Bachelor's thesis

Degree programme in Information Technology

NTIVIS14S

2018

NHUT TRAN

DEVELOPING AN EMBEDDED SYSTEM WITH CAN BUS PROTOCOL

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor of Information Technology

2019 | Total number of pages: 70

Author: Nhut Tran

DEVELOPING EMBEDDED SYSTEM WITH CAN BUS PROTOCOL

The purpose of this thesis was to develop an embedded system to manage a smart building with CAN bus protocol. However, the scope of implementation was reduced to a slave (tenant room) system. This system was developed on an STM32F446RE MCU with a DB18S20 (1-wire) temperature sensor, an LCD, a potentiometer and other electronic components. Besides that, in order to control the system, the software of this thesis was developed using the C programming language on Keil-MDK IDE. The results of this thesis were gaining knowledge of designing, developing an embedded system on ST MCU based on an ARM Cortex-M4 core, and the functional, stable system which is easy to use and can be developed further.

KEYWORDS:

embedded system, embedded application, home appliances, ARM Cortex-M4, STM32F446RE, CAN Bus Protocol

CONTENTS

FIGURES

LIST OF ABBREVIATIONS (OR) SYMBOLS

API	Application Programming Interface
GUI	Graphical User Interface
MCU	Microcontroller Unit
ECU	Electronic Control Unit
GPIO	General Purpose Input/output
AF	Alternate Function
ROM	Read-only Memory
USART	Universal Synchronous and Asynchronous Receiver-Transmitter
I^2C	Inter-Integrated Circuit
LCD	Liquid Crystal Display
HSI	High-speed Internal Clock
ACK	Acknowledge
OR	OR Bit Manipulation
ADC	Analog-to-Digital Conversion
DAC	Digital-to-Analog Conversion
PWM	Pulse Width Modulation
RTOS	Real-time Operating System
ISR	Interrupt Service Routine
NVIC	Nested Vectored Interrupt Controller
EXTI	External Interrupt/Event Controller
RMS	Rate-monotonic Scheduling
IDE	Integrated Development Environment
FPU	Floating-Point Unit
Kbps	Kilobits Per Second
ID	Identification

1 INTRODUCTION

In the modern life, embedded systems have become an essential part in many aspects of life. In fact, embedded systems have been applied widely in most of electronics and smart devices. For example, an embedded system could be in a radio, an oven, a mobile phone. They are not only applied in consumer devices, but also in industrial, in military devices and in other fields.

Furthermore, embedded systems have also been used to control, manage or measure large systems such as building management systems or industrial control systems.

Arising from the idea of controlling a system by using embedded system, the thesis aims to demonstrate the use of an embedded system in controlling a smart home system.

This thesis is divided into four chapters. Chapter 1 introduces the aim of the thesis . Chapter 2 introduces the theoretical background and the development tools. Chapter 3 provides an overview of this system and the limitations of this thesis. Chapter 4 discusses the experiential knowledge gained by the author through developing this system. Chapter 5 presents the testing methods and demo results.

2 THEORETICAL BACKGROUND

To understand the work carried out in this thesis, it is important to introduce the theoretical background of this work. Therefore, this chapter will introduce software packages, development tools, brief information about ST microcontroller with ARM core, real-time software (FreeRTOS) and tracing tools which have used to implement the work carried out in this thesis.

2.1 Keil MDK

Keil MDK is a complete software package from ARM which consists necessary development tools for developing embedded software on ARM Cortex-M MCUs[1]. For example, the Keil MDK package includes the µVision IDE, a debugger, an ARM compiler and other features.

- The µVision IDE: is very convenient for developer. It has many features which would help a developer to be able to manage project, build codes and debug embedded software[2].
    - The µVision IDE has many windows. For example, the µVision IDE has Project, Registers, Disassembly, and Watch windows.
        - The Project window manages the files of a project.
        - The Registers window tracks and, follows the data, mode and other status in registers of MCU.
        - The Watch window debugs, tests the value of variables in the software.
        - The example window in µVision IDE is illustrated in Figure I:

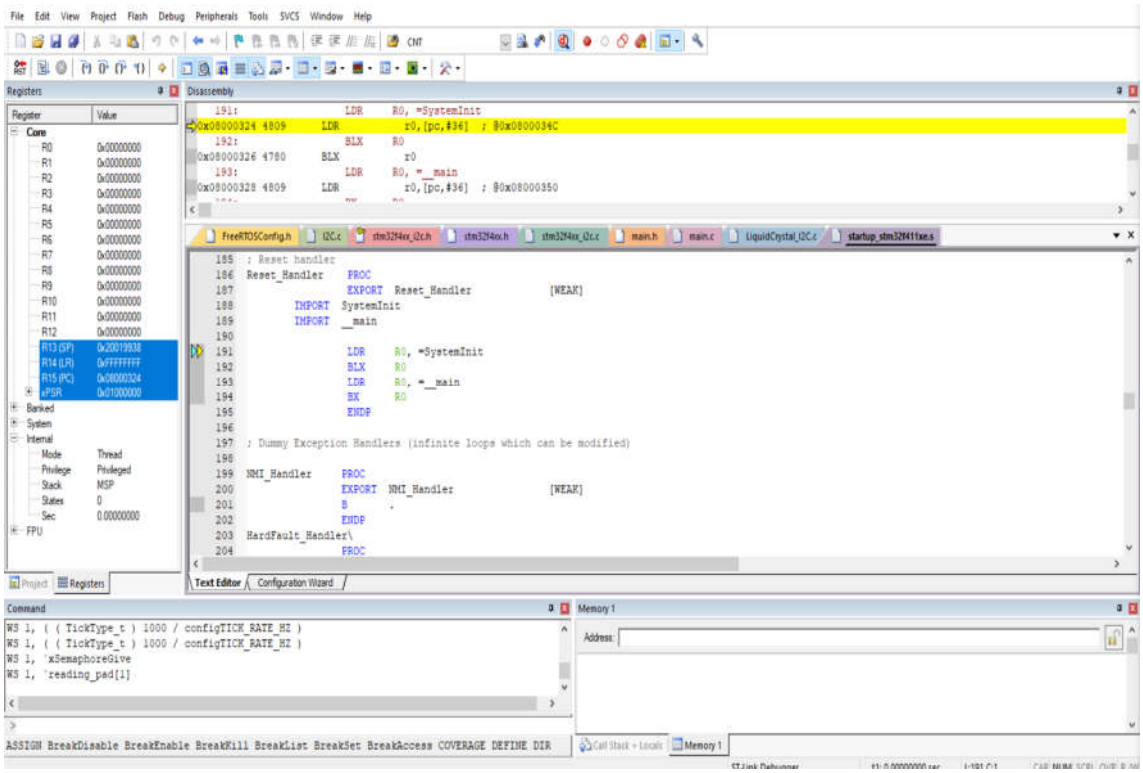


Figure I. µVision IDE Environment.

    - The µVision IDE also has the Manage Run-Time Environment feature which shall support essential files, packages, example projects for developing different MCU[2].
- The µVision Debugger supports different useful features for debugging. For example, the µVision Debugger supports break points, logic analyser software, data and event trace feature for debugging. However, to be able to use advanced features, Keil MDK must be upgraded to an upper version and use the ULINK debug adapter[3].
- The ARM Compiler is a toolchain which helps to compile codes, optimize software and support other features that would improve the performance of embedded software as well.

Besides that, Keil MDK also has different editions which depends-on developer demand. For example, Keil MDK-Lite is a free edition which shall be suitable for education, small projects (limited 32KBytes codes size) while MDK-Plus is another edition which shall support advanced features for commercial, professional projects (including different middleware – Ipv4 Networking, USB Device)[4].

2.2 ARM Cortex-M4 & STM32F446RE

2.2.1 ARM

ARM Cortex-M is the name of a 32-bit processor core group which is licensed by ARM. The ARM processor has been used in different microcontrollers, such as FPGAs, SoCs. Since Cortex-M was manufactured, it has replaced the old 8-bit microcontrollers which had been used widely.

Besides that, ARM does not provide or manufacture any microcontroller. Instead of that, ARM has licenced the processor designs, documents and provides them to other companies, parties for further designing, manufacturing microcontrollers.

In fact, the electronic companies have developed their own microcontrollers based on the ARM processor's architecture, such as the TI or ST company. In addition, to meet the different demands of the market, the electronic companies shall customize the microcontrollers with ARM architecture. For example, the microcontroller could be customized to be able to consume very low power or to have higher clock frequency[5].

Furthermore, there are different ARM Cortex-M cores ranging from M0 to M7. Each ARM Cortex-M core has different components, such as SysTick 24-bit Timer or Data cache. In addition, each core would have different instruction variation, for example, Cortex-M0 would take 3 stages of instruction pipeline whereas Cortex-M7 would take 6 stages[5].

2.2.2 ARM Cortex-M4

Cortex-M4 core would have:

- 1 to 240 interrupts.
- 12 cycle interrupt latency.

- Speed mode.
- 3-stages instruction pipeline.
- Other instruction set features.

Based on Cortex-M4 cores, there are many microcontrollers that have been manufactured. For example:

- Nordic nRF52.
- ST STM32 – F3,F4.
- Texas Instruments LM4F, MSP432.
- NXP LPC4000.

Besides that, the ARM Cortex-M4 cores have been included as secondary cores, for example, on NXP Vybrid VF6 (Cortex-A5 + Cortex-M4)[5].

2.2.3 STM32F446RE MCU

STM32F446RE is the name for microcontrollers which are manufactured by the ST company. STM32F446RE has been designed with an ARM Cortex-M4F core (Cortex-M4F means that the core has an FPU which shall enable the system to operate with floating point number[5][6])[7].

Furthermore, STM32F446RE (Figure II) has 512Kbytes of Flash memory, 128Kbyte of SRAM and back up SRAM is embedded up to 4Kbytes. It also has other useful features such as[7]:

- 2 different debug interfaces: SWD or JTAG
- 114 I/O ports. In 114 I/O ports, it has 111 fast I/O can speed up to 90MHz.
- 20 different communication interfaces:
    - I2C interfaces x 4
    - USART x 4 / UART x 2
    - SPI interfaces x 4
    - CAN interfaces x 2
- 3*12-bit ADC peripherals.
- 2*12-bit DAC peripherals.

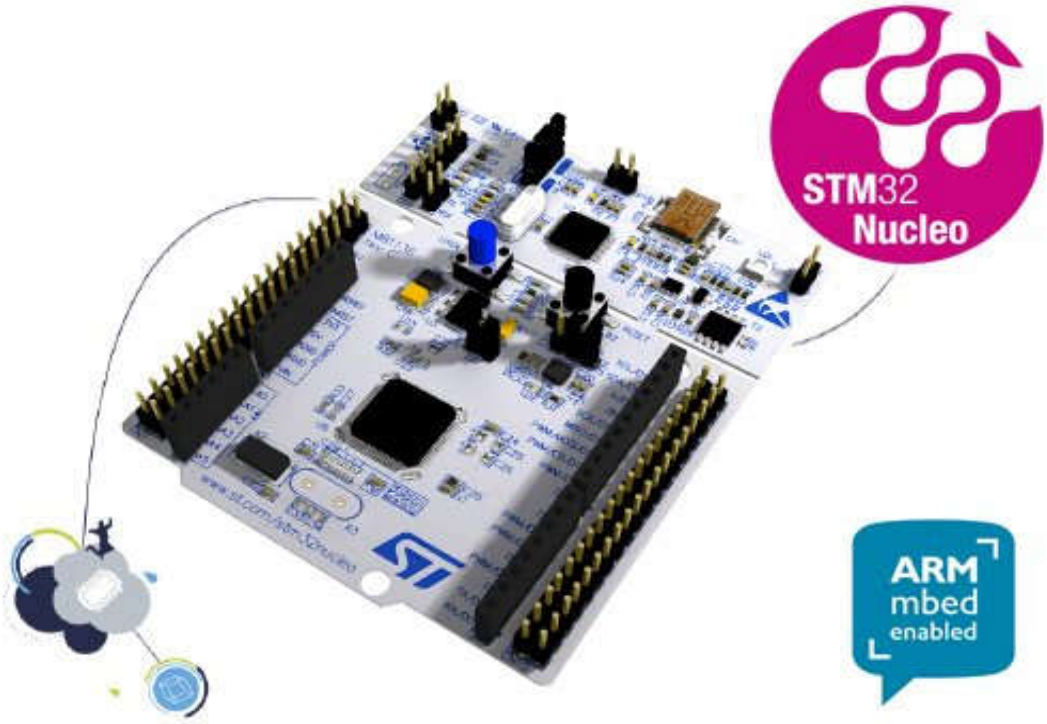


Figure II. STM32F446RE Board (https://www.amazon.com/STM32-Nucleo-64-development-STM32F446RE-NUCLEO-F446RE/dp/B01I8XLEM8).

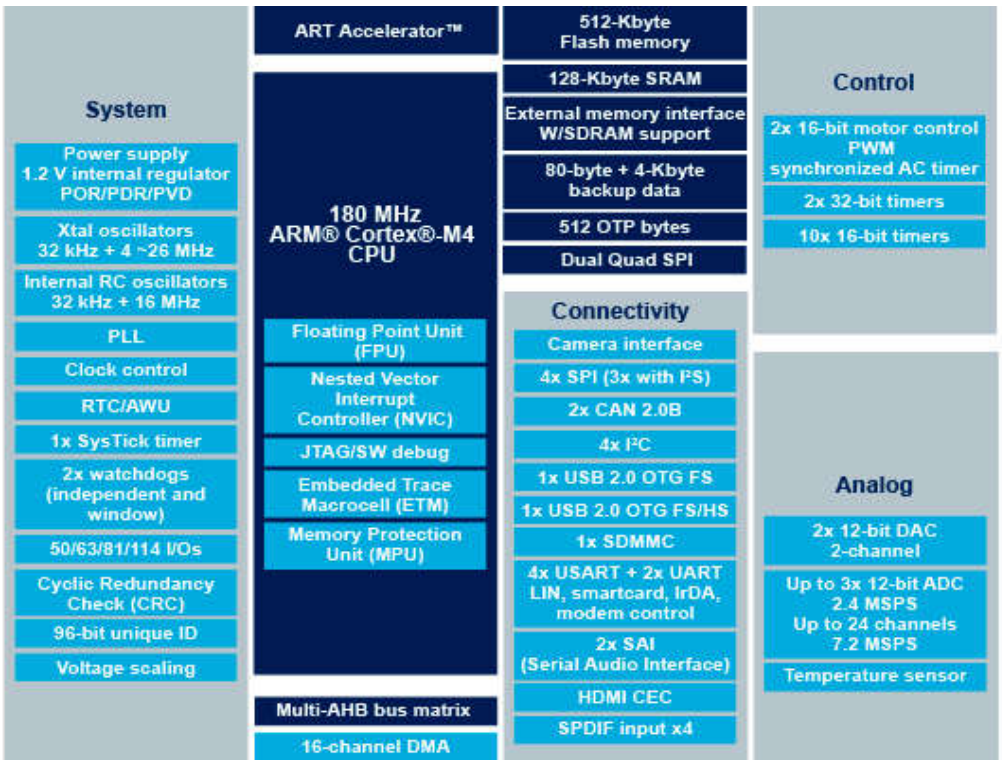


Figure III. Circuit Diagram of STM32F446RE Board (https://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series/stm32f446/stm32f446re.html).

2.3 FreeRTOS

Nowadays, there are plenty of RTOS software that have been developed to meet the requirements of real-time system for example, VxWorks, QNX, eCos, RTLinux [8]. One of most popular RTOS software is the FreeRTOS software which has been used widely in the embedded software field.

The FreeRTOS has been developed by Richard Barry and continued being maintained by Richard's company. Since 2017, FreeRTOS has continued its further successes with Amazon Web Services to bring the new experience, era in controlling system, securing connection to their users[9].

The reasons why FreeRTOS has become popular are the features of the software. First of all, it is very simple to use. For example, it provides many different API functions which have been written in conformity with the MISAR coding standard [10], therefore users shall be able to use, analyse the system easily. In addition, FreeRTOS has comprehensive features of real-time kernel such as memory allocation, locking mechanisms and different scheduling policies which are essential features for managing, scheduling tasks and protecting shared resources in real-time system.

FreeRTOS also has comprehensive documents, tutorial books with many different implementation examples. For example, it has the configuration of ***FreeRTOSConfig.h*** for different MCU architectures on their website, tutorial books and in software package.

Debugging the real-time system would be a challenge because errors might not be detected after few tests, however it would cause the system to stop or to work improperly at unpredictable time. For example, a deadlock would be a common problem in real-time system which would cause the system to stop at unpredictable time. Therefore, the FreeRTOS has supported many different trace hook functions and macros to debug the real-time system. For example, the StackOverflowHook function is used to detect the stack overflow fault or the traceTASK_SWITCHED_IN/_OUT function for tracing system while switching in or out tasks.

Moreover, FreeRTOS is not only a free software with comprehensive features for real-time system under MIT licenced which is modifiable, copiable for academic projects, but also able to use for commercial applications [11]. This also would be the primary reason why it had been downloaded every 120 seconds during 2017 [12].

2.4 Tracing Tool

Besides the useful hook functions and macros of FreeRTOS, there is also another powerful tracing tool from Percepio – the Tracealyzer software. Tracealyzer has been developed since 2004 with very friendly GUI and clear data visualization[13]. Traceanlyzer has support for different real-time operating systems, such as Amazon FreeRTOS or ARM Keil RT5. Furthermore, Tracealyzer has continued to support other RTOSs in its new upgrade versions.

Tracealyzer could be used for different purposes from debugging to analysing the behaviours of the real-time system along with different IDEs. With the Tracealyzer, the engineer shall be able to observe several different aspects on system level, have a deep understanding of real-time

issues in the system and trace the system's events. Besides that, Tracealyzer also has supported the new debugging methodologies which shall enable the engineer to observe the system in non-stop mode, instead of recording and stopping the system as the old debugging methodologies. This would be the powerful feature of Tracealyzer in debugging real-time system, especially in unstoppable system such as motor controlling system[13].

Tracealyzer has over 30 views to monitor the system. For example, in the Trace View window(Figure IV), it has task scheduling, task timing, task priority and other addition views to provide the comprehensive details of the system in staring and running time which would be the challenges with the old debugger[13].

