Multithreading-Based Electronic Stability Program (ESP) Prototype with Memory Protection Using FreeRTOS-MPU Port

Prototype Function:

ESP Prototype - simplified implementation of vehicle stability control.

MPU memory protection ensures this cannot happen.

1. **YAW RATE SENSOR** (MPU6050) to detect the **rate of rotation of the vehicle** and stores it in a SHARED RESOURCE - global variable (yaw_dps).
2. **CONTROL UNIT**

   a. Reads this SHARED RESOURCE and compares it with the **steer input** (in real world scenarios this quantity is calculated using a steering angle sensor - we have taken a simulated CONSTANT value for simplicity).

   b. An ALERT is issued (planning on improvising this by using different LEDS for the wheels which will be braked to show the corrective action)-

   - If **understeering** is detected - yaw is significantly less - a **YELLOW GPIO LED** is turned on.
   - If the yaw rate matches the steer input within a valid threshold - **GREEN onboard LED** turned on.
   - (This unit will be improvised by replacing a POTENTIOMETER connected to the ADC Peripheral of the MCU to simulate the steer input)

A logger task is simply used to read the SHARED RESOURCE - it doesn't take an corrective or alert-issuing action simply reads the value and used for display purposes.

MPU NEED: If an unprivileged or malicious task could write yaw_dps = 0 **ESP activation might be suppressed** resulting in failure of system.

WORK PHASES

We went about the development of our ESP prototype in 4 **phases.**

Phase 1 - **Bare metal program for the aforementioned units in STM32F446RE MCU**

a. STM32F446RE -chosen because of presence of MPU for later use
b. We used **HAL STM32 Library functions** and implemented the functionality of the 3 units - created source file **task1.c** - defined 4 functions:
   i. mpu_int() [innitialises the MPU6050 sensor]

ii. mpuread() -[continuously read real time yaw rate values - gyro values from the 16 bit GYRO_Z Register and store it in the shared resource GLOBAL varibale yaw_dps,
iii. controlunit() -[To read the shared resouce yaw_dps and compare and GPIO control]
iv. loggertask() - to simply read the global variable.

Phase 2 - **FreeRTOS integration**

a. With the bare metal program done- we moved to the **FreeRTOS KERNEL INTEGRATION** for multithreading application.
b. TASK 1 - One-time task - mpu_int()- immediately deleted after configuration using TASK DELETION API - vTaskDelay().
c. TASK2 - mpuread() - real time sensor data acquisition.
d. TASK3 - controlunit()
e. TASK4 - loggertask()

GitHub:

GitHub - shreyaavinod/PHASE-1-and-2-FreeRTOS-Multithreading-Electronic-Stability-Program-Pr...

main.c:

PHASE-1-and-2-FreeRTOS-Multithreading-Electronic-Stability-Program-Prototype/Core/Src/main....

- In Phase 2, we integrated FreeRTOS with the existing bare-metal code.
- The core functions that were previously called inside the while(1) loop—such as reading gyro values, running the control unit, and logging data—are now implemented as **separate FreeRTOS tasks**.
- Each function runs independently as a FreeRTOS task.
- The init_mpu function is also implemented as a task that is created at startup to initialize the MPU and gyroscope. Once initialization is complete, this **task deletes itself** to prevent re-initialization.

task1.c:

PHASE-1-and-2-FreeRTOS-Multithreading-Electronic-Stability-Program-Prototype/Core/Src/task1....

- This file implements a bare-metal program that works directly with the hardware without using an operating system. At the start, the mpu_int() function is called once to initialize the MPU and configure the gyroscope for operation.
- After initialization, the program enters an infinite loop (while(1)) where it continuously performs three main tasks: it **reads sensor data from the gyroscope**, processes this data through the **control unit** to compute necessary actions, and **logs the data** using the logger unit for monitoring or analysis. Each of these functions—reading, control, and logging—is called in every iteration of the loop to ensure real-time operation.

```
printf("Initialisation done.");
xTaskCreate(initfunc, "mpuinit", 200, NULL, 2, NULL);
//task 1 priivileged

xTaskCreate(taskfunc1,"mpuread",200, NULL, 2, &taskhandle1 );
// task 2 control unit privileged

xTaskCreate(taskfunc2, "control unit", 200, NULL, 2, &taskhandle2);

//task 3 logger task
xTaskCreate(taskfunc3,"logger",200,NULL,2,&taskhandle3);
vTaskStartScheduler();
```

- main.c task creation and starting of scheduler.

Phase 3 - **Bare metal Memory Protection Unit Configuration and implementation**

a. We then explored the **configuration of memory regions** and their protection using the **Memory Protection Unit feature of ARM CORTEX M4 Processor** in our STM32F446RE board.
b. We explored the **privileged mode and unprivileged mode** of code execution
c. Configured 3 memory regions :
   i. REGION 1 - SRAM (total 128KB) - starting address 0x20000000 upto 8KB - with access permission - MPU_REGION_PRIV_RW_URO.
   ii. REGION 2 - FLASH (total 512KB) - starting address 0x08000000 upto 512KB - with access permission- MPU_REGION_PRIV_RO_URO
   iii. REGION 3 - STACK REGION (starting 0x20010000 till end of SRAM) - with MPU_REGION_FULL_ACCESS
d. After successful configuration and enabling of MPU registers, we performed read and write operations using PRIVILEGED CODE , then changed the mode to UNPRIVILEGED and performed successful read operation in region 1. (VERIFIED FUNCTIONALITY)
e. We also performed an illegal write operation from unprivileged code - ending up in the **MemManage_Handler().**

GitHub:

🌐 GitHub - shreyaavinod/PHASE-3-Memory-Protection-Unit-Support-Electronic-Stability-P...

main.c

🌐 PHASE-3-Memory-Protection-Unit-Support-Electronic-Stability-Program-Prototype/Core...

