Concurrency in Bike System

About the Project

- Consists of 3 main components

About the Project

- Consists of 3 main components
- One strongly relies on concurrency

About the Project

- Consists of 3 main components
- One strongly relies on concurrency
- Another uses it to work faster

The Bike Computer

- Computes various real time metrics

The Bike Computer

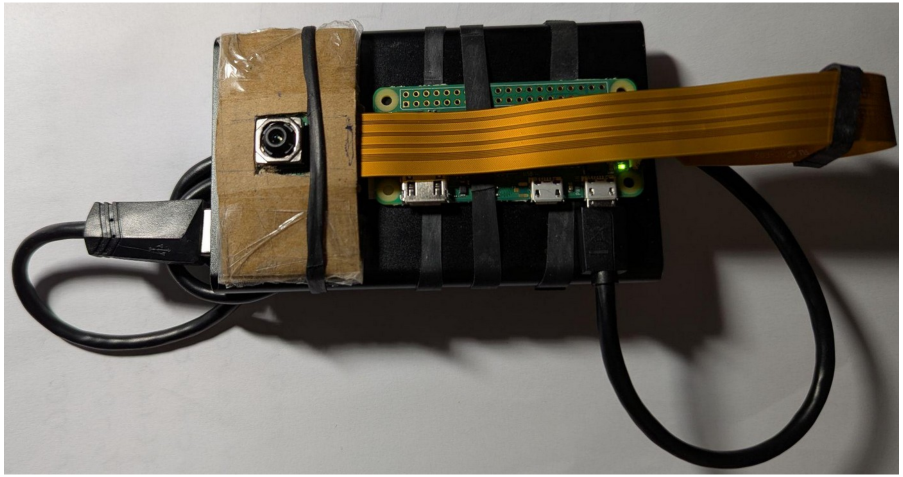
- Computes various real time metrics
- Displays them on the E-paper display

The Bike Computer

- Computes various real time metrics
- Displays them on the E-paper display
- Stores every data point on a micro SD