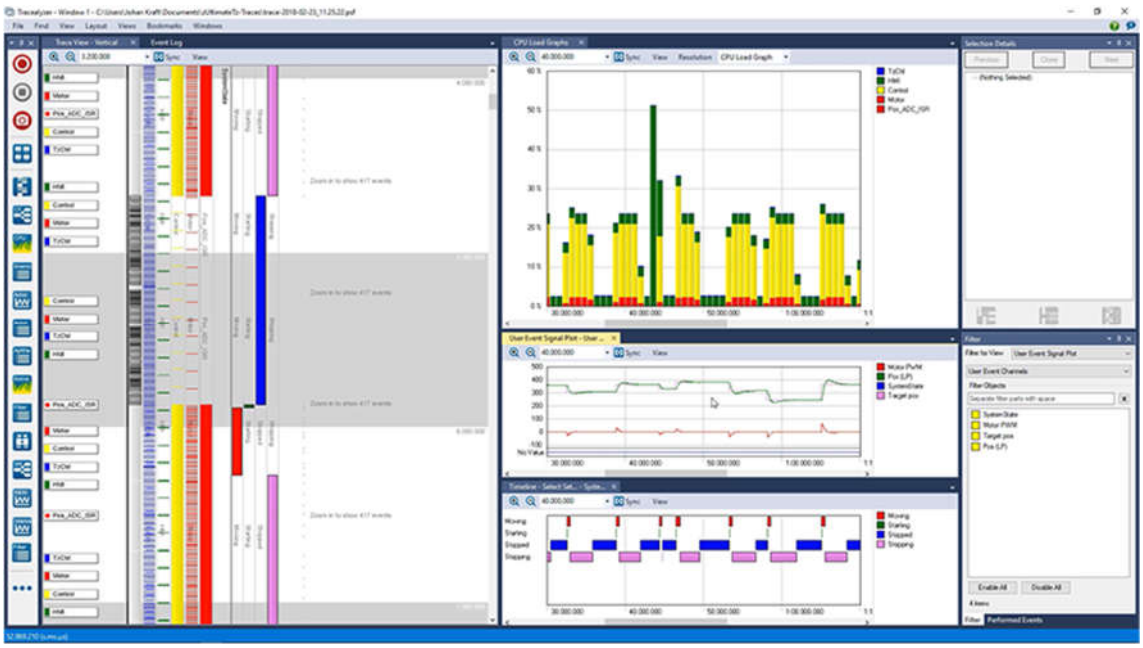


Figure IV. Tracealyzer Views (https://percepio.com/2018/08/21/tracealyzer-more-than-a-debugging-tool/).

Furthermore, Tracealyzer also has comprehensive technical documents, tutorial websites and white papers to guide the engineer step-by-step to configure the software, to use the software, to analyse the real-time system.

Even though the Tracealyzer software price is quite expensive, it has privileges for academic users. The academic users could acquire a free license for 1 year by sending the needed information to the support team of Percepio.

Debugging the real-time system is always a difficult challenge. In the past, engineers had to use different tools, electrical measuring instruments to debug, analyse and monitor the real-time system. Tracealyzer has been developed to reduce those ton of works to a convenient software package. Tracealyzer not only supports the traditional debugging methodologies (recoding and stopping to recode data), but also supports streaming methodology for unstoppable real-time system. This is the reason why it is used more widely.

2.5 CAN Bus Protocol

Arising from the requirements of connection in car system, the Controller Area Network (CAN) has been developed since 1983 by Robert Bosch GmbH, and the first CAN chips were manufactured by Intel and Philips in 1987. In the next few years, the CAN protocol was published as CAN 2.0 in 1991 which has two parts with different bit identifiers (part A has 11bit-identifiers and part B has 29bit-identifiers)[14].

Besides that, CAN bus standards were released in 1991 by International Organization for Standardization. In fact, there are three CAN protocol standards for data link layer and two physical layers based on transmitting speed.

The CAN protocol has been applied in variety of applications such as in automobile, navigation or elevator, building automation. In fact, in the automobile industry, the CAN protocol has helped to reduce the complexity and cost of wiring, and to connect different subsystems. Furthermore, the CAN bus protocol also has been applied on Shimano road bicycle since 2009[14].

Basically, the CAN bus protocol has two different physical layers: low-speed and high-speed with different architectures.

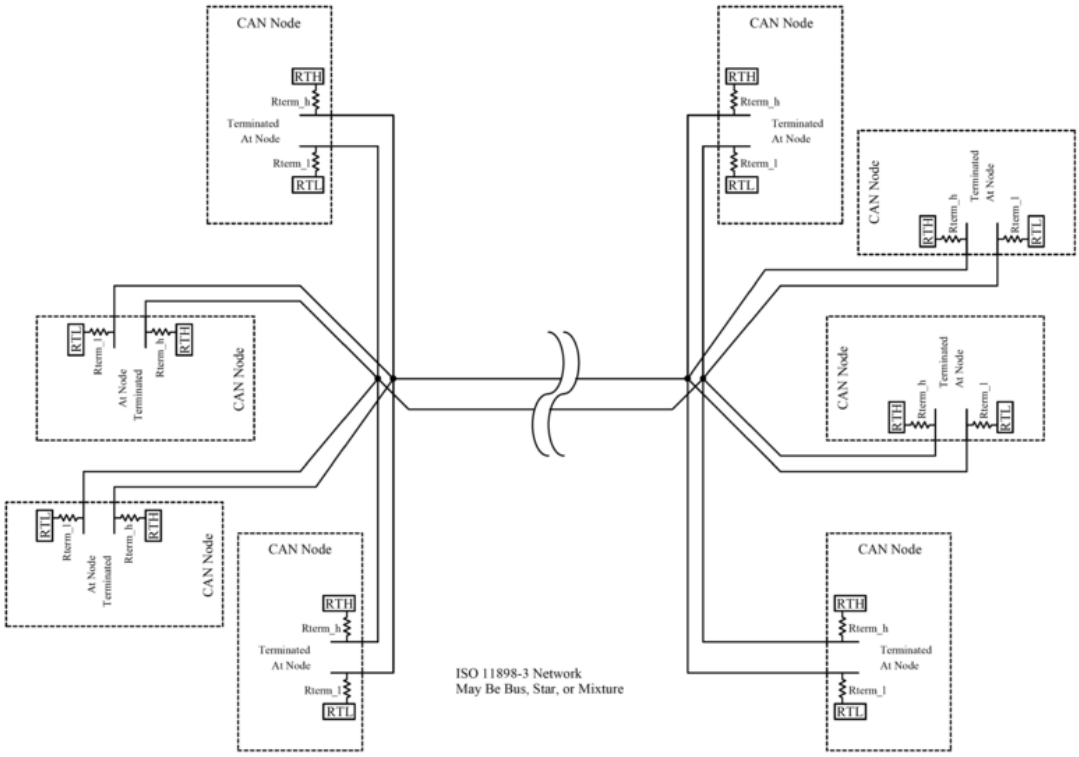


Figure V. Low-speed Fault Tolerant CAN Network (https://en.wikipedia.org/wiki/CAN_bus).

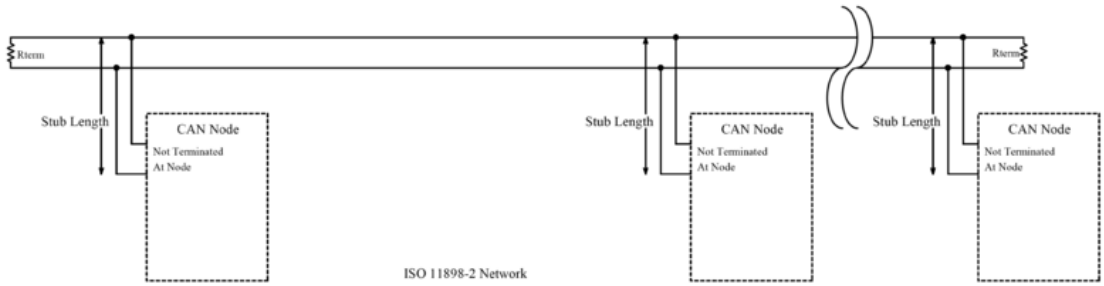


Figure VI. High-speed CAN Network (https://en.wikipedia.org/wiki/CAN_bus**).**

Low-speed CAN is an ISO- 11898-3 CAN standard (128 kbps) and is terminated at each node by a 100Ω resistor whereas High-speed CAN is ISO -11898-2 CAN standard (512 kbps) and terminated at each end of data buses by 120Ω resistor[14].

Basically, the microcontroller would have a CAN controller to transmit data to transceiver where the data is driven from/to the bus.

In addition, the CAN protocol also has some important characteristics which make the CAN protocol to be more reliable to be applied to real-time systems. For example, The CAN's transmission privileges higher priority messages to transmit before lower priority messages which infer that an important priority task would be responded, called on time without delay.

Transmission

The CAN protocol has been using a lossless bitwise arbitration method for synchronizing the sample data bit on CAN network. CAN has 3 different bit terms; they are dominant (0), recessive (1) and idle (0) bit. During the transmission, the transmitters will check the ID bits on CAN network and the node with a recessive (1) bit will be stopped if there is a node with dominant (0) bit on the bus[15]. Thus, the higher priority message will not be delayed by the other lower priorities. It is reasonable to infer that the higher priority ID has a lower number.

Layers

Basically, the CAN bus protocol has three different layers (Object, Transfer, Physical Layer), each layer will be responsible for different functions. For example, the Object layer will be responsible for filtering the noise on CAN network, the Transfer layer will be responsible for transmitting message, detecting error, and many other functions for the transmitting message between nodes on CAN network[14].

Frames

There are four types of frames in the CAN protocol and they are Data Frame, Remote Frame, Error Frame and Overload Frame.

Due to having two (2) different identifiers, the Data Frame of the CAN protocol has two different message formats. The base frame format is used for 11bit-identifiers (CAN 2.0 A) whereas the extended frame format is used for 29bit-identifiers (CAN 2.0 B).

The base data frame is as following:

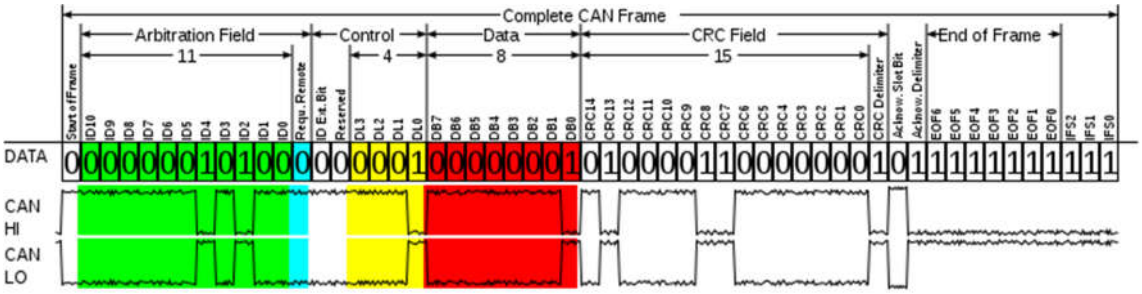


Figure VII. Base frame format CAN 2.0 A (https://en.wikipedia.org/wiki/CAN_bus).

The remote frame will be used for requesting data from the source by transmitting RTR-bit as recessive (1) bit and no data field [14].

The error frame will be transmitted if any frame detects an error [14].

The overload frame will be transmitted to delay the transmitting if the receiver requires a delay before receiving the next data frame or remote frame [14], or a higher priority messages is requiring the bus.

To sum up, the CAN bus protocol has privileges for high priority message which will be the essential feature for real-time system (without delay). Besides that, the CAN protocol also provides the high-speed of transmitting and convenient in connecting nodes. Those are undeniable reasons why CAN protocol still dominates in the automobile industry and has become more widely used in the embedded software fields.

3 PRODUCT CONTEXT

In the modern era, the smart home system has become widely known, controlling the light, temperature and other applications. Furthermore, to guarantee the safety of human life and properties, the smart home system of tenant rooms could be developed to be able to connect to the building management (master) system which would guard for the whole building.

The building management (master) system would be able to observe environmental data of tenant room (slave) systems and other addition features such as controlling RFID door lock of the building. The building management system not only acquire the environmental data for heating analysing, but also for safety purposes. For example, the building management (master) system would trigger the alarm and the fire department if the temperature of any tenant room is higher than threshold.

Besides that, there are several connecting protocols to connect master system with slave systems such as using SPI, I2C or wireless protocols. In addition, one of most reliable protocols is CAN bus protocol which is used widely in automobile industry for connection between MCUs, ECUs and sensors in car. Therefore, using CAN bus protocol for building management system would have many advantages, especially high transmitting speed (1 Mbit/s) [14].

The building management (master) system shall be designed to observe the temperature data of tenant rooms (slave systems) via CAN bus protocol and trigger the essential parts for safety.

The whole system would be as the following example diagram:

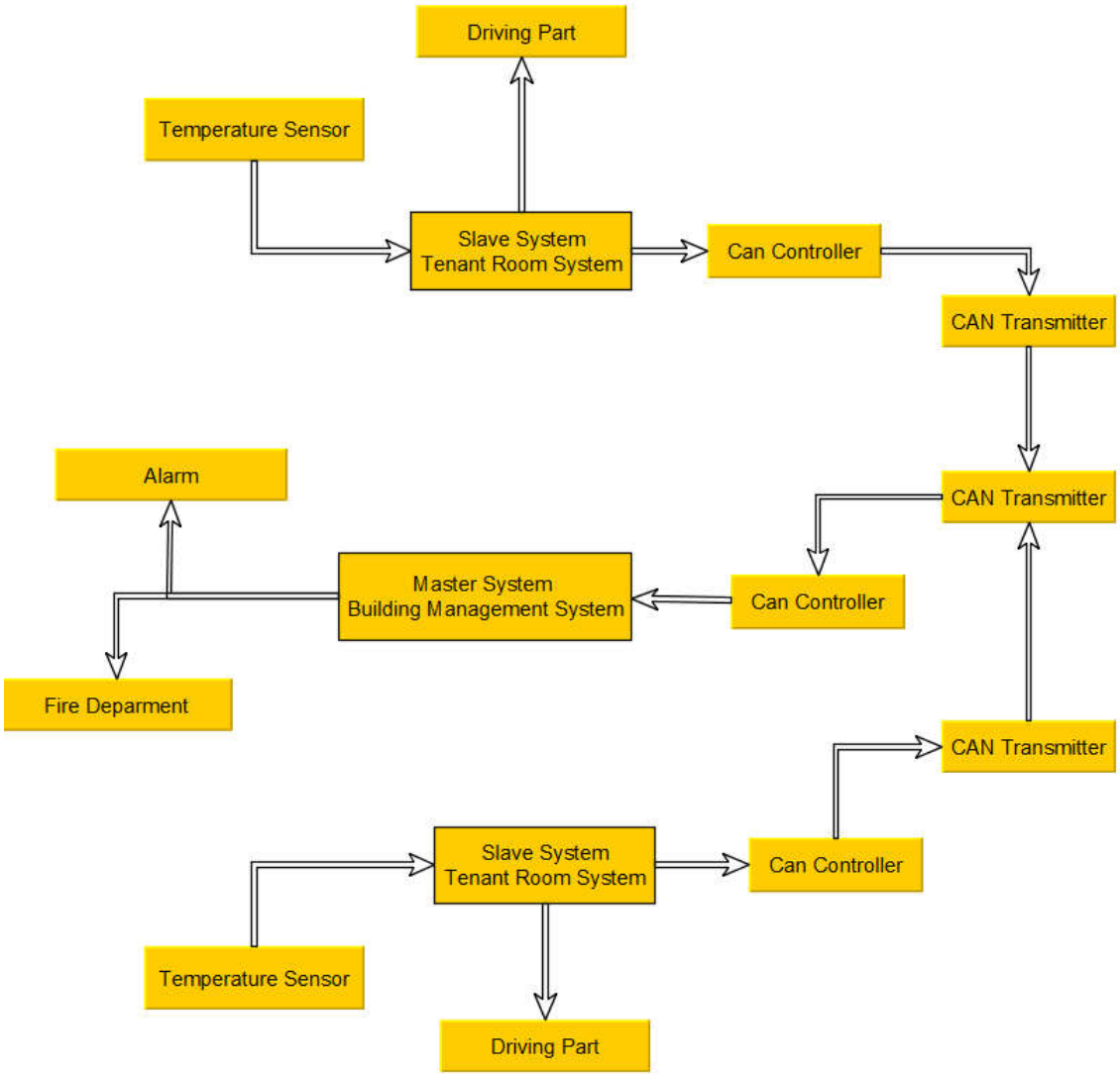


Figure VIII. Building Management System Diagram Example.

However, the building management system is a very large system which would be not suitable for a thesis. Hence, the scope of thesis is reduced to development and analysing the tenant room system (slave system).

The tenant room system (slave system) shall have data acquisition function to collect environmental data, driving parts to control, monitor the tenant room's environment.

4 EXPERIENTIAL PART

4.1 Requirements

4.1.1 System Requirements

The system must have data acquisition which shall collect temperature data from environment.

The system must have display function which shall monitor data for users.

The system must be able to control the fan's (motor) speed.

The system shall have two different modes: Auto-mode and User-mode

4.1.2 Device Requirements

The device shall be low-cost.

The device must have Timer for delay function.

The device must have DAC or Timer peripheral to generate clock pulse to drive fan (motor).

The device must have ADC peripheral to read/sample the input voltage from potentiometer.

The device must have USART peripheral for debugging through serial port.

The device must have I2C peripheral for communicating with external devices or/and sensors.

The device must have external interrupt/event controller for the user button.

4.1.3 Non-Functional Requirements

The system shall response immediately after the button is pressed.

The system shall run real-time – tasks meet deadlines and response in limited time.

