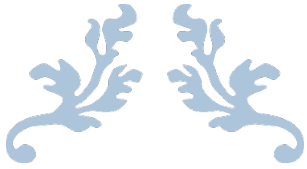


IMPLEMENTATION REPORT

Embedded System

YI JIE LIM 20104720

1. Table of Contents

IMPLEMENTATION REPORT

2. TASK IMPLEMENTATION

The task created for this report is a new FreeRTOS task which polls an ADC (Analog to Digital Converter) value (from a potentiometer), at a low frequency and takes actions depending on the value from the ADC. This task is written to run within a Body Control Module (BCM) containing the RTOS for controlling parallel tasks. The task also presents a visual indication of the ADC value via interaction with physical hardware components (e.g. LEDs); the task can also be used to modify parameters like speedometers or instrument lights on a rig, if needed.

This task serves a purpose to show the integration of a new and low priority FreeRTOS task in the system to poll hardware and output the result in the form of visible feedback (LEDs).

3. DESIGN AND CODE

The task is designed as follows:

- Initialization: First, the initial device setup is performed, ready ADC and LEDs. It can be done before calling the scheduler from main() function or inside the task itself.
- Polling: It constantly polls the ADC to get the value of its value. Condition Checking: The pre defined thresholds are compared with the ADC value.
- Output Control: Based on the ADC value: An LED is turned on if ADC value is over a specific threshold. The LED is turned off if ADC value is below the threshold.
- Task Sleep: Now once the appropriate actions have been taken call osDelay() to go into a low power state while other tasks can run and then check the ADC value again.

```
* AdcVoltage.c
 *
 *  Created on: Nov 29, 2024
 *      Author: YiJie
 */

#include "AdcVoltage.h"
#include "DAVE.h"

void runAdcVoltage (void *arg){
```

```
    static uint8_t adc_led_value;

    int32_t adc_value;
    uint8_t can_data_arr[8] = {0};

    while(1)
    {
      // Read ADC value (assuming a simple single channel measurement)
      adc_value =
ADC_MEASUREMENT_GetResult(&ADC_MEASUREMENT_Channel_A);  // Replace with
your ADC channel
      adc_led_value = (adc_value >> 8) & 0x03;  // Map ADC value to LED control
(example)

      // Control LEDs based on ADC value
      if(adc_led_value == 0)
      {
        DIGITAL_IO_SetOutputHigh(&LED_0);
        DIGITAL_IO_SetOutputLow(&LED_1);
        DIGITAL_IO_SetOutputLow(&LED_2);
      }
      else if(adc_led_value == 1)
      {
        DIGITAL_IO_SetOutputLow(&LED_0);
        DIGITAL_IO_SetOutputHigh(&LED_1);
        DIGITAL_IO_SetOutputLow(&LED_2);
      }
      else if(adc_led_value == 2)
      {
        DIGITAL_IO_SetOutputLow(&LED_0);
        DIGITAL_IO_SetOutputLow(&LED_1);
        DIGITAL_IO_SetOutputHigh(&LED_2);
      }

      // Example CAN message update based on ADC value (optional)
```

```
            can_data_arr[6] = adc_value & 0xFF;  // Store lower byte of ADC value for CAN
transmission
            CAN_NODE_MO_UpdateData(&BcmNode_LMO_01_Config, can_data_arr);
            CAN_NODE_MO_Transmit(&BcmNode_LMO_01_Config);

            osDelay(200);  // Delay for the task (adjust polling frequency)
        }
}
```