In this main.c file we configure 3 MPU regions using HAL functions and create 2 functions to change mode from Privileged to unprivileged and vice versa.

```c
void Enter_Unprivileged_Mode(void)
{
    __set_CONTROL(__get_CONTROL() | 0x1);  // Bit 0 = 1 → unprivileged
    __ISB();  // Instruction sync barrier
}
void Enter_Privileged_Mode(void)
{
        __asm("SVC #0");
}
void MPU_RegionConfig(void)
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    HAL_MPU_Disable();

    // Region 0: SRAM (global)
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;
    MPU_InitStruct.BaseAddress =0x20000000;
    MPU_InitStruct.Size = MPU_REGION_SIZE_8KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_PRIV_RW_URO;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;
    HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

```c
    // Region 1: Flash (code)
    MPU_InitStruct.Number = MPU_REGION_NUMBER1;
    MPU_InitStruct.BaseAddress = 0x08000000;
    MPU_InitStruct.Size = MPU_REGION_SIZE_1MB;
    MPU_InitStruct.AccessPermission = MPU_REGION_PRIV_RO_URO;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE; // code executes from here
    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    // Region 2: Peripheral region
    MPU_InitStruct.Number = MPU_REGION_NUMBER2;
    MPU_InitStruct.BaseAddress = 0x20010000;
    MPU_InitStruct.Size = MPU_REGION_SIZE_64KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;
    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}
```

Thes eMPU regions prevent illegal access to shared resource yaw_dps.

PHASE 4- **FreeRTOS +MPU SUPPORT**

a. The MPU SUPPORT offered by FreeRTOS was different compared to the baremetal MPU integration.
b. Here the individual tasks could be assigned **Privileged and Unprivileged status**.
c. A total of 8 MPU regions can be configured out of which 4 are configured by the Kernel itself which allowed us to allocate **3 regions for each task**.
d. We again allocated the region containing the global variable as a **privileged read write region** for the mpuread() task - privileged task, a **privileged read only** region for the controlunit() task - privileged task and a **read only region** for the loggertask() - unprivileged task

Github link:
GitHub - shreyaavinod/PHASE-4-FreeRTOS-MPU-Based-Task-Isolation-in-ESP-Prototype: In pHase...

mpu_demo.c:
PHASE-4-FreeRTOS-MPU-Based-Task-Isolation-in-ESP-Prototype/Core/Src/mpu_demo.c at master...

- In Phase 4 the FreeRTOS- MPU (Memory Protection Unit) port was used to build the application. Task specific memory regions were configured by specifying attributes and privilege level of each task.

The below image shows the TaskParameters_t structure for the logger task. In this configuration, specific memory regions are defined for the task, along with their access permissions and whether the task runs in privileged or unprivileged mode. These regions control how the task accesses memory. The task is created using the xTaskCreateRestricted API.

```
TaskParameters_t loggerTaskParameters =
{
        .pvTaskCode             = loggertask,
        .pcName                 = "RO",
        .usStackDepth   = configMINIMAL_STACK_SIZE,
        .pvParameters   = NULL,
        .uxPriority             = tskIDLE_PRIORITY,
        .puxStackBuffer = loggerTaskStack,
        .xRegions               =       {
                                                { yaw_dps,      SHARED_MEMORY_SIZE,     portMPU_REGION_READ_ONLY| portMPU_REGION_EXECUTE_NEVER},
                                                { 0,                            0,                                      0
                                                { 0,                            0,                                      0
                                        }
};


        /* Create an unprivileged task with RW access to ucSharedMemory. */
    xTaskCreateRestricted( &( mpuinitTaskParameters ), NULL );
        xTaskCreateRestricted( &( mpureadTaskParameters ), NULL );
        xTaskCreateRestricted( &( mpucontrolTaskParameters ), NULL );
        xTaskCreateRestricted( &( loggerTaskParameters ), NULL );
}
```

NOTES and REFERENCES :

🌐 MPU_Chapter.pdf

🌐 mpustm32.pdf

🌐 Building_on_FreeRTOS_Safety_Critical_Applications.pdf

ELECTRONIC STABILITY PROGRAM PROTOTYPE - MPU SUPPORT IMPLEMENTATION

BASICS - NOTES:

1. Helps prevent accidents in critical scenarios by combining:
   a. **Anti-lock Braking** - Brake is applied in a modulated manner preventing locking of wheels - allows driver to STEER while applying BRAKE
   b. **Traction control**
   c. Counteraction of skidding movements USING CONTROL UNIT
2. ESP COMPONENTS:
   a. Wheel Speed Sensor on each wheel- detects rotational speed of the wheel
   b. YAW RATE SENSOR - measures ROTATION rel. to vertical access.
   c. STEERING ANGLE SENSOR - registers the steering input
   d. CONTROL UNIT - sensor fusion and control
   e. HYDRAULIC UNIT - braking pressure control.

[SCENARIO: An obstacle is encountered - steering angle sensor registers the input (sharp turn towards either side of obstacle)- YAW RATE SENSOR detects vehicle is understeering (moving straight to the vehicle)

RESPONSE OF CONTROL UNIT: Within ms, the rear left wheel is braked allowing the turn]