4.2 Architecture – Designing

4.2.1 System

The system shall be as the following diagram:

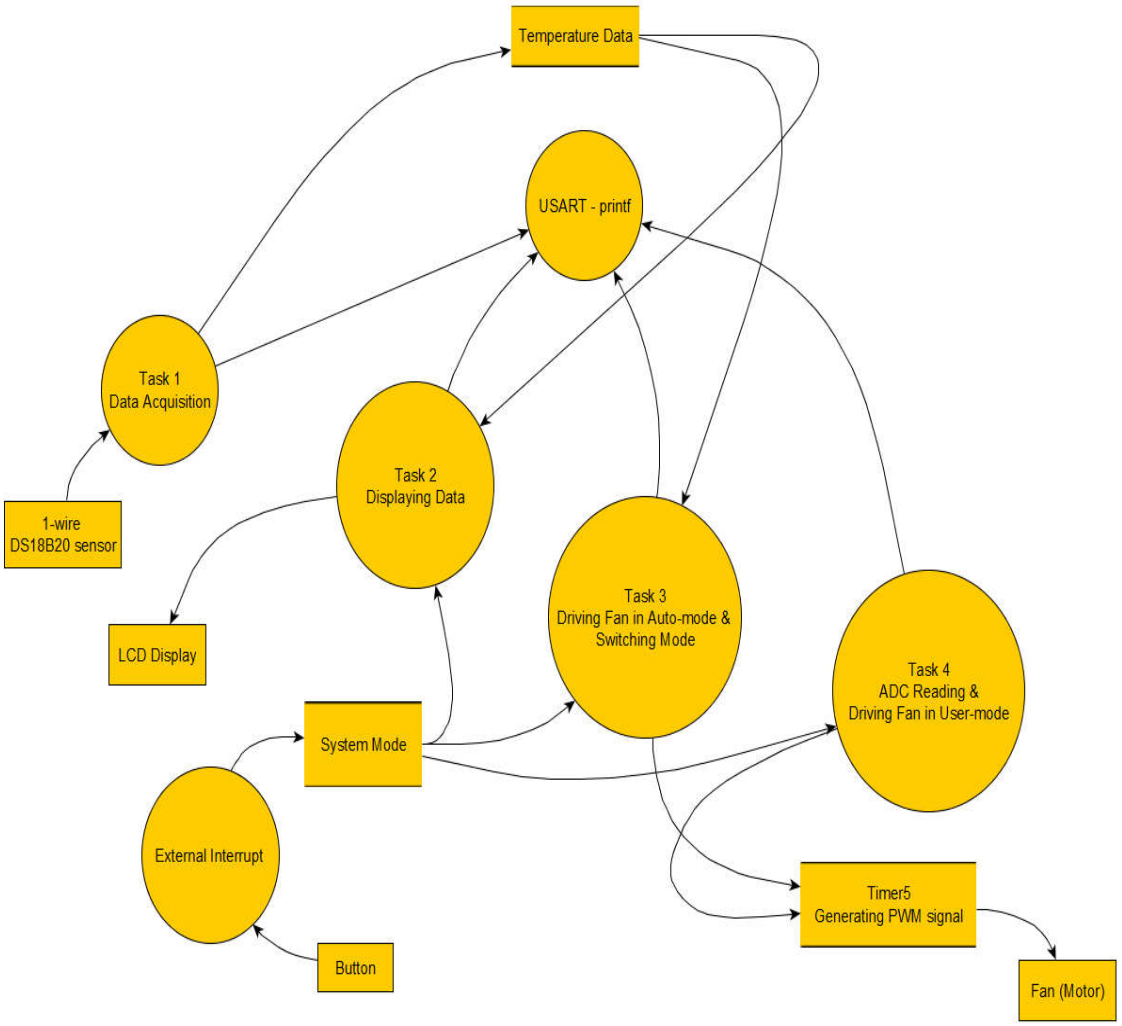


Figure IX. System Diagram.

4.2.2 Devices

This system shall use ST microcontroller STM32F446RE for device requirements. Because it not only meets the device requirements, but also has driver libraries, comprehensive documents for developing and the large community.

This system shall use LCD display 2004A – 20 characters x 4 lines has I^2C protocol, because of the ability of extension.

This system shall use DS18B20 – 1-wire sensor because it only needs 3 pin ports (1 common data pin port) and be able to connect up to 2^{64} 1-wire sensors/devices.

This system shall use fan – motor to simulate for the driving part.

This system shall use JBtek power supply to stabilize the power on breadboard.

This system shall use MOSFET - IRFD120 to control the voltage to drive fan (motor).

This system shall use Logic software and Logic analyser to analyse the logic output of GPIO pins.

This system shall use ST-Link on ST MCU to debug for the embedded software.

4.2.3 Software Development Tools

This system shall use Keil C (MDK-Lite) and C programming language to develop this project.

This system shall use FreeRTOS to manage and schedule tasks in system.

This system shall use Tracealyzer of Percepio to analyse the Real-time system.

This system shall use Realterm software for receiving data from serial port.

4.3 Implementation

4.3.1 System Clock

There are different clock sources for configuring system clock in ST MCU. It could be HSI or/and other clock sources.

To configure system clock, the system must define the value of clock source in header file.

For example, on STM32F446RE board, the system clock shall use HSI clock source (It is 16MHz). Therefore, the system must define it as the following example:

```
57 #if defined (HSI_VALUE)
58
59   #undef  HSI_VALUE
60   #define HSI_VALUE     ((uint32_t)16000000ul)
61 #endif
```

Figure X. Defining System Clock.

4.3.2 Delay Function

There are several different methods of generating delay function. It could be an empty loop to delay in an unpredictable time. However, in this project, the system must use Timer peripheral of MCU to generate delay functions to meet the precision requirements.

4.3.2.1 Hardware

There are several Timer peripherals on STM32F446RE, they would be 16bits to 32bits Timers. In this project, this system must use Timer 2 – 32bits Timer to be able to generate different delay functions (in microsecond, millisecond, second).

Basically, the idea of generating delay function in this system is based on the counter function of Timer on ST MCU. To generate the delay in this system, the system shall reset the counter of Timer to 0 and wait until the value of Timer's counter reaches the desired delay time.

4.3.2.2 Parameters

To initialize Timer, the system must configure some essential parameters for Timer according to the instructions of STM32F446RE (See p.521 STM32F446RE Reference Manual [16]) and Timer driver library of ST. The Timer's parameters shall be as the following example:

- Prescaler: shall be 15 (start from 0), then the counter clock frequency shall be (16[MHz] / 16) 1Mhz.
- Auto-reload (period): Because the system shall use solely one Timer for generating delay functions, therefore the Timer must have large value in auto-reload register which shall cause the counter overflow event when it reaches the setup value. In this project, the value of auto-reload register shall be the maximum value of Timer2 (2^{32}-bits value).
- Counter mode: up-counting.

4.3.2.3 Software

4.3.2.3.1 Initialization

To initialize the Timer peripheral, the system must follow the following steps:

- Enabling APB corresponding peripheral clock.
- Configuring Timer's parameters.
- Enabling the Timer.

Timer 2 shall be initialized as the following example:

```
3  void DELAY_Init()
4  {
5    TIM_TimeBaseInitTypeDef timerInitStructure;
6    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
7    timerInitStructure.TIM_Prescaler = 15; // counter rate is 1us (16MHz / (15 + 1) = 1MHz)
8    timerInitStructure.TIM_Period = 4294967295;
9    timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
10   timerInitStructure.TIM_ClockDivision = 0;
11   timerInitStructure.TIM_RepetitionCounter = 0;
12   TIM_TimeBaseInit(TIM2, &timerInitStructure);
13   TIM_Cmd(TIM2, ENABLE);
14 }
```

Figure XI. Delay Initialization Function.

4.3.2.3.2 Delay functions

Basically, the delay function shall set the Timer's counter to be 0 (by writing 0 into counter register TIMx_CNT) and wait until Timer's counter reaches the delay values.

The delay function shall be as the following examples:

```
18  void Delay_us(uint32_t us)
19  {
20    TIM_SetCounter(TIM2, 0);            // Make sure TIM2 Counter start from zero
21    while(TIM_GetCounter(TIM2) < us);   // Wait microseconds
22  }
23
24
25  void Delay_ms(uint32_t ms)
26  {
27    uint32_t temp = ms * 1000 ;
28
29    TIM_SetCounter(TIM2, 0);              // Make sure TIM2 Counter start from zero
30    while(TIM_GetCounter(TIM2) < temp);   // Wait ms Miliseconds
31  }
32
33  void Delay_s(uint32_t s)
34  {
35    uint32_t temp = s * 1000000 ;
36
37    TIM_SetCounter(TIM2, 0);              // Make sure TIM2 Counter start from zero
38    while(TIM_GetCounter(TIM2) < temp);   // Wait s second
39  }
40
```

Figure XII. Delay Functions.

4.3.3 USART

The USART has been used widely in embedded field. The mainly usages of USART are about sending and receiving data to/from the target device for debugging or controlling. In this project, USART shall be used for debugging.

4.3.3.1 Hardware:

There are several USART peripherals on STM32F446RE.

In this project, the system shall use the USART2.

To be able to use the USART peripheral of ST MCU, the system must to configure parameters for this peripheral and pass them into driver functions in USART driver library of ST company.

To enable the USART peripheral, the system must follow the following steps:

1. Enabling GPIO corresponding peripheral clock.
2. Enabling USART corresponding peripheral clock.
3. Configuring GPIO's parameters.
4. Configuring USART's parameters.
5. Enabling USART peripheral.

4.3.3.1.1 Parameters

- GPIO pins: PA_2(TX), PA_3(RX) is configured as AF pins (Alternate Function).
- Baud rate: 115200ul (unsigned long).
- Word length: 8-bits.
- Stop bit: 1-bit.
- Parity – checked bit: none.
- Mode: Transmitting and Receiving modes.

4.3.3.1.2 Transmitting

To transmit the data through USART, the system shall write the data into the USART_DR register. However, the system must wait for the TXE bit status register (USART_SR_TXE) until it is 1 (empty) to be able to write data into data register (USART_DR) (See p.801 STM32F446RE Reference Manual [16]).

4.3.3.1.3 Receiving

To read the data from UASRT, the system shall read the data in USART_DR register. However, the system must wait for the RXNE bit status register (USART_SR_RXNE) until it is 1 (data has been received) to be able to read data (See p.804 STM32F446RE Reference Manual [16]).

4.3.3.2 Software:

4.3.3.2.1 Initialization

As describe in Hardware section, the initialization function of USART shall be as the following example:

```
void USART2_Init()
{
  GPIO_InitTypeDef GPIO_InitStructure1;
  USART_InitTypeDef USART_InitStructure1;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);

  GPIO_InitStructure1.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
  GPIO_InitStructure1.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure1.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure1.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure1.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_Init(GPIOA, &GPIO_InitStructure1);

  GPIO_PinAFConfig(GPIOA, GPIO_PinSource2, GPIO_AF_USART2);
  GPIO_PinAFConfig(GPIOA, GPIO_PinSource3, GPIO_AF_USART2);

  USART_InitStructure1.USART_BaudRate = 115200ul;
  USART_InitStructure1.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure1.USART_StopBits = USART_StopBits_1;
  USART_InitStructure1.USART_Parity = USART_Parity_No;
  USART_InitStructure1.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure1.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
  USART_Init(USART2, &USART_InitStructure1);

  USART_Cmd(USART2, ENABLE);
}
```

Figure XIII. USART Initialization.

4.3.3.2.2 Transmitting Function

As described in Hardware section, the transmitting function shall be as the following example:

```
/*----------------------------------
  Write character to Serial Port
 *----------------------------------
int SER_PutChar (int ch) {

#ifdef __DBG_ITM
  ITM_SendChar (ch & 0xFF);
#else
  while (!(USART2->SR & 0x0080));
  USART2->DR = (ch & 0xFF);
#endif

  return (ch);
}
```

Figure XIV. Sending Data Through UASRT.

4.3.3.2.3 Receiving Function

As described in Hardware section, the receiving data function shall be as the following example:

```
/*----------------------------------
  Read character from Serial Port
 *----------------------------------
int SER_GetChar (void) {

#ifdef __DBG_ITM
  if (ITM_CheckChar())
    return ITM_ReceiveChar();
#else
  if (USART2->SR & 0x0020)
    return (USART2->DR);
#endif

  return (-1);
}
```

Figure XV. Reading Data Via USART.

4.3.3.2.4 Using printf & scanf

To use printf() and scanf() function to debug, the developer could use Retar.c file in example package of ST company.

Basically, Retar.c file shall redefine the fputc() and fgetc() function in <stdio.h> library to enable the developer to use printf() and scanf() function to send and receive data through serial port. On the PC target, the developer could use Realterm software for sending and receiving data.

4.3.4 1-Wire Temperature Sensor

The reasons for using 1-wire sensor DS18B20 are about its accuracy and the ability of extension.

4.3.4.1 Hardware

Basically, to be able to communicate with 1-wire device, the system must generate precise pulses and follow the transaction sequence.

According to the datasheet of DS18B20 [17], the system must follow these steps to communicate with the sensor:

1. Initialization – sending reset pulse to DS18B20.
2. ROM command – sending writing pulse and ROM command to DS18B20.
3. DS18B20 function command – sending writing pulse and function command (data) to DS18B20.

In addition, the 1-wire device shall need solely one port pin for communication, therefore the MCU could save GPIO pins for the other purposes. With 1-wire protocol, the system shall be able to connect up to 2^{64} (bit address) sensors, devices on the same GPIO pin.

In this project, the system shall use only one DS18B20 sensor for demo purpose. Therefore, after sending reset pulse to 1-wire sensor and receiving the present pulse from sensor, the system shall send “Skip ROM” command and be ready to send function command.

In practice, to read the temperate data on DS18B20, the system must follow the following steps:

- Sending reset pulse .
- Sending ROM command (Skip ROM).
- Sending function command (Convert Temperature).
- Sending reset pulse.
- Sending ROM command (Skip ROM).
- Sending function command(Read Scratchpad).
- Combining first two bytes of DS18B20 memory which contain the temperature data to be one 32bits variable.
- Converting that 32bits variable to be float number by dividing by 16.

The physical connection shall be as the following picture:

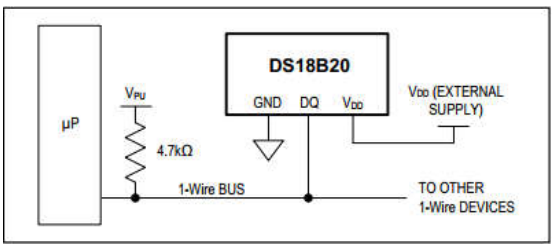


Figure XVI. Powering the DS18B20 with an External Supply - DS18B20 Datasheet.

4.3.4.2 Software

In this project, the software which use for communicating with DS18B20 has been fetched from the internet [18] and modified to be able to run on Real-time operating system (higher frequency).

3.4.2.1 Reset pulse

Timing diagram for reset – initialization:

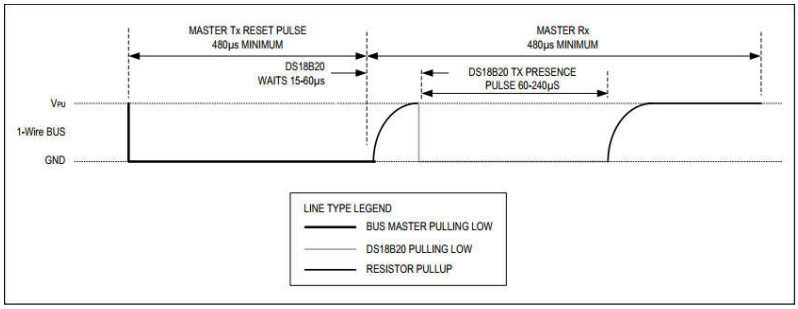


Figure XVII. Initialization Timing – DS18B20 Datasheet.

The reset pulse shall be generated as the following example:

```
171  void  SendInitialization(void)
172  {
173    ONEWIRE_OUTPUT_HIGH;       //pull to HIGH
174    ONEWIRE_CONFIG_OUTPUT;
175    Delay_us(50);
176
177    ONEWIRE_OUTPUT_LOW;        //pull to LOW
178    Delay_us(500);
179
180    ONEWIRE_CONFIG_INPUT;      //releasing bus
181    Delay_us(500);
182  }
```

Figure XVIII. 1-wire Reset Pulse - Initialization Function.

4.3.4.2.2 Sending/Writing data

To transmit ROM command or function command, the system also must to follow the 1-wire protocol.

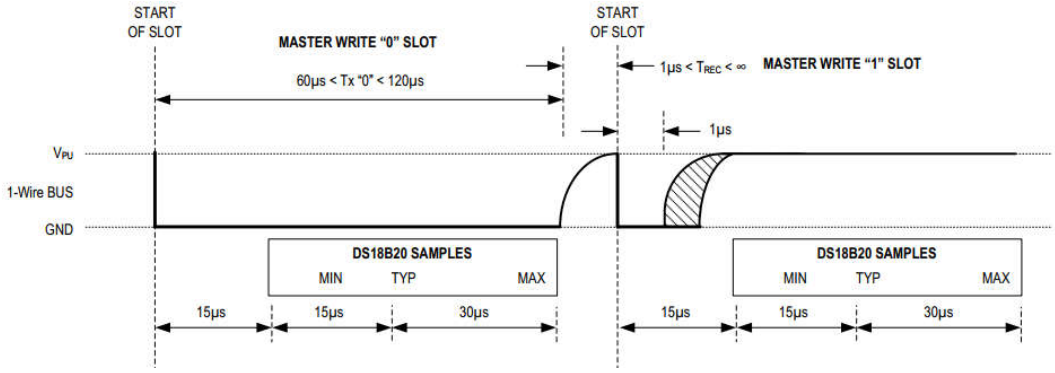


Figure XIX. Write Time Slot Timing Diagram - DS18B20 Datasheet.

The transmitting/writing function to DS18B20 shall be as the following example:

```
185  void  SendByte(uint8_t  val)
186  {
187    uint8_t       n;
188
189    for (n=0; n<8; n++)
190    {
191      ONEWIRE_OUTPUT_LOW;
192      ONEWIRE_CONFIG_OUTPUT;
193      Delay_us(5);
194      if (val & 1)  ONEWIRE_OUTPUT_HIGH;
195      Delay_us(95);
196      ONEWIRE_OUTPUT_HIGH;
197      Delay_us(5);
198      val = val >> 1;                          //Shifting bit to right for transmitting
199    }
200  }
```

Figure XX. Sending/Writing Function for DS18B20.

4.3.4.2.3 Receiving/Reading data

To receive data from DS18B20, the system also must follow the 1-wire protocol to be able to read the data from DS18B20 sensor.

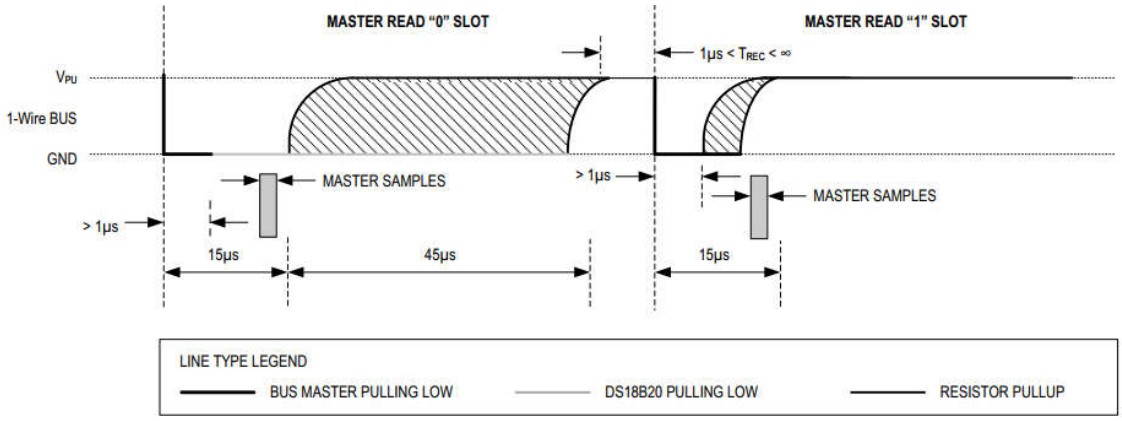


Figure XXI. Read Time Slot Timing Diagram – DS18B20 Datasheet.