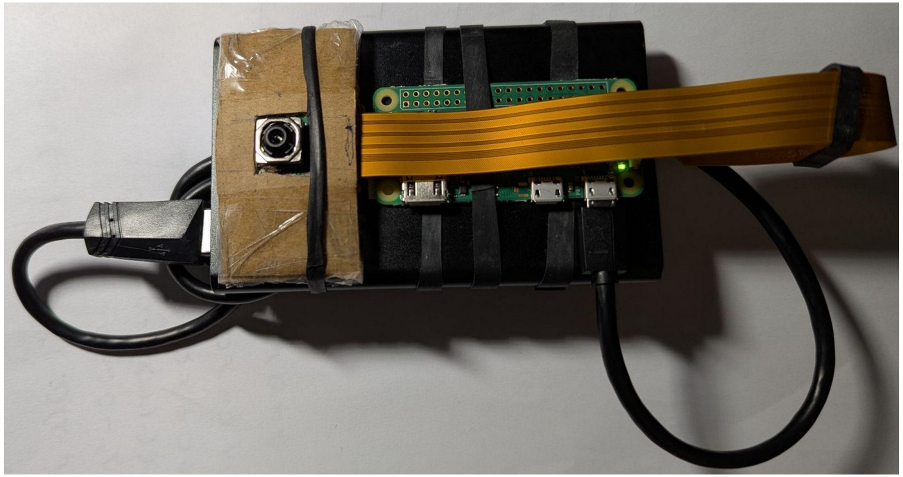
The Bike Camera

- A separate module mounted in front of the bike computer



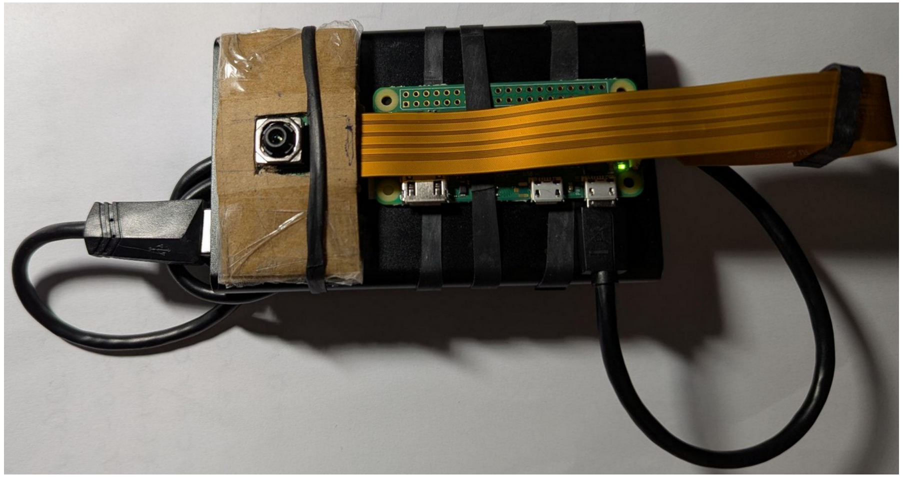
The Bike Camera

- A separate module mounted in front of the bike computer
- Takes photos and scans for street signs



The Bike Camera

- A separate module mounted in front of the bike computer
- Takes photos and scans for street signs
- Sends data about the detected signs to the bike computer to log

