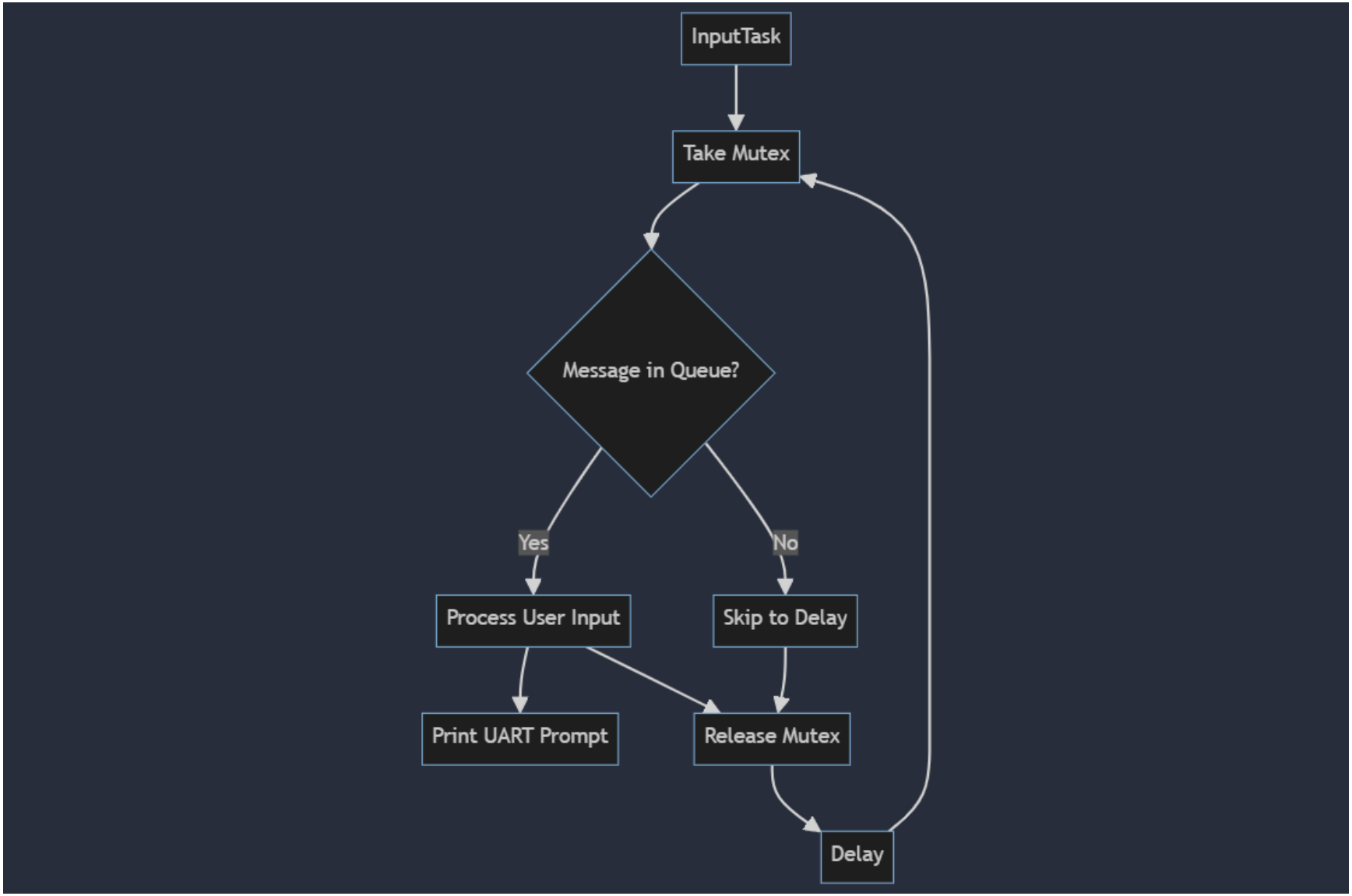
Receive Task