The receiving/reading data function shall be as the following example:

```
203  uint8_t  ReadByte(void)
204  {
205    uint8_t       n;
206    uint8_t       val;
207
208    val = 0;
209    for (n=0; n<8; n++)
210    {
211      val = val >> 1;                          //Shifting bit to right for receiving
212      ONEWIRE_OUTPUT_HIGH;
213      ONEWIRE_CONFIG_OUTPUT;
214      Delay_us(3);
215      ONEWIRE_OUTPUT_LOW;
216      ONEWIRE_CONFIG_INPUT;
217      Delay_us(10);
218      if (ONEWIRE_INPUT_READ)  {val |= 0x80;}   //0x80 = 1000 0000
219      Delay_us(60);
220    }
221    return  val;
222  }
```

Figure XXII. Reading/Receiving Data Function for DS18B20.

4.3.4.2.4 Reading temperature data

As described in Hardware section above, the reading temperature data function after sending "Convert Temperature" function command shall be as the following example:

```
97  float ReportTemperature_2(void)
98  {
99    uint32_t      val;
100   float         temp;
101   uint8_t       n;
102
103   SendInitialization();
104   Delay_us(100);
105   SendByte(SKIP_ROM);
106   SendByte(READ_SCRATCHPAD);
107   for (n=0; n<9; n++)
108   {
109     reading_pad[n] = ReadByte();
110   }
111   val = (reading_pad[1] <<8) | reading_pad[0];      //Combine two bytes
112   temp = (float)val/16;                             //Converting byte value into temperature data
113   printf("\n\r 2 byte of temperature");
114   printf("\n\r     %02x %02x", reading_pad[1], reading_pad[0]);
115   printf("\n\rTemperature is: %0.4f degrees C\r\n\n", temp);
116   return temp;
117 }
```

Figure XXIII. Reading Temperature Data Function for DS18B20.

4.3.5 I^2C

The advantages of I^2C protocol are about the ability of extension and clock stretching.

With the I^2C protocol, then the system shall be able to connect up to 2^7 (bit address) I^2C sensors, devices on the same SDA, SCL pins.

Clock stretching shall enforce the master/slave device wait until the host is able to continue the process.

4.3.5.1 Hardware

There are several I^2C peripherals on STM32F446RE. In this project, the system shall use I2C1 of the MCU.

The I2C1 has 2 pins: PB_8(I2C_SCL) and PB_9(I2C_SDA).

There are two different I^2C modes: Slave mode and Master mode. The slave mode is the default mode on STM32F446RE and the MCU shall wait for Start condition from the master whereas the MCU shall generate Start condition in master mode (See p.761 STM32F446RE Reference Manual) [16].

Basically, the I^2C protocol shall operate with data, read/write, start, stop and ack bits. Therefore, to be able to read/write data from/to slave device, the system must to follow transfer sequence diagrams for master/slave transmitter/receiver (See p. 761 STM32F446RE Reference Manual) [16].

For example, in the master mode to transmit the data to slave device, the system must follow the following steps:

- Generating start bit because the default mode is slave mode and checking I2C_EVENT_MASTER_ MODE_SELECT (Checking Master/Slave bit, Bus busy bit, Start bit).

- Sending address bytes.
- If the sending address is matched with slave's address then, the master shall receive ACK bit. In this step, the master must check the I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED (checking Bus busy(I2C_SR2_BUSY), Master/slave(I2C_SR2_MSL), Address sent(I2C_SR1_ADDR), Data register empty (I2C_SR1_TxE), Transmitter/Receiver bit(I2C_SR2_TRA)). After this step, the system is ready for transmission. (I2C driver library of ST)
- Transmitting data bytes.
- After transmitting data, the master must generate the Stop bit to end the transmission.

Besides that, to be able to communicate with the devices on the I^2C bus, the system must configure parameters for I^2C peripheral properly.

- Clock speed: the parameter which shall define the speed of transmission between master and slave device. In this system, there is not any requirement about the clock speed, thus the clock speed shall be the standard speed (100KHz).
- Duty cycle shall be 50% to be compatible with standard mode frequency (100KHz).
- In this project, the system shall use 7-bit address for transmit with LCD display.
- Besides that, the I^2C peripheral pins must be pulled-up to meet requirement of I2C protocol [19].

4.3.5.1.2 Analysing I^2C Protocol

The I^2C protocol could be analysed by using different electronic measurements. In this project, the I^2C protocol shall be analysed by using Logic Analyzer and Logic software.

The I^2C signal shall be as the following example (transmitting vale + ACK bit):

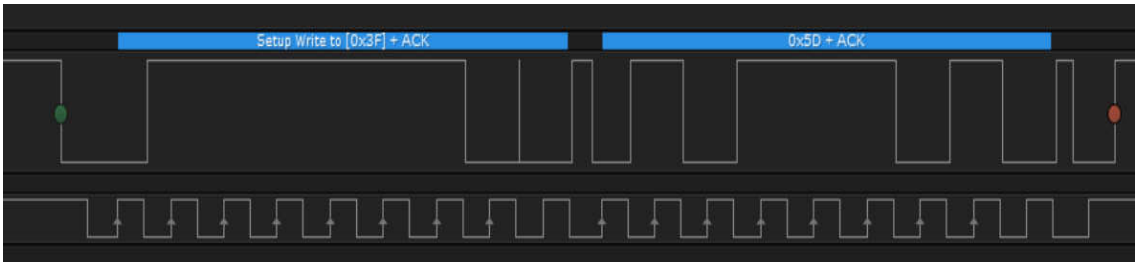


Figure XXIV. I^2C Signal - Analysed by Logic Analyzer and Logic Software.

4.3.5.2 Software

In this project, the software for I^2C has been fetched from the internet [20], and developed with the driver library of ST for configuring, checking and transmitting.

4.3.5.2.1 Initialization

As described in hardware section, the initialization function for I^2C shall be as the following example:

```
GPIO_InitTypeDef i2c_gpio;
I2C_InitTypeDef i2c;

void init_I2C1(void)
{
    // Enable periphrals clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB,ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

    // I2C output pin
    i2c_gpio.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9;
    i2c_gpio.GPIO_Mode = GPIO_Mode_AF;
    i2c_gpio.GPIO_Speed = GPIO_Speed_50MHz;
    i2c_gpio.GPIO_OType = GPIO_OType_OD;
    i2c_gpio.GPIO_PuPd  = GPIO_PuPd_UP;
    GPIO_Init(GPIOB, &i2c_gpio);

    GPIO_PinAFConfig(GPIOB, GPIO_PinSource8, GPIO_AF_I2C1);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource9, GPIO_AF_I2C1);

    // Setting value for I2C
    i2c.I2C_ClockSpeed = 100000;            // 100kHz
    i2c.I2C_Mode = I2C_Mode_I2C;
    i2c.I2C_DutyCycle = I2C_DutyCycle_2;    //50% Duty cycle
    // Setting slave device address
    i2c.I2C_OwnAddress1 = 0x00;
    i2c.I2C_Ack = I2C_Ack_Enable;
    i2c.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    I2C_Init(I2C1, &i2c);

    // Enable I2C
    I2C_Cmd(I2C1, ENABLE);
}
```

Figure XXV. I²C Initialization function.

4.3.5.2.2 Start Transmission

The transmission function must follow the transfer sequence which has been described in hardware section to generate the Start bit, transmit the 7-bit address to slave device and check status bits.

Therefore, the transmission function shall be as the following example:

```
/*****************************************************************/
void I2C_StartTransmission(I2C_TypeDef* I2Cx, uint8_t transmissionDirection,  uint8_t slaveAddress)
{
    // flag status from register
    while(I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY));
    // Generating I2C Start bit
    I2C_GenerateSTART(I2Cx, ENABLE);
    // Ждем пока взлетит нужный флаг
    //Checking the BUSY, MSL, ADDR, and relevant flags
    while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_MODE_SELECT));
    // Sending 7 bit slave address
    I2C_Send7bitAddress(I2Cx, slaveAddress<<1, transmissionDirection);
    // Checking transmittion direction
    if(transmissionDirection== I2C_Direction_Transmitter)
    {
      while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));
    }
    if(transmissionDirection== I2C_Direction_Receiver)
    {
      while(!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED));
    }
}
```

Figure XXVI. Start Transmission Function for I²C.

4.3.6 LCD Display

In this project, the LCD display is similar to the AMC2004A-I^{2}C module of the Orient Display company. This LCD Display shall be communicated by using I^{2}C protocol.

4.3.6.1 Hardware

Because the LCD display shall be communicated by using I2C protocol, the connection requires only SDA, SDL pins from the MCU's I^{2}C peripheral.

4.3.6.1.1 Initialization

To initialize the LCD, the system must initialize the LCD display by following the initializing sequence in the datasheet [21]. Basically, they shall be:

- Waiting after power on.
- Sending Function Set.
- Displaying On/Off.
- Clearing Display.
- Sending Entry Mode Set.
- Returning Home.

4.3.6.1.2 Reading/Writing Operation

This LCD display has two different registers, instruction (IR) and data register (DR)[21].

To initialize the LCD, the system must write data into instruction register.

To display data on LCD, the system must write data into data register.

The LCD can send the data in 4-bit or 8-bit operation [21], and it must be defined in Function Set command in initialization. In this project, the LCD shall receive the data in 4-bits.

To define which register the system shall write into, the system must send RS bit along with data bits. RS = 0 (Instruction register), RS = 1 (data register).

To define the command from system (reading or writing), the system must send the data along with R/W bit. R/W = 0 (Writing), R/W = 1 (Reading).

Besides that, to send reading/writing command to LCD, the system must follow the timing read/write diagram of LCD.

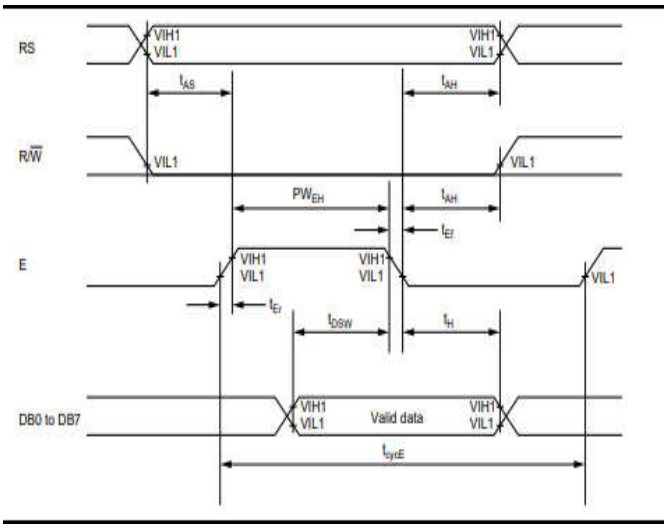


Figure XXVII. Writing Operation of LCD - LCD datasheet.

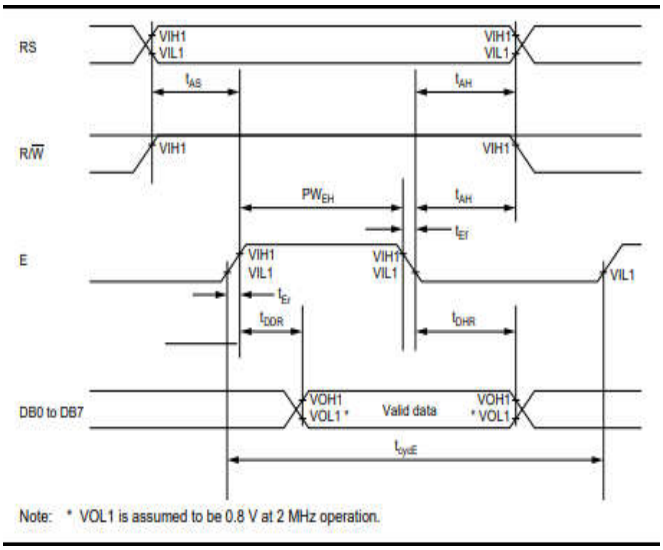


Figure XXVIII. Reading Operation of LCD - LCD datasheet.

Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	t_{cycE}	1000	—	—	ns
Enable pulse width (high level)	PW_{EH}	450	—	—	
Enable rise/fall time	t_{Er}, t_{Ef}	—	—	25	
Address set-up time (RS, R/$\overline{W}$ to E)	t_{AS}	60	—	—	
Address hold time	t_{AH}	20	—	—	
Data set-up time	t_{DSW}	195	—	—	
Data hold time	t_{H}	10	—	—	

Figure XXIX. Bus Timing Characteristics of LCD.

4.3.6.1.3 Display Operation

LCD has two different RAMs – DDRAM (Display data RAM) and CGRAM (Character Generator RAM).

When the system transmits the data to LCD along with RS bit (High) into DR register, the data shall be stored in DDRAM or CGRAM according to setting address.

If data is sent into DDRAM, the LCD shall match the data with display data in DDRAM and display it on the screen.

4.3.6.2 Software

In this project, the software for LCD display has been fetched from the internet [20], and developed by using I^2C driver library of ST company.

4.3.6.2.1 Initialization

As described in Hardware section, the system must send the configuration values to LCD and follow the initialization sequence. The initialization functions shall be as following examples:

- First, the system must take the configuration values:

```
45  void LCDI2C_init(uint8_t lcd_Addr,uint8_t lcd_cols,uint8_t lcd_rows)
46  {
47    lcdi2c.Addr = lcd_Addr;
48    lcdi2c.cols = lcd_cols;
49    lcdi2c.rows = lcd_rows;
50    lcdi2c.backlightval = LCD_NOBACKLIGHT;
51
52    init_I2C1(); // Wire.begin();
53    lcdi2c.displayfunction = LCD_4BITMODE | LCD_1LINE | LCD_5x8DOTS;
54    LCDI2C_begin(lcd_cols, lcd_rows);
55  }
```

Figure XXX. Initialization Function for LCD Display.

- Then, in the LCDI2C_begin function shall send the configuration values to LCD:

```
void LCDI2C_begin(uint8_t cols, uint8_t lines) {//, uint8_t dotsize) {
  if (lines > 1) {
    lcdi2c.displayfunction |= LCD_2LINE;
  }
  lcdi2c.numlines = lines;

  // for some 1 line displays you can select a 10 pixel high font
/*  if ((dotsize != 0) && (lines == 1)) {
    _displayfunction |= LCD_5x10DOTS;
  }*/

  // SEE PAGE 45/46 FOR INITIALIZATION SPECIFICATION!
  // according to datasheet, we need at least 40ms after power rises above 2.7V
  // before sending commands. Arduino can turn on way befer 4.5V so we'll wait 50
  Delay_ms(50);

  // Now we pull both RS and R/W low to begin commands
  LCDI2C_expanderWrite(lcdi2c.backlightval);  // reset expanderand turn backlight off (Bit 8 =1)
  Delay_ms(1000);

    //put the LCD into 4 bit mode
  // this is according to the hitachi HD44780 datasheet
  // figure 24, pg 46

    // we start in 8bit mode, try to set 4 bit mode
   LCDI2C_write4bits(0x03 << 4);
   Delay_ms(45); // wait min 4.1ms

   // second try
   LCDI2C_write4bits(0x03 << 4);
   Delay_ms(45); // wait min 4.1ms
```

Figure XXXI. Beginning Function for LCD Display - Part 1.

```
   // third go!
   LCDI2C_write4bits(0x03 << 4);
   Delay_ms(150);

   // finally, set to 4-bit interface
   LCDI2C_write4bits(0x02 << 4);


  // set # lines, font size, etc.
  LCDI2C_command(LCD_FUNCTIONSET | lcdi2c.displayfunction);

  // turn the display on with no cursor or blinking default
  lcdi2c.displaycontrol = LCD_DISPLAYON | LCD_CURSOROFF | LCD_BLINKOFF;
  LCDI2C_display();

  // clear it off
  LCDI2C_clear();

  // Initialize to default text direction (for roman languages)
  lcdi2c.displaymode = LCD_ENTRYLEFT | LCD_ENTRYSHIFTDECREMENT;

  // set the entry mode
  LCDI2C_command(LCD_ENTRYMODESET | lcdi2c.displaymode);

  LCDI2C_home();
}
```

Figure XXXII. Beginning Function for LCD Display - Part 2.

4.3.6.2.2 Transmitting function

Because of transmitting the data in 4-bits type in this project, the transmitting function must split the data byte into 4-bits data and add the mode-bit which shall indicate the data is command or data. The function shall be as the following example:

```
232 // write either command or data
233 void LCDI2C_send(uint8_t value, uint8_t mode) {
234   uint8_t highnib   = value&0xf0;
235   uint8_t lownib    = (value<<4)&0xf0;
236   LCDI2C_write4bits((highnib)|mode);
237   LCDI2C_write4bits((lownib)|mode);
238 }
```

Figure XXXIII. Splitting 4-bits Data Function.

In addition, as described in Hardware section, the system must follow the writing operation to send the data to LCD by pulling-up/pulling-down EN bit. The functions shall be as the following examples:

```
240 void LCDI2C_write4bits(uint8_t value) {
241   //LCDI2C_expanderWrite(value);
242   LCDI2C_pulseEnable(value);
243 }
```

Figure XXXIV. Transmitting 4-bit Data Function.

```
251 void LCDI2C_pulseEnable(uint8_t _data){
252   LCDI2C_expanderWrite(_data | En); // En high
253   Delay_us(1);    // enable pulse must be >450ns
254
255   LCDI2C_expanderWrite(_data & ~En);  // En low
256   Delay_us(40);   // commands need > 37us to settle
257 }
```

Figure XXXV. Transmitting Data with EN bit Function.

Afterall, to send the data by using I^2C protocol, the system must generate the Start condition on I^2C bus. The function shall be as the following example:

```
245 void LCDI2C_expanderWrite(uint8_t _data){
246   I2C_StartTransmission (I2C1, I2C_Direction_Transmitter, lcdi2c.Addr);
247   I2C_WriteData(I2C1, (int)(_data) | lcdi2c.backlightval);
248   I2C_GenerateSTOP(I2C1, ENABLE);
249 }
```

Figure XXXVI. Transmitting Data to LCD Display by using I2C connection.

4.3.6.2.3 Transmitting command function

To send the command to LCD display, the RS-bit must be LOW (0) and be sent along with data-bits. The function for transmitting command to LCD display shall be as the following example:

```
225 void LCDI2C_command(uint8_t value) {
226   LCDI2C_send(value, 0);
227 }
```

Figure XXXVII. Transmitting Command Function to LCD Display.

4.3.6.2.4 Transmitting data function

To send the data to LCD display, the RS-bit must be HIGH (1) and be sent along with data-bits. The function for transmitting data to LCD display shall be as the following example:

```
18 void LCDI2C_write(uint8_t value){
19   LCDI2C_send(value, Rs);
20 }
```

Figure XXXVIII. Transmitting Data Function to LCD Display.