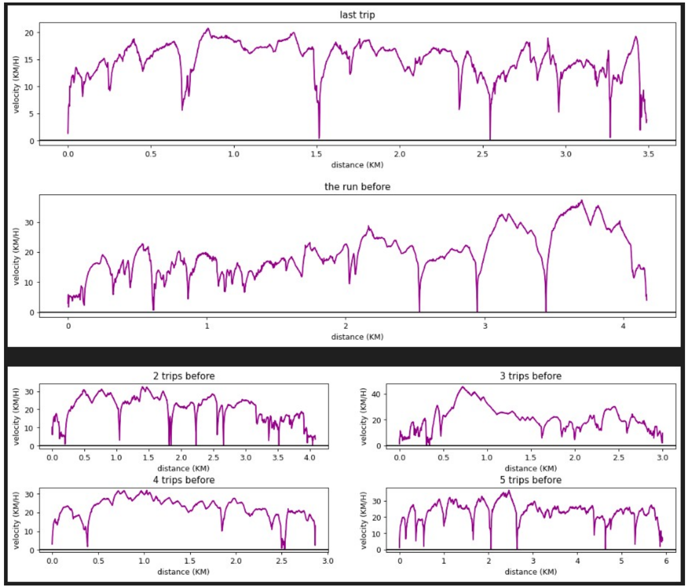


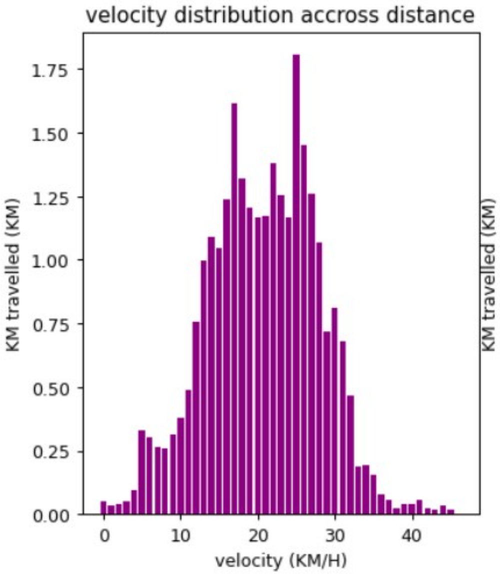
The Data Analysis Part

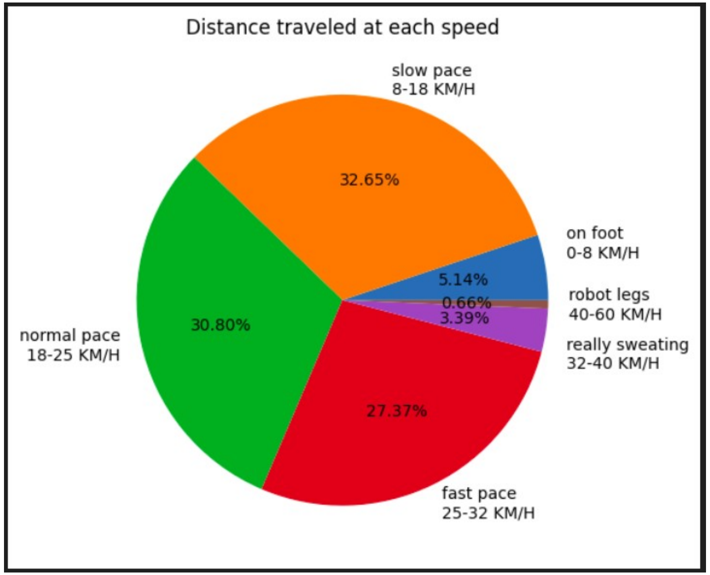
- A python script looks at the logged data and displays various metrics

The Data Analysis Part

- A python script looks at the logged data and displays various metrics







Concurrency in Bike Calculator

- Would not work without it

Concurrency in Bike Calculator

- Would not work without it
- 3 main tasks:
  - One to read the data from the sensors and buttons