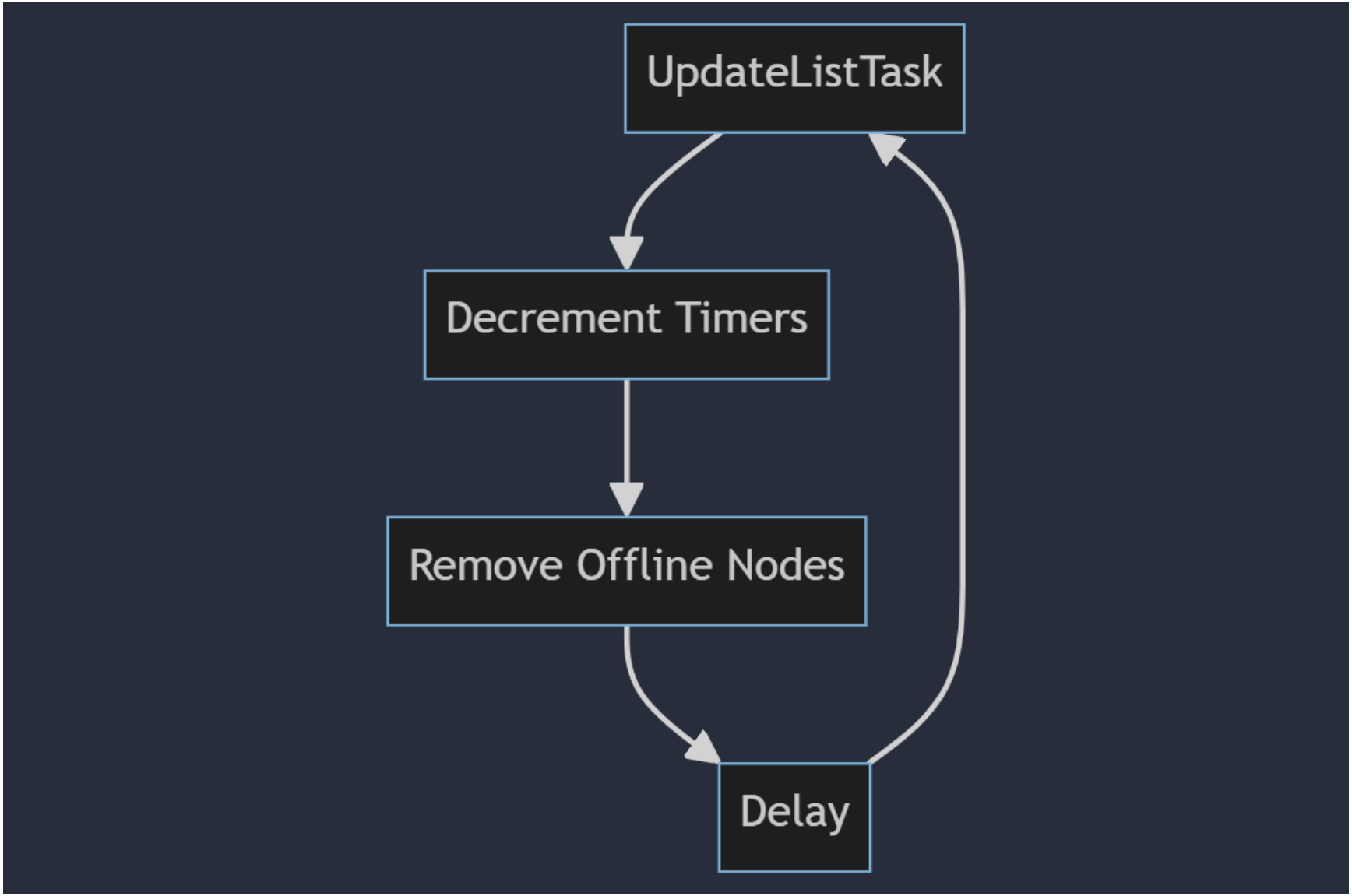
*Process Payload Logic *WITHIN THE RECEIVE*