The RS value must be defined in header file.

```
58 #define Rs 0x01  // Register select bit
```

Figure XXXIX. Defining RS-bit

4.3.6.2.5 Searching I2C Slave Address

The I^2C address of LCD device sometime is not exactly same with the description in datasheet. Due to that fault, the system could use Searching Address function to identify the I^2C address of LCD. The Searching Address function shall be as the following example:

```
302 //Searching I2C address
303 uint8_t LCDI2C_Searching_Address(void)
304 {
305   //uint16_t a = 0;
306   for(uint8_t i = 0; i < 128; i++)
307   {
308     lcdi2c.Addr = i;
309     while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
310     I2C_GenerateSTART(I2C1, ENABLE);
311     while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));
312     I2C_Send7bitAddress(I2C1, lcdi2c.Addr<<1, I2C_Direction_Transmitter);
313     Delay_ms(500);
314
315     if(I2C1->SR1 & (1 << 7))    //Checking Txe Flag Bit in SR1 register
316     {
317       I2C_GenerateSTOP(I2C1, ENABLE);
318       return i;                 //Return the matched address
319     }
320     else
321     {
322       I2C_GenerateSTOP(I2C1, ENABLE);
323     }
324   }
325   return 0;
326 }
```

Figure XL. Searching I2C Address of LCD device.

Basically, this function shall send address value in for loop (2^7 times = 2^7 address values) to LCD device and return the address value if the TxE bit in status register of I2C peripheral (I2C_SR1_TxE) is HIGH (1). Because the TxE bit is set HIGH (1) only when the system receives the

ACK bit (1) and the next bytes is transmitted successful without receiving PEC bit (1) (See p.787 in STM32F445 Reference Manual) [16].

4.3.7 Driving Fan (Motor)

There are several methods to drive, control the fan's (motor) speed (using Timer, DAC). In this project, the system shall use Timer to generate clock pulse (PWM signal) to drive fan.

4.3.7.1 Hardware

In this project, the system shall use output compare function of Timer5 to generate PWM signal to drive fan.

According to the instruction (See p.542 STM32F446RE Reference Manual [16]), the system must write the desired duty cycle value into TIMx_CCRx register to generate the PWM signal. In fact, the Timer shall compare the counter's value with input capture/compare (TIMx_CCRx) register value to generate PWM output signal and the output PWM signal is depended on the direction of the counter. For example, the output PWM signal shall be high as long as the TIMx_CNT(counter's value) < TIMx_CCRx else it shall be low (See p.543 in STM32F446RE Reference Manual) [16].

To enable Timer 5 of ST MCU to use output compare function, the system must follow the initialization sequence (See p.541 Reference Manual) [2] and Timer driver library of ST company. Basically, the initialization sequence shall be as the following steps:

- Enabling Timer peripheral clock
- Configuring corresponding output pin of Timer as AF (alternate function).
- Configuring the Timer base unit (prescaler, period, counting mode,…).
- Configuring the Timer output compare function with desired parameters (enabling output mode, setting polarity,…)[22].
- Enabling the Timer counter.

3.7.1.1 Parameters

Prescaler: 15 (16MHz / 16 = 1MHz).

Period: 65535 (0xFFFF – 16bits value).

Counting mode: up-counting.

Using PWM1 of Timer 5.

Polarity: active high (see [8] for example).

Default value of output: 65535 (0xFFFF – 0% duty cycle).

4.3.7.2 Software

4.3.7.2.1 Initialization

As described in hardware section, the initialization function shall configure output pin, Timer parameters and pass those parameters to functions in GPIO and Timer driver library of ST company to be able to generate PWM signal.

The initialization function shall be as the following example:

```
void TIM_PWM_Init(void)
{
  TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;
  TIM_OCInitTypeDef          TIM_OCInitStructure;
  GPIO_InitTypeDef           GPIO_InitStructure;

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
  GPIO_Init(GPIOA, &GPIO_InitStructure);


  GPIO_PinAFConfig(GPIOA, GPIO_PinSource0, GPIO_AF_TIM5);

  /* Time base configuration */
  TIM_TimeBaseStructure.TIM_Prescaler = 0xF;
  TIM_TimeBaseStructure.TIM_Period = 0xFFFF;   // 65535
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;

  TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);

  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High;
  TIM_OCInitStructure.TIM_Pulse = 0xFFFF;
```

```
  TIM_OC1Init(TIM5, &TIM_OCInitStructure);
  TIM_OC1PreloadConfig(TIM5, TIM_OCPreload_Enable);

  TIM_ARRPreloadConfig(TIM5, ENABLE);

  /* TIM1 enable counter */
  TIM_Cmd(TIM5, ENABLE);
}
```

Figure XLI. Initialization PWM Function.

4.3.7.2.2 Driving Fan

As described in Hardware section, the system must write the value to capture/compare register (TIMx_CCRx) to be able to adjust the PWM signal.

The driving_Fan function shall be as the following example:

```
350  void driving_Fan(const uint16_t *value)
351  {
352    TIM5->CCR1 = *value * 65535 / 100;
353  }
```

Figure XLII. Driving Fan function.

4.3.8 ADC – Reading User Control Fan (Motor)

To read the changing of input voltage from potentiometer , the system must use ADC peripheral of ST MCU.

4.3.8.1 Hardware

Basically, the ADC peripheral shall read the input voltage and convert it to digital value which is compatible with the system.

To be able to use ADC peripheral, the system must follow the initialization sequence (see p.361 STM32F446RE Reference Manual) [16] and ADC driver library of ST company. Basically, the initialization sequence shall be as the following steps:

- Enabling the ADC peripheral clock.
- Configuring ADC's corresponding input pin (configure GPIO pin as AN – analog).
- Configuring ADC's parameters (data align, resolution…).
- Configuring ADC's conversion mode.
- Configuring ADC's channel, sampling time.
- Enabling ADC peripheral.

4.3.8.1.1 Parameters

Due to no special requirement for the measuring input voltage, the system shall need only one ADC channel for reading input voltage value. Therefore, the scan mode shall be disable (see [23] for more information).

GPIO pin: PA_1 (ADC123_IN1 – ADC1/2/3_Channel1), configured as AN.

Conversion mode: Regular.

Resolution: 12bits. (ADC_CR1_RES)

Data align: Right (See p.366 STM32F446RE Reference Manual) (ADC_CR2_ALIGN).

Number of Conversion: 1 (because there is no demand for checking sequentially input value in many times). (ADC_SQR1_L).

Because of using regular channel, continuous conversion mode and external triggered conversion shall be disable.

Sampling time: 84 cycles [T_{conv} = 84 + 12 = 96 = 6µs (16MHz / 96) with ADCCLK = 16MHz] (see p.366 STM32F446 Reference Manual [16]).

4.3.8.1.2 Physical requirement

According to the ADC features of STM32F446RE (see p.354 STM32F446RE Reference Manual) [16], the ADC input range must be $V_{REF-} \leq V_{IN} \leq V_{REF+}$ (see p.141 STM32F446RE datasheet [15]). Thus, the V_{DD}(3.3v) of MCU shall be sufficient to supply for V_{DDA} of ADC.

4.3.8.1.3 Reading ADCs Conversion Value

To be able to read the ADC's conversion value in regular mode, the system must enable SWSTART bit in control register (ADC_CR2) to start the conversion and wait until ECO1 (end of conversion) flag is set (1) in status register (ADC_SR). After that, the system shall be able to read the ADC conversion value in regular data register (ADC_DR) (see p.361 STM32F446RE Reference Manual) [16].

4.3.8.2 Software

4.3.8.2.1 Initialization

As described in Hardware section, the initialization function shall configure essential parameters for ADC and pass them into API driver functions to be able to use ADC peripheral.

The initialization function shall be as the following example:

```
 4  void Adc_Start_Init()
 5  {
 6    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
 7
 8    GPIO_InitTypeDef GPIO_InitStruct;
 9    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AN;
10    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_1;
11    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
12    GPIO_Init(GPIOA, &GPIO_InitStruct);
13
14    // Init ADC1
15    ADC_InitTypeDef ADC_InitStruct;
16    ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;
17    ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;
18    ADC_InitStruct.ADC_ExternalTrigConv = DISABLE;
19    ADC_InitStruct.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
20    ADC_InitStruct.ADC_NbrOfConversion = 1;
21    ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
22    ADC_InitStruct.ADC_ScanConvMode = DISABLE;
23    ADC_Init(ADC1, &ADC_InitStruct);
24    ADC_Cmd(ADC1, ENABLE);
25
26    ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_84Cycles);
27  }
```

Figure XLIII. Initialization Function for ADC peripheral.

4.3.8.2.2 Reading Function

As describe in Hardware section, the reading function shall use functions in driver library of ST company to start conversion and read the conversion value.

The reading function shall be as the following example:

```
29  uint16_t ADC_Read()
30  {
31    // Start ADC conversion
32    ADC_SoftwareStartConv(ADC1);
33    // Wait until conversion is finish
34    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
35
36    return ADC_GetConversionValue(ADC1);
37  }
```

Figure XLIV. Reading ADC Conversion Value Function.

4.3.9 Real-Time Operating System

The purposes of using RTOS are to schedule, manage tasks and protect shared resources. Besides that, some features of RTOS are very useful for debugging and analysing the system.

In this project, the system shall use FreeRTOS to meet the requirements of system.

4.3.9.1 Using FreeRTOS

To be able to apply FreeRTOS into the system, the system must follow the RTOS start instruction [24]. Basically, those steps shall be:

- Downloading FreeRTOS software.
- Including corresponding files according to MCU.
- Modifying the FreeRTOSConfig header file according to MCU's characteristics.
- Building demo project.
- Applying FreeRTOS into the system.

In this project, the system has used STM32F446RE MCU, thus the following steps shall be needed for applying FreeRTOS into the system:

- Including files in "Source" directory (folder) and their header files in "include" directory (folder).
- Including ***port.c*** file in "/Source/portable/RVDS/ARM_CM4F/" and its header file.
- Including ***FreeRTOSConfig.h*** file in "/FreeRTOS/Demo/CORTEX_M4F…"
- Configuring ***FreeRTOSConfig.h*** file according to the system requirements and STM32F446RE characteristics.

See the ***FreeRTOSConfig.h*** example file in Appendix 3.

4.3.9.2 Tasks

To be able to manage and schedule works in the system, the system shall be divided up into 4 tasks.

- Task 1 – Data Acquisition Task.
- Task 2 – Displaying Task.
- Task 3 – Driving Fan in Auto-mode.

- Task 4 – Driving Fan in User-mode.

4.3.9.2.1 Task 1 – Data Acquisition Task

In Task 1, the system shall use readingFunction to read the temperature from 1-wire DS18B20 sensor and use the printf function to transmit the data through USART peripheral to target PC for debugging.

Basically, readingFunction shall initialize the communication with 1-wire DS18B20 by sending reset pulse and use the ReportTemperature_2 function to read the temperature data on DS18B20 sensor. The temperature data shall be stored in a global variable (reading_Temp).

The readingFunction function shall be as the following example:

```
322  void readingFunction()
323  {
324    StartConversion();
325    Delay_ms(1);
326    reading_Temp = ReportTemperature_2();
327  }
```

Figure XLV. readingFunction for Acquiring Temperature Data in Task 1.

Task 1 shall be as the following example:

```
127  void vTask1( void *pvParameters )
128  {
129  const char *pcTaskName = "Task 1 is running\r\n";
130  volatile unsigned long ul;
131  const TickType_t xDelay = 100 / portTICK_PERIOD_MS;
132  TickType_t xLastWakeTime;
133  //const TickType_t xFrequency = 500;
134
135    xLastWakeTime = xTaskGetTickCount();
136
137    /* As per most tasks, this task is implemented in an infinite loop. */
138    for( ;; )
139    {
140      #ifdef DelayUntil
141        vTaskDelayUntil( &xLastWakeTime, xDelay );
142      #endif
143      if(xSemaphoreTake( xSemaphore, ( TickType_t ) portMAX_DELAY )==pdTRUE)
144      {
145
146        #ifdef DEBUG_BY_TICK
147          printf("\r\rTich Count Task 1: %d\n", xTaskGetTickCount());
148        #endif
149
150        printf( "%s\n",pcTaskName );
151        readingFunction();
152
153        #ifdef DEBUG_BY_TICK
154          printf("\rTich Count END Task 1: %d\n", xTaskGetTickCount());
155        #endif
156
157        xSemaphoreGive( xSemaphore);
158
159        #ifdef DelayOnly
160          vTaskDelay( xDelay );
161        #endif
162      }
163    }
164  }
```

Figure XLVI. Task 1 - Data Acquisition Task.

4.3.9.2.2 Task 2 – Displaying Task

In Task 2, the system shall use displayFunction to display the temperature data on LCD along with system mode. Besides that, Task 2 shall use printf function to transmit the data through USART peripheral to target PC for debugging also.

Basically, the displayFunction shall erase the old values on LCD and display new values which has been acquired from Task 1. Besides that, the system must use sprintf function to convert float value of reading_Temp variable to char because the system can only transmit the data in char type.

The displayFunction shall be as the following example:

```
355  void displayFunction(char *str, char *mode)
356  {
357    LCDI2C_setCursor(6,1);
358    LCDI2C_write_String("     ");
359    LCDI2C_setCursor(6,1);
360    LCDI2C_write_String(mode);
361    LCDI2C_setCursor(13,2);
362    LCDI2C_write_String("        ");
363    LCDI2C_setCursor(13,2);
364    LCDI2C_write_String(str);
365    LCDI2C_setCursor(18,2);
366    LCDI2C_write(0xDF);
367    LCDI2C_setCursor(19,2);
368    LCDI2C_write_String("C");
369  }
```

Figure XLVII. displayFunction for Displaying Data in Task 2.

Task 2 shall be as the following example:

```
172  void vTask2( void *pvParameters )
173  {
174  const char                *pcTaskName = "Task 2 is running\r\n";
175  volatile unsigned long    ul;
176  char                      str_buf[7];
177  const                     TickType_t xDelay = 350 / portTICK_PERIOD_MS;
178  char                      *mode[] = {"Auto Mode", "User Mode"};
179    /* As per most tasks, this task is implemented in an infinite loop. */
180    for( ;; )
181    {
182      if(reading_Temp != 0 && xSemaphoreTake( xSemaphore, ( TickType_t ) portMAX_DELAY )==pdTRUE)
183      {
184        #ifdef DEBUG_BY_TICK
185          printf("\r\rTich Count Task 2: %d\n", xTaskGetTickCount());
186        #endif
187
188        printf( "%s\n",pcTaskName );
189        sprintf(str_buf, "%0.2f", reading_Temp);
190        if(user_Mode != 0)                //Enter User_Mode
191        {
192          displayFunction(str_buf, mode[1]);
193        }
194        else
195        {
196          displayFunction(str_buf, mode[0]);
197        }
198        #ifdef DEBUG_BY_TICK
199          printf("\rTich Count END Task 2: %d\n", xTaskGetTickCount());
200        #endif
201
202        xSemaphoreGive( xSemaphore);
203        vTaskDelay( xDelay );
204      }
205    }
206  }
```

Figure XLVIII. Task 2 - Displaying Task.

4.3.9.2.3 Task 3 – Driving Fan in Auto-mode

In Task 3, the system shall check the system mode (user_Mode variable) to be able to identify which mode the system is in.

In the Auto-mode (user_Mode is 0), the system shall check the existent of Task 4 by checking *TaskHandle_t* of Task 4 (xTask4). If Task 4 is exited, the system shall delete it to save the CPU

time for other purposes. Otherwise, the system shall use auto_Fan function to drive fan (motor) according to reading_Temp value.

In the User-mode (user_Mode is not 0), the system shall create Task 4 to drive the fan (motor) according to the input voltage from potentiometer.

The auto_Fan function shall be as the following example:

```
329  void auto_Fan(const volatile float *temperature)
330  {
331    uint16_t speed_Value;
332    if(*temperature > 20 && *temperature <= 24)
333    {
334      speed_Value = 75;                                  //    TIM5->CCR1 = 75 * 65535 / 100;  // 75% Duty cycle
335      driving_Fan(&speed_Value);
336    }
337    else if(*temperature > 24 && *temperature <= 27)
338    {
339      speed_Value = 40;                                  //    TIM5->CCR1 = 40 * 65535 / 100;  // 40% Duty cycle
340      driving_Fan(&speed_Value);
341    }
342    else if(*temperature > 27)
343    {
344      speed_Value = 10;                                  //    TIM5->CCR1 = 10 * 65535 / 100;  // 10% Duty cycle
345      driving_Fan(&speed_Value);
346    }
347  }
```

Figure XLIX. auto_Fan Function for Driving Fan in Task 3.

Task 3 shall be as the following example:

```
220   void vTask3( void *pvParameters )
221  {
222  const char *pcTaskName = "Task 3 is running\r\n";
223  volatile unsigned long ul;
224  const TickType_t xDelay = 150 / portTICK_PERIOD_MS;
225
226    /* As per most tasks, this task is implemented in an infinite loop. */
227    for( ;; )
228    {
229      if(xSemaphoreTake( xSemaphore, ( TickType_t ) portMAX_DELAY )==pdTRUE)
230      {
231        #ifdef DEBUG_BY_TICK
232          printf("\r\rTich Count Task 3: %d\n", xTaskGetTickCount());
233        #endif
234
235        printf( "%s\n",pcTaskName );
236  //      checking_Button();
237
238        /*Checking if *flag* from the interrupt function is SET.
239          It would delay from amount of time because of the unstable of the button.
240          Then reset the flag for the next time.
241          If the flag is SET -> switching modes (Auto_Mode and User_Mode)
242          Uncomment to used the below codes. This was the early version of switing mode \
243          by seting/reseting flag variable*/
244        /*
245        if(flag == SET)
246        {
247          vTaskDelay(150/portTICK_PERIOD_MS);
248          flag = RESET;
249          user_Mode = ~user_Mode;
250        }
251        */
252
253        /*Checking if *xButton_Semaphore* is Giving by external interrupt button.
254          It would delay from amount of time because of the unstable of the button.
255          Then switching modes (Auto_Mode and User_Mode)*/
256        if(xSemaphoreTake( xButton_Semaphore, ( TickType_t ) 0 ))
257        {
258          user_Mode = ~user_Mode;
259        }
```

```
259         }
260
261         //Enter the user_Mode or Auto_mode by checking the user_Mode variable
262         //The user_Mode would driving the FAN speed manually by changing the potentiometer - from the ADC
263         //The auto_Mode would driving the FAN speed automatically by its algorithms according to \
264         the temperature data from sensor
265         if(user_Mode != 0)                //Enter User_Mode
266         {
267           LED_ON();
268           printf("Enter User Mode\r\n\n");
269           if(xTask4 == NULL)
270           {
271             xTaskCreate( vTask4, "Task 4", 200, NULL, 1, &xTask4 );
272           }
273         }
274         else if(user_Mode == 0)           //Enter Auto_Mode
275         {
276           LED_OFF();
277           if( xTask4 != NULL)
278           {
279             vTaskDelete(xTask4);
280             xTask4 = NULL;
281           }
282           auto_Fan(&reading_Temp);
283           printf( "FAN speed: %d%%\r\n\n",100 - (TIM5->CCR1 *100) /65535);        //Print the FAN's speed
284         }
285
286         #ifdef DEBUG_BY_TICK
287           printf("\rTich Count END Task 3: %d\n", xTaskGetTickCount());
288         #endif
289
290         xSemaphoreGive( xSemaphore);
291         vTaskDelay( xDelay );
292       }
293     }
```

Figure L. Task 3 - Driving Fan in Auto-mode.