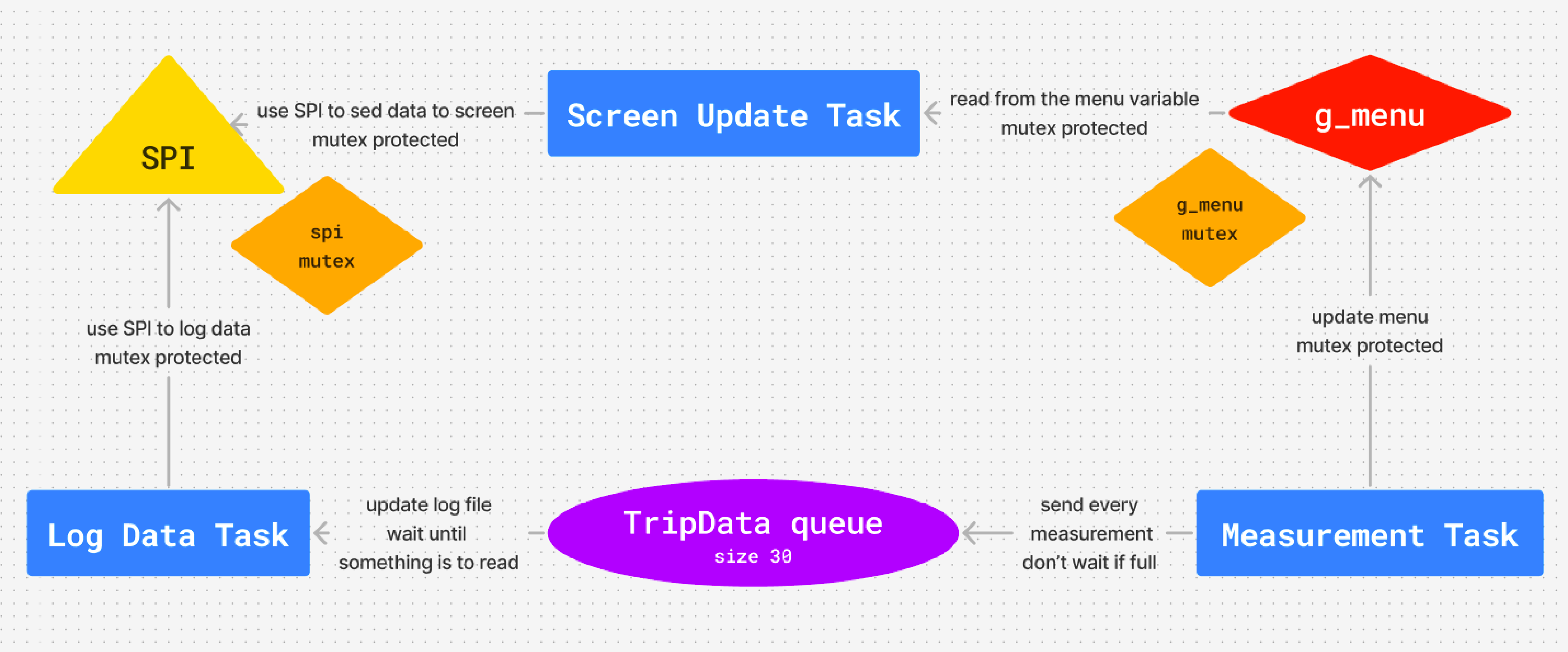
Concurrency in Bike Calculator

- Would not work without it
- 3 main tasks:
  - One to read the data from the sensors and buttons
  - One to log the data to the micro SD

Concurrency in Bike Calculator

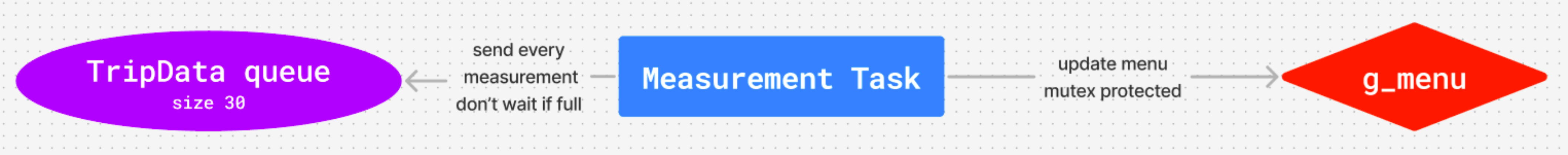
- Would not work without it
- 3 main tasks:
  - One to read the data from the sensors and buttons
  - One to log the data to the micro SD
  - One to display user information to the E-paper display

Process Diagram



The Measurement Task

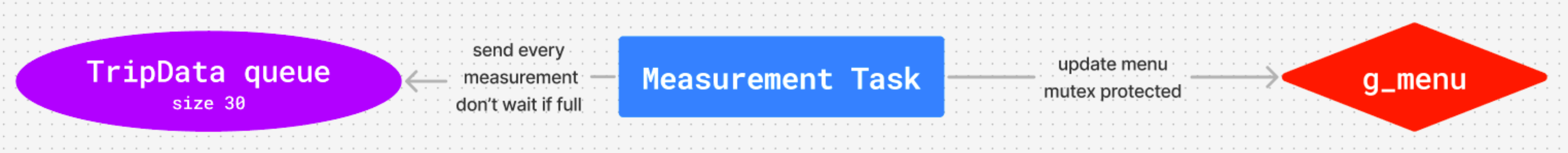
- Recieves sensor data and updates the global menu variable to display most recent state



The Measurement Task

- Recieves sensor data and updates the global menu variable to display most recent state
- This operation is protected by a mutex:

```
if (xSemaphoreTake(g_menuMutex, portMAX_DELAY))
{
    g_menu = menu;
    xSemaphoreGive(g_menuMutex);
}
```



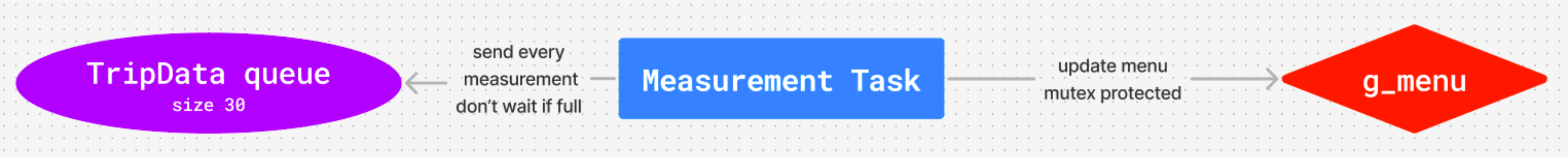
The Measurement Task

- Recieves sensor data and updates the global menu variable to display most recent state
- This operation is protected by a mutex
- portMAX_DELAY tells the mutex how much to wait for the resource if busy

```
if (xSemaphoreTake(g_menuMutex, portMAX_DELAY))
{
    g_menu = menu;
    xSemaphoreGive(g_menuMutex);
}
```