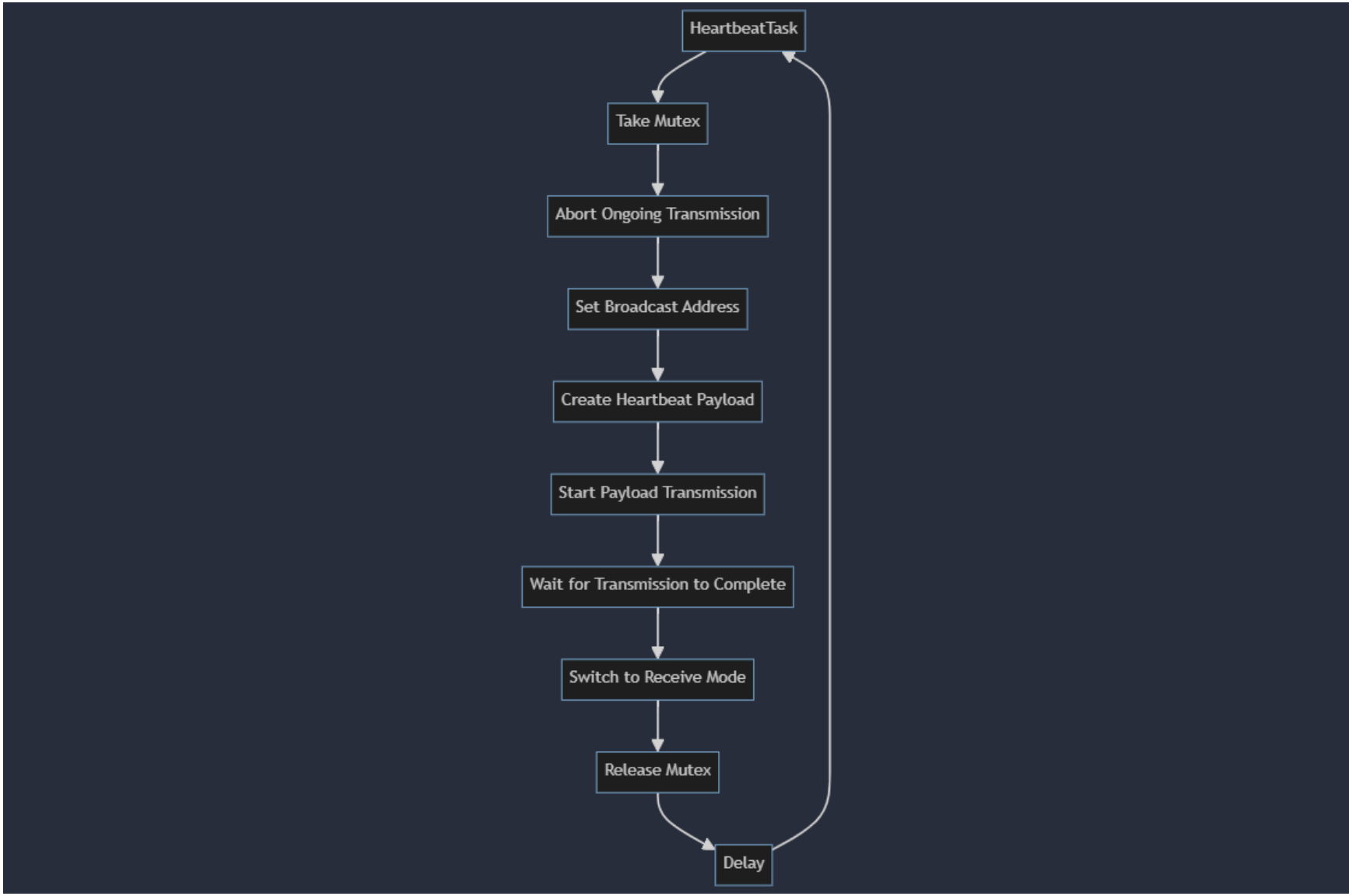
Input Task


InputTask
Take Mutex
Message in Queue?
Yes
No
Process User Input
Skip to Delay
Print UART Prompt
Release Mutex
Delay

Update List


UpdateListTask
Decrement Timers
Remove Offline Nodes
Delay

Heartbeat Task:



Code Listing

main.h:

```
#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C" {
  #endif

  #include "stm32l4xx_hal.h"

  typedef struct {
    char message[300];
    uint16_t len;
  }
  message_t;

  #define LOG_ON 0x1
  #define RESPONSE 0x2
  #define HEARTBEAT 0x3
  #define MESSAGE 0x4
  #define MY_ADDRESS 0x7F
  #define BROADCAST 0xFF
  #define MAXBUF 300

  void Error_Handler(void);

  #define SPIRIT1_GPIO3_Pin GPIO_PIN_7
  #define SPIRIT1_GPIO3_GPIO_Port GPIOC
  #define SPIRIT1_GPIO3_EXTI_IRQn EXTI9_5_IRQn
  #define SPIRIT1_SDN_Pin GPIO_PIN_10
  #define SPIRIT1_SDN_GPIO_Port GPIOA
  #define SPIRIT1_SPI_CSn_Pin GPIO_PIN_6
  #define SPIRIT1_SPI_CSn_GPIO_Port GPIOB

  extern char message[MAXBUF]; // for getting message from term
  extern char username[21]; // to keep my username
  extern uint8_t loggedOn; // flag to know whether or not I've logged on in USART
ISR
  extern uint16_t msgIndex; // for keeping track of message index in USART ISR

  #ifdef __cplusplus
}
#endif
#endif /* __MAIN_H */
```