4.3.9.2.4 Task 4 – Driving Fan in User-mode

In Task 4, the system shall read the changing value of potentiometer and drive the fan (motor) according the changing value. It also transmits the data through USART to target PC for debugging.

Task 4 shall be as the following example:

```
void vTask4( void *pvParameters )
{
  const char                *pcTaskName         = "Task 4 is running\r\n";
  volatile unsigned long    ul;
  const                     TickType_t xDelay   = 100 / portTICK_PERIOD_MS;
  uint16_t                  reading_ADC;
  uint16_t                  fan_Speed;
  for( ;; )
  {

    if(xSemaphoreTake( xSemaphore, ( TickType_t ) portMAX_DELAY )==pdTRUE)
    {
      #ifdef DEBUG_BY_TICK
        printf("\r\rTich Count Task 4: %d\n", xTaskGetTickCount());
      #endif

      printf( "%s\n",pcTaskName );
      reading_ADC = ADC_Read()  ;
      reading_ADC = (reading_ADC * 100) / 0x0FFF; //Caculating percentage of reading value in 12-bits value
      printf("FAN's speed: ");
      printf("%d%%\r\n\n",reading_ADC);           //Monitor in percentage
      fan_Speed = 100 - reading_ADC;              //Calculating fan_Speed to duty-cycle
      driving_Fan(&fan_Speed);                    //Driving motor - FAN
      #ifdef DEBUG_BY_TICK
        printf("\rTich Count END Task 4: %d\n", xTaskGetTickCount());
      #endif

      xSemaphoreGive( xSemaphore);
      vTaskDelay( xDelay );
    }
  }
}
```

Figure LI. Task 4 - Driving Fan in User-mode.

4.3.9.3 ISR

To respond virtual immediately (real-time) with user interaction, the system shall use the external interrupt button from line EXTI1 (PC_1 GPIO) which shall be configured by NVIC and EXTI of ARM Cortex-M4 to switch the system mode.

Furthermore, to synchronize the interrupt with system, the system shall use the semaphore for blocking user-mode task until it is released by interrupt function.

Besides that, to use NVIC of ARM Cortex-M4 with FreeRTOS, the ***FreeRTOSConfig*** header file must be configured according the instruction on FreeRTOS website [24] and the priority group of NVIC must be configured to be 4 (only preemption priority, not subpriority) as described in the instruction of FreeRTOS for running on ARM Cortex-M4 Core [25].

In addition, to configure the priority group of NIVC correctly, the system must call the NVIC_PriorityGroupConfig function before configuring the priority for any external/internal interrupt from NVIC. Moreover, the priority of interrupt from NVIC must be equal or higher than the definition of maximum system call interrupt in ***FreeRTOSConfig*** header file.

- #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
- Therefore, the interrupt priority of the button shall be 0x05 or higher (lower priority).

The initialization function for external interrupt from button shall be as the following example:

```
4  void BUTTON_INIT()
5  {
6    GPIO_InitTypeDef GPIO_InitStructure1;
7    EXTI_InitTypeDef EXTI_InitStruct;
8    NVIC_InitTypeDef NVIC_InitStruct;
9
10
11   /* Enable the GPIOA peripheral */
12   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
13   RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
14
15   /* Configure MCO1 pin(PC.13) in alternate function */
16   GPIO_InitStructure1.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_1 ;
17   GPIO_InitStructure1.GPIO_Speed = GPIO_High_Speed;
18   GPIO_InitStructure1.GPIO_Mode = GPIO_Mode_IN;
19   GPIO_InitStructure1.GPIO_PuPd = GPIO_PuPd_DOWN;
20   GPIO_Init(GPIOC, &GPIO_InitStructure1);
21
22   /* Tell system that you will use PC1 for EXTI_Line1 */
23   SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOC, EXTI_PinSource1);
24
25   /* PD0 is connected to EXTI_Line1 */
26   EXTI_InitStruct.EXTI_Line = EXTI_Line1;
27   /* Enable interrupt */
28   EXTI_InitStruct.EXTI_LineCmd = ENABLE;
29   /* Interrupt mode */
30   EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
31   /* Triggers on rising and falling edge */
32   EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising;
33   /* Add to EXTI */
34   EXTI_Init(&EXTI_InitStruct);
35
36   //Relevant to this: https://www.freertos.org/RTOS-Cortex-M3-M4.html
37   /* Add IRQ vector to NVIC */
38   /* PD0 is connected to EXTI_Line1, which has EXTI1_IRQn vector */
39   NVIC_InitStruct.NVIC_IRQChannel = EXTI1_IRQn;
40   /* Set priority */
41   NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x05;
42   /* Set sub priority */
43   NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x00;
44   /* Enable interrupt */
45   NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
46   /* Add to NVIC */
47   NVIC_Init(&NVIC_InitStruct);
48 }
```

Figure LII. Initialization Function for Interrupt Button.

The Initialization function for interrupt button is based on the tutorial example on the internet [26].

The external interrupt/event GPIO mapping could be seen in Appendix 4.

The interrupt function shall use xSemaphoreGiveFromISR API function of FreeRTOS to give semaphore (xButton_Semaphore) for switching system mode which had been described in Task 3 section.

The interrupt function shall be as the following example:

```
402  /* Handle PC1 interrupt */
403 void EXTI1_IRQHandler(void) {
404   /* Make sure that interrupt flag is set */
405   if (EXTI_GetITStatus(EXTI_Line1) != RESET) {
406
407     //flag = SET;       //another way for changing system mode
408
409     //Giving Semaphore from ISR to be able to change system mode
410     xSemaphoreGiveFromISR(xButton_Semaphore, NULL);
411     /* Clear interrupt flag */
412     EXTI_ClearITPendingBit(EXTI_Line1);
413   }
414 }
```

Figure LIII. External Interrupt Function.

4.3.9.4 Scheduling Tasks

FreeRTOS schedule tasks with Round-robin algorithm (tasks without priorities) or preemptive algorithm (tasks with priorities).

Besides that, there are several other algorithms for scheduling tasks in RTOS [27]. For example, cyclic code (CC), Rate-monotonic (RMS) and other scheduling algorithms. However, in this project, to meet the real-time requirements, the system must use the Rate-monotonic scheduling algorithm to calculate the schedulabilities of the system. (The measuring execution time table in Appendix 5)

Schedulability test equation for RMS [28]: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$

RMS schedulability calculation:

	Best Case (without interruption)	Worst Case (with interruption)
Without Task 4	$\frac{26}{100} + \frac{34}{300} + \frac{6}{150} \leq 3\left(2^{1/3} - 1\right)$ $=> 0.41 < 0.78$	$\frac{26}{100} + \frac{34}{300} + \frac{8}{150} \leq 3\left(2^{1/3} - 1\right)$ $=> 0.43 < 0.78$
With Task 4	$\frac{26}{100} + \frac{34}{300} + \frac{6}{150} + \frac{6}{100} \leq 4\left(2^{1/4} - 1\right)$ $=> 0.47 < 0.76$	$\frac{26}{100} + \frac{34}{300} + \frac{9}{150} + (\frac{6}{100}) \leq 4\left(2^{1/4} - 1\right)$ $=> 0.49 < 0.78$

Table 1. Calculation Schedulabilities of System.

According to the calculation schedulabilities of system table, all tasks in the system shall be schedulable and meet their deadlines which is the essential requirement of the real-time system.

4.3.9.5 Protecting Shared Resources

There are many different methods for protecting shared resource, such as using semaphore, disabling interrupt. In this project, the system shall use binary semaphore to protect the shared resources and synchronize tasks.

The reasons for using binary semaphore are about the system shall allow only one task to access the shared resource at a time and shall signal the next ready task when the running task is done.

Besides that, there are possible problems while using binary semaphore in the preemption system. For example, it could be deadlock situation while multiple tasks are trying to take the semaphore. To solve those problems, the system must be designed correctly.

In this project, because of protecting shared resources and using printf function for debugging which shall use the same USART peripheral to transmit data to target PC, every task must require and wait for a binary semaphore until it is available to be able to access the shared resource.

In addition, in this project, to use the binary semaphore, the system shall use API functions of FreeRTOS to create, give and take binary semaphore, for example:

- xSemaphoreCreateBinary [29].
- xSemaphoreGive [30].
- xSemaphoreTake [31].

The semaphore could be created, acquired and released as the following example:

```
82 SemaphoreHandle_t xSemaphore = NULL;
83 SemaphoreHandle_t xButton_Semaphore = NULL;
```

Figure LIV. Defining xSemaphore in FreeRTOS.

```
 99   xSemaphore = xSemaphoreCreateBinary();
100   xButton_Semaphore = xSemaphoreCreateBinary();
```

Figure LV. Creating Binary Semaphore in FreeRTOS.

```
147     if(xSemaphoreTake( xSemaphore, ( TickType_t ) portMAX_DELAY ) == pdTRUE)
```

Figure LVI. Requiring Semaphore in FreeRTOS.

```
161         xSemaphoreGive( xSemaphore);
```

Figure LVII. Releasing Semaphore in FreeRTOS.

4.3.9.6 Analysing – Debugging Real-Time System

The FreeRTOS software has different utilities, macros, and tools for analysing the real-time system and tracing tasks.

4.3.9.6.1 Trace Hook Macro

One of the most power full features for debugging and analysing RTOS in FreeRTOS is trace hook macros. Those macros have been defined as empty functions, therefore they shall not consume memory, timing, or impact to the system until they are redefined to be used [32].

Besides that, they are very easy to be implemented. The system shall define the macro in ***FreeRTOSConfig*** header file to 1 or redefine the macro to new function. For example:

- Defining STACK_OVERFLOW macro to be 1 to be able to see whether the system is crashed because of stack overflow problem.

```
84  #define configCHECK_FOR_STACK_OVERFLOW  1
```

Figure LVIII. Defining STACK_OVERFLOW Macro in FreeRTOSConfig Header File

```
483  void vApplicationStackOverflowHook(void)
484  {
485  }
```

Figure LIX. STACK_OVERFLOW Hook Function.

- Defining SWITCHED_IN/OUT macro to be able trace the system when it switches in or out task.

```
163  #define traceTASK_SWITCHED_IN() taskSwitchIn_Debugger(pxCurrentTCB->pcTaskName )
```

Figure LX. Defining SWITCHED_IN Hook Function

```
418  void taskSwitchIn_Debugger(char *taskName)
419  {
420    LED_ON();
421  }
```

Figure LXI. SWITCHED_IN Hook Function.

4.3.9.6.2 Tracealyzer

Besides hook macros of FreeRTOS, FreeRTOS also has a trace tool which is so-called Tracealyzer software from Percepio. This analysis tool is very useful, powerful for analysing the real-time system which is developed with FreeRTOS [33].

In this project, the system shall use the Tracealyzer also for analysing and tracing tasks. To use the Tracealyzer software, the system must follow the instruction on Percepio website [20]. Basically, they shall be:

- Fetching Tracealyzer software.
- Including code files and header files according to recording mode the system shall use.
- Defining TRACE_FACILITY macro in ***FreeRTOSConfig*** header file to be 1.
- Setting and modifying parameters in included files according to the system characteristics.

In this project, the system shall use the ST-Link debugger for debugging, therefore the system shall not be able to use the Streaming Mode of Tracealyzer software which require J-Link debugger. Thus, the system shall use Snapshot Mode for analysing.

4.3.9.6.2.1 Snapshot Mode

To use Tracealyzer in Snapshot Mode, the project must include corresponding files:

- Including ***trcKernelPort.c*** and ***trcSnapshotRecorder.c*** source files in "TraceRecorder" directory (folder) and their header files in "./include" directory (folder).
- Including ***trcConfig.h*** and ***trcSnapshotConfig.h*** header files in "./config" directory (folder).

To use Snapshot Mode of Tracealyzer correctly, the system must configure and define the following files:

- In ***trcConfig*** header file, the system must define the corresponding HARDWARE_PORT, RECODER_MODE and FREERTOS_VERSION macros. In this system, those macros have been defined as the following examples:

```
85 | #define TRC_CFG_HARDWARE_PORT TRC_HARDWARE_PORT_ARM_Cortex_M
```

Figure LXII. Defining HARDWARE_PORT in trcConfig Header File

```
101 | #define TRC_CFG_RECORDER_MODE TRC_RECORDER_MODE_SNAPSHOT
```

Figure LXIII.Defining RECODER_MODE in trcConfig Header File

```
118 | #define TRC_CFG_FREERTOS_VERSION TRC_FREERTOS_VERSION_10_0_0
```

Figure LXIV. Defining FREERTOS_VERSION in trcConfig Header File

In Snapshot Mode, the system shall have two different snapshot modes:

- RING_BUFFER: the old events and data shall be overwritten by the new data, events while recording [35].
- STOP_WHEN_FULL: recording system in particular time until the buffer is full [34].
- The recording mode of Snapshot mode shall be defined in ***trcSnapshotConfig*** header file as the following example:

```
70 | #define TRC_CFG_SNAPSHOT_MODE TRC_SNAPSHOT_MODE_RING_BUFFER
```

Figure LXV. Defining Recording Mode of Snapshot Mode in trcSnapshotConfig Header File

- Besides that, the buffer size for RING_BUFFER mode also shall be defined in ***trcSnapshotConfig*** header file.

To start recording data, the system must call the vTraceEnable(TRC_START) function. Then, in the debugging mode, the system shall be able to save the recorder data by the following instruction steps on Percepio website[34]. In this project, because of using Keil MDK/µVision IDE for developing, the system shall save the recorder data by following steps:

- Including vTraceEnable function before FreeRTOS scheduling tasks.
- Running the system for while.
- Stopping the system.
- In the Debugging Session, the system could save the recorder data in two different methods:
  - Enter the command which has been described on Percepio website [34]:

    ```
    exec("SAVE \"out.hex\" RecorderDataPtr , (RecorderDataPtr + 1)");
    ```

    Figure LXVI. Saving Recorder Data Command from Percepio Website

  - Or following the instruction from the Mastering RTOS tutorial on Udemy [35]:
    - Finding the start address and buffer size of RecorderData in *.map file. It shall be as the following example:

```
2951    RecorderData                    0x2000a2ec   Data        6424  trcsnapshotrecorder.o(.bss)
```

Figure LXVII. Starting Address and Buffer Size of RecorderData in .map File.

- Calculating the end address of RecorderData by converting buffer size to hex value and adding with start address. For example, in this case, the end address shall be 0x2000BC04 (0x2000a2ec + 0x000001918).
- Enter the command:
  - SAVE destination_directory\file_name.hex starting_address, ending_addess

4.3.9.6.2.2 Using Tracealyzer

After opening recording file (file_name.hex) which has been described in the Snapshot Mode section, the Tracealyzer software shall show different figures, charts, graphs to indicate the system work. They shall be as the following examples:

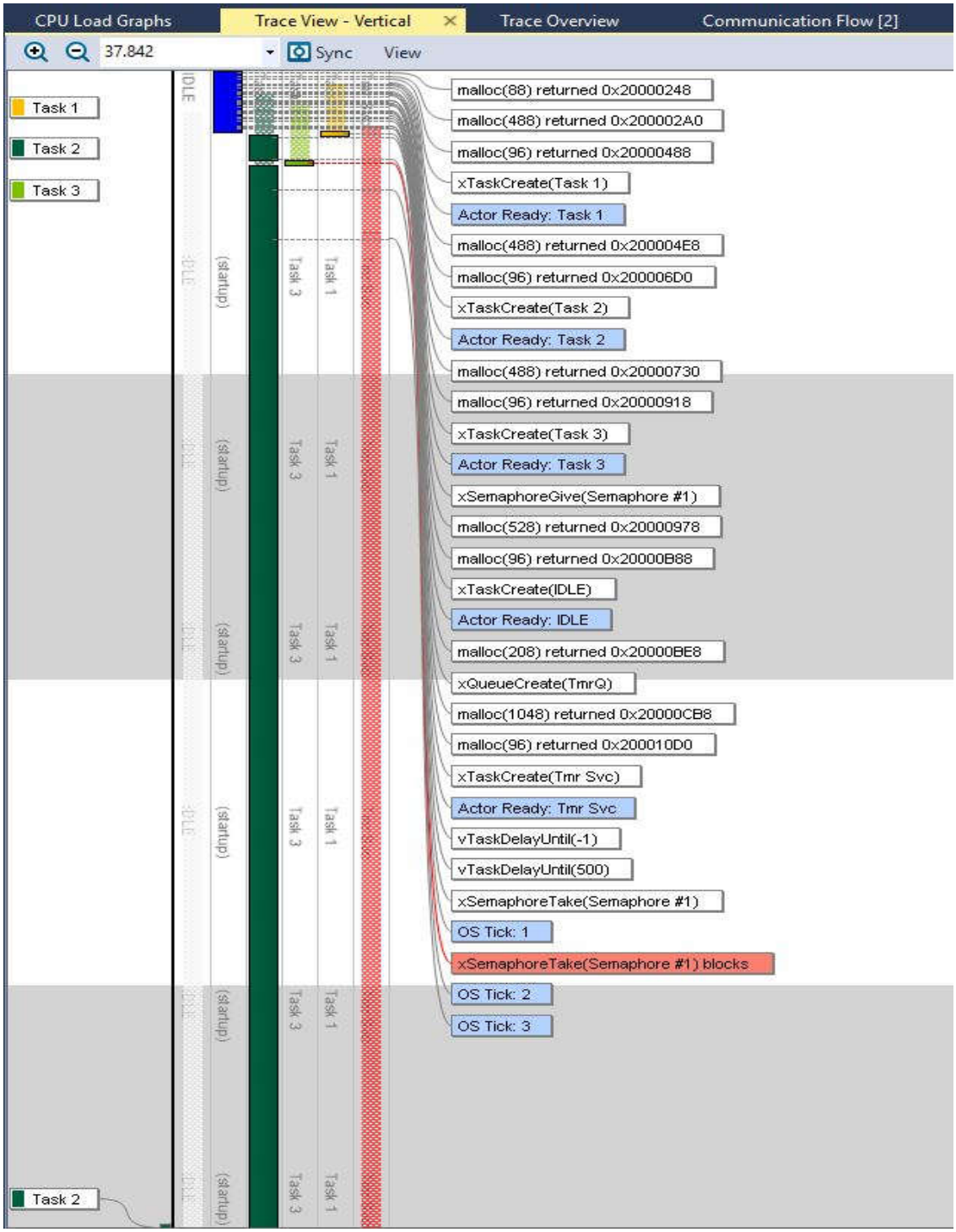


Figure LXVIII. Trace View - System Flow 1.