The Measurement Task

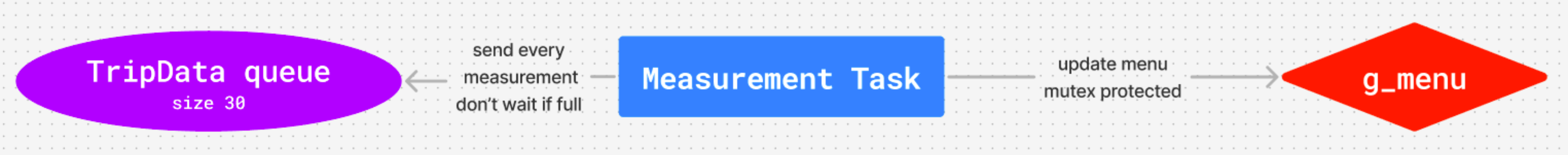
- Also sends an object of type TripData to a message queue for logging



The Measurement Task

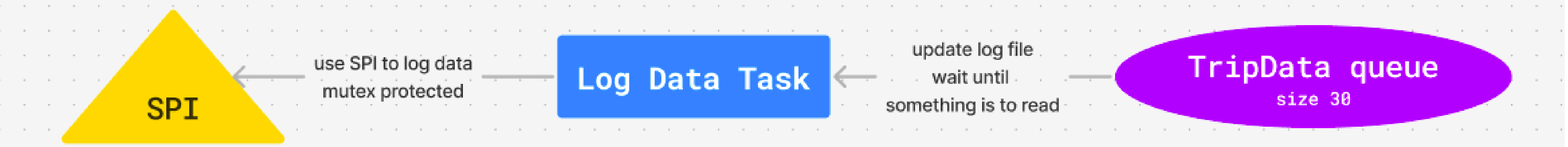
- Also sends an object of type TripData to a message queue for logging
- The function takes as arguments the queue, a refference to the object to send and the time to wait in case the queue is full

```
xQueueSend(g_tripDataQueue, &tripData, SEND_DATA_DELAY_TICKS);
```



The Logging Task

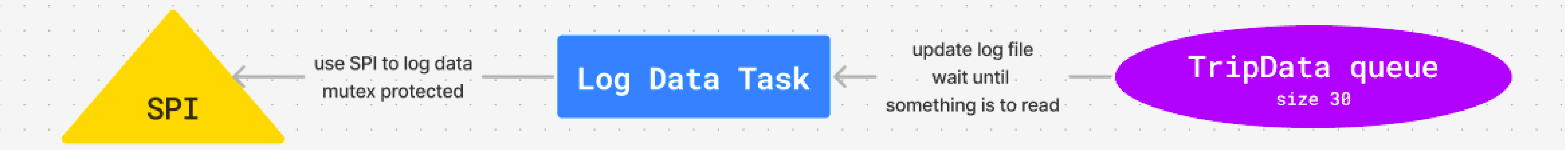
- Recieves data from the TripData waiting queue



The Logging Task

- Recieves data from the TripData waiting queue
- If the queue is empty restarts the loop (busy waiting)

```
xQueueReceive(g_tripDataQueue, &dataToWrite, SEND_DATA_DELAY_TICKS);
```



The Logging Task

- The SPI (serial periphelar interface) is a short distance communication protocol

The Logging Task

- The SPI (serial periphelar interface) is a short distance communication protocol
- Works on a Master-Slave model, where the ESP32 is the main device orchestrating communication with the E-paper display and, in this case, the micro SD data writing by driving the clock and chip select signals