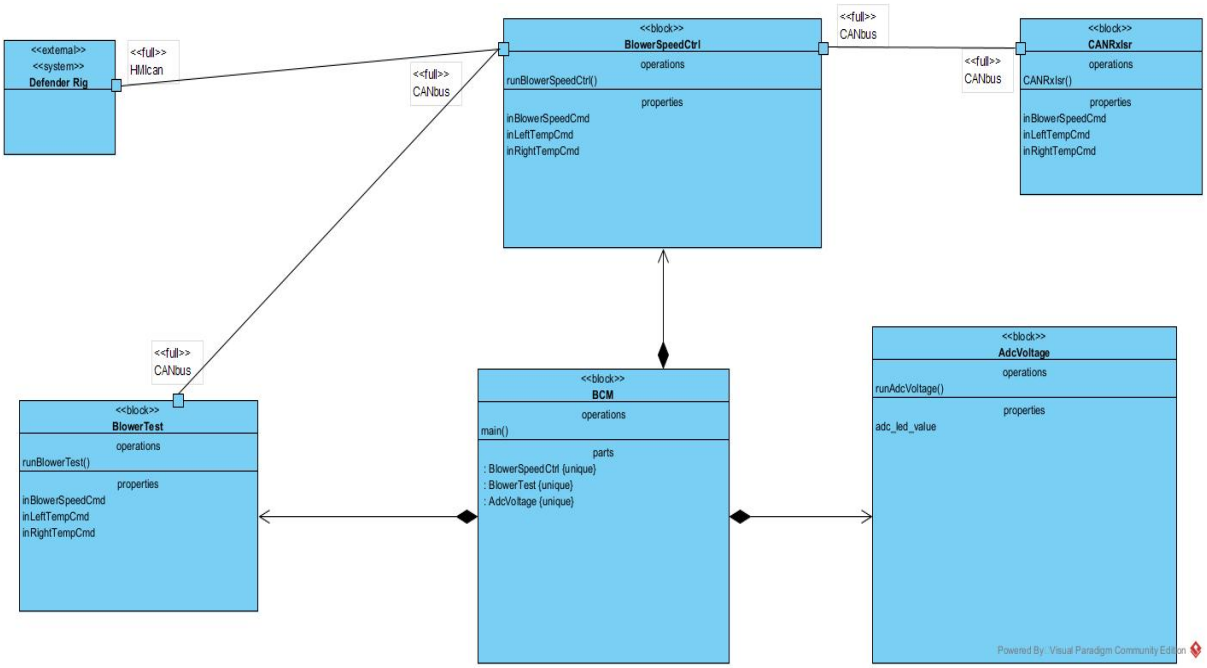
```
* main.c
 *
 *  Created on: 2024 Sep 26 15:31:53
 *  Author: YiJie
 */
```

```
#include "DAVE.h"
#include <BlowerSpeedCtrl.h>
#include <BlowerTest.h>
#include <CANRxIsr.h>
#include <AdcVoltage.h>
//Declarations from DAVE Code Generation (includes SFR declaration)

/**

 * @brief main() - Application entry point
 *
 * <b>Details of function</b><br>
 * This routine is the application entry point. It is invoked by the device startup code. It is responsible
for
 * invoking the APP initialization dispatcher routine - DAVE_Init() and hosting the place-holder for
user application
 * code.
 */
osThreadId tid_BlowerSpeedCtrlThread;
```

```
osThreadId tid_BlowerTestThread;

osThreadId tid_AdcVoltageThread;

osThreadDef ( runBlowerSpeedCtrl, osPriorityHigh, 1, 512);
osThreadDef ( runBlowerTest, osPriorityNormal, 1, 1024);
osThreadDef ( runAdcVoltage, osPriorityLow, 1, 1024);

osMessageQDef(message_q, 5, uint32_t);   //Declare a message queue
osMessageQId message_q_id;               //Declare an ID for the message queue


int main(void)
{
        DAVE_STATUS_t status;
        status = DAVE_Init();

        #ifndef TESTMODE

          /* Disable the CAN node for the test application to conserve energy if not operating in test
mode. */

          CAN_NODE_Disable( &TestNode );

        #endif

          if (status != DAVE_STATUS_SUCCESS)
          {
            /* Placeholder for error handler code. The while loop below can be replaced with an user
error handler. */
            XMC_DEBUG("DAVE APPs initialization failed\n");

            while(1U)
            {

            }
          }
```

```
  message_q_id = osMessageCreate(osMessageQ(message_q), runBlowerSpeedCtrl);

  tid_BlowerSpeedCtrlThread = osThreadCreate(osThread(runBlowerSpeedCtrl), NULL);

  tid_AdcVoltageThread = osThreadCreate(osThread(runAdcVoltage), NULL);

#ifdef TESTMODE
  tid_BlowerTestThread = osThreadCreate(osThread(runBlowerTest),NULL);
#endif

  osKernelStart();


  /* Placeholder for user application code. The while loop below can be replaced with user application
code. */
  while(1U)
  {
          __WFI();

  }
}

volatile uint32_t g_stat_timer_ticks;

void FreeRtosTraceIsr( void )
{
        g_stat_timer_ticks++;
}
```

4. SysML DIAGRAM



The Body Control Module (BCM) may be the central controller for an embedded system such as automotive or HVAC control systems, as shown in this UML diagram. The BCM manages three key components: BlowerSpeedCtrl (the blower speed control which handles blower speed control based on input command for blower speed and temperature), BlowerTest (performs blower functionality tests), and AdcVoltage (performs analog to digital voltage conversion). Using an CAN bus interface the CANRxIsr component receives input data like blower speed and temperature command. This CAN bus interacts with the system via an external Defender Rig. As a result of its modular design, the BCM's control of subsystem operations remains clear while having direct communication with the CAN bus.

5. CONCLUSION

The ability to implement a new low priority FreeRTOS task for ADC voltage polling on the BCM is shown with a demonstration. This task combines hardware interaction with LEDs in a way that's both clear and visible as ADC values. The principal features of RTOS for handling multiple tasks such as precise task scheduling, priority handling and seamless task integration are highlighted. Task's low CPU utilization and minimal interference with higher priority tasks is made possible by using the capabilities of the RTOS. Not only does it include message passing with CAN communication to demonstrate how the system can easily expand its functionality for the use of external rigs like the Defender or CANoe setups for integrated testing or full-fledged applications, but also. An implementation of this shows how these new features can be injected into the BCM without breaking current functions. The system is scalable, maintainable and flexible to future enhancements or to meeting new hardware requirements through adopting modular and reusable design principles.