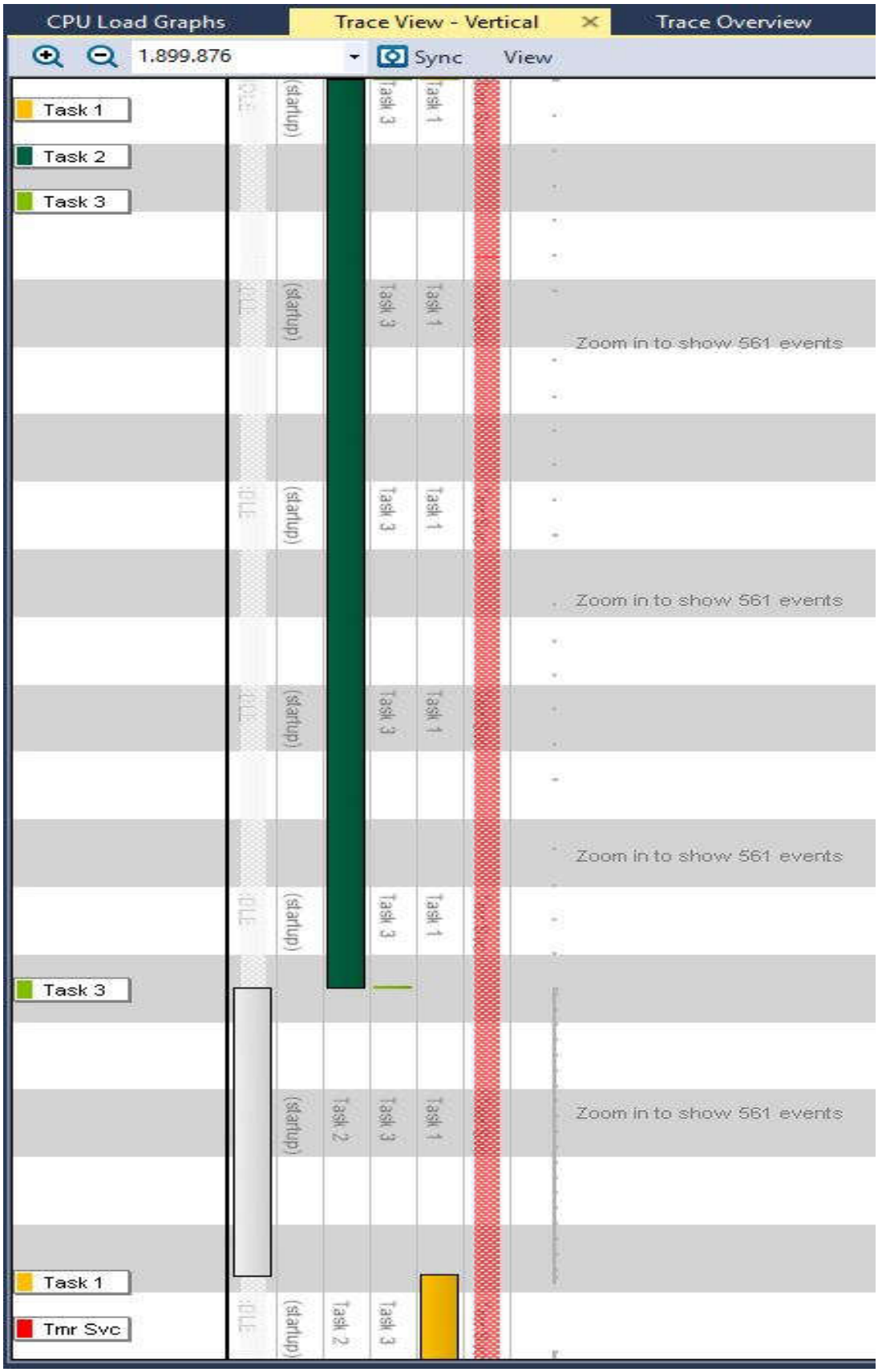


Figure LXIX. Trace View – System Flow 2.

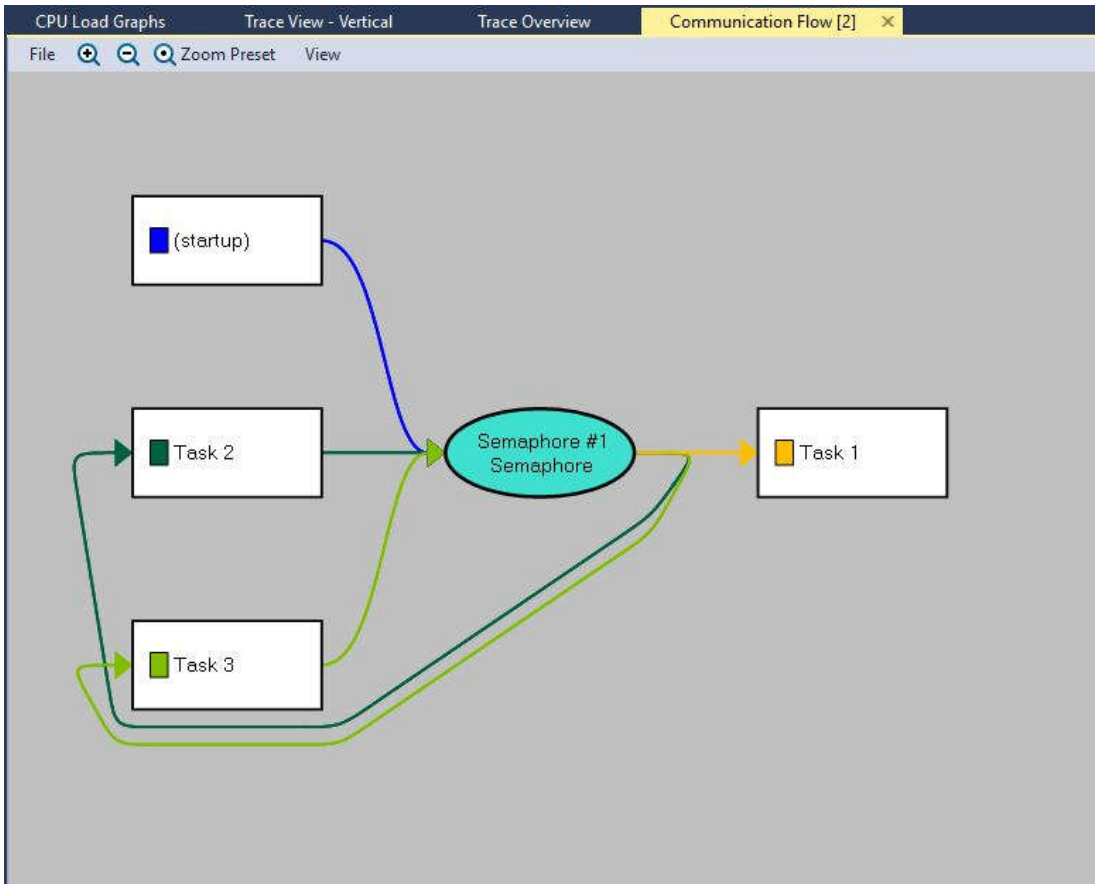


Figure LXX. Communication Flow – Tracealyzer.

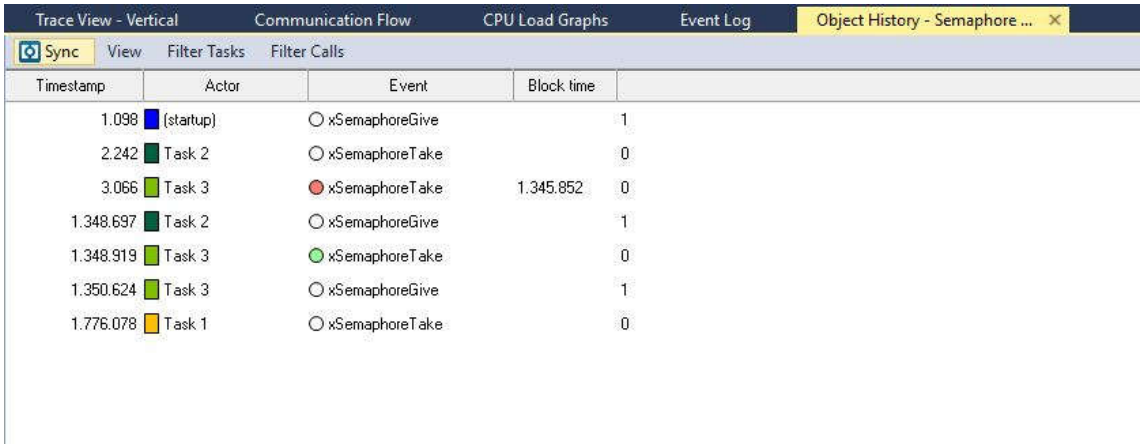


Figure LXXI. Object History - Tracing Semaphore.

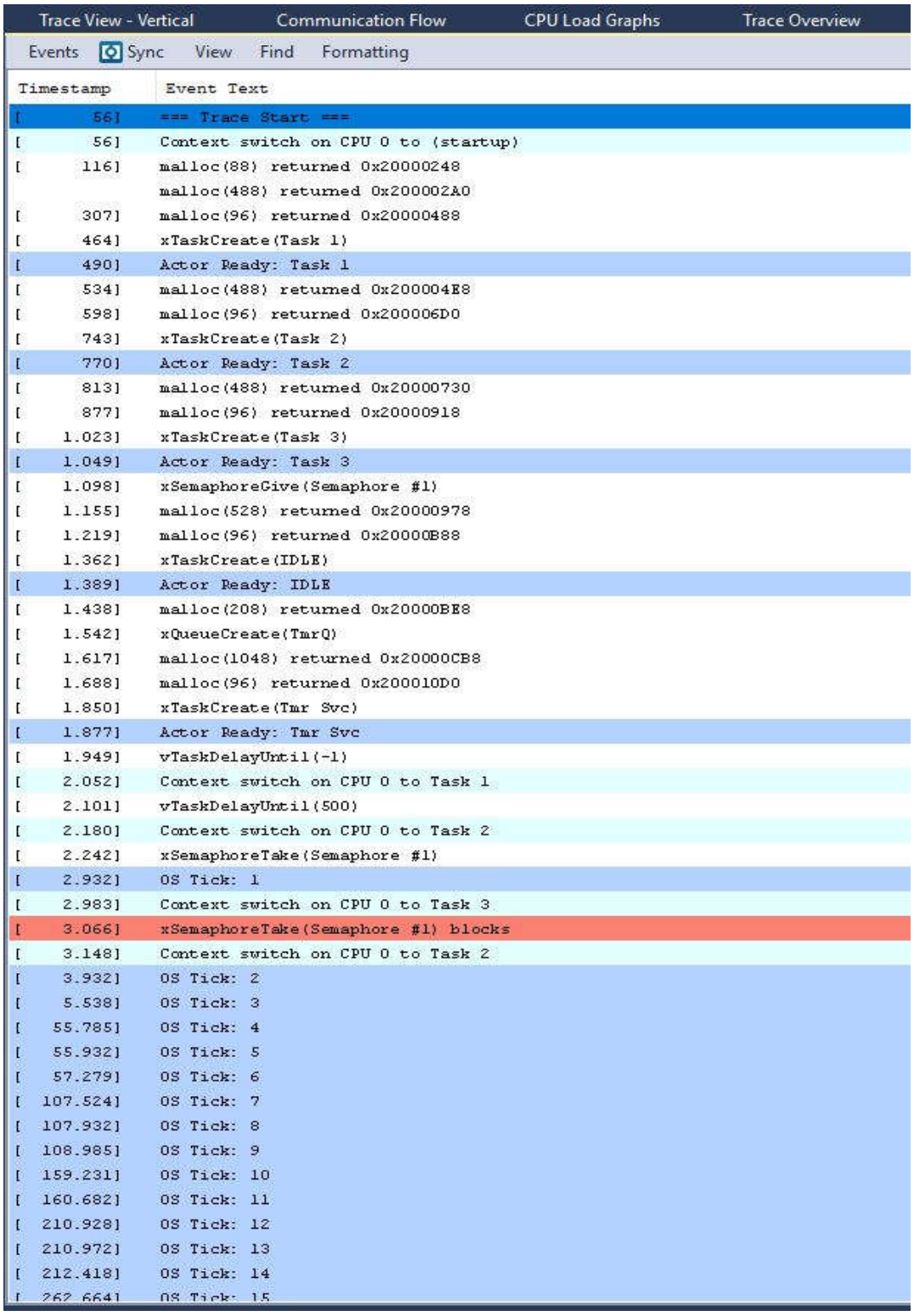


Figure LXXII. Event Logs – Tracealyzer.

4.3.9.6.2.3 Limitation

The Keil MDK/µVision is a free version of ST company, therefore it has its restrictions. One of those restrictions is about limiting the project on the code size (maximum is 32KByte code size) [36]. Thus, the system shall not be able to use Tracealyzer for further analysing especially when the project is exceeded 32Kbyte code size (including the Tracealyzer codes).

5 TESTING AND RESULTS

5.1 Testing

The system has been tested by using different testing methods. For example, using unit tests to test modules before integrating into the system. After all, the system testing method had been applied to ensure that the system shall work properly and meet the requirements.

5.2 Results

The below pictures demonstrate the different modes of the system:

- Auto-mode:

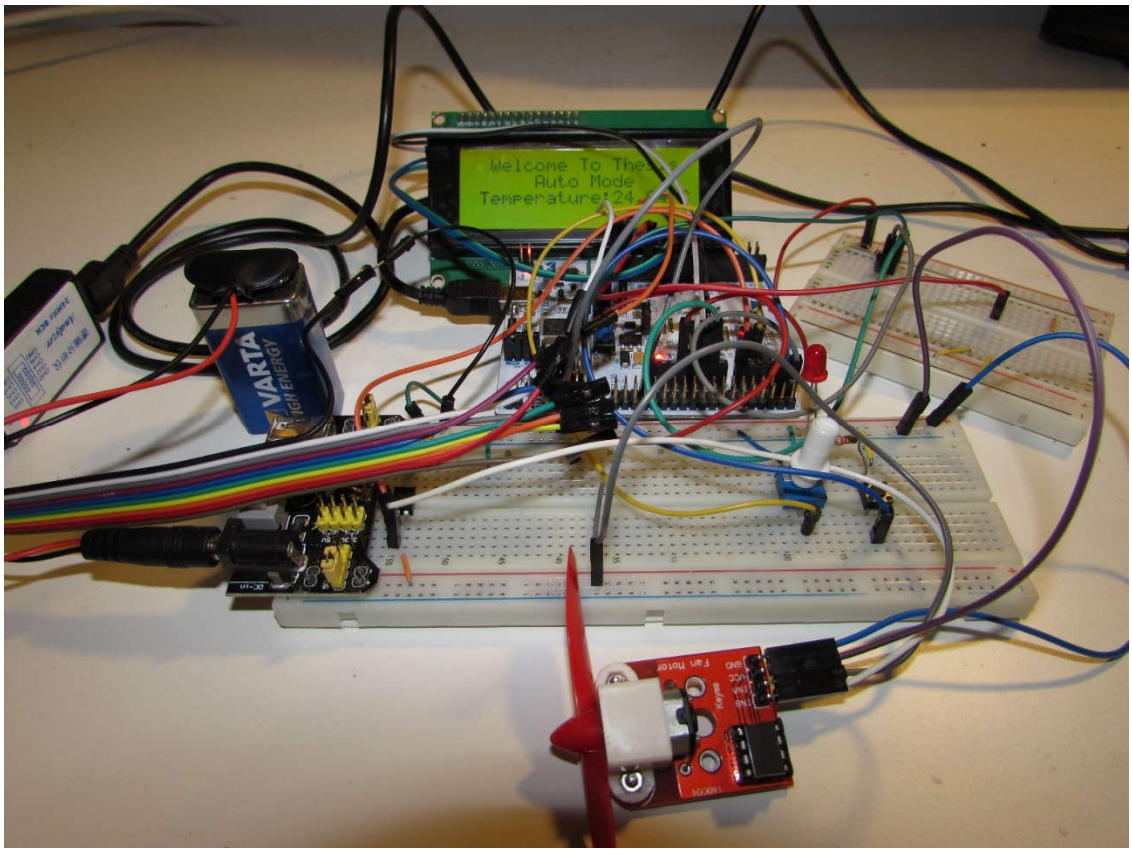


Figure LXXIII. Demonstration of Project in Auto-mode.

- User-mode:

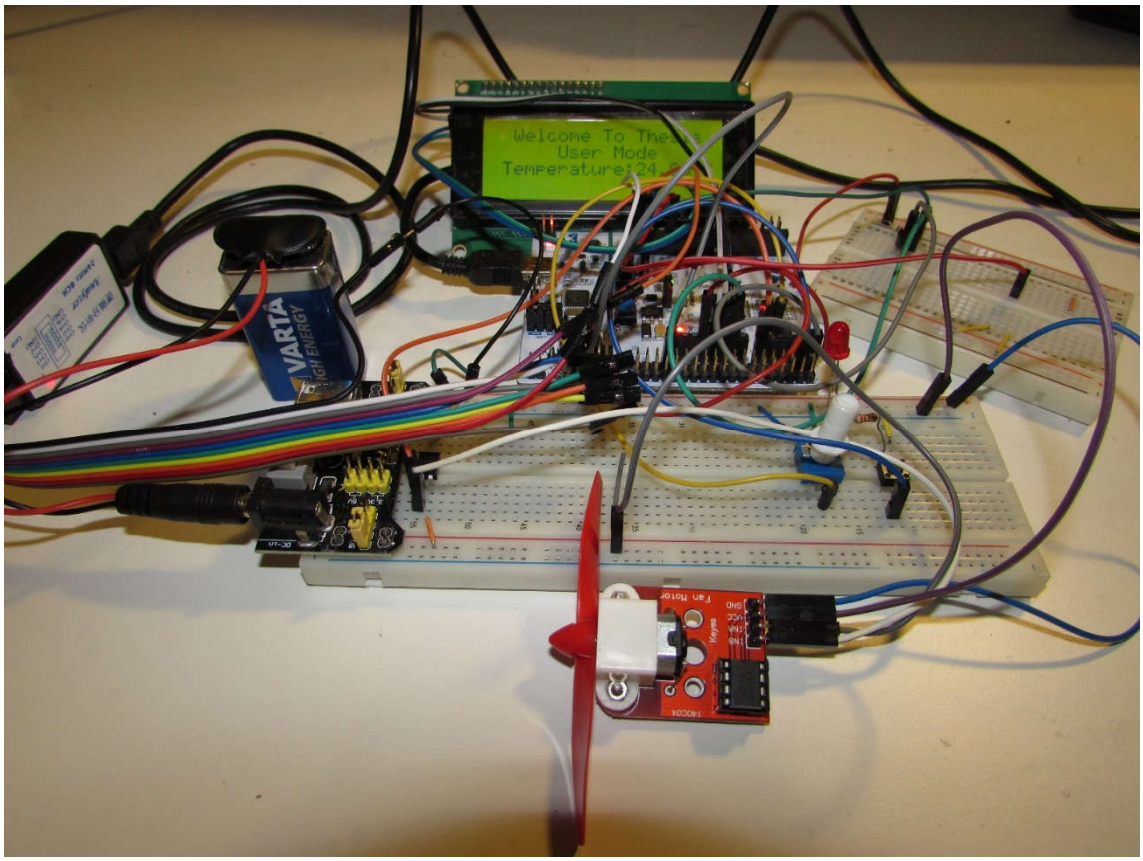


Figure LXXIV. Demonstration of Project in User-mode.