main.c:

```
/* USER CODE BEGIN Header */
/**
 ******************************************************************************
 * @file           : main.c
 * @brief          : Wireless Ping
 * A FreeRTOS program that utilizes a Spirit1 RF module to be a part of a
 * mesh network designed to be a simple chat. Functionality consists of
 * sending private messages, sending broadcast messages, and viewing
 * which nodes are online. The UI is rudimentary and displayed over UART.
 ******************************************************************************
 **/

/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

#include "cmsis_os.h"

#include "spi.h"

#include "gpio.h"

#include "spsgrf.h"

#include "task.h"

#include "semphr.h"

#include "queue.h"

#include "node.h"

#include "usart.h"

#include "utils.h"

TaskHandle_t LogOnHandler, ReceiveHandler, UpdateListHandler, HeartbeatHandler,
InputHandler;
SemaphoreHandle_t xMutex;
QueueHandle_t xMessageQueue;

volatile SpiritFlagStatus xTxDoneFlag = S_RESET;
volatile SpiritFlagStatus xRxDoneFlag = S_RESET;
```

```c
volatile unsigned long ulHighFrequencyTimerTicks; // used for task runtime stats

char message[MAXBUF] = {
  '\0'
}; // for getting message from term
char username[21] = {
  '\0'
}; // to keep my username
uint8_t loggedOn = 0; // flag to know whether or not I've logged on in USART ISR
uint16_t msgIndex = 0; // for keeping track of message index in USART ISR

void SystemClock_Config(void);

/* Task Declarations */
void logOnTask(void * argument);
void receiveTask(void * argument);
void updateListTask(void * argument);
void heartbeatTask(void * argument);
void inputTask(void * argument);

/* Method Declarations */
void configureTimerForRunTimeStats(void);
void RTOS_Stats_Timer_Init(void);

int _write(int file, char * ptr, int len) // FOR PRINTF DEBUGGING
{
  for (int i = 0; i < len; i++)
    ITM_SendChar(( * ptr++));
  return len;
}

int main(void) {
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_SPI1_Init();

  configureTimerForRunTimeStats();
  SPSGRF_Init();
  USART_Init();
  UART_SendString(clear_screen);
  UART_SendString(move_cursor_home);
```

```
  getUsername();

  /* FreeRTOS Initialization */
  // Create Mutex
  xMutex = xSemaphoreCreateMutex();
  if (xMutex == NULL) {
    while (1);
  }

  // Create Message Queue
  xMessageQueue = xQueueCreate(2, sizeof(message_t));
  if (xMessageQueue == NULL) {
    while (1);
  }

  // Create tasks
  BaseType_t retVal;
  retVal = xTaskCreate(logOnTask, "LogOnTask", configMINIMAL_STACK_SIZE * 4, NULL,
tskIDLE_PRIORITY + 8, & LogOnHandler);
  if (retVal != pdPASS) {
    while (1);
  } // check if task creation failed

  retVal = xTaskCreate(receiveTask, "ReceiveTask", configMINIMAL_STACK_SIZE * 4,
NULL, tskIDLE_PRIORITY + 6, & ReceiveHandler);
  if (retVal != pdPASS) {
    while (1);
  } // check if task creation failed

  retVal = xTaskCreate(updateListTask, "UpdateListTask", configMINIMAL_STACK_SIZE *
4, NULL, tskIDLE_PRIORITY + 6, & UpdateListHandler);
  if (retVal != pdPASS) {
    while (1);
  } // check if task creation failed

  retVal = xTaskCreate(heartbeatTask, "HBTask", configMINIMAL_STACK_SIZE * 4, NULL,
tskIDLE_PRIORITY + 6, & HeartbeatHandler);
  if (retVal != pdPASS) {
    while (1);
  } // check if task creation failed

  retVal = xTaskCreate(inputTask, "InputTask", configMINIMAL_STACK_SIZE * 4, NULL,
tskIDLE_PRIORITY + 7, & InputHandler);
  if (retVal != pdPASS) {
    while (1);
  } // check if task creation failed
```

```
  vTaskStartScheduler();

  while (1) {

  }
}

void SystemClock_Config(void) {
  RCC_OscInitTypeDef RCC_OscInitStruct = {
    0
  };
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {
    0
  };

  /** Configure the main internal regulator output voltage
   */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK) {
    Error_Handler();
  }

  /** Initializes the RCC Oscillators according to the specified parameters
   * in the RCC_OscInitTypeDef structure.
   */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
  RCC_OscInitStruct.MSIState = RCC_MSI_ON;
  RCC_OscInitStruct.MSICalibrationValue = 0;
  RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_10;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig( & RCC_OscInitStruct) != HAL_OK) {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
   */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK |
    RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig( & RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK) {
    Error_Handler();
  }
}
```

```
/*TASK DEFINITIONS*/
void logOnTask(void * argument) {
  // Set broadcast address
  SpiritPktStackSetDestinationAddress(BROADCAST);

  // Send the payload
  xTxDoneFlag = S_RESET;
  uint8_t payload[22] = {
    0
  };
  uint16_t bytesToSend = createPDU(LOG_ON, (uint8_t * ) payload, NULL, 0);
  SPSGRF_StartTx((uint8_t * ) payload, bytesToSend);
  while (!xTxDoneFlag) {};