The Logging Task

- A semaphore is used to synchronise SPI use between the logging and the display task

```
if (xSemaphoreTake(g_spiMutex, portMAX_DELAY))
{
```

The Logging Task

- A semaphore is used to synchronise SPI use between the logging and the display task
- If the SPI is busy, wait as long as possible

```
if (xSemaphoreTake(g_spiMutex, portMAX_DELAY))
{
```

The Logging Task

- A semaphore is used to synchronise SPI use between the logging and the display task
- If the SPI is busy, wait as long as possible
- This is done so no recording is left un-logged

```
if (xSemaphoreTake(g_spiMutex, portMAX_DELAY))
{
```

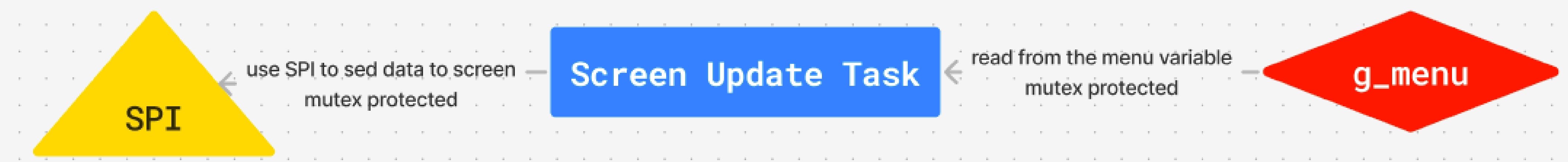
The Logging Task

- An interesting aspect of this task
  - If no micro SD card is inserted, it kills itself in order to let the Screen Update Task take full advantage of the SPI

```
else
{
    // no point to this task if it cannot write to the file system
    vTaskDelete(NULL);
    return;
}
```

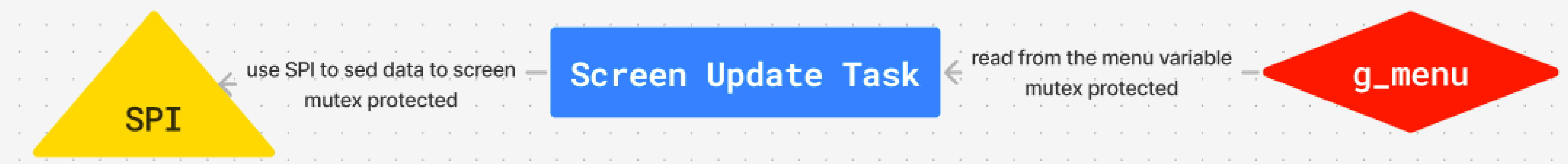
The Screen Update Task

- Reads the g_menu value which is protected by a mutex



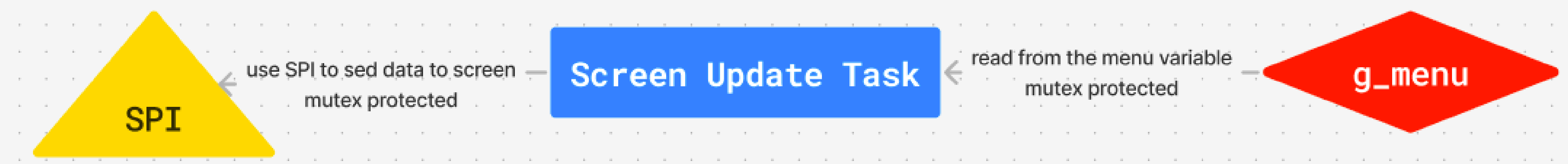
The Screen Update Task

- Reads the g_menu value which is protected by a mutex
- Uses the SPI protocol to update what is shown to the user



The Screen Update Task

- Reads the g_menu value which is protected by a mutex
- Uses the SPI protocol to update what is shown to the user
- This takes between 0.2 and one second depending on the type of refresh