6 CONCLUSION

The purpose of this thesis was to develop an embedded system with the ARM Cortex-M4 MCU, real-time operating system and CAN bus protocol. This embedded system is used to demonstrate the slave system in a building management system.

Generally, this embedded system has been able to provide the comprehensive functions for a tenant room (slave) system, including data acquisition function and driving functions. This system also could be a base system for further developing and analysing in the future. Furthermore, not only is this system developed for a slave system, but also it could be converted to be the master system with other functions to control the large system.

Although the system has been developed successfully, it cannot yet be used for commercial purpose. In addition, the testing results and testing methodologies are beyond the scope of this thesis. Thus, the system might need to be redesigned in a real-time system part, and additional tests need to be conducted ensure that the system meets the real-life requirements.

REFERENCES

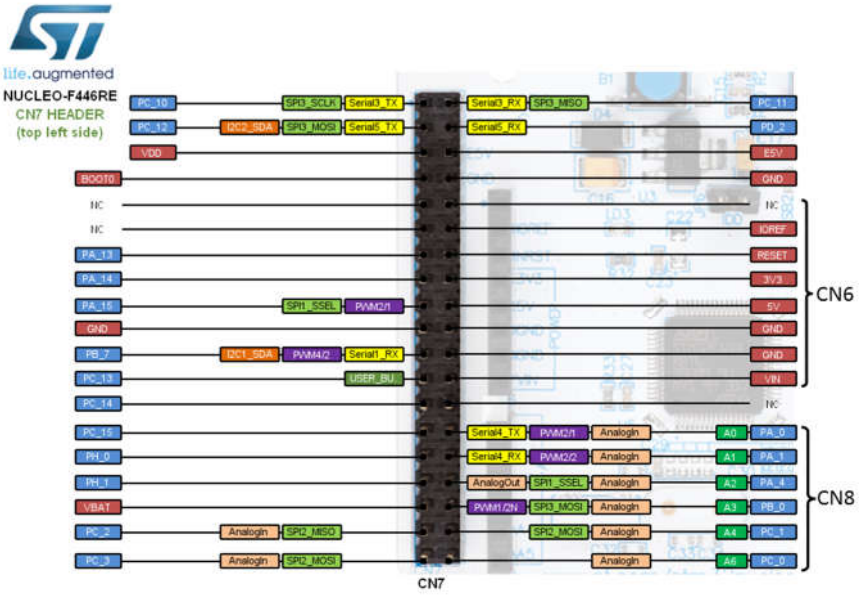
1. Embedded Development Tools [online]. Available at https://www.keil.com/ [Accessed 5th November 2018].
2. µVision IDE [online]. Available at http://www2.keil.com/mdk5/uvision/ [Accessed 5th November 2018].
3. µVision Debugger [online]. Available at http://www2.keil.com/mdk5/debug [Accessed 5th November 2018].
4. MDK Microcontroller Development Kit [online]. Available at http://www2.keil.com/mdk5 [Accessed 5th November 2018].
5. ARM Cortex-M [online]. Available at https://en.wikipedia.org/wiki/ARM_Cortex-M [Accessed 5th November 2018].
6. Floating-point Unit [online]. Available at https://en.wikipedia.org/wiki/Floating-point_unit [Accessed 5th November 2018].
7. STM32F446RE [online]. Available at https://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series/stm32f446/stm32f446re.html [Accessed 6th November 2018].
8. Comparison of Real-Time Operating Systems [online]. Available at https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems [Accessed 7th November 2018].
9. About FreeRTOS [online]. Available at https://www.freertos.org/RTOS.html [Accessed 7th November 2018].
10. Coding Standard and Style Guide [online]. Available at https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html [Accessed 7th November 2018].
11. License Details [online]. Available at https://www.freertos.org/a00114.html [Accessed 7th November 2018].
12. The FreeRTOS Kernel [online]. Available at https://www.freertos.org/index.html [Accessed 7th November 2018].
13. Percepio Tracealyzer [online]. Available at https://percepio.com/tracealyzer/ [Accessed 7th November 2018].
14. CAN bus [online]. Available at https://en.wikipedia.org/wiki/CAN_bus [Accessed 8th November 2018].
15. STM32F446xC/E. Datasheet – Production Data [online]. Available at https://www.st.com/resource/en/datasheet/stm32f446re.pdf Accessed 8th November 2018].
16. RM0390. Reference Manual. STM32F446xx Advanced Arm-based 32-bit MCUs [online]. Available at https://www.st.com/content/ccc/resource/technical/document/reference_manual/4d/ed/bc/89/b5/70/40/dc/DM00135183.pdf/files/DM00135183.pdf/jcr:content/translations/en.DM00135183.pdf [Accessed 9th November 2018].
17. DS18B20. Programmable Resolution 1-Wire Digital Thermometer. [online]. Available at https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf [Accessed 10th October 2018].

18. One-wire Demo on the STM32F4 Discovery Board [online]. Available at https://www.seanet.com/~karllunt/onewire_stm32f4.html [Accessed 10th October 2018].
19. I²C [online]. Available at https://en.wikipedia.org/wiki/I%C2%B2C [Accessed 11th October 2018].
20. STM32_LCD_I2C [online]. Available at https://github.com/Vendict/STM32_LCD_I2C [Accessed 11th October 2018].
21. HD44780U (LCD-II) – Dot Matrix Liquid Crystal Display Controller/Driver. HITACHI [online]. Available at http://site.gravitech.us/MicroResearch/Others/LCD-20x4B/HD44780.pdf [Accessed 12th October 2018].
22. STM32F100: Configuring Two PWM Channels, Polarity and Alignment [online]. Available at http://www.rtos.be/2013/11/stm32f100-configuring-two-pwm-channels-polarity-and-alignment/ [Accessed 12th October 2018].
23. AN3116. Application Note. STM32™'s ADC Modes and Their Applications. [online]. Available at https://www.st.com/content/ccc/resource/technical/document/application_note/c4/63/a9/f4/ae/f2/48/5d/CD00258017.pdf/files/CD00258017.pdf/jcr:content/translations/en.CD00258017.pdf [Accessed 10th October 2018].
24. FreeRTOS Quick Start Guide [online]. Available at https://www.freertos.org/FreeRTOS-quick-start-guide.html [Accessed 14th October 2018].
25. Running the RTOS on a ARM Cortex-M Core [online]. Available at https://www.freertos.org/RTOS-Cortex-M3-M4.html [Accessed 14th October 2018].
26. STM32F4 External Interrupts Tutorial [online]. Available at https://stm32f4-discovery.net/2014/08/stm32f4-external-interrupts-tutorial/ [Accessed 14th October 2018].
27. Selecting the Right RTOS Scheduling Algorithms Using System Modelling [online]. Available at https://www.embedded.com/design/programming-languages-and-tools/4420160/Selecting-the-right-RTOS-scheduling-algorithms-using-system-modelling [Accessed 15th October 2018].
28. Rate-monotonic Scheduling [online]. Available at https://en.wikipedia.org/wiki/Rate-monotonic_scheduling [Accessed 15th October 2018].
29. xSemaphoreCreateBinary [online]. Available at https://www.freertos.org/xSemaphoreCreateBinary.html [Accessed 15th October 2018].
30. xSemaphoreGive [online]. Available at https://www.freertos.org/a00123.html [Accessed 15th October 2018].
31. xSemaphoreTake [online]. Available at https://www.freertos.org/a00122.html [Accessed 15th October 2018].
32. Trace Hook Macros [online]. Available at https://www.freertos.org/rtos-trace-macros.html [Accessed 15th October 2018].
33. FreeRTOS+Trace [online]. Available at https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_Trace/FreeRTOS_Plus_Trace.shtml [Accessed 15th October 2018].
34. Trace Recorder Library [online]. Available at https://percepio.com/docs/FreeRTOS/manual/Recorder.html#Trace_Recorder_Library_Integrating_the_Recorder [Accessed 15th October 2018].
35. Mastering RTOS: Hands on with FreeRTOS, Arduino and STM32Fx [online]. Available at https://www.udemy.com/mastering-rtos-hands-on-with-freertos-arduino-and-stm32fx [Accessed 16th October 2018].

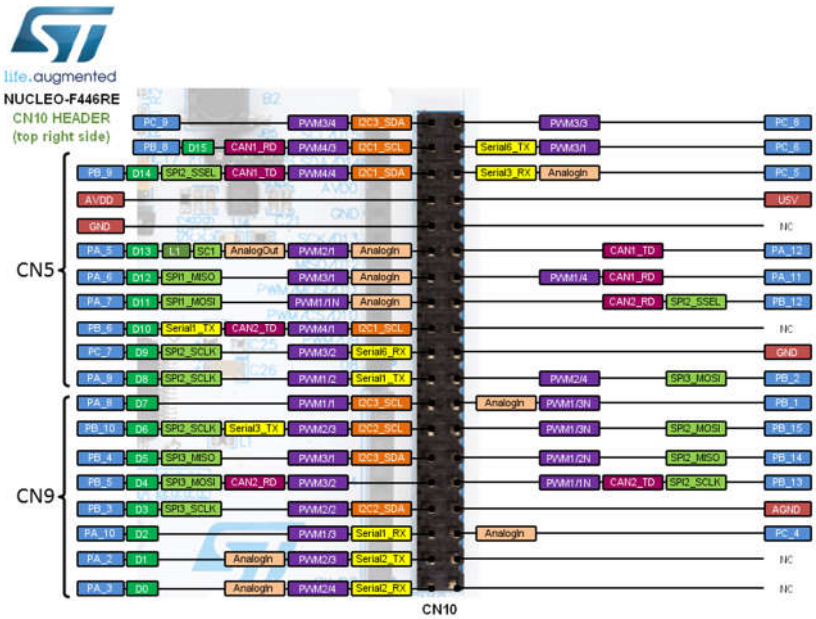
36. MDK-Lite Edition [online]. Available at http://www2.keil.com/mdk5/editions/lite [Accessed 16th October 2018].

Appendix

1. Nucleo pins



Appendix 1. Nucleo STM32F446RE Pins - Left (https://os.mbed.com/platforms/ST-Nucleo-F446RE/).



Appendix 2. Nucleo STM32F446RE Pins - Right (https://os.mbed.com/platforms/ST-Nucleo-F446RE/).

2. Example FreeRTOSConfig.h file for this project on STM32F446RE MCU.

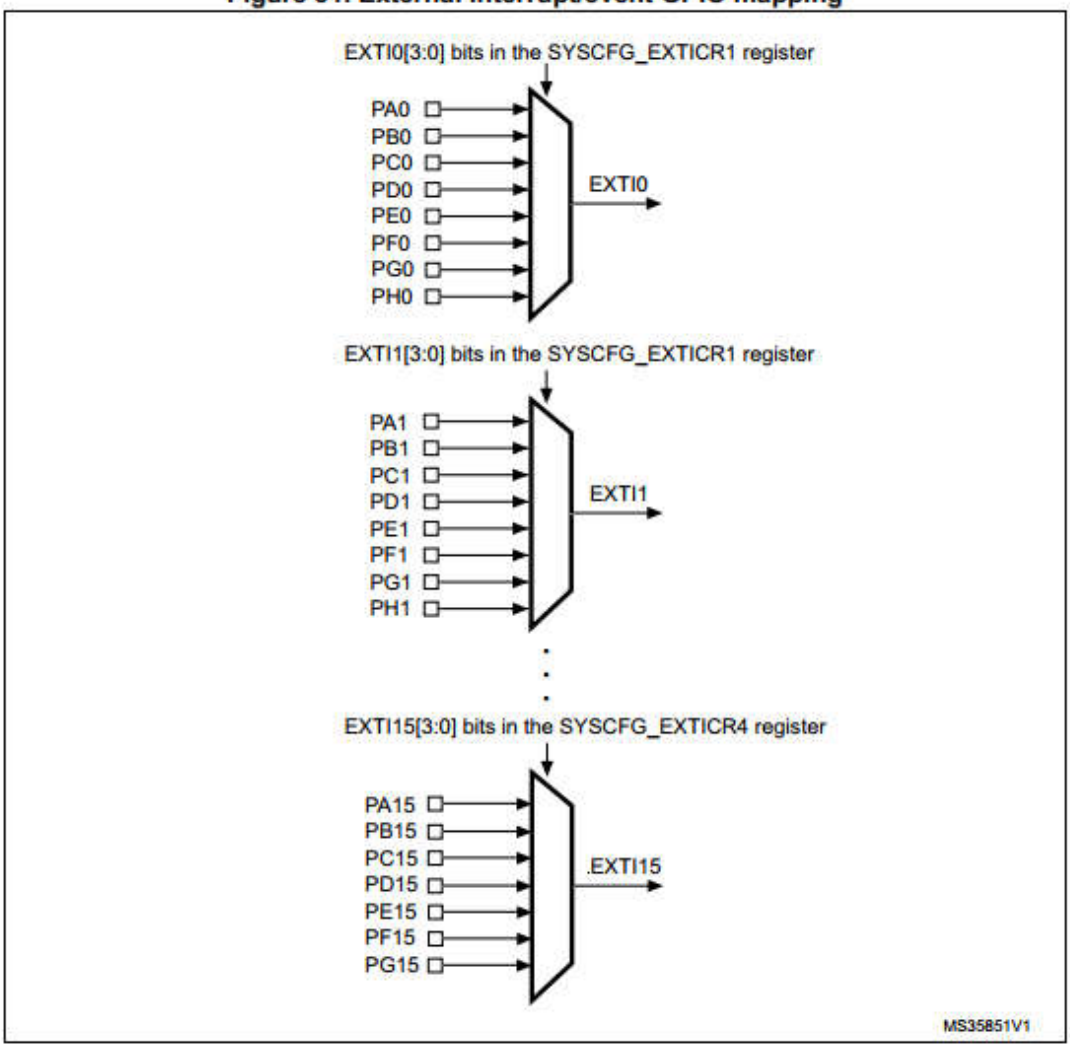
```
#if defined (__GNUC__) || defined (__ICCARM__)
/* Important: put #includes here unless they are also meant for the assembler.
 */
#include <stdint.h>
#endif

extern uint32_t SystemCoreClock;

#define configUSE_PREEMPTION      1
#define configUSE_IDLE_HOOK       0
#define configUSE_TICK_HOOK       0
#define configCPU_CLOCK_HZ        ( SystemCoreClock )         //16MHz
#define configTICK_RATE_HZ        ( ( portTickType ) 1000)
#define configMAX_PRIORITIES      (  15 )
#define configMINIMAL_STACK_SIZE  130
#define configTOTAL_HEAP_SIZE     ( ( size_t ) ( 100 * 1024 ) )
#define configMAX_TASK_NAME_LEN     ( 10 )
#define configUSE_TRACE_FACILITY    0
#define configUSE_16_BIT_TICKS      0
#define configIDLE_SHOULD_YIELD     1
#define configUSE_MUTEXES       1
#define configQUEUE_REGISTRY_SIZE   8
#define configCHECK_FOR_STACK_OVERFLOW  1
#define configUSE_RECURSIVE_MUTEXES   1
#define configUSE_MALLOC_FAILED_HOOK  1
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES 1
#define configSUPPORT_DYNAMIC_ALLOCATION 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES       0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS        1
#define configTIMER_TASK_PRIORITY   ( configMAX_PRIORITIES - 1 )
#define configTIMER_QUEUE_LENGTH    10
#define configTIMER_TASK_STACK_DEPTH  ( configMINIMAL_STACK_SIZE * 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet    1
#define INCLUDE_uxTaskPriorityGet   1
#define INCLUDE_vTaskDelete       1
#define INCLUDE_vTaskCleanUpResources 1
#define INCLUDE_vTaskSuspend      1
```

```
#define INCLUDE_vTaskDelayUntil     1
#define INCLUDE_vTaskDelay          1

/* FreeRTOS+CLI definitions. */

/* Dimensions a buffer into which command outputs can be written.  The buffer
can be declared in the CLI code itself, to allow multiple command consoles to
share the same buffer.  For example, an application may allow access to the
command interpreter by UART and by Ethernet.  Sharing a buffer is done purely
to save RAM.  Note, however, that the command console itself is not re-entrant,
so only one command interpreter interface can be used at any one time.  For
that reason, no attempt at providing mutual exclusion to the buffer is
attempted. */
#define configCOMMAND_INT_MAX_OUTPUT_SIZE 400


/* Cortex-M specific definitions. */

#ifdef __NVIC_PRIO_BITS
  /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
  #define configPRIO_BITS           __NVIC_PRIO_BITS
#else
  #define configPRIO_BITS           4        /* 15 priority levels */
#endif

/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY     0x0f

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY  5

/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY     ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define configMAX_SYSCALL_INTERRUPT_PRIORITY  ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )

/* Normal assert() semantics without relying on the provision of an assert.h
header file. */
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ) __asm volatile( "NOP" ); }
#define INCLUDE_MODULE_TEST 0

#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler

/* Integrates the Tracealyzer recorder with FreeRTOS */
#if ( configUSE_TRACE_FACILITY == 1 )
#include "trcRecorder.h"
#endif


//#define traceTASK_SWITCHED_IN() taskSwitchIn_Debugger(pxCurrentTCB->pcTaskName )
//#define traceTASK_SWITCHED_OUT() taskSwitchOut_Debugger(pxCurrentTCB->pcTaskName )
#endif /* FREERTOS_CONFIG_H */
```

Appendix 3. FreeRTOS Header File.

3. External interrupt/event GPIO mapping (p.245 STM32F446RE Reference Manual).

Figure 31. External interrupt/event GPIO mapping



Appendix 4. External Interrupt/Event GPIO Mapping - STM32F446RE Reference Manual.

4. Measuring execution time table

Time unit = System Tick = 1000 Hz = 1ms

System												
T1			T2			T3			T4			
Start	End	Exec	Start	End	Exec	Start	End	Exec	Start	End	Exec	
1000	1026	26										
1100	1126	26										
						1143	1148	5				
1200	1226	26										
						1300	1305	5				
1307	1334	27										
			1336	1370	34							
1400	1426	26										
						1457	1462	5				
1500	1526	26										
1600	1626	26										
						1628	1634	6				
1700	1726	26										
			1728	1762	34							
						1786	1794	8				Including interruption
									1797	1802	5	
1805	1831	26										
1900	1926	26										
									1928	1934	6	User_Mode
						1946	1951	5				
2000	2026	26										
									2036	2041	5	
2100	2126	26										
						2128	2137	9				Including interruption

Appendix 5. Measuring Execution Time Table.