  SpiritCmdStrobeRx();

  vTaskDelete(NULL);
}

void receiveTask(void * argument) {
  for (;;) {
    xRxDoneFlag = S_RESET;
    SPSGRF_StartRx();
    while (!xRxDoneFlag);

    if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
      uint8_t srcAddress = SpiritPktStackGetReceivedSourceAddress();
      uint8_t destAddress = SpiritPktStackGetReceivedDestAddress();

      char payload[MAXBUF] = {
        0
      };
      uint16_t bytesRecv = SPSGRF_GetRxData((uint8_t * ) payload);
      processPayload(srcAddress, destAddress, payload, bytesRecv);

      xSemaphoreGive(xMutex);
    }

  }
}

void inputTask(void * argument) {
  for (;;) {
    if (xSemaphoreTake(xMutex, 0) == pdTRUE) {
      // check queue for input backlog
      message_t msg_t;
      if (xQueueReceive(xMessageQueue, & msg_t, 0) == pdTRUE) {
```

```
        processInput( & msg_t);
        UART_SendString("$: ");
      }

      xSemaphoreGive(xMutex);
    }

    vTaskDelay(pdMS_TO_TICKS(200)); // to ensure other tasks don't starve
  }
}

void updateListTask(void * argument) {
  for (;;) {
    decrementTimers();
    removeOfflineNodes();
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}

void heartbeatTask(void * argument) {
  for (;;) {
    if (xSemaphoreTake(xMutex, 0) == pdTRUE) {
      SpiritCmdStrobeSabort();

      // Set broadcast address
      SpiritPktStackSetDestinationAddress(BROADCAST);

      // Send the payload
      uint8_t payload[22] = {
        0
      };
      uint16_t bytesToSend = createPDU(HEARTBEAT, (uint8_t * ) payload, NULL, 0);
      xTxDoneFlag = S_RESET;
      SpiritCmdStrobeFlushTxFifo();
      SPSGRF_StartTx(payload, bytesToSend);
      while (!xTxDoneFlag);

      xSemaphoreGive(xMutex);

      SpiritCmdStrobeRx();

      vTaskDelay(pdMS_TO_TICKS(5000));
    }

  }
}
```

```
/*-------------INTERRUPT HANDLERS-----------------------------*/
void USART2_IRQHandler(void) {
  if (USART2 -> ISR & USART_ISR_RXNE) {
    if (!loggedOn) // not logged on, fill username buffer
    {
      if (USART2 -> RDR == '\r') // username completed
      {
        msgIndex = 0; //reset index
        loggedOn = 1; // set flag high to break out of log on loop
      } else if (USART2 -> RDR == 0x7F) // backspace
      {
        USART2 -> TDR = USART2 -> RDR; // echo

        if (msgIndex > 0) {
          msgIndex--;
        }
      } else // still building username
      {
        USART2 -> TDR = USART2 -> RDR; // echo char
        username[msgIndex] = USART2 -> RDR; // add to str
        msgIndex++; // inc index
      }
    } else // logged on, fill message buffer
    {
      BaseType_t xHigherPriorityTaskWoken = pdFALSE;

      if (USART2 -> RDR == '\r') // message completed, send msg to Queue
      {
        UART_SendString("\r\n");
        message_t msg_t;
        memcpy(msg_t.message, message, msgIndex);
        msg_t.message[msgIndex] = '\0';
        msg_t.len = msgIndex;

        xQueueSendFromISR(xMessageQueue, & msg_t, & xHigherPriorityTaskWoken);

        //memset(message, '\0', sizeof(message)); // reset message
        msgIndex = 0; //reset message index

      } else if (USART2 -> RDR == 0x7F) // backspace
      {
        USART2 -> TDR = USART2 -> RDR; // echo
        if (msgIndex > 0) {
          msgIndex--;
        }
      } else {
        USART2 -> TDR = USART2 -> RDR; // copy received char to transmit buffer to
```

```
echo
        message[msgIndex] = USART2 -> RDR; //put char into string
        msgIndex++; // increment current string index
      }

      SpiritCmdStrobeRx();
      portYIELD_FROM_ISR( & xHigherPriorityTaskWoken);
    }
  }
}

/*GIVEN*/
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
  SpiritIrqs xIrqStatus;

  if (GPIO_Pin != SPIRIT1_GPIO3_Pin) {
    return;
  }

  SpiritIrqGetStatus( & xIrqStatus);
  if (xIrqStatus.IRQ_TX_DATA_SENT) {
    xTxDoneFlag = S_SET;
  }
  if (xIrqStatus.IRQ_RX_DATA_READY) {
    xRxDoneFlag = S_SET;
  }
  if (xIrqStatus.IRQ_RX_TIMEOUT) {
    SpiritCmdStrobeRx();
  }
}

/* Configure Timer to interrupt 100 kHz (100 times every Tick) */
void RTOS_Stats_Timer_Init(void) {
  RCC -> APB1ENR1 |= (RCC_APB1ENR1_TIM5EN); // turn on TIM5
  TIM5 -> DIER |= (TIM_DIER_UIE); // enable interrupts
  TIM5 -> SR &= ~(TIM_SR_UIF); // clear interrupt flag
  TIM5 -> ARR = CLK_FREQUENCY / 100000 - 1;
  TIM5 -> CR1 |= TIM_CR1_CEN; // start timer

  // enable interrupts
  NVIC -> ISER[0] = (1 << (TIM5_IRQn & 0x1F));
}

/* Timer 5 is used to collect runtime stats for FreeRTOS tasks*/
void TIM5_IRQHandler(void) {
  TIM5 -> SR &= ~(TIM_SR_UIF);
  ulHighFrequencyTimerTicks++;
```

```
}

/* Built in functions for using FreeRTOS runtime stats need to be defined*/
void configureTimerForRunTimeStats(void) {
  ulHighFrequencyTimerTicks = 0;
  RTOS_Stats_Timer_Init();
}

unsigned long getRunTimeCounterValue(void) {
  return ulHighFrequencyTimerTicks;
}

void Error_Handler(void) {
  __disable_irq();
  while (1) {}
}
```