The main loop task

- Its purpose is:
  - Create the other tasks

```
g_tripDataQueue = xQueueCreate(QUEUE_SIZE, sizeof(TripData));
g_spiMutex = xSemaphoreCreateMutex();
g_menuMutex = xSemaphoreCreateMutex();

TaskHandle_t writeToFsTask = NULL;
xTaskCreate(writeToFileTask, "writeToFsTask", FILE_WRITING_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &writeToFsTask);

TaskHandle_t displayTaskHandle = NULL;
xTaskCreate(displayManagement, "display", DEFAULT_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &displayTaskHandle);

TaskHandle_t measurementTaskHandle = NULL;
xTaskCreate(measurementTask, "measurement", MEASUREMENT_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &measurementTaskHandle);
```

The main loop task

- Its purpose is:
  - Create the other tasks
  - Create the waiting queues and semaphores

```
g_tripDataQueue = xQueueCreate(QUEUE_SIZE, sizeof(TripData));
g_spiMutex = xSemaphoreCreateMutex();
g_menuMutex = xSemaphoreCreateMutex();

TaskHandle_t writeToFsTask = NULL;
xTaskCreate(writeToFileTask, "writeToFsTask", FILE_WRITING_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &writeToFsTask);

TaskHandle_t displayTaskHandle = NULL;
xTaskCreate(displayManagement, "display", DEFAULT_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &displayTaskHandle);

TaskHandle_t measurementTaskHandle = NULL;
xTaskCreate(measurementTask, "measurement", MEASUREMENT_TASK_STACK_SIZE, NULL, DEFAULT_TASK_PRIORITY, &measurementTaskHandle);
```

The main loop task

- Its purpose is:
  - Create the other tasks
  - Create the waiting queues and semaphores
- After that is kills itself:

```
void loop()
{
    vTaskDelete(NULL);
};
```

ESP-IDF

- Esspressif IoT Development Framework

ESP-IDF

- Esspressif IoT Development Framework
- Based on FreeRTOS (real time operating system)

ESP-IDF

- Esspressif IoT Development Framework
- Based on FreeRTOS (real time operating system)
- Supports:
  - Tasks and Co-routines

ESP-IDF

- Esspressif IoT Development Framework
- Based on FreeRTOS (real time operating system)
- Supports:
  - Tasks and Co-routines
  - Queues, mutexes and semaphores

ESP-IDF

- Esspressif IoT Development Framework
- Based on FreeRTOS (real time operating system)
- Supports:
  - Tasks and Co-routines
  - Queues, mutexes and semaphores
  - Task notifications

Tasks and Co-routines

- A task is similar to a regular process:
  - Has its own stack

Tasks and Co-routines

- A task is similar to a regular process:
  - Has its own stack
  - Is not aware of context switching

Tasks and Co-routines

- A task is similar to a regular process:
  - Has its own stack
  - Is not aware of context switching
  - Supports prioritisation

Tasks and Co-routines

- A task is similar to a regular process:
  - Has its own stack
  - Is not aware of context switching
  - Supports prioritisation
  - Supports full preemption

Tasks and Co-routines

- Co-routines are not used as often as tasks

Tasks and Co-routines

- Co-routines are not used as often as tasks
- Exist only to serve very limited devices

Tasks and Co-routines

- Co-routines are not used as often as tasks
- Exist only to serve very limited devices
- Are similar to regular threads in nature:
  - Share a stack with all other co-routines

Tasks and Co-routines

- Co-routines are not used as often as tasks
- Exist only to serve very limited devices
- Are similar to regular threads in nature:
  - Share a stack with all other co-routines
  - Use prioritised cooperative scheduling (need to yeld to let other co-routines run)

Tasks and Co-routines

- Co-routines are not used as often as tasks
- Exist only to serve very limited devices
- Are similar to regular threads in nature:
  - Share a stack with all other co-routines
  - Use prioritised cooperative scheduling (need to yeld to let other co-routines run)
  - Work in an application with other preemptive tasks

Queues, mutexes and semaphores

- Work as expected:
  - Queues are the primary form of inter-task communication and are used as thread safe FIFO buffers