usart.h:

```
#ifndef INC_USART_H_
#define INC_USART_H_

#define CLK_FREQUENCY 32000000
#define BAUDRATE 861600

extern
const char * clear_screen;
extern
const char * move_cursor_home;
extern
const char * green_text;
extern
const char * red_text;
extern
const char * reset_text;

void USART_Init(void);
void UART_SendString(const char * str);

#endif /* INC_USART_H_ */
```

usart.c:

```
#include "main.h"

#include "usart.h"

#include <stdlib.h>

const char * clear_screen = "\033[2J";
const char * move_cursor_home = "\033[H";
const char * green_text = "\033[32m";
const char * red_text = "\033[31m";
const char * reset_text = "\033[0m";

void USART_Init(void) {
  RCC -> AHB2ENR |= (RCC_AHB2ENR_GPIOAEN);
  GPIOA -> AFR[0] &= ~(GPIO_AFRL_AFSEL2 | GPIO_AFRL_AFSEL3); // mask AF selection
  GPIOA -> AFR[0] |= ((7 << GPIO_AFRL_AFSEL2_Pos) | // select USART2 (AF7)
    (7 << GPIO_AFRL_AFSEL3_Pos)); // for PA2 and PA3

  GPIOA -> MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3); // enable alternate
function
  GPIOA -> MODER |= (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1); // for PA2 and PA3

  // Configure USART2 connected to the debugger virtual COM port
  RCC -> APB1ENR1 |= RCC_APB1ENR1_USART2EN; // enable USART by turning on system
clock
  USART2 -> CR1 &= ~(USART_CR1_M1 | USART_CR1_M0); // set data to 8 bits
  USART2 -> BRR = CLK_FREQUENCY / BAUDRATE; // baudrate for 861600
  USART2 -> CR1 |= USART_CR1_UE; // enable USART
  USART2 -> CR1 |= (USART_CR1_TE | USART_CR1_RE); // enable transmit and receive
for USART

  // enable interrupts for USART2 receive
  USART2 -> CR1 |= USART_CR1_RXNEIE; // enable RXNE interrupt on USART2
  USART2 -> ISR &= ~(USART_ISR_RXNE); // clear interrupt flag while (message[i] !=
0)

  NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4); // set NVIC Priority Grouping
  NVIC_SetPriority(USART2_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 5,
0)); // set interrupt priorities
  NVIC_EnableIRQ(USART2_IRQn);
}

void UART_SendString(const char * str) {
  uint16_t i;
  for (i = 0; str[i] != 0; i++) { // check for terminating NULL character
```

```
    while (!(USART2 -> ISR & USART_ISR_TXE)); // wait for transmit buffer to be
empty
    USART2 -> TDR = str[i]; // transmit character to USART
  }
}
```

utils.h:

```
#ifndef INC_UTILS_H_
#define INC_UTILS_H_

#include "main.h"

#include "usart.h"

#include "spsgrf.h"

#include <string.h>

#include <stdio.h>

#include <ctype.h>

extern volatile SpiritFlagStatus xTxDoneFlag;
extern volatile SpiritFlagStatus xRxDoneFlag;

void getUsername(void);
void printUsage(void);
void processPayload(uint8_t srcAddress, uint8_t destAddress, char * payload,
uint16_t bytesRecv);
void printPayload(uint8_t srcAddress, uint8_t destAddress, char * payload, uint16_t
bytesRecv);
void processInput(message_t * msg_t);
uint16_t createPDU(uint8_t flag, uint8_t * buf, uint8_t * message, uint16_t len);
void parseCommand(const char * input, uint8_t * number, char * message);

#endif /* INC_UTILS_H_ */
```

utils.c:

```
#include "utils.h"

#include "main.h"

#include "node.h"

#include "usart.h"

void printUsage(void) {
  UART_SendString("Usage:\r\n\tClear screen: <clear>\r\n\tSee who's online:
<%l>\r\n\tSend DM: <%m> <address in decimal> <message>\r\n\tSend broadcast: <%b>
<message>\r\n\t");
  UART_SendString(red_text);
  UART_SendString("Broadcasts printed in red\r\n\t");
  UART_SendString(green_text);
  UART_SendString("DM's printed in green\r\n\r\n");
  UART_SendString(reset_text);
}

void getUsername(void) {
  // Prompt for username
  UART_SendString("Welcome to Chat! Please enter a username: ");

  // Block until username is set
  while (!loggedOn);
  UART_SendString(clear_screen);
  UART_SendString(move_cursor_home);
  printUsage();
  UART_SendString("$: ");
}

uint16_t createPDU(uint8_t flag, uint8_t * buf, uint8_t * message, uint16_t len) {
  uint16_t bytesToSend = 0;

  memcpy(buf, & flag, 1);
  bytesToSend++;
  memcpy(buf + bytesToSend, username, strlen(username));
  bytesToSend += strlen(username);
  memcpy(buf + bytesToSend, "\0", 1);
  bytesToSend++;

  if (flag == MESSAGE) {
    memcpy(buf + bytesToSend, message, len);
    bytesToSend += len;
```

```
    memcpy(buf + bytesToSend, "\0", 1);
    bytesToSend++;
  }

  return bytesToSend;
}

void processInput(message_t * msg_t) {
  if (strncasecmp(msg_t -> message, "clear", 5) == 0) {
    UART_SendString(move_cursor_home);
    UART_SendString(clear_screen);
    printUsage();
  } else if (strncasecmp(msg_t -> message, "%L", 2) == 0) {
    printList();
  } else if (strncasecmp(msg_t -> message, "%M", 2) == 0 || strncasecmp(msg_t ->
message, "%B", 2) == 0) {
    uint8_t destAddress = 0;
    char buf[MAXBUF] = {
      0
    };
    if (strncasecmp(msg_t -> message, "%M", 2) == 0) {
      parseCommand(msg_t -> message, & destAddress, buf);
    } else if (strncasecmp(msg_t -> message, "%B", 2) == 0) {
      destAddress = BROADCAST;
      memcpy(buf, msg_t -> message + 3, msg_t -> len);
    }

    // Transmit Message
    uint8_t sendBuf[MAXBUF] = {
      0
    };
    uint16_t bytesToSend = createPDU(MESSAGE, (uint8_t * ) sendBuf, (uint8_t * )
buf, strlen(buf));

    SpiritCmdStrobeSabort();
    SpiritPktStackSetDestinationAddress(destAddress);
    xTxDoneFlag = S_RESET;
    SpiritCmdStrobeFlushTxFifo();
    SPSGRF_StartTx((uint8_t * ) sendBuf, bytesToSend);
    while (!xTxDoneFlag) {};
    SpiritCmdStrobeRx();
  }
}

void processPayload(uint8_t srcAddress, uint8_t destAddress, char * payload,
uint16_t bytesRecv) {
  if (payload[0] == LOG_ON) {
```

```
    // Get username from person logging on, add them to Online List
    char recvUser[21] = {
      '\0'
    };
    uint8_t recvUserLen = strlen(payload + 1);
    memcpy(recvUser, payload + 1, recvUserLen);
    addNode(srcAddress, recvUser, recvUserLen);

    SpiritPktStackSetDestinationAddress(srcAddress);

    // Reply with Announcement Packet
    uint8_t payload[22] = {
      0
    };
    uint16_t bytesToSend = createPDU(RESPONSE, (uint8_t * ) payload, NULL, 0);
    xTxDoneFlag = S_RESET;
    SpiritCmdStrobeFlushTxFifo();
    SPSGRF_StartTx(payload, bytesToSend);
    while (!xTxDoneFlag);

    SpiritCmdStrobeRx();
  } else if (payload[0] == RESPONSE) // process announcement packet from others
  {
    char recvUser[21] = {
      '\0'
    };
    uint8_t recvUserLen = strlen(payload + 1);
    memcpy(recvUser, payload + 1, recvUserLen);
    addNode(srcAddress, recvUser, recvUserLen);
  } else if (payload[0] == HEARTBEAT) {
    if (inList(srcAddress)) // If already online, refresh TTL
    {
      refreshNode(srcAddress);
    } else // If not online, add to online list (in case announcement packet or log
on was missed)
    {
      char recvUser[21] = {
        '\0'
      };
      uint8_t recvUserLen = strlen(payload + 1);
      memcpy(recvUser, payload + 1, recvUserLen);
      addNode(srcAddress, recvUser, recvUserLen);
    }
  } else if (payload[0] == MESSAGE) // print message
  {
    if (inList(srcAddress)) // If already online, refresh TTL
    {
```

```
      refreshNode(srcAddress);
    } else // If not online, add to online list (in case announcement packet or log
on was missed)
    {
      char recvUser[21] = {
        '\0'
      };
      uint8_t recvUserLen = strlen(payload + 1);
      memcpy(recvUser, payload + 1, recvUserLen);
      addNode(srcAddress, recvUser, recvUserLen);
    }

    printPayload(srcAddress, destAddress, payload, bytesRecv);
  }
}

void printPayload(uint8_t srcAddress, uint8_t destAddress, char * payload, uint16_t
bytesRecv) {
  if (inList(srcAddress)) {
    // Get node
    connectionNode * cur = getNode(srcAddress);
    char buf[MAXBUF] = {
      0
    };
    UART_SendString("\r\n");

    if (destAddress == BROADCAST) {
      UART_SendString(red_text);
      snprintf(buf, sizeof(buf), "%s (%u): %s\r\n", cur -> name, srcAddress,
payload + 1 + strlen(cur -> name) + 1);
      UART_SendString(buf);
      UART_SendString(reset_text);
    } else if (destAddress == MY_ADDRESS) {
      UART_SendString(green_text);
      snprintf(buf, sizeof(buf), "%s (%u): %s\r\n", cur -> name, srcAddress,
payload + 1 + strlen(cur -> name) + 1);
      UART_SendString(buf);
      UART_SendString(reset_text);
    }

    UART_SendString("$: ");
  }
}

void parseCommand(const char * input, uint8_t * number, char * message) {
  const char * ptr = input;
```

```
  ptr += 3; // skip past command (assume perfect input)

  // get the address
  uint16_t tempNumber = 0;
  while (isdigit((unsigned char) * ptr)) {
    tempNumber = tempNumber * 10 + ( * ptr - '0');
    ptr++;
  }

  * number = (uint8_t) tempNumber;

  // skip the space after the address
  if ( * ptr == ' ') {
    ptr++;
  }

  // copy the remaining message
  size_t i = 0;
  while ( * ptr != '\0') {
    message[i++] = * ptr++;
  }
  message[i] = '\0'; // null-terminate the message
}
```