Queues, mutexes and semaphores

- Work as expected:
  - Queues are the primary form of inter-task communication and are used as thread safe FIFO buffers
  - Binary semaphores and mutexes are used for mutual exclusion and syncronisation purposes

Queues, mutexes and semaphores

- Work as expected:
  - Queues are the primary form of inter-task communication and are used as thread safe FIFO buffers
  - Binary semaphores and mutexes are used for mutual exclusion and syncronisation purposes
  - A task notification is an event sent directly to a task, rather than indirectly to a task via an intermediary object such as a queue

Queues, mutexes and semaphores

- Work as expected:
  - Queues are the primary form of inter-task communication and are used as thread safe FIFO buffers
  - Binary semaphores and mutexes are used for mutual exclusion and syncronisation purposes
  - A task notification is an event sent directly to a task, rather than indirectly to a task via an intermediary object such as a queue
  - It can be 45% faster for unblocking a task than an intermediary object such as a mutex for example

Concurrency in Bike Camera

- A regular C++ executable running on Linux
- Uses std::thread to parallelize finding signs rather than trying to find each sign in sequence

```
std::thread bright_red_gw_thread(detect_gw_thread, &p_img, &bright_red_mask, &white_mask, &detection_number, &mtx);
std::thread dark_red_gw_thread(detect_gw_thread, &p_img, &dark_red_mask, &white_mask, &detection_number, &mtx);
bright_red_gw_thread.join();
dark_red_gw_thread.join();
```

Concurrency in Bike Camera

```
void detect_gw_thread(cv::Mat *p_img, cv::Mat *p_red_mask, cv::Mat *p_white_mask, int32_t *p_detection_number, std::mutex *p_mutex)
{
    cv::Mat labels;
    cv::Mat stats;
    cv::Mat centroids;
    cv::connectedComponentsWithStats(*p_red_mask, labels, stats, centroids);

    // look for dark red chunks
    for(int i=1; i < stats.rows; i++)
    {
        float detection_res = find_gw_in_chunk(*p_img, *p_white_mask, labels, stats, i);
        if(detection_res > 0)
        {
            p_mutex->lock();
            (*p_detection_number) ++;
            std::cout << "detection number in thread:" << *p_detection_number << std::endl;
            p_mutex->unlock();
        }
    }
}
```

Performance improvement

Sequential performance

```
took control pic...
on avg the loop takes 175.96ms
took control pic...
took control pic...
on avg the loop takes 209.622ms
took control pic...
took control pic...
on avg the loop takes 193.943ms
took control pic...
took control pic...
on avg the loop takes 186.191ms
detection number in thread:1
took control pic...
took control pic...
on avg the loop takes 194.092ms
^C
```

Performance improvement

Sequential performance

```
took control pic...
on avg the loop takes 175.96ms
took control pic...
took control pic...
on avg the loop takes 209.622ms
took control pic...
took control pic...
on avg the loop takes 193.943ms
took control pic...
took control pic...
on avg the loop takes 186.191ms
detection number in thread:1
took control pic...
took control pic...
on avg the loop takes 194.092ms
^C
```

Concurrent performance

```
on avg the loop takes 208.631ms
took control pic...
on avg the loop takes 192.882ms
took control pic...
took control pic...
on avg the loop takes 192.653ms
took control pic...
detection number in thread:1
detection number in thread:1
took control pic...
on avg the loop takes 190.049ms
took control pic...
took control pic...
on avg the loop takes 195.06ms
took control pic...
^C
```

Things left to improve

- Use both processors on ESP32

Things left to improve

- Use both processors on ESP32
- Try to use notifications rather than mutexes

Things left to improve

- Use both processors on ESP32
- Try to use notifications rather than mutexes
- Eliminate the busy waiting in Logging Task

Things left to improve

- Use both processors on ESP32
- Try to use notifications rather than mutexes
- Eliminate the busy waiting in Logging Task
- Only start the std::thread on the Raspberry pi once then only send the cv::Mat objects to them every time