node.h:

```
#ifndef INC_NODE_H_
#define INC_NODE_H_

#include "main.h"

#define TIME_TO_LIVE 110

typedef struct {
  char name[21];
  uint8_t address;
  uint8_t timer;
  uint8_t online;
  struct connectionNode * next;
}
connectionNode;

void addNode(uint8_t address, char * name, uint8_t len);
connectionNode * getNode(uint8_t address);
void removeOfflineNodes(void);
void refreshNode(uint8_t address);
void decrementTimers(void);
int inList(uint8_t address);
void printList(void);
void freeList(void);

#endif /* INC_NODE_H_ */
```

node.c:

```
#include "main.h"

#include "node.h"

#include <string.h>

#include <stdio.h>

#include "FreeRTOS.h"

#include "usart.h"

static connectionNode * headNode = NULL;

void addNode(uint8_t address, char * name, uint8_t len) {
  if (inList(address)) {
    return;
  }

  connectionNode * newNode = (connectionNode * )
pvPortMalloc(sizeof(connectionNode));
  memcpy(newNode -> name, name, len + 1);
  newNode -> address = address;
  newNode -> timer = TIME_TO_LIVE;
  newNode -> online = 1;
  newNode -> next = NULL;

  connectionNode ** cur = & headNode;
  if ( * cur == NULL) {
    * cur = newNode;
  } else {
    while ( * cur != NULL) {
      cur = (connectionNode ** ) & (( * cur) -> next);
    }

    * cur = newNode;
  }
}

connectionNode * getNode(uint8_t address) {
  connectionNode * retVal = NULL;

  connectionNode ** cur = & headNode;
  while ( * cur != NULL) {
    if (( * cur) -> address == address) {
```

```
      retVal = * cur;
      break;
    }

    cur = (connectionNode ** ) & (( * cur) -> next);
  }

  return retVal;
}

void removeOfflineNodes(void) {
  connectionNode ** cur = & headNode;
  connectionNode * temp;

  while ( * cur != NULL) {
    if (( * cur) -> timer == 0) {
      temp = * cur;
      * cur = (connectionNode * )( * cur) -> next; // bypass the node
      vPortFree(temp);
    } else {
      cur = (connectionNode ** ) & (( * cur) -> next);
    }
  }
}

void refreshNode(uint8_t address) {
  connectionNode * cur = headNode;
  while (cur != NULL) {
    if (cur -> address == address && cur -> online == 1) {
      cur -> timer = TIME_TO_LIVE; // reset timer to initial value
      break;
    }
    cur = (connectionNode * ) cur -> next;
  }
}

void decrementTimers(void) {
  connectionNode * cur = headNode;
  while (cur != NULL) {
    cur -> timer -= 1;
    cur = (connectionNode * ) cur -> next;
  }
}

int inList(uint8_t address) {
  int found = 0;
  connectionNode * cur = headNode;
```

```
  while (cur != NULL) {
    if (cur -> address == address && cur -> online == 1) {
      found = 1;
      break;
    }

    cur = (connectionNode * ) cur -> next;
  }
  return found;
}

void printList(void) {
  char userBuf[50] = {
    0
  };
  snprintf(userBuf, sizeof(userBuf), "\tYOUR USERNAME: %s\r\n\r\n", username);
  UART_SendString(userBuf);

  connectionNode * cur = headNode;
  while (cur != NULL) {
    char buffer[64] = {
      0
    };
    snprintf(buffer, sizeof(buffer), "\tName: %s, Address: %u, TTL: %d\r\n", cur ->
name, cur -> address, cur -> timer);
    UART_SendString((const char * ) buffer);
    cur = (connectionNode * ) cur -> next;
  }
}

void freeList(void) {
  connectionNode * cur = headNode;
  connectionNode * prev = NULL;
  while (cur != NULL) {
    prev = cur;
    cur = (connectionNode * ) cur -> next;
    vPortFree(prev);
  }

  headNode = NULL;
}
```